Concurrent Patterns and Best Practices

Build scalable apps with patterns in multithreading, synchronization, and functional programming

Packt

www.packt.com

Atul S. Khot

Concurrent Patterns and Best Practices

Build scalable apps with patterns in multithreading, synchronization, and functional programming

Atul S. Khot

Packt

BIRMINGHAM - MUMBAI

Concurrent Patterns and Best Practices

Commissioning Editor: Richa Tripathi
Acquisition Editor: Denim Pinto
Content Development Editor: Nikhil Borkar
Technical Editor: Divya Vadhyar
Copy Editor: Muktikant Garimella
Project Coordinator: Ulhas Kambali
Proofreader: Safis Editing
Indexer: Rekha Nair
Graphics: Disha Haria
Production Coordinator: Nilesh Mohite

First published: September 2018

Production reference: 1260918

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78862-790-0

`www.packtpub.com`

Contributors

About the authors

Atul S. Khot is a self-taught programmer and has written software programmes in C and C++. Having extensively programmed in Java and dabbled in multiple languages, these days, he is increasingly getting hooked on Scala, Clojure, and Erlang. Atul is a frequent speaker at software conferences and a past Dr. Dobb's product award judge. He was the author of *Scala Functional Programming Patterns* and *Learning Functional Data Structures and Algorithms*, published by Packt Publishing.

About the reviewer

Anubhava Srivastava is a Lead Engineer Architect with more than 22 years of systems engineering and IT architecture experience. He has authored the book Java 9 Regular Expressions published by Packt Publishing. He's an active contributor to Stackoverflow and figures in its top 0.5% overall reputation. As an open source evangelist he actively contributes to various open source development and some popular computer programming Q&A sites like Stack Overflow with a reputation/score of more than 170k.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Thank you for purchasing this book! We live in a concurrent world and concurrent programming is an increasingly valuable skill.

I still remember the Aha! moment when I understood how UNIX shell pipeline works. I fell headlong in love with Linux and the command line and tried out many combination filters (a filter is a program reading from standard input and writes to standard output) connected via pipes. I was amazed by the creativity and power brought about by the command line. I was working with concurrent programs.

Then, there was a change of project and I was thrown headlong into writing code using the multithreaded paradigm. The language was C or C++, which I loved deeply; however, to my surprise I found that it was a herculean effort to maintain a legacy code base, written in C/C++ that was multithreaded. The shared state was managed in a haphazard way and a small mistake could throw us into a debugging nightmare!

Around that time, I came to know about **object oriented (OO)** design patterns and some multithreaded patterns as well. For example, we wanted to expose a big in-memory data structure safely to many threads. I had read about the Readers/Writer lock pattern, which used smart pointers (a C++ idiom) and coded a solution based on it.

Voila! It just worked. The concurrency bugs simply disappeared! Furthermore, the pattern made it very easy to reason about threads. In our case, a writer thread needed infrequent but exclusive access to the shared data structure. The reader threads just used the structure as an immutable entity—and look mom, no locks!

No locks! Well, well, well... this was something new! As the locks disappeared, so did any possibility of deadlocks, races, and starvation! It felt just great!

There was a lesson I learned here! Keep learning about design patterns and try to think about the design problem at hand - in terms of patterns. This also helped me reason better about the code! Finally, I had an inkling of how to tame the concurrency beast!

Design patterns are reusable design solutions for common design problems. The design solutions are design patterns. Your problem domain may vary, that is, the business logic you need to write will be applicable to solving the business problem at hand. However, once you apply the pattern, things just fall in place, more or less!

For example, when we code using the OO paradigm, we use the **Gang Of Four (GOF)** design patterns (`http://wiki.c2.com/?DesignPatternsBook`). This famous book provides us a catalog of design problems and their solutions. The strategy pattern is used by people in various contexts—the pattern though remains the same.

Some years later, I moved to the Java world and used the executor service to structure my code. It was very easy to develop the code and it worked without any major problem for months (There were other issues, but no data races and no nightmarish debugging!).

Subsequently, I moved to the functional world and started writing more and more Scala. This was a new world with immutable data structures as the norm. I learned about a very different paradigm; Scala's `Futures` and Akka `Actors` gave a whole different perspective. I could feel the power these tools give you as a programmer. Once you pass the learning curve (admittedly a little daunting at the beginning), you are equipped to write a lot safer concurrent code which is also a lot easier to reason about.

The book you are reading talks of many concurrent design patterns. It shows the rationale behind these patterns and highlights the design theme.

Who this book is for

You should have done some Java programming and ideally have played with multithreaded Java programs. Ideally you should have some familiarity with the **gang of four (GoF)** design patterns. You should also be comfortable running Java programs via maven.

This book will take you to the next level while showing you the design theme behind many concurrency patterns. This book hopes to help developers who want to learn patterns to build scalable and high performing apps.

What this book covers

`Chapter 1`, *Concurrency - An Introduction*, provides an introduction to concurrent programming. As you will see, concurrency is a world in itself. You will look at Unix processes and the pipes and filters for concurrency pattern. The chapter covers an overview of concurrent programming. You probably know some of this material already.

`Chapter 2`, *A Taste of Some Concurrency Patterns*, covers some essential basic concepts. The essence of the Java Memory Model is introduced. You will also look at race conditions and problems arising out of the shared state model. You will get a taste of the first concurrency pattern—hand-over-hand locking.

`Chapter 3`, *More Threading Patterns*, covers explicitly synchronizing the mutable state and the monitor pattern. You will see how this approach is fraught with problems. We wrap up with a detailed look at the Active Object design pattern.

`Chapter 4`, *Thread Pools*, covers how threads communicate via the producer/consumer pattern. The concept of thread communication is introduced. Then, the Master/Slave design pattern is explained. The chapter wraps up with another look at the map-reduce pattern as a special case of the fork-join pattern.

`Chapter 5`, *Increasing the Concurrency*, talks about the building blocks. Things such as blocking queues, bounded queues, latches, FutureTask, semaphores, and barrier are discussed. We talk about being live and safety. Finally, we describe immutability and how immutable data structures are inherently thread safe.

`Chapter 6`, *Functional Concurrency Patterns*, introduces futures and talks about its monadic nature. We cover the transformation and monad patterns and illustrate how futures compose. We also look at promises.

`Chapter 7`, *Actors Patterns*, introduces the Actor paradigm. We recall the Active Object again and then explain the actor paradigm—especially the no explicit locks nature of things. We discuss patterns such as ask versus tell, the become pattern (and stressing immutability), pipelining, and half sync or half async. We discuss these patterns and illustrate them via example code.

To get the most out of this book

To get the most out of this book, you should have a decent level of Java programming knowledge. It is assumed that you know the basics of Java threading. Here I cover the essential aspects as a refresher. Having worked through some Java threading examples would be definite plus. You should also be comfortable using maven, the Java build tool.

Using an IDE of your choice IntelliJ Idea, Eclipse or Netbeans would be helpful - but is not strictly necessary. To illustrate the functional concurrency patterns - the last two chapters use Scala. The code uses basic Scala constructs. It would be good to go through an introductory Scala tutorial - and you should be good to go.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Concurrent-Patterns-and-Best-Practices. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
html, body, #map {
 height: 100%;
 margin: 0;
 padding: 0
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
$ mkdir css
$ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email customercare@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Concurrency – An Introduction

–What are concurrency and parallelism? Why should we study them? In this chapter, we will look at the many aspects of the concurrent programming world. The chapter starts with a brief overview of parallel programming and why we need it. We cover ground pretty fast here, touching upon the basic concepts.

As two major forces, *huge data size* and *fault tolerance* drive concurrent programming. As we go through this chapter, some of our examples will touch upon some clustered computing patterns, such as *MapReduce*. Application scaling is an extremely important concept for today's developer. We will look at how concurrency helps applications scale. *Horizontal scaling* (`https://stackoverflow.com/questions/11707879/difference-between-scaling-horizontally-and-vertically-for-databases`) is the magic behind today's massively parallel software systems.

Concurrency entails communication between the concurrent entities. We will look at two primary concurrency models: *message passing* and *shared memory.* We will describe the message passing model using a UNIX shell pipeline. We will then describe the *shared memory model* and show how explicit synchronization creates so many problems.

A design pattern is a solution to a design problem *in context.* Knowledge of the catalog of patterns helps us to come up with a good design for specific problems. This book explains the common concurrency design pattern.

We will wrap up the chapter by looking at some alternative ways of achieving concurrency, namely the actor paradigm and software transactional memory.

In this chapter, we will cover the following topics:

- Concurrency
- Message passing model
- Shared memory and shared state model
- Of patterns and paradigms

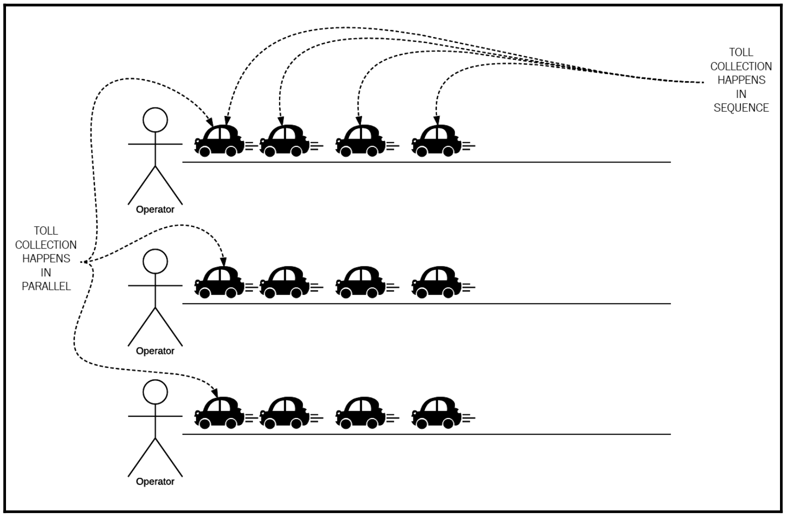
For complete code files you can visit `https://github.com/PacktPublishing/Concurrent-Patterns-and-Best-Practices`

Concurrency in a breeze

We start with a simple definition. When things *happen at the same time*, we say that things are happening *concurrently.* As far as this book is concerned, whenever parts of an executable program run at the same time, we are dealing with concurrent programming. We use the term *parallel programming* as a synonym for concurrent programming.

The world is full of concurrent occurrences. Let's look at a real-life example. Say that there are a certain number of cars driving on a multilane highway. In the *same lane*, though, cars need to follow other cars, the ones that are already ahead of them. A road lane, in this case, is a resource to be shared.

When a toll plaza is built, however, things change. Each car stops in its lane for a minute or two to pay the toll and collect a receipt. While the toll attendant is engaged with the car, other cars behind it need to *queue up* and wait. However, a toll plaza has more than one payment gate. There are attendants at each gate, attending to different cars at the same time. If there are three attendants, each serving one gate, then three cars could pay the toll *at the same point in time;* that is, they can get serviced *in parallel*, as shown in the following diagram:


TOLL COLLECTION HAPPENS IN SEQUENCE
TOLL COLLECTION HAPPENS IN PARALLEL
Operator
Operator
Operator

Note that the cars queuing up at the same booth get serviced in sequence. At any given time, a toll attendant can service only one car, so others in the queue need to wait for their turn.

It would be really odd to see a toll booth with just one gate! People wouldn't be served in parallel. Strictly sequential processing of toll charges would make life unbearable for the frequent traveler.

Even when there are multiple gates and an abnormally large influx of cars (say on vacations), each gate becomes a *bottleneck;* there are far fewer resources for servicing the workload.

The push for concurrency

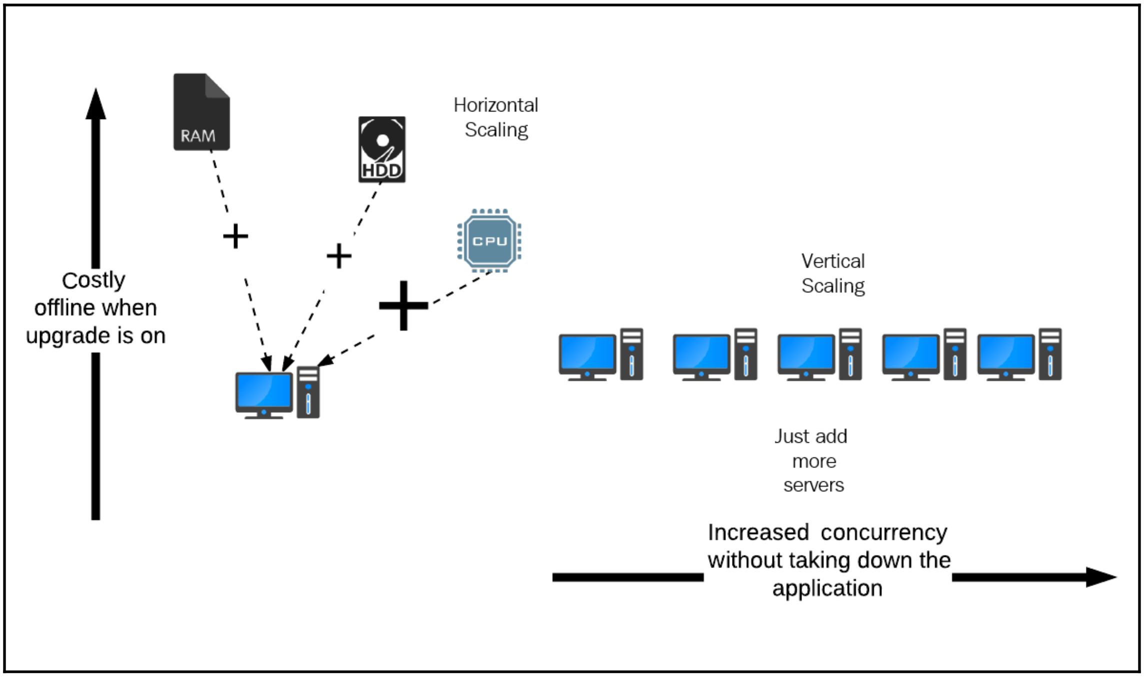
Let's come back to the software world. You want to listen to some music at the same time that you are writing an article. Isn't that a basic need? Oh yes, and your mail program should also be running so that you get important emails in time. It is difficult to imagine life if these programs don't run in parallel.

As time goes by, software is becoming bigger and demand for faster CPUs is ever increasing; for example, database transactions/second are increasing in number. The data processing demand is beyond the capabilities of any single machine. So a *divide and conquer* strategy is applied: many machines work concurrently on different data partitions.

Another problem is that chip manufacturers are hitting limits on how fast they can make chips go! Improving the chip to make the CPU faster has inherent limitations. See `http://www.gotw.ca/publications/concurrency-ddj.htm` for a lucid explanation of this problem.

Today's big data systems are processing trillions of messages per second, and all using *commodity hardware* (that is, the hardware you and me are using for our day-to-day development)—nothing fancy, like a supercomputer.

The rise of the cloud has put provisioning power in the hands of almost everyone. You don't need to spend too much upfront to try out new ideas—just rent the processing infrastructure on the cloud to try out the implementation of the idea. The following diagram shows both scaling approaches:


RAM
HDD
CPU
Horizontal Scaling
Costly offline when upgrade is on
Vertical Scaling
Just add more servers
Increased concurrency without taking down the application

The central infrastructure design themes are *horizontal* versus *vertical* scaling. Horizontal scaling essentially implies the use of a distributed concurrency pattern; it's cost effective, and a prominent idea in the big data world. For example, NoSQL databases (such as Cassandra), analytics processing systems (such as Apache Spark), and message brokers (such as Apache Kafka) all use **horizontal scaling,** and that means distributed and concurrent processing.

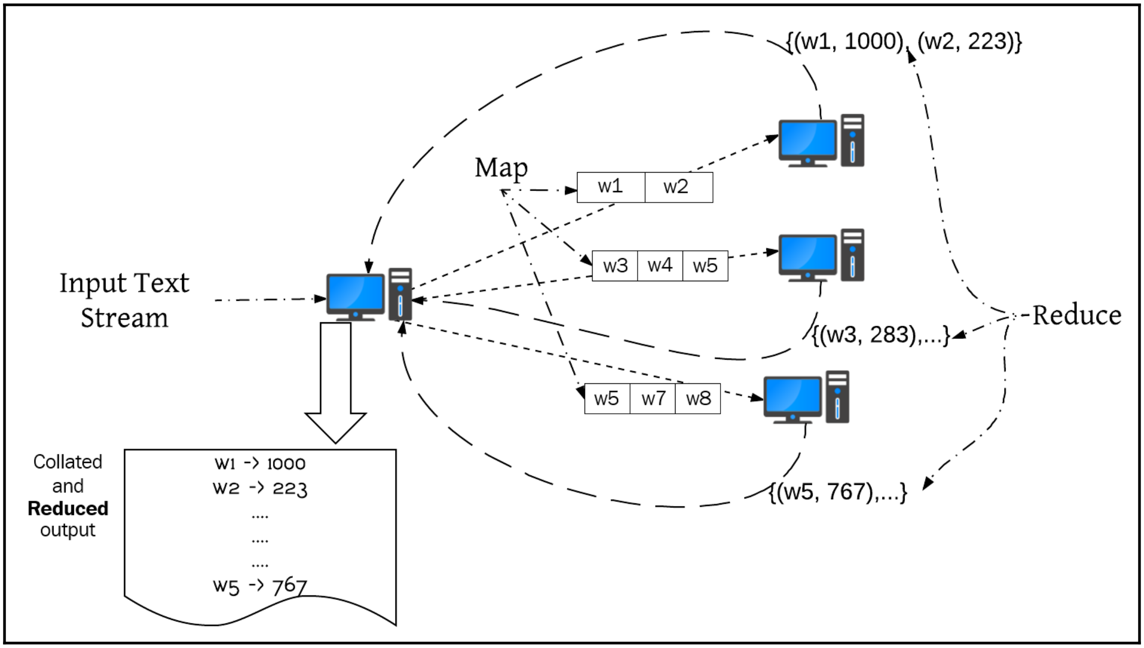
On the other hand, installing more memory or processing power in *a single computer* is a good example of **vertical scaling**. See `https://www.g2techgroup.com/horizontal-vs-vertical-scaling-which-is-right-for-your-app/` for a comparison between the two scaling approaches.

We will look at two common concurrency themes for horizontally scaled systems: M*apReduce and fault tolerance.*

The MapReduce pattern

The *MapReduce* pattern is an example of a common case where concurrency is needed. The following diagram shows a word frequency counter; given a huge text stream of trillions of words, we need to see how many times every word occurs in the text. The algorithm is super simple: we keep the running count of each word in a hash table, with the word as the *key* and the counter as the *value.* The hash table allows us to quickly look up the next word and increment the associated value (counter).

Given the size of the input text, one single node does not have the memory for the entire hash table! Concurrency provides a solution, using the *MapReduce* pattern, as shown in the following diagram:



The solution is *divide and conquer:* we maintain a *distributed hash table* and run the same algorithm, suitably adapted for a cluster.

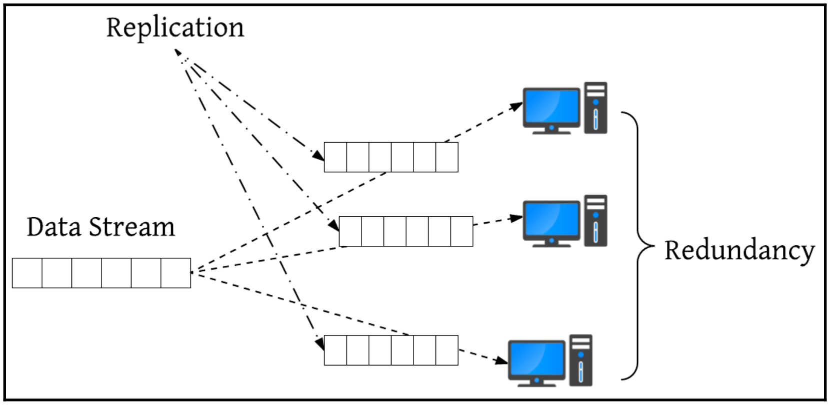
The *master* node reads—parses—the text and pushes it to a set of *slave* processing nodes. The idea is to distribute the text in such a way that one word is processed by one slave node only. For example, given three processing nodes, as shown in the preceding diagram, we would divide *rangewise:* push nodes starting with the characters `{a..j}` to node 1, `{k..r}` to node 2 and the rest—`{s..z}`—onto node 3. This is the *map* part (distribution of work).

Once the stream is exhausted, each slave node *sends its frequency result* back to the master, which prints the result.

The slave nodes are all doing the same processing *concurrently.* Note that the algorithm would run faster if we add, more slave nodes; that is, if we *scale it horizontally*.

Fault tolerance

Another common approach is to build in intentional redundancy to provide *fault tolerance;* for example, big data processing systems, such as Cassandra, Kafka, and ZooKeeper, can't afford to go down completely. The following diagram shows how concurrently replicating the input stream protects against any one slave node going down. This pattern is commonly used by Apache Kafka, Apache Cassandra, and many other systems:



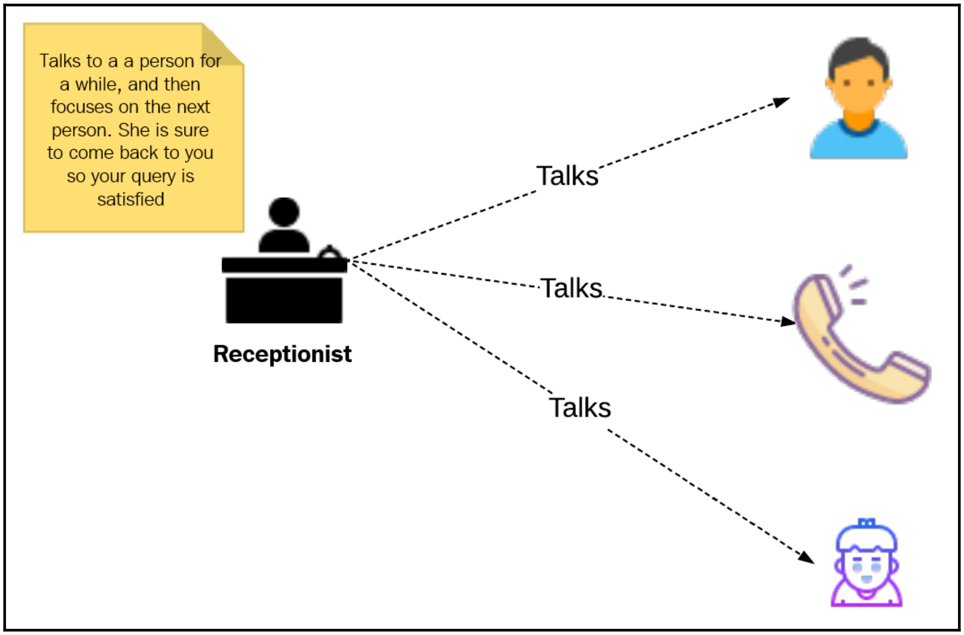
The right side of the diagram shows *redundant* machines on which a data stream is replicated.

In case any one node goes down (hardware failure), other *redundant* nodes take its place, thus ensuring that the system as a whole is never down.

Time sharing

In the real world, we also perform a number of tasks concurrently. We attend to a task and then if another task also needs our attention, we switch to it, attend to it for a while, and then go back to the first task. Let's look at a real-world example of how an office receptionist deals with their tasks.

When you visit any office, there is usually a receptionist who receives you and asks for your business. Say that, just as they are asking about who you need to meet, the office buzzer rings. They take the call, say "hello," speak for a while, and then ask you to wait for a second. Once the call is finished, they resume talking to you. These actions are shown in the following diagram:


Talks to a a person for a while, and then focuses on the next person. She is sure to come back to you so your query is satisfied
Receptionist
Talks
Talks
Talks

The receptionist is sharing her time among all the parties interested in talking to her. She is working in a way so that everyone gets a slice of her time.

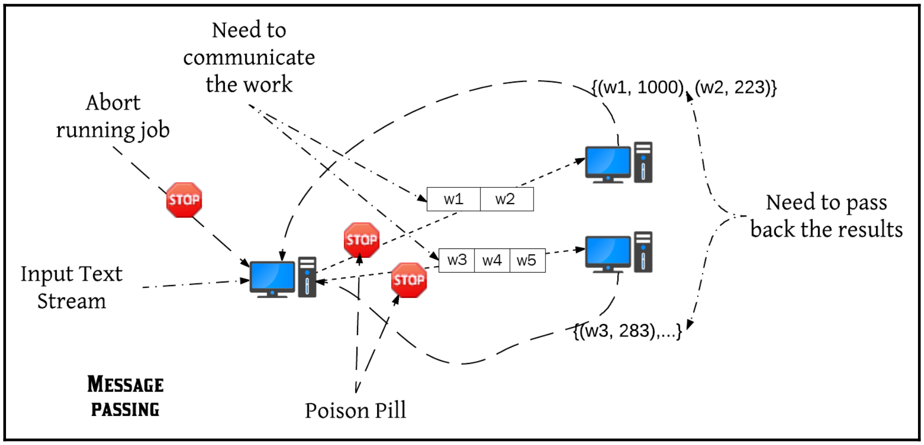
Now, keeping the toll plaza and the receptionist in mind, replace the toll operators with a *CPU core* and the cars with *tasks*, and you get a fairly good mental model of today's concurrent hardware. If we increase the number of toll operators from three to six, we will increase the number of cars getting serviced in parallel (at the exact same time) to six. A pleasant side effect is that the queued cars will also *spread out*, and every car will get serviced faster. The same holds true when we execute a concurrent program. Hence, things are faster overall.

Just as the receptionist is doing multiple things at the same time, such as time sharing between the visitors, a CPU shares time with processes (running programs). This is how concurrency gets supported on a single CPU.

Two models for concurrent programming

Concurrency implies that multiple tasks are happening in parallel to achieve a common goal. Just like communication with a group, we need to communicate and coordinate with the concurrently executing entities.

For example, let us say that we present the previously mentioned word frequency counter via a UI functionality. A user uploads a huge file and clicks the start button, resulting in a long-running MapReduce job. We need to distribute the work among the slaves. To send the workload, we need a way to communicate with them. The following diagram shows the various streams of communications that are required in this system:



If the user changes their mind and aborts the job, we need to communicate the stop message to each concurrent entity, as any further processing is futile.

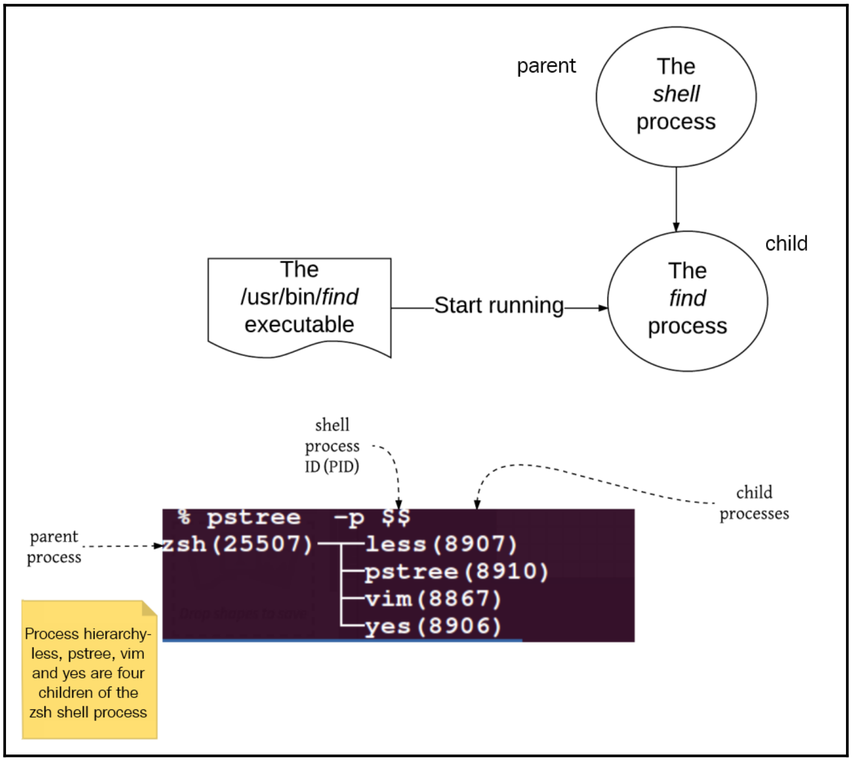
There are two prominent models for concurrent communications: *message passing* and *shared memory*. The preceding diagram shows a message passing model.

We will first discuss the message passing model, using the celebrated UNIX shell pipeline as an example. Next, we will see the shared memory approach in depth and the problems that are associated with it.

The message passing model

Before we dive into the details of the message passing model, we will look at a bit of basic terminology.

When an executable program runs, it is a *process.* The shell looks up the executable, talks to the **operating system (OS)** using system calls, and thereby creates a *child process.* The OS also allocates memory and resources, such as file descriptors. So, for example, when you run the `find` command (the executable lives at `/usr/bin/find`), it becomes a child process whose parent process is the shell, as shown in the following diagram:


parent
The *shell* process
child
The /usr/bin/*find* executable
Start running
The *find* process
shell process ID (PID)
child processes
parent process
Process hierarchy- less, pstree, vim and yes are four children of the zsh shell process

```
% pstree  -p $$
zsh(25507)─┬─less(8907)
           ├─pstree(8910)
           ├─vim(8867)
           └─yes(8906)
```

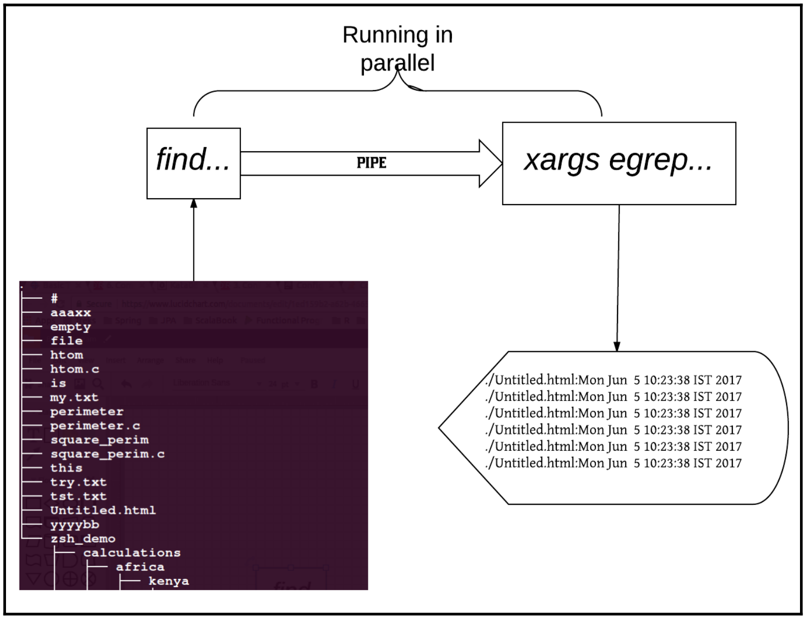
In case you don't have the `pstree` command, you could try the `ptree` command instead. The `ps --forest` command will also work to show you the tree of processes.

Here is a UNIX shell command recursively searching a directory tree for HTML files containing a word:

```
 % find . -type f -name '*.html' | xargs egrep -w Mon /dev/null
./Untitled.html:Mon Jun 5 10:23:38 IST 2017
./Untitled.html:Mon Jun 5 10:23:38 IST 2017
./Untitled.html:Mon Jun 5 10:23:38 IST 2017
./Untitled.html:Mon Jun 5 10:23:38 IST 2017
```

We see a shell pipeline in action here. The `find` command searches the directory tree rooted at the `current` directory. It searches for all files with the `.html` extension and outputs the filenames to standard output. The shell creates a process from the `find` command and another process for the `xargs` command. An active (running) program is called a **process**. The shell also arranges the output of the `find` command to go to the input of the `xargs` command via a pipe.

The `find` process is a producer here. The list of files it produces is *consumed* by the `xargs` command. `xargs` collects a bunch of filenames and invokes `egrep` on them. Lastly, the output appears in the console. It is important to note that both the processes are running concurrently, as shown in the following diagram:



```
.
├── #
├── aaaxx
├── empty
├── file
├── htom
├── htom.c
├── is
├── my.txt
├── perimeter
├── perimeter.c
├── square_perim
├── square_perim.c
├── this
├── try.txt
├── tst.txt
├── Untitled.html
├── yyyybb
└── zsh_demo
    ├── calculations
    │   ├── africa
    │   │   ├── kenya
```

```
./Untitled.html:Mon Jun  5 10:23:38 IST 2017
./Untitled.html:Mon Jun  5 10:23:38 IST 2017
./Untitled.html:Mon Jun  5 10:23:38 IST 2017
./Untitled.html:Mon Jun  5 10:23:38 IST 2017
./Untitled.html:Mon Jun  5 10:23:38 IST 2017
./Untitled.html:Mon Jun  5 10:23:38 IST 2017
```

Both these processes are collaborating with each other, so our goal of recursively searching the directory tree is achieved. One process is producing the filenames. The other is searching these files. As these are running in parallel, we start getting the results as soon as there are some qualifying filenames. We start getting results faster, which means the system is *responsive*.

Quiz: What would happen if both these processes ran one after another? How would the system arrange that the result of the `find` command is communicated to the `xargs` command?

Just as in real life, collaboration needs communication. The *pipe* is the mechanism that enables the `find` process to communicate with the `xargs` process. The pipe acts both as a coordinator and as a communication mechanism.

Coordination and communication

We need to make sure that when the `find` process has nothing more to report, which means that it has found all the qualifying filenames, `egrep` should also stop!

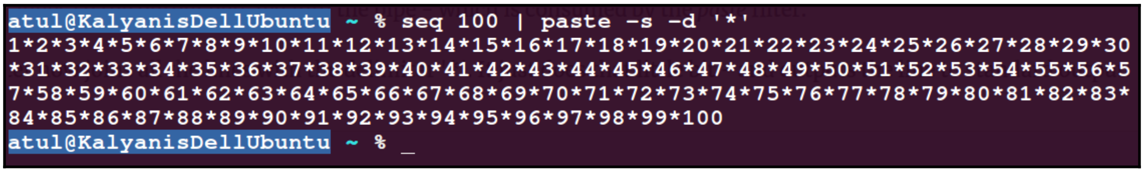
Similarly, if any of the processes in the pipeline quits for any reason, then the entire pipeline should stop.

For example, here is a pipeline that computes the factorial of 1,000:

```
% seq 1000 | paste -s -d '*' | bc
40238726007709377354370243392300398571937486421071463254379991042993\
85123986290205920442084869694048004799886101971960586316668729948085\
.... rest of the output truncated
```

The pipeline has three filters: `seq`, `paste`, and `bc`. The `seq` command just prints numbers from 1 to 1,000 and puts them in the console. The shell arranges things so that the output gets fed into the pipe that is consumed by the `paste` filter.

The `paste` filter now joins all the lines with the `*` delimiter. It just does that little bit, and outputs the line to standard output, as shown in the following screenshot:



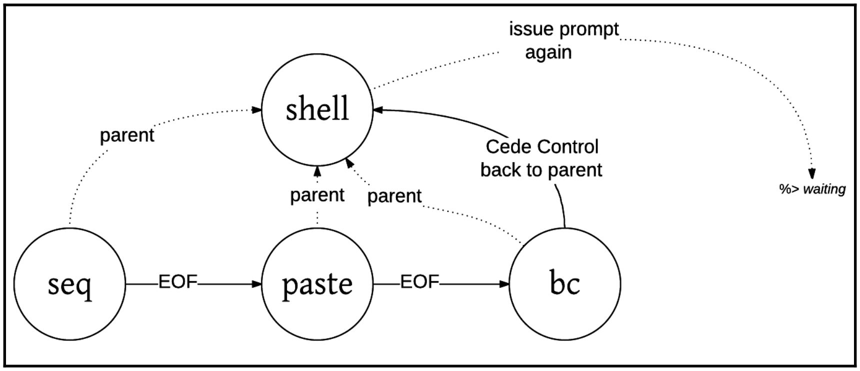
```
atul@KalyanisDellUbuntu ~ % seq 100 | paste -s -d '*'
1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19*20*21*22*23*24*25*26*27*28*29*30
*31*32*33*34*35*36*37*38*39*40*41*42*43*44*45*46*47*48*49*50*51*52*53*54*55*56*5
7*58*59*60*61*62*63*64*65*66*67*68*69*70*71*72*73*74*75*76*77*78*79*80*81*82*83*
84*85*86*87*88*89*90*91*92*93*94*95*96*97*98*99*100
atul@KalyanisDellUbuntu ~ % _
```

The paste command writes to the console; the shell has arranged the output to go into a pipe again. At the other end, the consumer is bc. The bc command or filter is capable of arbitrary precision arithmetic; in simpler terms, it can perform very large computations.

When the seq command exits normally, this triggers an **EOF (end of file)** on the pipe. This tells paste that the input stream has nothing more to read, so it does the joining, writes the output on the console (which is going to a pipe really), and quits in turn.

This quitting results in an EOF for the bc process, so it computes the multiplication, prints the result to the standard output, which is really a console, and finally quits. This is an *ordered shutdown;* no more work needs to be done, so exit and relinquish the computing resources for other concurrent processes, if there are any. The melodramatic term for this marker is *poison pill.* See https://dzone.com/articles/producers-and-consumers-part-3 for more information.

At this point, the pipeline processing is done and we get back to the shell prompt again, as shown in the following diagram:



Unbeknownst to all the filters participating in the pipeline, the parent shell has arranged for this coordination. This ability of the framework to be composed of smaller parts *without the parts themselves being aware of the composition* is a great design pattern, called *pipes and filters.* We will see how composition is one central theme, yielding robust concurrent programs.

What happens when the seq process produces numbers way too fast? Would the consumer (paste in this case) get overwhelmed? Aha, no! The pipeline also has an implicit *flow control* built into it. This is yet another central theme, called *back-pressure*, where the faster producer (or consumer) is forced to wait so the slower filter catches up.

Let's next look at this *flow control* mechanism.

Flow control

The wonderful idea behind the previously mentioned pipeline is that the `find` producer and the `xargs` consumer don't know each other. That is, you could compose any filters using pipes. This is the celebrated *pipes and filters* design pattern in action. The shell command line gives you a framework that enables you to compose any filters together into a pipeline.

What does it give us? You can reuse the *same filter* in unexpected and creative ways to get your work done. Each filter just needs to follow a simple protocol of accepting input on file descriptor 0, writing output to file descriptor 1, and writing errors to descriptor 2.

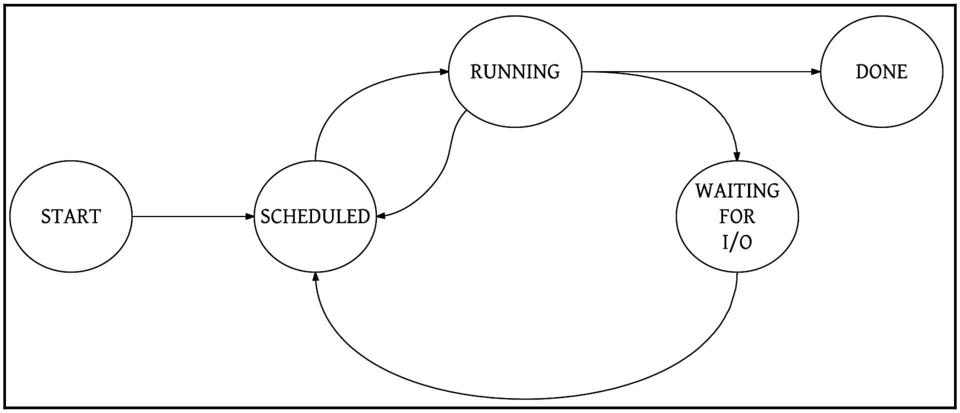
You can refer to a UNIX shell programming guide for more information on descriptors and related ideas. My personal favorite is *UNIX Power Tools, 3rd Ed.* by Jerry Peek et al.

Flow control means we are trying to regulate the flow of something. When you tell someone to talk slowly so that you can follow their meaning, you are trying to control the flow of words.

Flow control is essential in ensuring that the *producer* (such as a fast speaker) does not *overwhelm* the *consumer* (such as a listener). In the example we have been working on, the `find` process could produce filenames faster; the `egrep` process might need more time to process each file. The `find` producer works at its own pace, and does not care about a slow consumer.

If the pipe gets full because of the slower consumption by `xargs`, the output call by `find` is blocked; that is, the process is waiting, and so it can't run. This pauses `find` until the consumer has finally found the time to consume some filenames and the pipe has some free space. It works the other way around as well. A fast consumer blocks an empty pipe. Blocking is a process-level mechanism, and `find` (or any other filter) does not know it is blocking or unblocking.

The moment a process starts *running,* it will perform its computation for the *find* filter, ferret out some filenames, and output these to the console. Here is a simplified state diagram , showing a process's life cycle:



What is this *scheduled* state? As mentioned, a running process could get blocked waiting for some I/O to happen, and thus it cannot use the CPU. So it is put on the back burner for a while, and other processes, waiting their turn, are given a chance to run. Drawing a parallel with the previously mentioned receptionist scenario, the receptionist can ask us to be seated and wait a while, and then move on to the next guest in the queue.

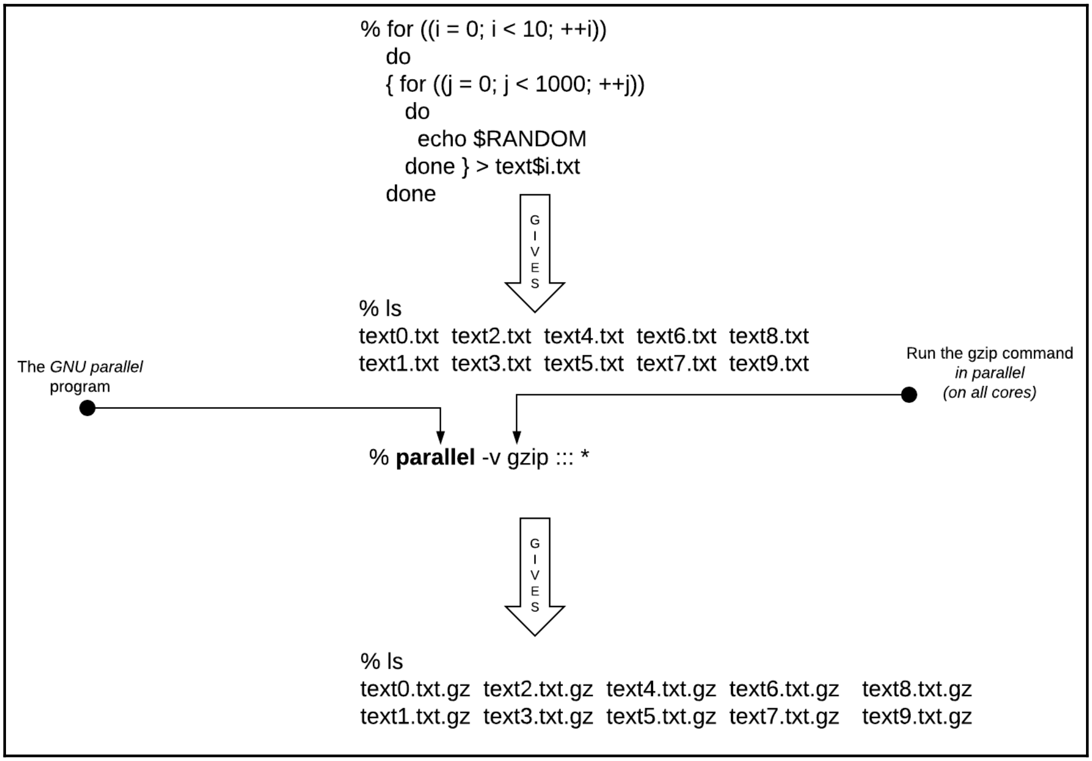
The other idea is that the process has run its allocated *slice of time,* so other processes should now get a chance, too. In this case, even though the process can run and utilize the CPU, it is moved back to the *scheduled* state, and can run again once other processes have used their run slices. This is *preemptive* multitasking we have here, which makes it a fair world to live in! Processes need to run so that useful work can happen. Preemptive scheduling is an idea to help each process get a slice of CPU time.

However, there is another notion that could throw a spanner into this scheme of things. A process with a *higher priority* is given preference over *lower priority* processes.

A real-world example should help make this clear. While driving on roads, when we see an ambulance or a police car with a screaming siren, we are required to make way for them. Similarly, a process executing a piece of business logic may need more priority than the data backup process.

Divide and conquer

GNU parallel (`https://www.gnu.org/software/parallel/`) is a tool for executing commands in parallel on one or more nodes. The following diagram shows a simple run where we generate 10 text files and zip them (using the `gzip` command) in parallel. All the available cores are used to run `gzip` , thereby reducing the *overall* processing time:



```
% for ((i = 0; i < 10; ++i))
   do
   { for ((j = 0; j < 1000; ++j))
      do
        echo $RANDOM
      done } > text$i.txt
   done
```

```
% ls
text0.txt  text2.txt  text4.txt  text6.txt  text8.txt
text1.txt  text3.txt  text5.txt  text7.txt  text9.txt
```

```
% parallel -v gzip ::: *
```

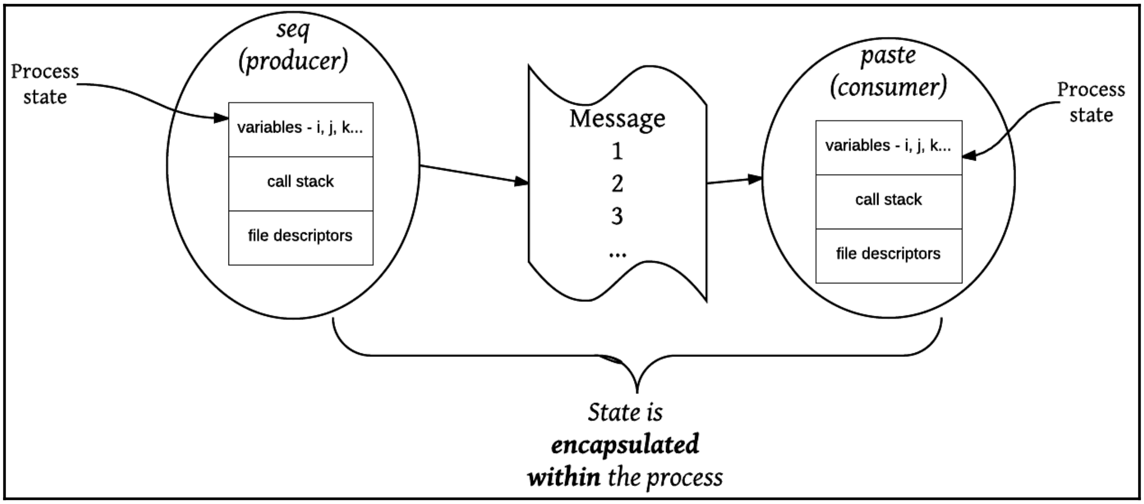
```
% ls
text0.txt.gz  text2.txt.gz  text4.txt.gz  text6.txt.gz   text8.txt.gz
text1.txt.gz  text3.txt.gz  text5.txt.gz  text7.txt.gz   text9.txt.gz
```

GIVES
The *GNU parallel* program
Run the gzip command *in parallel* *(on all cores)*
GIVES

The core principle at work is *divide and conquer*. We see the same principle again and again: a *parallelizable* job is split into pieces, each of which is processed in parallel (thereby overlapping processing and reducing the time). The `parallel` command also allows you to distribute long-running jobs on different nodes (machines), thereby allowing you to harness the idle (possibly unused) cores to process jobs quickly.

The concept of state

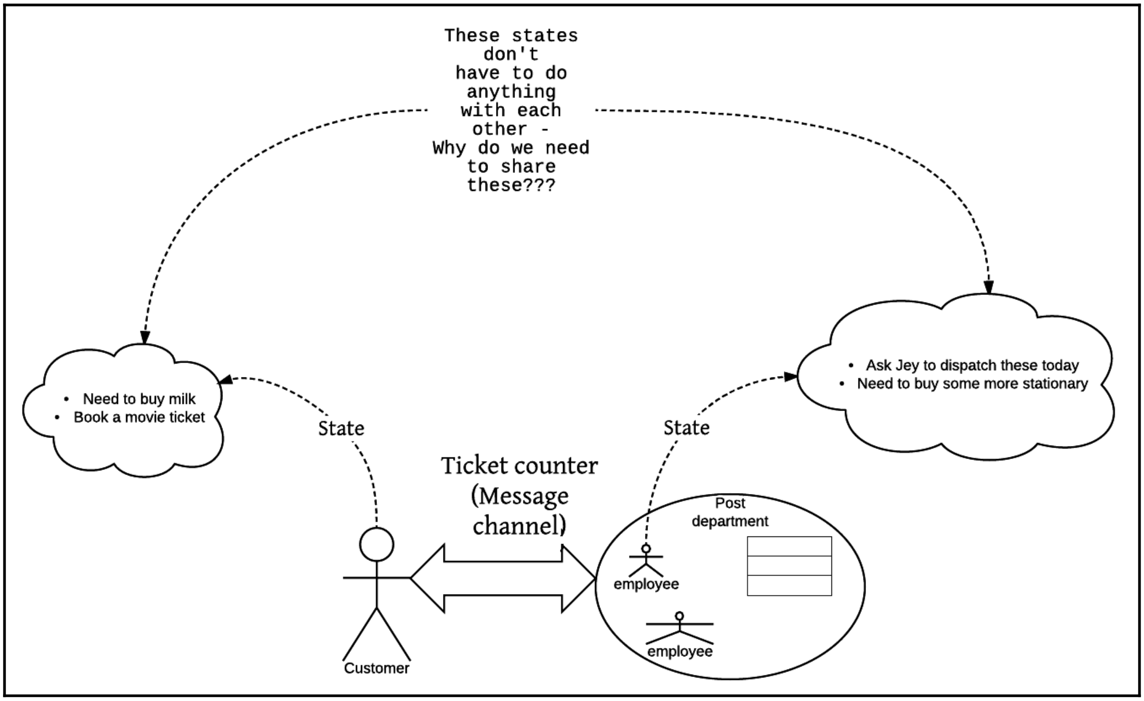
The communication depicted in the preceding section could be looked at as *message passing;* `find` is passing on the filename as a message to the `egrep` process, or `seq` is passing messages (numbers) to the `paste` process. Generally speaking, a producer is sending messages to the consumer for consuming, as shown in the following diagram:



As shown in the preceding diagram, each process has its own state by design, and this state is hidden from other processes. The processes communicate with explicit messaging channels, in the same way that a pipe directs the flow of water.

This notion of state is very important to understand the various upcoming concurrency patterns. We could look at the state as data in a certain stage of processing. For example, the paste process could be using program counters to generate the numbers. It could also be writing the numbers to the standard output (file descriptor 1; by default, the console). At the same time, the paste process is processing its input and writing data to its standard output. Both processes do not care about each other's state; in fact, they don't even know anything about the other process.

The real world is full of encapsulated states. The following diagram shows an example:


These states don't have to do anything with each other - Why do we need to share these???
Need to buy milk
Book a movie ticket
State
Ask Jey to dispatch these today
Need to buy some more stationary
State
Ticket counter (Message channel)
Post department
employee
employee
Customer

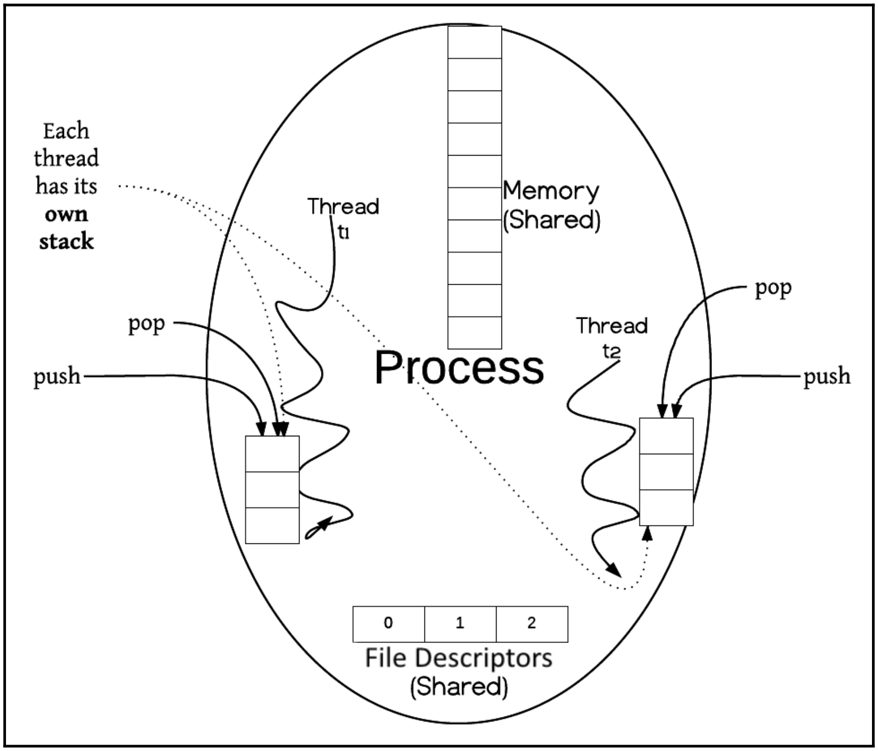
It defeats common sense to share the state (*the need to buy milk*) with the postal department employee. It is useless for him to know it, and it could create confusion.

Likewise, the employee will be going about his daily tasks and has a state of his own. Why do we, as consumers, need to know the internal details *(*state*)* of how he is going to manage his work (*dispatch this big stack of letters*)? The world is concurrent, and the various entities in it also hide unnecessary details from each other to avoid confusion. If we don't hide the internal details (that is, the state), it would create havoc.

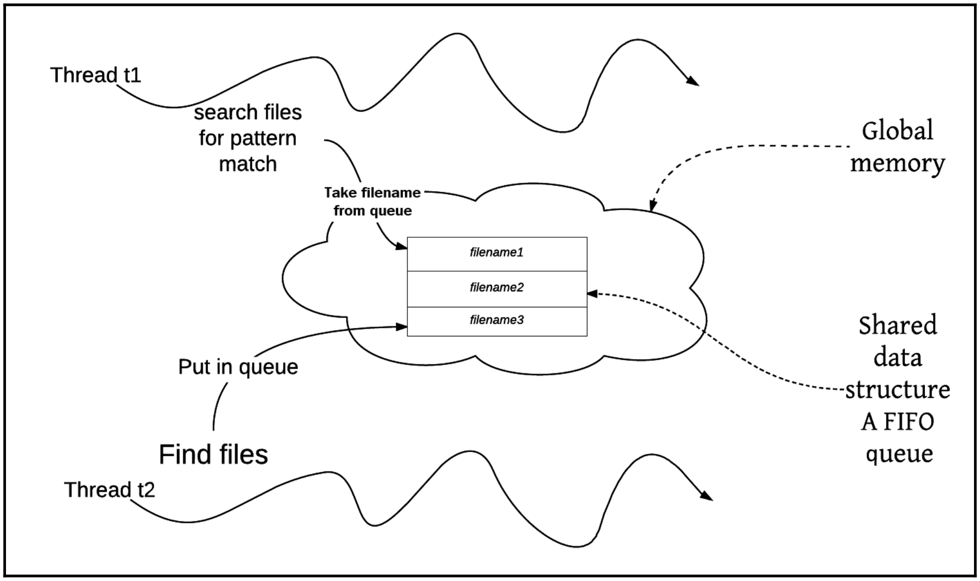
We could ask whether there is a *global shared memory*. If there is, then we could use it as a *message channel.* Using a shared data structure of our choice, the producer could put the data in it for subsequent consumption; that is, the memory is used as a channel of communication.

The shared memory and shared state model

What if we write a multithreaded program to achieve the same result? A ***thread*** *of execution* is a *sequence* of programming instructions, *scheduled and managed by the operating system.* A process could contain multiple threads; in other words, a process is a *container* for *concurrently executing* threads, as shown in the following diagram:



As shown in the preceding diagram, multiple threads share the process memory. Two concurrently running processes do not share memory or any other resources, such as file descriptors. In other words, different concurrent processes have their own address space, while multiple threads within the same process share their address space. Each thread also has a stack of its own. This stack is used for returning after a process call. Locally scoped variables are also created on the stack. The relationships between these elements are shown in the following diagram:



As shown in the preceding diagram, both the threads communicate via the process's global memory. There is a **FIFO (first in first out)** queue in which the producer thread `t1` enters the filenames. The consumer thread, `t2`, picks up the queue entries as and when it can.

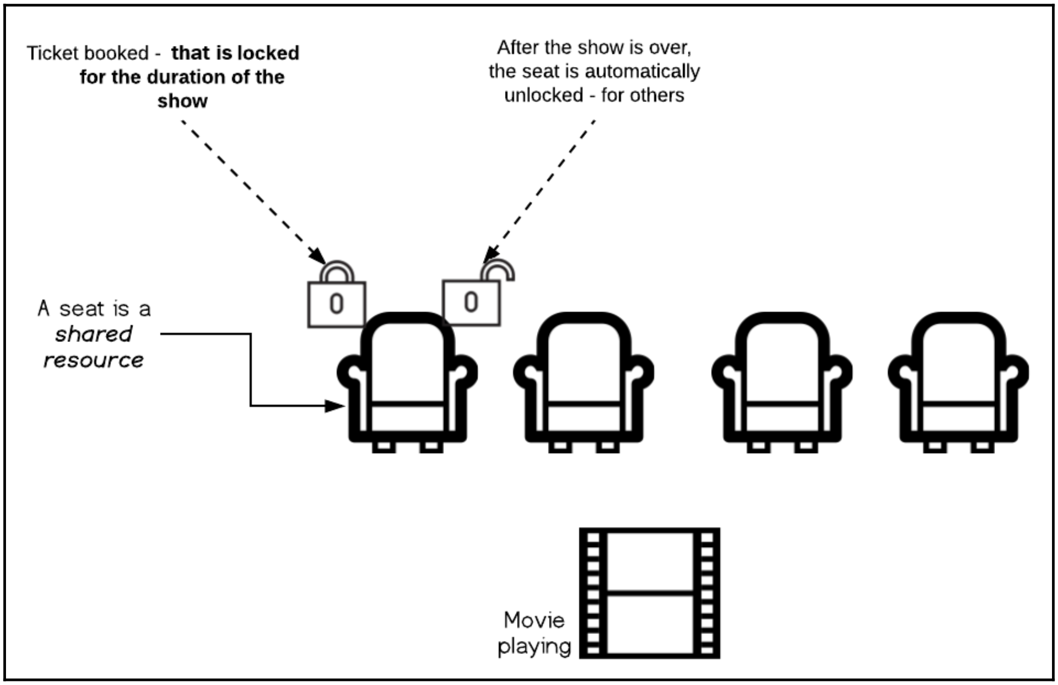
What does this data structure do? It works on a similar principle as the aforementioned pipe. The producer can produce items as fast or slow as it can. Likewise, the consumer thread picks the entries from the queue as needed. Both work at their own pace, without caring or knowing of each other.

Exchanging information this way looks simpler. However, it brings with it a host of problems. Access to the shared data structure needs to be *synchronized correctly*. This is surprisingly hard to achieve. The next few sections will deal with the various issues that crop up. We will also see the various paradigms that shy away from the shared state model and tilt towards message passing.

Threads interleaving – the need for synchronization

The way threads get scheduled to run is within the realm of the operating system. Factors such as the system load, the number of processes running at a time on the machine, make thread scheduling *unpredictable.* A good example is a seat in a movie hall.

Let's say that the movie that is playing is a big crowd-puller. We need to follow a protocol; we book a ticket by *reserving* a seat. The following diagram shows the rules that are based around the ticket booking system:

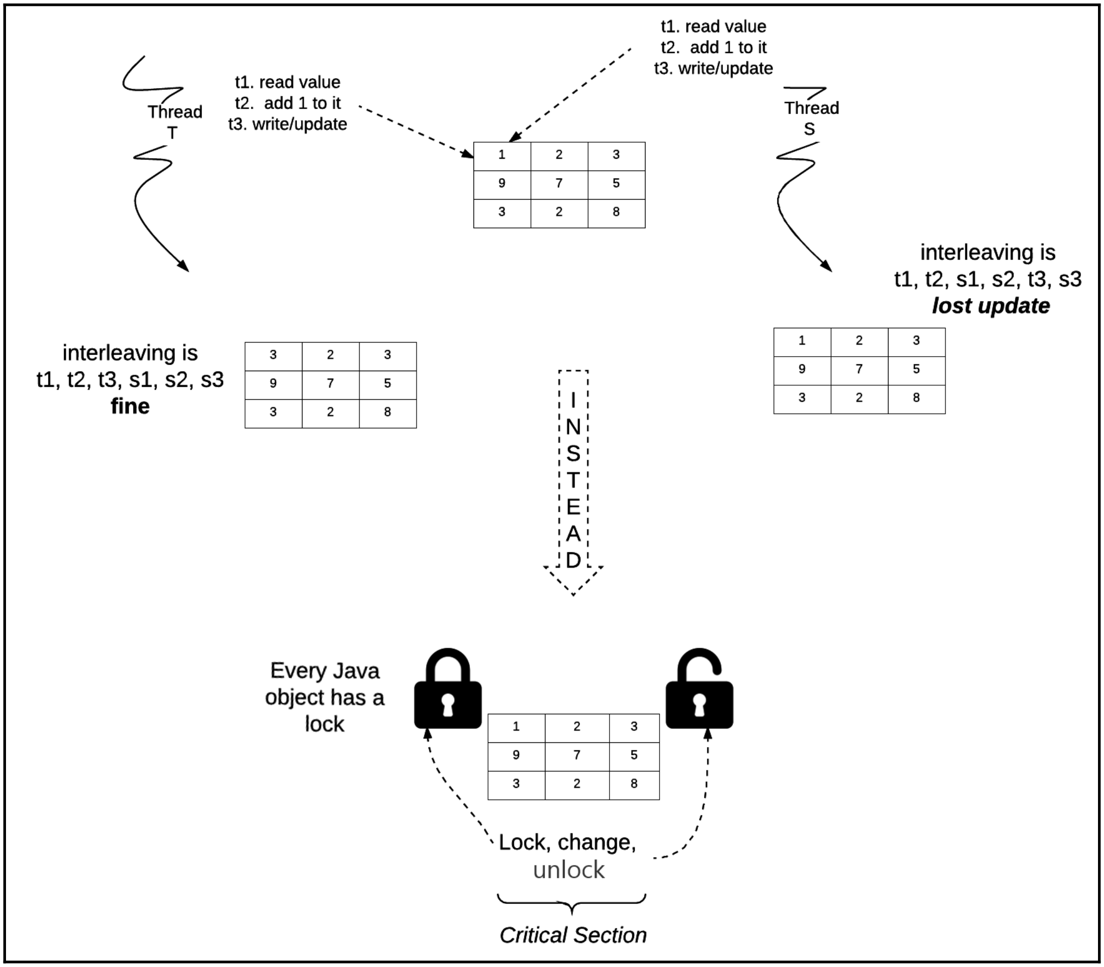

Ticket booked - **that is locked for the duration of the show**
After the show is over, the seat is automatically unlocked - for others
A seat is a *shared resource*
0
0
Movie playing

What if the booking is erroneously given to two people at the same time? The result would be chaotic, as both would rightfully try to go and occupy the seat *at the same time.*

There is a certain framework at work to make sure this situation does not happen in practice. The seat is shared among interested people and one person needs to book it in advance.

Likewise, threads need to lock a resource (that is, a shared, mutable data structure). The problem is with explicit locking. If the onus of correctly synchronizing is with the application, then someone, someday may forget to do it right and all hell will break loose.

To illustrate the need for correct synchronization, the following diagram shows an integer variable shared between two threads:



As shown in the preceding diagram, if the interleaving happens to be right, things may work as expected. Otherwise, we will have a *lost update* to deal with.

Instead, just like a movie ticket acting as a lock, every Java object has a lock. A thread acquires it, performs the state mutations, and unlocks. This entire sequence is a *critical section.* If your critical section needs to mutate a set of objects, you need to lock each one separately.

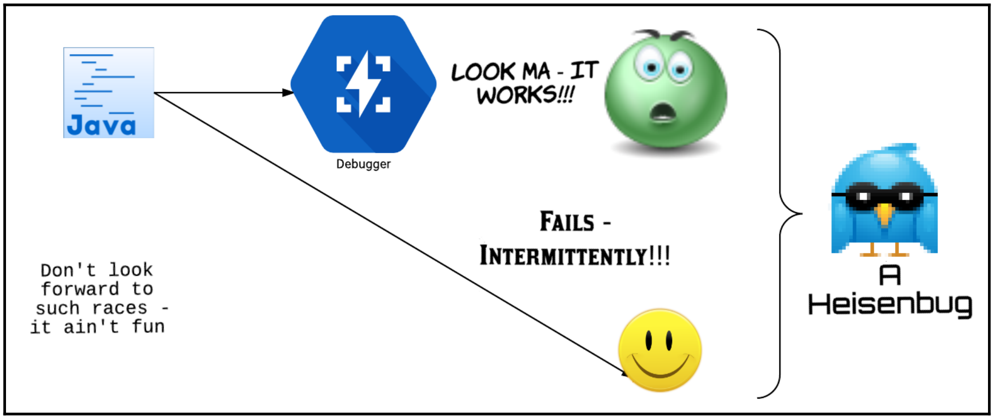
Generally, the advice is to keep the critical section as small as possible. We will discuss the reason in the next section.

Race conditions and heisenbugs

The lost update is an example of a **race condition**. A race condition means that the correctness of the program (the way it is expected to work) depends on the relative timing of the threads getting scheduled. So sometimes it works right, and sometimes it does not!

