Modern Java

Java 7 and Polyglot Programming on the JVM

Adam L. Davis

Modern Java

Java 7 and Polyglot Programming on the JVM

Adam L. Davis

This book is for sale at http://leanpub.com/modernjava

This version was published on 2014-01-31

Tweet This Book!

Please help Adam L. Davis by spreading the word about this book on Twitter!

The suggested hashtag for this book is #modernjava.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#modernjava

Dedicated to my wife and son.

Contents

Introduction

Always be learning

It's an exciting time to be a programmer, especially in the JVM space. Java 7 is widely used and Java 8 is closer to completion while other JVM languages, like Groovy and Scala, have been increasing in popularity. With the accelerating pace of change in technology, it is important as a programmer to always be learning and looking forward to the latest and greatest technology.

Who is this book for?

This book is meant for Java programmers or aspiring programmers who want to know about the latest Java and JVM technology.

What is this book about?

This book should help you understand the latest in Java 7; concurrent programming; build, testing, and web frameworks; and the best JVM languages so you can advance your career in software development.

What this book is not about

This book will not cover the basics of HTML, CSS, SQL, or any other non-JVM languages and will assume you know the basics of programming.

Tech Predictions

It's important to think ahead in this ever changing world, especially as a programmer. Making predictions of technology in the near future is somewhat easy, but it gets more difficult when you predict five years ahead or more. Here are some predictions relevant to programmers (which are basically just observations of current trends circa 2014):

Multi-core Processors are Commonplace

Dual-cores are already showing up in our smart-phones, and due to the simple progression of an updated "Moore's law", we should have 16-cores in smart-phones and tablets by 2020. This will make concurrent programming more mainstream as many of us already know. So learning about modern concurrent programming approaches (agents, immutability, etc.) is important.

Multi-touch Becomes Standard

It seems almost self-evident that multi-touch will become the standard interaction model. Tablets and smart-phones are soaring in popularity, and multi-touch is even moving into the dwindling desktop and laptop markets. It makes sense then to stop assuming mouse and/or keyboard interaction if you deal with any sort of user-interface design. Java on Android is an obvious winner, but so are HTML5-based solutions like Sencha Touch[1].

Linux has Arrived

With Ubuntu, Android, and ChromeOS, Linux is already forging ahead into mainstream technology. Does this effect you? Probably not if you're a web-developer, but if you develop apps for Windows or OSX, you will need to diversify. Linux (in its many different forms) might just (finally) disrupt the OS market.

Smart Phones will be Old Hat

To some people smart-phones are already old news; the big new thing is Google Glass[2] or smart watches like Pebble[3]. There will probably be several competitors to Google Glass and Pebble, so the "glasses" and/or smart-watch market might look much like the smart-phone market of today.

[1]http://www.sencha.com/products/touch

[2]http://en.wikipedia.org/wiki/Project_Glass

[3]https://getpebble.com/

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[4], particularly for server-side web applications.

Open-ness

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[5].

[4]http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[5]http://news.cnet.com/8301-1023_3-57440235-93/

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

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[6] - Open-source project by the Eclipse Foundation.
- NetBeans[7] - Sun's (now Oracle's) open-source Java IDE.
- IntelliJ IDEA[8] - 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.

[6]http://eclipse.org/
[7]http://netbeans.org/
[8]https://www.jetbrains.com/idea/

Java 5 & 6

Java 1.5

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:

```java
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[9], 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:

```java
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:

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

However, the creators of Java recommend you use static import very sparingly[10], so don't get carried away.

[9]http://code.google.com/p/google-guice/
[10]http://docs.oracle.com/javase/1.5.0/docs/guide/language/static-import.html

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
```

Concurrency utilities

These will be discussed in later chapters (`ExecutorService` and `Future`).

Java 1.6

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).

- **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

As Oracle will no longer provide free updates to Java 6 (as of Feb. 2013), Java 7 is now the de-facto standard version of Java. It is already in use in many production systems, so if you are currently using Java 6, it's time to upgrade. Java 7 has some performance benefits and new features[11] 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:

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

[11] 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[12].

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:

[12]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:

```java
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.

```java
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
An interface for visiting files.

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

A quick word on Watching

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

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

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)