Modern Java

Second Edition

Java 21 and the Java Ecosystem

Adam L Davis

Modern Java: Second Edition

Java 21 and the Java Ecosystem

Adam L. Davis

This book is available at https://leanpub.com/modernjavasecondedition

This version was published on 2025-08-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Adam L. Davis

What's New in Java 8

Dedicated to all those that work to protect human rights.

Contents

Part I: Java

Java

History

Java™ was first developed in the 90's by James Gosling. It borrows much of its syntax from C and C++ to be more appealing to existing programmers at the time. Java was owned by Sun Microsystems which was then acquired by Oracle in 2010.

Java is a *statically typed, object-oriented* language. Statically typed means every variable and parameter must have a defined type (as opposed to languages like Javascript which are dynamically typed). Object-oriented (OO), means that data and functions are grouped together into objects (functions are usually referred to as *methods* in OO languages).

Java code is compiled to byte-code which runs on a virtual machine (the Java Virtual Machine, JVM). The virtual machine handles garbage collection and allows Java to be compiled once, and run on any OS or hardware that has a JVM. This is an advantage over C/C++ which has to be compiled directly to machine code and has no automatic garbage collection (the programmer needs to allocate and deallocate memory).

The standard implementation of Java comes in two different packages, the JRE (Java Runtime Environment) and the JDK (Java Developement Kit). The JRE is strictly for running Java as an end user, while the JDK is for developing Java code. The JDK comes with the "javac" command for compiling Java code to byte-code, among other things.

At the time of writing, Java is one of the most popular programming languages in use[1], particularly for server-side web applications.

Openness

In an attempt to make Java more open and community based, Sun Microsystems started the Java Community Process (JCP), which allows a somewhat democratic evolution of Java and JVM specifications. Also, Sun relicensed most of its Java technologies under the GNU General Public License in May 2007 which has resulted in multiple open-source implementations of the JVM (OpenJDK is the official one). Although Sun has many patents on some aspects of the JVM, historically it has not used these patents to sue other companies, which has allowed a healthy ecosystem of competing JVM's to emerge. Although Oracle sued Google over its use of Java in Android, Oracle eventually lost the case in May 2012[2].

Generally when we refer to the JVM, we are referring Oracle's JVM, but OpenJDK or any other JVM can be used.'

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[2] http://news.cnet.com/8301-1023_3-57440235-93/

The Java Ecosystem

The Java Ecosystem is huge. It is mainly composed of JVM's, libraries, tools, and IDE's. It is so huge, there's no way to really summarize it in one book, but we will cover some of the highlights.

The three most popular IDE's are (in no particular order):

- Eclipse[3] - Open-source project by the Eclipse Foundation.
- NetBeans[4] - Sun's (now Oracle's) open-source Java IDE.
- IntelliJ IDEA[5] - A commercial IDE with a community edition.

We will discuss some of the more promising new libraries and tools in the Java ecosystem, such as the following:

- Maven, gradle and other build tools.
- Libraries for concurrent programming.
- JUnit, spock and other test frameworks.
- Groovy, Scala, and other JVM languages.
- Grails, Play, and other web-frameworks.
- JVM Cloud providers.

[3] http://eclipse.org/
[4] http://netbeans.org/
[5] https://www.jetbrains.com/idea/

Java Syntax and Conventions

This chapter covers some of the basic Java syntax and conventions as well as updates in Java 5 and 6 (1.5 and 1.6).

Java JDK

The core class types included in Java is called the JDK (Java Development Kit). It includes all the basic tools you would need in a modern application, everything from collection types and queues to web sockets to files and image processing.

As an object-oriented language, methods and properties are organized into classes which are organized in packages. Each Java class should be defined in one file named for that class. The classes under `java.lang` such as `String` are always available, otherwise you need to explicitly import the classes using an `import` statement at the top of your Java file. An instance of a class is called an *object*. All objects are passed by reference. Unlike in C you cannot modify a reference pointer.