This is a situation that is very hard to debug. We need to *reproduce* a problem to investigate it, possibly running it in a debugger. What makes it hard is that the race condition cannot be reproduced! The sequence of interleaved instructions depends on the relative timing of events that are strongly influenced by the environment. Delays are caused by other running programs, other network traffic, OS scheduling decisions, variations in the processor's clock speed, and so on. A program containing a race condition may exhibit different behavior, at different times.

A heisenbug and race conditions are explained in the diagram:


Java
Debugger
LOOK MA - IT WORKS!!!
FAILS - INTERMITTENTLY!!!
Don't look forward to such races - it ain't fun
A Heisenbug

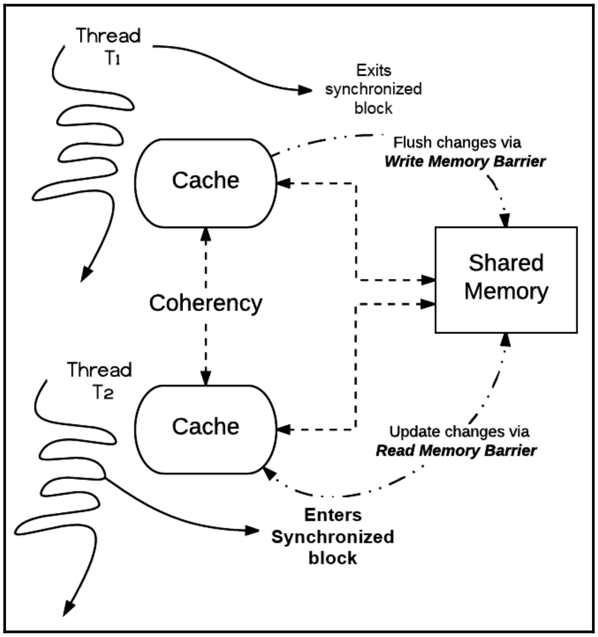
These are **heisenbugs**—essentially nondeterministic and hard to reproduce. If we try debugging a heisenbug by attaching a debugger, the bug may disappear!

There is simply no way to debug and fix these. There is some tooling support, such as the *tha* tool (`https://docs.oracle.com/cd/E37069_01/html/E54439/tha-1.html`) and *helgrind* (`http://valgrind.org/docs/manual/drd-manual.html`); however, these are language or platform specific, and don't necessarily prove the absence of races.

Clearly, we need to avoid race conditions *by design*, hence the need to study concurrency patterns and this book.

Correct memory visibility and happens-before

There is yet another problem that could come up with incorrect synchronization: incorrect *memory visibility*. The **synchronized** keyword prevents the execution of critical sections by more than one thread. The synchronized keyword also makes sure the thread's local memory syncs up correctly with the shared memory, as shown in the following diagram:


Thread T1
Exits synchronized block
Flush changes via *Write Memory Barrier*
Cache
Coherency
Shared Memory
Thread T2
Cache
Update changes via *Read Memory Barrier*
Enters Synchronized block

What is this local memory? Note that on a multicore CPU, each CPU has a *cache* for performance reasons. This cache needs to be synced with the main shared memory. The *cache coherence* needs to be ensured so that each thread running on a CPU has the right view of the shared data.

As shown in the preceding diagram, when a thread exits a synchronized block, it issues a write barrier, thereby syncing the changes in its cache to the shared memory. On the other hand, when a thread enters a synchronized block, it issues a *read barrier*, so its local cache is updated with the latest changes in the shared memory.

Note that this is again not easy. In fact, very seasoned programmers were tripped up when they proposed the *double-checked locking pattern.* This seemingly brilliant optimization was found to be flawed in light of the preceding memory synchronization rules.

For more information on this botched optimization attempt, take a look at `https://www.javaworld.com/article/2074979/java-concurrency/double-checked-locking--clever--but-broken.html`.

However, Java's `volatile` keyword guarantees correct memory visibility. You don't need to synchronize just to ensure correct visibility. This keyword also guarantees ordering, which is a *happens-before* relationship. A happens-before relationship ensures that any memory writes done by a statement are visible to another statement, as shown in the following code:

```
private int i = 0;
private int j = 0;
private volatile boolean k = false;
// first thread sets values
i = 1;
j = 2;
k = true;
```

All the variable values will be set to have a happens-before relationship because of the volatile that is being set. This means that after the variable `k` is set, all the previous changes are guaranteed to have happened! So the value of the *i* and *j* variables are guaranteed to be set, as shown in the following snippet:

```
        // second thread prints them
System.out.println("i = " + i + ", j = " + j + ", k = " + k) // the i and j
values will have been flushed to memory
```

The `volatile` keyword, however, *does not* guarantee *atomicity. S*ee `http://tutorials.jenkov.com/java-concurrency/volatile.html` for more information.

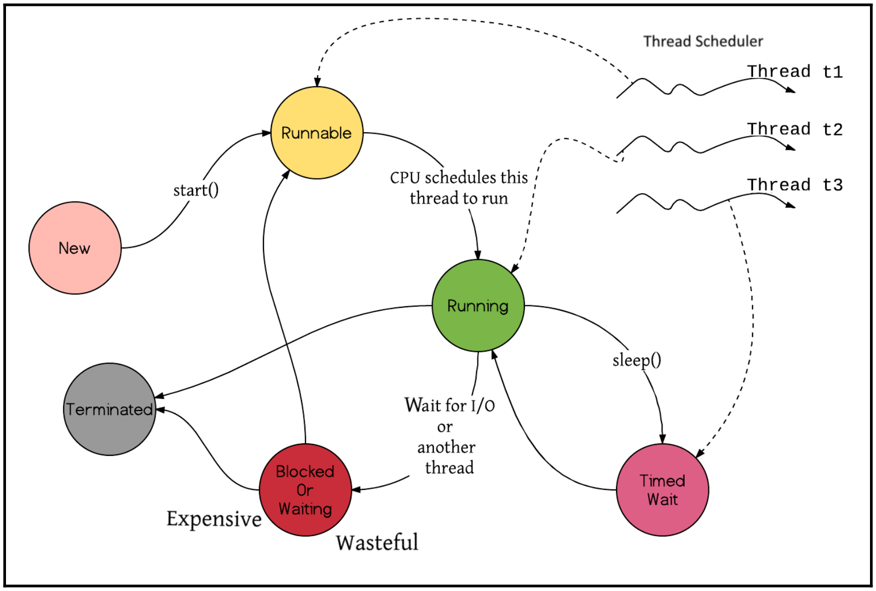
Sharing, blocking, and fairness

Just like the process life cycle, threads also have a life cycle. The following figure shows the various thread states. It shows three threads, t1, t2, and t3, in the *Runnable*, *Running*, and *Timed Wait* states. Here is a brief explanation of each state:

- **New**: When a `Thread` object is just created. The thread is ***not alive***, as yet.
- **Runnable:** When the `start()` function is called on the `thread` object, its state is changed to `runnable`. As shown in the following diagram, a **thread scheduler** kicks in to decide when to schedule this thread to be run.

- **Running:** Eventually, the thread scheduler picks one of the threads from the `runnable` thread pool and changes its state to `Running`. This is when the the thread starts executing. The CPU starts the execution of this thread.
- **Blocked:** The thread is waiting for a monitor lock. As noted previously, for a shared resource such as a *mutable* memory data structure, only the thread can access/read/mutate it. While a thread has the lock, other threads will be *blocked.*
- **Waiting**: Wait for another thread to perform an action. Threads commonly block while doing I/O.
- **Timed Wait:** The thread waits for an event for a finite amount of time.
- **Terminated:** The thread is dead and cannot go back to any other state.

A thread goes back to the `Runnable` state once the event it waited for happens:



As shown in the preceding diagram, a blocking thread is expensive and wasteful. Why is this so? Remember, a thread is a **resource** itself. Instead of a thread that is just blocked and doing nothing, it would be far more optimal to employ it for processing something else. Wouldn't it be good to allocate the thread to do something useful?

Keeping the critical sections small is one way to be fair to all threads. No thread holds the lock for a long time (although this is can be altered).

Could we avoid blocking the thread and instead use it for something else? Well, that brings us to the theme of *asynchronous* versus *synchronous* executions.

Asynchronous versus synchronous executions

Blocking operations are bad, as they waste resources. By *blocking*, we mean operations that take a long time to complete. Synchronous execution allows tasks to execute in a sequence, waiting for the current operation to complete before starting with the next. For example, making a phone call is synchronous. We dial the number, wait for the person on other side to say "hello," and then proceed with the conversation.

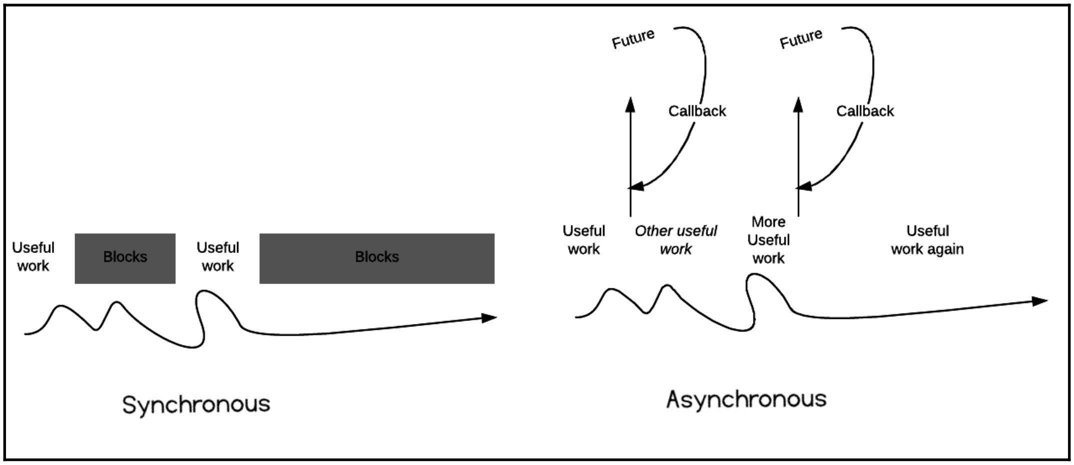
On the other hand, posting a letter is done *asynchronously.* One does not post a letter and block for its response. We post it and then go our own way, doing other stuff. Some time in the future, we can expect a response (or an error if the letter could not be delivered).

As another example, some restaurants give you a lunch token. You pay for lunch and get a token, which is a promise that you will get to eat *in the near future.* If there is a big queue at the counter, you may occupy yourself with something else in the meantime and then try again later.

This is an **asynchronous model**.

Imagine what would happen in a case where there is no token system in place. You pay and then just wait for your turn to be served, blocked while users at the front of the queue are served.

Coming back to the software world, file and network I/O are blocking. So are database calls using blocking drivers, as shown in the following diagram:



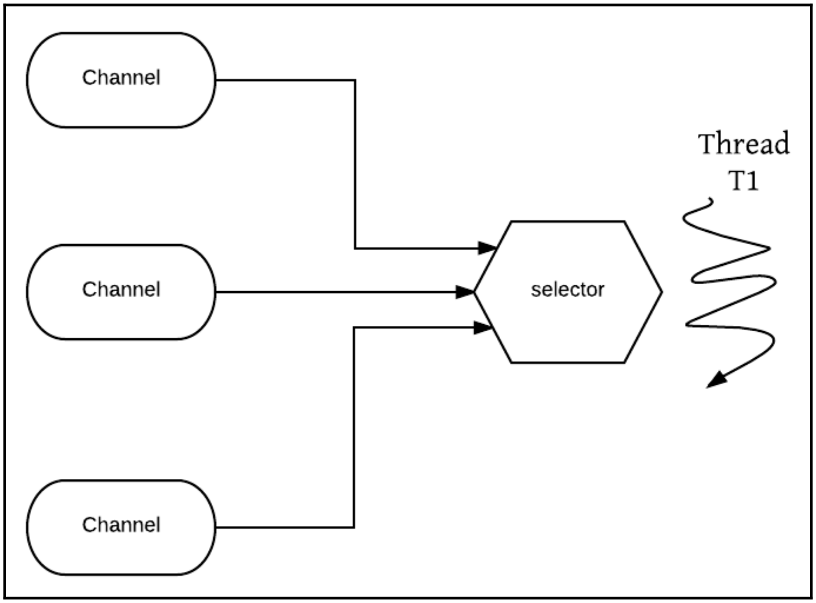
Instead of blocking and wasting the thread doing nothing, we could look at the workflow as a mix of *unblocking* and *blocking* tasks. We would then handle a blocking task using a *future:* an abstraction that will complete eventually and call us back with the results or an error.

This is a change in the paradigm, where we start thinking of designing our tasks differently and representing them using higher-level abstractions, such as a *future* (which we discussed previously), and not deal with the threads directly. *Actors* are another abstraction over threads, that is, another paradigm.

Futures offer *composability.* They are *monads.* You can create a pipeline of future operations to perform higher-level computations, as we will soon see in an upcoming chapter.

Java's nonblocking I/O

Java NIO (New IO) is a nonblocking I/O API for Java. This NIO is an alternative to the standard Java I/O API. It provides abstractions such as channels, buffers, and selectors. The idea is to provide an implementation that can use the most efficient operations provided by the operating system, as shown in the following screenshot:



A channel is just a bidirectional I/O stream. A single thread can monitor all the channels an application has opened. Data arriving at any channel is an *event*, and the listening thread is notified of its arrival.

The ***selector*** uses event notification: a thread can then check whether the I/O is complete without any need for *blocking.* A single thread can handle multiple concurrent connections.

This translates into two primary benefits:

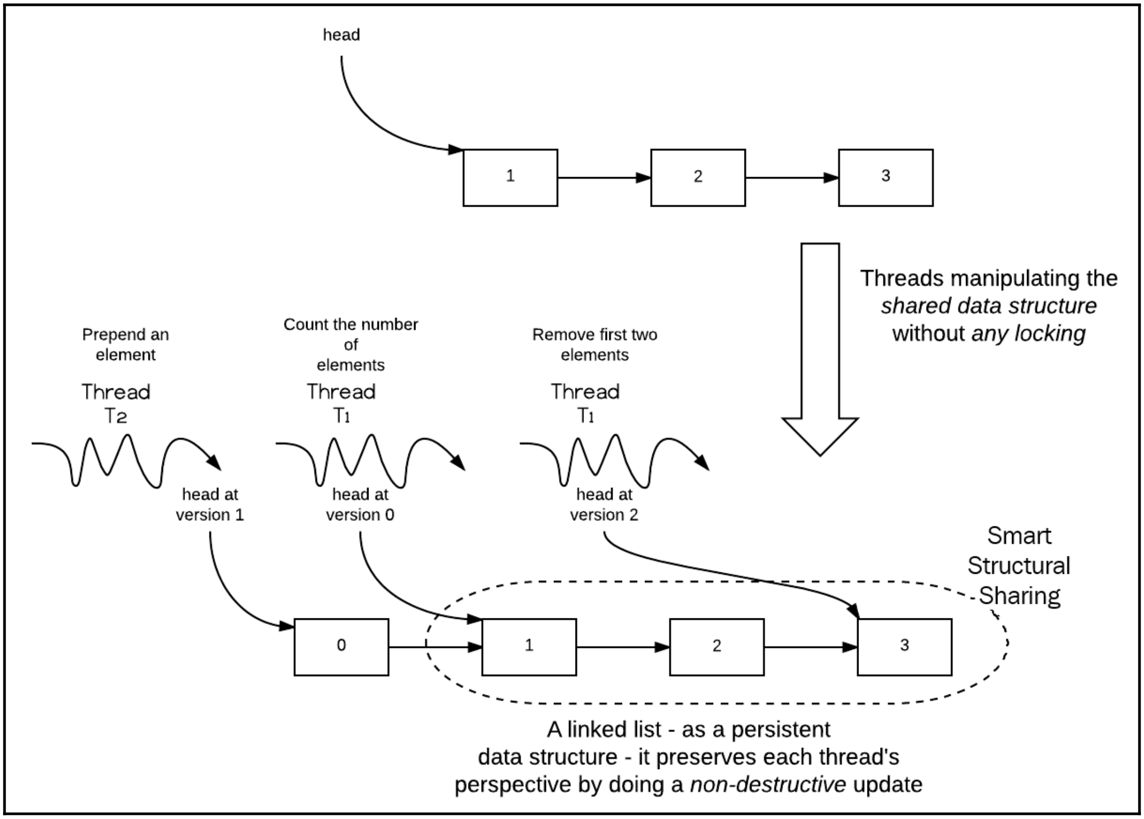
- Overall, you would need fewer threads. As a thread has a memory footprint, the memory management would have less overhead.
- Threads could do something useful when there is no I/O. This opens up the possibility of optimization, as threads are a valuable resource.

The *Netty* framework (`https://netty.io/`) is an NIO-based client-server framework. The *Play* framework is a high-performance, reactive web framework based on Netty.

Of patterns and paradigms

Moving away from explicit state management is a very prominent theme in programming. We always need a higher level of abstraction over the shared state model. As explained earlier, explicit locking does not cut it.

The various concurrency patterns that we will study in this book try to stay away from explicit locking. For example, *immutability* is a major theme, giving us *persistent data structures.* A persistent data structure performs a smart copy on a write, thereby avoiding mutation altogether, as shown in the following diagram:


head
1
2
3
Threads manipulating the *shared data structure* without *any locking*
Prepend an element
Thread T2
head at version 1
Count the number of elements
Thread T1
head at version 0
Remove first two elements
Thread T1
head at version 2
Smart Structural Sharing
0
1
2
3
A linked list - as a persistent data structure - it preserves each thread's perspective by doing a *non-destructive* update

As shown in the preceding diagram, the original linked list has three elements, {1, 2, 3}. The head element of the list has the value 1. Thread T1 starts counting the number of elements in the list.

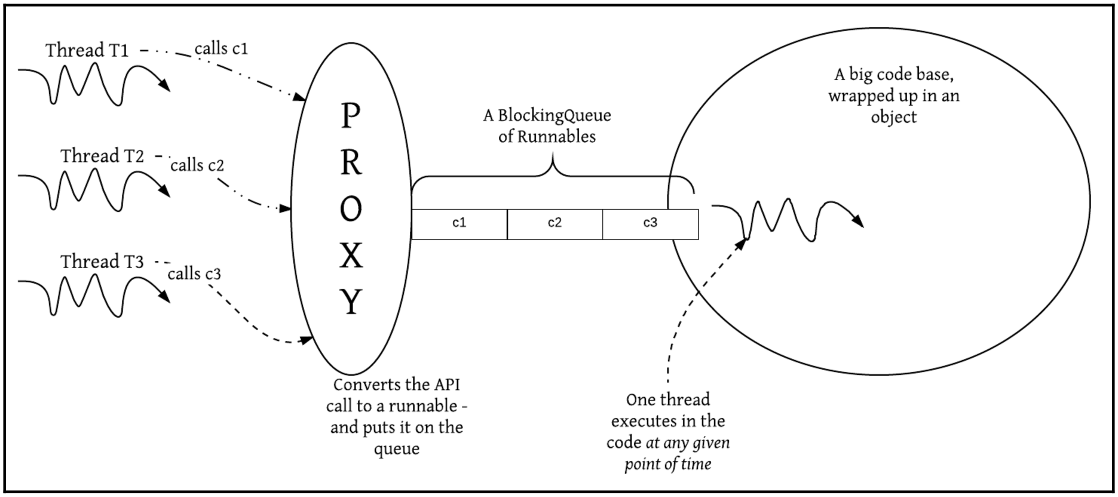
At any point in time, thread T2 can prepend an element to the original list. This should not disturb the world of thread T1; it should still see the original list as it is. In other words, T1's version of the list as it sees it is preserved. Any change in the list creates a new version of the data structure. As all the versions live as long as they are needed (that is, are *persistent*), we don't need any locking.

Similarly, thread T2 removes the first two elements. This is achieved by just setting its head to the third element; again, this doesn't disturb the *state* as seen by T1 and T2.

This is essentially *copy-on-write*. Immutability is a cornerstone of functional programming languages.

A typical concurrency pattern is an *active object.* For example, how would you consume a legacy code base from multiple threads? The code base was written without any parallelism in mind, the state is strewn around, and it is almost impossible to figure out.

A brute-force approach could be to just wrap up the code in a big *God object.* Each thread could lock this object, use it, and relinquish the lock. However, this design would hurt concurrency, as it means that other threads would simply have to wait! Instead, we could use an *active object,* as shown in the following diagram:


Thread T1
calls c1
Thread T2
calls c2
Thread T3
calls c3
P R O X Y
Converts the API call to a runnable - and puts it on the queue
A BlockingQueue of Runnables
c1
c2
c3
A big code base, wrapped up in an object
One thread executes in the code *at any given point of time*

To use this active object, a proxy sits in between the caller threads and the actual code base. It converts each invocation of the API into a *runnable* and puts it in a blocking queue (a thread-safe FIFO queue).

There is just one thread running in the `God` object. It executes the runnables on the queue one by one, in contrast to how a typical Java object method is invoked (passively). Here, the object itself executes the work placed on the queue, hence the term *active object.*

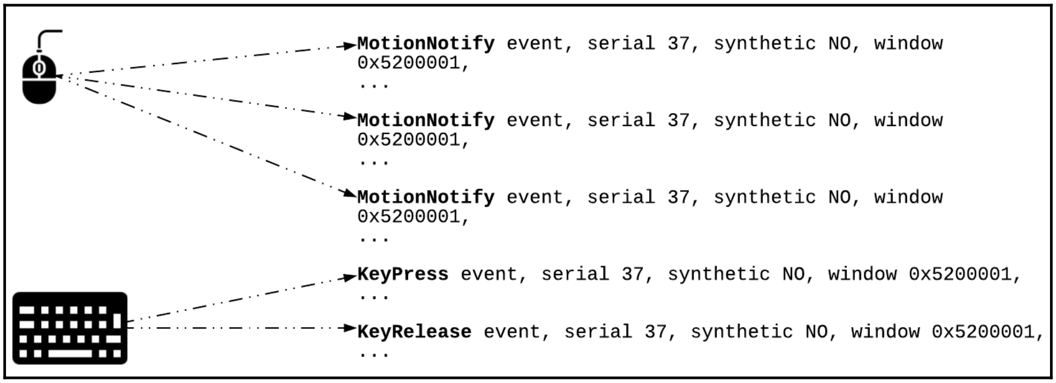
The rest of this chapter describes the many patterns and paradigms, that have evolved over the years, and are used in order to avoid the explicit locking of the shared state.

Event-driven architecture

Event-driven programming is a programming style in which code executes in response to an *event*, such as a keypress or a mouse click. In short, the flow of a program is driven by events.

GUI programming is an example of event-driven programming. For example, X Windows (driving most of your Linux GUI) processes a series of XEvents. Every keypress, mouse button press or release, and mouse movement generates a series of events. If you are on Linux, there is a command called `xev`. Running it via Terminal spawns a window. When moving a mouse over the window or pressing some keys, you can see the events that are generated.

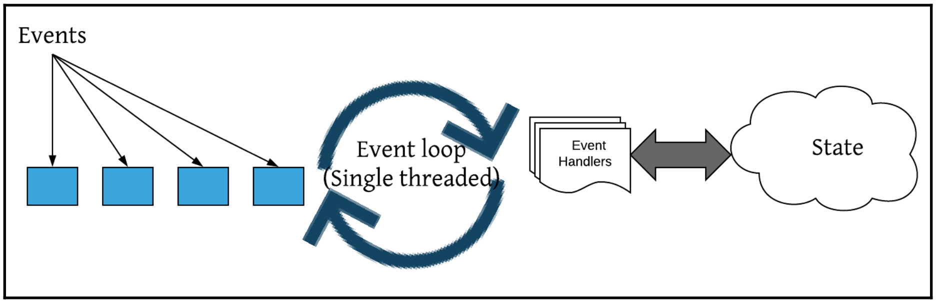
Here is a capture of the `xev` program on my Linux laptop:



```
MotionNotify event, serial 37, synthetic NO, window
0x5200001,
...
MotionNotify event, serial 37, synthetic NO, window
0x5200001,
...
MotionNotify event, serial 37, synthetic NO, window
0x5200001,
...
KeyPress event, serial 37, synthetic NO, window 0x5200001,
...
KeyRelease event, serial 37, synthetic NO, window 0x5200001,
...
```

You can plug in a *callback*, which gets triggered upon the reception of such an event. For example, an editor program could use keypress events to update its state (resulting in its documents being edited). Traditional event-driven programming could create a complex *callback flow*, thereby making it hard to figure out the control flows in the code.

Event-driven architecture **(EDA)** helps in decoupling a system's modules. Components communicate using events, which are encapsulated in *messages*. A component that emits an event does not know anything about the consumers. This makes EDA extremely loosely coupled. The architecture is inherently asynchronous. The producer is oblivious of the consumers of the event messages. This process is shown in the following diagram:



Given one thread and an *event loop,* with the callbacks executing quickly, we have a nice architecture. How does all this relate to concurrency? There could be multiple event loops running on a pool of threads. Thread pooling is an essential concept, as we will see in the upcoming chapters.

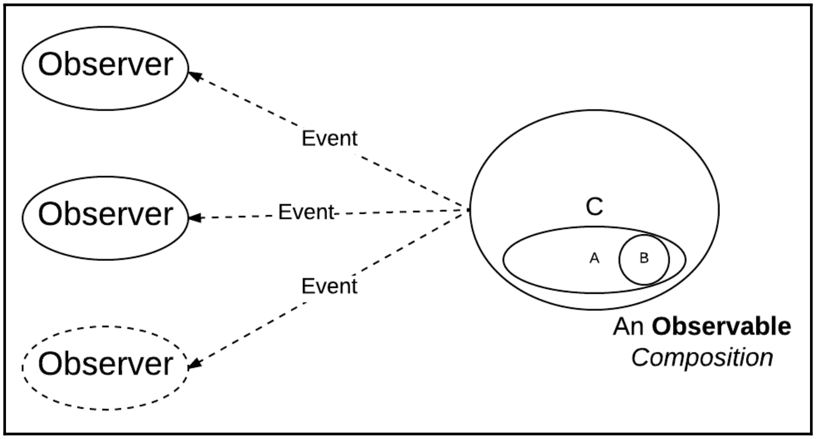
As we have seen, an event loop manages events. The events are passed on to an installed handler, where they are processed. The handler can react to an event in two ways: either it succeeds or it fails. A failure is passed to the event loop again as another event. The handler for the exception decides to react accordingly.

Reactive programming

Reactive programming is a related programming paradigm. A spreadsheet is an excellent example of a reactive application. If we set a formula and change any column value, the spreadsheet program reacts and computes the new result columns.

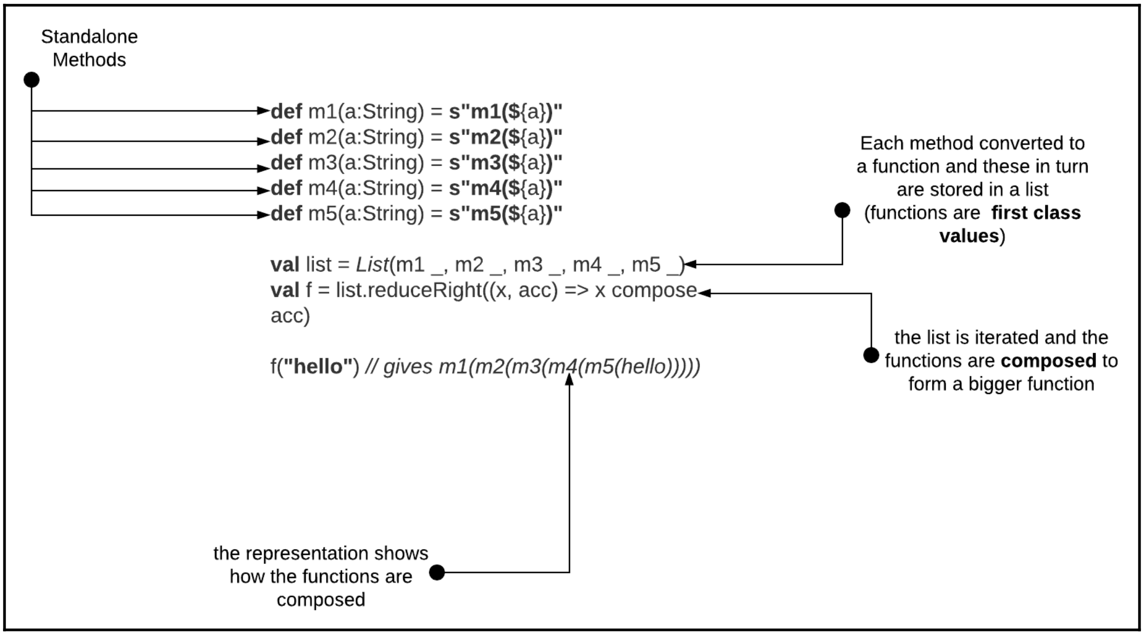
A *message-driven architecture* is the foundation of Reactive applications. A message-driven application may be event-driven, actor-based, or a combination of the two.

The following is a diagram of observable composition:



Composable *event streams* make event handling easier to understand. **Reactive Extensions (Rx)** is a framework that provides composable observables. At the heart of this framework is the *observer pattern*, with a functional flavor. The framework allows us to compose multiple observables. The observers are given the resulting event stream in an *asynchronous* fashion. For more information, see `http://reactivex.io/intro.html`.

Function of composition is shown in the following code:



```
def m1(a:String) = s"m1(${a})"
def m2(a:String) = s"m2(${a})"
def m3(a:String) = s"m3(${a})"
def m4(a:String) = s"m4(${a})"
def m5(a:String) = s"m5(${a})"

val list = List(m1 _, m2 _, m3 _, m4 _, m5 _)
val f = list.reduceRight((x, acc) => x compose
acc)

f("hello") // gives m1(m2(m3(m4(m5(hello)))))
```

Each method converted to a function and these in turn are stored in a list (functions are **first class values**)

the list is iterated and the functions are **composed** to form a bigger function

the representation shows how the functions are composed

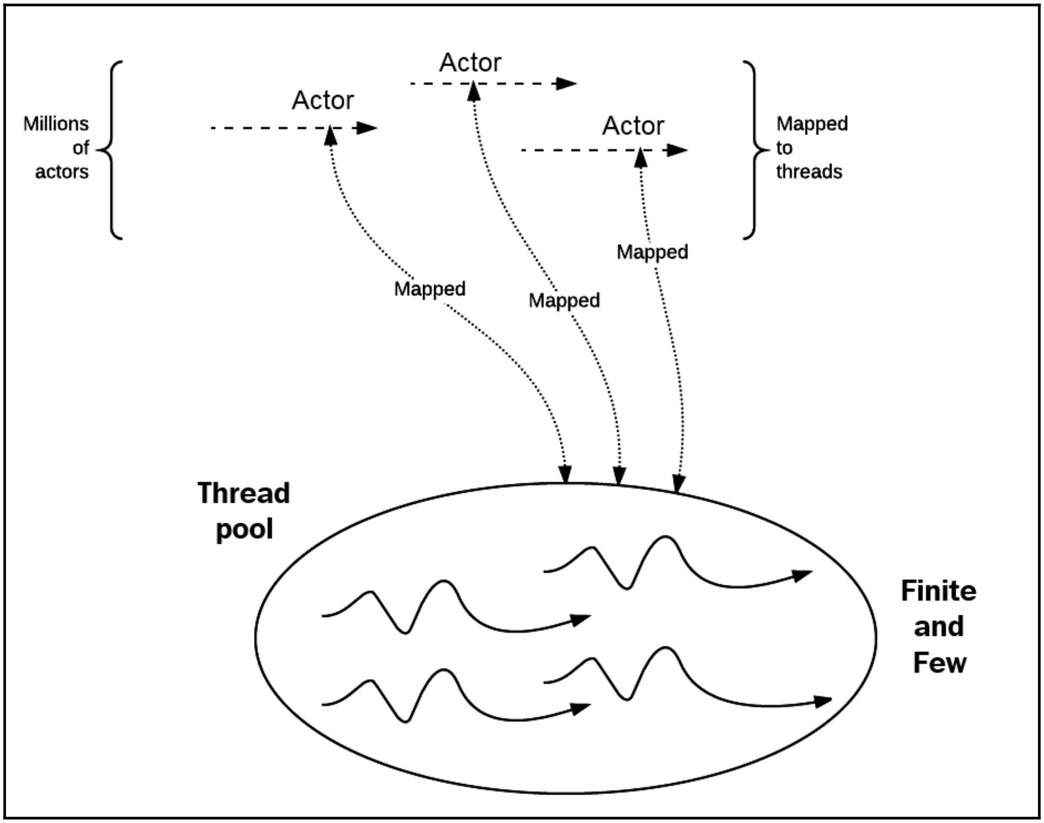
This Scala code shows five standalone methods. Each method is converted to a function and then collected in a variable, `list`. The `reduceRight`call iterates over this list and composes all the functions into a bigger one, *`f`*.

The `f("hello")` call shows that the composition has worked!

The actor paradigm

All this concurrent programming is tricky. What is the correct synchronization and visibility? What if we could go back to our simpler sequential programming model and let the platform handle concurrency for us?

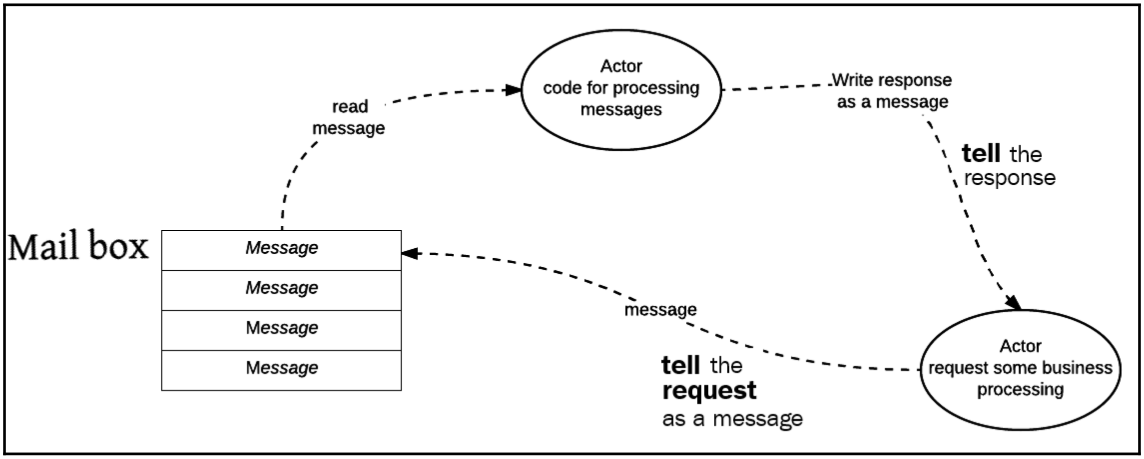
Look at the following diagram:



Actors are the abstraction over threads. We write our code using the message passing model only. The only way of talking to an actor is by sending it a message.

Looking back at our UNIX shell model, the concurrency is there, but we don't deal with it directly. Using actors, we write code as if it were for a sequential message processor.

We need to be aware of the underlying threading model, though. For example, we should always use the *tell* and not the *ask* pattern, as shown in the picture. The *tell* pattern is where we send a message to an actor and then forget about it, that is, we don't block for an answer. This is essentially the *asynchronous* way of doing things:



An actor is a lightweight entity (threads are heavyweight). The creation and destruction of actors, monetarily speaking, is similar to the creation and destruction of Java objects. Just as we don't think of the cost while designing a UNIX pipeline (we focus largely on getting our job done), actors give us the same freedom.

Actors also allow us to add *supervision* and *restart capabilities*, thereby allowing us to write robust, resilient systems. This is the *let it crash* philosophy.

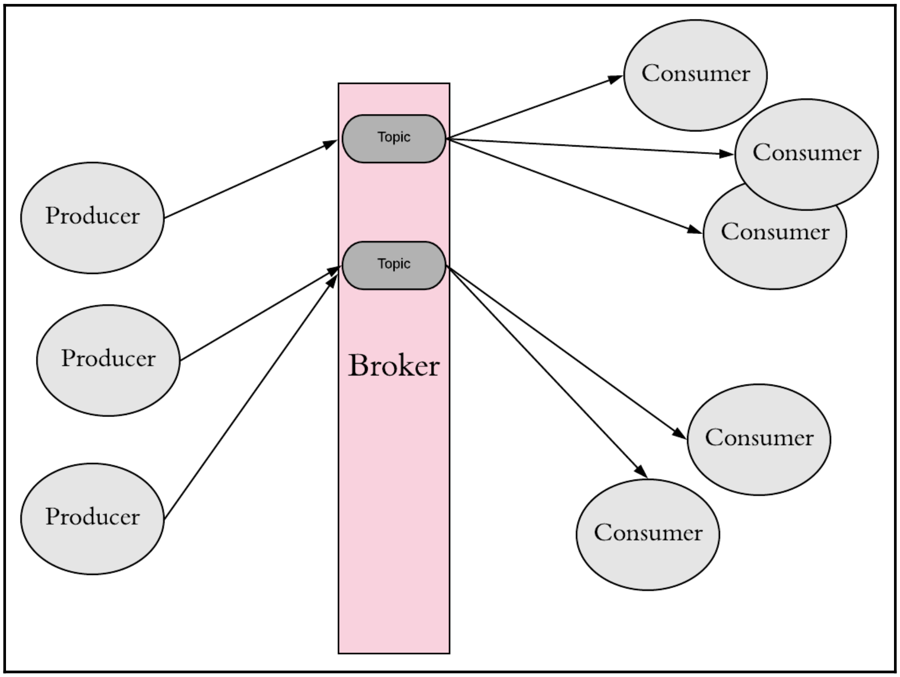
Actors are pretty old as a design; the paradigm was tried and tested in the Telecom domain using the Erlang language.

We will be looking at the actor model and the Akka library in detail in an upcoming chapter.

Message brokers

A *message broker* is an architectural pattern for enabling application integrations via a message-driven paradigm. You can, for example, make a Python application and integrate it with another that is written in C (or Java). Integrations are vital to an enterprise where different applications are made to cooperate with each other.

Concurrent processing is obviously implied here. As the producers and consumers are completely decoupled (they don't even know if the others exist), the producer and consumer applications could even run on different machines, thereby overlapping the processing and increasing the overall throughput:



Decoupling is really a central concept when you start thinking about concurrent systems. Designing systems consisting of loosely coupled component systems gives us many benefits. For example, we could reuse the components, which allows us to cut down on development and maintenance costs. It also paves the way for enabling greater concurrency.

What happens when a consumer produces messages too fast? The messages will be buffered in the broker. This essentially means there is an inherent *flow control* mechanism at work here. A slow consumer can consume at its own pace. Likewise, the producer can produce messages at its own (faster) pace. As both are oblivious of each other, the overall system works smoothly.

Software transactional memory

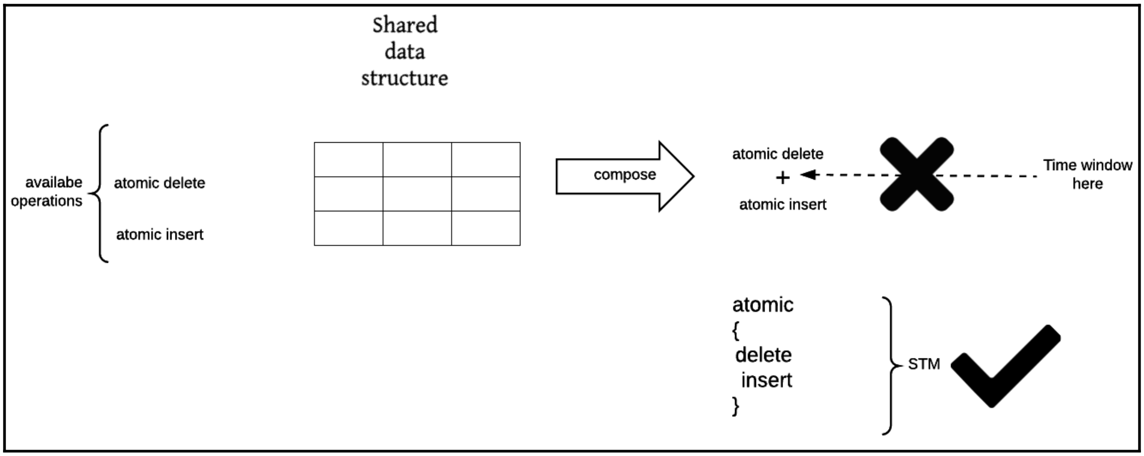
The idea of *database transactions* is also based around concurrent reads and writes. A transaction embodies an *atomic* operation, which means that either *all or none of the steps in the operation* are completed. If all the operations are completed, the transaction succeeds; otherwise, the transaction aborts. The s*oftware transactional memory* is a concurrency control mechanism on similar lines. It, again, is a different paradigm, an alternative to *lock-based synchronization.*

Just like a database transaction, a thread makes modifications and then tries to *commit* the changes. Of course, if some other transaction wins, we *roll back* and retry. If there is an error, the transaction aborts and we retry again.

This scheme of things is called *optimistic locking*, wherein we don't care about other possible concurrent transactions. We just make changes and hope the commit succeeds. If it fails, we keep trying until it eventually succeeds.

What are the benefits? We get increased concurrency, as there is no *explicit locking,* and all threads keep progressing; only in the case of a conflict will they retry.

STM simplifies our understanding of multithreaded programs. This, in turn, helps make programs more maintainable. Each transaction can be expressed as a single-threaded computation, as shown in the following diagram. We don't have to worry about locking at all:



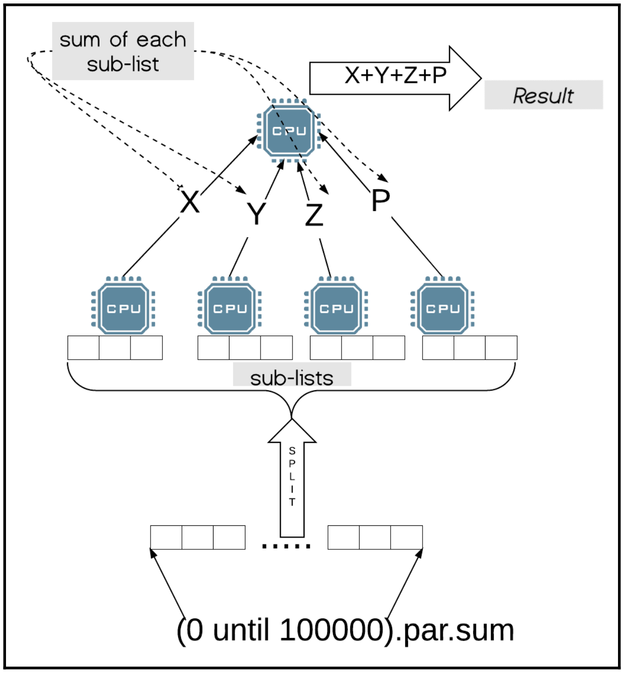
Composability is a big theme: lock-based programs do not compose. You cannot take two atomic operations and create one more atomic operation out of them. You need to specifically program a critical section around these. STM, on the other hand, can wrap these two operations inside a transaction block, as shown in the preceding diagram.

Parallel collections

Say that I am describing some new and exciting algorithm to you. I start telling you about how the algorithm exploits hash tables. We typically think of such data structures as all residing in memory, locked (if required), and worked upon by one thread.

For example, take a list of numbers. Say that we want to sum all these numbers. This operation could be parallelized on multiple cores by using threads.

Now, we need to stay away from explicit locking. An abstraction that works concurrently on our list would be nice. It would split the list, run the function on each sublist, and collate the result in the end, as shown in the following diagram. This is the typical *MapReduce* paradigm in action:



The preceding diagram shows a Scala collection that has been *parallelized* in order to use concurrency internally.

What if the data structure is so large that it cannot all fit in the memory of a single machine? We could split the collection across a cluster of machines instead.

The *Apache Spark* framework does this for us. Spark's ***Resilient Distributed Dataset (RDD)*** is a partitioned collection that spreads the data structure across cluster machines, and thus can work on huge collections, typically to perform analytical processing.

Summary

So, this was a whirlwind tour of the world of concurrency, dear reader. It served more as a memory refresher for many of the things you probably knew already.

We saw that concurrency is very common in the real world, as well as in the software world. We looked at the message passing and shared memory models, and saw how many common themes drive these two models.

If the shared memory model uses explicit locking, a host of problems emerge. We discussed race conditions, deadlocks, critical sections, and heisenbugs.

We wrapped up with a discussion of asynchronicity, the actor paradigm, and the software transactional memory. Now that we have all this background knowledge, in the next chapter, we will look at some core concurrency patterns. Stay tuned!

2
A Taste of Some Concurrency Patterns

In the previous chapter, we touched upon the problem of races. We have race conditions in real life too! The next example is a little hypothetical. People don't miss one another that much, given the technology available these days. However, let's pretend we don't have this technology (I know it's hard to imagine it—let's just pretend for a while though). Let's say it's a Friday and I come home, planning to go for dinner and a movie. I call my wife and daughter, and they reach the movie theater a little ahead of time. However, while driving back home, I get caught in the Friday evening rush and get delayed in traffic. Because I'm running so late, my wife decides to take a stroll and decides to call on a friend working nearby for a cozy chat.

Meanwhile, I arrive at the movie theater, park my car and rush to the movie theater. However, my wife is nowhere to be seen so I decide to check the nearby diner.

By the time I'm in the diner, my wife arrives at the movie theater and doesn't find me there. So, she decides to check the parking lot.

This could go on and on and we could end up missing our Friday movie and dinner in the process. But, thanks to the cell phone, we can easily *synchronize* our movements and meet each other in time for the movie.

In this chapter, we will look at a thread's context. Understanding this concept is central to understanding the way Java's concurrency works. We start by looking at the **singleton design pattern** and the problem of a shared state. However, let's look at some background information first.

For complete code files you can visit `https://github.com/PacktPublishing/Concurrent-Patterns-and-Best-Practices`

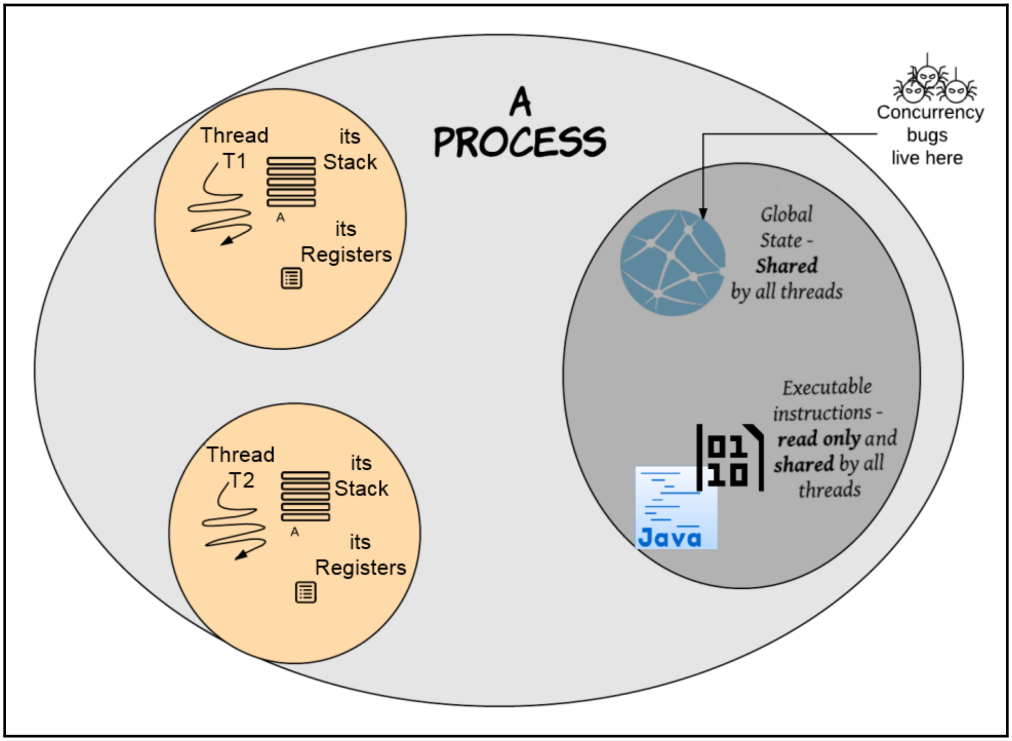
A thread and its context

As we saw in `Chapter 1`, *Concurrency – An Introduction*, a process is a container for threads. A process has executable code and global data; all threads share these things with other threads of the same process. As the following diagram shows, the binary executable code is read-only. It can be freely shared by threads as there is nothing mutable there.

The global data *is* mutable though and, as shown in the diagram, this is the source of concurrency bugs! Most of the techniques and patterns we will study in this book are ways to avoid such bugs.

Threads of the same process run concurrently. How is this achieved, given there is just one set of registers? Well, here's the answer: the *thread context*. This context helps a thread keep its runtime information independent of another thread. The thread context holds the register set and the stack.

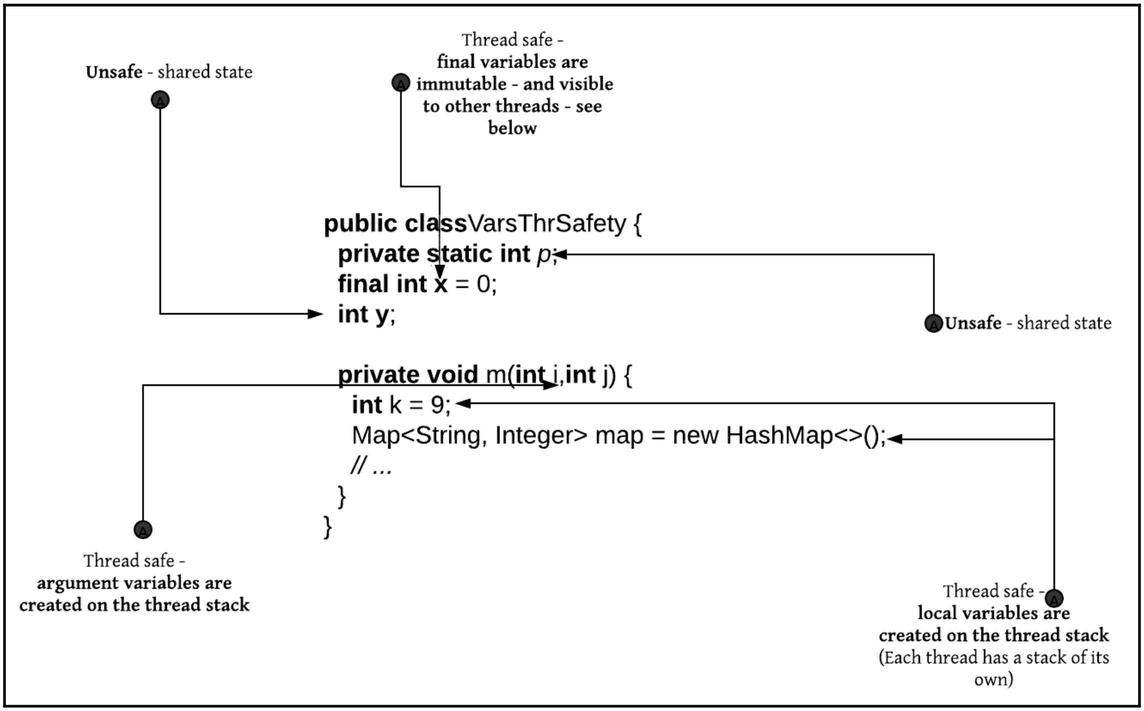
The following diagram shows the various pieces:


A PROCESS
Thread T1
its Stack
its Registers
Thread T2
its Stack
its Registers
Concurrency bugs live here
Global State - Shared by all threads
Executable instructions - read only and shared by all threads
Java
01
10

When the scheduler preempts a running thread, it backs up the thread context—that is to say, it saves the contents of the CPU registers and the stack. Next, it chooses another *runnable* thread and loads its context instead, meaning it restores the thread register's contents as they were the last time (its stack is as it was the last time, and so on) and resumes executing the thread.

What about the executable binary code? Processes running the same code can share the same piece because the code will not change during runtime. (For example, the code for shared libraries is shared across processes.)

Here are some simple rules to understand the thread safety aspect of the state, as represented by various variables:



```
public classVarsThrSafety {
  private static int p;
  final int x = 0;
  int y;

  private void m(int i,int j) {
    int k = 9;
    Map<String, Integer> map = new HashMap<>();
    // ...
  }
}
```

Unsafe - shared state
Thread safe - final variables are immutable - and visible to other threads - see below
Unsafe - shared state
Thread safe - argument variables are created on the thread stack
Thread safe - local variables are created on the thread stack (Each thread has a stack of its own)

As shown, the final and local variables (including function arguments) are always thread safe. You don't need any locks to protect them. The final variables are immutable (that is, read-only) so there is no question of multiple threads changing the value at the same time.

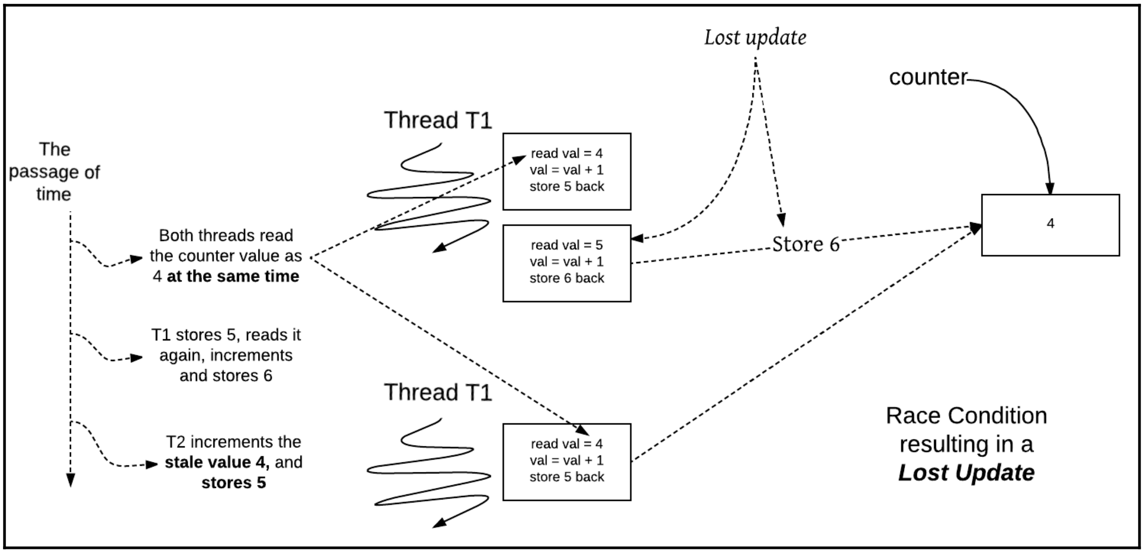
Final Variables also enjoy a special status with respect to *visibility*. We will cover what this means in detail, later. **Mutable Static** and **Instance Variables** are unsafe! If these are not protected, we could easily create **Race Conditions**.

Race conditions

Let's start by looking at some concurrency bugs. Here is a simple example of incrementing a counter:

```
public class Counter {
  private int counter;
  public int incrementAndGet() {
    ++counter;
    return counter;
}
```

This code is not thread-safe. If two threads are running using the same object concurrently, the sequence of *counter* values each gets is essentially *unpredictable.* The reason for this is the `++counter` operation. This simple-looking statement is actually made up of three distinct operations, namely *read* the new value, *modify* the new value, and *store* the new value:



Lost update
counter
Thread T1
The passage of time
read val = 4
val = val + 1
store 5 back
read val = 5
val = val + 1
store 6 back
Store 6
4
Both threads read the counter value as 4 **at the same time**
T1 stores 5, reads it again, increments and stores 6
Thread T1
T2 increments the **stale value 4,** and **stores 5**
read val = 4
val = val + 1
store 5 back
Race Condition resulting in a ***Lost Update***

As shown in the preceding diagram, the thread executions are *oblivious* of each other, and hence interfere unknowingly, thereby introducing a *lost update.*

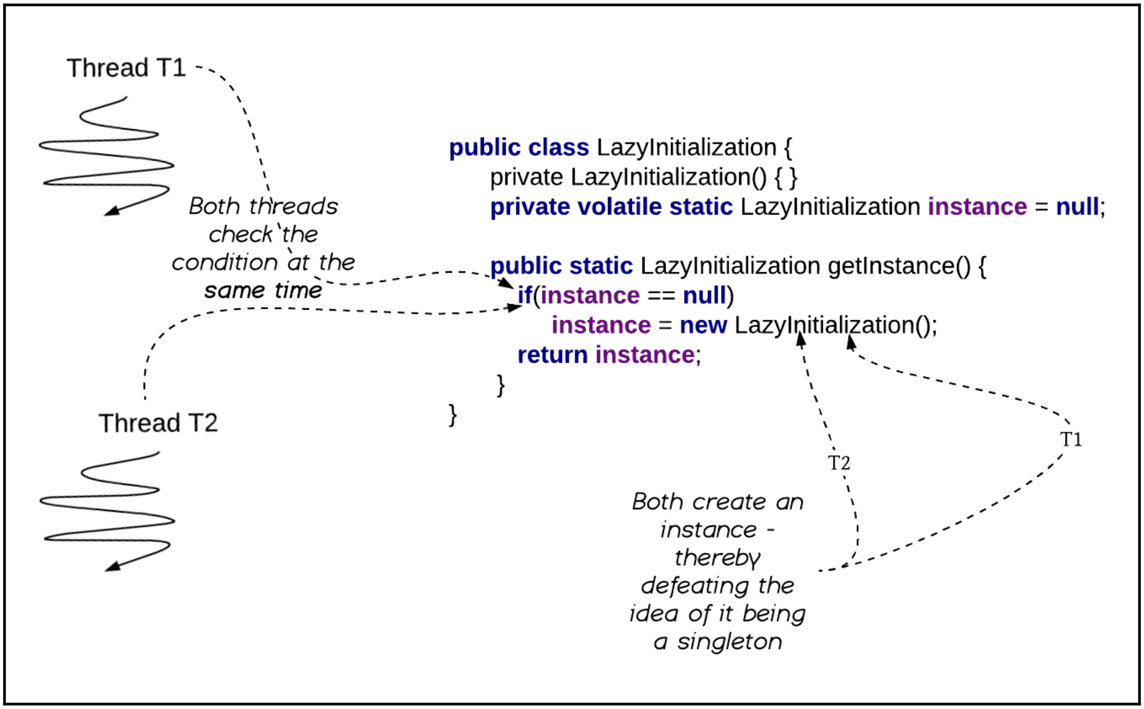
The following code illustrates the *singleton* design pattern. To create an instance of `LazyInitialization` is expensive in terms of time and memory. So, we take over object creation. The idea is to delay the creation till its first use, and then create just one instance and *reuse* it:

```
package chapter02;

public class LazyInitialization {
  private LazyInitialization() { } // force clients to use the factory
method
  // resource expensive members—not shown
  private volatile static LazyInitialization instance = null;

  public static LazyInitialization getInstance() {
    if(instance == null)
      instance = new LazyInitialization();
      return instance;
  }
}
```

When we want to take over instance creation, a common design trick is to make the constructor private, thereby forcing the client code to use our public factory method, `getInstance()`.

Singleton and *factory method* are two of the many *creational* design patterns from the famous **Gang Of Four** (**GOF**) book. These patterns help us force design decisions, for example, in this case making sure we only ever have only a single instance of a class. The need for singletons is pretty common; a classic example of a singleton is a logging service. Thread pools are also expressed as singletons (we will be looking at thread pools in an upcoming chapter). Singletons are used as *sentinel nodes* in tree data structures, to indicate terminal nodes. A tree could have thousands of nodes holding various data items. However, a terminal node does not have any data (by definition), and hence two instances of a terminal node are exactly the same. This property is exploited by making the terminal node a *singleton.* This brings about significant memory savings. While traversing the tree, writing conditionals to check whether we have hit a sentinel node is a snap: you just compare the reference of the sentinel node. Please see `https://sourcemaking.com/design_patterns/null_object` for more information. Scala's None is a `null` object.

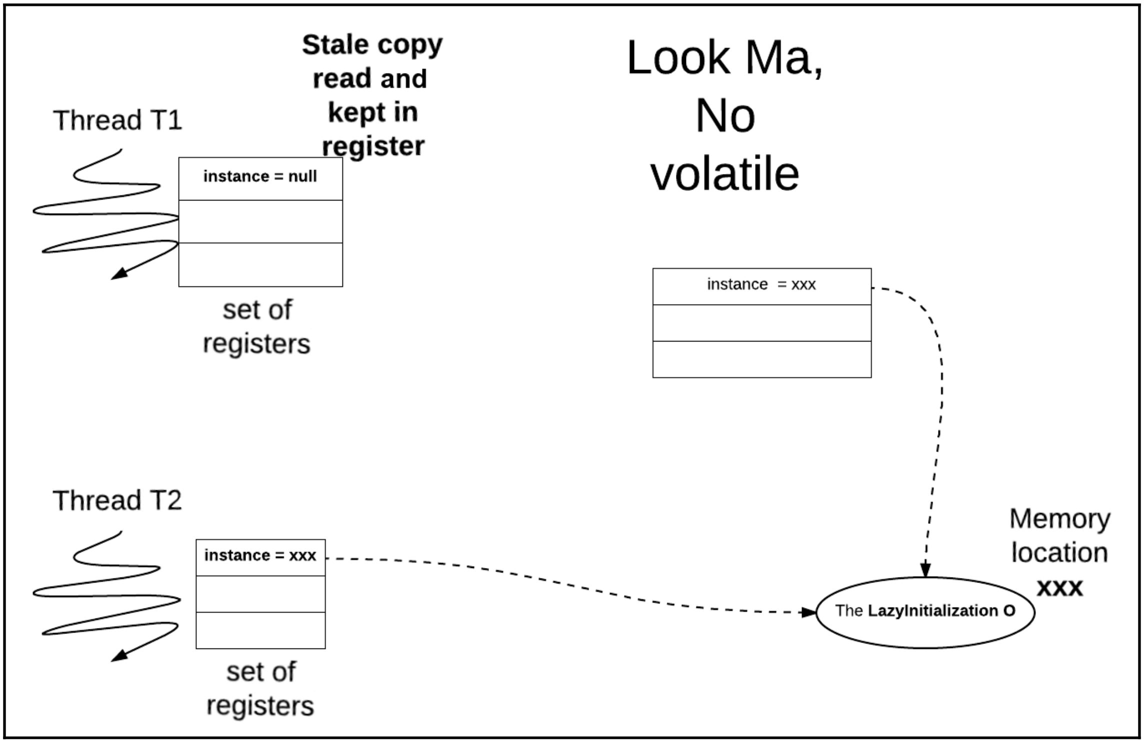
On the first call of the `getter` method, we create and return the new instance. For subsequent calls, the same instance is returned, thereby avoiding expensive construction:



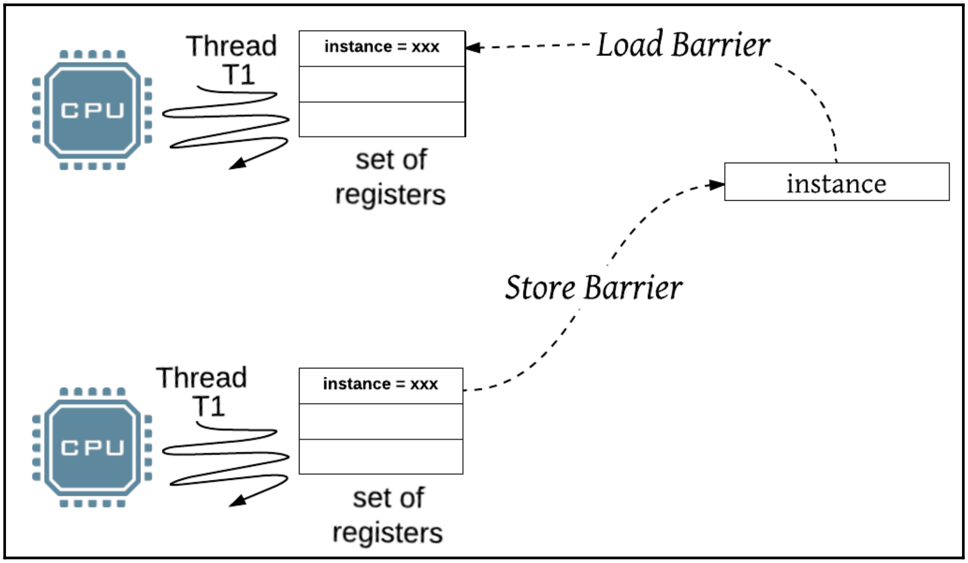
```
public class LazyInitialization {
    private LazyInitialization() { }
    private volatile static LazyInitialization instance = null;

    public static LazyInitialization getInstance() {
      if(instance == null)
         instance = new LazyInitialization();
      return instance;
    }
}
```

Thread T1
Thread T2
Both threads check the condition at the same time
T2
T1
Both create an instance - thereby defeating the idea of it being a singleton

Why do we need to declare the instance variable as a *volatile*? Compilers are allowed to optimize our code. For example, the compiler may choose to store the variable *in a register.* When another thread initializes the *instance* variable, the first thread could have a *stale copy:*



Putting a *volatile* there solves the problem. The instance reference is *always kept up to date* using a **Store Barrier** *after* writing to it, and a **Load Barrier** *before* reading from it. A store barrier makes all CPUs (and threads executing on them) aware of the state change:



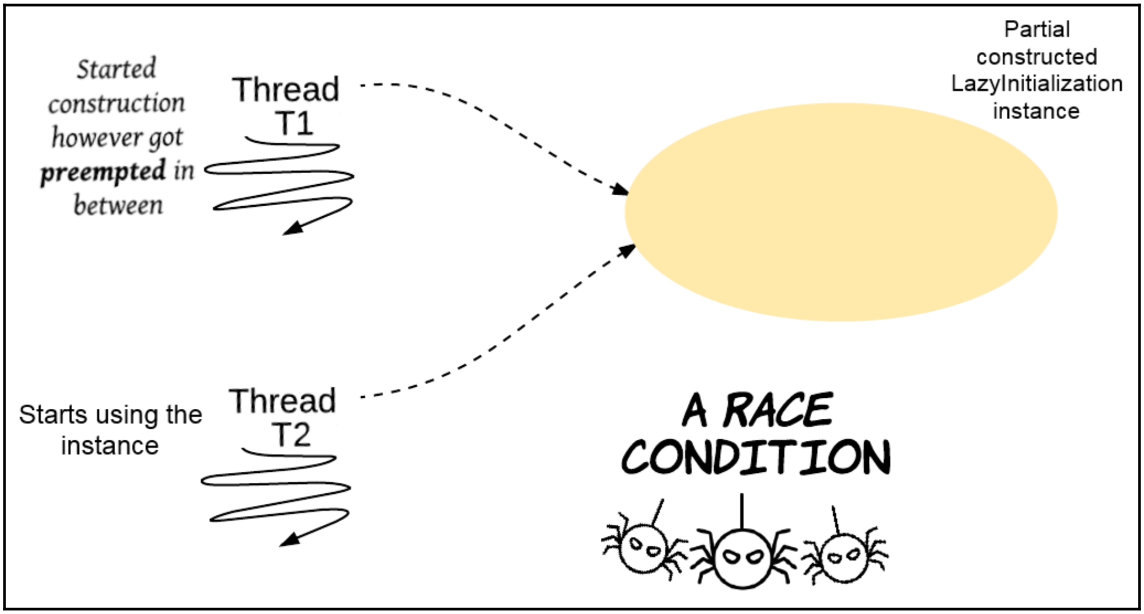
Similarly, a **Load Barrier** makes all CPUs read the latest value, thereby avoiding the stale state issue. See `https://dzone.com/articles/memory-barriersfences` for more information.

The code suffers from a *race condition.* Both threads check the condition, but sometimes, the first thread has not completed initializing the object. (Remember, it is an expensive object to initialize, and that is why we are going through all this rigmarole in the first place.) Meanwhile, the second thread gets scheduled, takes the reference and starts using it—that is to say, it starts using a *partially constructed* instance, which is a buggy state of affairs!

How is this partially constructed instance possible? The JVM can rearrange instructions, so the actual result won't change, but the performance will improve.

When the `LazyInitialization()` expression is being executed, it can first allocate memory, return the reference to the allocated memory location to the *instance variable,* and then start the initialization of the object. As the reference is returned before the constructor has had a chance to execute, it results in an object whose reference is *not null*; however, the constructor is not done yet.

As a result of executing the partially initialized object, mysterious exceptions may result, and they are pretty hard to reproduce! Have a look at the following condition:


Partial constructed LazyInitialization instance
Started construction however got **preempted** in between
Thread T1
Starts using the instance
Thread T2
A RACE CONDITION

Race conditions such as this are essentially *unpredictable.* The times when threads are scheduled depend on external factors. Most of the times, the code will work as expected; however, once in a while, things will go wrong. And, as noted earlier, how can you debug that?

The debugger won't help; we need to make sure the race cannot happen *by design.* Enters the **monitor pattern**.

The monitor pattern

The operation we saw previously of incrementing a counter has the following steps:

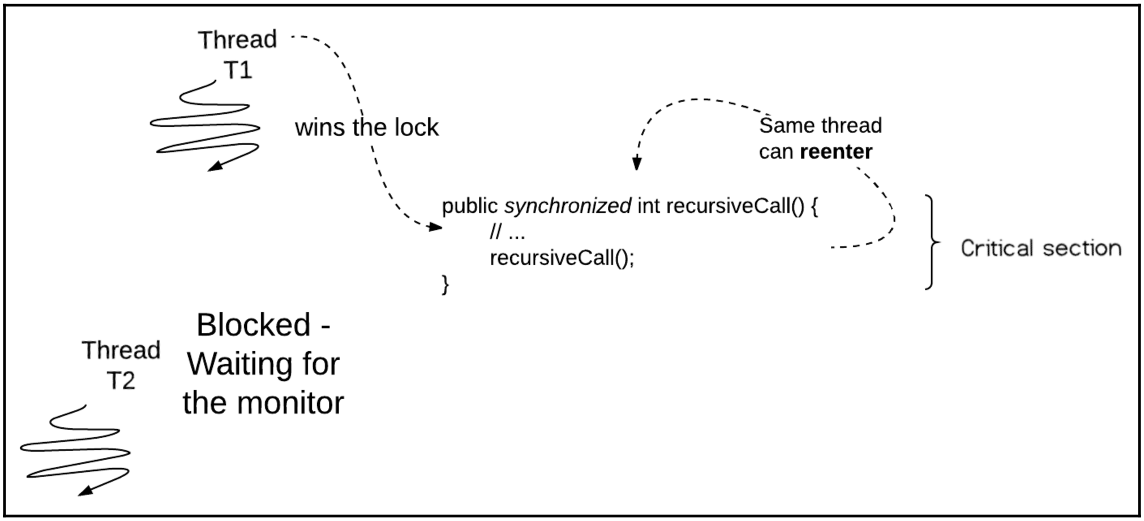
```
/* the counter case */
read existing counter value // should not be stale
increment the value
write it back

/* singleton case */
if (instance == null)
 create the object and return its reference
```

```
else
  return the instance
```

These steps should be *atomic,* that is, *indivisible;* either a thread executes *all* of these operations or *none* of them. The monitor pattern is used for making such a sequence of operations atomic. Java provides a monitor via its `synchronized` keyword:

```
public class Counter {
  private int counter;
  public synchronized int incrementAndGet() {
    ++counter;
    return counter;
  }
}
```

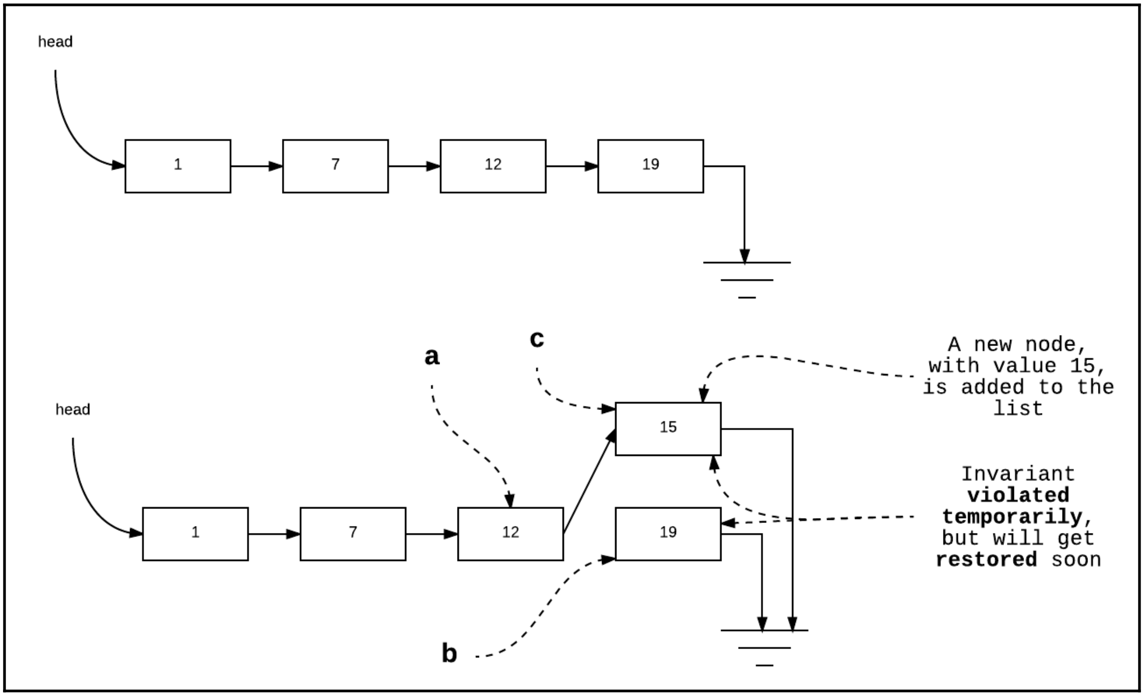
As shown, the counter code is now thread-safe. Every Java object has a *built-in* lock, also known as an *intrinsic lock.* A thread entering the synchronized block acquires this lock. The lock is held till the block executes. When the thread exits the method (either because it completed the execution *or* due to an exception), the lock is *released:*



The `synchronized` blocks are *reentrant:* the same thread holding the lock can *reenter* the block again. If this were not so, as shown in the previous diagram, a *deadlock* would result. The thread itself would not proceed, as it will wait for the lock (held by itself in the first place) to be released. Other threads obviously will be locked out, thereby bringing the system to a halt.

Thread safety, correctness, and invariants

An *invariant* is a good vehicle for knowing the *correctness* of the code. For example, for a singly linked list we could say that *there is at most one non-null node whose next pointer is null:*


head
1
7
12
19
a
c
head
15
1
7
12
19
b
A new node, with value 15, is added to the list
Invariant **violated temporarily**, but will get **restored** soon

In the preceding diagram, the first part shows a singly linked list, with the invariant established. We add a node, with a value of 15, just before the last node, with a value of 19. While the insertion algorithm is in the middle of adjusting the pointer links, the second part shows the state before it has set the *next* pointer of node *c* to *null*.

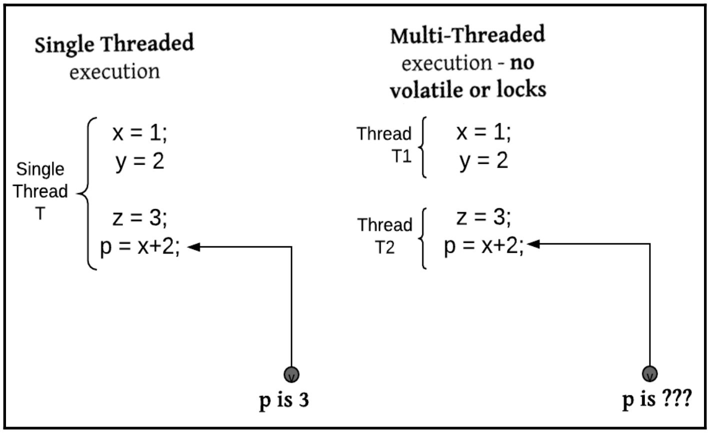
Whether we assume a sequential, single-threaded model or a multi-threaded model, the code invariants should hold.

Explicit synchronization opens up the possibility of exposing the system state with violated invariants. For the linked list example previously shown, we have to synchronize all the states of the data structure, to make sure the invariants hold *all the time.*

Let's say there is a `size()` method counting the list nodes. While the first thread is at the second snapshot (in the middle of inserting the node), if another thread gets access to the second snapshot and calls `size()`, we have a nasty bug. Just once in a while, the `size()` method would return `4`, instead of the expected `5`, and how debuggable is that?

Sequential consistency

Another tool for learning more about concurrent objects is **Sequential Consistency**. Consider the following execution flow:



As shown in the preceding diagram, we read and understand code by assuming the value of **x** is **1** while evaluating the assignment to **p**. We start at the top and work down; it is so intuitive, and obviously correct.

The left-side execution is *sequentially consistent*, as we see the result of earlier steps completed while evaluating the later steps.

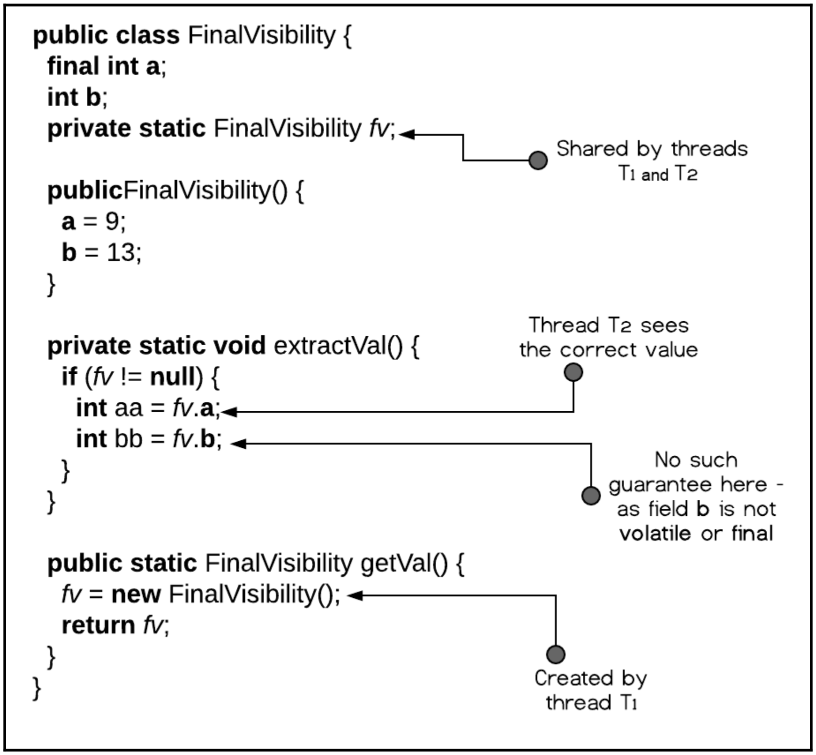
However, the Java memory model does not work quite this way under the hood. Though hidden from us, things are not that linear, as the code is optimized for performance. However, the run time works to make sure our expectations are met; everything's fine in the single-threaded world.

Things are not so rosy when we introduce threads. This is shown on the right-hand side in the previous diagram. There is no guarantee that thread **T2** will read the correct, latest value of the **x** variable , while evaluating **p**.

The locking mechanism (or volatile) guarantees correct visibility semantics.

Visibility and final fields

As we know, final fields are immutable. Once initialized in the constructor, final fields cannot be changed afterward. A final field is also visible to other threads; we don't need any mechanism such as locking or a volatile for this to happen:



```
public class FinalVisibility {
 final int a;
 int b;
 private static FinalVisibility fv;

 publicFinalVisibility() {
  a = 9;
  b = 13;
 }

 private static void extractVal() {
  if (fv != null) {
   int aa = fv.a;
   int bb = fv.b;
  }
 }

 public static FinalVisibility getVal() {
  fv = new FinalVisibility();
  return fv;
 }
}
```

Shared by threads T1 and T2

Thread T2 sees the correct value

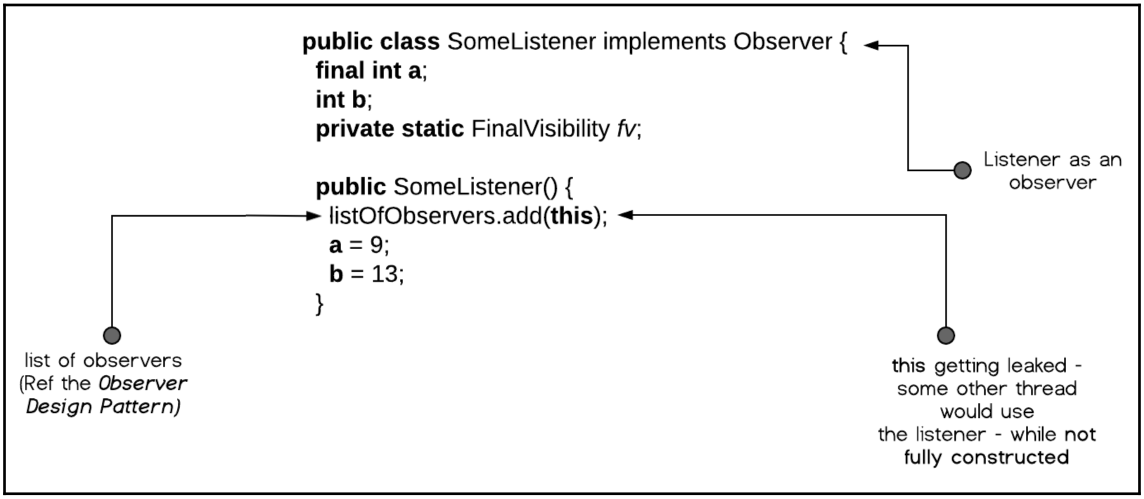
No such guarantee here - as field **b** is not volatile or final

Created by thread T1

As shown in the preceding diagram, both threads share the `fv` static field. The `a` field is declared final and is initialized to a value of `9` in the constructor. In the `extractVal()` method, the correct value of *a* is visible to other threads.

The `b`field, however, enjoys no such guarantees. As it is declared neither final not volatile—and there being no locking present—we cannot say anything definite regarding the value of `b`, as seen by other threads.

There is one catch though, final fields *should not leak out* of the constructor:



```
public class SomeListener implements Observer {
  final int a;
  int b;
  private static FinalVisibility fv;

  public SomeListener() {
    listOfObservers.add(this);
    a = 9;
    b = 13;
  }
```

Listener as an observer
list of observers (Ref the *Observer Design Pattern)*
this getting leaked - some other thread would use the listener - while not fully constructed

As shown, before the constructor execution completes the *this* reference is leaked out to the constructor of `someOtherServiceObj`. There could be another thread concurrently using `someOtherServiceObj`. It uses this object and, indirectly, the `FinalVisibility` class instance.

As the constructor of `FinalVisibility` is not done yet, the value of the final field, *a,* is not visible to this other thread, thereby introducing a heisenbug.

See `http://www.javapractices.com/topic/TopicAction.do?Id=252` for more information and a discussion on leaking references out of constructors.

Double-checked locking

Using intrinsic locking, we can write a thread-safe version of the singleton.

Here it is:

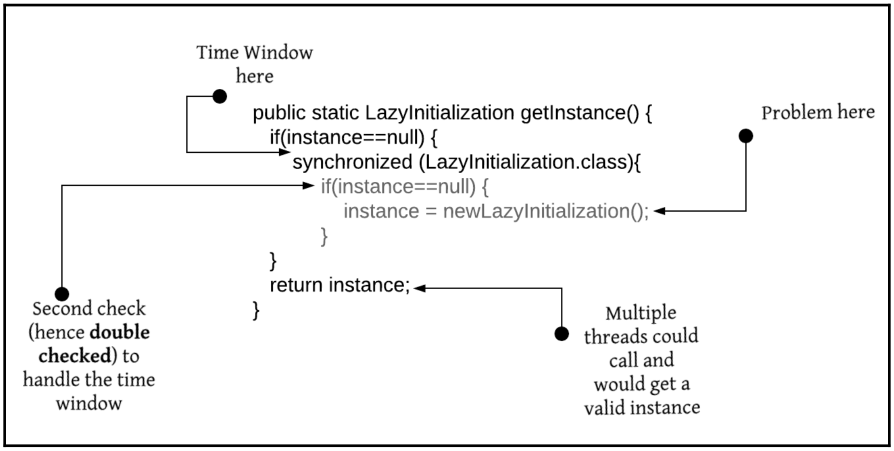
```
public synchronized static LazyInitialization getInstance() {
  if(instance == null)
    instance = new LazyInitialization();
  return instance;
}
```

As the method is synchronized, only one thread can execute it at any time. If multiple threads are calling `getInstance()` multiple times, the method could quickly become a *bottleneck*. Other threads competing for the access to method will *block waiting for the lock,* and won't be able to do anything productive in the meantime. The *liveness* of the system will suffer; this bottleneck could adversely affect the *concurrency* of the system.

This prompted the development of the *double-checked locking* pattern, as illustrated by the following code snippet:

```
public class LazyInitialization {
  private LazyInitialization() {
    } // force clients to use the factory method
    // resource expensive members—not shown

  private volatile static LazyInitialization instance = null;

  public static LazyInitialization getInstance() {
    if (instance == null) {
      synchronized (LazyInitialization.class) {
      if (instance == null) {
        instance = new LazyInitialization();
      }
  }
    return instance;
  }
}
```

This is smart thinking: the locking ensures the safe creation of the actual instance. Other threads will either get a `null` or the *updated instance* value—due to the `volatile` keyword. You can see as follows, the code is still broken:



```
public static LazyInitialization getInstance() {
  if(instance==null) {
    synchronized (LazyInitialization.class){
      if(instance==null) {
        instance = newLazyInitialization();
      }
  }
  return instance;
}
```

Time Window here
Problem here
Second check (hence **double checked**) to handle the time window
Multiple threads could call and would get a valid instance

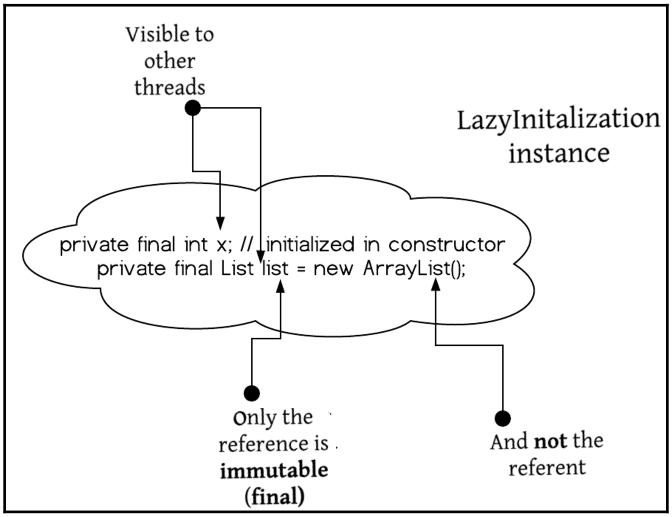
Pause a moment to study the code. There is a time window after the first check, where a context switch can happen. Another thread gets a chance and enters the `synchronized` block. We are synchronizing on a lock variable here: *the class lock*. The second check is synchronized and is executed by only one thread; let's say it's `null`. The thread owning the lock then goes ahead and creates the instance.

Once it exits the block and carries on with its execution, the other threads gain access to the lock one by one. They purportedly find the *instance is fully constructed,* so they use the fully constructed object. What we are trying to do is *safely publish* the shared *instance* variable.

Safe publication

When one thread creates a shared object (as in this case), other threads will want to use it too. The term here is that the creator thread publishes the object *as ready for use other threads.*

The problem is that just making the `instance` variable volatile does not guarantee that other threads get to see a *fully constructed* object. The volatile works for the *instance reference* publication itself, but not for the *referent* object (in this case the `LazyInitialization` object) if it contains *mutable members.* We could get partially initialized variables in this case:


Visible to other threads
LazyInitalization instance
private final int x; // initialized in constructor
private final List list = new ArrayList();
Only the reference is **immutable** (final)
And **not** the referent

When the `LazyInitialization` constructor exits, all *final fields* are *guaranteed to be visible* to other threads accessing them. See `https://www.javamex.com/tutorials/synchronization_final.shtml` for more on the relationship between the `final` keyword and safe publication.

Using the volatile keyword *does not guarantee the safe publication* of mutable objects. You can find a discussion regarding this here: `https://wiki.sei.cmu.edu/confluence/display/java/CON50J.+Do+not+assume+that+declaring+a+reference+volatile+guarantees+safe+publication+of+the+members+of+the+referenced+object.`

Next comes a design pattern that simplifies the lazy creation of our instance, without needing all this complexity.

Initializing a demand holder pattern

So, we seem to be in a fix. On one hand, we don't want to pay for needless synchronization. On the other hand, the double-checked locking is broken, possibly publishing a partially constructed object.

The following code snippet shows the *lazy loaded singleton.* The resulting code is simple and does not depend on subtle synchronization semantics. Instead, it exploits the class loading semantics of the JVM:

```
public class LazyInitialization {
  private LazyInitialization() {
  }
  // resource expensive members—not shown

  private static class LazyInitializationHolder {
    private static final LazyInitialization INSTANCE = new
LazyInitialization();
  }

    public static LazyInitialization getInstance() {
        return LazyInitializationHolder.INSTANCE;
    }
}
```

The `getInstance()` method uses a static class, `LazyInitializationHolder`. When the `getInstance()` method is first invoked, the static class is loaded by the JVM.

Now the **Java Language Specification** (**JLS**) ensures that the class initialization phase is sequential. All subsequent concurrent executions will return the same and correctly initialized instance, without needing *any synchronization.*

The pattern exploits this feature to completely avoid any locks and still achieve correct lazy initialization semantics!

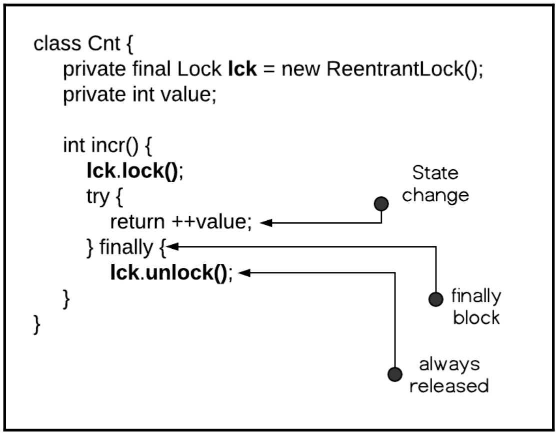
Singletons are often rightly criticized as representing the *global state.* However, as we saw, their functionality is needed at times, and the pattern is a nice, reusable solution, that is to say, a concurrency design pattern.

You can also use an enum for creating singletons; please see `https://dzone.com/articles/java-singletons-using-enum` for more on this design technique.

Explicit locking

The `synchronized` keyword is an *intrinsic* locking mechanism. It is pretty convenient, there are a couple of limitations as well. We cannot interrupt a thread waiting for an intrinsic lock, for example. There is also no way to time out the wait while *acquiring* the lock.

There are use cases where these capabilities are needed; at such times, we use *explicit locking*. The *Lock* interface allows us to overcome these limitations:



```
class Cnt {
    private final Lock lck = new ReentrantLock();
    private int value;

    int incr() {
       lck.lock();
       try {
          return ++value;
       } finally {
          lck.unlock();
    }
}
```

`ReentrantLock` duplicates the functionality of the synchronized keyword. A thread already holding it can acquire it again, just like with synchronized semantics. Memory visibility and mutual exclusion guarantees are the same in both cases.

Additionally, `ReentrantLock` gives us a *non-blocking* `tryLock()` and *interruptible* locking.

Using `ReentrantLock` puts the onus on us; we *need* to make sure the lock is released via all return paths; even in the case of exceptions.

Such explicit locking lets us control *lock granularity;* here, we can see an example of a concurrent set data structure, implemented using a sorted linked list:

```
public class ConcurrentSet {
```

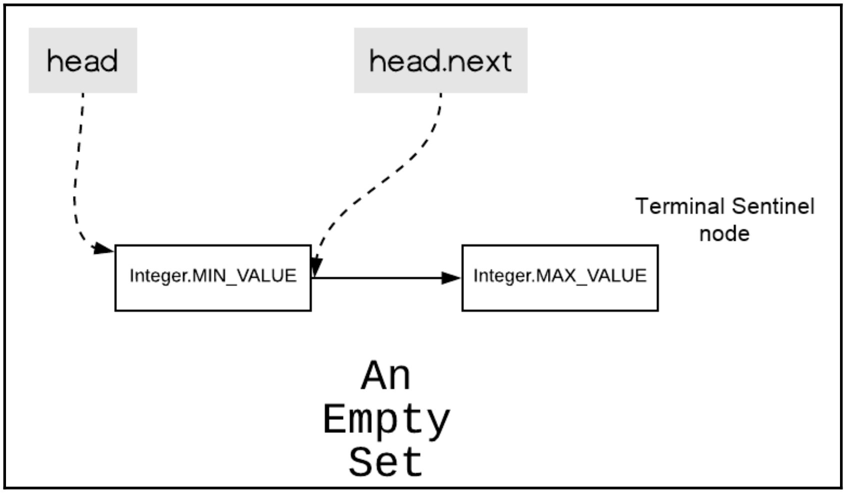
Where, the concurrent set holds a linked list of `Node` nodes; the definition of it is as follows:

```
private class Node {
  int item;
  Node next;

  public Node(int i) {
    this.item = i;
    }
}
```

The `Node` class represents a node of the linked list. Here is the default constructor:

```
public ConcurrentSet() {
  head = new Node(Integer.MIN_VALUE);
  head.next = new Node(Integer.MAX_VALUE);
}
```

The following diagram shows the state after constructor execution is complete:



As shown, the default constructor initializes an empty set, a linked list of two *node*s. The head node always holds the minimum value (`Integer.MIN_VALUE`), and the last node contains the maximum (`Integer.MAX_VALUE`). Using such sentinel nodes is a common algorithm design technique, which simplifies the rest of the code, as we will soon see:

```
private Node head;
private Lock lck = new ReentrantLock();
```

`ConcurrentSet` also has a field named `lck`, which is initialized to `ReentrantLock`. Here is our `add` method:

```
private boolean add(int i) {
  Node prev = null;
  Node curr = head;

  lck.lock();

  try {
    while (curr.item < i) {
      prev = curr;
      curr = curr.next;
      }
      if (curr.item == i) {
        return false;
       } else {
        Node node = new Node(i);
        node.next = curr;
        prev.next = node;
        return true;
       }
    } finally {
        lck.unlock();
    }
}
```

The `add(int)` method starts by acquiring the lock. As the list is a set, all elements are unique and the elements are stored in ascending order.

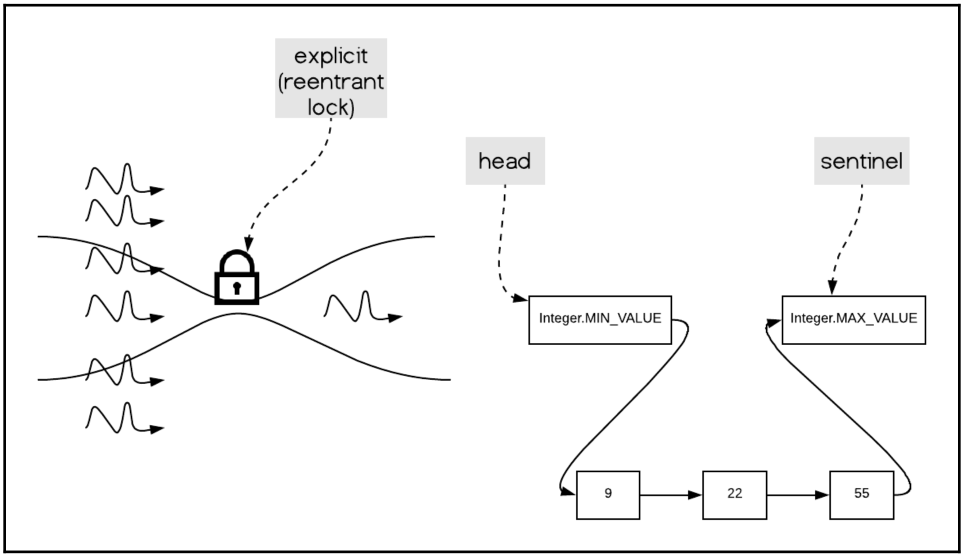
Next is the `lookUp(int)` method:

```
private boolean lookUp(int i) {
  Node prev = null;
  Node curr = head;

  lck.lock();

  try {
    while (curr.item < i) {
    prev = curr;
    curr = curr.next;
      }
      if (curr.item == i) {
        return true;
       }
      return false;
    } finally {
        lck.unlock();
    }
}
```

The `lookUp(int)` method searches our set, and if it finds the argument element, it returns `true`; otherwise, it returns `false`. Finally, here is the `remove(int)` method. It juggles the next pointers so the node containing the element is removed:

```
private boolean remove(int i) {
  Node prev = null;
  Node curr = head;

  lck.lock();

  try {
    while (curr.item < i) {
      prev = curr;
      curr = curr.next;
      }
      if (curr.item == i) {
        prev.next = curr.next;
        return true;
       } else {
         return false;
        }
    } finally {
        lck.unlock();
    }
}
```

The problem is we are using coarse-grained synchronization: we are holding a global lock. If the set holds a big number of elements, *only one thread at a time* can be doing either an *add, remove, or lookup.* The execution is essentially *sequential:*



The synchronization is *obviously correct.* The code is easier to understand! However, due to its being *coarse-grained,* if many threads are contending for the lock, they end up waiting for it. The time that could be spent doing productive work is instead spent on waiting! The lock is a bottleneck.

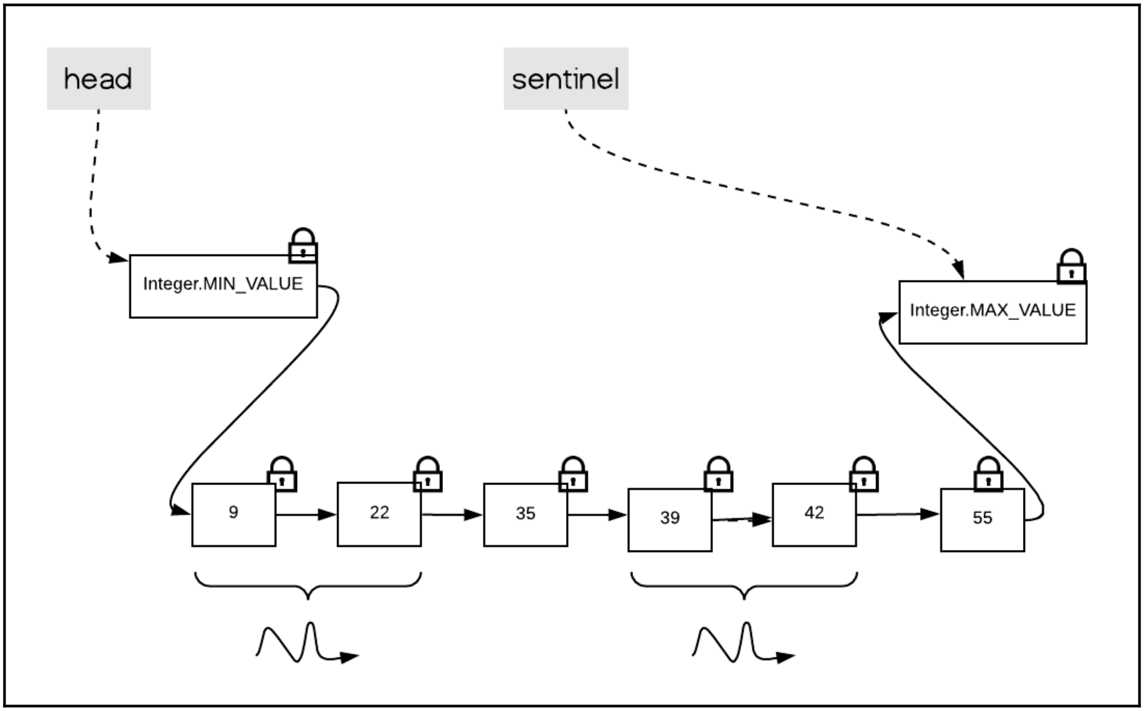
The hand-over-hand pattern

The coarse-grained synchronization explained in the previous section, hurts concurrency. Instead of locking the list as a whole, we could improve things by *locking both the previous and the current node*. If a thread follows this while traversing the list, thereby doing a *hand-over-hand locking,* it allows other threads to concurrently work on the list as shown here:

```
private class Node {
 int item;
 Node next;
 private Lock lck = new ReentrantLock();

 private Node(int i) {
   this.item = i;
 }
```

```
    private void lock() {
      lck.lock();
    }

    private void unlock() {
      lck.unlock();
        }
  }
```

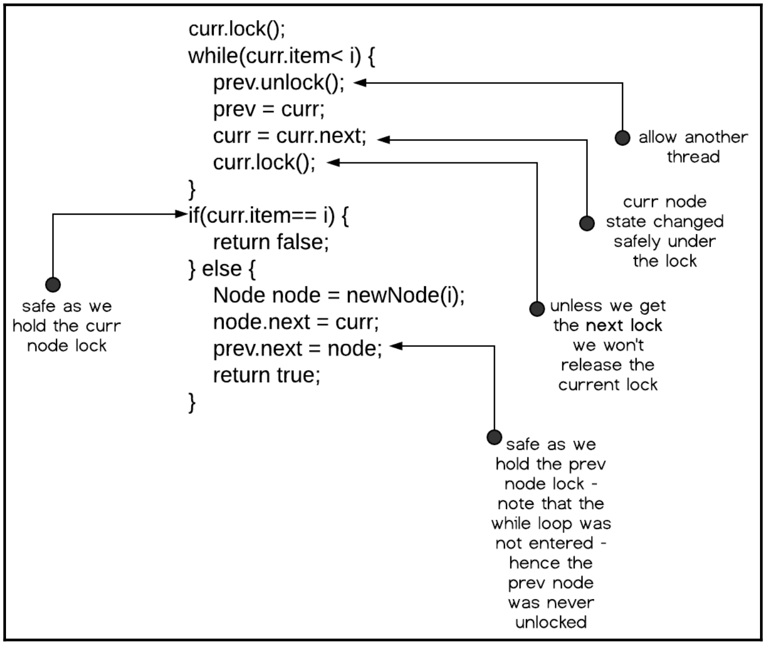
Note that we are talking about *lock a node*—this entails removing our global lock—and instead creating a lock field in the node itself. We provide two primitives, `lock()` and `unlock()`, for readable code:



The add(int) method, rewritten to use this pattern, is shown as follows:

```
private boolean add(int i) {
  head.lock();
  Node prev = head;

  try {
    Node curr = prev.next;

    curr.lock();

    try {
      while (curr.item < i) {
        prev.unlock();
        prev = curr;
        curr = curr.next;
        curr.lock();
        }
      if (curr.item == i) {
        return false;
      } else {
          Node node = new Node(i);
          node.next = curr;
          prev.next = node;
          return true;
        }
       } finally {
           curr.unlock();
        }
    } finally {
        prev.unlock();
    }
}
```

As before, we need to protect the locking with a try or finally. So, in the case of an exception, releasing the lock is guaranteed:



```
curr.lock();
while(curr.item< i) {
    prev.unlock();
    prev = curr;
    curr = curr.next;
    curr.lock();
}
if(curr.item== i) {
    return false;
} else {
    Node node = newNode(i);
    node.next = curr;
    prev.next = node;
    return true;
}
```

allow another thread
curr node state changed safely under the lock
unless we get the **next lock** we won't release the current lock
safe as we hold the curr node lock
safe as we hold the prev node lock - note that the while loop was not entered - hence the prev node was never unlocked

The previous code snippet explains the various concurrent scenarios. Here is the `remove(int)` method:

```
private boolean remove(int i) {
  head.lock();
  Node prev = head;

  try {
    Node curr = prev.next;

    curr.lock();

    try {
      while (curr.item < i) {
        prev.unlock();
        prev = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.item == i) {
        prev.next = curr.next;
        return true;
       }
        return false;
        } finally {
          curr.unlock();
        }
    } finally {
        prev.unlock();
    }
}
```

The `remove(int)` method works on the same lines. The code balances the trade off—it unlocks as soon as possible but makes sure it holds both the `prev` and `curr` node locks—to eliminate any possibility of a race condition:

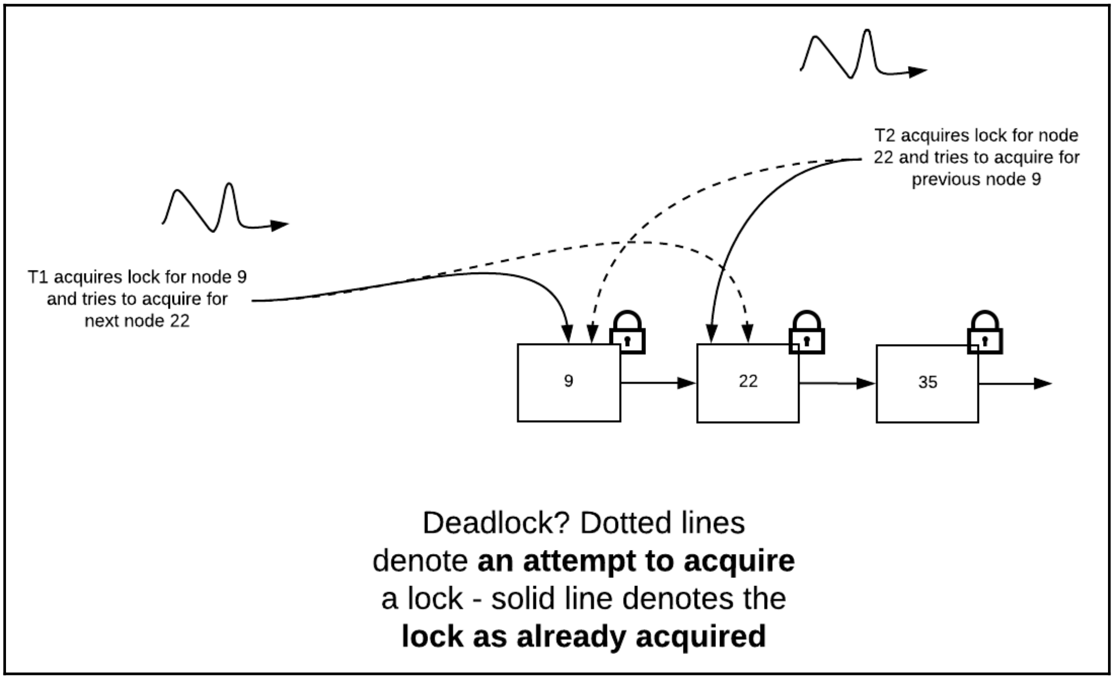
```
public static void main(String[] args) {
  FGConcurrentSet list = new FGConcurrentSet();

  list.add(9);
  list.add(1);
  list.add(1);
  list.add(9);
  list.add(12);
  list.add(12);

  System.out.println(list.lookUp(12));
  list.remove(12);
  System.out.println(list.lookUp(12));
  System.out.println(list.lookUp(9));
}
// prints true, false, true
```

This code is a test driver; note that it is single threaded. Writing a multi-threaded driver, by spawning off two or more threads sharing the concurrent set, will help you understand the code better. Writing the `lookUp(int)` method is similar to the `add` and `remove` method; both are left as exercises for the reader.

Observations – is it correct?

Why does this code and the hand-over-hand pattern work? Here is some reasoning that helps us establish confidence about the code. For example, while managing multiple locks, avoiding deadlocks is a challenge. How does the previous code help us avoid deadlocks?

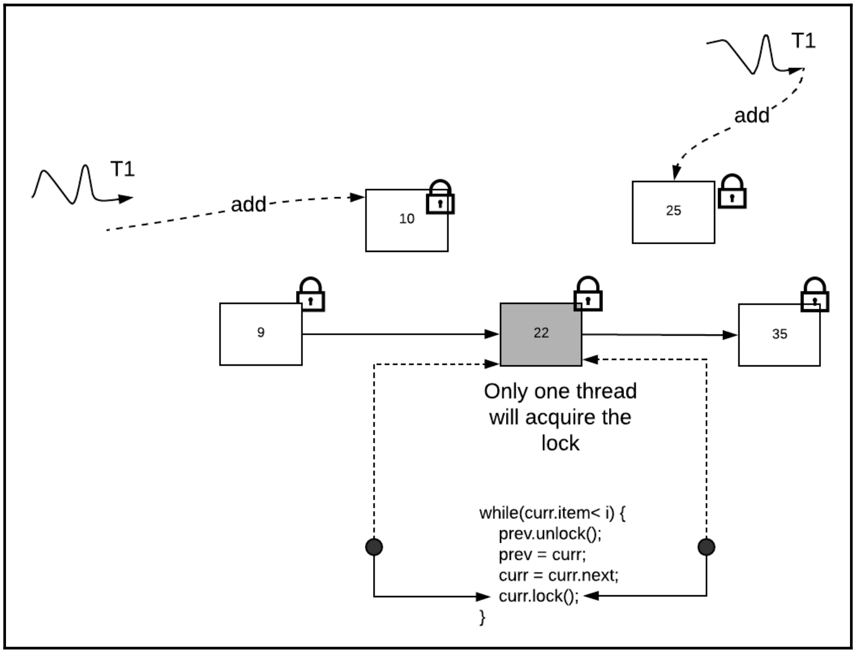
Let's say thread **T1** calls the `add()` method, and at the same time, thread **T2** calls `remove()`. Could the situation shown in the following diagram arise?


T2 acquires lock for node 22 and tries to acquire for previous node 9
T1 acquires lock for node 9 and tries to acquire for next node 22
9
22
35

Deadlock? Dotted lines denote **an attempt to acquire** a lock - solid line denotes the **lock as already acquired**

This code guarantees that a *deadlock situation is impossible*! We make sure the locks are *always acquired* in order, beginning from the *head* node. Hence, the locking order shown can't possibly happen.

What about two concurrent `add(int)` calls? Let's say the set holds {9, 22, 35} and **T1** adds **10**. At the same time **T2** adds **25**:



As shown, there is always a common node—and hence a common lock—that needs to be acquired by two (or more) threads, as by definition only one thread can win, forcing the other thread(s) to wait.

It's hard to see how we could have used Java's intrinsic locking (the *synchronized* keyword) to implementing the *hand-over-hand* pattern. Explicit locking gives us more control and allows us to implement the pattern easily.

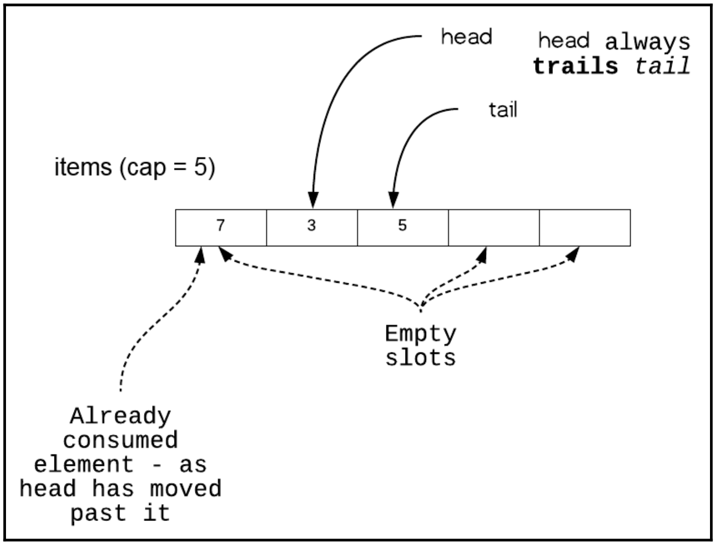
The producer/consumer pattern

In the previous chapter, we saw that threads need to cooperate with one another to achieve significant functionality. Cooperation entails communication; `ReentrantLock` allows a thread to signal to other threads. We use this mechanism to implement a `Concurrent` FIFO queue:

```
public class ConcurrentQueue {
  final Lock lck = new ReentrantLock();
  final Condition needSpace = lck.newCondition();
```

```
      final Condition needElem = lck.newCondition();
      final int[] items;
      int tail, head, count;

      public ConcurrentQueue(int cap) {
        this.items = new int[cap];
      }
```

The class holds a reentrant lock in its `lck` field. It also has two *conditions,* namely `needSpace` and `needElem`. *W*e will see how these are used. The queue elements are stored in an array named *items:*


head
head always **trails** *tail*
tail
items (cap = 5)
7
3
5
Empty slots
Already consumed element - as head has moved past it

The `head` points to the next element to be consumed. Similarly, the `tail` points to an empty slot where we store a new element. The constructor allocates an array of capacity named `cap`*:*

```
    public void push(int elem) throws InterruptedException {
      lck.lock();

      try {
        while (count == items.length)
        needSpace.await();
        items[tail] = elem;
        ++tail;
        if (tail == items.length)
          tail = 0;
          ++count;
          needElem.signal();
        } finally {
          lck.unlock();
        }
    }
```

There is some subtlety here. Let's first understand the simple stuff. A *producer* thread tries to push items into the queue. It starts off by *acquiring* the `lck` lock. The rest of the method code executes under this lock. The `tail` variable holds the index of the next slot where we could store the new number. The following code *pushes* a new element to the queue:

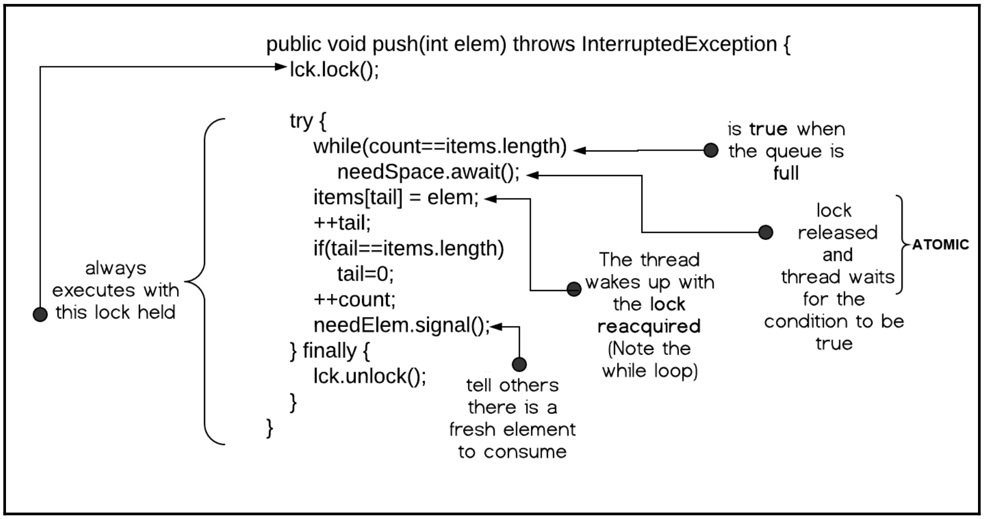
```
items[tail] = elem;
++tail;
```

If we have used up all the array slots, `tail` wraps back to 0:

```
if (tail == items.length)
   tail = 0;
++count;
```

The `count` variable holds the current count of elements available for consumption. As we have produced one more element, `count` is incremented.

Next, let's look at the concurrency aspect, shown in the following diagram:



```
public void push(int elem) throws InterruptedException {
  lck.lock();

  try {
    while(count==items.length)
      needSpace.await();
    items[tail] = elem;
    ++tail;
    if(tail==items.length)
      tail=0;
    ++count;
    needElem.signal();
  } finally {
    lck.unlock();
  }
}
```

always executes with this lock held
is **true** when the queue is **full**
lock released and thread waits for the condition to be true
ATOMIC
The thread wakes up with the **lock reacquired** (Note the while loop)
tell others there is a fresh element to consume

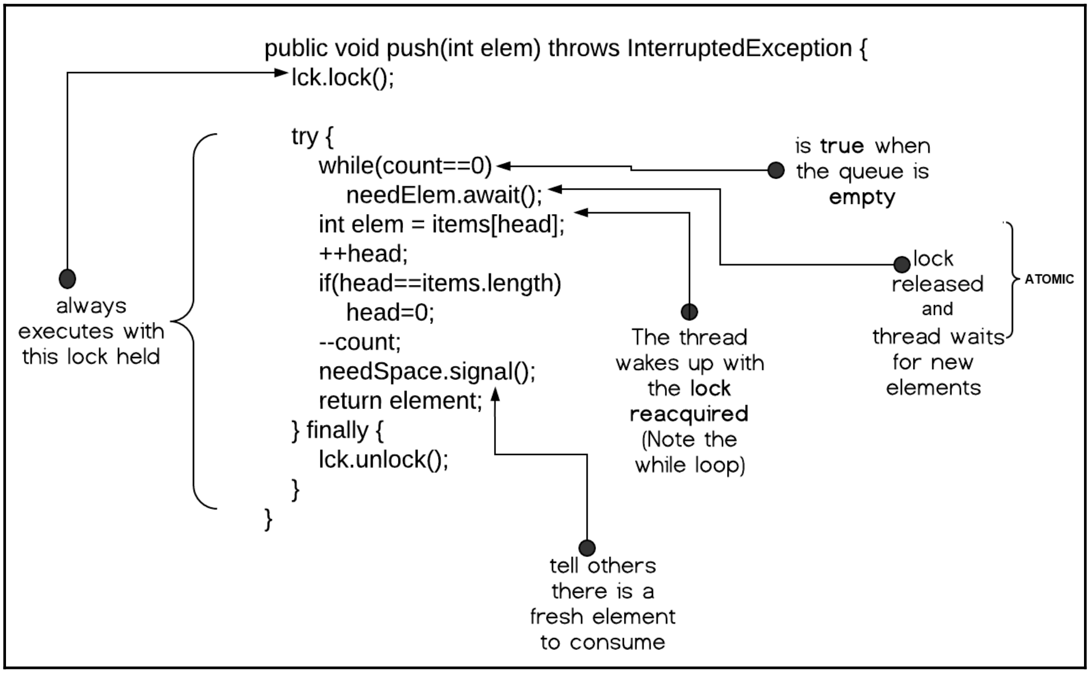
As the *items* array, has a finite capacity (it can hold at most *cap* elements), we need to deal with the possibility that the *queue is full*. The producer needs to wait for a consumer to pick one or more elements from the queue.

This waiting is accomplished by calling `await()` on the `needSpace` condition variable. It is important to realize that the *thread is made to wait* and that the *lock*— `lck`— *is released;* these two are *atomic* operations.

Let's say that someone has consumed one or more items from the queue (we will soon see how in the `pop()` method). When this happens, the producer thread wakes up *with the lock acquired.* Lock acquisition is imperative for the rest of the code to work correctly:

```
public int pop() throws InterruptedException {
  lck.lock();

  try {
    while (count == 0)
      needElem.await();
      int elem = items[head];
      ++head;
      if (head == items.length)
        head = 0;
        --count;
        needSpace.signal();
     return elem;
 } finally {
     lck.unlock();
   }
}
```

The `pop` method works along similar lines. Except for the popping logic, it is a mirror image of the `push` method:



```
public void push(int elem) throws InterruptedException {
  lck.lock();

  try {
    while(count==0)
      needElem.await();
    int elem = items[head];
    ++head;
    if(head==items.length)
      head=0;
    --count;
    needSpace.signal();
    return element;
  } finally {
    lck.unlock();
  }
}
```

always executes with this lock held
is **true** when the queue is **empty**
lock released and thread waits for new elements
ATOMIC
The thread wakes up with the **lock reacquired** (Note the while loop)
tell others there is a fresh element to consume

The consumer thread pops an element off the queue using the following lines:

```
int elem = items[head];
++head;
```

The `head` is then moved to the next available element (if any):

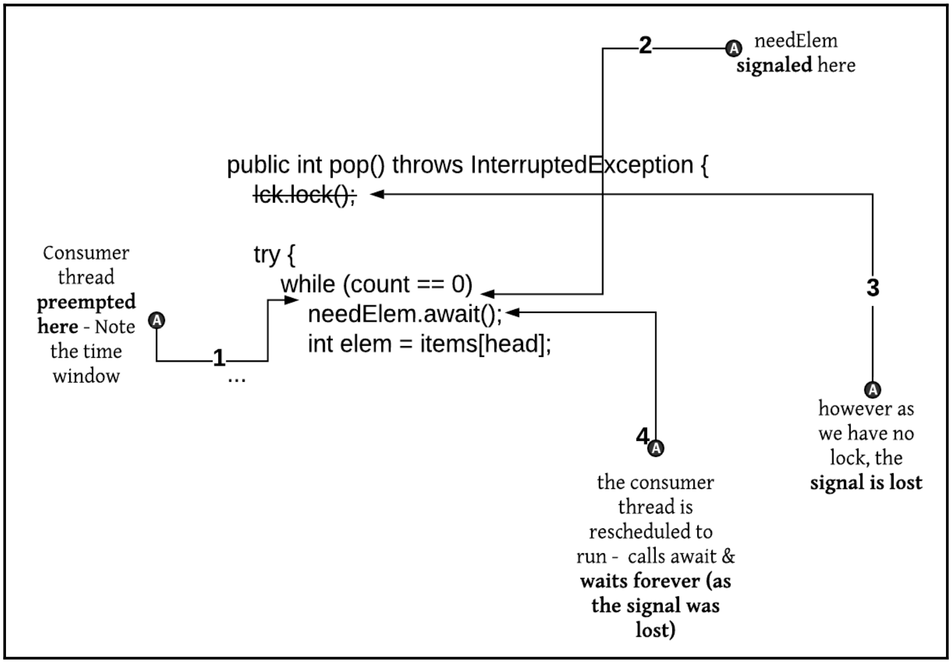
```
if (head == items.length)
  head = 0;
--count;
```

Note that, just like the `tail` variable, we keep rewinding the *head*. We decrement the count as the number of available elements is now reduced by one.

Spurious and lost wake-ups

Why do we need to acquire the lock first? Off course, the *count* variable is a shared state between the producers and consumers.

There is one more reason though: we need to call await *after acquiring the* `lck` *lock.*As mentioned at `https://docs.oracle.com/cd/E19455-01/806-5257/sync-30/index.html, the following situation can occur:`


2
needElem **signaled** here
public int pop() throws InterruptedException {
~~lck.lock();~~
Consumer thread **preempted here** - Note the time window
try {
while (count == 0)
needElem.await();
int elem = items[head];
1
...
3
4
the consumer thread is rescheduled to run - calls await & **waits forever (as the signal was lost)**
however as we have no lock, the **signal is lost**

As shown, there is no lock held, so the signal is lost. There is *no one to wake up,* so the *signal is lost.* For correct signal semantics, `await()` needs to be locked.

It is also imperative to check for the condition in a loop. In other words, after the thread wakes up, *it must retest the condition before proceeding further.* This is required to handle *spurious* and *lost wake-ups:*

```
public int pop() throws InterruptedException {
  lck.lock();

  try {
    /* while (count == 0) */
    if (count == 0) // we need a loop here
    needElem.await();
    int elem = items[head];
...
```

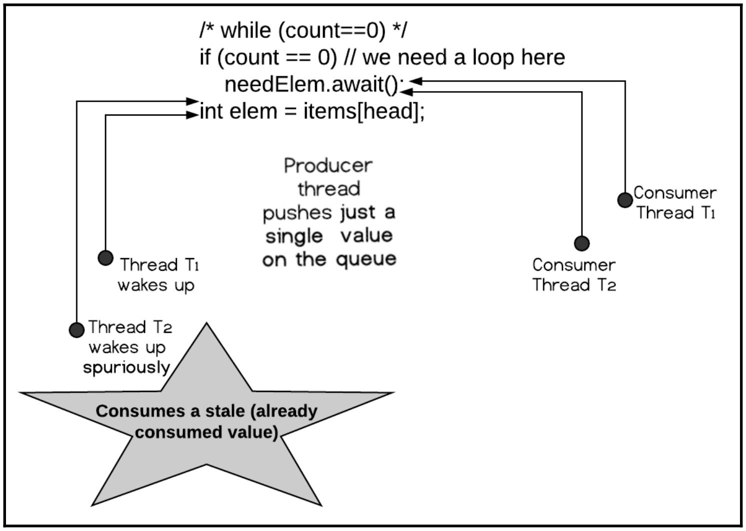
If we use the `if` condition, there is an insidious bug waiting to spring up. Due to arcane platform efficiency reasons, the `await()` method can return spuriously (without any reason)!

> *When waiting upon a condition, a spurious wake-up is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs, as a condition should always be waited upon in a loop, testing the state predicate that is being waited for. An implementation is free to remove the possibility of spurious wake-ups, but it is recommended that application programmers always assume that they can occur and so always wait in a loop.*

A quote from the relevant Java documentation at the mentioned link is as follows:

`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html`

Here is the possible buggy scenario:


/* while (count==0) */
if (count == 0) // we need a loop here
needElem.await();
int elem = items[head];
Producer thread pushes just a single value on the queue
Consumer Thread T1
Consumer Thread T2
Thread T1 wakes up
Thread T2 wakes up spuriously
Consumes a stale (already consumed value)

As shown, if we test the condition in a loop, *a producer* thread always wakes up, checks again, and then proceeds with the correct semantics.

Comparing and swapping

Locks are expensive; a thread that is blocked while trying to acquire the lock is suspended. Suspending and resuming threads is pretty expensive. Instead, we could use a **CAS** (**Compare And Set**) instruction to update a concurrent counter.

A CAS operation works on the follows:

- The variable's memory location (*x*)
- The *expected* value (*v*) of the variable
- The new value (*nv*) that needs to be set

The CAS operation automatically updates the value in x *to* nv*, but only if the existing value in* x *matches* v; o*therwise, no action is taken*.

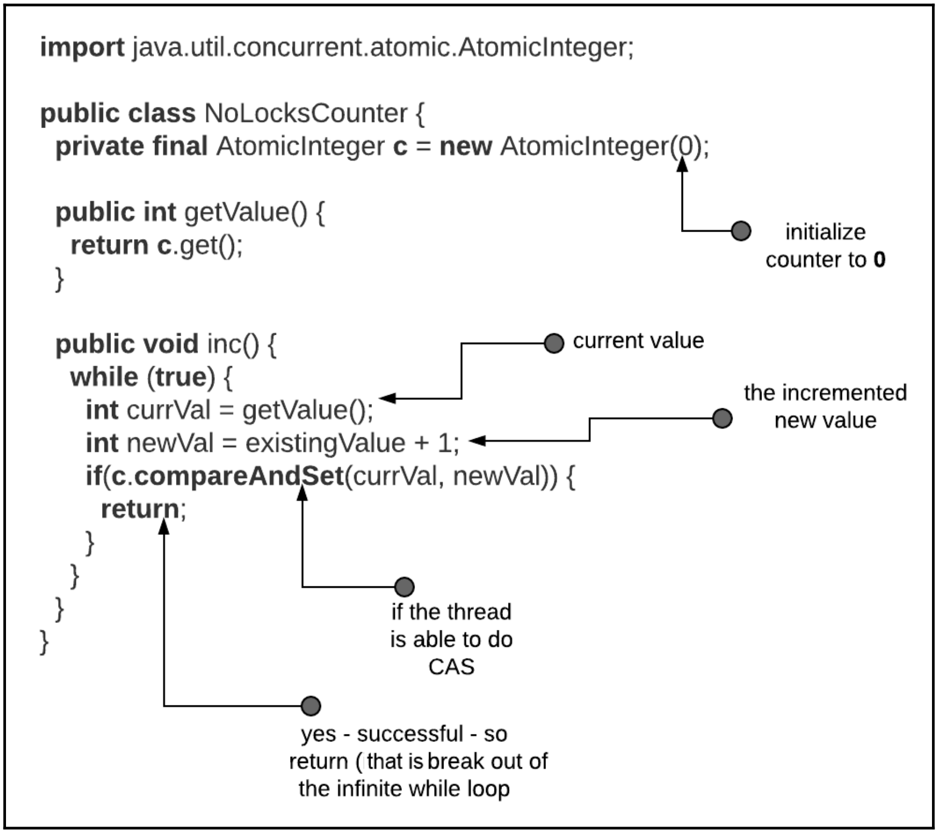
In both cases, the existing value of *x* is returned. For each CAS operation, the following three operations are performed:

1. Get the value
2. Compare the value
3. Update the value

The three operations specified are executed as *single, atomic* machine instructions.

When multiple threads attempt the CAS operation, *only one thread wins* and updates the value. *However, other threads are not suspended*. The threads for whom the CAS operation failed can reattempt the update.

The big advantage is that *context switches are completely avoided:*

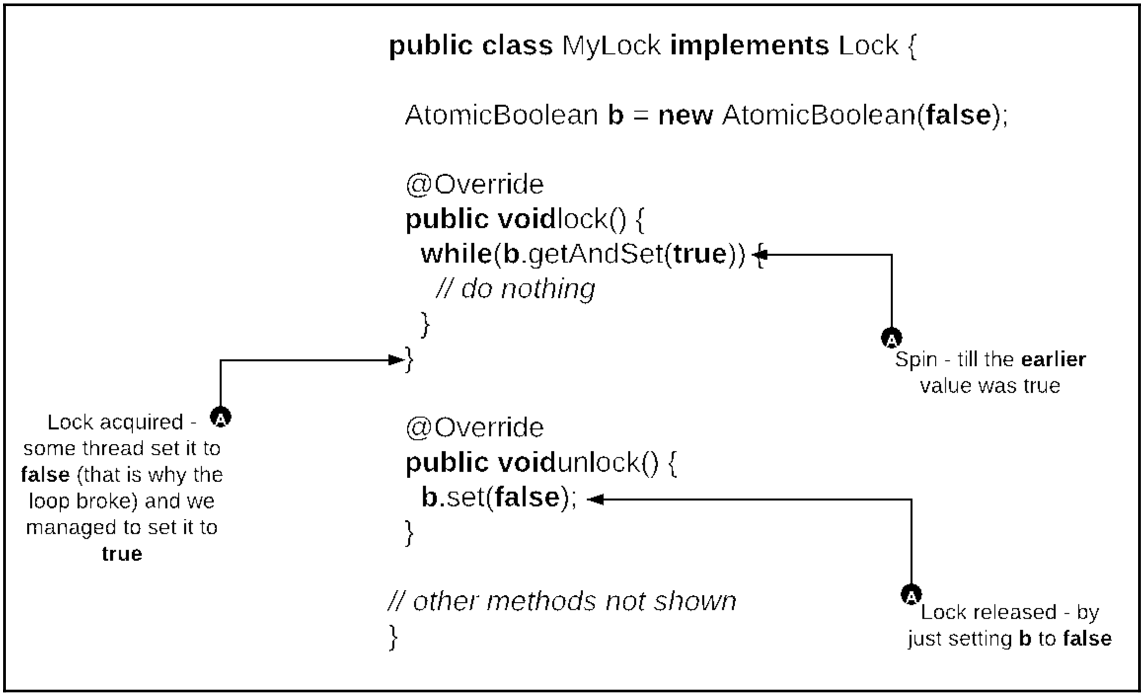


```
import java.util.concurrent.atomic.AtomicInteger;

public class NoLocksCounter {
  private final AtomicInteger c = new AtomicInteger(0);

  public int getValue() {
    return c.get();
  }

  public void inc() {
    while (true) {
      int currVal = getValue();
      int newVal = existingValue + 1;
      if(c.compareAndSet(currVal, newVal)) {
        return;
      }
    }
  }
}
```

initialize counter to **0**
current value
the incremented new value
if the thread is able to do CAS
yes - successful - so return (that is break out of the infinite while loop

As shown, a thread keeps looping and trying to win by attempting a CAS operation. The call takes the current value and the new value, and returns true *only when the update succeeds*! If some other thread won, the loop repeats, thereby trying again and again.