A particular aspect of Java is that there are special types called "primitives". Unlike objects, primitives do not have methods and always have a value (they can never be `null`).

Primitives and Arrays

Primitive types in Java refer to different ways to store numbers (and have historical but also practical significance):

- char: A single character, such as 'A' (the letter A).
- byte: A number from -128 to 127 (8 bits[6]). Typically a way to store or transmit raw data.
- short: A 16-bit signed integer. It has a maximum of around 32 thousand.
- int: A 32-bit signed integer. Its maximum is around 2 to the 31st power.
- long: A 64-bit signed integer. Maximum of 2 to the 63rd power.
- float: A 32-bit floating point number. This is a non-precise value that is used for things like simulations.
- double: Like *float* but with 64-bits.
- boolean: Has only two possible values: `true` and `false` (much like 1 bit).

[6] A bit is the smallest possible amount of information. It corresponds to a 1 or 0.

See Java Tutorial - Data Types[7] for more information.

In Java you can define arrays of primitives or classes. For example, `String[] strArray = {"a", "b", "c"};` creates an array of three Strings. Once you define an array, you cannot directly change its length. If you need a list of expanding size, use `java.util.ArrayList`.

All other types other than primitives and arrays are considered objects.

Classes

To define a new class, create a new file named `Classname.java`. For example, let's create a Dragon class in a file named `Dragon.java`:

```
1 import java.util.*;
2 public class Dragon {
3 }
```

In this case the class does not have a package. If it did we would declare it in the first line and the file must be in a directory structure matching the package.

The first line above imports everything in the `java.util` package. This includes `List`, `ArrayList`, `Map`, and `HashMap` for example.

Properties and Methods

Next you might want to add some properties and methods to your class. A *property* is a value associated with a particular object. A *method* is a block of code on a class.

```
1 package com.example.mpme;
2 public class SmallClass {
3     String name;
4     String getName() {return name;}
5     void print() {System.out.println(name);}
6 }
```

In the above code `name` is a String property and `getName` and `print` are methods. The method `getName` returns a String (the `name`) and `print` uses the built in `System` class to print out the name to the standard output stream.

[7] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Comments

As a human, it is sometimes useful for you to leave notes in your source code for other humans and even for yourself later. We call these notes *comments*. You write comments thusly:

```
1     String gold = "Au"; // this is a comment
2     String a = gold; // a is now "Au"
3     String b = a; // b is now "Au"
4     b = "Br";
5     /* b is now "Br".
6        this is still a comment */
```

Those last two lines demonstrate multiline comments. So in summary:

- Two forward slashes denote the start of a single-line comment.
- Slash-asterisk marks the beginning of a multiple-line comment.
- Asterisk-slash marks the end of a multiple-line comment.

Java 5

Java 5 added several new features to the language. If you're not familiar with Java 5 or would like a refresher, keep reading. We're going to assume you understand these concepts in the remainder of the book.

Java 5 added the following features:

- Generics
- Annotations
- More concise `for` loops
- Static imports
- Autoboxing/unboxing
- Enumerations
- Varargs
- Concurrency utilities in package `java.util.concurrent`

Generics

Generics were a huge addition to the language. They improved the type-safety of Java, but also added a lot of complexity to the language.

Generics are used most commonly to specify what type a Collection holds. This reduces the need for casting and improves type-safety. For example, declaring a `List` of Strings is the following:

```
List<String> strings = new ArrayList<String>();
```

Declaring a Map of `Long` to `String` would appear as the following:

```
Map<Long,String> map = new HashMap<Long,String>();
```

The need to repeat the generic type twice in the declaration is one of Java's harshest criticisms. However various libraries, such as Google's guava, make this less painful by using static methods. For example declaring the above map would be as simple as the following:

```
Map<Long,String> map = Maps.newHashMap();
```

Also, Java 7 will ameliorate this situation with the *diamond operator*, which we will discuss later.

Annotations

Java annotations allow you to add meta-information to Java code that can be used by the compiler, various API's, or even your own code at runtime.