The CAS update operation is *atomic,* and more importantly it avoids the suspension (and the subsequent resumption) of a thread. Here, we use CAS to implement a flavor of our own locking:



```
public class MyLock implements Lock {

  AtomicBoolean b = new AtomicBoolean(false);

  @Override
  public voidlock() {
    while(b.getAndSet(true)) {
      // do nothing
    }
  }

  @Override
  public voidunlock() {
    b.set(false);
  }

// other methods not shown
}
```

Spin - till the **earlier** value was true

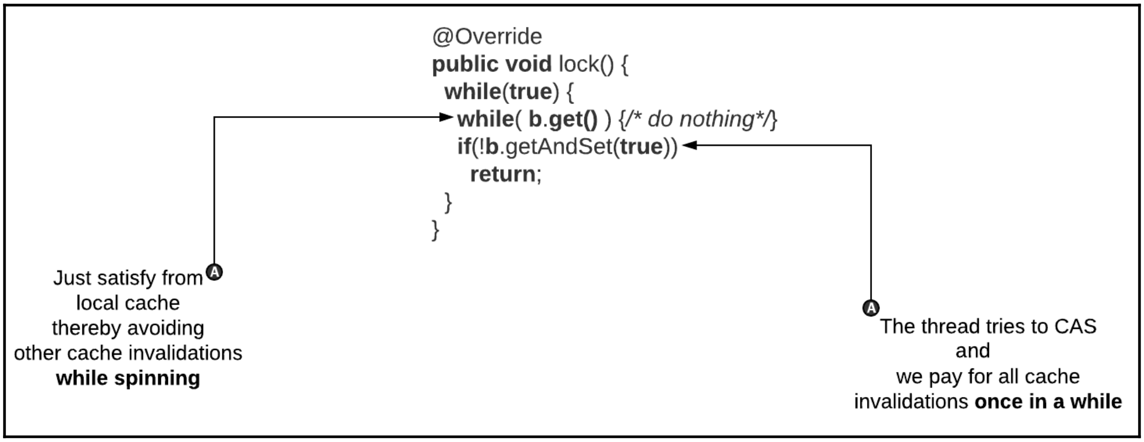
Lock acquired - some thread set it to **false** (that is why the loop broke) and we managed to set it to **true**

Lock released - by just setting **b** to **false**

The `getAndSet()` method *tries to set* the new value *and returns the previous one.* So, if the *previous value was false and we managed to set it to true* (remember that *compare and set* is atomic), we have acquired a lock.

As shown in the previous diagram, the CAS operation is used to implement locking by extending the l*ock* interface without blocking any threads! However, when multiple threads contend for the lock, thereby resulting in a *higher contention,* the performance deteriorates.

This is why, threads are running on a core. Each core has a cache. This cache stores a copy of the lock variable. The `getAndSet()` call causes all cores to *invalidate cached copies of the lock*. So, as we have more threads and more such spinning locks, there is too much unnecessary cache invalidation:



```
@Override
public void lock() {
  while(true) {
    while( b.get() ) {/* do nothing*/}
    if(!b.getAndSet(true))
      return;
  }
}
```

Just satisfy from local cache thereby avoiding other cache invalidations **while spinning**

The thread tries to CAS and we pay for all cache invalidations **once in a while**

The previous code improves things by *spinning* (that is to say, waiting for the lock) using the cached variable, **b**. When the value of **b** becomes false (thereby implying unlocking), the while loop breaks. Now, a `getAndSet()` call is made in order to acquire the lock.

Summary

We began this chapter by looking at race conditions and saw the need for synchronization, in real life situations as well as in concurrent code. We had a detailed look at race conditions and saw the role that the *volatile* keyword plays.

Next, we looked at the singleton pattern, which represents a program's global state. We saw how to safely share the state using monitors. We also *correct visibility* semantics and looked at an optimization called *double-checked locking*. We also saw how the *initialization on demand holder* design pattern resolves these problems.

We looked at a use case, a concurrent set implementation, using sorted linked lists. Using locks could lead to coarse-grained locking. Though semantically correct, this scheme allows only a single thread, and this could hurt concurrency.

The solution was to use the hand-over-hand design pattern. We studied it in depth, thereby understanding how *explicit locking* can give us a better solution, preserving the correctness and also improving the concurrency.

Finally, we covered the *producer/consumer* design pattern. We saw how conditions are used by threads for communication. We covered the subtlety involved in correctly using the conditions.

So, we have covered a lot of ground here, dear reader. Now, look at more design patterns in the next chapter.

3
More Threading Patterns

In this chapter, we will look at more synchronization patterns. We will start with a detailed look at **bounded buffers**. We will look at different design approaches, such as **client-throwing exceptions** and **polling**. We will see how to make the writer sleep when the buffer is full (and how to make the reader sleep when the buffer is empty), and this makes for an elegant client contract.

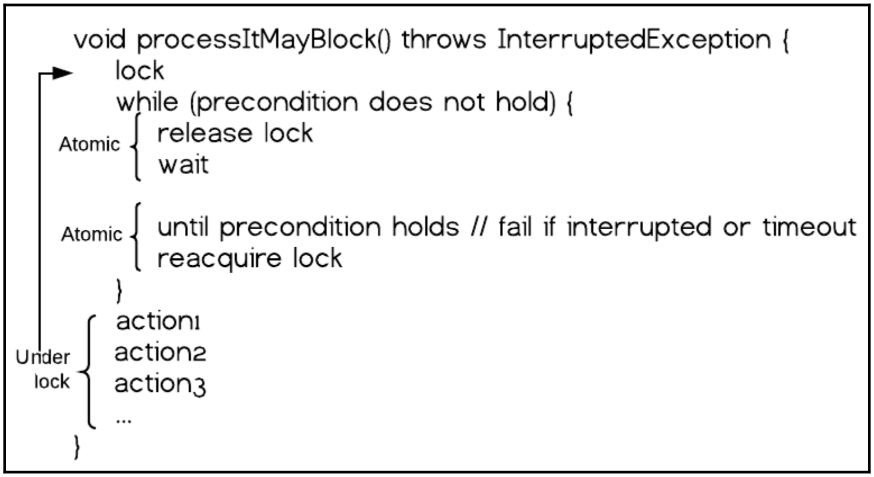
We will also look at r**eaders or writers lock**, a primitive synchronization to allow either multiple concurrent readers or a single writer. The idea is to increase the system's concurrency with correctly preserved concurrency semantics. We will look at two variations—**reader-friendly locks** and **fair locks**.

Next, we will discuss **counting semaphores**; these are used for implementing resource pooling. We will see how easily can we implement this construct.

We also implement a version of our own: `ReentrantLock`.

The chapter wraps up with a look at **countdown latches**, **cyclic barriers,** and **future tasks**.

The following is a general outline of the various patterns we will cover in this chapter:



```
void processItMayBlock() throws InterruptedException {
    lock
    while (precondition does not hold) {
        release lock
        wait

        until precondition holds // fail if interrupted or timeout
        reacquire lock
    }
    action1
    action2
    action3
    ...
}
```

We always acquire a lock; this makes sure that the following statement is checking for some precondition executes atomically. We will then check some preconditions specific to the algorithm, and if they are not met, we release the lock and wait. As shown in preceding diagram, these two actions are performed *atomically*.

Subsequently, the state of things changes, as another thread makes a substantial change to the program state, and broadcasts a signal.

Upon the receipt of this signal, the sleeping thread wakes up. Note that there could be more than one waiting thread. One or more of these wake up, acquire the lock, and *recheck* the condition. Again, both these operations are atomic.

We have seen in the previous chapter why this rechecking is needed. The thread then proceeds with the rest of the method logic. Hence, we shall be addressing the following topics:

- Bounded buffers
- Client-throwing exceptions and polling
- Readers or writers lock
- Reader-friendly locks
- Fair locks
- Counting semaphores
- ReentrantLock

- Countdown latches
- Cyclic barriers
- Future tasks

For complete code files you can visit `https://github.com/PacktPublishing/Concurrent-Patterns-and-Best-Practices`

A bounded buffer

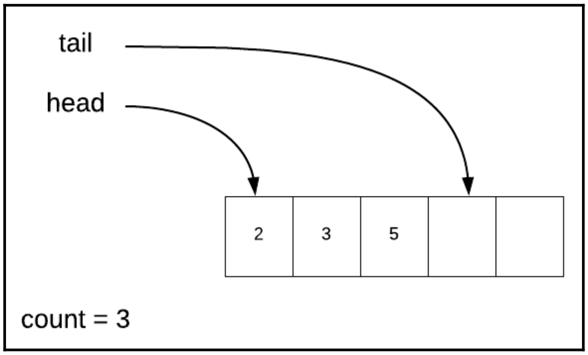
A **bounded buffer** is one with a finite capacity. You can only buffer a certain amount of elements in it. When there is no more space left to store the elements, the producer threads putting the elements wait for someone to consume some of the elements.

On the other hand, the consumer threads cannot take elements from an empty buffer. In such cases, the consumer thread will need to wait for someone to insert elements into the buffer.

Here comes the code, which we will explain as we go:

```
public abstract class Buffer {
  private final Integer[] buf;
  private int tail;
  private int head;
  private int cnt;
```

The elements are stored in an array of integers, `buf`:



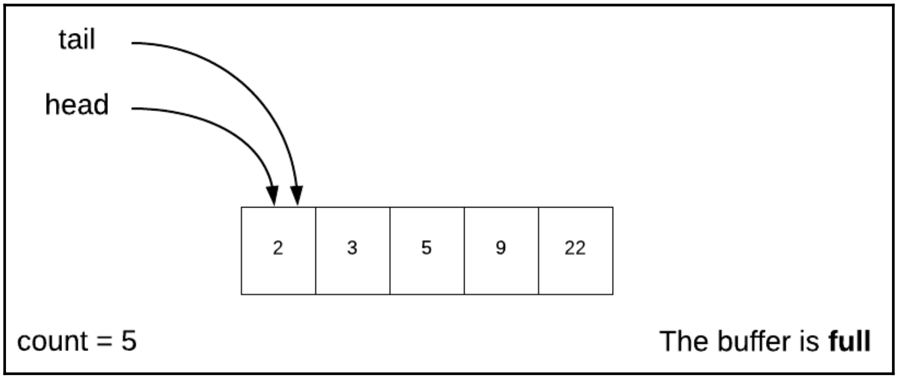
As shown in the preceding diagram, the field `tail` points to the next empty position to put the element in. In the provided diagram, three elements were inserted and none were taken. The first element to be consumed is *2*, which resides at the index 0 of the internal array. This is the index we hold in the `head` field. The value of *count* is *3;* as the array capacity is *5*, the buffer is not full:

```
protected Buffer(int capacity) {
  this.buf = new Integer[capacity];
  this.tail = 0;
  this.head = 0;
  this.cnt = 0;
}
```

The preceding snippet shows the constructor; it allocates the array and initializes the `buf` field with the array reference. The fields `tail`, `head`, and `cnt` are all initialized to 0:

```
protected synchronized final void putElem(int v) {
  buf[tail] = v;
  if (++tail == buf.length)
    tail = 0;
    ++cnt;
}
```

The *put* method, as shown here, puts a new element into the buffer:

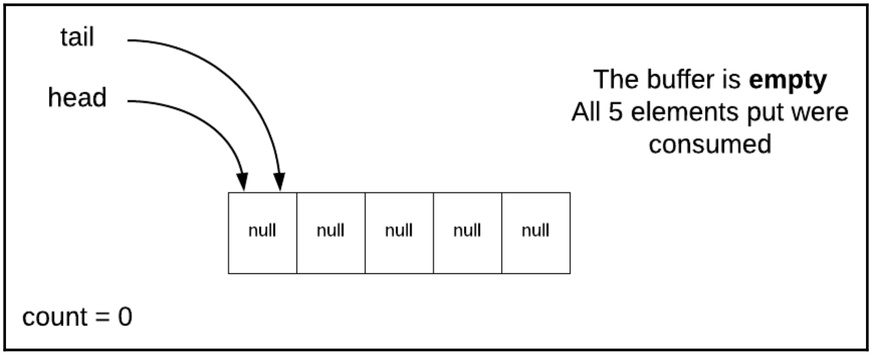


What should we do when the buffer is full? One design choice is to throw an exception. Another design choice is to wait till some thread consumes one or more elements:

```
protected synchronized final Integer getElem() {
  Integer v = buf[head];
  buf[head] = null;
  if (++head == buf.length)
    head = 0;
```

```
        --cnt;
      return v;
    }
```

Once we take an element from the buffer, we put a null in a slot, marking it as empty. The `get()` method could leave the buffer empty. This is shown in the following diagram:



The following are two helper methods:

```
    public synchronized final boolean isBufFull() {
      return cnt == buf.length;
      }

      public synchronized final boolean isBufEmpty() {
        return cnt == 0;
      }
    } // The buffer class completes here
```

When the count, `cnt`, equals the buffer length (5, in our example), the buffer is full. This is checked by the `isBufFull()` method. The `isBufEmpty()` method, likewise, checks the `cnt` field, which is 0 when the buffer is empty.

Whether to throw an error or to make the caller wait is a *strategy.* This is an example of a strategy pattern.

Strategy pattern – client polls

The following shows a design that throws an error when the buffer is empty and a get call is made, or full when a put call is made:

```
public class BrittleBuffer extends Buffer {
  public BrittleBuffer(int capacity) {
    super(capacity);
  }
```

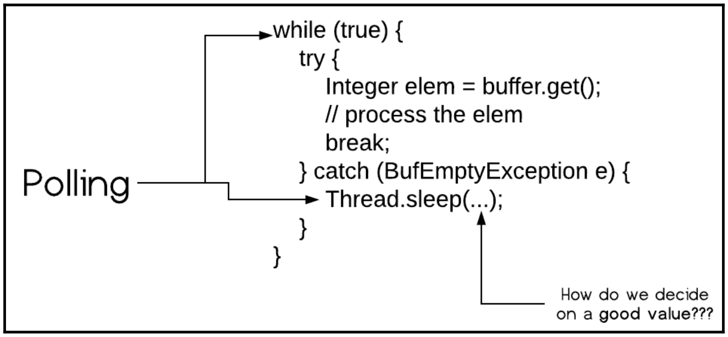
The constructor override just passes on the initial capacity to the superclass `super`:

```
public synchronized void put(Integer v) throws BufFullException {
  if (isBufFull())
    throw new BufFullException();
    putElem(v);
}
```

If the buffer is full when we execute a `put(v)` call, then we throw an error, a runtime exception, `BufFullException`:

```
public synchronized Integer get() throws BufEmptyException {
  if (isBufEmpty())
    throw new BufEmptyException();
    return getElem();
}
```

If the buffer is empty when we execute a `get()` call, we throw an error, `BufEmptyException`, which is a runtime exception again. The onus of handling these exceptions is now with the client of this code:



```
while (true) {
  try {
    Integer elem = buffer.get();
    // process the elem
    break;
  } catch (BufEmptyException e) {
    Thread.sleep(...);
  }
}
```

As shown in the preceding diagram, the client code needs to keep *polling*, to take an element from a buffer. If the buffer is not empty, things proceed well; however, when the buffer is empty, we catch the exception and keep repeating the call till we succeed.

There are two problems with this approach. Firstly, the client is using exceptions for flow control. We are using exceptions as sophisticated GOTO statements. The code is also harder to read; the actual element processing is hidden within all the exception handling.

See `https://web.archive.org/web/20140430044213/http://c2.com/cgi-bin/wiki?DontUseExceptionsForFlowControl` for more on this topic.

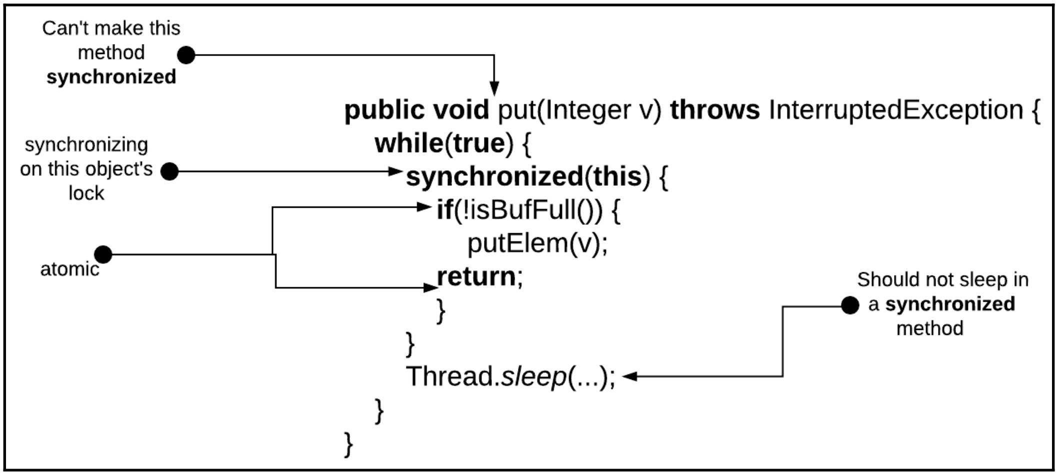
Secondly, what is a good amount of time for the thread to sleep? There is no clear answer here. This is essentially telling us to try an alternative design, so let's look at that next.

Strategy – taking over the polling and sleeping

The following version makes it somewhat easier for the client to use our interface:

```
public class BrittleBuffer2 extends Buffer {
  public BrittleBuffer2(int capacity) {
    super(capacity);
  }

  public void put(Integer v) throws InterruptedExcecption {
    while (true) {
      synchronized (this) {
        if (!isBufFull()) {
          putElem(v);
          return;
        }
      }
      Thread.sleep(...);
    }
  }
```

The put(Integer) method is an improvement, when compared to the previous version. We still have the polling; however, it is no longer part of the client's contract. We hide the polling from the client and make it simpler for them:



```
public void put(Integer v) throws InterruptedException {
  while(true) {
    synchronized(this) {
      if(!isBufFull()) {
        putElem(v);
      return;
      }
    }
    Thread.sleep(...);
  }
}
```

Can't make this method synchronized
synchronizing on this object's lock
atomic
Should not sleep in a synchronized method

There is a subtlety we should be aware of, as shown in the diagram given; the method is *not synchronized.* If we sleep in a synchronized method, other threads will never get the lock!

It would be a deadlock situation, as no other thread would be able to proceed; the state won't change and the infinite loop will keep checking a condition that will never be true.

The solution is to use the *synchronized (this)* idiom, which allows us to inspect the buffer state in a thread-safe way. If the condition is not met, the lock is released just before the sleep call. This ensures that other threads can proceed with the state change, and, as a result, the overall system would clock in progress:

```
public Integer get() throws InterruptedException {
  while (true) {
    synchronized (this) {
      if (!isBufEmpty())
        return getElem();
    }
      Thread.sleep(...);
  }
}
```

The get method is also used along similar lines. We hide the polling from the client. However, how long to sleep is still an issue.

Strategy – using condition variables

The problem of coming up with the right sleep time is tricky. There is no one value that fits all the situations. If instead, we don't use polling, and go for an alternate solution, it would be the best of both worlds!

Polling is busy waiting. We keep checking, which is unproductive, and running these checks takes time and CPU cycles. We will see how staying away from polling is a general theme in concurrent programming. We will revisit this theme again when we discuss the actor paradigm and the *tell versus ask* pattern.

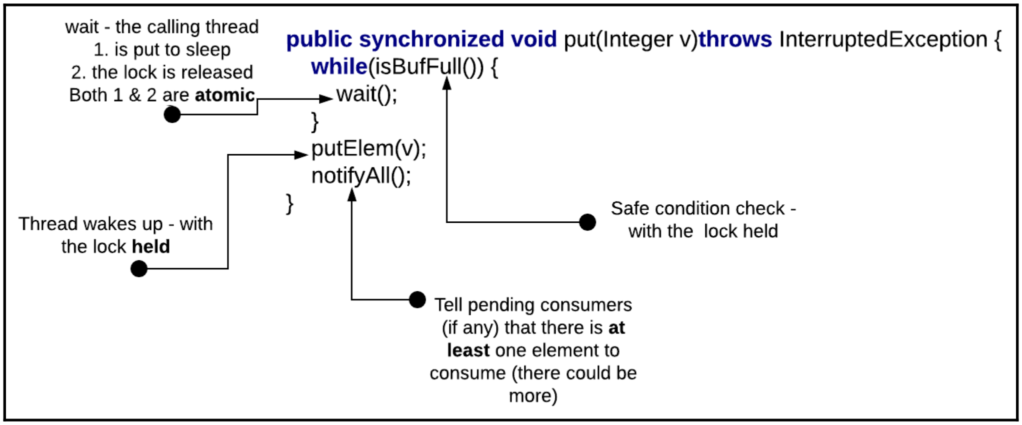
So, here goes the `CappedBuffer` class:

```
public class CappedBuffer extends Buffer {
  public CappedBuffer(int capacity) {
    super(capacity);
  }
```

The preceding code is similar to other strategies; no surprises in the following either:

```
public synchronized void put(Integer v) throws InterruptedException {
  while (isBufFull()) {
    wait();
  }
  putElem(v);
  notifyAll();
}
```

The `put(v)` method is super simple. The following diagram shows the subtle points about the code:



```
public synchronized void put(Integer v)throws InterruptedException {
  while(isBufFull()) {
    wait();
  }
  putElem(v);
  notifyAll();
}
```

wait - the calling thread
1. is put to sleep
2. the lock is released
Both 1 & 2 are **atomic**
Thread wakes up - with the lock **held**
Safe condition check - with the lock held
Tell pending consumers (if any) that there is **at least** one element to consume (there could be more)

The important snippet is this:

```
while (isBufFull()) {
  wait();
}
```

Note that the condition is checked inside a *synchronized* method. As a *synchronized* method implies an implicit lock, we are checking the condition in a thread-safe fashion.

Either the thread woke up from pending on a buffer that was full or the buffer was not full. We reach the `putElem(v)` statement. At this point, we surely know that there is at least one element to consume. We broadcast this fact to other possibly pending threads on an empty buffer:

```
public synchronized Integer get() throws InterruptedException {
  while (isBufEmpty()) {
    wait();
    }
  Integer elem = getElem();
  notifyAll();
  return elem;
}
```

The `get()` method is on the same lines:

```
while (isBufEmpty()) {
  wait();
}
```

If the buffer is empty, the thread is made to wait. After passing the condition, as we know this is the only thread executing the `getElem()` call, the following line is thread-safe:

```
 Integer elem = getElem();
```

We are guaranteed to get a *non-null* element:

```
notifyAll();
return elem;
```

Finally, we notify all the pending threads on the buffer-full condition and tell them that there is at least one free space to insert an element. Now, the element is returned.

Reader or writer locks

A **readers–writer** *(***RW***)* or **shared-exclusive** lock is a primitive synchronization that allows concurrent access for read-only operations, as well as exclusive write operations. Multiple threads can read the data concurrently, but for writing or modifying the data, an exclusive *lock* is needed.

A writer has *exclusive access for* writing data. Till the current writer is done, other writers and readers will be blocked. There are many cases where data is read more often than it is written.

The following code shows how we use the locks to provide concurrent access to a Java Map<K, V>. The code synchronizes the internal map, using an RW lock:

```
public class RWMap<K, V> {
 private final Map<K, V> map;
 private final ReadWriteLock lock = new ReadWriteLock();
 private final RWLock r = lock.getRdLock();
 private final RWLock w = lock.getWrLock();

 public RWMap(Map<K, V> map) {
 this.map = map;
 }

 public V put(K key, V value) throws InterruptedException {
 w.lock();
 try {
 return map.put(key, value);
 } finally {
 w.unlock();
 }
 }

 public V get(Object key) throws InterruptedException {
 r.lock();
 try {
 return map.get(key);
 } finally {
 r.unlock();
 }
 }

 public V remove(Object key) throws InterruptedException {
 w.lock();
 try {
 return map.remove(key);
 } finally {
 w.unlock();
 }
 }

 public void putAll(Map<? extends K, ? extends V> m) throws
InterruptedException {
 w.lock();
 try {
 map.putAll(m);
 } finally {
 w.unlock();
 }
 }


 public void clear() throws InterruptedException {
 w.lock();
 try {
 map.clear();
 } finally {
 w.unlock();
 }
 }
}
```

The `get` method uses a reader lock; this allows any number of reader threads to concurrently access the *map* field. On the other hand, if a thread needs to update the map—it acquires a writer lock.

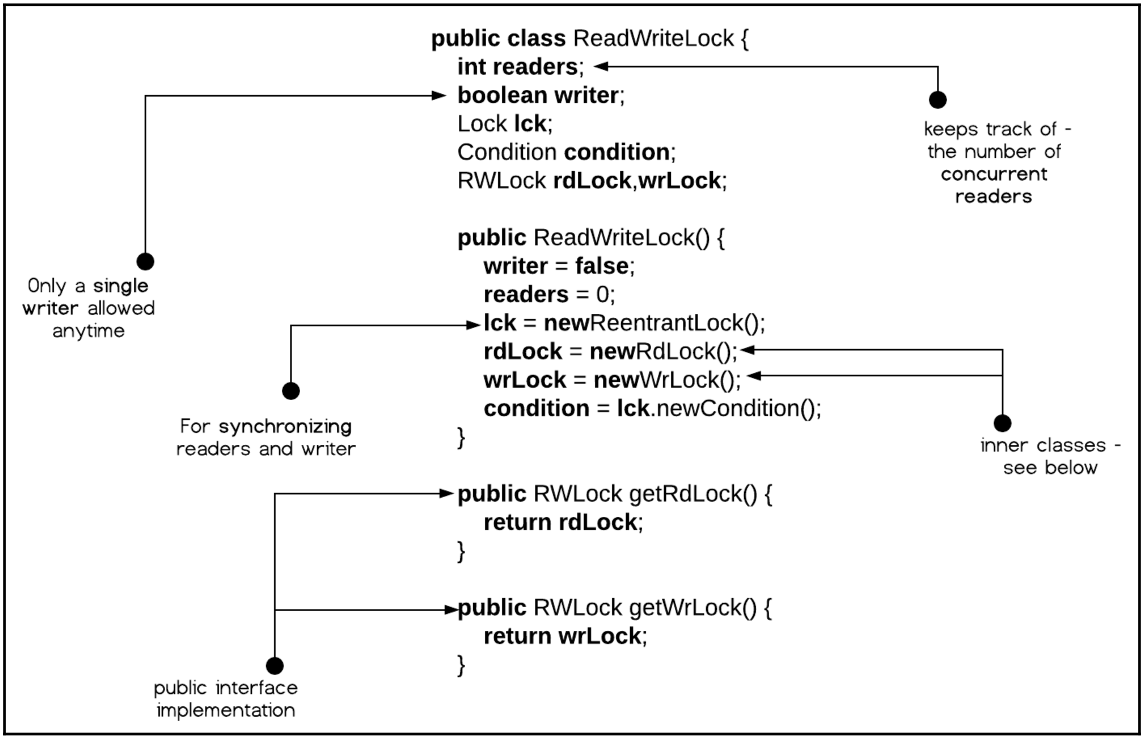
The writer lock ensures that the writer thread has *exclusive access* to the map. This preserves the thread safety, and still ensures increased read concurrency.

However, the lock needs to be *fair,* too! We will soon see what this means—read on...

A reader-friendly RW lock

RW locks can give priority to readers (*read-friendly*) or writers (*write-friendly*). We will see how *concurrency* and *starvation* aspects come up during the design.

The following diagram shows the `ReadWriteLock` class:



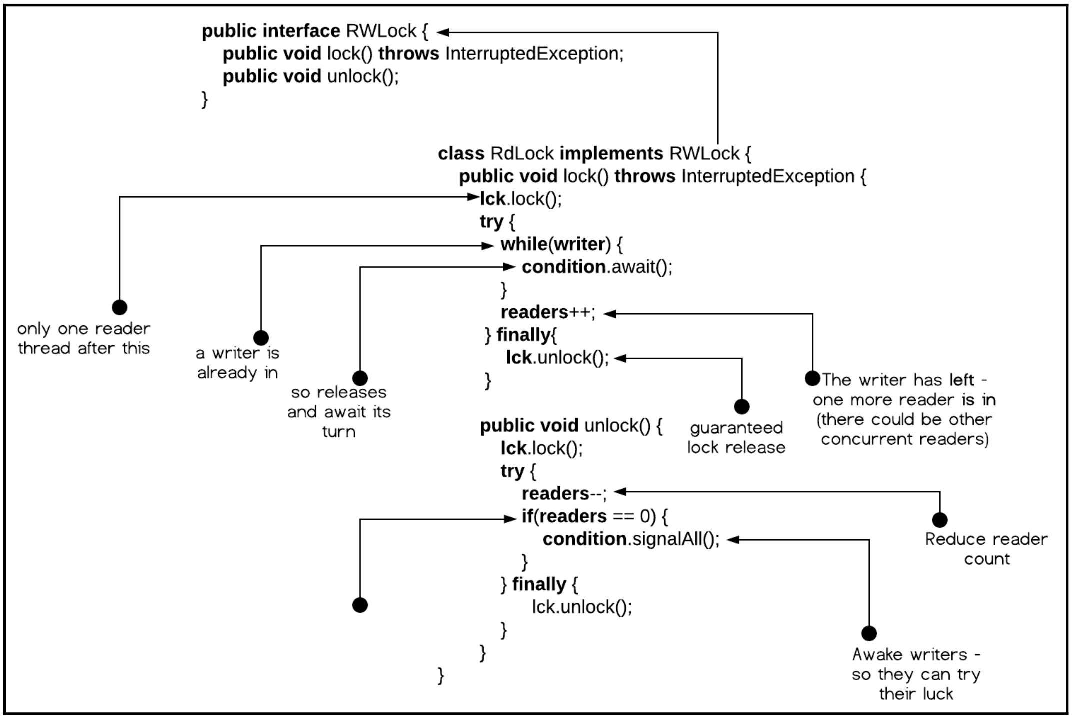
```
public class ReadWriteLock {
   int readers;
   boolean writer;
   Lock lck;
   Condition condition;
   RWLock rdLock,wrLock;

   public ReadWriteLock() {
      writer = false;
      readers = 0;
      lck = newReentrantLock();
      rdLock = newRdLock();
      wrLock = newWrLock();
      condition = lck.newCondition();
   }

   public RWLock getRdLock() {
      return rdLock;
   }

   public RWLock getWrLock() {
      return wrLock;
   }
```

keeps track of - the number of concurrent readers
Only a single writer allowed anytime
For synchronizing readers and writer
inner classes - see below
public interface implementation

The *readers* field is an *int*—it represents the number of readers at any given instance of time. On the other hand, the *writer* field is a boolean—there can be only *one writer,* or there can be *multiple readers*—so a boolean suffices. We synchronize on an external lock, represented by the `lck` field.

The following diagram shows the design of the reader and writer locks, `RdLock` and `WrLock`. These are the inner classes of `ReadWriteLock`. The `ReadWriteLock` class will be used by the application code to acquire read and write locks; it is a *facade:*

A *facade* is a design pattern from the GoF book. It is used to hide the complexity of the system, so it is easier to use. For example, when we order pizza over the phone, we talk to a salesperson at the pizza outlet. They will take the order, charge the card, and, within some time, it is delivered to us. We are thereby hidden from all the complexities of pizza-making. The sales representative is a *facade* for us. They hide all the complexity and facilitate the pizza-buying process. A facade is also a facilitator.



```
public interface RWLock {
   public void lock() throws InterruptedException;
   public void unlock();
}

            class RdLock implements RWLock {
               public void lock() throws InterruptedException {
                  lck.lock();
                  try {
                     while(writer) {
                        condition.await();
                     }
                     readers++;
                  } finally{
                     lck.unlock();
                  }

                  public void unlock() {
                     lck.lock();
                     try {
                        readers--;
                        if(readers == 0) {
                           condition.signalAll();
                        }
                     } finally {
                          lck.unlock();
                     }
                  }
            }
```

only one reader thread after this
a writer is already in
so releases and await its turn
guaranteed lock release
The writer has left - one more reader is in (there could be other concurrent readers)
Reduce reader count
Awake writers - so they can try their luck

As shown in the preceding diagram, the `RdLock` design firstly makes sure it is the *only* thread, as implied by the `lck.lock()` statement. If there is already a writer active, the reader thread relinquishes the lock, `lck`, and waits for the writer to release the lock:

```
while (writer) {
  condition.await();
}
readers++;
```

Either way, when the `reader` passes this check (either there is no `writer` or the reader woke up as a result of the `writer` unlocking and resumes), it increments the number of `reader` fields in the system.

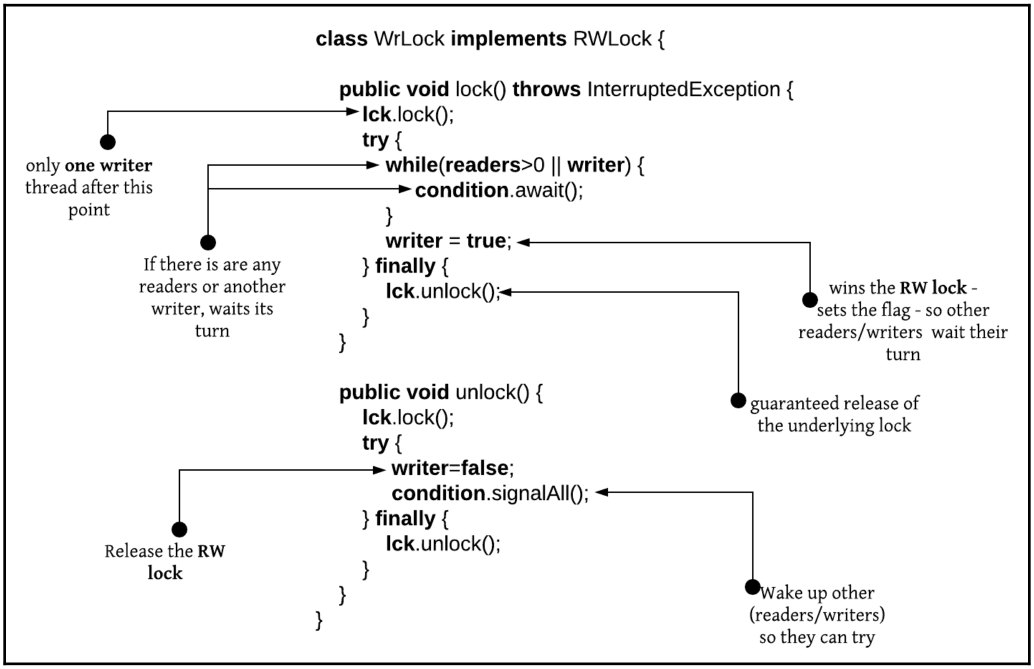
Releasing an `RdLock` is also similar:

```
readers--;
if (readers == 0) {
  condition.signalAll();
}
```

The `reader` count is decremented. If this were the last reader, it would try to wake up pending `writer` fields, if there were any!

This is a read-friendly `RwLock`; it allows for maximum read concurrency, but in the case of high-contention can starve writers. There is nothing to stop a flood of readers from starving the writers. Writer threads would starve, as the lock cannot be acquired as long as there is at least one reader thread.

The following diagram shows the writer lock, `WrLock`. We again make sure that the lock, `lck`, is acquired, thereby making sure there is only ever one writer checking and fiddling with the internal state:



```
class WrLock implements RWLock {

    public void lock() throws InterruptedException {
       lck.lock();
       try {
          while(readers>0 || writer) {
             condition.await();
          }
          writer = true;
       } finally {
          lck.unlock();
       }
    }

    public void unlock() {
       lck.lock();
       try {
          writer=false;
          condition.signalAll();
       } finally {
          lck.unlock();
       }
    }
}
```

only **one writer** thread after this point
If there is are any readers or another writer, waits its turn
wins the **RW lock** - sets the flag - so other readers/writers wait their turn
guaranteed release of the underlying lock
Release the **RW lock**
Wake up other (readers/writers) so they can try

The following is a snippet:

```
while (readers > 0 || writer) {
  condition.await();
}
writer = true;
```

It makes sure that the writer sleeps if there are one or more readers, or another writer. In that case, the writer thread is put to sleep, as seen in the previous chapter; this also atomically releases the lock, `lck`, associated with the condition.

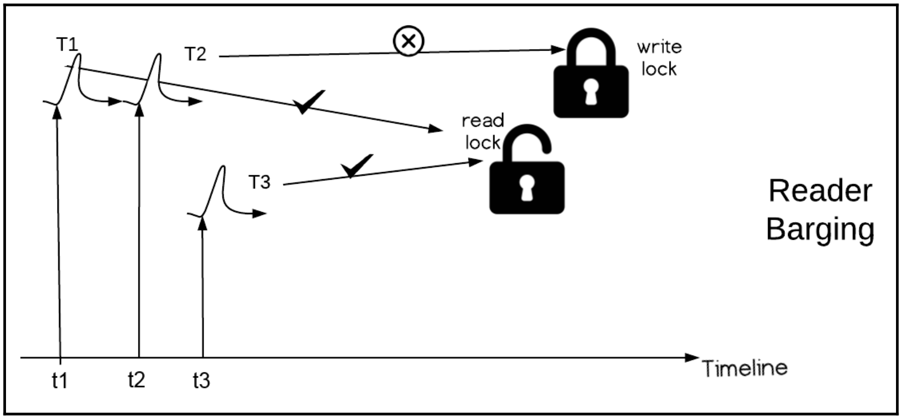
Once the precondition is established, where either all readers or the writer have released the lock—the writer turns the *writer* flag on. Note that the condition wake-up semantics ensure that the writer will always wake up with the lock held.

This makes sure that the flag is updated in a thread-safe manner, and that completes the locking call.

Unlocking the write lock is shown in the following snippet:

```
writer = false;
condition.signalAll();
```

The flag is just turned off, and the wake-up call is issued for any possible pending readers or writers, so they can try their luck acquiring the lock:



The preceding diagram shows *reader barging*. This happens when the reader threads keep acquiring the lock; the writer may not get any chance to write—even though it came earlier. This is a read-friendly lock and could be unfair to the writer threads.

This is *starvation*—the writer could be kept waiting forever, thereby starving it of the lock.

A fair lock

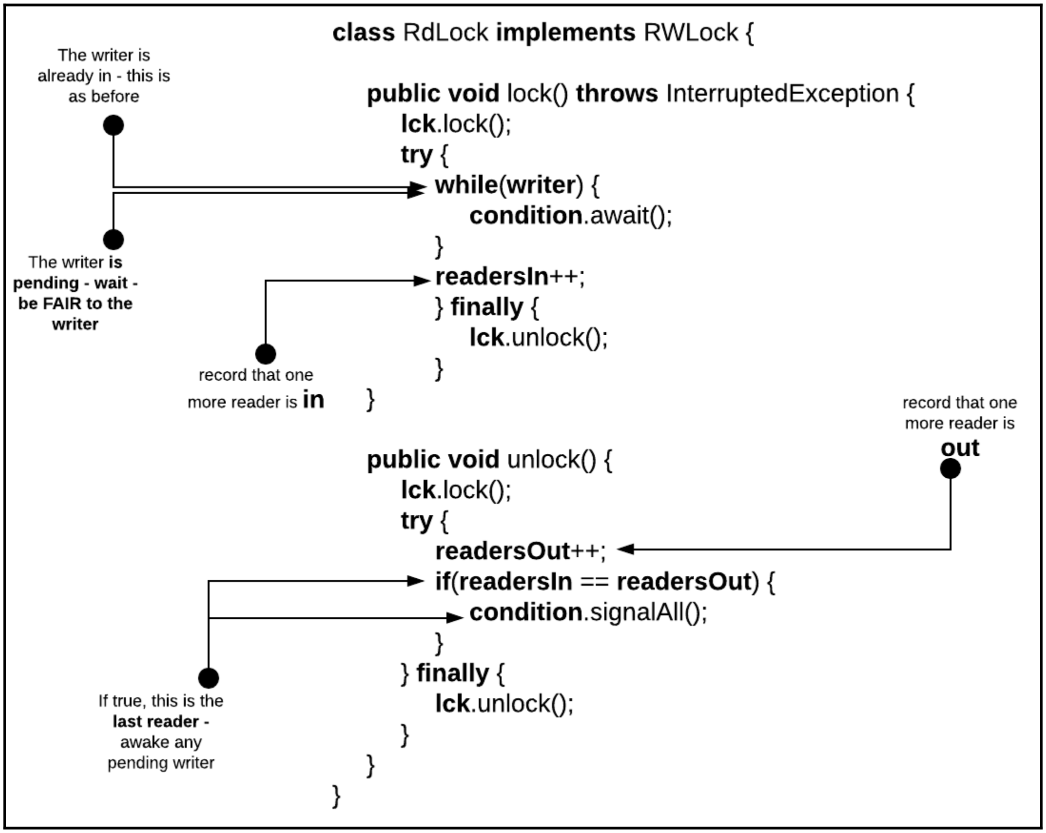
The following code shows an implementation that is *fair* to the writer. If the writer demanded the lock earlier, it gets it.

First come the facade changes:

```
public class FairReadWriteLock {
  int readersIn, readersOut;
  boolean writer;
  Lock lck;
  Condition condition;
  RWLock rdLock, wrLock;

  public FairReadWriteLock() {
    readersIn = readersOut = 0;
    writer = false;
    lck = new ReentrantLock();
    condition = lck.newCondition();
    rdLock = new RdLock();
    wrLock = new WrLock();
  }

  public RWLock getRdLock() {
    return rdLock;
  }

  public RWLock getWrLock() {
    return wrLock;
  }
```

The code is mostly identical to the previous version. The notable thing is that we have replaced the *readers* field with two fields, `readersIn` and `readersOut`.

The following diagram shows how the design works toward a *fairer* write lock:



```
class RdLock implements RWLock {

  public void lock() throws InterruptedException {
    lck.lock();
    try {
      while(writer) {
        condition.await();
      }
      readersIn++;
      } finally {
        lck.unlock();
      }
  }

  public void unlock() {
    lck.lock();
    try {
      readersOut++;
      if(readersIn == readersOut) {
        condition.signalAll();
      }
    } finally {
      lck.unlock();
    }
  }
}
```

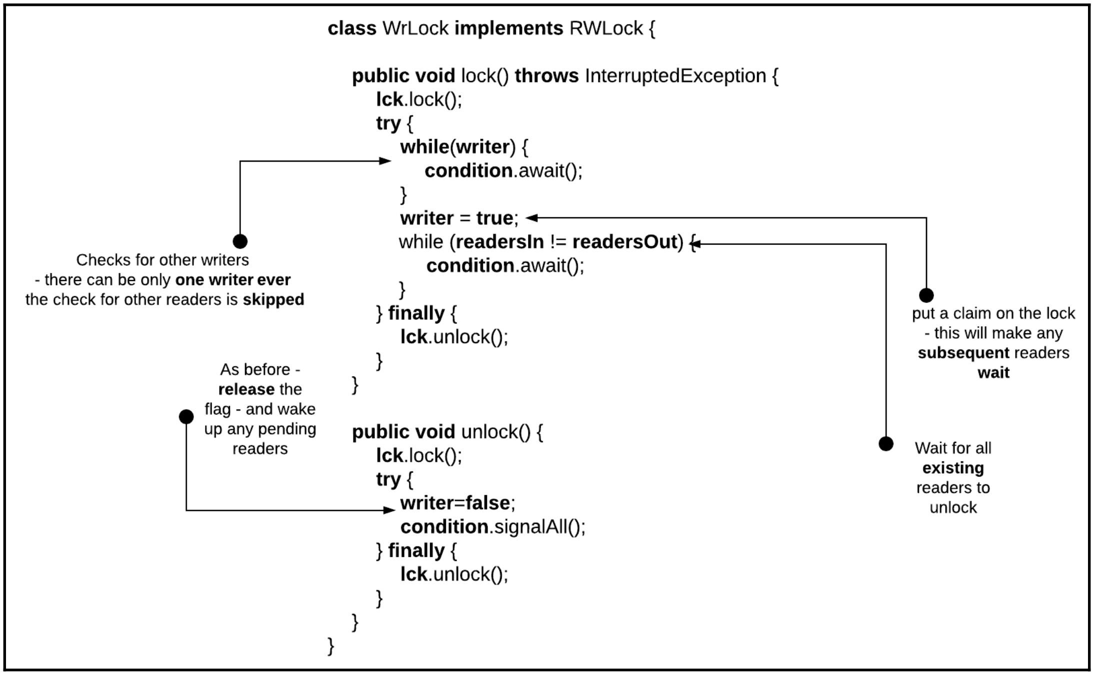
The writer is already in - this is as before
The writer **is pending - wait - be FAIR to the writer**
record that one more reader is **in**
record that one more reader is **out**
If true, this is the **last reader -** awake any pending writer

The reader lock method, as before, waits to see whether the `writer` flag is on. However, as noted, the writer being on could also mean that a writer is *pending*. In that case, the reader waits.

Otherwise, the reader continues and increments the `readersIn` field. The unlock method, on the other hand, increments the `readersOut` field. Note that the `readersOut` variable will always be less than or equal to the `readersIn` variable:

```
while (readersIn != readersOut) {
  condition.await();
}
```

When the two match, it means that the thread is the *last reader* unlocking the `RwLock`. So, it signals the condition, thereby awakening any pending writers:



```
class WrLock implements RWLock {

    public void lock() throws InterruptedException {
        lck.lock();
        try {
            while(writer) {
                condition.await();
            }
            writer = true;
            while (readersIn != readersOut) {
                condition.await();
            }
        } finally {
            lck.unlock();
        }
    }

    public void unlock() {
        lck.lock();
        try {
            writer=false;
            condition.signalAll();
        } finally {
            lck.unlock();
        }
    }
}
```

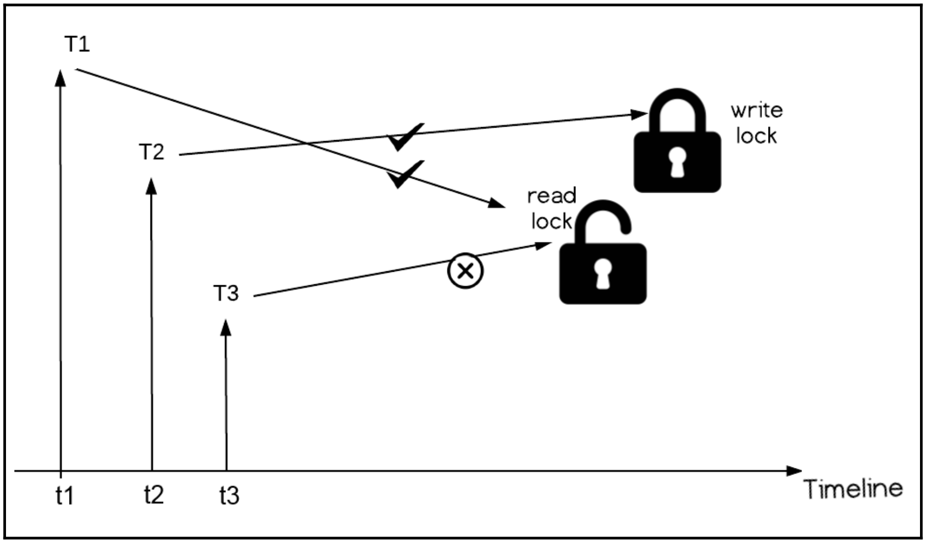
Checks for other writers - there can be only **one writer ever** the check for other readers is **skipped**

put a claim on the lock - this will make any **subsequent** readers **wait**

Wait for all **existing** readers to unlock

As before - **release** the flag - and wake up any pending readers

The design allows the writer to have a go, if it came earlier. This is shown in the following diagram:

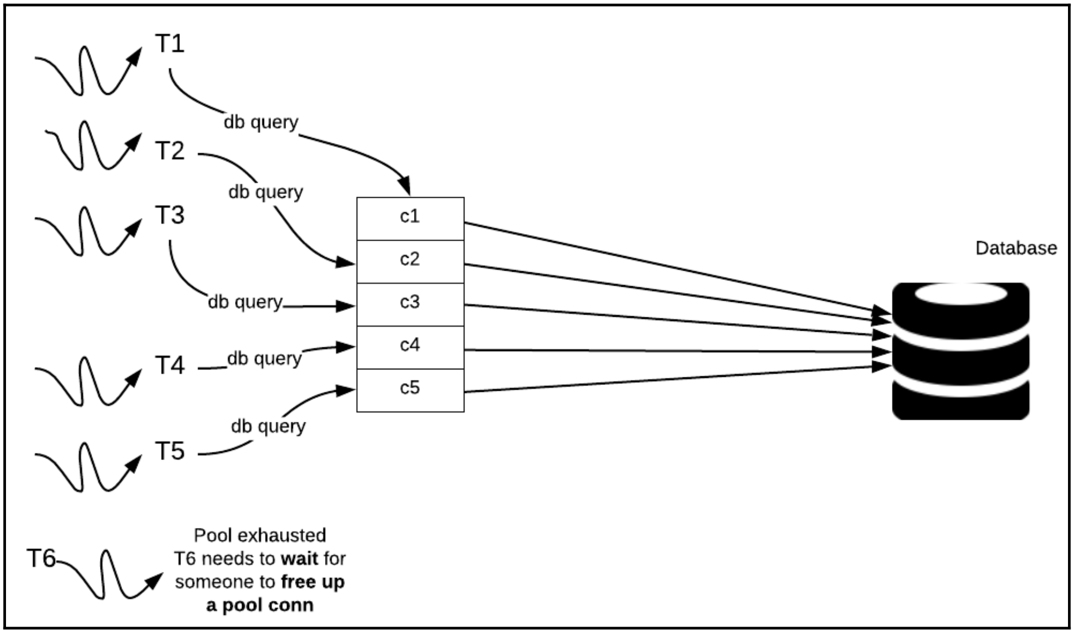


Due to the way things are structured, T2 is waiting longer than T3. The moment T1 releases the reader lock, T2 gets a chance to set the *writer* flag. This holds all the prospective readers, in this case, T3, from acquiring the reader lock and, as a result, the writer can proceed.

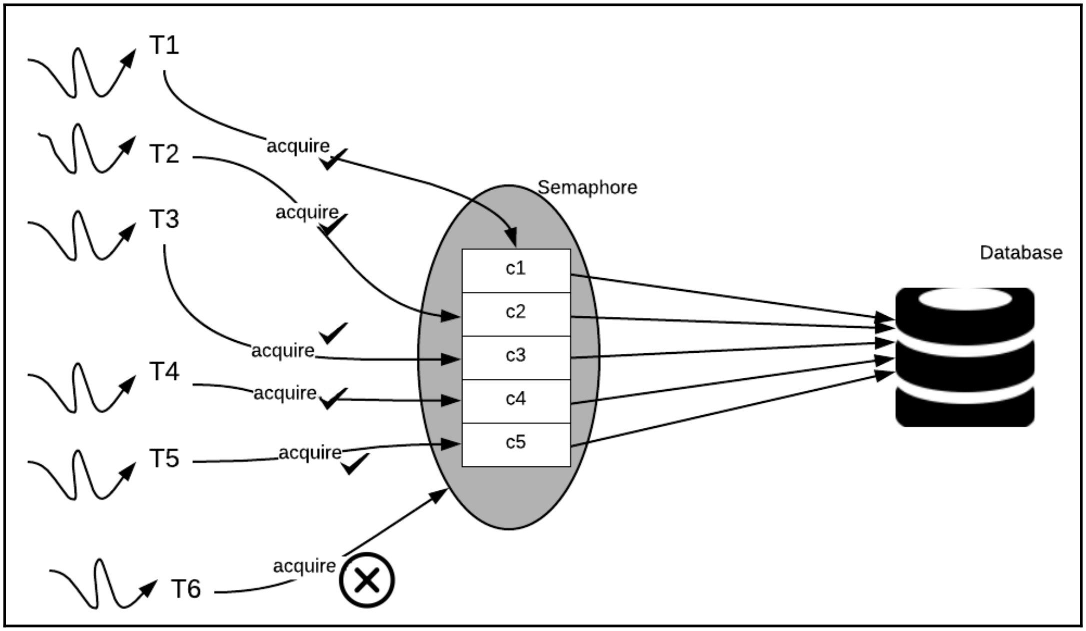
Counting semaphores

Concurrent applications usually have a pool of resources. For example, we have *thread pooling* and *connection pooling.* Creating and destroying such a connection as we go is costly. Instead, a pool is created, and whenever the app needs a resource, it goes and asks the pool.

The pool is configured to hold a *certain number* of these resources. For example, 20 database connections or 355 threads. When the demand is high, the pool could get exhausted. Unless some resources are released, the requesting thread should be put to sleep:



A *semaphore* would come, handy in implementing such scenarios. The semaphore is initialized with an initial capacity, *cap*, representing the configured pool size:



The following listing shows a semaphore implementation:

```
public class Semaphore {
 private final int cap;
 private int count;
 private final Lock lck;
 private final Condition condition;

 public Semaphore(int cap) {
 this.cap = cap;
 count = 0;
 lck = new ReentrantLock();
 condition = lck.newCondition();
 }

 public void acquire() throws InterruptedException {
 lck.lock();
 try {
 while (count == cap) {
 condition.await();
 }
 count++;
 } finally {
 lck.unlock();
 }
```

```
    }

    public void release() {
    lck.lock();
    try {
    count--;
    condition.signalAll();
    } finally {
    lck.unlock();
    }
    }

   }
```

The interesting part is the snippet where the lock is granted:

```
                while (count == cap) {
                    condition.await();
                }
                count++;
```

Till the count reaches the capacity, we keep granting the request. At some point, the count equals the capacity. This means that the pool is exhausted:

```
                count--;
                condition.signalAll();
```

When a thread releases a connection, the condition is signalled, so the pending thread(s) wake up and try getting the pool resource.

Our own reentrant lock

The following code shows how we could implement the reentrant lock ourselves. The code shows the reentrancy semantics, too. We just show the *lock* and *unlock* method:

```
public class YetAnotherReentrantLock {
 private Thread lockedBy = null;
 private int lockCount = 0;
```

The class has two fields—a `lockedBy` thread reference and a `lockcount`—which are used to track how many times the thread recursively locked itself:

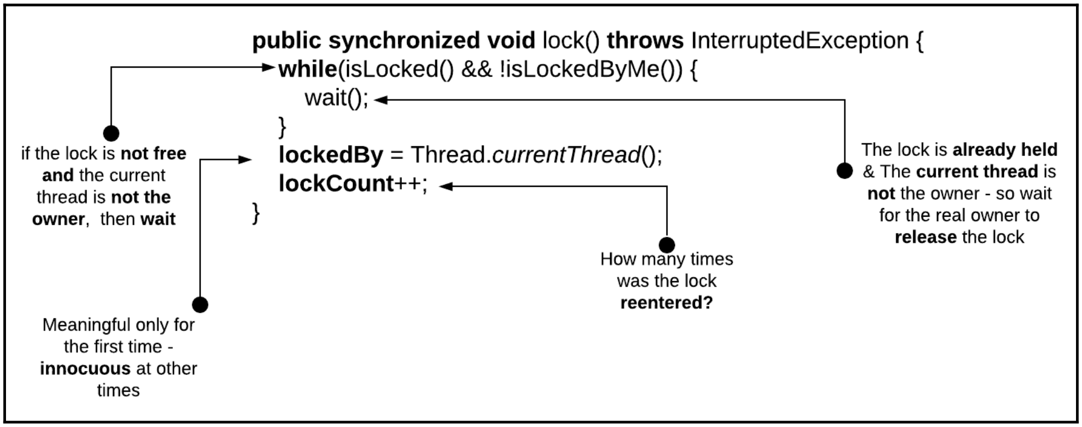
```
private boolean isLocked() {
  return lockedBy != null;
}
```

```
  private boolean isLockedByMe() {
    return Thread.currentThread() == lockedBy;
  }
```

The preceding snippet shows two helper methods. Using such helpers help make the code more readable:

```
  public synchronized void lock() throws InterruptedException {
    while (isLocked() && !isLockedByMe()) {
      this.wait();
   }
   lockedBy = Thread.currentThread();
   lockCount++;
  }
```

Here is a pictorial analysis of the `lock()` method:



```
public synchronized void lock() throws InterruptedException {
  while(isLocked() && !isLockedByMe()) {
    wait();
  }
  lockedBy = Thread.currentThread();
  lockCount++;
}
```

if the lock is **not free and** the current thread is **not the owner**, then **wait**

Meaningful only for the first time - **innocuous** at other times

How many times was the lock **reentered?**

The lock is **already held** & The **current thread** is **not** the owner - so wait for the real owner to **release** the lock

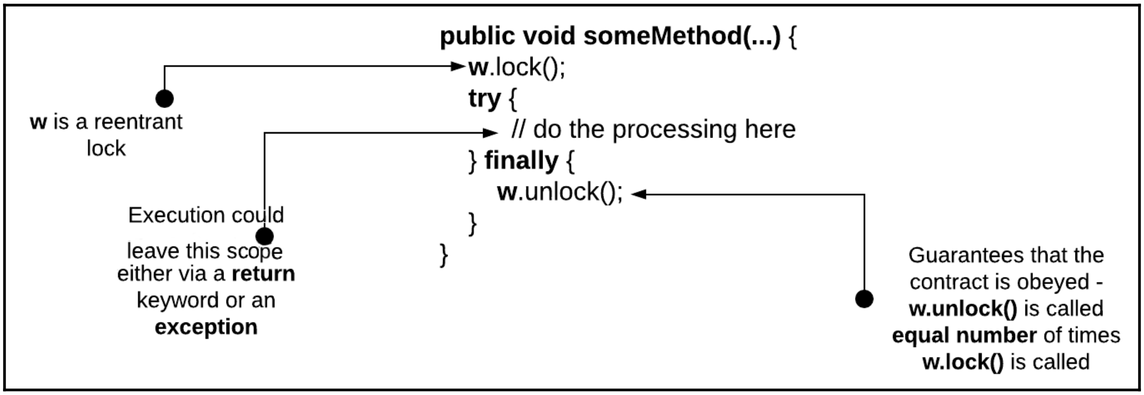
Keep in mind the lock semantics—the lock is granted if and only if it's in a released state! Otherwise, some other thread holds the lock, so the thread needs to wait.

Ownership is tracked using the thread reference. As no two threads can have the same object reference, this works fine:

```
  public synchronized void unlock() {
    if (isLockedByMe()) {
      lockCount--;
    }
    if (lockCount == 0) {
     lockedBy = null;
     this.notify();
    }
  }
```

The unlock() call is symmetrical. The following is the client contract:

As many times the lock is entered (via the lock() method) you need to unlock them (via the unlock() method) an equal number of times:



```
public void someMethod(...) {
    w.lock();
    try {
        // do the processing here
    } finally {
        w.unlock();
    }
}
```

w is a reentrant lock
Execution could leave this scope either via a return keyword or an exception
Guarantees that the contract is obeyed - w.unlock() is called equal number of times w.lock() is called

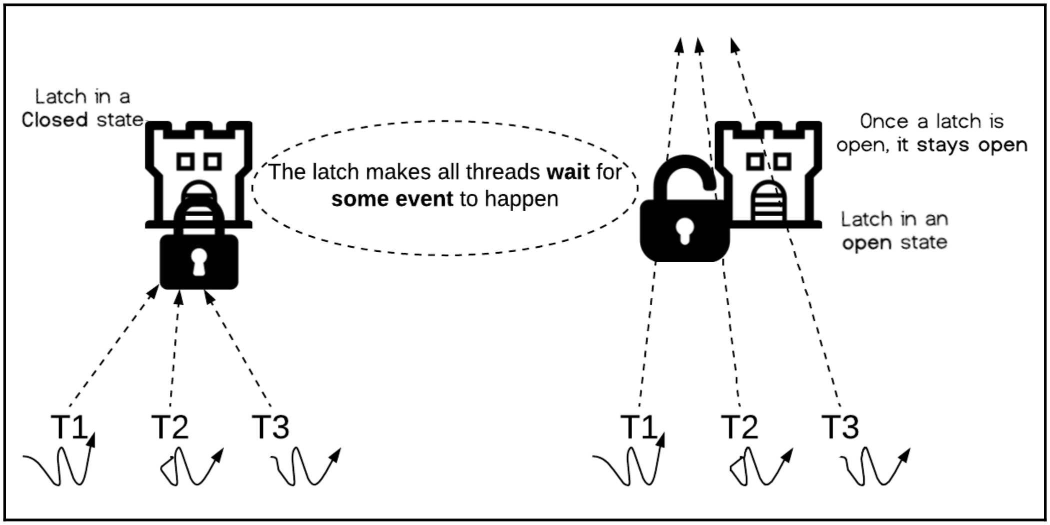
The *try* and *finally* functions we used previously ensure this contract. However, the program flow goes out, and the finally clause makes sure that both of these times match:

```
if (lockCount == 0) {
  lockedBy = null;
  this.notify();
}
```

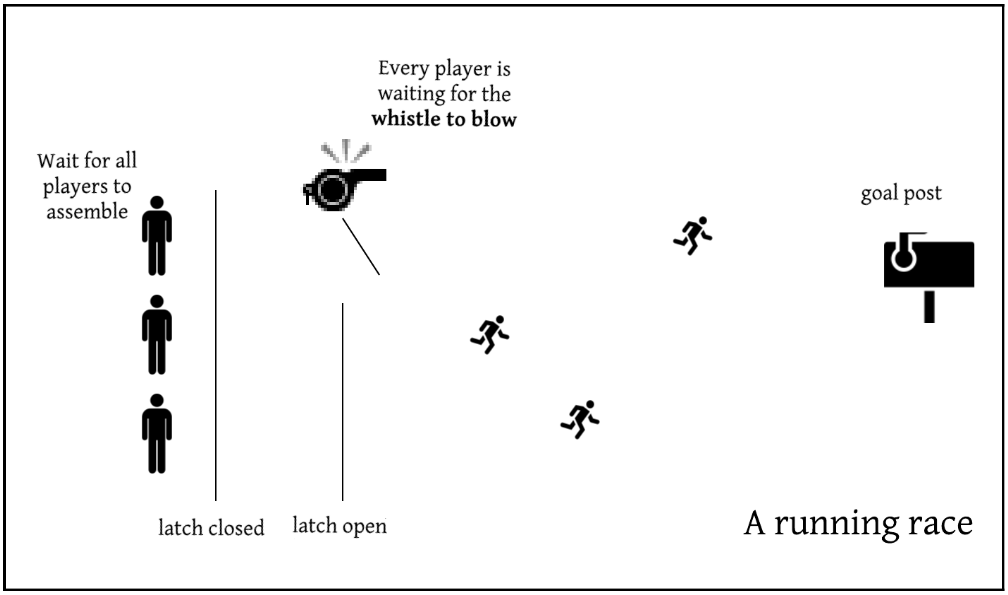
If the thread has released the lock correctly, as described previously, the lockCount variable will be decremented back to 0. We reset the lockBy field to null, thereby releasing the lock, and broadcast the availability of the lock to other possibly pending threads.

Countdown latch

A *latch* is yet another synchronizer. It acts as a gate—threads wait for the gate to open; once the gate— that is—the latch opens, all threads enter it:


Latch in a
Closed state
The latch makes all threads wait for some event to happen
Once a latch is open, it stays open
Latch in an open state
T1
T2
T3
T1
T2
T3

Why do we need latches? A latch is used to ensure that, unless some essential, precursor activity has happened, other activities wait for it. Let's look at a real-life example:


Every player is waiting for the whistle to blow
Wait for all players to assemble
goal post
latch closed
latch open
A running race

The preceding diagram shows how running a race happens on a nice evening. All the players need to assemble at the starting point and wait for the whistle to blow. Once the whistle goes, the players start running, and the race starts.

The starting point is the place of rendezvous: everyone needs to come and wait there for the important activity of the whistle blowing. We can't even imagine a race without it!

So, the runners are threads, the whistle being blown is a one-time activity, such as initialization, and the goal post is, again, an application-specific goal each thread is trying to complete.

The following code shows the latch in action:

```
import java.util.concurrent.CountDownLatch;

public class AppCountDownLatch {
  public static void main(String[] args) throws InterruptedException {
    CountDownLatch latch = new CountDownLatch(3);
```

We have a main program, using a `countdownlatch` from Java's threading library. The latch is initialized to 3:

```
Runnable w1 = createWorker(3000, latch, "W1");
Runnable w2 = createWorker(2000, latch, "W2");
Runnable w3 = createWorker(1000, latch, "W3");
```

We create three runnables, *w1*, *w2*, and *w3*. Each runnable is sent the millisecond it needs to sleep, a name to identify it, and the latch.

The general idea is that every thread will put down the latch, once it is done with its processing. In this case, it is just sleeping for some time! We start off all three threads as follows:

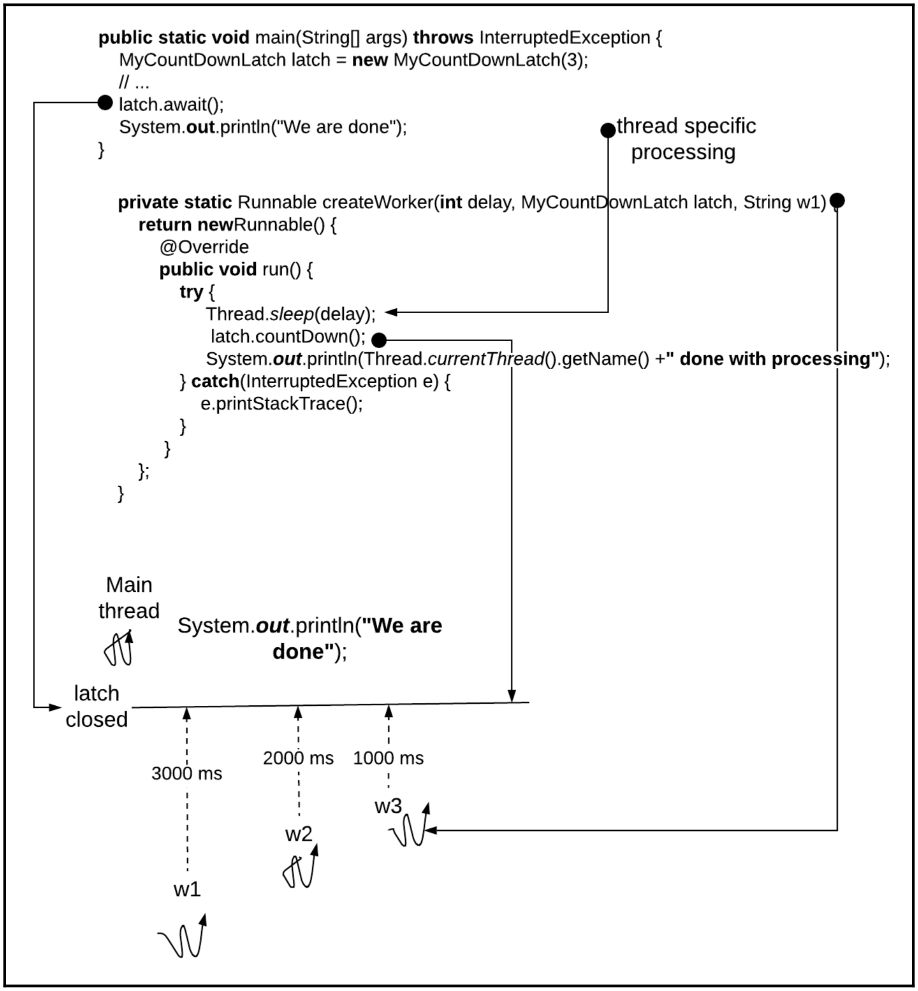
```
new Thread(w1).start();
new Thread(w1).start();
new Thread(w1).start();

latch.await(); // await it to open
System.out.println("We are done");
```

We need to make sure *all these threads complete* before the main thread exits. So, we wait for the latch to open, that is, we wait for all the threads to complete their respective processing:

```
private static Runnable createWorker(int delay, CountDownLatch latch,
String w1) {
  return new Runnable() {
    @Override
    public void run() {
      try {
        Thread.sleep(delay);
        latch.countDown(); // decrement the latch
```

```
            System.out.println(Thread.currentThread().getName()
            + " done with processing");
          }
          catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      };
    }
```

As expected, the *"We are done"* message comes last, making sure all threads are done. The following is a pictorial representation:



```
public static void main(String[] args) throws InterruptedException {
   MyCountDownLatch latch = new MyCountDownLatch(3);
   // ...
   latch.await();
   System.out.println("We are done");
}

   private static Runnable createWorker(int delay, MyCountDownLatch latch, String w1)
      return newRunnable() {
         @Override
         public void run() {
            try {
               Thread.sleep(delay);
                latch.countDown();
               System.out.println(Thread.currentThread().getName() +" done with processing");
            } catch(InterruptedException e) {
               e.printStackTrace();
            }
         }
      };
   }
```

thread specific processing
Main thread
System.out.println("We are done");
latch closed
3000 ms
2000 ms
1000 ms
w3
w2
w1

Here, the one-time event is the completion of all the processing threads. Whichever thread decrements the latch to 0 opens it, so the main thread can continue. All it does here is print the done message, and then it exits.

Implementing the countdown latch

We can implement the latch as follows. We call the class, `MyCountDownLatch`:

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class MyCountDownLatch {
 private int cnt;
 private ReentrantLock lck;
 private Condition cond;
```

The field `cnt` holds the latch count. By now, the fields `lck` and `cond` should be very familiar to you. These are used to implement the synchronization semantics:

```
public MyCountDownLatch(int cnt) {
  this.cnt = cnt;
  lck = new ReentrantLock();
  cond = lck.newCondition();
}
```

The constructor initializes the latch count, `cnt`, and the `lck` and `cond` variables. The `cond` variable needs to be associated with the lock, `lck`:

```
public void await() throws InterruptedException {
 lck.lock();
 try {
 while (cnt != 0) {
 cond.await();
 }

 } finally {
 lck.unlock();
 }
}
```

The `await()` method checks whether the count, `cnt`, has gone down to 0. If so, the latch is open, and the method returns. If the count is non-zero, the calling thread is put to sleep, and the lock, `lck`, is released:

```
public void countDown() {
 lck.lock();
 try {
 --cnt;
 if (cnt == 0) {
 cond.signalAll();
 }
 } finally {
 lck.unlock();
 }
}
```

Finally, the `countDown()` method acquires the lock and decrements the count, `cnt`. If the count reaches 0, a broadcast happens on the condition variable.

This wakes up the pending threads (if any) waiting in the `await()` method.

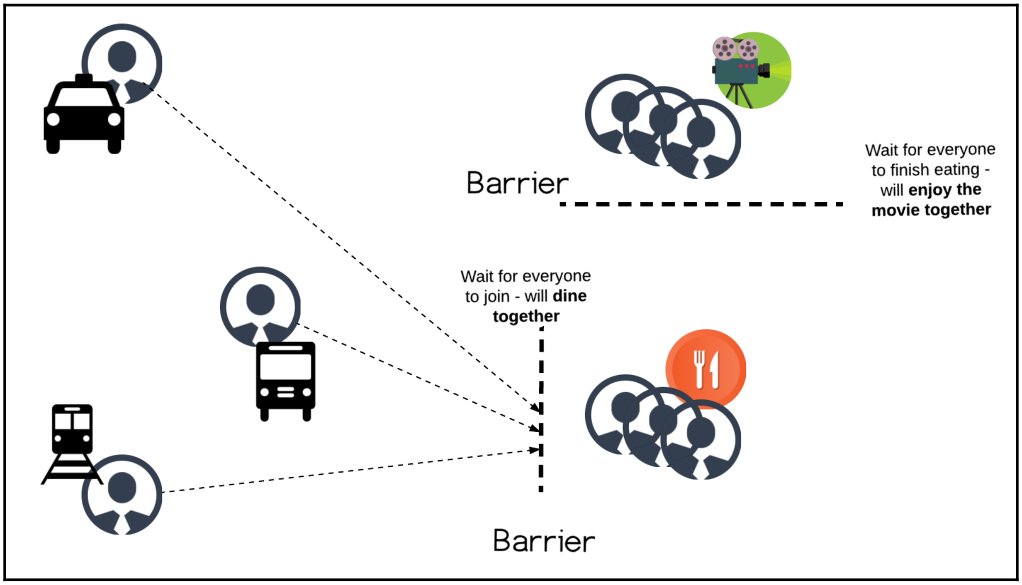
A cyclic barrier

A **cyclic barrier** is one more synchronization mechanism where all threads need to wait at a point before any can proceed.

A real-life example should make it clear. Three long-time friends happen to be in the same city on business, and they plan to have dinner together, and then go and see a movie, just to relive old times.

One of them is going to take a car, and they happen to be the nearest to the restaurant. So, they arrive quickly and *wait at the venue.* The other buddies are taking a train and a bus, respectively, so everyone waits for the other(s) to arrive.

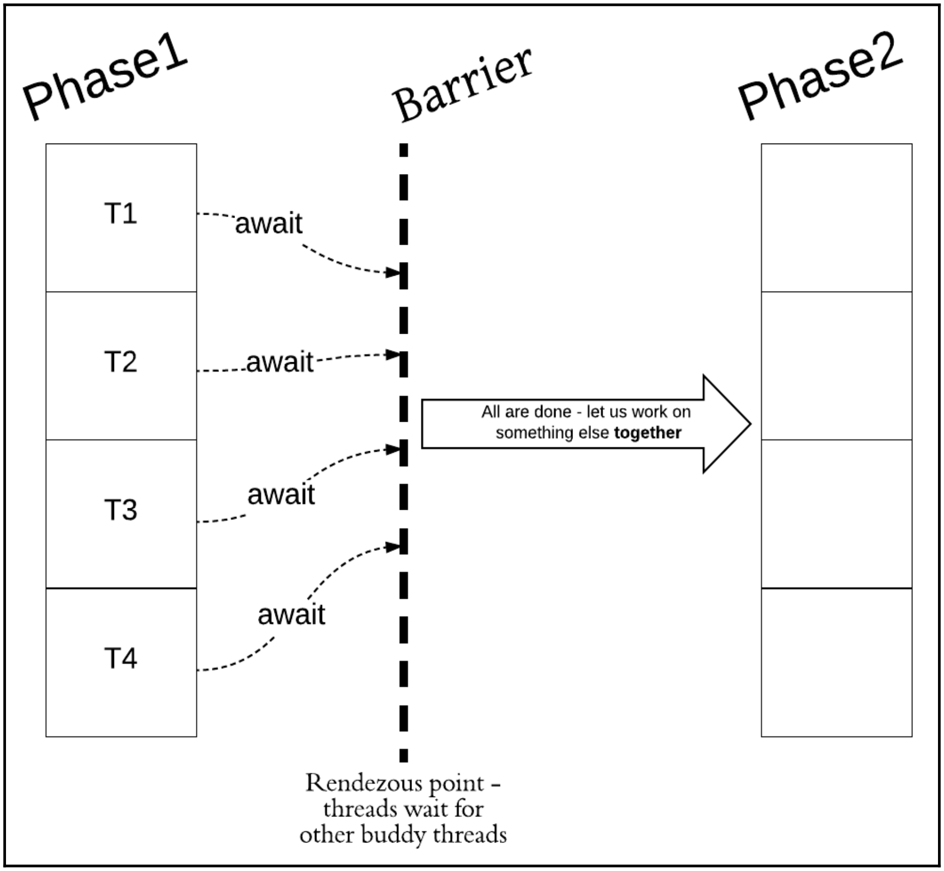
Once everyone arrives, the dinner can begin. This agreement of waiting upon other buddies to join, and then only begin the fun, is a *barrier:*


Barrier
Wait for everyone to finish eating - will **enjoy the movie together**
Wait for everyone to join - will **dine together**
Barrier

After the dinner is over, and as people eat at different rates, not everyone finishes at the same time. So, again, they wait (barrier), and then, once everyone is done, they go to the movie.

A barrier is very much like a countdown latch, with the only difference being that all threads instead wait for their other buddy threads to complete.

The following diagram shows a process divided up into *phases*. Each phase is worked upon by multiple threads. When all are done with their allotted task, the group moves on to the next phase:


Phase1
Barrier
Phase2
T1
T2
T3
T4
await
await
await
await
All are done - let us work on something else **together**
Rendezous point - threads wait for other buddy threads

Here is the example code showing a barrier in action:

```
public class AppCyclicBarrier {
  public static void main(String[] args) throws BrokenBarrierException,
InterruptedException {
    Runnable barrierAction = new Runnable() {
      public void run() {
        System.out.println("BarrierAction 1 executed ");
      }
    };

    CyclicBarrier barrier = new CyclicBarrier(3, barrierAction);
```

We create a `barrier` and install a `barrierAction`, a runnable that gets called when all threads have `awaited` on the barrier:

```
        Runnable w1 = createWorker(barrier);
        Runnable w2 = createWorker(barrier);

        new Thread(w1).start();
        new Thread(w2).start();

        barrier.await();
        System.out.println("Done");
    }
```

We have two threads, with runnables *w1* and *w2*. The main thread spawns them and then waits at the barrier:

```
    private static Runnable createWorker(final CyclicBarrier barrier) {
      return new Runnable() {

        @Override
        public void run() {
          try {
            Thread.sleep(1000);
            System.out.println("Waiting at barrier");
            try {
              barrier.await();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                  }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                  }

            }
      };
    }
```

Each *worker* thread just sleeps, prints a message, and then waits at the barrier. Once everyone has arrived (after they have done their respective processing), the barrier opens, and all threads proceed to completion.

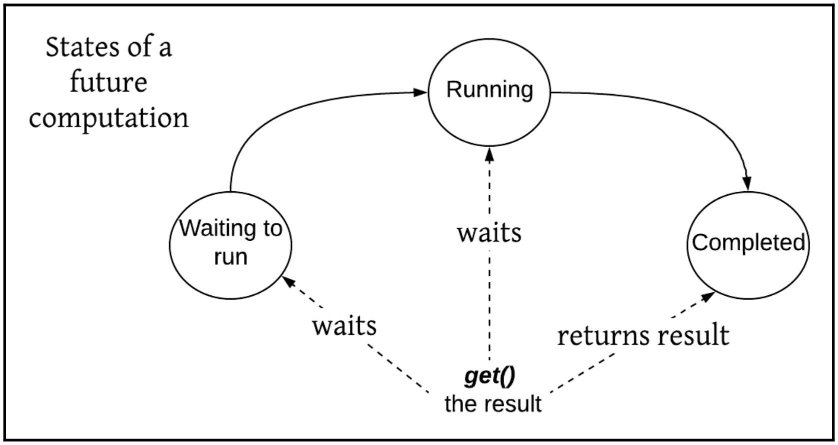
Implementing your own version of barriers is left as an exercise.

A future task

A **future task** is essentially an *asynchronous* construct. As its name signifies, it is a wrapper for some *expensive* computation whose result will be available sometime in the *future*. Once the computation is completed, it returns the result via a `get()` method call.

As listed and shown in the figure, the future task have three states:

1. Waiting to run
2. Running
3. Completed



When we try to obtain a result from the future task, the behavior depends on which state it is in. If it is not completed yet, the thread calling `get()` sleeps till the computation is done, and the result is available. There is also an overloaded `get(long timeout, TimeUnit unit)` method that avoids waiting forever. So, *task* `get(5L, TimeUnit.SECONDS)` would wait for only five seconds. If the method does not return within the time prescribed, a `TimeOutException` will be thrown.

The following code shows the future task in action:

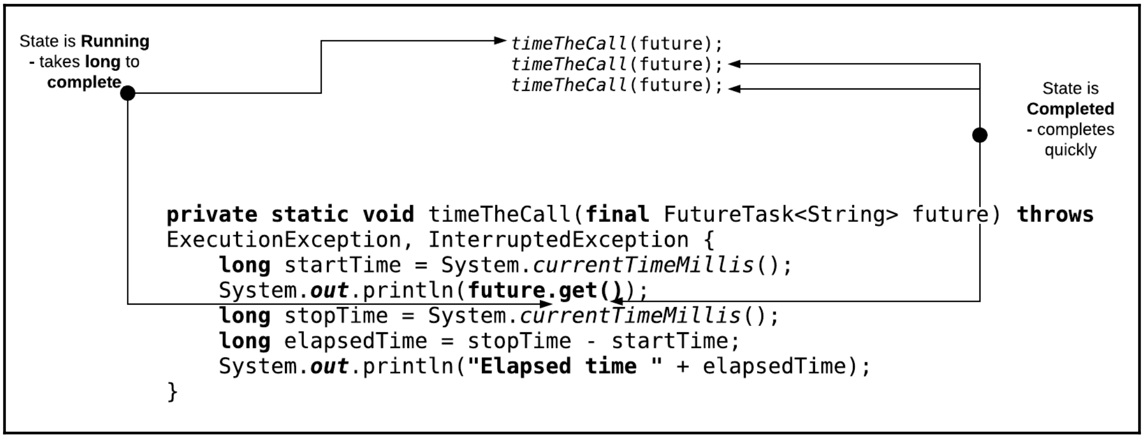
```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class AppFutureTask {
  private static FutureTask<String> createAFutureTask() {
    final Callable<String> callable = new Callable<String>() {
      @Override
      public String call() throws InterruptedException {
        Thread.sleep(4000);
        return "Hello World";
      }
    };
    return new FutureTask<String>(callable);
}
```

The `createAFutureTask()` method creates a future task, where the expensive computation is represented by a c*allable.*

This callable's `call()` method makes the thread sleep for four seconds, and then returns a string result. We wrap this computation in a future task and return it:

```
private static void timeTheCall(final FutureTask<String> future) throws
ExecutionException, InterruptedException {
  long startTime = System.currentTimeMillis();
  System.out.println(future.get());
  long stopTime = System.currentTimeMillis();
  long elapsedTime = stopTime - startTime;
  System.out.println("Elapsed time " + elapsedTime);
}
```

The preceding method is just a handy helper; it times the `future.get()` method invocations:


State is **Running** - takes **long** to **complete**
State is **Completed** - completes quickly

```
timeTheCall(future);
timeTheCall(future);
timeTheCall(future);
```

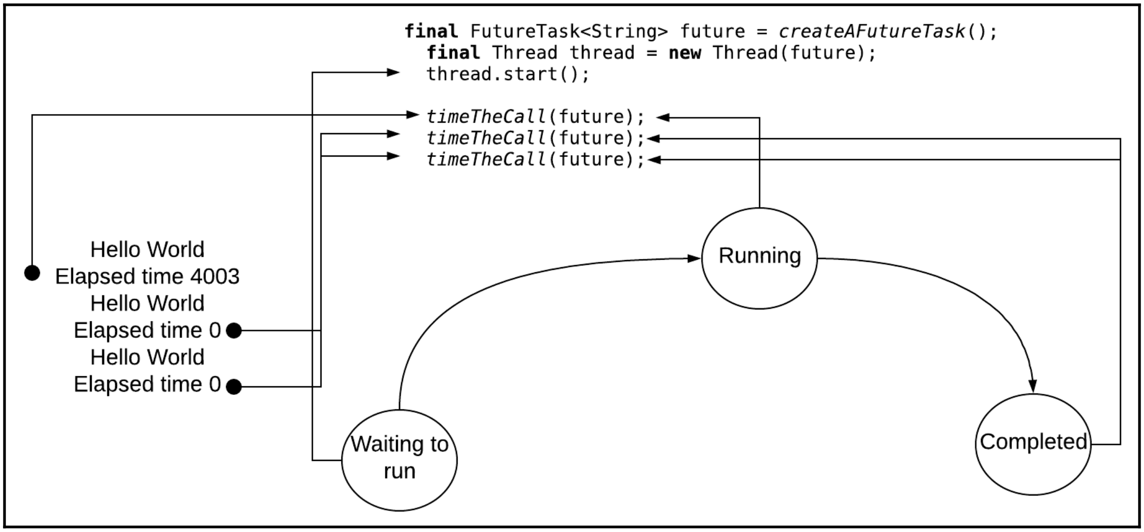
```
private static void timeTheCall(final FutureTask<String> future) throws
ExecutionException, InterruptedException {
    long startTime = System.currentTimeMillis();
    System.out.println(future.get());
    long stopTime = System.currentTimeMillis();
    long elapsedTime = stopTime - startTime;
    System.out.println("Elapsed time " + elapsedTime);
}
```

As shown in the preceding image, the first time we call the `future.get()` method (invoked as a result of the `timeTheCall(future)` method), the future is in the running state. So, the result is not available as of yet. So, the first call takes a long time to finish.

The second and subsequent calls, however, complete instantly, as the future is in the completed state. The result is readily available and so is returned quickly:

```
    public static void main(String[] args) throws ExecutionException,
  InterruptedException {

      final FutureTask<String> future = createAFutureTask();
      final Thread thread = new Thread(future);
      thread.start();

      timeTheCall(future);
      timeTheCall(future);
      timeTheCall(future);
    }
  }
```

The shown method is a driver that creates the future and starts the asynchronous computation:



```
final FutureTask<String> future = createAFutureTask();
  final Thread thread = new Thread(future);
  thread.start();

  timeTheCall(future);
  timeTheCall(future);
  timeTheCall(future);
```

```
Hello World
Elapsed time 4003
Hello World
Elapsed time 0
Hello World
Elapsed time 0
```

As shown in the preceding diagram, when we call `get` on a future in the w*aiting* or r*unning* state, the call completes in 4,003 milliseconds. The thread sleeps for 4,000 milliseconds, and the rest of the processing takes three milliseconds. The other two get calls complete instantly. The call completes so fast that millisecond granularity is not enough to catch it.

Your times may vary a bit. However, the overall behavior should be the same.

Writing a future implementation is left as an exercise.

Summary

We saw many primitive synchronizations in this chapter. We started with the bounded buffer and saw how it prevents an overloaded application from running out of memory. The client contract is realized using reentrant locks.

Next, we discussed the readers-writer locking; this is the pattern that increases read concurrency. We also looked at counting semaphores, countdown latches, barriers, and future tasks. We will be looking at the applications of these primitives in the upcoming chapters.

Resource pooling is realized using counting semaphores. For example, database connection pooling and thread pooling allow an application to pool and use resources efficiently.

Thread pools offer the same benefit for thread management. `java.util.concurrent` provides a flexible thread pool implementation as part of the executor framework.

We will take a detailed look at thread pooling in the next chapter.

4
Thread Pools

In this chapter, we will be looking at more patterns for staying away from explicit locking and state management. The theme is to let us focus on the business logic and the rest of the boilerplate of explicit thread creation and management handled by a framework.

This set of design patterns yields robust code as we *reuse* tried and tested, well-proven library code. We start with thread pooling as a major step toward focusing on our business logic as *tasks.* The pooling patterns give us a facility to run these tasks concurrently.

Firstly, we will cover the need for thread pools and the notion of a task. A *task* is a manifestation of the *command design pattern,* decoupling the task definition from its execution. We then look at **ExecutorService**, the pooling facility given by Java's threading library. Blocking queues are at the heart of this implementation. We will use the blocking queues to home grow our own pooling implementation.

Fork-Join is a major pooling implementation that appeared in Java 7. This is a dynamic thread pool that takes into account the number of cores and the task load. This is also the *default dispatcher* for the actor systems we will cover in subsequent chapters.

We will look at this API and see the important concept of *work stealing.*

Finally, we look at the active object design pattern in depth. We will wrap up with a discussion and code implementation of this design pattern.

So, we will be discussing the following topics:

- More patterns for staying away from explicit locking and state management
- Thread pooling— need and notion
- Executor service
- Fork-join
- Active object design pattern

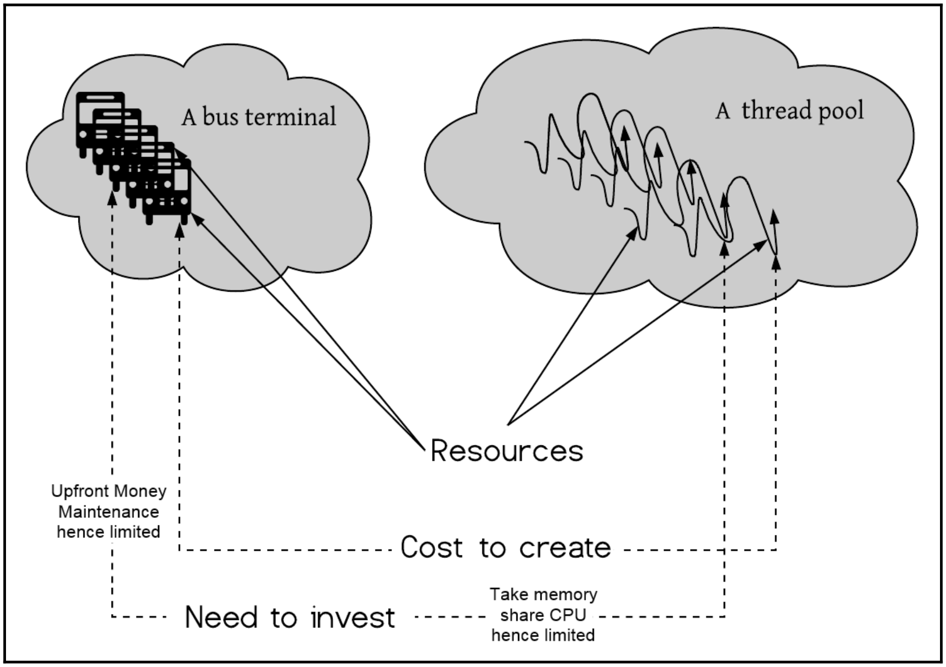
For complete code files you can visit `https://github.com/PacktPublishing/Concurrent-Patterns-and-Best-Practices`

Thread pools

What are *thread pools*? Why do we need these? Let's take a real-life example, consider a bus terminal that has a certain number of buses in the pool. The buses are added *once* to the pool and then are *reused* as required to serve various routes.

Why does the bus terminal work this way? Why don't they buy buses as needed and discard (sell them) as per their demand? To buy a bus, you need to shell out money. There is an additional cost for maintaining them!

So, the designers take a call, take into account the average travelers using the service, arrive at a certain number of buses, and reuse them as much as possible to maximize the *return on investment:*



Drawing a parallel, threads are expensive to create. So, we need to limit the number of threads in an application. Every thread has a stack of its own, which takes up memory. Each Java thread maps to an operating system thread. Thus, creation involves a *system call*, which is expensive. See this `stack overflow link` for more information.

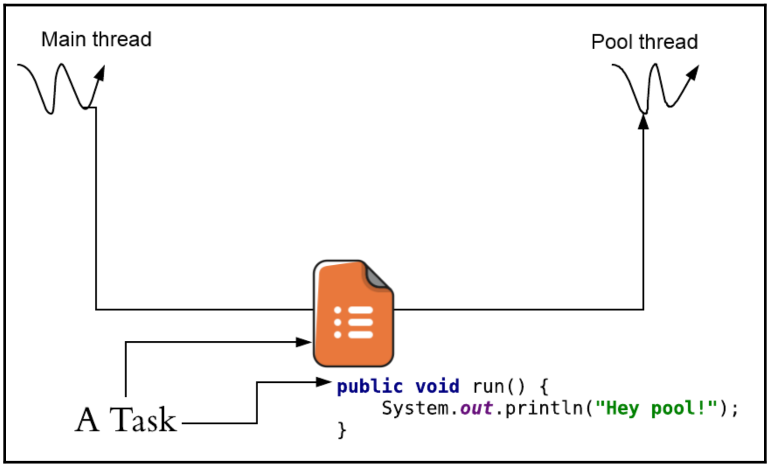
What is the alternative? Instead of spawning a new thread, we can use a thread pool.

Multi-threaded servers typically use thread pools. Each client connection hitting at the server is wrapped as a task and serviced via a thread pool. Java 5 comes with built-in thread pools in the `java.util.concurrent` package.

The following code shows such a pool in action. A client thread *delegates* a task to the service, which executes it in the background:

```
public static void main(String[] args) {
  ExecutorService executorService = Executors.newFixedThreadPool(10);

  executorService.execute(new Runnable() {
    public void run() {
      System.out.println("Hey pool!");
    }
  });

  executorService.shutdown();
}
```

The `ExecutorService` is an interface. It sends the task to the pool; an ideal pool thread picks up and executes the task. This diagram shows what happens behind the scenes:

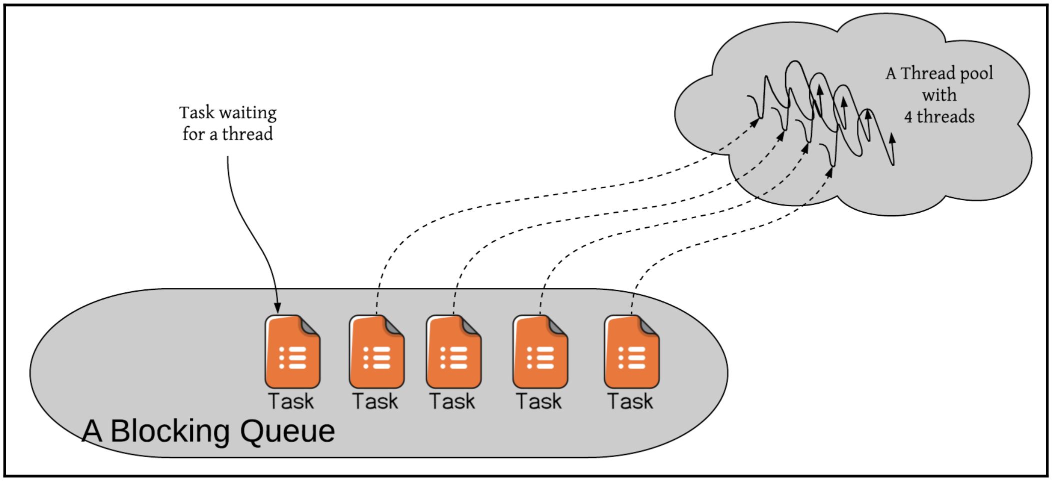


```
public void run() {
    System.out.println("Hey pool!");
}
```

How is the task shared, though? There must be a thread-safe channel via which the main thread sends over the task to the pool thread.

The channel is a blocking queue; tasks are enqueued by the client(s) and are dequeued by the threads in the pool. An ideal thread picks up the inserted task and executes it. Other idle threads in the pool will wait for new tasks to come.

This diagram shows this in action:



The pool is shown to have four threads. Each thread picks up the tasks and gets busy executing them. The last task waits in the queue for a thread to free up!

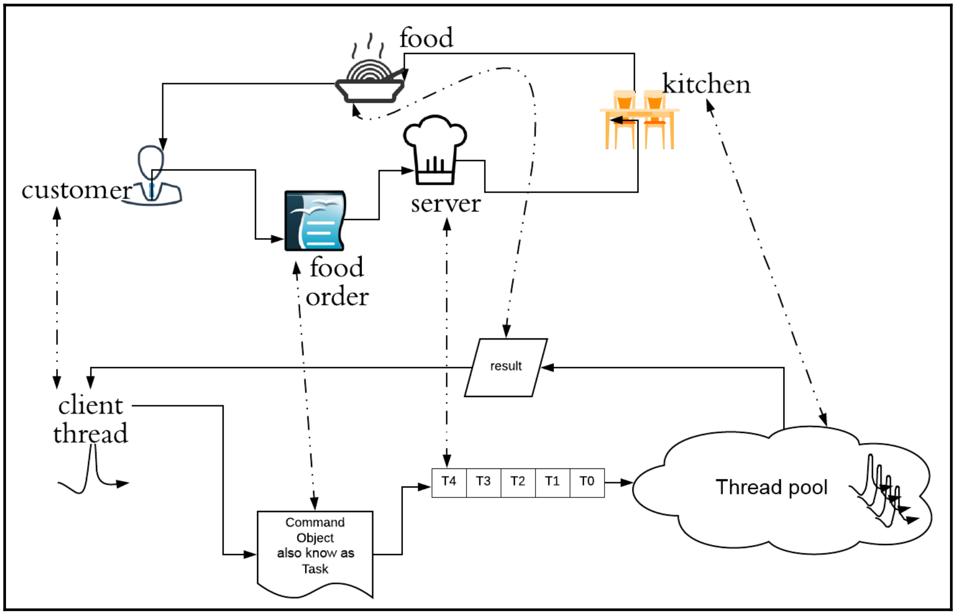
The command design pattern

What do we mean by decoupling a task definition from its execution? This is a very important theme. Here is a real-life example.

The upper part of the following diagram shows how a typical dine-in restaurant food order is processed. The customer places the order with the server, who notes it down. The *order* is then passed on to the kitchen for *execution*, that is, preparing the food dishes.

Once the food is ready, it is served to the customer. Note that the customer really does not know (or care), where the kitchen is located or who really prepared the dish for them! This is *decoupling* in action. The customer does not know who made the dish, just that the dish is made nice and tasty, and the kitchen chef does not know which customer the dish is being made for!

The chef tries to make all dishes tasty and within a *certain time frame,* and that is all they worry about:



The preceding diagram also shows how the command pattern works when applied to thread pools and draws a parallel with the restaurant entities.

A client thread corresponds to the customer. It creates a runnable as a command (whose *run* method expresses the task). This corresponds to the jotted-down food order. The task is inserted into a First-In/First-Out queue. The queue roughly corresponds to the server.

The kitchen corresponds to a thread pool; a thread is like a chef who works on the r*unnable* command. Eventually, the result is computed, which corresponds to the food. The result is routed back to the client thread.

Counting words

The following is the ubiquitous word-counting program. It reads a text file and counts the number of words in it. For simplicity, words are strings separated with a white space.

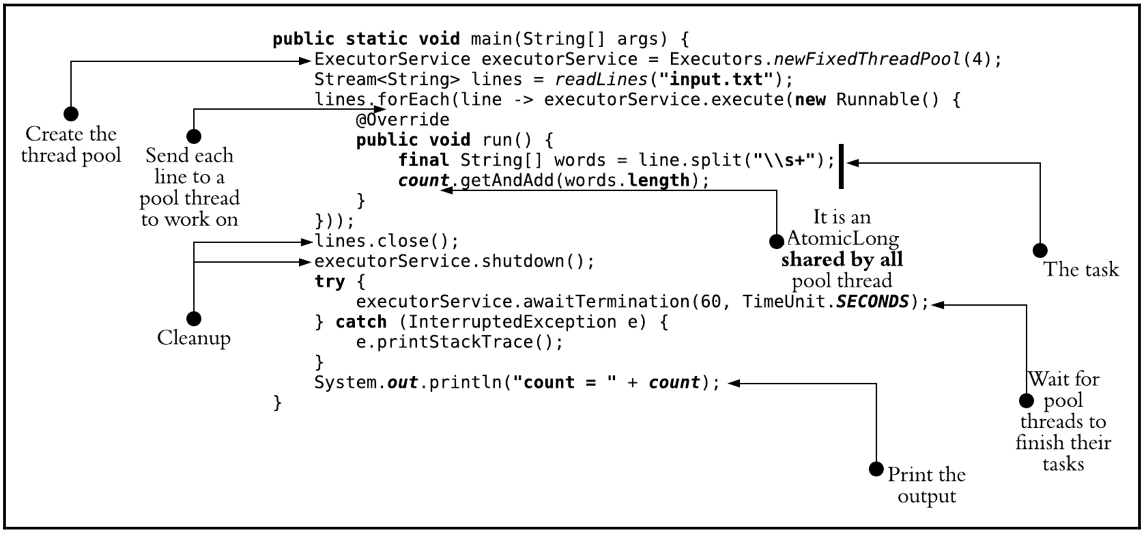
The driver reads each line and sends it to the thread pool. Each thread counts the words by incrementing a *shared* count variable. This code snippet shows the starting part. We will discuss the code piecemeal to see how everything fits together:

```
public class WordCount {
  final static AtomicLong count = new AtomicLong();
```

Next, comes a helper method that reads a file and returns a *Stream<String>*. This is a boilerplate function we use for other versions of the program too:

```
private static Stream<String> readLines(String fileName) {
  Path path = null;
  Stream<String> lines = null;
  try {
    path =
Paths.get(Thread.currentThread().getContextClassLoader().getResource(fileNa
me).toURI());
    lines = Files.lines(path);
  } catch (URISyntaxException | IOException e) {
      throw new RuntimeException(e);
    }
    return lines;
}
```

As shown in the preceding code, the file is read from the `resources` folder and a stream of the lines is returned. The following code snippet shows how the driver reads the lines and passes each to the thread pool:



```
public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(4);
    Stream<String> lines = readLines("input.txt");
    lines.forEach(line -> executorService.execute(new Runnable() {
        @Override
        public void run() {
            final String[] words = line.split("\\s+");
            count.getAndAdd(words.length);
        }
    }));
    lines.close();
    executorService.shutdown();
    try {
        executorService.awaitTermination(60, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("count = " + count);
}
```

Create the thread pool
Send each line to a pool thread to work on
It is an AtomicLong **shared by all** pool thread
The task
Cleanup
Wait for pool threads to finish their tasks
Print the output

We have a threadpool of four threads. We install a *runnable* that just splits the line and increments the `count` field. Once the stream is done, we close the stream and the `executorService`.

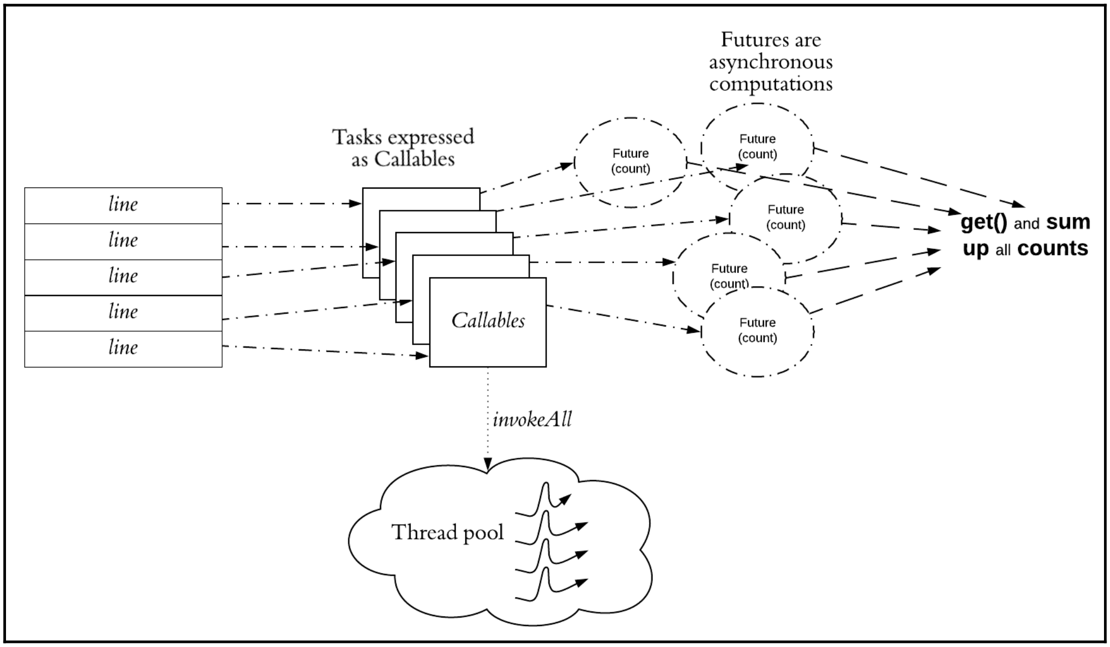
The `waitTermination(...)` call waits for all the pool threads to finish execution.

Another version

The aforementioned version is updating the `count` variable, which is a *shared global state.* What if each thread returns its count and the upper layer could sum that up instead?

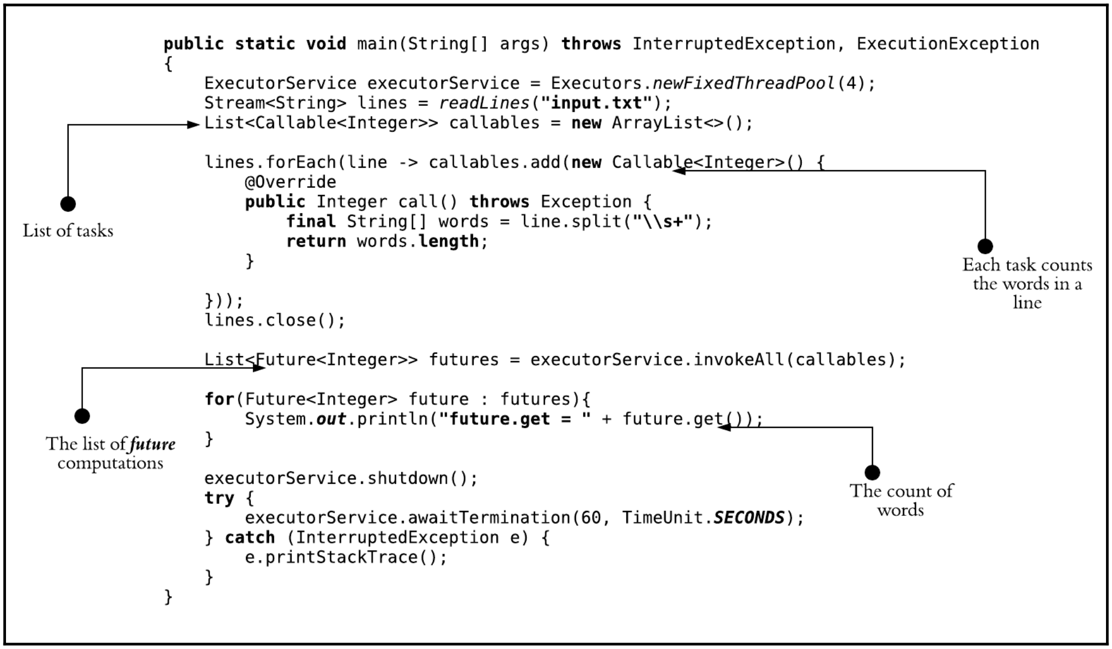
The problem with this approach is that, you cannot return a value from the `run()` method. The method that cannot return anything is declared as a `void`.

In such cases, the `Callable` interface comes in handy. The following version uses a list of callables, upon completion, and each callable returns a *future:*



The *future* helps avoid blocking. We fire off all the computations, and then finally call each future's `get()` method. This method *could block* (if the computation is a long-running one).

The following code shows the future-based version:



```
public static void main(String[] args) throws InterruptedException, ExecutionException
{
    ExecutorService executorService = Executors.newFixedThreadPool(4);
    Stream<String> lines = readLines("input.txt");
    List<Callable<Integer>> callables = new ArrayList<>();

    lines.forEach(line -> callables.add(new Callable<Integer>() {
        @Override
        public Integer call() throws Exception {
            final String[] words = line.split("\\s+");
            return words.length;
        }

    }));
    lines.close();

    List<Future<Integer>> futures = executorService.invokeAll(callables);

    for(Future<Integer> future : futures){
        System.out.println("future.get = " + future.get());
    }

    executorService.shutdown();
    try {
        executorService.awaitTermination(60, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

List of tasks
Each task counts the words in a line
The list of *future* computations
The count of words

As it stands, the code does not sum up the counts. Rather, it shows the individual word count of each line. This allows us to verify that the callable or future combination works as expected.

The blocking queue

The blocking code implements a thread pool; we just show the relevant part that is changing; namely, the pool implementation used is our own.

The driver is mostly unchanged from what we mentioned previously:

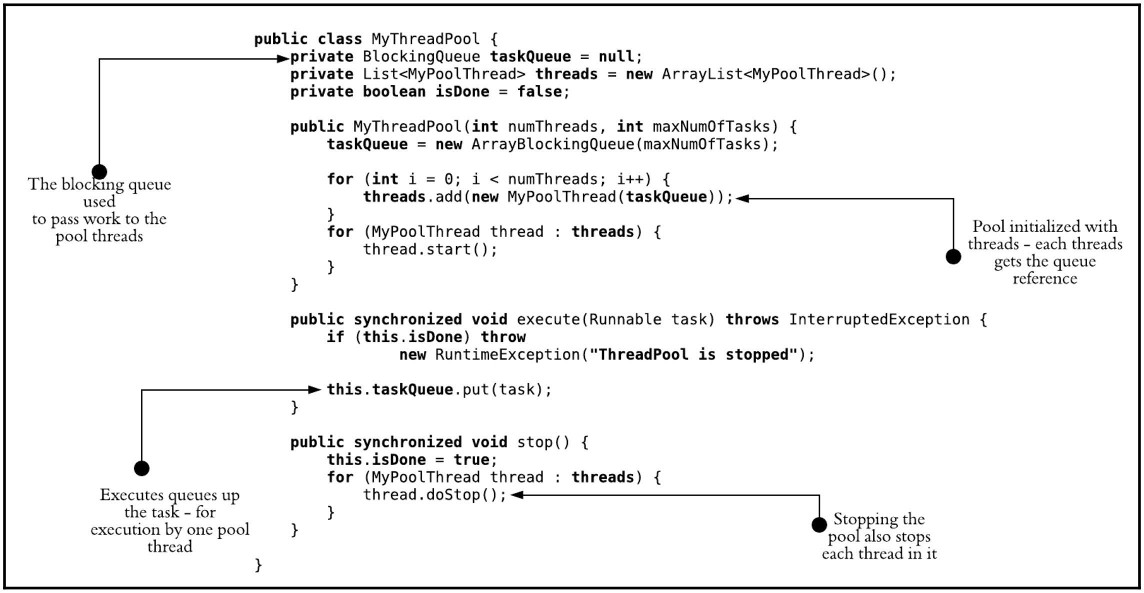
```
public static void main(String[] args) {
  MyThreadPool threadPool = new MyThreadPool(4, 20);
  Stream<String> lines = readLines("input.txt");

  lines.forEach(line -> {
    try {
      threadPool.execute(new Runnable() {
        @Override
        public void run() {
          final String[] words = line.split(" ");
          count.getAndAdd(words.length);
        }
      });
  } catch (InterruptedException e) {
            e.printStackTrace();
    }
  });
  lines.close();
  threadPool.stop();
  System.out.println("count = " + count);
}
```

Let's dissect the `MyThreadPool` class. We use an `ArrayBlockingQueue` to pass the tasks to the pool. Go through the following line:

```
taskQueue = new ArrayBlockingQueue(maxNumOfTasks);
```

It initializes the queue to hold a certain number of items (its capacity), which is 20 in our driver code. If some producer tries to put more than the queue capacity, then the producer blocks it.

Limiting the number of tasks keeps the queue in control. We have seen this theme earlier: the producer is made to wait, in case the queue is full:



```
public class MyThreadPool {
    private BlockingQueue taskQueue = null;
    private List<MyPoolThread> threads = new ArrayList<MyPoolThread>();
    private boolean isDone = false;

    public MyThreadPool(int numThreads, int maxNumOfTasks) {
        taskQueue = new ArrayBlockingQueue(maxNumOfTasks);

        for (int i = 0; i < numThreads; i++) {
            threads.add(new MyPoolThread(taskQueue));
        }
        for (MyPoolThread thread : threads) {
            thread.start();
        }
    }

    public synchronized void execute(Runnable task) throws InterruptedException {
        if (this.isDone) throw
              new RuntimeException("ThreadPool is stopped");

        this.taskQueue.put(task);
    }

    public synchronized void stop() {
        this.isDone = true;
        for (MyPoolThread thread : threads) {
            thread.doStop();
        }
    }
}
```

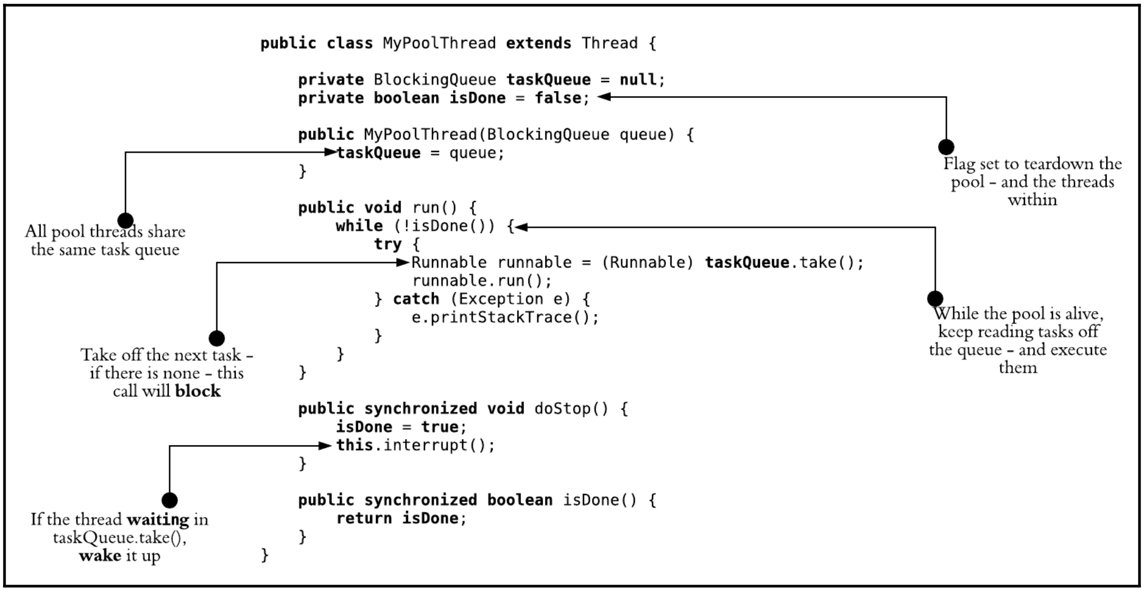
The blocking queue used to pass work to the pool threads

Pool initialized with threads - each threads gets the queue reference

Executes queues up the task - for execution by one pool thread

Stopping the pool also stops each thread in it

We use a `MyPoolThread` class, which subclasses the `Thread` class. Once the threads are added to the pool, each thread's `start()` method is called:



```
public class MyPoolThread extends Thread {

    private BlockingQueue taskQueue = null;
    private boolean isDone = false;

    public MyPoolThread(BlockingQueue queue) {
        taskQueue = queue;
    }

    public void run() {
        while (!isDone()) {
            try {
                Runnable runnable = (Runnable) taskQueue.take();
                runnable.run();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public synchronized void doStop() {
        isDone = true;
        this.interrupt();
    }

    public synchronized boolean isDone() {
        return isDone;
    }
}
```

Flag set to teardown the pool - and the threads within

All pool threads share the same task queue

While the pool is alive, keep reading tasks off the queue - and execute them

Take off the next task - if there is none - this call will **block**

If the thread **waiting** in taskQueue.take(), **wake** it up

Note the `doStop()` method; we set the `isDone` flag to true and then *interrupt* the thread! Note carefully that just setting the flag is *not enough.* The thread is probably waiting on the queue for any future work and none is going to come (as the `doStop()` method was invoked).

Note the overridden `run()` method:

```
public void run() {
  while (!isDone()) {
    try {
      Runnable runnable = (Runnable) taskQueue.take();
      runnable.run();
        } catch (Exception e) {
            e.printStackTrace();
          }
  }
}
```

The method keeps looping, taking a task off the queue and invoking its `run` method. The loop is exited upon the `isDone` flag being set to true.

It is possible that the thread is waiting in the take() call and could conceivably wait forever; it will never see the change of the flag value!

This is why we need to interrupt the thread at the time we set the flag, as follows:

```
public synchronized void doStop() {
  isDone = true;
  this.interrupt(); //break pool thread out of dequeue() call.
}
```

The thread throws an InterruptedException, breaks out of the taskQueue.take() method, notes the flag change, and exits the run() method.

This cleans up the thread, and the pool exits cleanly.

Thread interruption semantics

The interrupt() method *cancels* the current thread operation. The operation needs to be designed for interruption, though!

One common example of such an operation is the Thread.sleep(...) method. The following code shows how a sleeping thread is woken up using the interrupt() method:

```
public class ThreadInterruption {
  public static void main(String[] args) throws InterruptedException {
    Runnable r = new Runnable() {
      @Override
      public void run() {
        try {
          Thread.sleep(40000);
             } catch (InterruptedException e) {
                 System.out.println("I was woken up!!!");;
               }
        System.out.println("I Am done");
      }
    };
    Thread t = new Thread(r);
    t.start();
    t.interrupt();
    Thread.sleep(1000);
    }
}
```

The run() method just sleeps for 40 seconds (the quantum is in milliseconds). The main thread spawns the sleeping thread and then interrupts it.

As noted before, the `sleep()` method is aware of interruptions, it wakes up accordingly, prints a message, and exits.

The reason we should never ignore interruption should be clear by now. If we ignore the interruption for our thread pool, the pool thread will keep on waiting forever!

The fork-join pool

Java 7 introduced a specialized executor service, namely, the f*ork-join* API. It dynamically manages the number of threads, based on the available processors, and other parameters such as the number of concurrent tasks. It also employs an important pattern, *work stealing*—we will soon discuss this.

Egrep – simple version

Let's see the fork-join API in action. We will look at two examples to understand how the API works. The idea is to find a word in a text file. The driver class is `EgrepWord`:

```
public class EgrepWord {
  private final static ForkJoinPool forkJoinPool = new ForkJoinPool();
```

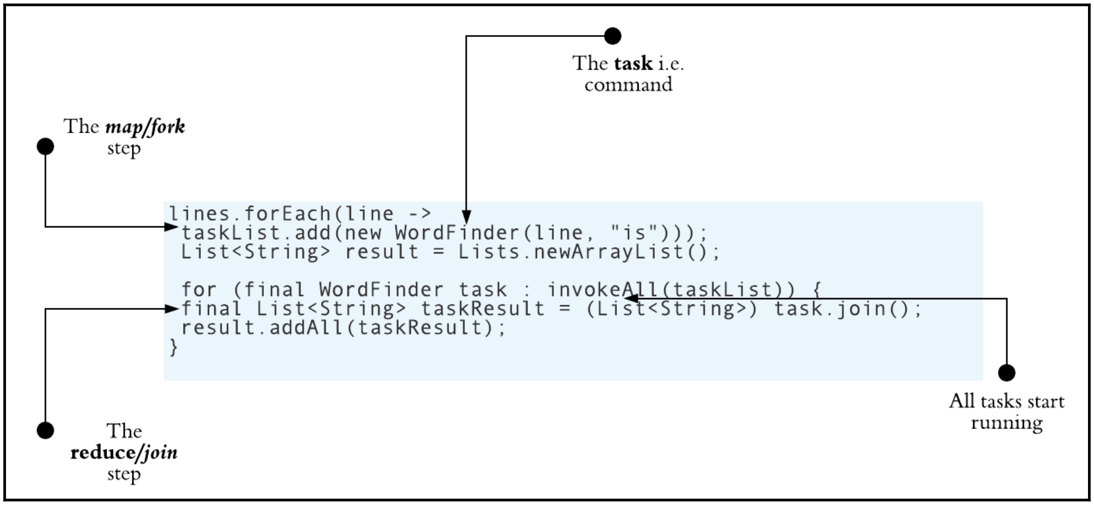
The principal theme in the fork-join API is a recursive task. The following class extends a parameterized `RecursiveTask`—on a *List<String>*. The following snippet shows the constructor that accepts a line of text and the word to search for:

```
private static class WordFinder extends RecursiveTask<List<String>> {
  final String line;
  final String word;

  private WordFinder(String line, String word) {
    this.line = line;
    this.word = word;
}
```

The driver comes next. It reads a file from the resource folder, and, assuming it is a text file, it goes and searches for the given word in it:

```
public static void main(String[] args) {
  Stream<String> lines = readLines("input.txt");
  List<WordFinder> taskList = new ArrayList<>();
  lines.forEach(line ->taskList.add(new WordFinder(line, "is")));
  List<String> result = new ArrayList<>();
    for (final WordFinder task : invokeAll(taskList)) {
      final List<String> taskResult = (List<String>) task.join();
      result.addAll(taskResult);
    }
    for(String r: result) {
      System.out.println(r);
    }

}
```

The following diagram shows the central concepts:



```
lines.forEach(line ->
 taskList.add(new WordFinder(line, "is")));
 List<String> result = Lists.newArrayList();

 for (final WordFinder task : invokeAll(taskList)) {
 final List<String> taskResult = (List<String>) task.join();
 result.addAll(taskResult);
}
```

The task is defined in terms of the `WordFinder` class. We will create a list of such tasks in the `taskList` *variable.* When `invokeAll` is called on this task list, all the tasks are *forked,* that is, they are *executed* by the pool threads.

Finally, all the task results are printed on a standard output. This prints all the lines in which we found the string *in*.

Why use a recursive task?

The tasks could *fork* more subtasks themselves . To look at this aspect, let's extend the preceding `egrep` program to allow a recursive grep—given a directory, the program will recursively find all occurrences of the word in all the files in the directory tree.

We change the previous code suitably, as shown here:

```
public class EgrepWord1 {
  private final static ForkJoinPool forkJoinPool = new ForkJoinPool();

  private static class WordFinder extends RecursiveTask<List<String>> {

    final File file;
    final String word;

    private WordFinder(File file, String word) {
      this.file = file;
      this.word = word;
    }
```

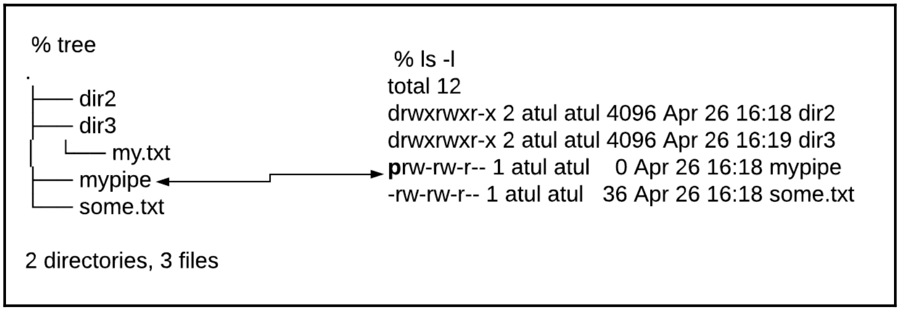
The task works on two kinds of files: if it is a directory, the task enters the directory and *spawns off more subtasks* to process each child entry (these, in turn, could be directories themselves):

```
@Override
protected List<String> compute() {
  if (file.isFile()) {
    return grepInFile(file, word);
}
```

If the file is really a text file, the `grepInFile(...)` method finds the word in the file. The helper method is shown as follows:

```
else {
  final File[] children = file.listFiles();
  if (children != null) {
```

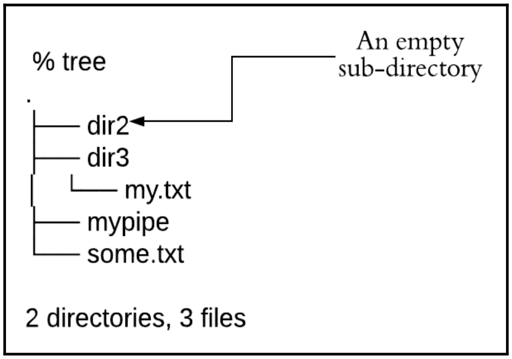
If the file is not a directory, then the `listFiles()` method returns a *null*. This case is shown as follows when the traversal hits the `mypipe` file which is a *fifo* (created using the `mkfifo` command), we then skip the entire processing:



```
% tree
.
├── dir2
├── dir3
│   └── my.txt
├── mypipe
└── some.txt

2 directories, 3 files
```

```
% ls -l
total 12
drwxrwxr-x 2 atul atul 4096 Apr 26 16:18 dir2
drwxrwxr-x 2 atul atul 4096 Apr 26 16:19 dir3
prw-rw-r-- 1 atul atul    0 Apr 26 16:18 mypipe
-rw-rw-r-- 1 atul atul   36 Apr 26 16:18 some.txt
```

Otherwise, it is a directory. The directory itself could be empty. In that case, the *children* array will also be empty. This is as given by the `File.listFiles(...)` contract. Again, in this case, we skip the subtask processing. This case is shown here:

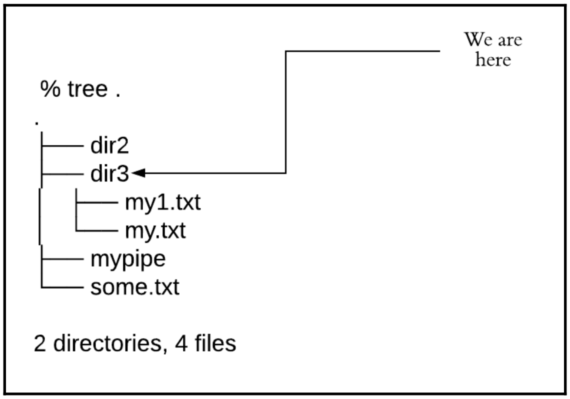


```
% tree
.
├── dir2
├── dir3
│   └── my.txt
├── mypipe
└── some.txt

2 directories, 3 files
```

Having checked these boundary conditions, we proceed to the nested processing of the directory:

```
List<ForkJoinTask<List<String>>> tasks = Lists.newArrayList();
List<String> result = Lists.newArrayList();
for (final File child : children) {
```

If the for loop is entered, the directory is not empty. The following diagram helps see this in context:



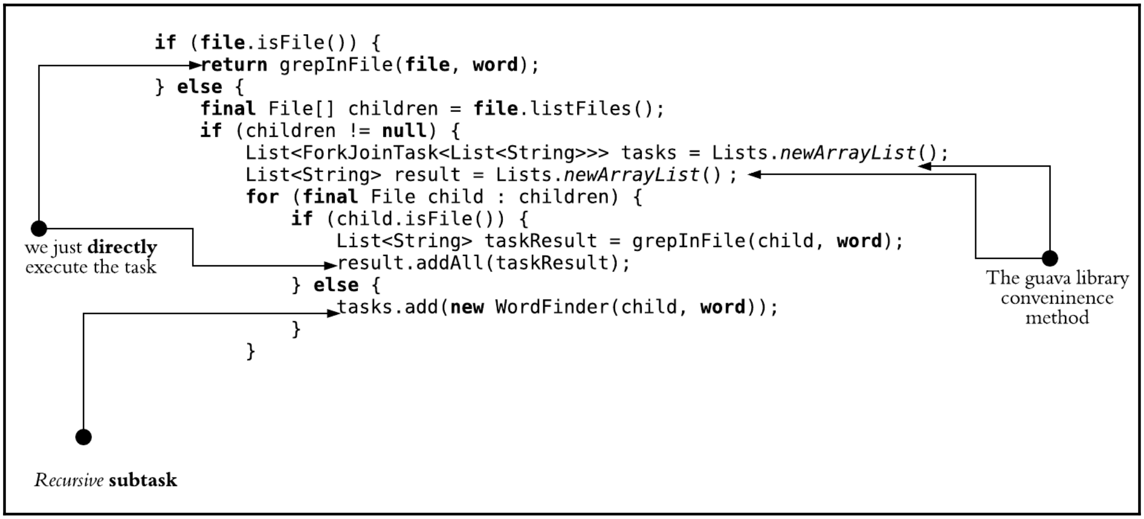
```
% tree .
.
├── dir2
├── dir3
│   ├── my1.txt
│   └── my.txt
├── mypipe
└── some.txt

2 directories, 4 files
```

The following code snippet is the heart of it:

```
if (child.isFile()) {
  List<String> taskResult = grepInFile(child, word);
  result.addAll(taskResult);
} else {
    tasks.add(new WordFinder(child, word));
}
```

If the child is a file, we just do the grep processing, and accumulate the result.

The following diagram helps us see how the tasks and subtasks are processed:



```
if (file.isFile()) {
    return grepInFile(file, word);
} else {
    final File[] children = file.listFiles();
    if (children != null) {
        List<ForkJoinTask<List<String>>> tasks = Lists.newArrayList();
        List<String> result = Lists.newArrayList() ;
        for (final File child : children) {
            if (child.isFile()) {
                List<String> taskResult = grepInFile(child, word);
                result.addAll(taskResult);
            } else {
                tasks.add(new WordFinder(child, word));
            }
        }
```

we just **directly** execute the task
The guava library conveninence method
Recursive **subtask**

Could we simplify the following snippet?

```
if (child.isFile()) {
  List<String> taskResult = grepInFile(child, word);
  result.addAll(taskResult);
} else {
    tasks.add(new WordFinder(child, word));
}
```

This is left as an exercise for the reader.

Finally, there is the grep processing; it is super simple. We create a stream of lines from the file:

```
private List<String> grepInFile(File file, String word) {
  final Stream<String> lines = readLines(file);
  final List<String> result = lines
    .filter(x -> x.contains(word))
    .map(y -> file + ": " + y)
    .collect(Collectors.toList());

 return result;
}
```

The stream is filtered based on whether the line contains the given word. For all such lines, it tacks on the filename and then generates the list using the stream's *collect* method:

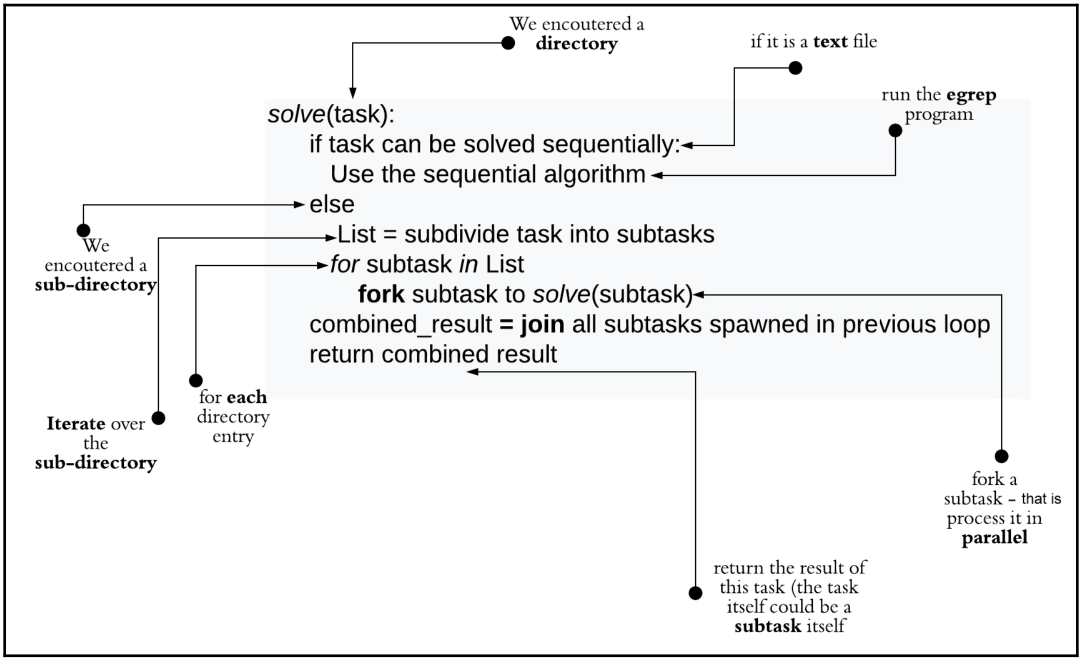
```
public static void main(String[] args) throws URISyntaxException {
  final Path dir1 = getResourcePath("dir1");
  final List<String> result = forkJoinPool.invoke(new
WordFinder(dir1.toFile(), "in"));

  for(String r: result) {
    System.out.println(r);
    }
}
```

The provided driver code drives the code and prints the result on the console.

Task parallelism

The fork-join is a *parallel* design pattern. We set up and execute it so that execution branches off in parallel when we encounter a directory. The result is *joined* (merged) with the higher level.