The most common annotation you will see is the `@Override` annotation which declares to the compiler that you are overriding a method. This is useful because it will cause a compile-time error if you mistype the method name for example.

Other useful annotations are those in `javax.annotation` such as `@Nonnull` and `@Nonnegative` which declare your intentions.

Annotations such as `@Autowired` and `@Inject` are used by direct-injection frameworks like Spring and Google Guice[8], respectively, to reduce "wiring" code.

More concise `for` loops

You can write for loops in a concise way for an array or any class that implements `Iterable`. For example:

```
String[] strArray = {"a", "b", "c"};
for (String str : strArray)
        out.println(str);
```

"Wait, don't you need a System there?" you're probably thinking. Not necessarily in Java 5 with the *static import* feature.

Static import

In Java 5 you can use the words `import static` to import a static member of another class. This can help your code be more concise as shown in the above section. To do this, you would need the following at the top of the class file:

[8] http://code.google.com/p/google-guice/

```
import static java.lang.System.out;
```

However, the creators of Java recommend you use static import very sparingly[9], so don't get carried away.

Autoboxing, Enums, Varargs

Autoboxing
: The Java compiler will automatically wrap a primitive type in the corresponding object when it's necessary. For example, when assigning a variable or passing in parameters to a function, as in the following: `printSpaced(1, 2, 3)`

Unboxing
: This is simply the reverse of Autoboxing. The Java compiler will unwrap an object to the corresponding primitive type when possible. For example, the following code would work: `double d = new Double(1.1) + new Double(2.2)`

Enums
: The `enum` keyword creates a typesafe, ordered list of values. For example, `enum Letter { A, B, C; }`

Varargs
: You can declare a method's last parameter with an elipse (...) and it will be interpreted to accept any number of parameters (including zero) and convert them into an array in your method. For example, see the following code:

  void printSpaced(Object... objects) { for (Object o : objects) out.print(o + " "); }

Putting it all together, you have the following code (with output in comments):

```
printSpaced(Letter.A, Letter.B, Letter.C); // A B C
printSpaced(1, 2, 3); // 1 2 3
```

Java 6

Java 6 did not have as many big changes as Java 5, but it did add the following:

- **Web Services** - First-class support for writing XML web services.
- **Scripting** - the ability to plug-in scripting engines (for Javascript, Ruby, and Groovy for example).
- **Java DB** (Apache Derby) is co-bundled in the JRE.
- **JDBC 4.0** adds many feature additions like special support for XML as an SQL datatype and better integration of Binary Large OBjects (BLOBs) and Character Large OBjects (CLOBs).

[9] http://docs.oracle.com/javase/1.5.0/docs/guide/language/static-import.html

- **More Desktop APIs** - SwingWorker, JTable, and more.
- **Monitoring and Management** - Jhat for forensic explorations of core dumps.
- **Compiler Access** - The compiler API opens up programmatic access to javac for in-process compilation of dynamically generated Java code.
- **Override interface methods** - The @Override annotation can be used to declare you're overriding an interface method.

Java 7

Java 7 has some performance benefits and new features[10] that many programmers have been expecting for years.

Language Updates

The following features have been added to Java the language:

- Diamond Operator
- Strings in switch
- Automatic resource management
- Improved Exception handling
- Numbers with underscores

Diamond Operator

The *Diamond Operator* simplifies the declaration of generic classes. The generic types are inferred from the definition of the field or variable. For example, in the following code, the second line is now equivalent to the first in Java 7:

```
1  Map<String, List<Double>> nums = new HashMap<String, List<Double>> ();
2  Map<String, List<Double>> nums = new HashMap <> ();
```

Strings in Switch

You can now use *Strings in switch* statements. For example, the following code would compile and work in Java 7:

[10] http://openjdk.java.net/projects/jdk7/features/

```
public static <T> Collection<T> makeNew(String type, Class<T> tClass) {
        switch (type) {
        case "set":
                return new HashSet<>();
        case "lset":
                return new LinkedHashSet<>();
        case "treeset":
                return new TreeSet<>();
        case "vector":
                return new Vector<>();
        case "array":
                return new ArrayList<>();
        case "deque":
        case "queue":
        case "list":
        default:
                return new LinkedList<>();
        }
}
```

As seen above, Strings can now be used just like any primitive would in a switch statement.