It is the *divide-and-conquer* strategy at work here. The following diagram shows the generalized theme (pattern) at work:

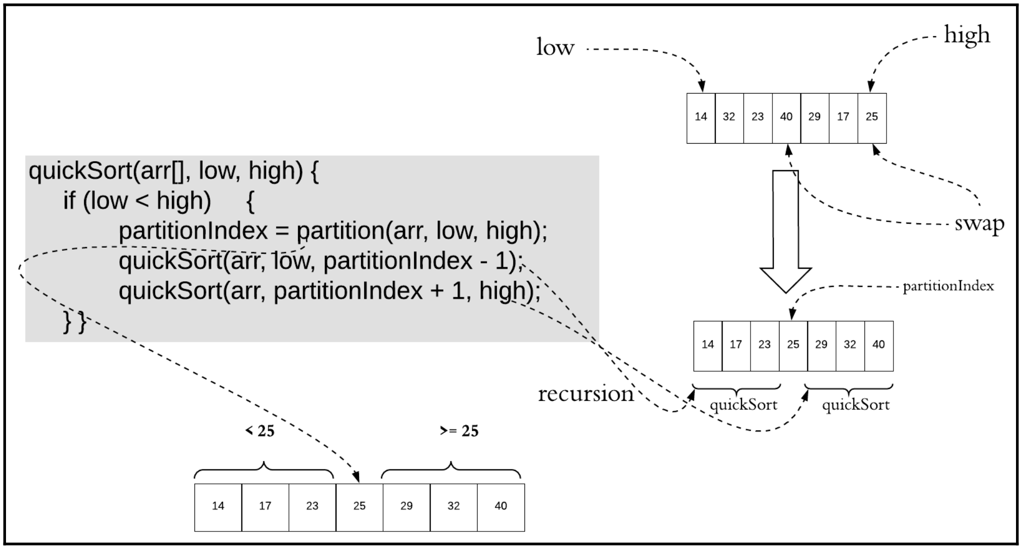


```
solve(task):
    if task can be solved sequentially:
        Use the sequential algorithm
    else
        List = subdivide task into subtasks
        for subtask in List
            fork subtask to solve(subtask)
    combined_result = join all subtasks spawned in previous loop
    return combined result
```

We encoutered a directory
if it is a text file
run the egrep program
We encoutered a sub-directory
Iterate over the sub-directory
for each directory entry
fork a subtask – that is process it in parallel
return the result of this task (the task itself could be a subtask itself

Fork-join will fit nicely to any processing that is amenable to the previous task or the subtask division.

One famous example of *divide and conquer* algorithms is *quicksort*. This famous algorithm sorts an array by dividing it up into two halves, using a median value. For more information on quicksort, please refer to `https://www.geeksforgeeks.org/quick-sort/` . This diagram shows the essence of this algorithm:



```
quickSort(arr[], low, high) {
    if (low < high)     {
            partitionIndex = partition(arr, low, high);
            quickSort(arr, low, partitionIndex - 1);
            quickSort(arr, partitionIndex + 1, high);
    } }
```

When the elements in the array are small, that is, around 100 or less, then the *divide and conquer* algorithm does not yield any benefits. In that case, we could sort the array using either a *selection sort* or an *insertion sort.*

The quicksort algorithm is amenable to a fork-join design. Could you spot the fork and join points? Give it a thought before you read on...

Quicksort – using fork-join

The following code snippet shows the driver for the quicksort implementation using the fork-join API. We use an array of 1,000 elements as input.

The array is populated using random numbers:

```
public class QuickSortForkJoin {
  public static final int NELEMS = 1000;
  public static void main(String[] args) {
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    Random r = new Random();

    int[] arr = new int[NELEMS];
    for (int i = 0; i < arr.length; i++) {
      int k = r.nextInt(NELEMS);
      arr[i] = k;
    }
    ForkJoinQuicksortTask forkJoinQuicksortTask = new
ForkJoinQuicksortTask(arr,0, arr.length - 1);
    final int[] result = forkJoinPool.invoke(forkJoinQuicksortTask);
      System.out.println(Arrays.toString(result));
    }
}
```

The following expression generates a random number between the *0 .. NELEMS* (*0* is inclusive, while *NELEMS* is exclusive):

```
int k = r.nextInt(NELEMS);
arr[i] = k;
```

We kick off the sorting by constructing an instance of a `ForkJoinQuicksortTask` class:

```
ForkJoinQuicksortTask forkJoinQuicksortTask = new
ForkJoinQuicksortTask(arr);
```

The constructor expects to be given the array. As we need to sort all of the array, the constructor can figure out its low and high bounds, which are 0 and `arr.length-1`:

```
final int[] result = forkJoinPool.invoke(forkJoinQuicksortTask);
System.out.println(Arrays.toString(result));
```

The result obtained by invoking the fork-join processing is printed to the console. We use a helper method of the `Arrays` class to print the sorted array contents.

The ForkJoinQuicksortTask class

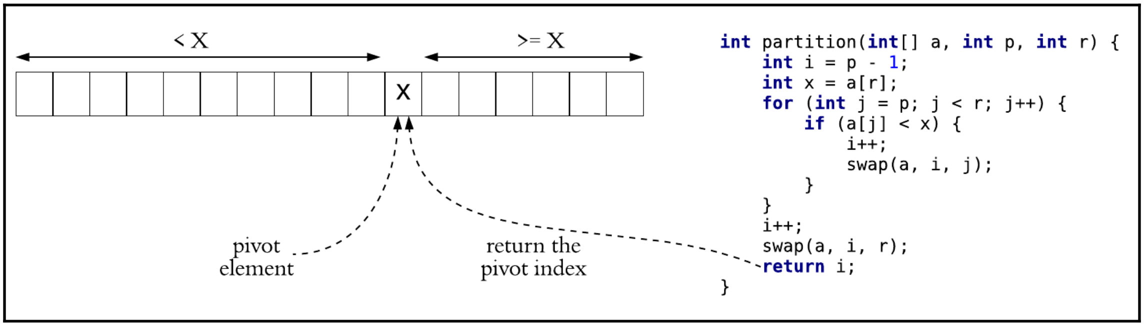
The class extends the `RecursiveTask`, as needed by the contract of the fork join API. The class joins and returns an array of int as follows:

```
class ForkJoinQuicksortTask extends RecursiveTask<int[]> {
    public static final int LIMIT = 100;
    int[] arr;
    int left;
    int right;

    public ForkJoinQuicksortTask(int[] arr) {
      this(arr, 0, arr.length-1);
    }

    public ForkJoinQuicksortTask(int[] arr, int left, int right) {
      this.arr = arr;
      this.left = left;
      this.right = right;
    }
```

As noted in the previous code, `quicksort` does not work too well on small arrays (when the number of elements is less than or equal to 100). The `LIMIT` constant defines the number of these elements when we stop the *divide and conquer* and fall back to other methods to sort the small array:

```
public static final int LIMIT = 100;
```

The array is partitioned around a pivot element, so that all elements less than the pivot element are to the left of it. Elements that are to the right of the pivot element are greater or equal to it. The essence of the pivot method is shown in the following diagram:



```
int partition(int[] a, int p, int r) {
    int i = p - 1;
    int x = a[r];
    for (int j = p; j < r; j++) {
        if (a[j] < x) {
            i++;
            swap(a, i, j);
        }
    }
    i++;
    swap(a, i, r);
    return i;
}
```

Taking a paper and pencil, and working out a trace of the `partition()` method would help you understand how the method works...

The method is shown here:

```
int partition(int[] a, int p, int r) {
  int i = p - 1;
  int x = a[r];
  for (int j = p; j < r; j++) {
    if (a[j] < x) {
    i++;
    swap(a, i, j);
    }
  }
  i++;
  swap(a, i, r);
  return i;
}
```

The swap method is just a helper method, which uses a temp variable to sort two elements of the array:

```
  void swap(int[] a, int p, int r) {
    int t = a[p];
    a[p] = a[r];
    a[r] = t;
  }

  private boolean isItASmallArray() {
    return right - left <= LIMIT;
  }
}
```

Given all of this background, let's look at the compute() method, which is the meat of our fork-join processing:

```
@Override
protected int[] compute() {
  if (isItASmallArray()) {
    Arrays.sort(arr, left, right + 1);
    return arr;
  } else {
      List<ForkJoinTask<int[]>> tasks = Lists.newArrayList();
      int pivotIndex = partition(arr, left, right);

      int[] arr0 = Arrays.copyOfRange(arr, left, pivotIndex);
      int[] arr1 = Arrays.copyOfRange(arr, pivotIndex + 1, right + 1);

      tasks.add(new ForkJoinQuicksortTask(arr0));
      tasks.add(new ForkJoinQuicksortTask(arr1));

      int[] result = new int[]{arr[pivotIndex]};
      boolean pivotElemCopied = false;
      for (final ForkJoinTask<int[]> task : invokeAll(tasks)) {
        int[] taskResult = task.join();
        if (!pivotElemCopied) {
        result = Ints.concat(taskResult, result);
        pivotElemCopied = true;
      } else {
          result = Ints.concat(result, taskResult);
        }
      }
        return result;
    }
}
```

The if clause calls the `isItASmallArray()`, checks whether the array is too small, and should instead fall back on other sorting methods. If it is, we just sort it using `Arrays.sort(...)` to keep it simple:

```
if (isItASmallArray()) {
  Arrays.sort(arr, left, right + 1);
  return arr;
    }
```

The `else` clause is more interesting. We create a list of recursive fork join tasks, all held in the `tasks` list:

```
else {
  List<ForkJoinTask<int[]>> tasks = Lists.newArrayList();
  int pivotIndex = partition(arr, left, right);
```

The array is partitioned into two and the `pivotIndex` is returned. The following line adds the two subarrays as fork join tasks:

```
int[] arr0 = Arrays.copyOfRange(arr, left, pivotIndex);
int[] arr1 = Arrays.copyOfRange(arr, pivotIndex + 1, right + 1);

tasks.add(new ForkJoinQuicksortTask(arr0));
tasks.add(new ForkJoinQuicksortTask(arr1));
```

We create two new subarrays by copying the appropriate ranges. This helps us use the `Ints.concat(...)` method to concatenate two arrays in the join phase. The `Ints.concat(...)` method comes from Google's excellent `Guava` library.

Why are we doing all of this? You guessed right: we are setting up the stage to sort both these subarrays *in parallel.* The remaining lines, as shown ahead, join the sorted subarrays:

```
int[] result = new int[]{arr[pivotIndex]};
boolean pivotElemCopied = false;
for (final ForkJoinTask<int[]> task : invokeAll(tasks)) {
  int[] taskResult = task.join();
  if (!pivotElemCopied) {
    result = Ints.concat(taskResult, result);
    pivotElemCopied = true;
  } else {
      result = Ints.concat(result, taskResult);
    }
}
return result;
```

We initialize the `result` variable as a single array element holding the pivot element. We append the pivot element to the first sorted array as a result. Next, we append the second sorted array to the result, giving us a complete sorted array.

The copy-on-write theme

Note that we do not change the source array; the sort does not happen *in place*. The input array is not modified; we instead return a sorted copy of the input array. The following line copies the subarray:

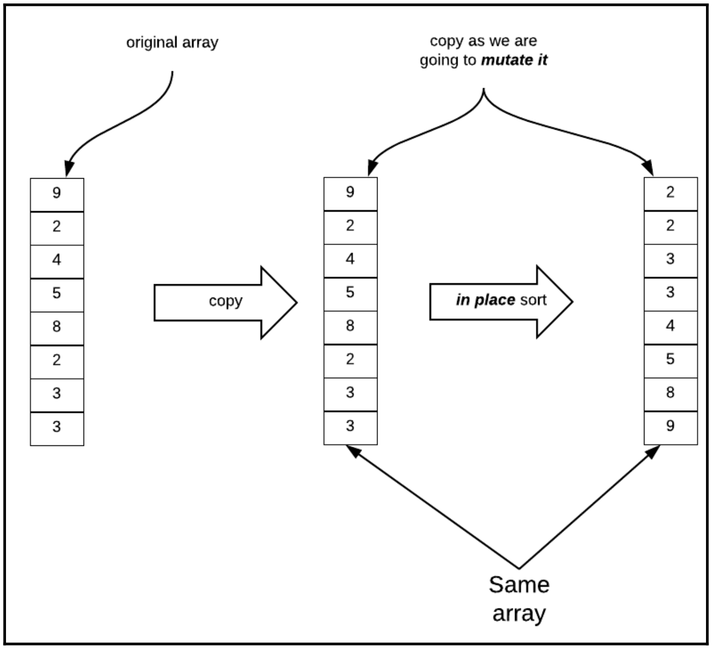
```
int[] arr0 = Arrays.copyOfRange(arr, left, pivotIndex);
```

The code then passes it on for further sorting. If we make sure never to *mutate the data structure* in place, then we don't need to *synchronize* it. Any number of threads can read it freely, knowing well in advance that no one can ever change it!

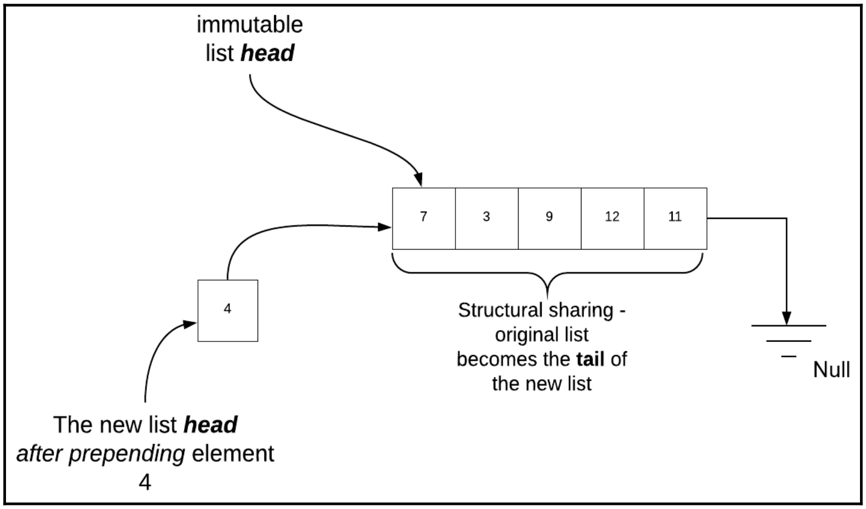
You may change the driver code to see this in action:

```
System.out.println(Arrays.toString(arr)); // [240, 565, 485, 357, 437,
316...
System.out.println(Arrays.toString(result)); // [0, 0, 1, 2, 4, 7, 8, 8,
10...
```

Of course, the previous scheme allocates many arrays. It would be easier to copy the entire array and then do the sort in place. The following diagram shows the scheme in action:



We need to balance a trade-off while making the array immutable, and we need to copy just enough so that unnecessary copying is avoided. In this case, of course, we have no option but to copy the entire array. However, as we will soon see, an immutable link list can offer a very efficient prepend operation, using *structural sharing:*


immutable list ***head***
7
3
9
12
11
4
Structural sharing - original list becomes the **tail** of the new list
Null
The new list ***head*** *after prepending* element 4

Using immutable data structures to represent a program state increases concurrency—threads do not have to synchronize, resulting in less contention and fewer bugs due to incorrect synchronization.

It also makes for easier reasoning about the programming flow.

In-place sorting

The following code shows an in-place version of quick sorting using the fork-join API. The following code also shows the relevant snippet and outline. The complete code is available in this book's code repository:

```
public class InPlaceFJQuickSort {

  public static final int NELEMS = 1000;

  public static void main(String[] args) {
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    Random r = new Random();

    int[] arr = new int[NELEMS];
      for (int i = 0; i < arr.length; i++) {
        int k = r.nextInt(NELEMS);
        arr[i] = k;
      }

    ForkJoinQuicksortAction forkJoinQuicksortAction = new
ForkJoinQuicksortAction(arr,0, arr.length - 1);
    forkJoinPool.invoke(forkJoinQuicksortAction);
    System.out.println(Arrays.toString(arr)); // The array is sorted, in
place
  }
}
```

The driver class uses the workhorse class, `ForJoinQuickSortAction`, which extends `RecursiveAction`. The overridden `compute(...)` method has a return type of `void`. This suits us nicely, as we need not return anything! The forked task changes the array *in place*. The class starts, as shown here:

```
class ForkJoinQuicksortAction extends RecursiveAction {
```

The following snippet shows the `compute(...)` method:

```
@Override
protected void compute() {
  if (isItASmallArray()) {
    Arrays.sort(arr, left, right + 1);
  } else {
      int pivotIndex = partition(arr, left, right);
      ForkJoinQuicksortAction task1 = new ForkJoinQuicksortAction(arr,
left, pivotIndex - 1);
      ForkJoinQuicksortAction task2 = new ForkJoinQuicksortAction(arr,
pivotIndex + 1, right);
      task1.fork();
      task2.compute();
      task1.join();
    }
}
```

The method is much simpler. We fork the left partition, so it is handled by a different thread, and `compute` the right partition. The line just waits for the left partition to get sorted. Other methods remain the same and hence are not shown. Using the *in-place* sorting version by copying the array is left as an exercise:

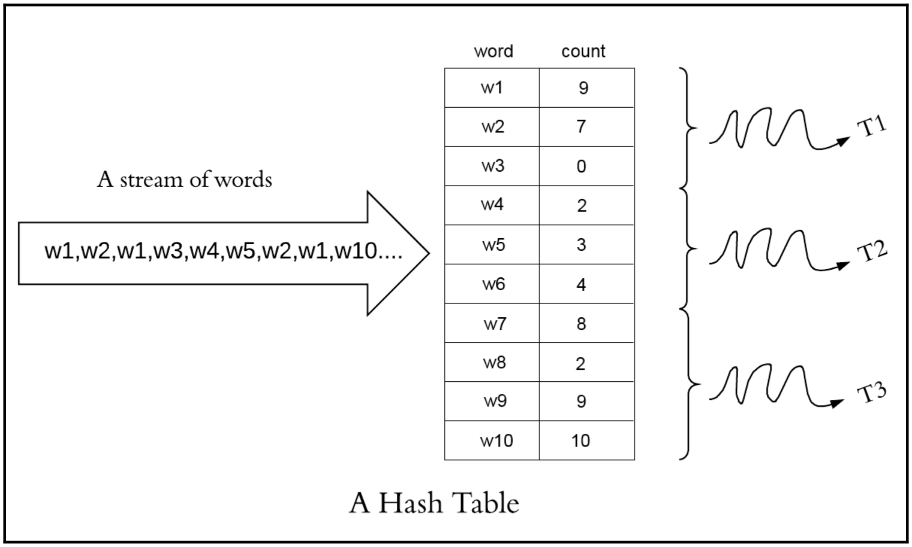
```
task1.join();
```

The map-reduce theme

We are essentially distributing work across different threads. As we have already seen, the join step reduces (collates) the subresults into a single result. *Map* and *reduce* computations in various garbs essentially work on the same principle.

The ubiquitous example is a word-count program. We have a stream of words (with all punctuations stripped off), and we try to count the *frequency* of each word—how many times it occurs in the stream.

The following diagram shows how we could use a *hash table* to compute the frequencies:



word	count
w1	9
w2	7
w3	0
w4	2
w5	3
w6	4
w7	8
w8	2
w9	9
w10	10

A Hash Table

The hash table is a *shared* data structure among different threads. The algorithm is simple:

```
    key <- hash(word)
      if (key is present in the hash table) {
      increment associated count
      } else {
        put key in the table - and initialize the associated count to 1
}
```

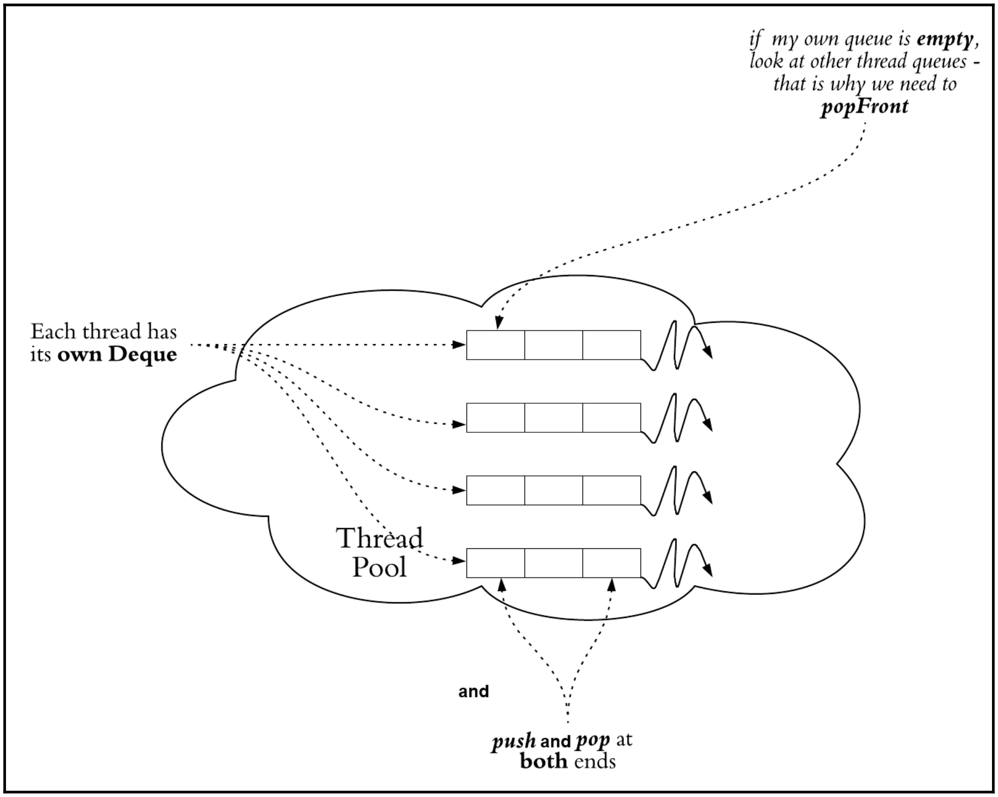
As we are concurrent, the idea is to deal with multiple words *concurrently.* The previous algorithm should be running in multiple threads, each responsible for a *subsection of the hash table.*

The join step just iterates over the hash table and outputs the frequencies. In the next chapter, we will see how such a hash table could be designed and you will be introduced to the **lock-striping design pattern**.

Work stealing

ExecuterService is an interface, and the *ForkJoinPool* is one implementation of it. This pool will look for the available CPU and create that many worker threads. The load is then distributed evenly across each thread.

The tasks are *distributed* to each thread using a thread-specific *deque*. The following diagram shows each thread having its own buffer of tasks. The buffer is a *deque*—a data structure that allows pushing and popping from *either end* of the buffer:

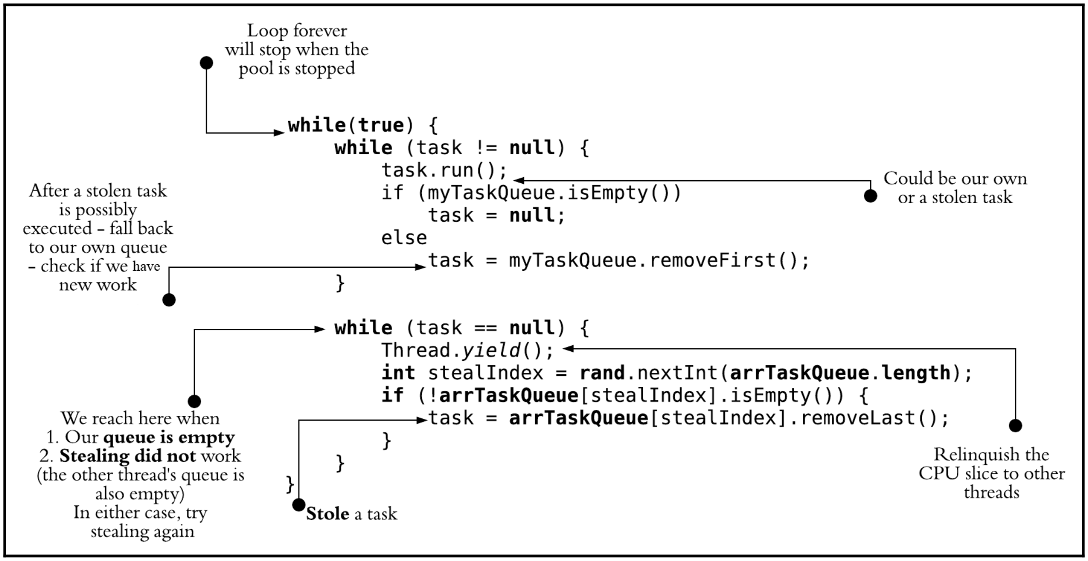

if my own queue is ***empty****,*
look at other thread queues -
that is why we need to
popFront
Each thread has
its **own Deque**
Thread
Pool
and
push **and** ***pop*** at
both ends

The deque allows threads to employ *work stealing*. It could happen that some tasks are computation heavy, and, as a result, the processing threads might take longer. On the other hand, other pool threads might get lighter tasks and won't have any work left to do.

The free threads could *steal the task* from the deque of some overloaded, random thread. This design makes for the efficient handling of tasks. The following code shows how the pool thread works. The thread is represented by the,`TaskStealingThread` class:

```
import java.util.Deque;
import java.util.Random;

public class TaskStealingThread extends Thread {
  final Deque<Runnable>[] arrTaskQueue;
  final Random rand;
  int   myId;

    public TaskStealingThread(Deque<Runnable>[] arrTaskQueue, Random rand)
{
      this.arrTaskQueue = arrTaskQueue;
      this.rand = rand;
    }
```

As shown in the preceding code, the field `arrTaskQueue` is an array of deques. The field `rand` is used to generate the random index, which is used to index into `arrTaskQueue`. The `myId` field holds an index for this thread's deque in `arrTaskQueue`:



```
while(true) {
    while (task != null) {
        task.run();
        if (myTaskQueue.isEmpty())
            task = null;
        else
            task = myTaskQueue.removeFirst();
    }
    while (task == null) {
        Thread.yield();
        int stealIndex = rand.nextInt(arrTaskQueue.length);
        if (!arrTaskQueue[stealIndex].isEmpty()) {
            task = arrTaskQueue[stealIndex].removeLast();
        }
    }
}
```

Loop forever will stop when the pool is stopped
Could be our own or a stolen task
After a stolen task is possibly executed - fall back to our own queue - check if we have new work
We reach here when 1. Our queue is empty 2. Stealing did not work (the other thread's queue is also empty) In either case, try stealing again
Stole a task
Relinquish the CPU slice to other threads

The preceding diagram helps to see how the stealing works. There are two levels of nested while loops. The first is a deliberate infinite loop. It keeps the pool thread alive (meaning it will only quit on an interrupt, when the pool shuts down):

```
@Override
public void run() {
  int myId = (int) getId();
  Deque<Runnable> myTaskQueue = arrTaskQueue[myId];
  Runnable task = null;
  if (!myTaskQueue.isEmpty()) {
    task = myTaskQueue.pop();
    }
```

The preceding code gets the deque corresponding to this thread instance. It tries to get a new task by popping the deque:

```
while(true) {
```

The first (infinite) loop starts off:

```
while (task != null) {
  task.run();
  task = myTaskQueue.removeFirst();
}
```

If the `task` is not null, this means that the thread has enough tasks to work on. So, it keeps taking tasks from the queue and runs them:

```
while (task == null) {
  Thread.yield();
  int stealIndex = rand.nextInt(arrTaskQueue.length);
  if (!arrTaskQueue[stealIndex].isEmpty()) {
    task = arrTaskQueue[stealIndex].removeLast();
  }
}
```

As the preceding diagram shows us, the task could be null as our task queue is exhausted. So, we enter the second while loop with the intent of stealing work. However, we need to call `yield()` here so other threads get a chance to run first. Note that we prefer the thread owning the queue to run its assigned tasks if possible.

The `stealIndex` variable is set to one random location (*0* .. `arrTaskQueue.length()`*-1*). We keep looking at the task queues of other pool threads and try taking a task from one of them.

Once we take the task and run it, we fall back to check our own queue first. If the thread has got work assigned to it, it goes back again, running tasks in its own queue.

The cycle continues—thereby interweaving *work stealing* with normal task processing.

Active objects

Here is a classic problem given a piece of legacy code written without any threading consideration. How do we make it thread-safe?

The following class illustrates this problem:

```
private class LegacyCode {
  int x;
  int y;

    public LegacyCode(int x, int y) {
      this.x = x;
      this.y = y;
    }
    // setters and getters are skipped
    // they are there though
```

There are two methods, `m1()` and `m2()`, which change the instance state in some way. Here is the `m1()` method:

```
public void m1() {
  setX(9);
  setY(0);
}
```

It sets the *x* field to 9 and the *y* field to 0:

```
public void m2() {
  setY(0);
  setX(9);
}
```

The `m2()` method does the opposite: it sets *x* to `0` and *y* to `9`.

If we try to make this class concurrent, using threads, you know we need to carefully synchronize access to all *shared* states. Of course, any legacy code has many other ramifications—side effects, exceptions, shared data structures, and so on.

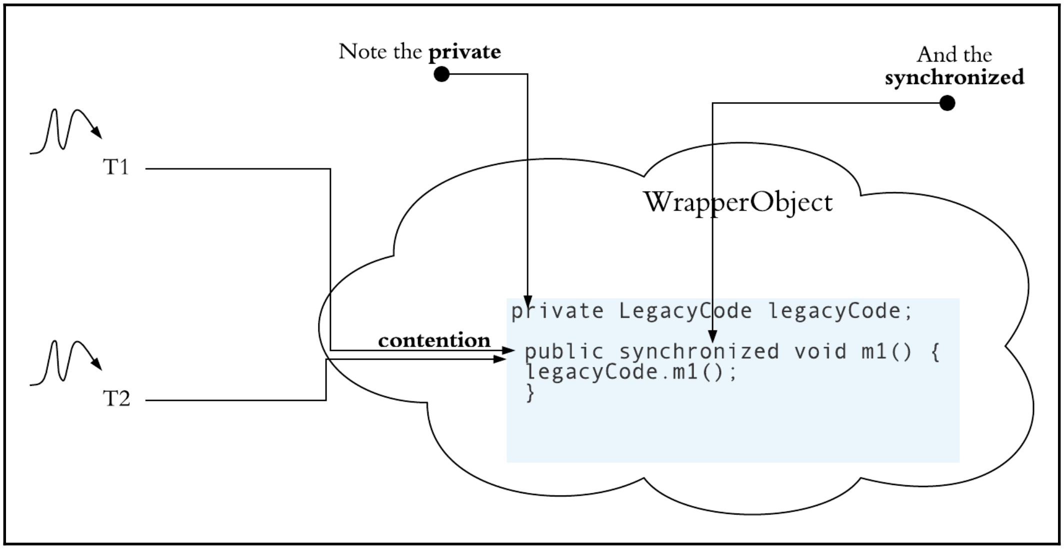
It would indeed be a herculean effort to correctly synchronize all the changes. This is a case when we could resort to the *big lock* solution.

Hiding and adapting

The solution is to provide *controlled access* to the legacy code. For example, we could hide the legacy code as a private instance and provide delegate methods:

```
public class WrapperObject {
  private LegacyCode legacyCode;

  public synchronized void m1() {
    legacyCode.m1();
    }
    ...
```

The `WrapperObject` class's intrinsic lock is used to synchronize across threads. This will make it thread-safe, but the concurrency will get serialized. The thread contention would be a major issue:



```
private LegacyCode legacyCode;

 public synchronized void m1() {
 legacyCode.m1();
 }
```

The strategy works, but could we do better? It seems as if *active objects* are what the doctor ordered.

Using a proxy

This pattern beautifully exploits the *proxy* design pattern. There is a blocking queue that acts as a *task queue*. Here is its declaration:

```
private BlockingQueue<Runnable> queue = new
LinkedBlockingQueue<Runnable>();

private Thread processorThread;
```

The following method starts a *single* consumer thread, which consumes from the task queue. It is just a thread and a runnable. The thread is tracked in a field called `processorThread`:

```
public void startTheActiveObject() {
  processorThread = new Thread(new Runnable() {

    @Override
    public void run() {
```

The `run()` method runs forever till the thread is interrupted. Upon interruption, a message is printed and the processor thread exits. The method starts the thread and exits:

```
while (true) {
  try {
    queue.take().run();
      } catch (InterruptedException e) {
                    // terminate
          System.out.println("Active Object Done!");
          break;
        }
     }
  }
});
processorThread.start();
}
```

The proxy *encodes the logic as a runnable and puts it on the queue as a task!* The following shows a *proxy* to the wrapped method:

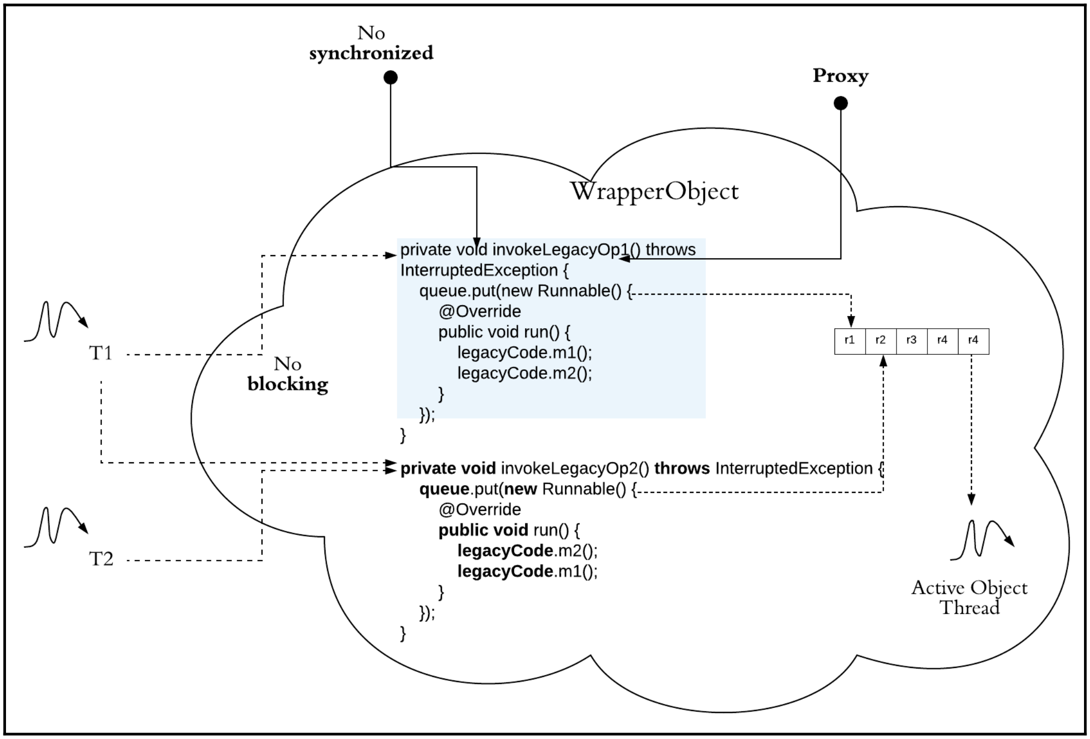
```
private void invokeLegacyOp1() throws InterruptedException {
  queue.put(new Runnable() {
    @Override
    public void run() {
      legacyCode.m1();
      legacyCode.m2();
    }
  });
}
```

The other methods are also similar:

```
private void invokeLegacyOp2() throws InterruptedException {
  queue.put(new Runnable() {
    @Override
    public void run() {
      legacyCode.m2();
      legacyCode.m1();
```

```
        }
      });
    }
```

The following diagram helps us understand how the pieces fit together and the pattern is realized:



```
private void invokeLegacyOp1() throws
InterruptedException {
    queue.put(new Runnable() {
        @Override
        public void run() {
            legacyCode.m1();
            legacyCode.m2();
        }
    });
}
private void invokeLegacyOp2() throws InterruptedException {
    queue.put(new Runnable() {
        @Override
        public void run() {
            legacyCode.m2();
            legacyCode.m1();
        }
    });
}
```

The following is a driver: it exercises all the previous machinery. We create the wrapper object and call the methods, *oblivious to the fact that the methods are really proxies*!

We verify that the methods run as the relevant messages are printed on the console:

```
public static void main(String[] args) throws InterruptedException {
  WrapperObject wrapperObject = new WrapperObject();
  wrapperObject.startTheActiveObject();

  wrapperObject.invokeLegacyOp1();
  wrapperObject.invokeLegacyOp2();

  Thread.sleep(5000);
  wrapperObject.stop();
}

private void stop() {
  thread.interrupt();
}
```

Finally, we stop the processor thread by calling the `stop()` method. This method simply *interrupts* the thread, which prints a suitable farewell message and exits!

So, we have now seen how the interruption semantics work.

Summary

Thread creation, scheduling, and destruction—all of these are costly—and they take a substantial amount of computation. Creating threads *on demand* and destroying them once they have finished their tasks is an inefficient way to organize a multi-threaded computation.

Thread pools are used to solve this problem. Each thread in the pool repeatedly waits for a *task,* a short-lived unit of computation. The thread is reused, executes a task, and then goes back to the pool to await the next one.

We implemented our own thread pool and used it to exercise the driver. Then, we had a detailed look at the fork-join API and studied how it uses *work stealing.*

We looked at the active object design pattern next, showing you how the idea is to hide the internal concurrency using a proxy.

We also touched upon the map-reduce theme and introduced concurrent hashing. We will be taking a closer look at this fascinating data structure in the next chapter! Stay tuned.

5
Increasing the Concurrency

This chapter covers the various strategies for increasing the concurrency of data structures. We will look at the **lock-free** variants of stacks and queues. These lock-free versions use **compare and swap (CAS)** instead of explicit synchronization. This is a complex programming model, and as we will soon see, it requires extreme caution and deep analysis to make sure there are no subtle concurrency bugs, such as the ABA problem. The ABA problem is also described in this chapter, along with a strategy that can be used to deal with it effectively.

This chapter will also cover commonly used data structures, such as the following:

- Concurrent stacks
- Queues
- Hash tables

Finally, we will have a look at *hash tables*, which are used to efficiently implement a **set** abstraction. A set holds unique elements and needs to cater for a fast `lookup` operation, whether a value is present in the set or not. First, we will look at the solution for this using explicit locking and the **lock striping design pattern** to increase the concurrency.

For complete code files you can visit `https://github.com/PacktPublishing/Concurrent-Patterns-and-Best-Practices`

A lock-free stack

As noted in the introduction, lock-free algorithms are more complicated than equivalent lock-based ones. Essentially, the principle behind them is based on making atomic changes to a single variable while maintaining data consistency.

A **last in, first out** (**LIFO**) stack is a very common data structure in programming. We will use a singly linked list to represent the stack abstraction. Each node of the list holds a value and a pointer to the next node, if there is another one; otherwise, it will hold `null`. The pointer is an **atomic reference**.

Atomic references

`AtomicReference` is just like an `AtomicInteger`, where multiple threads can update the reference without causing any inconsistencies. To update such a reference, we use its `compareAndSet` method. This method internally uses a CAS (*compare and swap)* instruction. See `chapter 3`, *More Threading Patterns,* for a refresher on CAS if you need to jog your memory.

The following snippet shows an atomic reference in action:

```
public class TryAtomicReference {
  public static void main(String[] args) {
  String firstRef = "Reference Value 0";

  AtomicReference<String> atomicStringReference =new
AtomicReference<String>(firstRef);
  System.out.println(atomicStringReference.compareAndSet(firstRef,
"Reference Value 1"));
  System.out.println(atomicStringReference.compareAndSet(firstRef,
"Reference Value 2"));

  System.out.println(atomicStringReference.get());
  }
}
```

Upon running the program, the output is as follows:

```
true
false
Reference Value 1
```

The `atomicReference` variable is of the `AtomicReference` type. It is initialized with a reference to a string value, `Reference Value 0`.

As shown in the preceding output, the first `compareAndSet` call succeeds. We replace the value with `Reference Value 1`. However, the second call fails, as shown by `false` in the output.

It fails, as the internal reference is no more `firstRef`. We have already replaced it with a new reference value, and as a result, there is a mismatch.

Finally, we print the contained value, which clearly shows that setting the value to `Reference Value 2` has failed.

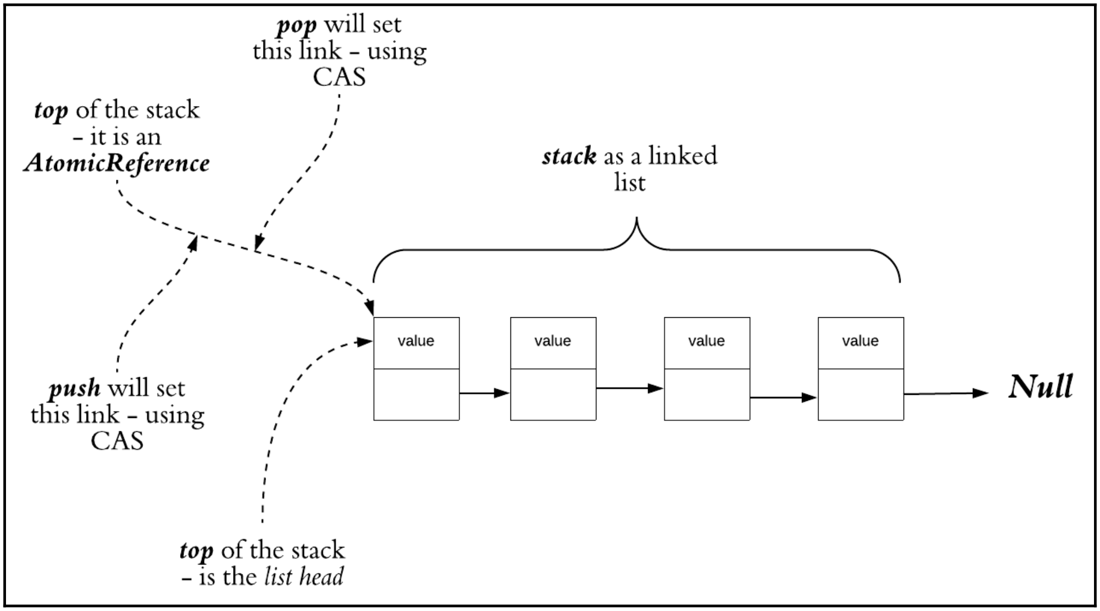
Fixing this error is left as a trivial exercise for you.

The stack implementation

The lock-free stack is a singly linked list of node elements. In this arrangement, a node contains a value and a link to the next element. The implementation uses atomic references to maintain the *top* of the stack. The `push` method allocates a new node whose next field is set to the current top of the stack, and then uses CAS *to try* to change the top, to make it the new top node. Once this CAS attempt succeeds, the `push` operation completes. In case the CAS fails, we retry again. This implies a `while` loop, as shown in the code snippet on from the following diagram.

Similar logic exists for the `pop` method to remove nodes from the stack.

No matter whether the CAS succeeds or fails, the stack is always in a consistent state. The following diagram shows the overall arrangement of things:


pop will set this link - using CAS
top of the stack - it is an *AtomicReference*
stack as a linked list
value
value
value
value
Null
push will set this link - using CAS
top of the stack - is the *list head*

The following code shows the node definition. Note that the next links are normal Java references. The `Node` class is shown in the following code:

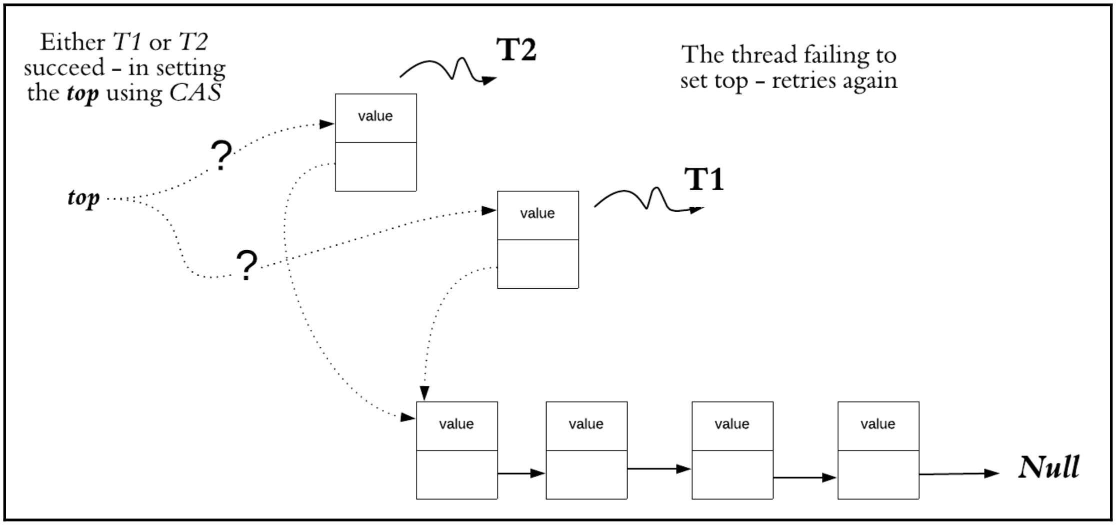
```
private static class Node <E> {
  public final E item;
  public Node<E> next;
  public Node(E item) {
    this.item = item;
    }
}
```

The following code shows the `stack` class and the `push` method. The `top` is a field that is an `AtomicReference`*:*

```
public class LockFreeStack <T> {
  AtomicReference<Node<T>> top = new AtomicReference<Node<T>>();
  public void push(T item) {
    Node<T> newHead = new Node<T>(item);
    Node<T> oldHead;
    do {
      oldHead = top.get();
      newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }
```

The `newHead` variable points to the node that we are trying to insert and that we want to become the new `top` of the stack. The `oldHead` variable is the existing `Node` reference, which is the current `top` of the stack.

We try to set the `newHead`, expecting the `oldHead` to be the old value. The `do-while` loop takes care of the CAS failing. If another thread came-in between and changed the `top`, our assumption of the `oldHead` as being `top` of the stack will be false. Remember the CAS semantics: if the value is changed, the CAS succeeds and returns `true`; in the case of a failure, it returns `false`. This arrangement is shown in the following diagram:


Either *T1* or *T2* succeed - in setting the ***top*** using *CAS*
T2
The thread failing to set top - retries again
value
?
top
T1
value
?
value
value
value
value
Null

The loop is exited when the CAS succeeds. Note the negation on the conditional.

The `pop` method comes next, as shown in the following code:

```
public T pop() {
  Node<T> oldHead;
  Node<T> newHead;
    do {
      oldHead = top.get();
      if (oldHead == null)
        return null;
        newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item;
}
```

The reasoning is similar to the `push` method: we are changing the value of the `top` field. The only difference is that we are removing and returning an element from the list instead of adding to it!

A lock-free FIFO queue

A **FIFO (first in, first out)** queue is a data structure where the elements are popped out in the same order in which they were inserted. This is in contrast to a stack, where the order is LIFO (last in, first out). In case you need to refresh your memory of these terms, head to `https://www.geeksforgeeks.org/queue-data-structure/`.

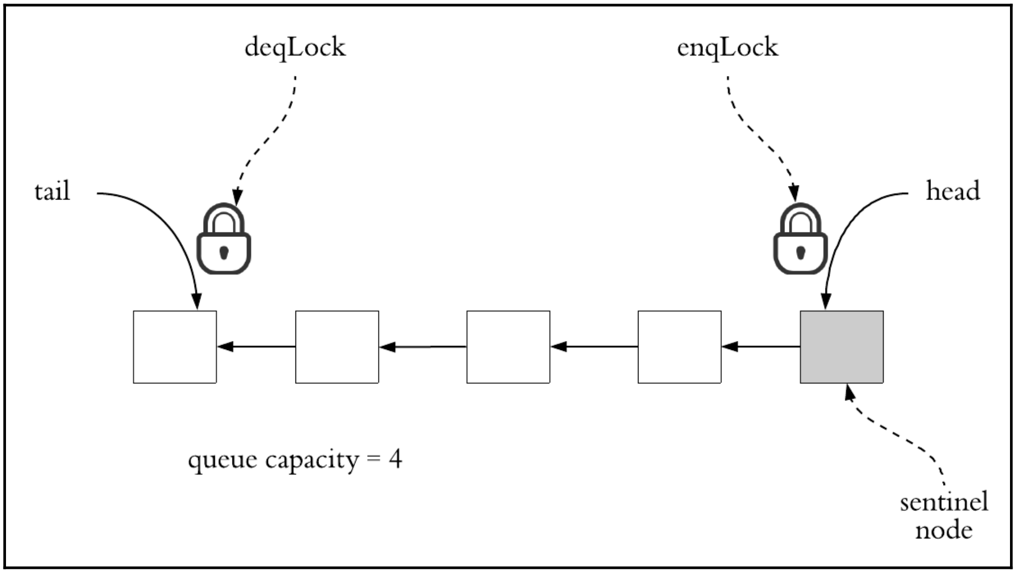
One obvious way for making a queue safer is to use a single lock to make it thread-safe. We could use either an explicit lock (a `ReentrantLock`) or an intrinsic lock by just making the methods synchronized.

This will, of course, work; however, it will hurt concurrency. At any point, only one thread will be able to push or pop the queue.

Our goal is to increase the concurrency while at the same time ensuring thread safety. Could we allow two threads, one producing elements to the queue and another consuming elements from it?

The following class shows a thread-safe and bounded FIFO queue using two locks. One lock is used to protect the insertion (*the enqueuing of elements*) and another lock takes elements from the queue (*dequeuing of elements*).

The following is a diagrammatic representation of the design:



The following class shows the implementation:

```
public class ThreadSafeQueue<T> {
  protected class Node {
    public T value;
    public volatile Node next;
    public Node(T value) {
      this.value = value;
      this.next = null;
    }
  }

  private ReentrantLock enqLock, deqLock;
  Condition notEmptyCond, notFullCond;
  AtomicInteger size;
  volatile Node head, tail;
  final int capacity;
```

We have two conditions to deal with: an empty and a full queue, respectively. The `size` field keeps track of the number of queue elements. In the following code, we will see how its value is used to signal the correct condition variable. Finally, the `head` and `tail` variables are *volatile*. Refer to `chapter 2`, *A Taste of Some Concurrency Patterns*, for more information on volatile variables.

The following snippet shows the constructor. The queue will hold `cap` elements at maximum:

```
public AThreadSafeQueue(int cap) {
  capacity = cap;
  head = new Node(null);
  tail = head;
  size = new AtomicInteger(0);
  enqLock = new ReentrantLock();
  deqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  notEmptyCondition = deqLock.newCondition();
}
```

The following method shows the element insertion:

```
public void enq(T x) throws InterruptedException {
  boolean awakeConsumers = false;
  enqLock.lock();
  try {
    while (size.get() == capacity)
      notFullCond.await();
      Node e = new Node(x);
      tail.next = e;
      tail = e;
      if (size.getAndIncrement() == 0)
        awakeConsumers = true;
  } finally {
      enqLock.unlock();
    }
    if (awakeConsumers) {
      deqLock.lock();
      try {
        notEmptyCond.signalAll();
        } finally {
            deqLock.lock();
          }
    }
}
```

We take the `enqLock` and then check whether the queue is full. When the number of enqueued elements equals the queue `capacity`, the caller needs to wait.

If so, the caller waits on the `notFullCond` condition. As noted in `Chapter 3`, *More Threading Patterns*, a condition variable and lock go hand in hand. The `condition.await()` call releases the lock and `goto` to wait for the condition to become `true`. In this case, the condition required is that there should be at least one empty slot to put the new element in to.

When someone uses `deque` (pop) to take an element off the queue, it signals the condition variable. **Condition variables** are communication mechanisms between threads.

Finally, when the `enqLock` is released, one element has been added to the queue. If the queue is empty, and we have just produced this element, the consumers should be told. The following lines of code mark the flag:

```
if (size.getAndIncrement() == 0)
  awakeConsumers = true;
```

The `getAndIncrement()` method of an atomic integer *increments the variable and returns its previous value.* So, if the size were `0`, there could possibly be sleeping consumers, waiting for elements to appear.

As we have just produced an element, we signal the consumers (if there are any) to inform them about the new element. The `deq()` method to pop an element off the queue is shown in the following queue:

```
public T deq() throws InterruptedException {
  T result;
```

```
      boolean awakeProducers = false;
      deqLock.lock();
      try {
        while (size.get() == 0)
        notEmptyCond.await();
        result = head.next.value;
        head = head.next;
        if (size.getAndDecrement() == capacity)
          awakeProducers = true;
        } finally {
            deqLock.unlock();
        }
        if (awakeProducers) {
          enqLock.lock();
          try {
            notFullCond.signalAll();
          } finally {
              enqLock.unlock();
              }
        }
        return result;
    }
```

The logic is similar to the `enq()` method. This method just checks the opposite condition, the queue being full. If so, it awakes any blocked producers, notifying them that there is space available in the queue.

The `head` node is a sentinel node. Its value is *meaningless*. Once we pop a node, it becomes a sentinel.

How the flow works

There are two locks, and though the `head` and `tail` fields are *volatile,* we know that a volatile field just ensures that the latest value of the variable is read. There is no guarantee against *race conditions* due to *lost updates*, though.

If you look carefully, you'll notice that the `enq(v)` method does not refer to the `head` field at all! Similarly, the `deq()` method never uses the `tail` field. This ensures that we do not change these in any erroneous ways.

Next, we have two locks again (this point bears stressing). What happens when the `enq()` method is in the middle of adding an element? The `deq()` thread could come and pop a half-initialized node. What prevents this?

This is impossible because of the way the logic is structured. The following is the relevant `enq()` method snippet again:

```
while (size.get() == capacity)
  notFullCond.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (size.getAndIncrement() == 0)
    awakeConsumers = true;
```

There are two cases: the size is 0 or it is nonzero. In the first case, any consumer will be blocked and explicitly awakened after the element is produced completely in the queue.

On the other hand, if the size is nonzero, then there is at least one element to consume. This means that the producer and consumer threads differ *by at least one element*. Hence, when the `enq()` thread is producing the element, the `deq()` thread correctly pops off the head element.

A lock-free queue

We will now discuss creating a queue implementation without using any locks whatsoever! Before diving into this pattern, we need to understand one important element—the atomic reference.

Going lock-free

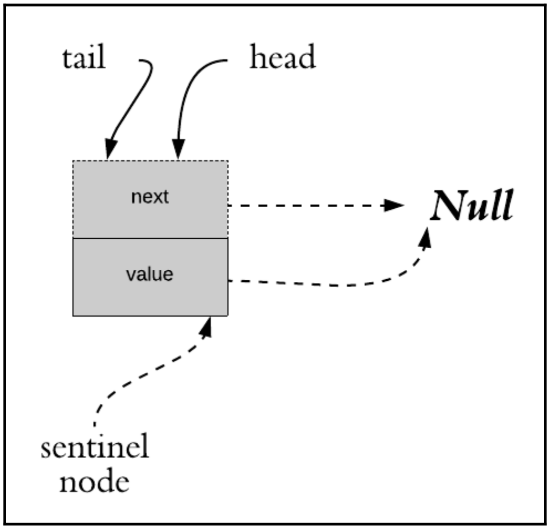
Given that this is an introduction to the `AtomicReference`, the following code is a concurrent queue implementation that doesn't use any explicit locking. The following snippet shows the class definition:

```
public class NoLocksQueue<T> {
  protected class Node {
    public T value;
    public AtomicReference<Node> next;

    public Node(T value) {
      this.value = value;
      this.next = new AtomicReference<>(null);
    }
  }

  volatile AtomicReference<Node> head, tail;
```

This queue is unbounded. Note the absence of any capacity field. The node has a value and the next pointer, which is an AtomicReference. Both the head and tail are also atomic references, and these are *volatile.* The constructor is as follows:

```
public NoLocksQueue() {
  final Node sentinel = new Node(null);
  head = new AtomicReference<>(sentinel);
  tail = new AtomicReference<>(sentinel);
}
```

The following diagram shows the state of things just after the constructor completes. This essentially signifies an empty queue, as shown in the following diagram:



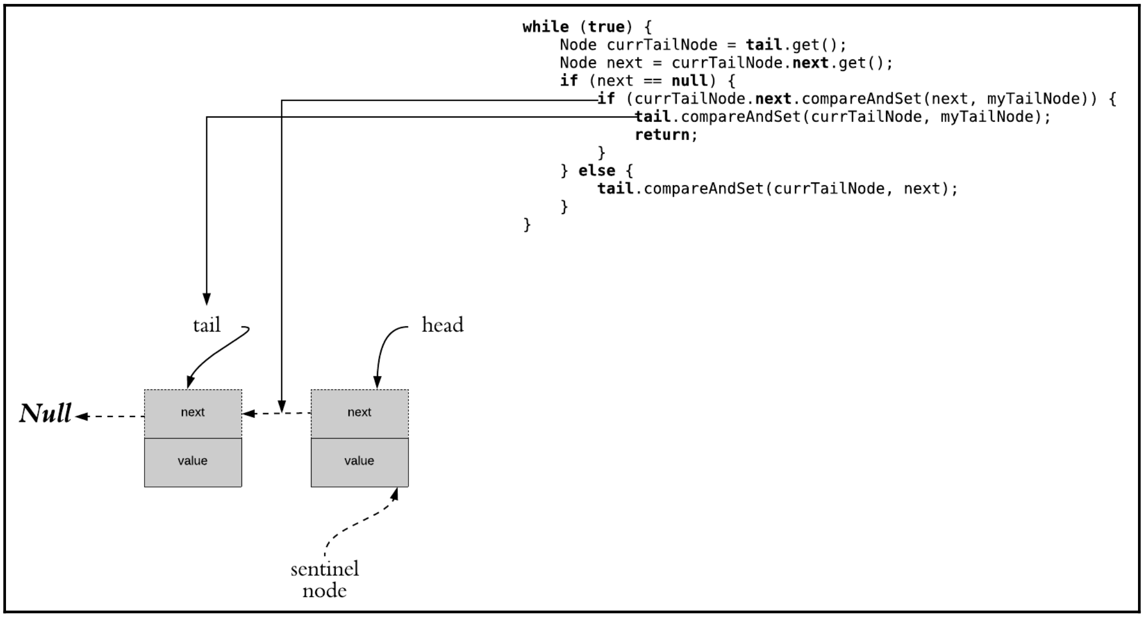
The sentinel node's value is meaningless.

The enque(v) method

The enque(v) method is shown in the following code. Note the enveloping while(true) loop—our changes may fail because of competing threads:

```
public void enque(T v) {
  Node myTailNode = new Node(v);
  while (true) {
    Node currTailNode = tail.get();
    Node next = currTailNode.next.get();
    if (next == null) {
      if (currTailNode.next.compareAndSet(next, myTailNode)) {
        tail.compareAndSet(currTailNode, myTailNode);
        return;
      }
    } else {
        tail.compareAndSet(currTailNode, next);
      }
  }
}
```

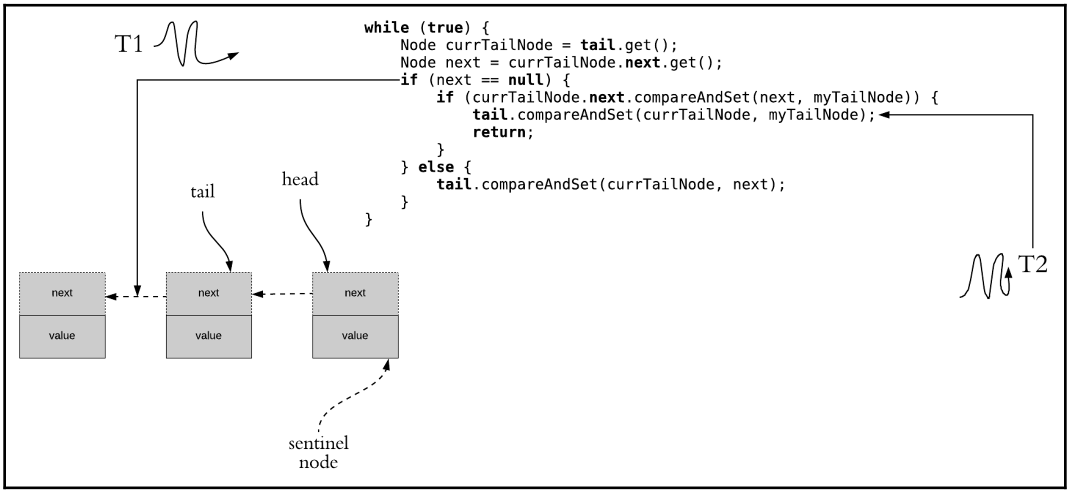
The execution flows are rather subtle. Let's take them one by one, hold on, and you will understand. Let us take the case of only one thread enqueuing an element, with no other thread enqueuing or dequeuing elements. Let us also say that the queue is empty. See the preceding diagram and that the queue is also empty when the constructor execution is complete. This scenario is depicted in the following diagram:



```
while (true) {
    Node currTailNode = tail.get();
    Node next = currTailNode.next.get();
    if (next == null) {
        if (currTailNode.next.compareAndSet(next, myTailNode)) {
            tail.compareAndSet(currTailNode, myTailNode);
            return;
        }
    } else {
        tail.compareAndSet(currTailNode, next);
    }
}
```

As shown in the preceding diagram, we use the `AtomicReference`. We also use the `compareAndSet(...)` call to change values. As shown in the previous section, this call could fail, as some other thread could have raced ahead and made the change before us. However, as we have assumed that there is just one thread, this call will succeed, and we will have added an element to the queue.

For the next scenario, let us add two competing threads, both trying to enqueue an element, as shown in the following diagram:



```
while (true) {
    Node currTailNode = tail.get();
    Node next = currTailNode.next.get();
    if (next == null) {
        if (currTailNode.next.compareAndSet(next, myTailNode)) {
            tail.compareAndSet(currTailNode, myTailNode);
            return;
        }
    } else {
        tail.compareAndSet(currTailNode, next);
    }
}
```

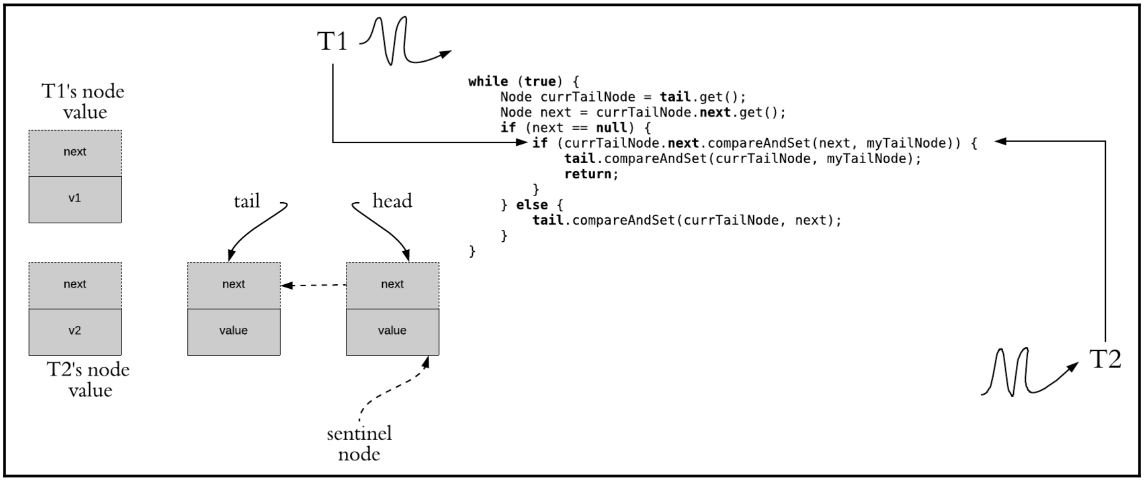
As shown in the preceding diagram, we have two competing threads, each trying to enqueue an element. Their executions interleave, as shown in the following code:

```
Thread T2 sets tail.next variable to the new node (the tail is not set yet)
Thread T1 sets the local next variable to the new node (just enqueued by
T2)
```

This makes `T1` take the `else` path—either `T1` or `T2` could set the `tail` variable. One attempt would succeed and the other would fail (you should check for yourself why this would be the case).

As a result, the element is enqueued correctly.

The third scenario is when both threads read the correct `tail.next` value as `null`, as shown in the following diagram:



```
while (true) {
    Node currTailNode = tail.get();
    Node next = currTailNode.next.get();
    if (next == null) {
        if (currTailNode.next.compareAndSet(next, myTailNode)) {
            tail.compareAndSet(currTailNode, myTailNode);
            return;
        }
    } else {
        tail.compareAndSet(currTailNode, next);
    }
}
```

In this case, one wins, and the other loops back (remember the `while` loop?) and retries. Try it out with more than two threads; draw some diagrams like the previous one and convince yourself that it works in all cases.

The deq() method

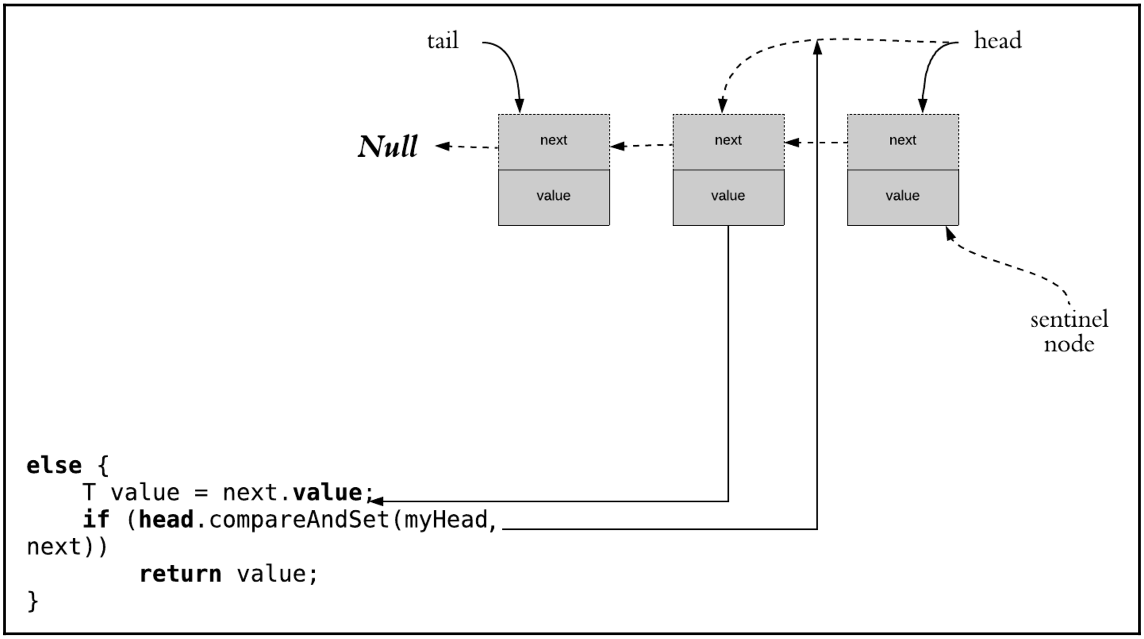
The `deq()` method for popping an element off the queue comes next. It too relies on carefully orchestrating the `compareAndSet` primitive to achieve thread safety. Note that if the queue is empty, the method throws an exception. You can see the details of this arrangement in the following code:

```
public T deque() {
  while (true) {
    Node myHead = head.get();
    Node myTail = tail.get();
    Node next = myHead.next.get();
    if (myHead == head.get()) {
      if (myHead == myTail) {
        if (next == null) {
          throw new QueueIsEmptyException();
         }
        tail.compareAndSet(myTail, next);
      } else {
          T value = next.value;
          if (head.compareAndSet(myHead, next))
```

```
                return value;
                  }
              }
          }
      }
```

Let's trace the execution of this method. Let's assume that there is just one thread to begin with. If the queue is empty, all three `if` statements return `true` (please see the previous empty queue method diagram), and the method throws an exception.

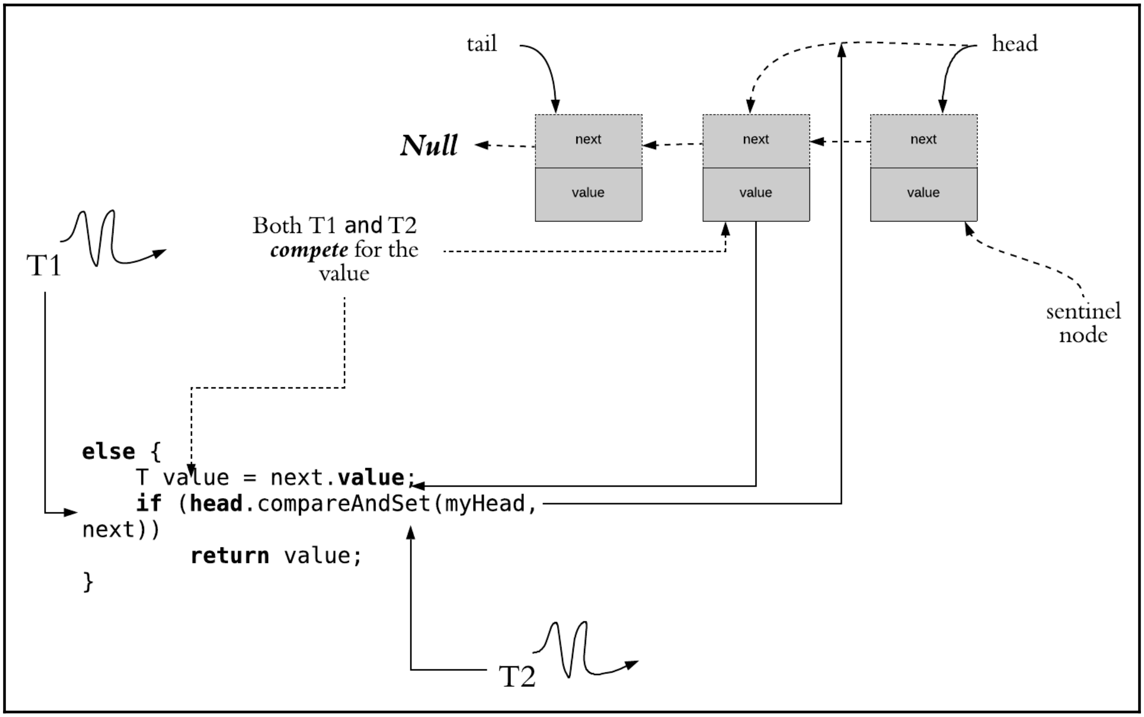
Let's now say that the queue is not empty, and there is still just one thread. In that case, the `first != last` (note that an empty queue only has the sentinel node, with both the head and tail's `next` value pointing at it!) and the `else` clause execute, as shown in the following diagram:



```
else {
    T value = next.value;
    if (head.compareAndSet(myHead,
next))
        return value;
}
```

As shown in the preceding diagram, the head is changed correctly as there is assumed to be just *one dequeuing thread*! Things work well for the sequential execution,which is the simplest case!

Now, let's add two threads to the mix, as shown in the following diagram:



```
else {
    T value = next.value;
    if (head.compareAndSet(myHead,
next))
        return value;
}
```

Consider an interleaving scenario, where both threads reach the `else` clause at the same time. Both are competing for the same value, and the correct semantics should ensure that any one thread should get this value. The other thread should retry!

The correct semantics are guaranteed by the `compareAndSet(...)` call. One thread will succeed and the next one will fail as the old value of `head` won't be equal to the stale *first* value! As this is happening inside a loop, the thread for which the `compareAndSet(...)` returned `false` will retry.

Concurrent execution of the enque and deque methods

Armed with all of this knowledge, let's now tackle the more complex case—namely, two threads, one enqueuing and one dequeuing at the same time! A conflict is possible if a node is just enqueued, but before the `tail` is updated! Let's look at the following code:

```
public T deque() {
  while (true) {
    Node myHead = head.get();
```

```
        Node myTail = tail.get();
        Node next = myHead.next.get();
        if (myHead == head.get()) {
          if (myHead == myTail) {
            if (next == null) {
              throw new QueueIsEmptyException();
            }
            tail.compareAndSet(myTail, next);
```

The `first == last` condition can succeed; however, the `next == null` condition may not be `true`! How is this possible? It is possible because of the following code from the `enque()` method:

```
    if (next == null) {
      if (last.next.compareAndSet(next, myTailNode)) {
```

In this case, the `deque()` method will advance the head node, and leave the `enque()` thread alone to update the `tail`.

Working out such scenarios in the abstract, before they happen, will help us understand the code at a deeper level.

The ABA problem

A CAS, in essence, asks "Is the value of V still A?", and updates it to a new value if the answer is yes. If we plan to manage the pool of nodes ourselves, then the ABA problem could hit us. However, let's look at one more basic concept—the **thread local**.

Thread locals

The `ThreadLocal` class allows us to create variables that are owned by threads. As there is an explicit ownership, you don't need any synchronization.

The following code shows how thread locals work:

```
    public class TryThreadLocal {
      public static class MyRunnable implements Runnable {
        private int state;
        private ThreadLocal<Integer> threadLocal;
        public MyRunnable(int state) {
          this.state = state;
          this.threadLocal = new ThreadLocal<Integer>();
        }
        @Override
        public void run() {
          this.threadLocal.set(state);
          for (int i = 0; i < 25; ++i) {
            final Integer v = threadLocal.get();
            System.out.println("Thread " + Thread.currentThread().getId() + ",
    value = " + v);
            threadLocal.set(v + 1);
            try {
              Thread.sleep(2000);
                } catch (InterruptedException e) {
                        // nothing
                    }
                }
            }
        }

        public static void main(String[] args) throws InterruptedException {
          MyRunnable sharedRunnableInstance = new MyRunnable(6);
          Thread thread1 = new Thread(sharedRunnableInstance);
          Thread thread2 = new Thread(sharedRunnableInstance);

          thread1.start();
          thread2.start();

          thread1.join(); //wait for thread 1 to terminate
          thread2.join(); //wait for thread 2 to terminate
        }
    }
```

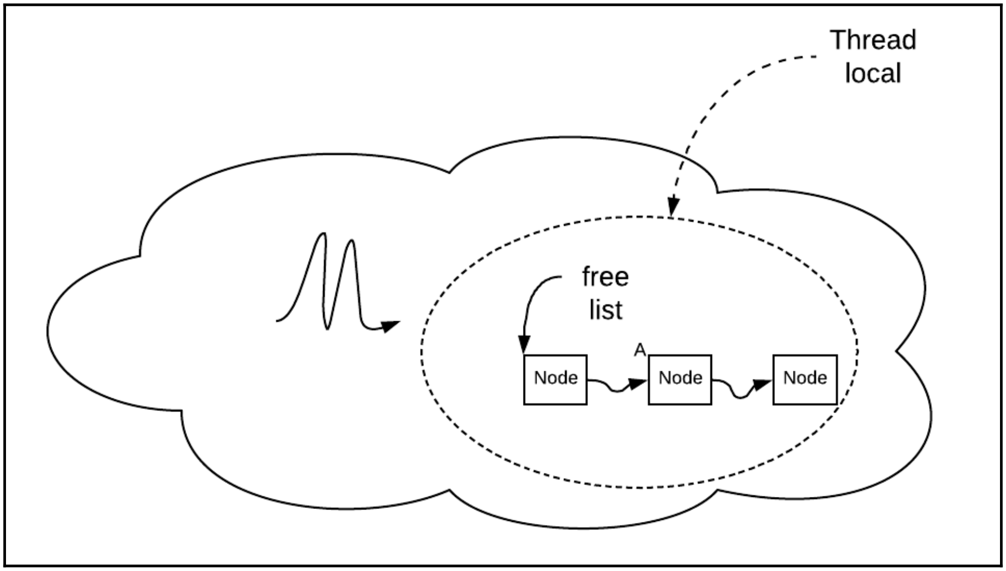
The runnable instance has a state variable—when the threads start running, both put the starting value in a thread local. Both threads loop for a while, incrementing and printing the thread local variable.

Running the code shows two threads incrementing their local variable, without interfering with each other.

Pooling the free nodes

Going back to the lock-free queue example, we might want to manage our own pool of nodes by recycling them. When the sentinel is moved, instead of the node getting garbage collected, we add it to a pool, thereby making the overall memory management more efficient.

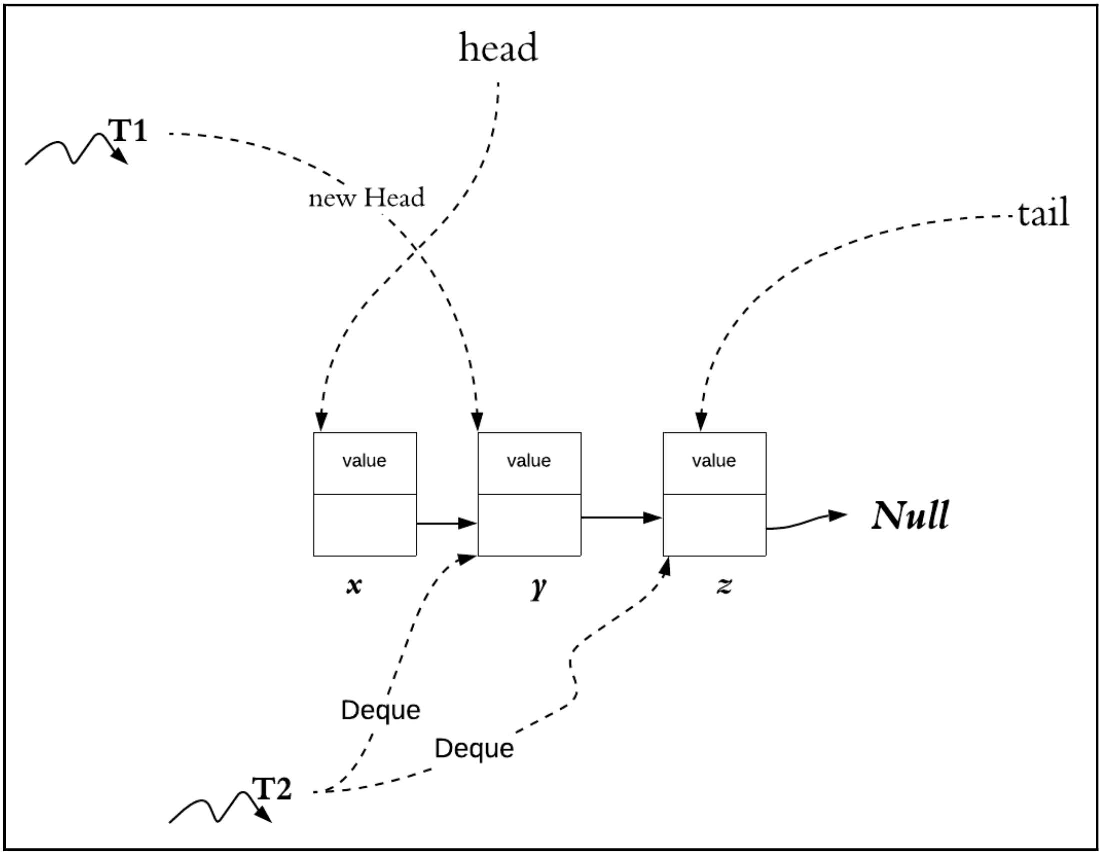
Each thread could maintain its own *list* of free nodes—you guessed right—in a thread pool. The following diagram shows the design:



When the thread enqueues, it picks a node from this local pool. When it dequeues, it adds the node back to the free list. In the case that the pool is empty, the thread will allocate a new node using the `new` operator, as before. Adding and removing nodes from the pool does not need to be synchronized, as the list is stored in a *thread local*!

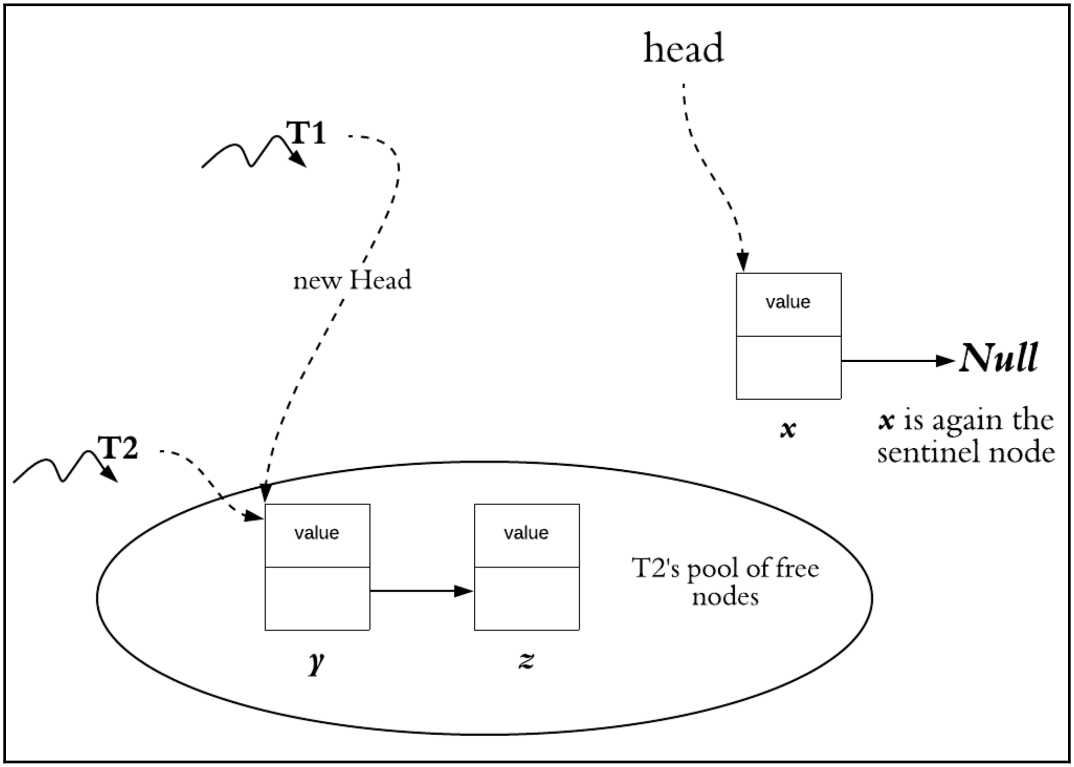
If a thread, on average, performs the same amount of enqueuing and dequeuing, then this design would work very well. However, if we use just the `AtomicReference`, then there is a nasty bug—the ABA problem.

Given the preceding queuing strategy—and given the following queue state—with three nodes, the following sequence will take place:



The `T1` thread dequeues and is about to CAS the head from *x* (the old value) to *y* (the new value), but before it is able to CAS, it is preempted, and the `T2` thread is run.

The `T2` thread dequeues nodes *y* and *z*, thereby adding both *x* and *y* to the local pool's free list. Node *x* is enqueued back and is again dequeued, thereby becoming a sentinel node. This scenario is shown in the following diagram:



Now, the `T1` thread wakes up and starts running. As shown in the preceding diagram, `T1` performs a CAS and succeeds, as the old value *x* is once more the sentinel! The head now points at *y*, which is already in the free pool!

This is where the name ABA comes from, as the reference changes from A (*x*) to B (*y* and *z*) to A(*x*) again! The `T1` thread is oblivious to all these changes happening in the interim, and works on an outdated state of things! A plain atomic reference does not cut it here—the solution is to make it aware of these changes using an `AtomicStampedReference` instead!

The atomic stamped reference

The following code illustrates an `AtomicStampedReference`. It is an `AtomicReference` coupled with a stamp—a variable that is incremented upon every update of the variable:

```
public class TryAtomicStampedReference {
  public static void main(String[] args) {
    String firstRef = "Reference Value 0";
    int stamp1 = 0;
    AtomicStampedReference<String> atomicStringReference =new
AtomicStampedReference<String>(firstRef,
stamp1);

    String newRef = "Reference Value 1";
    int newStamp = stamp1 + 1;

    boolean r = atomicStringReference.compareAndSet(firstRef, newRef,
stamp1, newStamp);
    System.out.println("r: " + r);

    r = atomicStringReference.compareAndSet(firstRef, "new string",
newStamp, newStamp + 1);
    System.out.println("r: " + r);

    r = atomicStringReference.compareAndSet(newRef, "new string", stamp1,
newStamp + 1);
    System.out.println("r: " + r);

    r = atomicStringReference.compareAndSet(newRef, "new string", newStamp,
newStamp + 1);
    System.out.println("r: " + r);
    }
}
```

We have a string `Reference Value 0` stored in an `AtomicStampedReference`, along with a stamp, whose value is `0`. Next, we try to update the value to `Reference Value 1` and the stamp `0`. As both the value and stamp expectations match, the value is changed. The stamp is also updated to `1`.

The second attempt fails as the old value does not match with the current value in the variable. Next, we update the old value, but pass in an outdated stamp, so that update attempt also fails.

Lastly, we pass in the correct old value and the old stamp—as a result, the value is changed successfully.

When we run the program, the following is the console output:

```
r: true
r: false
r: false
r: true
```

To solve the ABA problem illustrated here, we could use this kind of stamped reference! We will leave you to think about the ABA problem using the `AtomicStampedReference` as an exercise.

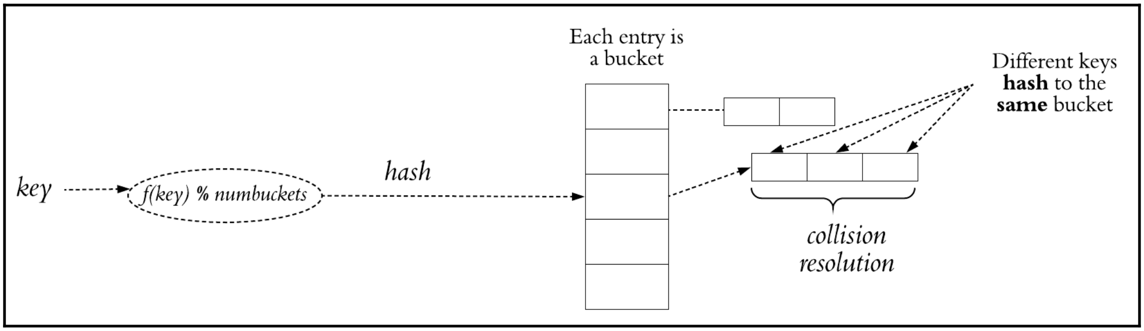
Concurrent hashing

A hash table is typically implemented as an array. Each array entry is a *list* of one or more *items*. A *hash function* maps values to indices in this array. Each Java object has a `hashCode()` method that gives an integer for an object. This *number*'s modulo to the *array length* gives us an index into this array.

For any operation to add, remove, or check whether the set contains an item, we first index into the array (or table), and then perform the rest of the processing.

The basic idea is that, given a table and a *hash function,* we provide the `contains(key)`, `add(key, value)`, and `remove(key)` methods, with a constant average time.

The following diagram shows how a typical hash table works:



The key is run through a hash function (`f()`). The resulting modulo of the hash function to the number of entries gives us a *hash*—that is, an offset into the table.

We use the hash table to represent a *hash set.* A set contains unique elements.

The following list shows the method contracts:

- The `add(v)` method tries to add the element to the set. If the v value is already present, then the method just returns `false`. Otherwise, the method inserts the value and returns `true`.

- The `contains(v)` method returns `true` if the set contains `v`; otherwise, it returns `false`.
- The `remove(v)` method tries to removes the v element from the set. If the value is found, it is removed and the method returns `true`. Otherwise, it returns `false`.

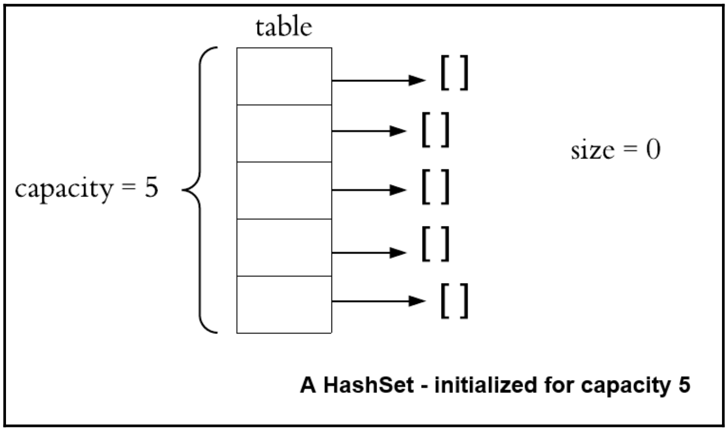
The following `HashSet<T>` class shows our abstraction:

```
public abstract class HashSet<T> {
  final static int LIST_LEN_THRESHOLD = 100;
  protected List<T>[] table;
  protected AtomicInteger size;
  protected AtomicBoolean needsToResize;

  public HashSet(int capacity) {
    size = new AtomicInteger(0);
    needsToResize = new AtomicBoolean(false);
    for (int i = 0; i < capacity; ++i) {
      table[i] = new ArrayList<>();
    }
  }
```

The class uses a list to represent the hash table. Each element of this list is a list itself. We initialize each element to an empty list.

Note that this class is *abstract*. The strategy of how to synchronize a shared state is left open for the derived classes.

The following diagram shows the initialization pictorially:



The add(v) method

The `add(v)` method comes next, as shown in the following code. Note that it uses the `lock(x)` and `unlock(x)` methods, which are abstract:

```
public boolean add(T x) {
  boolean result = false;
  lock(x);
  try {
    int bucket = x.hashCode() % table.length;
    if (!table[bucket].contains(x)) {
    table[bucket].add(x);
    result = true;
    size.incrementAndGet();
    if (table[bucket].size() >= LIST_LEN_THRESHOLD)
    needsToResize.set(true);
      }
  } finally {
        unlock(x);
    }
    if (shouldResize())
      resize();
    return result;
}
```

The idea is simple. First, we acquire the lock on the mutable state—the table—by calling the `lock(x)` method. Read on to see why we pass in the `x` element to the `lock()` and `unlock()` methods.

The following line computes the bucket for the x element, by taking its hash code modulo to the table size:

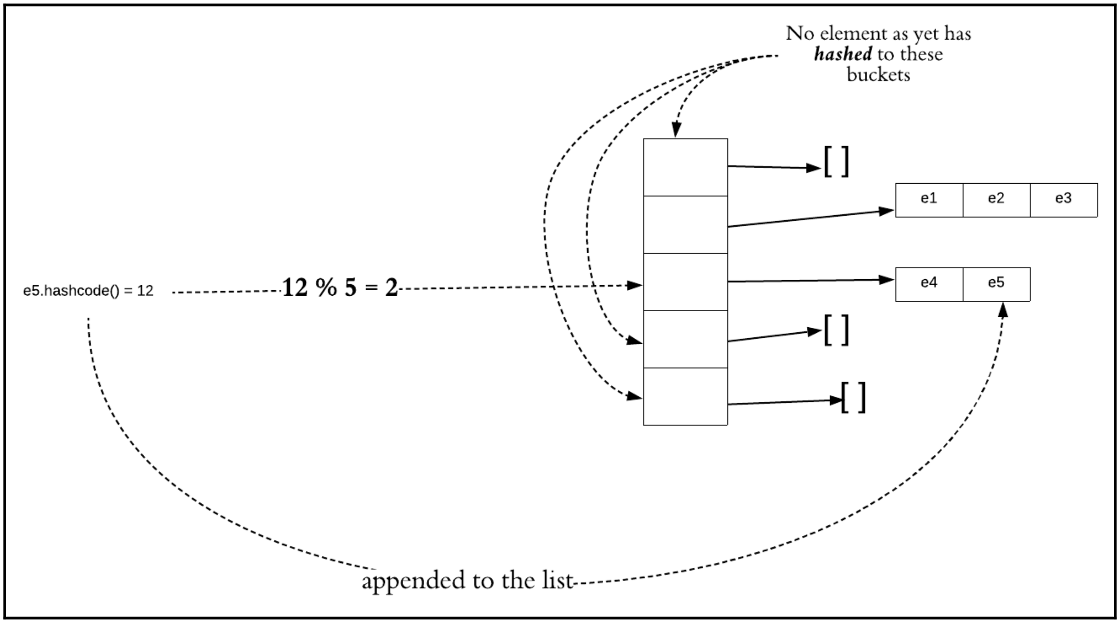
```
int bucket = x.hashCode() % table.length;
```

Next, it tries to add the element into the list that is held in the array index, `bucket`:

```
if (!table[bucket].contains(x)) {
  table[bucket].add(x);
  result = true;
  size++;
}
```

The `table[bucket]` is an array list. We look up the element in the list. If it is present, we cannot add it again (remember the uniqueness guarantee?), so we return `false`. Otherwise, we add the element to the list, increase the `size` instance field to reflect the fact that the set has got one more element, and return `true`.

The following diagram will help us understand this operation. The set already holds **{e1, e2, e3, e4}**, and we are trying to add **e5**. Let's say that `e5.hashCode()` gives 12, and 12 % of 5 gives us 2. So, we index into the table at index 2 and append element **e5** to the list that is found there:



The method ends with a `finally`—as noted in earlier chapters, wrapping the code in a `try` or `finally` will make sure that the lock is released, no matter how the execution returns. You can wrap the code in this way, as shown in the following code:

```
        } finally {
            unlock(x);
        }
        if (shouldResize())
          resize();
        return result;
    }
```

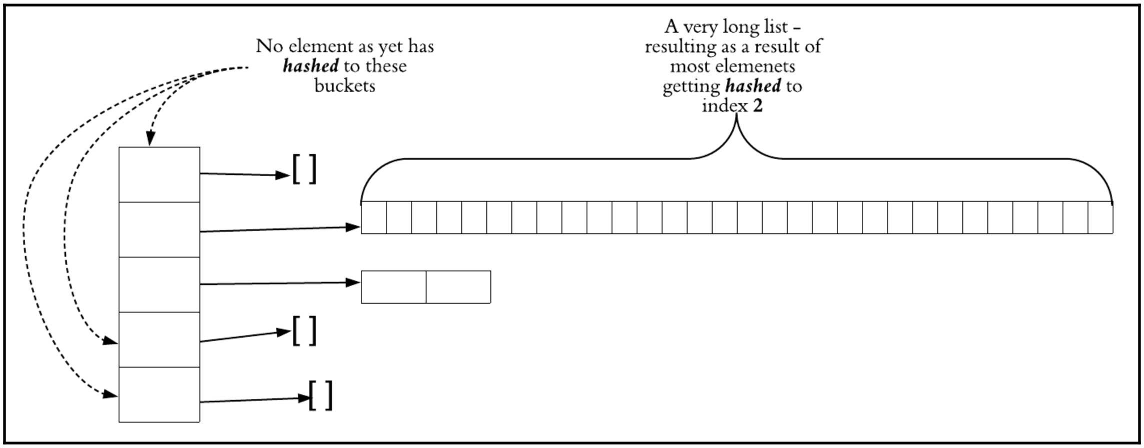
Both the `shouldResize()` and `resize()` methods are abstracts. The mechanics of when and how to resize is left with derived classes.

The need to resize

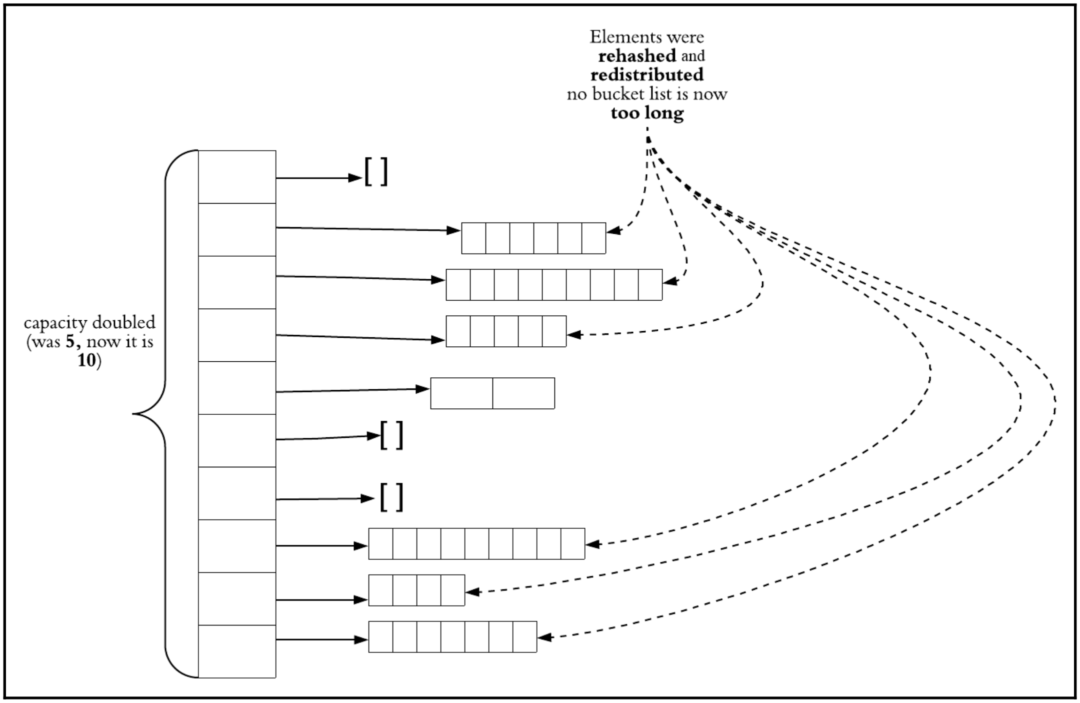
Why do we need this resizing? Note that each bucket holds a list of elements. If far too many elements hash to the same bucket, we would end up searching a very long, *unsorted* list, resulting in degraded performance. The complexity of searching an element in an unsorted list is *O(n)*.

See `https://www.studytonight.com/data-structures/time-complexity-of-algorithms` for a refresher on algorithm complexities.

The complexity involved in algorithm searches is shown in the following diagram:


No element as yet has ***hashed*** to these buckets
A very long list - resulting as a result of most elemenets getting ***hashed*** to index **2**
[ ]
[ ]
[ ]

The idea is to redistribute the elements by reallocating the buckets array to twice its capacity and rehashing all the elements again. This will ensure more of a uniform distribution of elements to all bucket lists. Upon a resize, a (desirable) restructuring happens, as shown in the following diagram:


Elements were **rehashed** and **redistributed** no bucket list is now **too long**
capacity doubled (was **5,** now it is **10**)
[ ]
[ ]
[ ]

Now, as most lists have sort of similar lengths, the lookup will not degenerate.

When do we decide to use `resize()`? This is a policy decision, implemented by the `shouldResize()` method in the subclass.

As shown in the preceding diagram, we would resize when any one bucket list becomes too long—that is, when its length exceeds a certain global threshold.

The contains(v) method

The `contains(v)` method searches for an element in the hash set. If the element is found, the `contains(v)` method returns `true`; otherwise, it returns `false`. The method is shown in the following code:

```
public boolean contains(T x) {
  lock(x);
  try {
    int myBucket = x.hashCode() % table.length;
    return table[myBucket].contains(x);
  } finally {
      unlock(x);
    }
}
```

We have already seen the mechanics of this. There are two steps involved:

1. Compute the hash to arrive at the bucket index.
2. Each array entry holds a linked list. We search in the linked list using its `contains(x)` method.

Now, as the background is nicely laid out and under our belt, let us look at the various strategies and patterns to expose this hash set in a thread-safe manner.

The big lock approach

Our first design, the big lock design, allows only one thread at a time! The following class illustrates this:

```
public class BigLockHashSet<T> extends HashSet<T> {
  final Lock lock;
  final int LIST_LEN_THRESHOLD = 100;

  public BigLockHashSet(int capacity) {
    super(capacity);
    lock = new ReentrantLock();
}
```

As shown in the preceding code, the class is a subclass of `HashSet<T>` and uses a `ReentrantLock`. As noted earlier, a reentrant lock is one that allows the owner thread to reacquire it. The `LIST_LEN_THRESHOLD` constant is used in the `shouldResize()` method that is described in the next section.

The `lock()` and `unlock()` methods are overridden, and they just ignore the `x` parameter. The methods are shown in the following code:

```
@Override
protected void unlock(T x) {
  lock.unlock();
}

@Override
protected void lock(T x) {
  lock.lock();
}
```

As shown in the preceding code, this implementation just locks and unlocks the reentrant lock; it does not make any use of the element that is passed in!

The resizing strategy

As noted earlier, we resize the hash set when any one list becomes too long—that is, when it crosses a threshold. The following code shows both of these methods:

```
private boolean recheck() {
  for (List<T> list : table) {
    if (list.size() >= LIST_LEN_THRESHOLD) {
      return true;
    }
  }
    return false;
}

@Override
protected boolean shouldResize() {
  return needsToResize.get();
}
```

The shouldResize() method just returns the value of the needsToResize flag. The recheck() method just iterates over all the lists and compares their lengths against the threshold. Next comes the actual resize() method:

```
@Override
protected void resize() {
  lock.lock();
  try {
    if (shouldResize() && recheck()) {
      int currCapacity = table.length;
      int newCapacity = 2 * currCapacity;
      List<T>[] oldTable = table;
      table = (List<T>[]) new List[newCapacity];
      for (int i = 0; i < newCapacity; ++i)
      table[i] = new ArrayList<>();
      for (List<T> list : oldTable) {
        for (T elem : list) {
          table[elem.hashCode() % table.length].add(elem);
        }
      }
    }
} finally {
        lock.unlock();
  }
}
```

In the resize() method, we acquire the lock and then call shouldResize() again! It is possible that some other thread could have done the resizing, as there is a time window between the different calls.

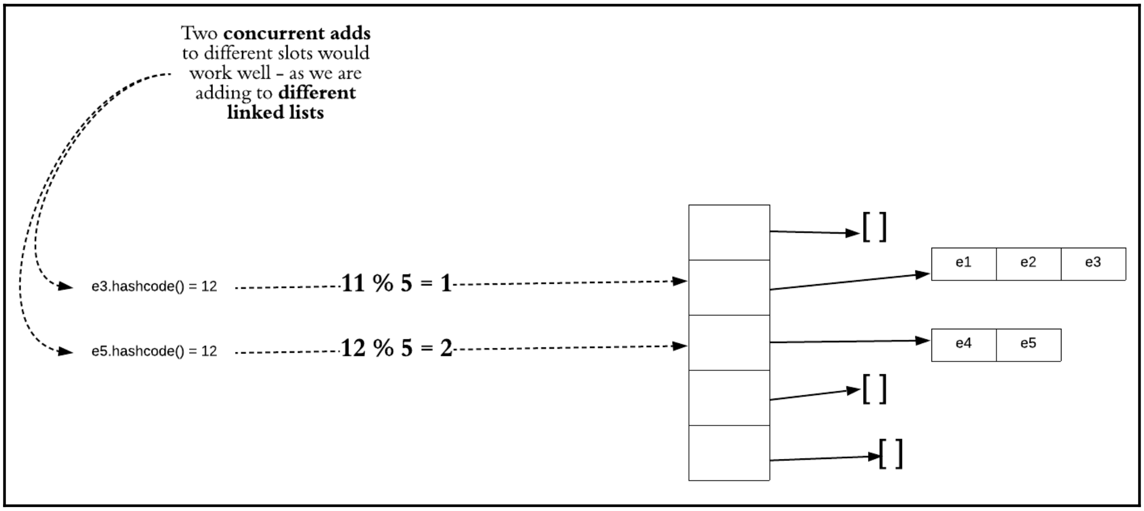
If we need to resize, we allocate a new table of twice the capacity of the original one. We iterate all the elements of the original table and rehash and reinsert the elements into the new table.

Why do we need the recheck() method call? It could have been the case that, between our calling add and performing the resize, some other thread might have done the resizing, so we check once that the resizing conditions are holding well before performing the resize.

The lock striping design pattern

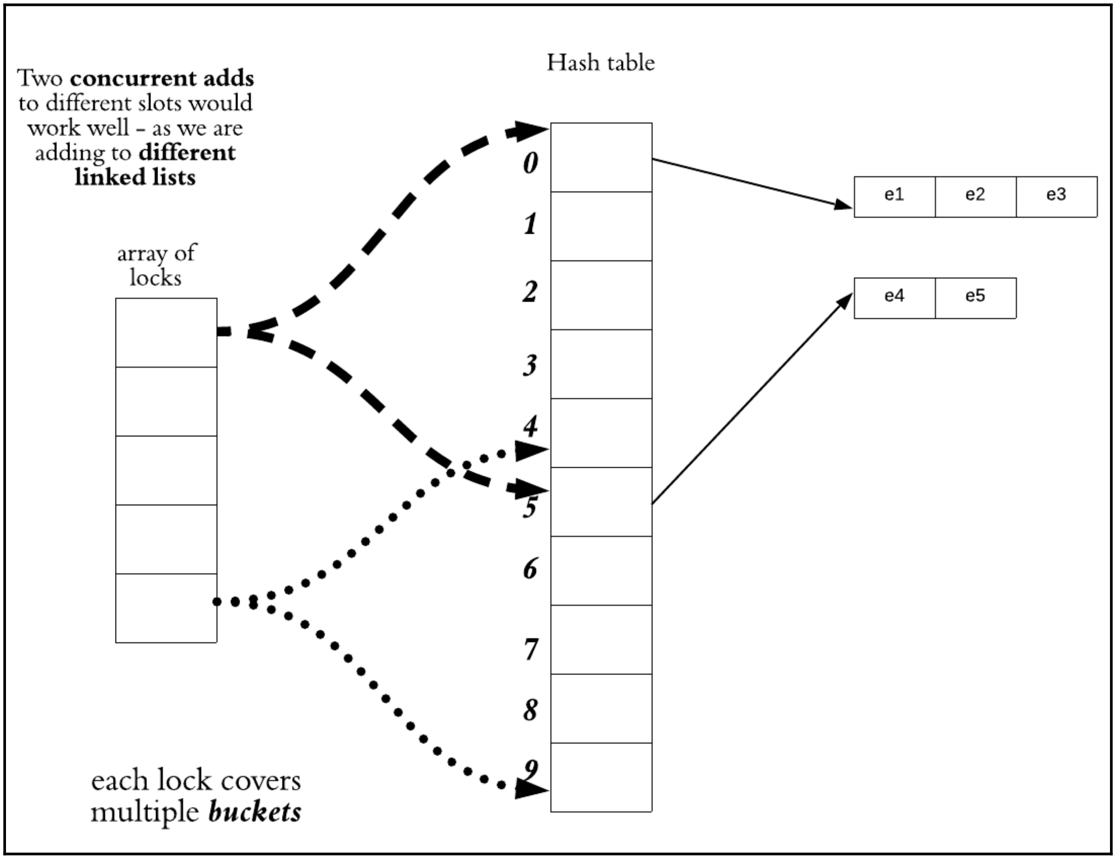
What is the problem with the resizing strategy? It hurts concurrency. We have actually reduced the affair to a strictly sequential execution, which is a bottleneck. As a goal, we should strive for allowing more concurrency threads to work on the hash set, which still ensures thread safety!

The following diagram shows two concurrent adds. As both the threads work on *separate* shared mutable states, we could allow both of them the access:


Two concurrent adds to different slots would work well - as we are adding to different linked lists
e3.hashcode() = 12
11 % 5 = 1
e5.hashcode() = 12
12 % 5 = 2
[ ]
e1
e2
e3
e4
e5
[ ]
[ ]

Two (or more) threads could be adding elements to *different* buckets, while at the same time other threads could be searching the hash set.

The *lock striping* pattern allows us to do just that. The following diagram shows how the pattern works:


Two **concurrent adds** to different slots would work well - as we are adding to **different linked lists**
Hash table
array of locks
0
1
2
3
4
5
6
7
8
9
e1
e2
e3
e4
e5
each lock covers multiple ***buckets***

Instead of a single lock, we maintain an *array of locks*. The following code shows the algorithm:

```
x <- compute the hash for the element
lock(x % length of locks array) // 5 for the above diagram
insert the element for the list at (x % length of table) // 10 for the
above
```

The following is the code for this arrangement:

```
public class LockStripedHashSet<T> extends HashSet<T> {
  final Lock[] locks;

  public LockStripedHashSet(int capacity) {
    super(capacity);
    locks = new Lock[capacity];
    for (int i = 0; i < locks.length; ++i) {
      locks[i] = new ReentrantLock();
    }
  }
```

The number of `locks` in the `locks` array is initially equal to the capacity (the number of buckets in the hash table). However, as the resizing happens, we just reallocate the buckets array and table as before, *but we do not reallocate the* `locks` *array.*

So, as the number of elements in the hash set grows, the locks go on covering more buckets and elements, as shown in the following code:

```
@Override
protected void lock(T x) {
  locks[x.hashCode() % locks.length].lock();
}

@Override
protected void unlock(T x) {
  locks[x.hashCode() % locks.length].unlock();
}
```

As noted in the preceding algorithm, the element hash code is used to refer to a lock. The `lock` and `unlock` methods use the hash to reach the lock that is associated with the element.

The reason that both the `lock` and `unlock` methods accept the element that is being hashed is that the earlier version did not make any use of it, but this version needs the element for the locking to work. This is shown in the following code:

```
@Override
protected void resize() {
  for (Lock lck: locks) {
    lck.lock();
  }
  try {
    if (shouldResize() && recheck()) {
      int oldCapacity = table.length;
      int newCapacity = 2 * oldCapacity;
      List<T>[] oldTable = table;
      table = (List<T>[]) new List[newCapacity];
        for (int i = 0; i < newCapacity; ++i)
          table[i] = new ArrayList<>();
          for (List<T> bucket : oldTable) {
            for (T x : bucket) {
              table[x.hashCode() % table.length].add(x);
            }
          }
```

```
           }
        needsToResize.set(false);
      } finally {
          for (Lock lck: locks) {
            lck.unlock();
          }
      }
  }
  private boolean recheck() {
    for (List<T> list : table) {
      if (list.size() >= LIST_LEN_THRESHOLD) {
        return true;
      }
    }
    return false;
  }
```

The important thing to remember is that we require *all the locks* to do the resizing. This is needed to ensure that we have exclusive access to the table and all the lists that are needed to reallocate the table and redistribute all the elements.

What about any possible deadlocks? What if two or more threads call `resize()`? Note that we always lock the locks array in order. As noted in earlier chapters, acquiring the locks in order prevents any deadlocks.

What if another thread already owns the lock, possibly reading or writing in the list? In that case, the resizing will wait for the thread to free up the lock.

Summary

This chapter covered some well-known patterns in concurrent programming. We focused on how to increase the concurrency of data structures, allowing multiple threads to make progress. Using implicit (or explicit) synchronization is a bottleneck, so we explored the alternatives.

We looked at lock-free data structures, using the *compare and set* (CAS) primitive provided by Java's concurrent library. We implemented a lock-free LIFO stack and then the more involved lock-free queue. We looked at and compared both variants of the queue: a lock-based queue and a lock-free queue.

Lock-free algorithms are more complex than their lock-synchronized counterparts. We looked at the `AtomicReference`, the basis for these CAS-based algorithms. We also looked at the kind of situation where the ABA problem happens, and how the `AtomicStampedReference` solves it.

Finally, we looked at *hashing* and how the lock striping concurrency pattern helps us to increase the concurrency for hash tables.

Armed with all this knowledge, let's look at immutability and functional programming, and how this exciting paradigm helps us write better concurrent programs. Stay tuned!

6
Functional Concurrency Patterns

In the shared state model, problems start cropping up due to state being a mutable one. We learned how hard it becomes to correctly synchronize the thread state, keeping in mind the ability to work with correctness, starvation, and deadlocks.

In this chapter, we will look at concurrency patterns, largely from a functional perspective. **Functional Programming** (**FP**) is a functional paradigm and a cornerstone of FP is immutability. We will use Scala to study this aspect. Immutable data structures use structural sharing and persistent data structures to ensure performance along with safety guarantees.

We will also look at future abstraction as a representation of asynchronous computation. Asynchronous computations use threads in an optimum way. A Scala future is also a monad, offering composability.

Here is what this chapter will cover:

- Immutability
- Futures

This coverage should prepare us well to understand the actor paradigm, coming up in the next chapter.

Immutability

Immutable objects are thread-safe. When an object is immutable, you cannot make changes to the object. Many threads can read the object at the same time and when a thread needs to change a value, it creates a modified copy. For example, Java strings are immutable. Consider the following code snippet:

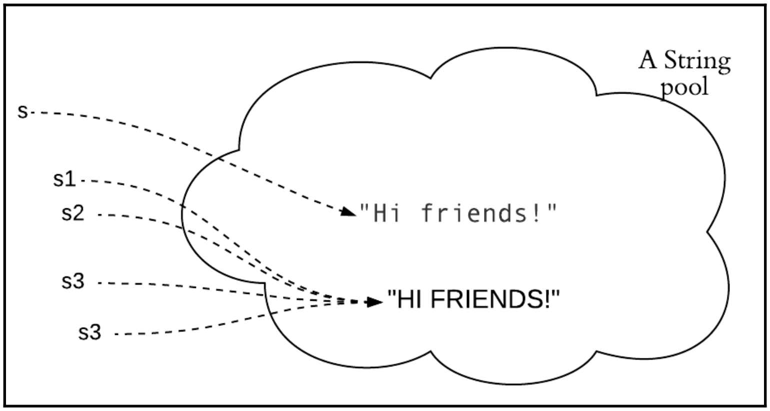
```
import java.util.HashSet;
import java.util.Set;

public class StringsAreImmutable {
  public static void main(String[] args) {
    String s = "Hi friends!";
    s.toUpperCase();
    System.out.println(s);

    String s1 = s.toUpperCase();
    System.out.println(s1);

    String s2 = s1;
    String s3 = s1;
    String s4 = s1;

    Set set = new HashSet<String>();
    set.add(s);
    set.add(s1);
    set.add(s2);
    set.add(s3);

    System.out.println(set);
  }
}
```

Running the code gives the following output:

```
Hi friends!
 HI FRIENDS!
 [Hi friends!, HI FRIENDS!]
```

When we invoke the `toUpperCase()` method on the string variable `s`, as the output shows, there is no mutation happening in place. As a result, the first call result is lost. The next invocation stores the result into another variable, `s1`, which contains the uppercase string, as expected. Adding `s2` and `s3` does not change anything as we have already added `s1`. (Note that a `Set` always hold unique elements).

The following diagram shows the conceptual pooling and sharing of objects:



As shown in the preceding figure, Java strings are pooled. As the strings are immutable, they can be readily shared. As shown, all `s1`, `s2`, `s3`, and `s4` variables refer to the same string object in memory. When we invoke the `toUpperCase()` method, we get another, modified copy of the string. The benefits of this copy-on-write scheme are obvious. Threads can modify strings without worrying about thread safety. As the code shows, due to the immutability guarantee, strings can readily become members of a set. There is no danger of someone changing the object without our knowledge and violating the contract set. (See `https://www.programcreek.com/2013/09/java-hashcode-equals-contract-set-contains/` for more information.)

Immutability helps create programs that are easier to understand.

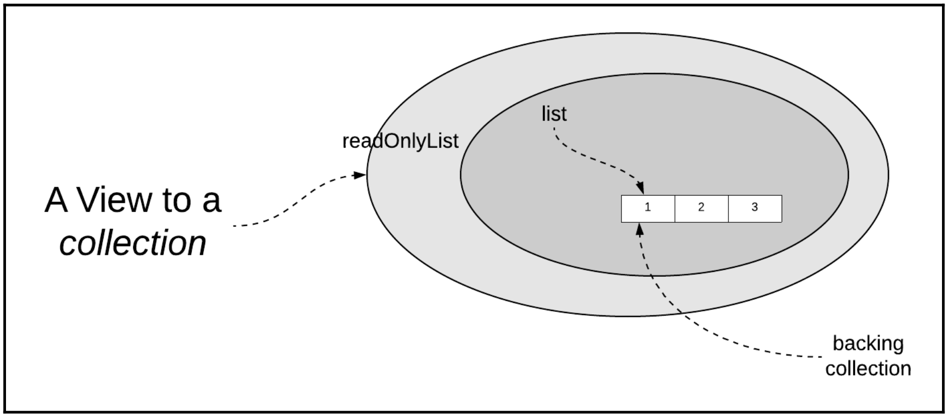
Unmodifiable wrappers

The following code shows Java's unmodifiable collections—a way to grow immutability on mutable collections. This is an example of the *decorator* pattern (`https://sourcemaking.com/design_patterns/decorator/java/1`):

```
package com.concurrency.book.chapter07;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class UnmodifiableWrappers {

  public static List<Integer> createList(Integer... elems) {
    List<Integer> list = new ArrayList<>();
    for(Integer i : elems) {
      list.add(i);
    }
        return Collections.unmodifiableList(list);
  }

  public static void main(String[] args) {
    List<Integer> readOnlyList = createList(1, 2, 3);
    System.out.println(readOnlyList);
    readOnlyList.add(4);
  }
}
```

The `createList()` method creates, fills up, and wraps up the `ArrayList` , which is a mutable collection . This way of writing code makes sure that we don't leak the mutable list inadvertently. The original, mutable list is a **backing object**. The wrapper controls operations on this backing object.

We create a list and print its content. As this is a `read` operation, it succeeds. The decorator, `readOnlyList`, forwards any read calls on to the mutable object. However, when we try to mutate the list, we get an exception:



The output is as follows:

```
[1, 2, 3]
 Exception in thread "main" java.lang.UnsupportedOperationException
 at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
 at
com.concurrency.book.chapter07.UnmodifiableWrappers.main(UnmodifiableWrappe
rs.java:20)
```

Note that this wrapping make only the list immutable. In the example code, the list contains integers, which themselves are immutable. However, if we had a list of mutable objects—they themselves are mutable. See `https://stackoverflow.com/questions/37446594/when-creating-an-immutable-class-should-collections-only-contain-immutable-obje` for information.

We need to make both the collection and its elements immutable. This would ensure that the data structure is thread safe for all kinds of access. In the following Scala code, both of these requirements are met.

If you need to change the contents, you first need to copy the list and then change the contents while the new list is being created. Won't this hurt performance? Let us look at some functional Scala code and try to understand the performance.

Persistent data structures

The term **persistence** does not mean disk persistence. The term refers to multiple versions of the same data structure being maintained. The multiple versions come into existence due to copy-on-write semantics, any unused versions are garbage collected.

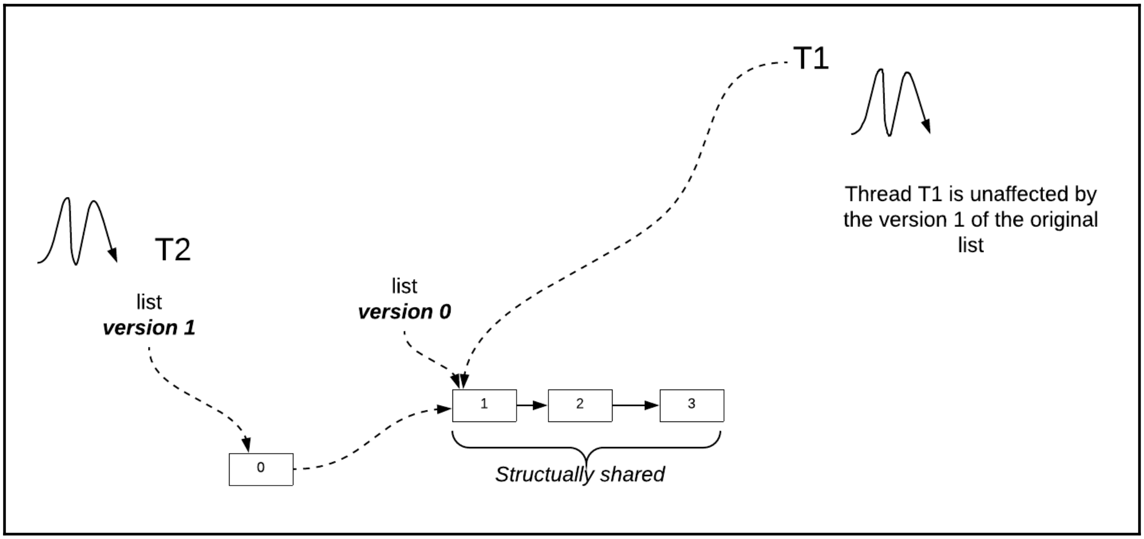
The following code shows an immutable Scala list and the `append(elem)` and `prepend(elem)` methods. Note that both of these methods need to copy and create a modified *version* of the data structure.

Both methods are recursive. The code does not use any mutable iterators at all. For a `prepend` operation, we pattern match against the list and if the list is empty, we create a new list with a single element.

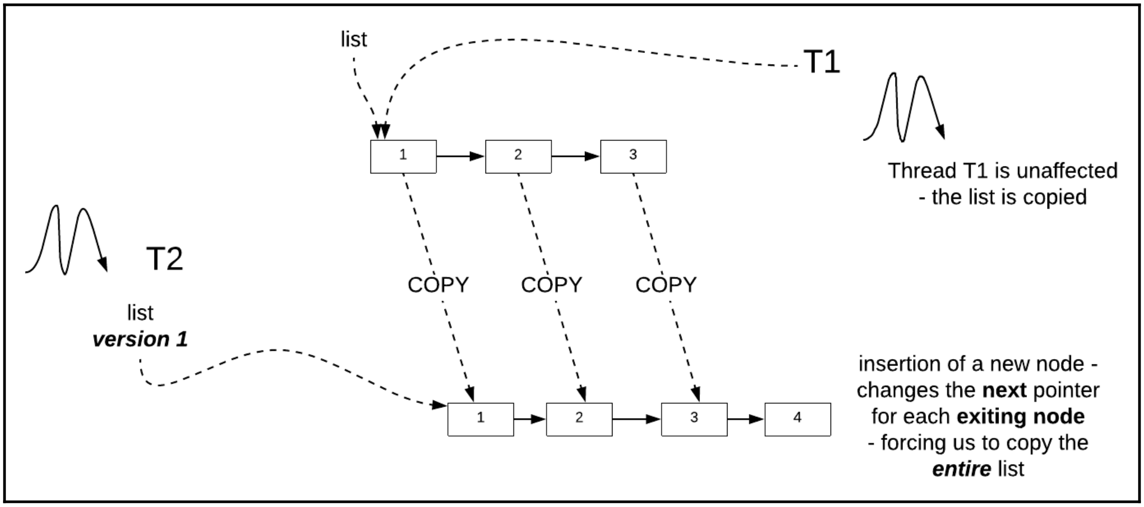
The interesting part is when the list is not empty. We just tack the new element at the head of the list. This creates a new version of the data structure; the original list is structurally shared between both the versions. The algorithmic complexity of this operation is *O(1)*:

```
package com.concurrency.book.chapter07

object ListOps extends App {

  def prepend( elem: Int, list: List[Int] ) = list match {
    case Nil => List(elem)
    case _ => elem :: list
  }

  def append( elem: Int, list: List[Int] ): List[Int] = list match {
    case Nil => List(elem)
    case x :: xs => x :: append(elem, xs)
  }

  val list = List(1, 2, 3)
  println(prepend(0, list))
  println(append(4, list))

}
```

The `prepend` operation is shown in the following diagram:


T1
Thread T1 is unaffected by the version 1 of the original list
T2
list
version 1
list
version 0
1
2
3
0
Structually shared

The `append` operation, on the other hand, has to copy all of the list. When we are appending to a read-only list, structural sharing is not possible as each node's next pointer gets changed. The reason is seen in the following diagram:



The complexity for an append is *O(n)* due to all the nodes getting copied. This is the reason we try avoiding list appends. If the list is too large, there will be too much copying. While designing programs, we need to look at this algorithmic aspect carefully while ensuring thread safety.

Recursion and immutability

In Scala, whenever you use a collection, by default it is an immutable collection. These are auto imported and are ready to use. There are mutable collections too, however, you have to specifically import them:

```
scala> val list = List(1,2,3)
 list: List[Int] = List(1, 2, 3)
 scala> list.append(4)
<console>:14: error: value append is not a member of List[Int]
list.append(4)
scala> import scala.collection.mutable.ListBuffer
 scala> val list1 = ListBuffer(1, 2, 3)
list1: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3)
scala> list1.append(4)
scala> list1
res3: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3, 4)
```

To keep track of iterations, we use recursion to implement traversal and looping. This style avoids any mutable state such as counters. Recursion can lead to stack overflow issues though.

The following code snippet shows a tail recursive implementation of a program that counts elements in an immutable list:

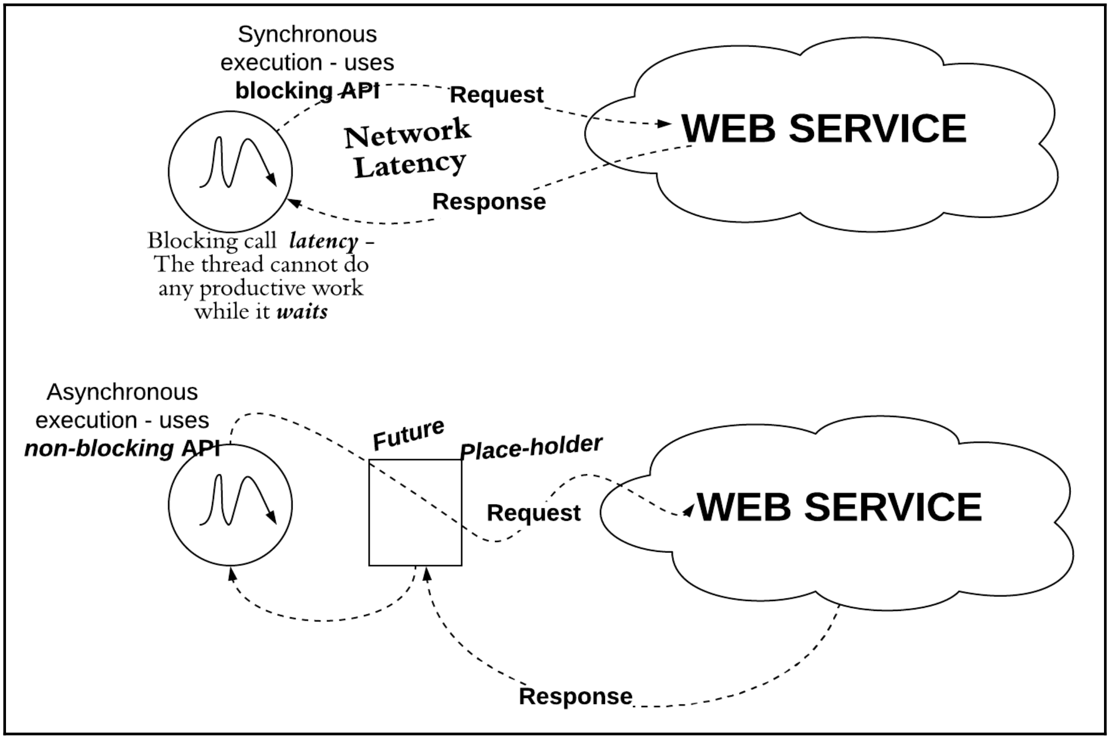
```
package com.concurrency.book.chapter07

import scala.annotation.tailrec

object CountListElems extends App {

  def count(l: List[Int]): Int = {

    @tailrec
    def countElems(list: List[Int], count: Int): Int = list match {
      case Nil => count
      case x :: xs => countElems(xs, count + 1)
    }

    countElems(l, 0)
  }

  println(count(List(1, 2, 3, 4, 5)))
  println(count(( 1 to 100000 ).toList))
}
```

The code uses the accumulator idiom to implement a tail recursive version. **Tail Recursion Optimization** (**TCO**) kicks in to avoid the stack overflow issues. The `@tailrec` Scala annotation makes sure the code is really tail recursive.
See `https://alvinalexander.com/scala/fp-book/tail-recursive-algorithms` for more information on this topic.

As we will see from now on, a mutable state is avoided. We will instead use immutability all along.

Futures

If a thread blocks, for example, a thread waiting for an I/O operation to complete, this is wasteful. The following diagram shows a thread using a **blocking API call** for getting a response from a web service over the network. The sequential execution model needs to wait as the flow cannot proceed otherwise. On the other hand, asynchronous execution does not block the calling thread. A future is used for expressing such asynchronous computations. The following diagram shows how it works:


Synchronous execution - uses **blocking API**
Request
WEB SERVICE
Network Latency
Response
Blocking call *latency* – The thread cannot do any productive work while it *waits*
Asynchronous execution - uses ***non-blocking API***
Future
Place-holder
Request
WEB SERVICE
Response

As shown, a future is a placeholder. It will *eventually* contain the response or can timeout if the call takes too long to complete.

How does the calling thread work with the future though?

The apply method

A future is a trait in the `scala.concurrent` package, which has an accompanying companion *object.* This companion provides the `apply` method, the signature of which is as follows:

```
        def apply[T](body: => T)(implicit executor: ExecutionContext):
   Future[T]
```

The `body: => T` syntax is a *by-name* parameter. This is a **curried** form of the method. See `https://dzone.com/articles/understanding-currying-scala` for more on this technique.

Implicit parameters are like normal parameters. You can explicitly specify them as normal parameters. You can also choose not to specify them. In this case, the Scala compiler searches for a value in the surrounding scope.

by-name parameters

The following code explains the construct:

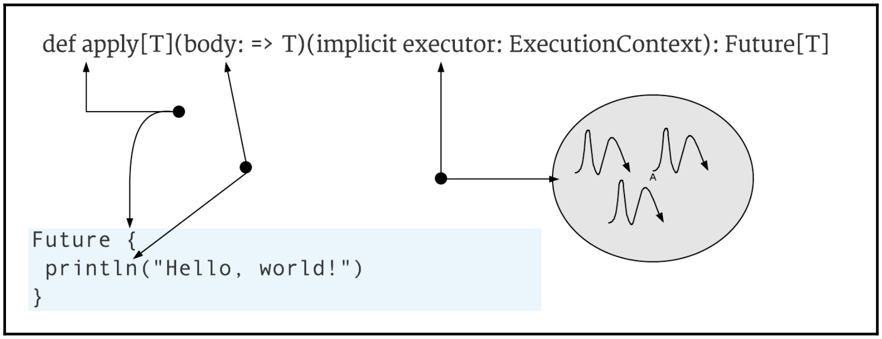
```
package com.concurrency.book.chapter07

object Eval extends App {
  def eagerEval( b: Boolean, ifTrue: Unit, ifFalse: Unit ) =
    if ( b )
      ifTrue
    else
      ifFalse

  def delayedEval( b: Boolean, ifTrue: => Unit, ifFalse: => Unit ) =
    if ( b )
      ifTrue
    else
      ifFalse

      eagerEval(9 == 9, println("9 == 9 is true"), println("9 == 9 is
false"))
      println("---------")
      delayedEval(9 == 9, println("9 == 9 is true"), println("9 == 9 is
false"))
      delayedEval(9 != 9, println("9 == 9 is true"), println("9 == 9 is
false"))
}
```

The intent of the code is to simulate conditional execution. If the method's first argument is `true`, we wish to evaluate the second argument. Otherwise, we evaluate the third argument. Here is the output:

```
9 == 9 is true
 9 == 9 is false
 ---------
 9 == 9 is true
 9 == 9 is false
```

As shown, for the `eagerEval(...)` method, both arguments are evaluated at the **call site**. This defeats the intent—the second method, `delayedEval(...)`, uses Scala's by-name parameters. Note the => placed after the type and colon.

The *body* parameter is a by-name parameter. The code passed in is invoked from the `Future.apply(...)` method:



```
def apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]
```

```
Future {
 println("Hello, world!")
}
```

The following code shows how a thread uses a future by using its `apply` method:

```
package com.concurrency.book.chapter07

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object WorkingWithAFuture extends App {
  Future { println("Hello, world!") }
  println("How are things?")
  Thread.sleep(2000)
}
```

The output is shown as follows:

```
How are things?
Hello, world!
```

Note that the future block's output was printed after the main thread's output. It could be the other way around also—so essentially futures execute in a non-deterministic fashion.

For this particular run on my machine, the thread running the future was scheduled to run after the main thread. The main thread created the future. It did not wait for it (this makes it asynchronous) and then continued running its logic which was to print a message. The future's thread executes next, prints its message, and the future exits.