Github Repo

You can find the source-code for these examples on github modern-java-examples[11].

Automatic resource management

The new *Automatic resource management* feature makes dealing with resources, such as files, much easier. Before Java 7 you needed to explicitly close all open streams, causing some very verbose code. Now you can just do the following:

[11] https://github.com/adamd/modern-java-examples

```
public void writeWithTry() {
        try (FileOutputStream fos = new FileOutputStream("books.txt");
                        DataOutputStream dos = new DataOutputStream(fos)) {
                dos.writeUTF("Modern Java");
        } catch (IOException e) {
                // log the exception
        }
}
```

Improved Exception handling

Improved Exception handling in Java 7 means that you can catch more than one exception in one catch statement. Previously, you had to write a different catch block for each exception. This may seem trivial, but will make Java development somewhat easier. Here's an example of the new style:

```
public static Integer fetchURLAsInteger(String urlString) {
        try {

                URL url = new URL(urlString);
                String str = url.openConnection().getContent().toString();
                return Integer.parseInt(str);

        } catch (NullPointerException | NumberFormatException | IOException e) {
                return null;
        }
}
```

The above code would fetch content from the given url and attempt to convert it to an Integer. If anything goes wrong it returns null. Although this is a contrived example, similar situations do occur in real code.

Numbers with underscores

Numbers with underscores is exactly what you think. Humans have a hard time reading long streams of numbers, so Java 7 allows you to put underscores in numeric literals to make them easier to understand. For example, three million would be written as follows:

```
int lemmings = 3_000_000;
```

Fork/Join

There are new Java concurrency APIs (JSR 166y) referred to as the **Fork-join framework**. It is designed for tasks that can be broken down and takes advantage of multiple processors. The core classes are the following (all located in `java.util.concurrent`):

- `ForkJoinPool`: An ExecutorService for running ForkJoinTasks and managing and monitoring the tasks.
- `ForkJoinTask`: This represents the abstract task that runs within the ForkJoinPool.
- `RecursiveTask`: This is a subclass of ForkJoinTask whose compute method returns some value.
- `RecursiveAction`: This is a subclass of ForkJoinTask whose compute method does not return any value.

As an example of using this framework, let's find the sum of 2000 integers. This is a trivial example but will hopefully demonstrate proper use of the ForkJoin framework.

In this example we will divide the array of integers in half and assign each half to a `RecursiveTask`. If the array size is less than 20 elements then we assign it to another RecursiveTask that computes the sum of the array.

Here is the RecursiveTask for computing the sum:

```
class SumCalculatorTask extends RecursiveTask<Integer>{
        int [] numbers;
        SumCalculatorTask(int[] numbers){
                this.numbers = numbers;
        }

        @Override
        protected Integer compute() {
                int sum = 0;
                for (int i : numbers){
                        sum += i;
                }
                return sum;
        }
}
```

The compute method has to be overridden with the actual task to be performed. In the above case its iterate through the elements of the array and return the computed sum.

We create a RecursiveTask for dividing the array into two parts and assign each part to another RecursiveTask for further dividing. We continue dividing the array and stop dividing when the array has less than 20 elements.

```
class NumberDividerTask extends RecursiveTask<Integer>{
        int [] numbers;
        NumberDividerTask(int [] numbers){
                this.numbers = numbers;
        }

        @Override
        protected Integer compute() {
                int sum = 0;
                List<RecursiveTask<Integer>> forks = new ArrayList<>();
                if (numbers.length > 20){
                        NumberDividerTask task1 =
                                new NumberDividerTask(Arrays
                                  .copyOfRange(numbers, 0, numbers.length/2));
                        NumberDividerTask task2 =
                                new NumberDividerTask(Arrays
                                  .copyOfRange(numbers, numbers.length/2, numbers.length));
                        forks.add(task1);
                        forks.add(task2);
                        task1.fork();
                        task2.fork();
                } else {
                        SumCalculatorTask sumCalcTask = new SumCalculatorTask(numbers);
                        forks.add(sumCalcTask);
                        sumCalcTask.fork();
                }
                //Combine the result from all the tasks
                for (RecursiveTask<Integer> task : forks) {
                        sum += task.join();
                }
                return sum;
        }
}
```

The above `NumberDividerTask` spawns either two other `NumberDividerTask`'s or a `SumCalculatorTask`. Each task keeps a track of the sub-tasks it has created. At the end of the task we wait for all the tasks in the forks list to finish by invoking the `join()` method and compute the sum of those values returned from the sub-tasks.

To invoke the above defined tasks we make use of ForkJoinPool and create a NumberDividerTask task by giving it the array whose sum we wish to compute.

```
public class ForkJoinTest {
        static ForkJoinPool forkJoinPool = new ForkJoinPool();
        public static final int LENGTH = 2000;

        public static void main(String[] args) {
                int [] numbers = new int[LENGTH];
                // Create  an array with some values.
                for(int i=0; i<LENGTH; i++){
                        numbers[i] = i * 2;
                }
                int sum = forkJoinPool.invoke(new NumberDividerTask(numbers));

                System.out.println("Sum: "+sum);
        }
}
```

After running the above code the output should be: `Sum: 3998000`.

Although this is a simple example, the same concept could be applied to any "divide and conquer" algorithm.

New IO (nio)

Java 7 adds several new classes and interfaces for manipulating files and file-systems. This new API allows developers to access many low-level OS operations that were not available from the Java API before, such as the `WatchService` and the ability to create links (in *nix operating systems).

The following list defines some of the most important classes and interfaces of the NIO API:

Files
: This class consists exclusively of static methods that operate on files, directories, or other types of files.

FileStore
: Storage for files.

FileSystem
: Provides an interface to a file system and is the factory for objects to access files and other objects in the file system.

FileSystems
: Factory methods for file systems.

LinkPermission
: The Permission class for link creation operations.

Paths
: This class consists exclusively of static methods that return a Path by converting a path string or URI.

FileVisitor<T>
: An interface for visiting files.

WatchService
: An interface for watching varies file-system events such as create, delete, modify.

Using a WatchService

To watch a directory you would register a Path object with the WatchService, as follows:

```
// import the standard events: ENTRY_MODIFY, ENTRY_DELETE, etc.
import static java.nio.file.StandardWatchEventKinds.*;
import java.nio.file.*;
// later on in some method...
Path path = Paths.get("/usr/local");
WatchService watchService = FileSystems.getDefault().newWatchService();
WatchKey watchKey = path.register(watchService, ENTRY_CREATE);
```

This registers the WatchService to watch the given path. For example, if you register the directory "/usr/local" as above, the WatchService will be notified whenever a file is created in that directory.

To monitor events you can use either the `take` or `poll` method of the WatchService. The first is a blocking call, and second, `poll`, is non-blocking and returns null if no events are avaiable. Keep in mind that `take` will block the Thread until something happens. Both methods return a `WatchKey` instance that needs to be reset by calling `reset` before continuing.

For example, the following code loops forever calling `take` and doing something with each `WatchEvent` that is returned:

```
while (true) {
        WatchKey key = watchService.take();
        List<WatchEvent<?>> pollEvents = key.pollEvents();
        try {
                for (WatchEvent<?> event : pollEvents) {
                        Path p = (Path) event.context();
                        // do something with p
                }
        } catch (Exception e) { e.printStackTrace();
        } finally { key.reset(); }
}
```

JVM Benefits

Java 7 adds some new features to the JVM, the language, and the runtime libraries.