Why do we need `sleep` at the end? Let us look at how futures map to threads in the next section.

Future – thread mapping

The last part of the `apply(...)` method is an implicit execution context. An execution context is something that can execute a future. It is essentially a thread pool. A future is run on a thread from this pool. If you recall the discussion of thread pools from `chapter 4`, *Thread Pools*, you shall remember the fork join thread pool.

Scala provides a global `ExecutionContext` which uses a `ForkJoinPool`. You don't have to worry about tasks anymore though, the global `ExecutionContext` takes the future computation and wraps it in a fork join task.

The following code shows how an execution context makes it simple to map a future to a thread: as the preceding diagram shows, an execution context decides how to map a future to a thread and run it. The following code uses the global execution context:

```
package com.concurrency.book.chapter07

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object FutWithDefaultContext extends App {

  Future {
    Thread.sleep(2000)
    println("Hello, world!")
  }
  println("How are things?")
}
```

The output is as follows:

```
How are things?
```

Whatever happened to the future? The problem is the default execution context runs your futures on daemon threads. Java's daemon threads don't stop a JVM from exiting. (See `https://stackoverflow.com/questions/2213340/what-is-a-daemon-thread-in-java` for a refresher on daemon threads.)

That made the last `Thread.sleep(...)` call necessary to the main thread. The main thread needs to sleep for a few seconds to allow the daemon thread running the future to complete.

In the following code snippet, we will use an execution context with non-daemon threads:

```
package com.concurrency.book.chapter07

import java.util.concurrent.Executors

import scala.concurrent.{ExecutionContext, Future}

object FutWithMyContext extends App {
  implicit val execContext =
ExecutionContext.fromExecutor(Executors.newCachedThreadPool)

  Future {
    Thread.sleep(2000)
    println("Hello, world!")
  }
  println("How are things?")
}
```

`newCachedThreadPool` is used instead. This makes the future run on a non-daemon thread. As a result, the future completes successfully and you get the desired output.

When the threads in `newCachedThreadPool` are not used for 60 seconds, they are terminated. If you wait for a minute, you will see that the program terminates. Here is the output:

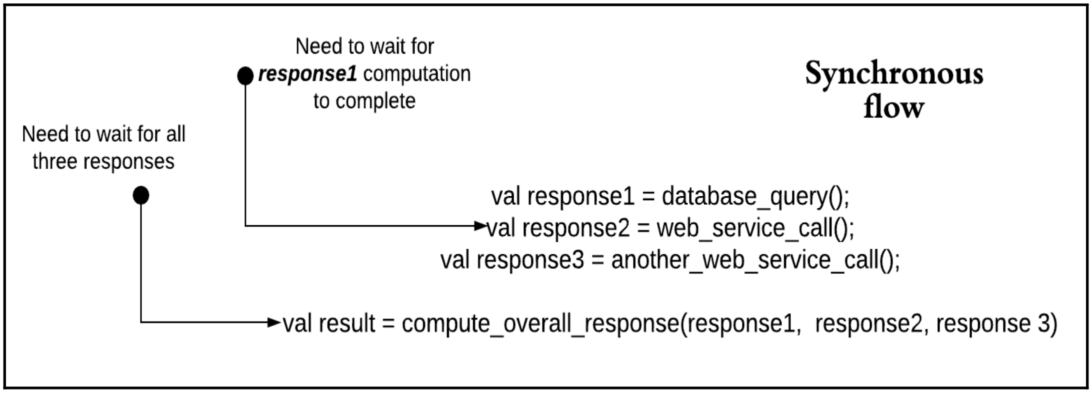
```
How are things?
 Hello, world!
```

Given this background, let us look at how multiple futures overlap the execution.

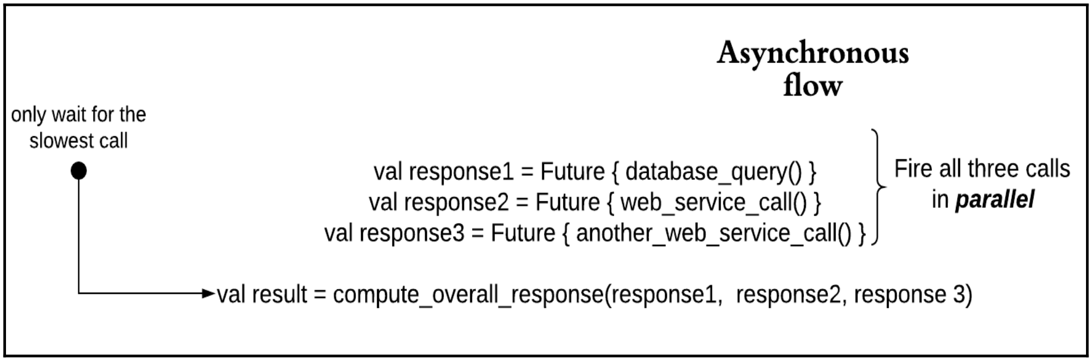
Futures are asynchronous

Whenever we think of a program flow, we have a mental model—statements execute one after another. The flow waits for the current computation to complete before it starts working with the next statement. The following diagrams compare three computations.

The first diagram shows three sequential computations, with only one computation executing at any time. The latency of an operation is the time spent after the request is fired and before the response is obtained. As each computation has a certain latency, the overall latency of the result computation is equal to all three latencies added together:


Need to wait for
response1 computation
to complete
Synchronous
flow
Need to wait for all
three responses
val response1 = database_query();
val response2 = web_service_call();
val response3 = another_web_service_call();
val result = compute_overall_response(response1, response2, response 3)

The following diagram shows the same computations expressed via an asynchronous flow. All three requests are fired at the same time, without waiting for each other to complete. Only when the overall result is computed do we need all three values. However, now, as the latencies of the three operations *overlap*, the overall latency is the maximum of all three computations:


Asynchronous
flow
only wait for the
slowest call
val response1 = Future { database_query() }
val response2 = Future { web_service_call() }
val response3 = Future { another_web_service_call() }
Fire all three calls
in ***parallel***
val result = compute_overall_response(response1, response2, response 3)

The following program simulates calls to external services. The `longRunningMethod` method sleeps for a specified number of seconds. We will try to time the sequential invocations of this method:

```
package com.concurrency.book.chapter07
```

```
object SynchronousExecution extends App {

  def longRunningMethod( i: Int ) = {
    Thread.sleep(i * 1000)
    i
  }

  val start = System.currentTimeMillis()

  val result = longRunningMethod(2) + longRunningMethod(2) +
longRunningMethod(2)

  val stop = System.currentTimeMillis()

  println(s"Time taken ${stop - start} ms")

  println(result)

  Thread.sleep(7000)
}
```

When we run the program, the output is around 6 seconds, as shown:

```
Time taken 6001 ms
 6
```

This is hardly surprising, as all three calls sleep for 2 seconds each. The overall latency adds up. The next code snippet shows the same computation, expressed using futures:

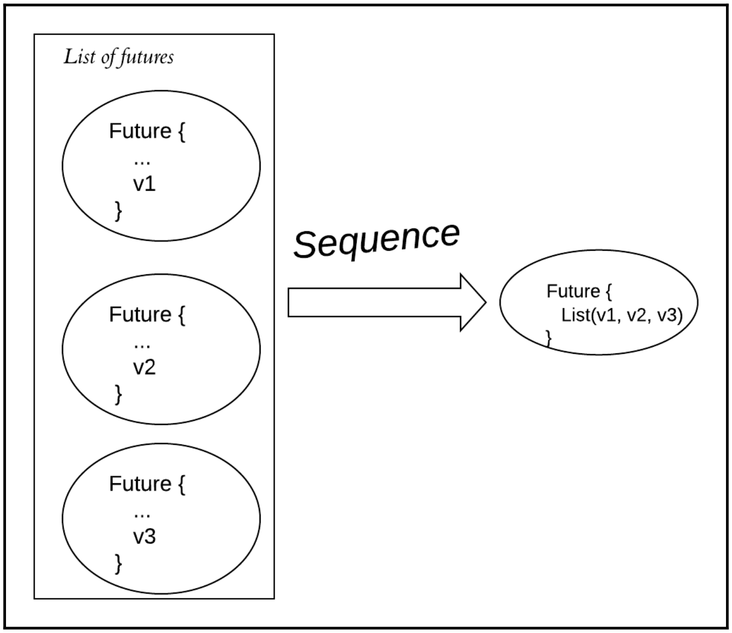
```
package com.concurrency.book.chapter07

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.concurrent.{Await, Future}

object AsynchronousComputation extends App {

  def longRunningMethod( i: Int ) = Future{
    Thread.sleep(i * 1000)
    i
  }

  val start = System.currentTimeMillis()

  val f1 = longRunningMethod(2)
  val f2 = longRunningMethod(2)
  val f3 = longRunningMethod(2)
```

```
    val f4: Future[List[Int]] = Future.sequence(List(f1, f2, f3))

    val result = Await.result( f4, 4 second)

    println(result.sum)

    val stop = System.currentTimeMillis()
    println(s"Time taken ${stop - start} ms")

    Thread.sleep(4000)
    }
```

The output is as follows:

```
  6
   Time taken 2297 ms
```

Wait a second! What is that `Future.sequence` method? The `Future.sequence(...)` method takes a list of futures and converts that to a future of list values. The following diagram should make it clear:



The sequence operation returns another future. We await for all futures to complete and then get back the list, packed with all result values.

Blocking is bad

Some APIs are inherently blocking ones. For example, JDBC does not have a non-blocking API. In such cases, there is a provision to hint the blocking context to create more threads.

The following code illustrates the problem:

```
package com.concurrency.book.chapter07

import scala.concurrent.duration.Duration
import scala.concurrent.{Await, Future}
import scala.concurrent.ExecutionContext.Implicits.global

object BlockingFutures extends App {
  val start = System.currentTimeMillis()

  val listOfFuts = List.fill(16)(Future {
    Thread.sleep(2000)
    println("-----")
  })

  listOfFuts.map(future => Await.ready(future, Duration.Inf))

  val stop = System.currentTimeMillis()
  println(s"Time taken ${stop - start} ms")
  println(s"Total cores = ${Runtime.getRuntime.availableProcessors}")
}
```

We create a list of 16 futures. Each future's computation sleeps for 2 seconds and then prints a string. By default, the threads in the default execution context are equal to the number of cores available.

We use a Scala idiom to create a list with 16 futures. Here is a Scala REPL session, to illustrate the idiom:

```
scala> val random = new scala.util.Random()
random: scala.util.Random = scala.util.Random@3d98d138

scala> val rndNumList = List.fill(16)(random.nextInt(20))
rndNumList: List[Int] = List(18, 1, 19, 5, 5, 4, 18, 4,
8, 14, 8, 2, 2, 3, 16, 16)
```

Next, we await the future execution. Once all futures complete, we print the time taken.

The output is as follows:

```
-----
 -----
 -----
 -----

 delay

 -----
-----
-----
-----
delay
...
Time taken 8291 ms
Total cores = 4
```

As four threads block due to the `sleep()` call, the rest of the futures don't get any threads to run on. As there are four batches, we get each blocking for 2 seconds.

Now change the code as follows:

```
import scala.concurrent.{Await, Future, blocking}
// rest of the code, as before...
blocking {
 Thread.sleep(2000)
  println("-----")
}
// rest of the code as before
```

We have just wrapped up the blocking piece inside the `blocking` method invocation. When you run the code next, the delay is gone:

```
-----
 -----
 -----
 -----
 // 12 more lines
 Time taken 2337 ms
 Total cores = 4
```

The `blocking` method made the global execution context spawn additional threads when it detected that there were not sufficient threads to do the work.

Functional composition

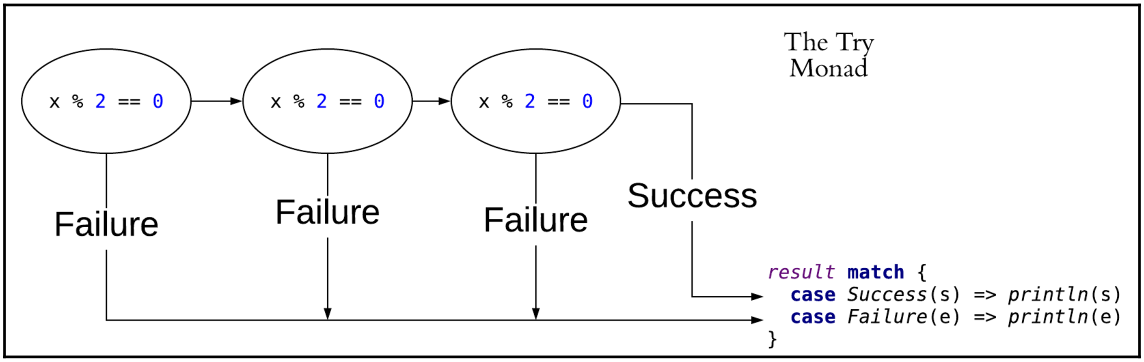
In functional programming, simpler values are composed into more complex ones by using higher-order functions. These higher-order functions are called **combinators**. For example, the map method on a collection produces a new collection containing elements from the original collection, mapped with a specified function.

The following code shows a validation pipeline created using Scala's `Try` monad. All of the checks execute one after another. If any of the checks fail, the pipeline processing stops—as in other checks down the line are skipped:

```
package com.concurrency.book.chapter07

import scala.util.{Failure, Success, Try}

object TryAsAMonad extends App {
  def check1(x: Int) = Try {
    x match {
      case _ if x % 2 == 0 => x
      case _ => throw new RuntimeException("Number needs to be even")
    }
  }

  def check2(x: Int) = Try {
    x match {
      case _ if x < 1000 => x
      case _ => throw new RuntimeException("Number needs to be less than
1000")
    }
  }

  def check3(x: Int) = Try {
    x match {
      case _ if x > 500 => x
      case _ => throw new RuntimeException("Number needs to be greater than
500")
    }
  }

  val result = for {
    a <- check1(400)
    b <- check2(a)
    c <- check3(b)
  } yield "All checks ran just fine"

  result match {
    case Success(s) => println(s)
```

```
        case Failure(e) => println(e)
      }
    }
```

We have composed validations together, resulting in a pipeline. The following diagram shows the composition:



Note that we are stringing together these operations using Scala's `for` comprehension, which is just syntactic sugar for a succession of `flatMap` calls, terminating with a map. The following listing shows the lengthy and hard-to-read version:

```
val result = check1(400).flatMap(a => check2(a).
  flatMap(b => check3(b).map(c => "All checks ran just fine")))
```

A future is also a monad. The following code shows how we could use `for` comprehensions for manipulating them:

```
package com.concurrency.book.chapter07

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.{Await, Future}
import scala.concurrent.duration._

object FutureWithForComprehension extends App {

  def longRunningMethod( i: Int ) = Future{
    Thread.sleep(i * 1000)
    i
  }

  val start = System.currentTimeMillis()

  val f1 = longRunningMethod(2)
```

```
    val f2 = longRunningMethod(2)
    val f3 = longRunningMethod(2)

    val f4: Future[Int] = for {
      x <- f1
      y <- f2
      z <- f3
    } yield ( x + y + z )

    val result = Await.result( f4, 4 second)

    println(result)

    val stop = System.currentTimeMillis()
    println(s"Time taken ${stop - start} ms")

    Thread.sleep(4000)
  }
```

Running this code gives the following output:

```
6
 Time taken 2334 ms
```

This is as expected. Note that the futures need to be started outside of the comprehension. Let us change the code a bit, as shown:

```
// Don't use this version...
val f4: Future[Int] = for {
 x <- longRunningMethod(2)
 y <- longRunningMethod(2)
 z <- longRunningMethod(2)
} yield ( x + y + z )
```

Lo and behold! The `Await` call is `TimeoutException`:

```
Exception in thread "main" java.util.concurrent.TimeoutException: Futures
timed out after [4 seconds]
 at scala.concurrent.impl.Promise$DefaultPromise.ready(Promise.scala:255)
 ...
```

The problem is we have got all of the futures executing sequentially! Till the first future completes, the second is not even started. Writing down the de-sugared version of the comprehension clarifies the issue:

```
val f4 = longRunningMethod(2).flatMap(x => longRunningMethod(2).
  flatMap(y => longRunningMethod(2).map(z => x + y + z)))
```

As seen,when the first future does not complete, its `flatMap()` method is not called. The time adds up and we get a timeout.

Summary

We looked at two important themes in this chapter. An immutable value is never changed once it is constructed. Immutability rules out, by design, any shared state issue as you cannot change the values. Immutable code makes for increased thread safety. Copy-on-write is used when a thread needs to modify an immutable data structure. We looked at persistent data structures, which are multiple versions (copies) of the same data structure. Structural sharing helps ensure algorithmic performance.

Next, we looked at Scala's futures, an abstraction used to express asynchronous computations. We saw how futures map with threads, and how to avoid blocking the underlying thread. Futures allows for functional composition—a functional design pattern for creating processing pipelines.

With all this know-how, let's now look at the **Actor** paradigm. Stay tuned!

7
Actors Patterns

In the preceding chapter, we discussed functional concurrency patterns using various code-examples. In this chapter, we will discuss actors patterns and functionality with the help of various code-examples.

In `Chapter 1`, *Concurrency – An Introduction,* we looked at two major paradigms for designing concurrent systems. The shared state paradigm requires careful state management. We provided reasons for using locks for safely sharing the state—we have to be aware of visibility and consistency issues.

Locking makes state manipulations *atomic.* However, this approach could result in locking bottlenecks. As an alternative, we took a look at *lock-free* data structures. However, lock-free structures are inherently complex.

In this chapter, we will take a look at the message-driven paradigm for creating concurrent programs. The shared state goes away. Instead, actors encapsulate the state, and the only way to change the state is via sending a message to it.

Message driven concurrency

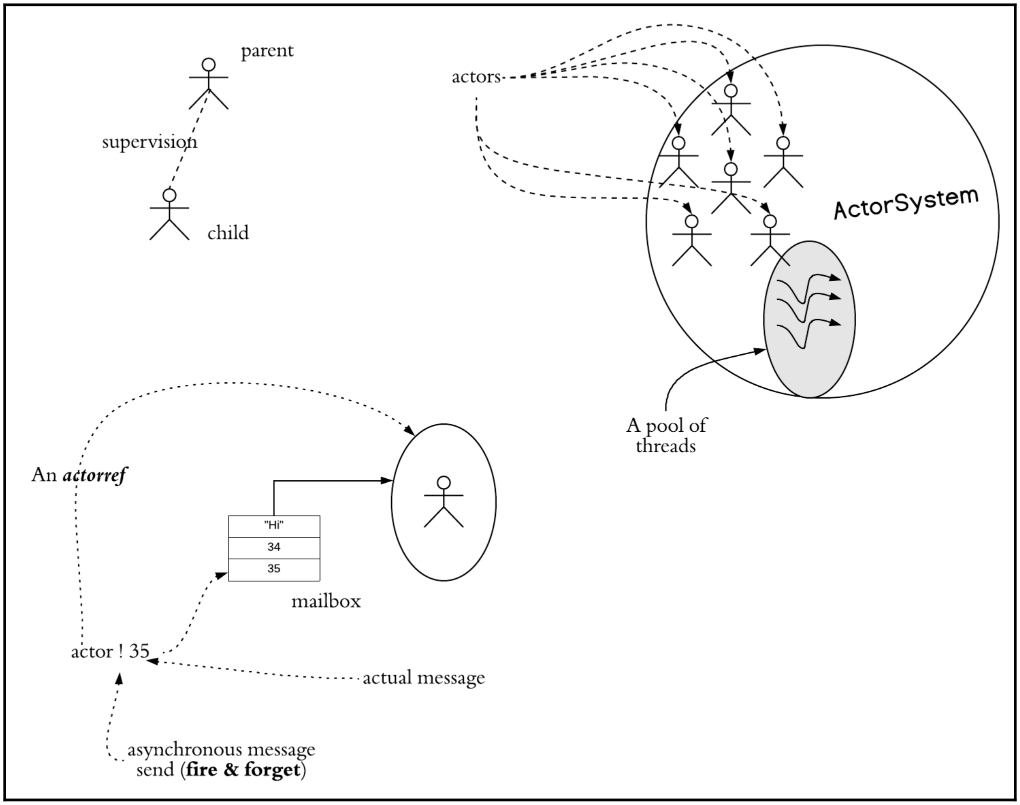
The following listing shows our first actor-based program in action. We show a single *actor*, living in an *actor system* and we talk to this actor by sending messages. These messages land in the actor's *mailbox* and each message is processed *sequentially,* one at a time:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorSystem, Props}

class MyActor extends Actor {

  override def receive: PartialFunction[Any, Unit] = {
    case s: String => println(s"<${s}>")
    case i: Int => println(i+1)
  }

}

object MyActor extends App {
  def props() = Props(new MyActor)

  val actorSystem = ActorSystem("MyActorSystem")

  val actor = actorSystem.actorOf(MyActor.props(), name = "MyActor")

  actor ! "Hi"
  actor ! 34

  case class Msg( msgNo: Int)

  actor ! Msg(3)

  actor ! 35

  actorSystem.terminate()
}
```

We have an actor, named `MyActor`, which lives in an actor system named `MyActorSystem`. We create the actor and hold its `actorReference` in the actor variable.

Next, we send messages to the actor, using the asynchronous message send method, curiously named as `!`. This `!` operator is called as **tell operator**. This is a **fire & forget** calling mechanism. We keep firing messages, without waiting for any replies.

We send four messages, `Hi`, `34`, `Msg(3)`, and `35`:



The actor has a `receive` method, where the messages received are processed. We process all strings and all integers.

The `receive` method returns a partial function object of the `PartialFunction[Any, Unit]` type. If the partial function is defined, that is, if the pattern match succeeds, the message is processed by the matching block. Otherwise, the message is discarded.

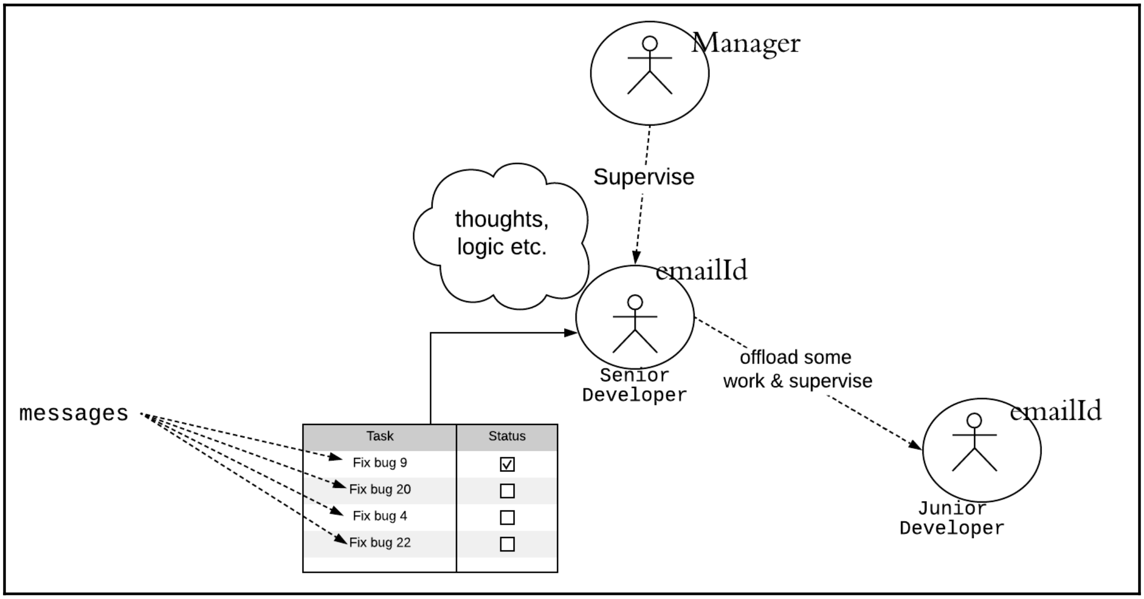
If we run the code, we get the following output:

```
<Hi>
 35
 36
```

Note that the `Msg(3)` message is ignored, as there is no pattern match for it.

What is an actor?

An actor has *state, behavior, a mailbox, child actors*, and *a parent*, which is also a *supervisor*. The only way to send messages to an actor is via its *actor reference*. An actor system contains actors and a thread pool on which the actors are dispatched:



Task	Status
Fix bug 9	☑
Fix bug 20	□
Fix bug 4	□
Fix bug 22	□

Let's consider a software development company. It is instructive to equate people in the company with the actor model. Let's consider the following three people (actors): a manager, a senior developer reporting to him, and a junior developer reporting to the senior manager.

An actor reference is the equivalent of an email ID. A person could be in the office (co-located) or traveling and you can still send a *message* to them using their email ID. They can also reply back to the sender.

The state is equivalent to the thoughts of any of these people. It is well encapsulated within the person. The junior developer is supervised by the senior developer, who, in turn, is supervised by the manager.

The mailbox is a todo list. The manager may assign work as and when it comes to the senior developer, that is, send messages using some form of collaboration tool such as Jira. The developer will take up each task from the todo list and start working on it.

In the actor code example, some of these entities are very much at play. You cannot see the mailbox directly in the code. It is working behind the scenes, keeping the message processing loop running.

The following line initializes the variable actor with an actor reference. As we use *type inference* in Scala, we won't see the type explicitly written:

```
val actor = actorSystem.actorOf(MyActor.props(), name = "MyActor")
```

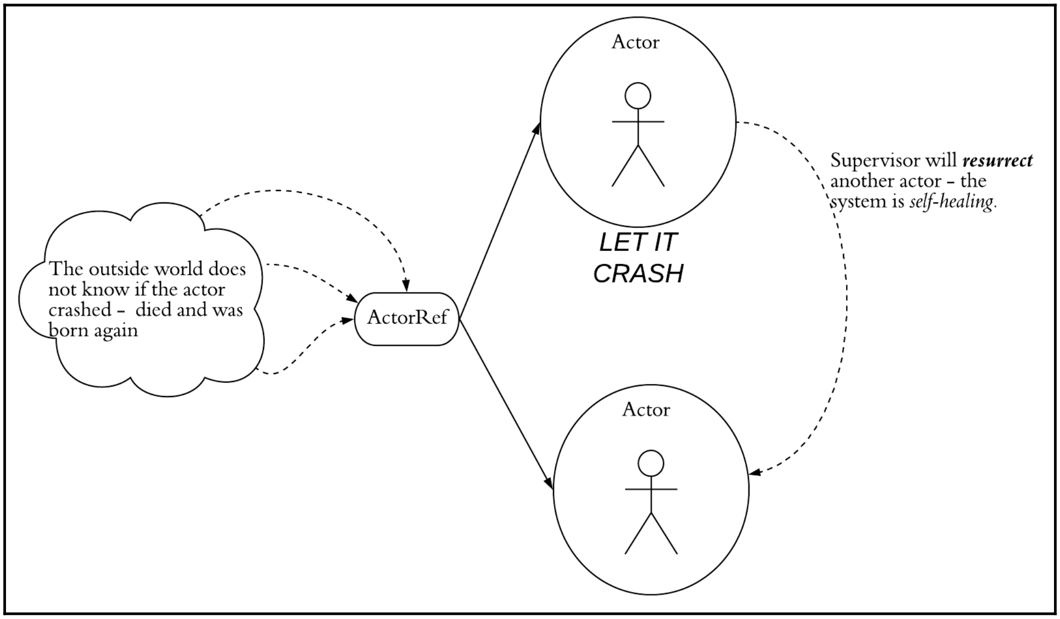
However, the line really means the following:

```
val actor: ActorRef = actorSystem.actorOf(MyActor.props(), name =
"MyActor")
```

So, the `actor` variable is an actor reference. Let's take a look at what this actor reference provides.

Let it crash

An actor reference encapsulates the actual actor object reference. The idea is to insulate the callers if the actor crashes and restarts. We will soon take a look at an example of this actor resurrection. The following figure shows the let-it-crash philosophy:


Actor
Supervisor will ***resurrect*** another actor - the system is *self-healing.*
LET IT CRASH
The outside world does not know if the actor crashed - died and was born again
ActorRef
Actor

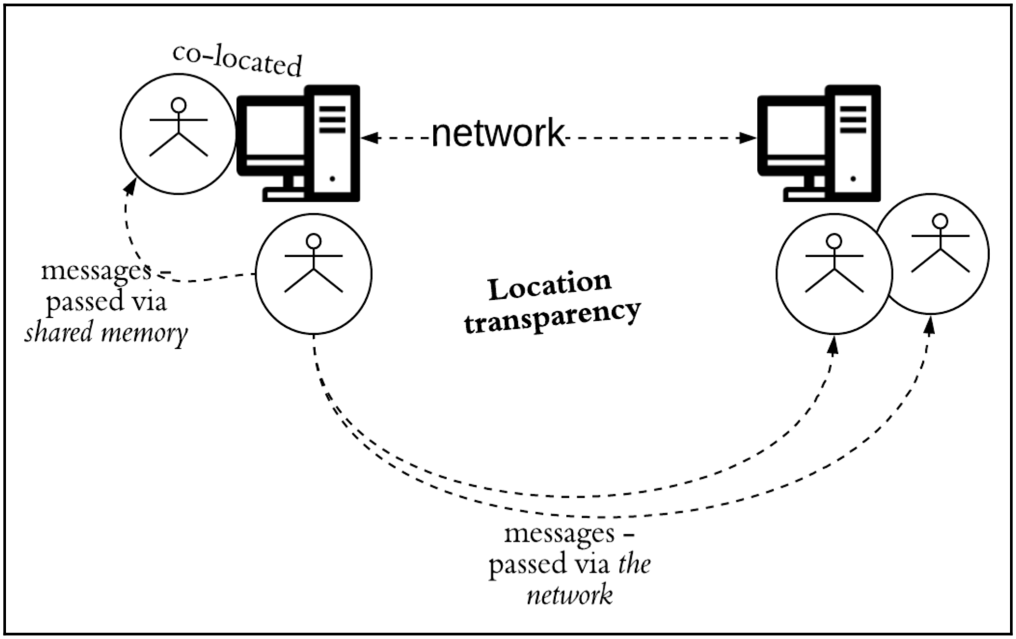
If anything goes wrong with the actor, the philosophy is let it crash. The actor's *supervisor,* which is also its parent, replaces the actual JVM object with another copy of the same actor. That is the reason a client does not hold the JVM object reference, and instead uses a *proxy*—`ActorRef`.

This is an example of a proxy design pattern. Check out `https://www.geeksforgeeks.org/proxy-design-pattern/` for more information on this topic.

Location transparency

You should be able to redeploy actors so that some of them run on different machines. Co-located actors run on the same machine. While you are developing, co-location would work well. However, you may need to horizontally scale the production deployment. Refer to `Chapter 1`, *Concurrency – An Introduction*, for an explanation of horizontal scaling.

The following diagram shows how an actor reference helps maintain Location transparency:


co-located
network
messages - passed via *shared memory*
Location transparency
messages - passed via *the network*

The idea is to deploy actors remotely—the code sending messages should work the same. As long as the actor is reachable, the code should remain unaffected. The *location transparency* is one major factor for using the actor reference. We can *deploy* the actors anyway we see fit—one machine is not the boundary. We could easily use a cluster instead.

Actors are featherlight

As we have seen earlier, threads are costly to create, and they also tear down. So, we use thread pools, and reuse threads for various tasks. Actors, on the other hand, have a very light memory footprint. You can create millions of actors.

Here is an example: we create `10000` actors of the `MyActor` class and send each actor a message. The message is a number, which will help us ensure that these actors were indeed created:

```
package com.concurrency.book.chapter08

import akka.actor.ActorSystem
import com.concurrency.book.chapter08.MyActor.actorSystem

object ThousandsOfActors extends App {
  val actorSystem = ActorSystem("MyActorSystem")

  (1 to 10000).
    map(k => k -> actorSystem.actorOf(MyActor.props(), name =
s"MyActor${k}")).
    foreach { case (k, actor) => actor ! k }

  Thread.sleep(14000)
  actorSystem.terminate()
}
```

We have a range of numbers—`1 to 10000`. We map over each number and create an actor corresponding to each number in the collection. So, in all, we created `10000` actors. We generate a list of *tuples*—each tuple is a pair—of the form `(num, actorRef)`.

Finally, we will send the number as a message to `actorRef` with the following output:

```
11
 9
 13
 15
 7
 10
... // you will find 9999, 10000 somewhere
```

Try to modify the program to print the actor name and the thread it runs on.

State encapsulation

What do we mean by state encapsulation? In object-oriented programming, we use a class and its private variables. For example, as long as the public APIs of the class work as per the contract, the class internals don't matter to the outside world. Similarly, how an actor does its work should not matter to anyone, as long as the task is completed as expected.

Here is an example:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorSystem, Props}

class CountMessagesActor extends Actor with ActorLogging {

  var cnt = 0

  override def receive: PartialFunction[Any, Unit] = {
    case s: String =>
      cnt += 1
      println(s"${cnt} <${s}>")
    case i: Int =>
      cnt += 1
      println(s"${cnt} ${i + 1}")
  }

}

object CountMessagesActor extends App {
  def props() = Props(new CountMessagesActor())

  val actorSystem = ActorSystem("MyActorSystem")

  val actor = actorSystem.actorOf(CountMessagesActor.props(), name =
"CountMessagesActor")

  actor ! "Hi"
  actor ! 34

  case class Msg( msgNo: Int )

  actor ! Msg(3)

  actor ! 35
```

```
    actorSystem.terminate()
  }
```

We have a shared mutable state in the form of a `count` variable. The variable keeps track of the total number of messages received and processed till now. As indicated by *var,* the variable is mutable and represents the state of the actor. If we run the code, we will note the following output:

```
2 <Hi>
3 35
4 36
```

As you see, there is no way we can change the variable from outside. We need to send a message to the actor, and it is up to the actor to change the state the way it thinks is appropriate.

Where is the parallelism?

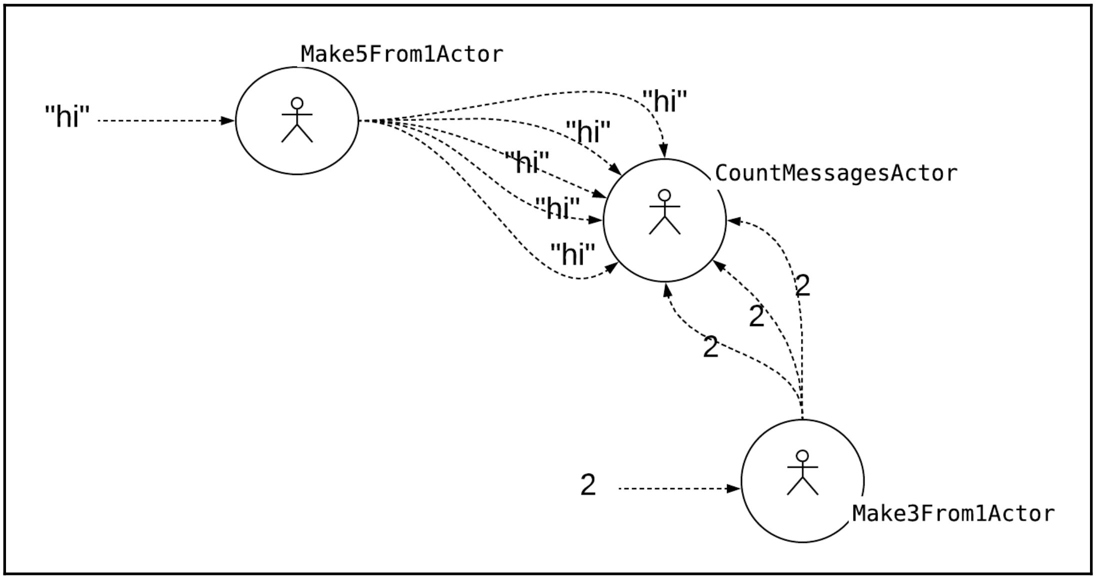
Where is the concurrency in all this? Where are threads? We showed a thread pool before this. Why don't we see threads anywhere? Actors are higher abstractions and are run over threads. The following listing shows how the actors are dispatched to run using threads:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, Props}

class Make5From1Actor( actor: ActorRef) extends Actor with ActorLogging {
  override def receive: Receive = {
    case s =>
      log.info(s"Received msg $s")
      (0 to 4).foreach(p => actor ! s)
  }
}

class Make3From1Actor( actor: ActorRef) extends Actor with ActorLogging {
  override def receive: Receive = {
    case s =>
      log.info(s"Received msg $s")
      (0 to 2).foreach(p => actor ! s)
  }
}

object MultipleActorsHittingOneActor extends App {
  val actorSystem = ActorSystem("MyActorSystem")
```

```
    val actor = actorSystem.actorOf(CountMessages.props(0), name =
  "CountMessagesActor")

    def props5From1() = Props(new Make5From1Actor(actor))
    def props3From1() = Props(new Make3From1Actor(actor))

    val actor1 = actorSystem.actorOf(props5From1(), name = "Actor5From1")
    val actor2 = actorSystem.actorOf(props3From1(), name = "Actor3From1")

    actor1 ! 34
    actor1 ! "hi"
    actor2 ! 34
    actor2 ! "hi"

    Thread.sleep(1000)
    actorSystem.terminate()
  }
```

We have two actor classes, namely `Make3From1` and `Make5From1`. The first receives a message and sends it three times to the `CountMessages` actor. The second one does the same, however, sends it five times instead.

These actors also receive a reference in the constructor. This is the actual destination actor for these messages. So, any message received, in turn, is forwarded to the destination actor. The following diagram shows the message flow and actor communication:



When we run this code, we will receive the following output (trimmed to show the relevant information):

```
[INFO]... Received msg 34
[INFO]... Received msg 34
[INFO]... Received msg 34 - cnt = 1
 ... rest of the output snipped
```

The preceding output is instructive. All three actors run on three different threads. The name of the thread is `MyActorSystem-akka.actor.default-dispatcher-4`.

Also, note how we are mixing the `ActorLogging` trait. This gives a preconfigured logger for the actor.

Unhandled messages

When we send a message for which there is no pattern match, the message is discarded. Here is a code snippet illustrating this behavior:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, Props}

class UnhandledMsgActor extends Actor with ActorLogging {
  override def receive: Receive = PartialFunction.empty

  override def unhandled( message: Any ) = message match {
    case msg: Int => log.info(s"I got ${msg} - don't know what to do with
it?")
    case msg => super.unhandled(msg)
  }
}

object UnhandledMsgActor extends App {
  def props() = Props(new UnhandledMsgActor)

  val actorSystem = ActorSystem("MyActorSystem")

  val actor: ActorRef = actorSystem.actorOf(UnhandledMsgActor.props(), name
= "UnhandledMsgActor")

  actor ! "Hi"
  actor ! 12
  actor ! 3.4

  Thread.sleep(1000)
```

```
    actorSystem.terminate()
  }
```

The `receive` method does nothing. It simply does not handle any messages. If we run the program, we will note that the `unhandled` method gets called with the message as an argument. The following will be the output:

```
[INFO]... I got 12 - don't know what to do with it?
```

The `unhandled` method, in turn, treats any `Int` messages specially. All other type of messages are delegated to the super class.

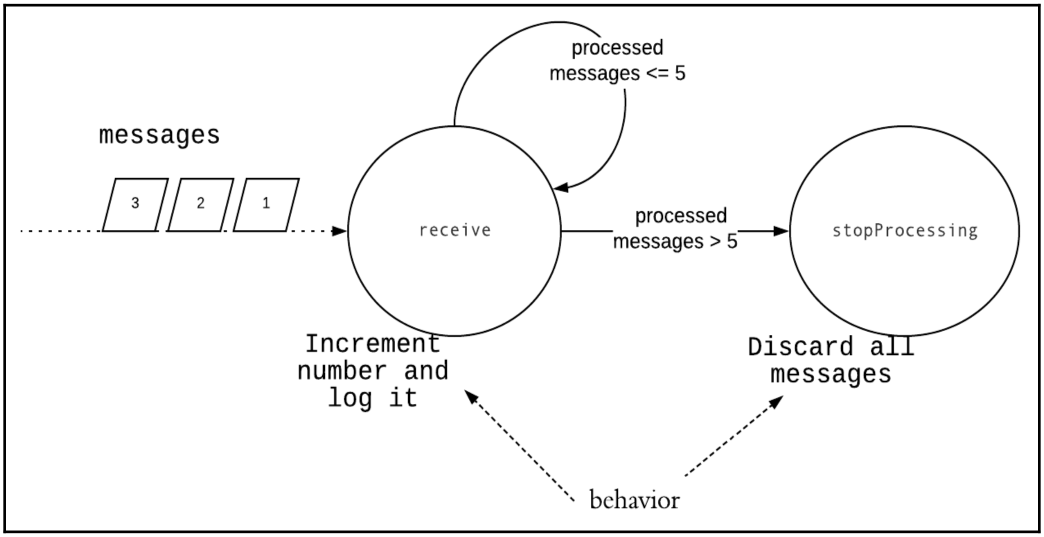
The become pattern

An actor's behavior is defined using the `receive` method. We can change the behavior of the actor using an `become` method on the actor's context object. The following code shows an example of how we can change an actor's behavior:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, Props}

class HandlesOnlyFiveMessages extends Actor with ActorLogging {

  var cnt = 0

  def stopProcessing: Receive = PartialFunction.empty

  override def receive: Receive = {
    case i: Int =>
      cnt += 1
      log.info(s"Received msg $i - cnt = ${cnt}")
      if (cnt == 5) context.become(stopProcessing)
  }

}

object HandlesOnlyFiveMessages extends App {
  def props() = Props(classOf[HandlesOnlyFiveMessages])

  val actorSystem = ActorSystem("MyActorSystem")

  val actor: ActorRef =
actorSystem.actorOf(HandlesOnlyFiveMessages.props(), name =
"HandlesOnlyFiveMessagesActor")
```

```
      (0 to 10).foreach(x => actor ! x)

      Thread.sleep(1000)

      actorSystem.terminate()

    }
```

There are two methods, `receive` and `stopProcessing`. As long as the actor has processed less than five messages, for each message, a number is incremented and logged. Once the message count crosses five, the actor changes its behavior to discard all messages. The flow is shown in the following diagram:



When we run the program, we will receive the following output:

```
[INFO]... Received msg 0 - cnt = 1
[INFO]... Received msg 1 - cnt = 2
[INFO]... Received msg 2 - cnt = 3
[INFO]... Received msg 3 - cnt = 4
[INFO]... Received msg 4 - cnt = 5
```

We sent 10 messages using the following snippet:

```
(0 to 10).foreach(x => actor ! x)
```

However, all messages after five are discarded by the actor.

Making the state immutable

Since we are using Scala, we naturally prefer immutable variables. We want the counter variable to be a `val` variable instead of a `var` variable.However, in that case, how do you update the counter?

Again, we use the *become* pattern. The following is a message counter actor, using an immutable `cnt` variable. Note that, in Scala, all method variables are `val` unless you explicitly declare them as `var`.

```
scala> def m1(p: Int) = p = p + 1
 <console>:11: error: reassignment to val
 def m1(p: Int) = p = p + 1
 ^
 scala>
```

Interestingly, changing an argument is a *code smell* in Java too. Refer to `https://softwareengineering.stackexchange.com/questions/245767/is-modifying-an-incoming-parameter-an-antipattern` for more information. Scala anyway does not allow it, as the preceding snippet shows.

The following program uses this feature, along with the `context.become()` method to implement the counter:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, Props}

class ImmutableCountingActor extends Actor with ActorLogging {

  def process( cnt: Int): PartialFunction[Any, Unit] = {
    case s: String =>
      log.info(s"Received msg $s - cnt = ${cnt+1}")
      context.become(process(cnt+1))
    case i: Int =>
      log.info(s"Received msg $i - cnt = ${cnt+1}")
      context.become(process(cnt+1))
  }

  override def receive: Receive = process(0)
}

object ImmutableState extends App {
  def props() = Props(classOf[ImmutableCountingActor])

  val actorSystem = ActorSystem("MyActorSystem")
```

```
    val actor: ActorRef = actorSystem.actorOf(
      ImmutableState.props(), name = "ImmutableCountingActor")

    (0 to 10).foreach(x => actor ! x)

    Thread.sleep(1000)

    actorSystem.terminate()

  }
```

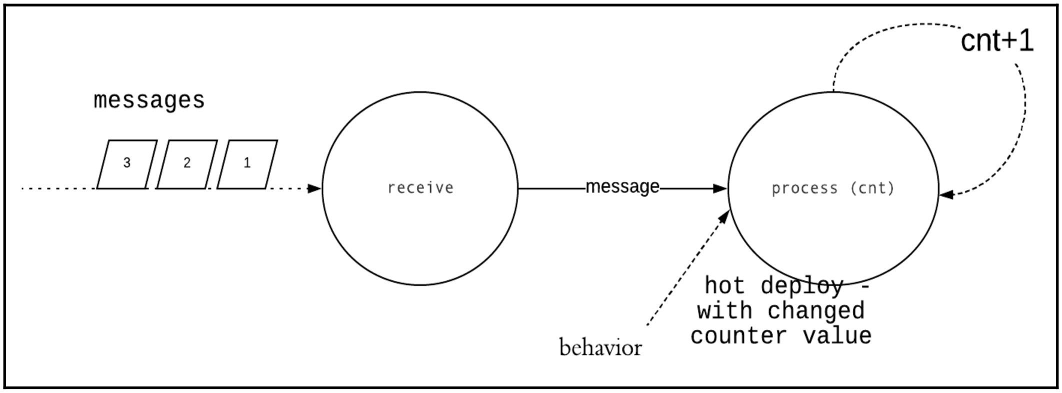
The behavior is changed to reflect the new value of counter. The following single line is all that is needed to effect the change:

```
 context.become(process(cnt+1))
```

This is similar to the way we use recursion in Scala to avoid mutable variables. For example, here is a Scala code snippet to compute the number of elements in a list:

```
package com.concurrency.book.chapter08

import scala.annotation.tailrec

object CountElems extends App {

  def size(l: List[Int]) = {

    @tailrec
    def countElems(list: List[Int], count: Int): Int = list match {
      case Nil     => count
      case x :: xs => countElems(xs, count+1)
    }

    countElems(l, 0)
  }

  val list = List(1, 2, 3, 4, 5)

  println(size(list))
}
```

Compare the preceding version with the *become* pattern. The following diagram shows the flow:



The preceding diagram provides the following output:

```
[INFO]... Received msg 0 - cnt = 1
[INFO]... Received msg 1 - cnt = 2
[INFO]... Received msg 2 - cnt = 3
 ...
```

Let it crash - *and recover*

We mentioned the resilience aspect of the system. The following code is an example of how *let-it-crash* works in Akka. The parent actor is also the supervisor for the child. We can set it up so that, for every actor that crashes, another actor is resurrected and put in place.

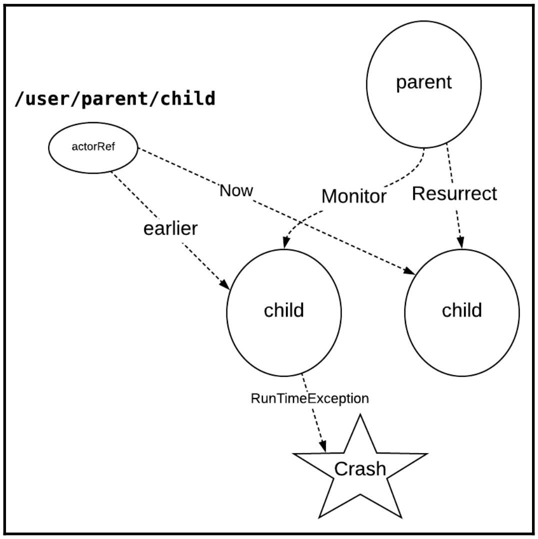
The following code shows how this scheme works:

```
package com.concurrency.book.chapter08

import akka.actor.SupervisorStrategy.{Escalate, Restart}
import akka.actor.{Actor, ActorKilledException, ActorLogging, ActorRef,
ActorSystem, Identify, OneForOneStrategy, Props}
import akka.util.Timeout

import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global
import akka.pattern.ask
import com.concurrency.book.chapter08.Child.BadMessageException
import scala.concurrent.{Await, Future}

class Child extends Actor with ActorLogging {
  import Child._
  def receive = {
    case i: Int => log.info(s"${i + 1}")
    case _ => throw BadMessageException("Anything other than Int messages
is not supported")
  }
}

object Child {
  def props( ) = Props(classOf[Child])

  case class BadMessageException(errStr: String) extends RuntimeException
}

class Supervisor extends Actor with ActorLogging {
  val child = context.actorOf(Child.props(), "child")

  def receive = {
    case _: Any => sender ! child
  }

  override val supervisorStrategy =
    OneForOneStrategy(){
      case e: BadMessageException => Restart
      case _ => Escalate
    }
}

object Supervisor extends App {
  val actorSystem = ActorSystem("MyActorSystem")
  implicit val timeout = Timeout(5 seconds)

  def props( ) = Props(classOf[Supervisor])

  val actor = actorSystem.actorOf(Supervisor.props(), "parent")

  val future = actor ? "hi"

  val child = Await.result(future, 5 seconds).asInstanceOf[ActorRef]

  child ! 1
  child ! 2

  child ! "hi"
```

```
    child ! 3

    Thread.sleep(2000)

    actorSystem.terminate()
  }
```

There is an actor hierarchy at work here. The parent actor creates a child actor and puts `supervisorStrategy` in place. By default, `AKKA` stops the actor.

The child actor accepts only `Int` messages. For any other messages, it throws a `RunTimeException`. (Actually it throws a *BadMessageException* which extends *RuntimeException*). Due to the supervision strategy, the supervisor steps in and recreates the actor. The flow is shown in the following diagram:



As shown in the following output (trimmed down suitably), the first two messages are processed normally. Then, we send a string message instead of a number. This results in the exception. However, we sent one more message, and as seen in the following output, the message is processed correctly:

```
[INFO]... [akka://MyActorSystem/user/parent/child] 2
[INFO]... [akka://MyActorSystem/user/parent/child] 3
[ERROR]... [akka://MyActorSystem/user/parent] null
 com.concurrency.book.chapter08.Child$BadMessageException
 ....
[INFO]... [akka://MyActorSystem/user/parent/child] 4
```

One noteworthy thing is we use the `actorSystem.actorSelection(actorPath)` call to reach for the child actor reference. Check out `https://doc.akka.io/docs/akka/current/general/addressing.html` for more information on actor paths.

Actor communication – the ask pattern

Actors also need to communicate with each other as they need to collaborate. Just like people in a company need to talk to each other to work as a team.

A request-response pattern is very common in everyday programming. For example, an actor can *ask* another one for some service. The caller needs to wait for this response.

On the other hand, an actor should never block. If an actor blocks, it blocks the underlying thread, thereby starving the other actors.

The `akka.pattern` package provides the `ask` operator, named **?**. This operator sends a message to the target actor. The following code illustrates the use of ask from the main method of the `ActorTestAsk` object. The `ActorTestAsk` class is *not an actor:*

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorSystem, Props}
import akka.util.Timeout
import akka.pattern.ask
import scala.concurrent.duration._

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

class ActorTestAsk extends Actor with ActorLogging {
  override def receive: Receive = {
    case s: String => sender ! s.toUpperCase
    case i: Int => sender ! (i + 1)
  }
}

object ActorTestAsk extends App {
  def props() = Props(classOf[ActorTestAsk])
  val actorSystem = ActorSystem("MyActorSystem")

  val actor = actorSystem.actorOf(ActorTestAsk.props(), name = "actor1")
  implicit val timeout = Timeout(5 seconds)

  val future = actor ? "hello"
```

```
    future foreach println

    Thread.sleep(2000)
    actorSystem.terminate()
  }
```

We define an actor as the instance that receives either a `string` or an `int` message and returns an appropriate response. The main method uses the ask operator to send a `string` message to the actor. The result is of the `Future[Any]` type.

We use the `foreach` combinator to print the result of the future, whenever it becomes available. We also need to provide an implicit timeout and an execution context. You can refer to `Chapter 7`, *Functional Concurrency Patterns*, for a quick revision, in case you need these concepts.

When we run the program, the output is as follows:

```
HELLO
```

Next, we will look at how an actor talks to another actor, using the `ask` pattern.

Actors talking with each another

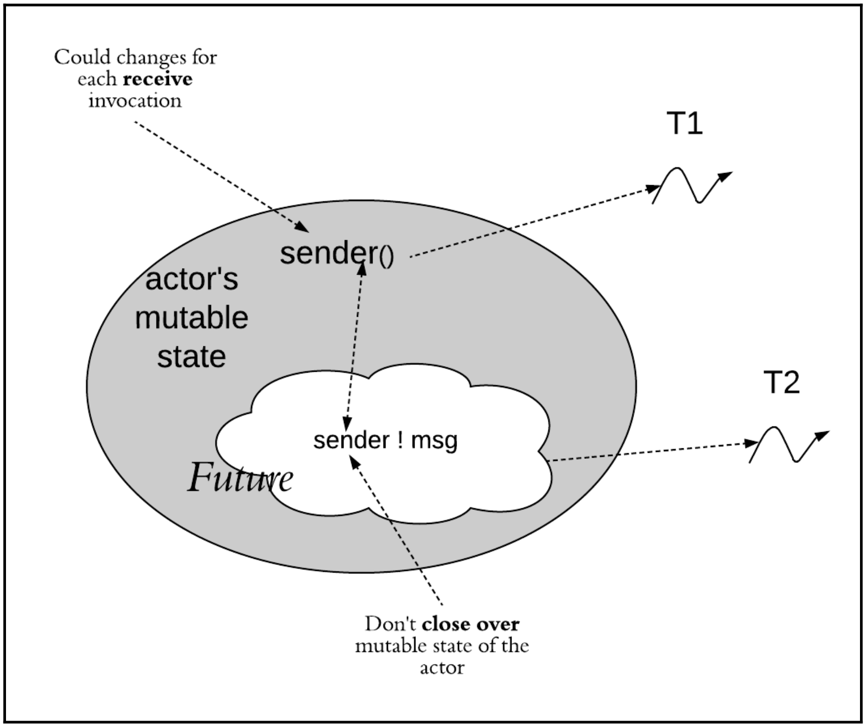
The earlier snippet showed how a nonactor code used the asked pattern. The upcoming snippet shows how an actor *asks* another for some service.

We have two actors. `Actor1` receives an `Actor2` reference in its constructor. When `Actor1` receives a string message, it delegates the message to `Actor2`.

The `Actor2` message processing simulates a delay of one second and then upcases the argument string and returns it back as shown in the following:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, Props}
import akka.pattern.ask
import akka.util.Timeout

import scala.concurrent.{Await, Future}
import scala.concurrent.duration._
import scala.util.{Failure, Success}
import scala.concurrent.ExecutionContext.Implicits.global

object Actor1 {
  def props(workActor: ActorRef) = Props(new Actor1(workActor))
}

class Actor1(workActor: ActorRef) extends Actor with ActorLogging {
  override def receive: Receive = {
    case s: String => {
      implicit val timeout = Timeout(20 seconds)

      val future = workActor ? s.toUpperCase
      future onComplete {
        case Success(s) => log.info(s"Got '${s}' back")
        case Failure(e) => log.info(s"Error'${e}'")
      }
    }
  }
}

class Actor2 extends Actor with ActorLogging {
  override def receive: Receive = {
    case s: String => {
      val senderRef = sender() //sender ref needed for closure
      Future {
        val r = new scala.util.Random
        val delay = r.nextInt(500)+10
        Thread.sleep(delay)
        s.toUpperCase
      } foreach { reply =>
        senderRef ! reply
      }
    }
  }
}

object ActorToActorAsk extends App {
  val actorSystem = ActorSystem("MyActorSystem")

  val workactor = actorSystem.actorOf(Props[Actor2], name = "workactor")

  val actor = actorSystem.actorOf(Actor1.props(workactor), name = s"actor")

  val actorNames = (0 to 50).map(x => s"actor${x}")
  val actors = actorNames.map(actorName =>
actorSystem.actorOf(Actor1.props(workactor), name = actorName))

  (actorNames zip actors) foreach { case (name, actor) => actor ! name }

}
```

To send a response back, `Actor2` uses the `sender()` method. The processing simulates a random delay between 10 to 500 milliseconds. After this delay, the argument string is upcased and sent back to the sender, which is `Actor1`. The important point to note is that the future task and actors run on different threads! The following figure should help highlight the problem:


Could changes for each **receive** invocation
T1
sender()
actor's mutable state
T2
sender ! msg
Future
Don't **close over** mutable state of the actor

In Scala, a variable *closes over* the surrounding environment. The following code shows the issue:

```
object ClosesOver extends App {
  class Foo {
    def m1( fun: ( Int ) => Unit, id: Int ) {
      fun(id)
    }
  }

  var x = 1

  def addUp( num: Int ) = println(num + x)

  val foo = new Foo
```

```
    foo.m1(addUp, 1)

    x = 2

    foo.m1(addUp, 1)
  }
```

The `addUp()` method is closing over the mutable variable `x`. In Scala, due to *Eta Expansion* (refer to `https://alvinalexander.com/scala/fp-book/how-to-use-scala-methods-like-functions` for more information) our method is converted in to a function, when passed as an argument to `Foo.m1`.

This function (method) is not pure. It refers to `x` from the surrounding scope (that is, closes over it) and when `x` is changed, the method behavior changes too.

Going back to the actor code, if the future task references the `sender()` method by the time it runs, another message from a different actor could be under processing! So, if the future uses the following line to send back the response, you have got a race to deal with:

```
sender() ! arg.toUpperCase // Don't do this
```

In case of a bug, the response would go to *a wrong sender!*

Instead, we save the sender into a `val` variable, `senderRef`, and use that inside the future!

```
val senderRef = sender() //sender ref needed for closure
```

To test the program, we create 51 sender actors (`Actor1`) all sending messages to the same work actor (`Actor2`). We generate 51 actor names, and each actor sends its name as a string message to the work actor.

The output logs the messages. It is easy to verify the message against the actor:

```
[INFO]... [akka://MyActorSystem/user/actor2] Got 'ACTOR2' back
[INFO]... [akka://MyActorSystem/user/actor1] Got 'ACTOR1' back
[INFO]... [akka://MyActorSystem/user/actor50] Got 'ACTOR50' back
[INFO]... [akka://MyActorSystem/user/actor4] Got 'ACTOR4' back
[INFO]... [akka://MyActorSystem/user/actor0] Got 'ACTOR0' back
[INFO]... [akka://MyActorSystem/user/actor6] Got 'ACTOR6' back
```

Try replacing `senderRef` with `sender()` and run the (buggy!) program again. See how the correlation goes for a toss.

Actor communication – the tell pattern

The preceding code seems brittle as we need to think of the correct value for the timeout. You need a very good guess for how long it will take for the overloaded work actor to return a response and adjust the timeout value based on it. This is not a full-proof solution.

Instead, we use the *tell* pattern. The resulting code is simpler, and more importantly, does not need any timeout, as we don't use the future.

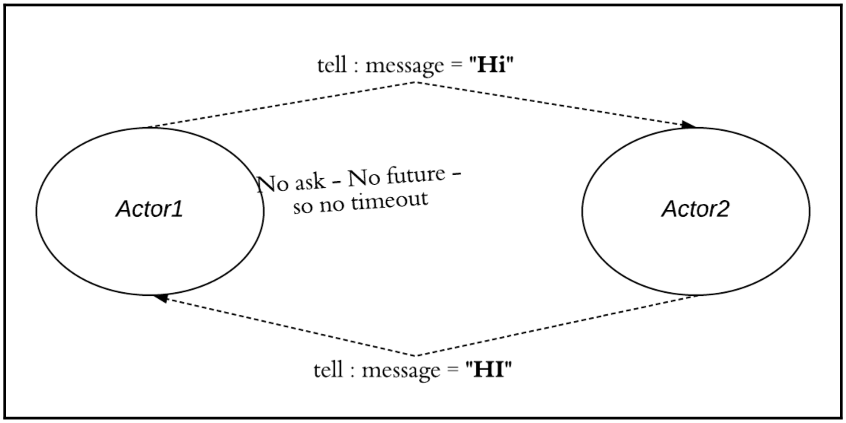
However, instead of sending back the response, we send a new type of message. The result of the future processing is wrapped into a `Result(String)` message, which is just a case class, defined in the companion object of `Actor1`. It is a recommended practice to define an actor's messages in its companion object.

Here is the code using the `tell` pattern:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, Props}

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object Actor1 {
  def props(workActor: ActorRef) = Props(new Actor1(workActor))
  case class Result(s: String)
}

class Actor1(workActor: ActorRef) extends Actor with ActorLogging {
  override def receive: Receive = {
    case s: String        => workActor ! s
    case Actor1.Result(s) => log.info(s"Got '${s}' back")
  }
}

class Actor2 extends Actor with ActorLogging {
  override def receive: Receive = {
    case s: String => {
      val senderRef = sender() //sender ref needed for closure
      val arg = s
      Future {
        Thread.sleep(1000)
```

```
          Actor1.Result(arg.toUpperCase)
        } foreach { reply =>
          senderRef ! reply
      }
    }
  }

  object ActorToActorTell extends App {
    val actorSystem = ActorSystem("MyActorSystem")

    val workactor = actorSystem.actorOf(Props[Actor2], name = "workactor")

    val actor = actorSystem.actorOf(Actor1.props(workactor), name = s"actor")

    val actorNames = (0 to 50).map(x => s"actor${x}")
    val actors = actorNames.map(actorName =>
  actorSystem.actorOf(Actor1.props(workactor), name = actorName))

    (actorNames zip actors) foreach { case (name, actor) => actor ! name }

    Thread.sleep(14000)

    actorSystem.terminate()

  }
```

The following diagram shows the flow:



The output is similar to the ask version.

The pipeTo pattern

You need the ask pattern at times. For example, the service actor may make a web service call or fire a database query. The result may be necessary for further processing.

This is shown in the following code:

```
package com.concurrency.book.chapter08

import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, Props}

import scala.concurrent.Future
import akka.pattern.{ask, pipe}
import scala.concurrent.ExecutionContext.Implicits.global

class PipeToActor extends Actor with ActorLogging{
 override def receive: Receive = process(List.empty)

 def process(list: List[Int]): Receive = {
 case x : Int => {
 val result = checkIt(x).map((x, _))
 pipe(result) to self
 }
 case (x: Int, b: Boolean) => log.info(s"${x} is ${b}")
 }

 def checkIt(x: Int): Future[Boolean] = Future {
 Thread.sleep(1000)
 x % 2 == 0
 }
}

object PipeToActor extends App {
 val actorSystem = ActorSystem("MyActorSystem")

 def props( ) = Props(classOf[PipeToActor])

 val actor = actorSystem.actorOf(PipeToActor.props(), "actor1")

 (1 to 5).foreach(x => actor ! x)

 Thread.sleep(5000)

 actorSystem.terminate()
}
```

Have a look at the following line:

```
pipe(result) to self
```

This takes the result of the future and converts it into another message to self! This message is processed by the following clause:

```
case (x: Int, b: Boolean) => log.info(s"${x} is ${b}")
```

Running the code will give you the following output:

```
[INFO]... [akka://MyActorSystem/user/actor1] 3 is false
[INFO]... [akka://MyActorSystem/user/actor1] 1 is false
[INFO]... [akka://MyActorSystem/user/actor1] 2 is true
[INFO]... [akka://MyActorSystem/user/actor1] 4 is true
[INFO]... [akka://MyActorSystem/user/actor1] 5 is false
```

This is an elegant solution, as we again don't need to specify any timeout. We are talking to a future-based API and converting the result into another message.

Summary

In this chapter, we introduced the actor paradigm. We used a real-world analogy, a software company and its employees, to correlate various terms used in the paradigm. There are reasons for using `ActorRef` as a proxy to refer to an actor. We saw how *let it crash* and *location transparency* are realized due to the `ActorRef` encapsulation, working as a proxy to the actual actor reference.

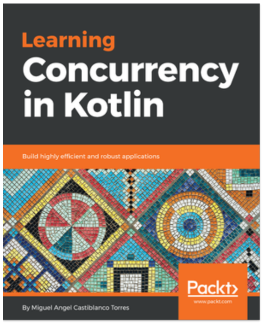
We covered the aspect of actors for *state encapsulation.* We also saw what actors are and how they map to threads. Next, we covered some essential and common actors patterns.

The *become* pattern is used for changing an actor's behavior. We saw how it helps in making the actor state immutable. Next, we saw how the supervision model works, so if an actor crashes, another copy is restarted.

Finally, we covered basic actor communication patterns such as `ask`, `tell`, and `pipeTo`. We also looked at the care needed when we use actors with future tasks.

Other Books You May Enjoy

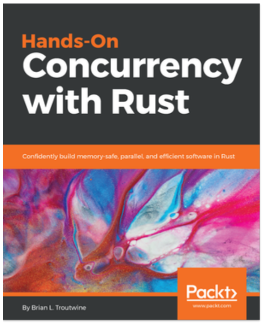
If you enjoyed this book, you may be interested in these other books by Packt:



Learning Concurrency in Kotlin
Miguel Angel Castiblanco Torres

ISBN: 978-1-78862-716-0

- Understand Kotlin's approach to concurrency
- Implement sequential and asynchronous suspending functions
- Create suspending data sources that are resumed on demand
- Explore the best practices for error handling
- Use channels to communicate between coroutines
- Uncover how coroutines work under the hood



Hands-On Concurrency with Rust
Brian L. Troutwine

ISBN: 978-1-78839-997-5

- Probe your programs for performance and accuracy issues
- Create your own threading and multi-processing environment in Rust
- Use coarse locks from Rust's Standard library
- Solve common synchronization problems or avoid synchronization using atomic programming
- Build lock-free/wait-free structures in Rust and understand their implementations in the crates ecosystem
- Leverage Rust's memory model and type system to build safety properties into your parallel programs
- Understand the new features of the Rust programming language to ease the writing of parallel programs

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index