The JVM has the following new features:

- Serviceability features (JRockit/hotspot convergence)
  - Java Mission Control (monitor, manage, profile)
  - Java Flight Recorder (profiling, problem analysis, debugging) (in progress)
- jdk introspection
  - `jcmd` - list running java processes
  - `jcmd <pid> GC.class_histogram` - size of classes
- Better garbage collection.

Performance Benefits

There are also Performance Benefits in the JVM and runtime libraries:

- Runtime compiler improvements.
- Sockets Direct Protocol (SDP)
- Java Class Libraries
  - Avoid contention in Date: changed from HashTable to ConcurrentHashMap
  - BigDecimal improvements (CR 7013110)
- Crypto config. files updates, CR 7036252
  - User land crypto for SPARC T4
  - Adler32 & CRC32 on T-series
- String(byte[], string) and String.getBytes(String) 2-3x performance.
- HotSpot JVM
  - updated native compilers `-XX:+UseNUMA` on Java 7 (Linux kernel 2.6.19 or later; glibc 2.6.1).
  - Partial PermGen removal (full removal in JDK 8) -interned String moved to Java heap.
  - Default Hashtable table size is 1009 increase size if needed `-XX:StringTableSize=n`
  - Distinct class names: `XX:+UnlockExperimentalVMOptions -XX:PredictedClassLoadCount=#`
- Client library updates (Nimbus Look&Feel; JLayer; translucent windows, Optimized 2d rendering)
- JDBC 4.1 updates (allow Connection, ResultSet, and Statement objects be used in try-with-resources statement)
- JAXP 1.4.5 (Parsing) (bug fixes, conformance, security, performance)
- JAXB 2.2.3 (Binding)
- Asynchronous I/O in `java.io` for both sockes and files (uses native platform when available)
- x86 (intel) improved 14x over 5 processor releases (jdk5 jdk 6)
- JDK 7u4 faster than Java6 and JRockit.

Backwards Compatibility

There are some issues to watch out for when upgrading to Java 7 on a large project:

- More stringent bytecode verifier for Java 7 (only issue when doing bytecode modification; work-around `-XX:-UseSplitVerifier`)
- Order of methods return from `getMethods()` has changed (not guaranteed to be in declaration order)

Idiomatic Java 8: Lambdas, Streams, and Dates

As of publication, Java 8 is the de-facto standard version of Java. It is already in use in many production systems, so if you are currently using an older version of Java, it's time to upgrade.

Java 8 includes the following:

- Lambda expressions
- Method references
- Default Methods (Defender methods)
- A new Stream API.
- Optional
- A new Date/Time API.
- Nashorn, the new JavaScript engine
- Removal of the Permanent Generation
- and more…

The best way to read this book is with a Java 8 supporting IDE running so you can try it out.

Code examples can be found on github[12].

[12] https://github.com/adamd/hellojava8

Guava

No discussion of modern Java development would be complete without mentioning Guava.

Google started releasing some internal Java code as open-source under the name Google Collections back in 2007[13]. Its creation and architecture were partly motivated by generics inroduced in JDK 1.5. This became much more than only collection support and was rebranded as *guava*. Guava contains a lot of extremely useful code and gives some hints into modern Java practice.

Collections

It adds a bunch of very useful Collection-related classes and interfaces:

- `Collections2` - Utility methods for filtering, tranforming, and getting all possible permutations of Collections.
- `BiMap` - A Map that goes both ways (one-to-one mapping where values can map back to keys).
- `Multimap` - A Map that can associate keys with an arbitrary number of values. Use instead of `Map<Foo, Collection<Bar>>`.
- `Multiset` - A set that also keeps tracks of the number of occurances of each element.
- `Table` - Uses a row and column as keys to values.

For every Collection type, it also has a static utility class with useful methods, for example:

- Lists: `newArrayList, asList, partition, reverse, transform`
- Sets: `newHashSet, filter, difference, union`
- Maps: `newHashMap, newTreeMap, filterKeys, filterValues, asMap`

Objects

Guava's `Objects` class contains a bunch of useful methods for dealing with a lot of the boilerplate code in generic Java, such as writing equals and hashCode methods.

- `Objects.equal(Object, Object)` - null safe equals.
- `Objects.hashCode(Object...)` - an easy way to get a hash code based on multiple fields of your class.
- `Objects.firstNonNull(Object,Object)` - one way to deal with null-return values (returns the first non-null value).

[13] http://publicobject.com/2007/09/series-recap-coding-in-small-with.html

Concurrency

It also contains some concurrency support, such as the following:

ListenableFuture
: A ListenableFuture allows you to register callbacks to be executed once the computation is complete, or if the computation is already complete, immediately. This simple addition makes it possible to efficiently support many operations that the basic Future interface cannot support.

```
ListeningExecutorService srv = MoreExecutors
        .listeningDecorator(Executors.newFixedThreadPool(10));
ListenableFuture<Rocket> rocket = srv.submit(new Callable<Rocket>(){
  public Rocket call() {
    return launchIntoSpace();
  }
});
Futures.addCallback(rocket, new FutureCallback<Rocket>() {
  // we want this handler to run immediately after we launch!
  public void onSuccess(Rocket rocket) {
    navigateToMoon(rocket);
  }
  public void onFailure(Throwable thrown) {
    launchEscapePod();
  }
});
```

Functional Programming

Guava contains a lot of Functional programming paradigms. It has interfaces (Function, Predicate) and utility classes (Functions, Predicates) for dealing with functional programming in Java. However, the Guava team warns against overuse of these classes in the following from the guava wiki[14].

> Excessive use of Guava's functional programming idioms can lead to verbose, confusing, unreadable, and inefficient code. These are by far the most easily (and most commonly) abused parts of Guava, and when you go to preposterous lengths to make your code "a one-liner," the Guava team weeps.

[14] https://code.google.com/p/guava-libraries/wiki/FunctionalExplained#Functions_and_Predicates

Optional

Guava also has Optional[15] for avoiding null return values (which is similar to Nat Pryce's Maybe[16] class and Scala's `Option` class we will discuss later).

You can use `Optional.of(x)` to wrap a non-null value, `Optional.absent()` to represent a missing value, or `Optional.fromNullable(x)` to create an Optional from a reference that may or may not be null.

After creating an instance of Optional, you then use `isPresent()` to determine if the there is a value. Optional provides a few other helpful methods for dealing with missing values:

- `or(T)` - Returns the given default value if the Optional is empty.
- `or(Supplier<T>)` - Calls on the given Supplier to provide a value if the Optional is empty.
- `or(Optional<? extends T>)` - Useful for method-chaining; returns the given Optional if the Optional is empty.
- `orNull()` - simply unwraps the value (not recommended).
- `asSet()` - Returns a set of one element if there is a value, otherwise an empty set.

Other Useful Classes

Guava also contains tons of helpful utilities for general software development, such as the following:

EventBus
: EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other).

CacheBuilder
: Builds caches that can load and evict values. Caches are tremendously useful in a wide variety of use cases. For example, you should consider using caches when a value is expensive to compute or retrieve, and you will need its value on a certain input more than once.

BloomFilter
: Bloom filters are a probabilistic data structure, allowing you to test if an object is definitely not in the filter, or was probably added to the Bloom filter.

ComparisonChain
: A small, easily overlooked class that's useful when you want to write a comparison method that compares multiple values in succession and should return when the first difference is found. It removes all the tedium of that, making it just a few lines of chained method calls.

CharMatchers
: A really fast way to match characters, such as whitespace and digits.

[15] http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/base/Optional.html
[16] http://www.natpryce.com/articles/000776.html

Throwables

Lets you do some nice things with throwables, such as `Throwables.propagate` which rethrows a throwable if it's a RuntimeException or an Error and wraps it in a RuntimeException and throws that otherwise.

Guava has great documentation available on the google-code wiki[17].

[17] https://code.google.com/p/guava-libraries/wiki/GuavaExplained