COMPLETE

JAVA TUTORIAL

ADVANCED

Contents

1 How to create and destroy objects **1**

1.1 Introduction . . . 1

1.2 Instance Construction . . . 1

1.2.1 Implicit (Generated) Constructor . . . 1

1.2.2 Constructors without Arguments . . . 1

1.2.3 Constructors with Arguments . . . 2

1.2.4 Initialization Blocks . . . 2

1.2.5 Construction guarantee . . . 3

1.2.6 Visibility . . . 4

1.2.7 Garbage collection . . . 4

1.2.8 Finalizers . . . 5

1.3 Static initialization . . . 5

1.4 Construction Patterns . . . 5

1.4.1 Singleton . . . 6

1.4.2 Utility/Helper Class . . . 7

1.4.3 Factory . . . 7

1.4.4 Dependency Injection . . . 8

1.5 Download the Source Code . . . 9

1.6 What's next . . . 9

2 Using methods common to all objects **10**

2.1 Introduction . . . 10

2.2 Methods equals and hashCode . . . 11

2.3 Method toString . . . 13

2.4 Method clone . . . 14

2.5 Method equals and == operator . . . 15

2.6 Useful helper classes . . . 15

2.7 Download the Source Code . . . 16

2.8 What's next . . . 16

3 How to design Classes and Interfaces **17**

3.1 Introduction . . . 17
3.2 Interfaces . . . 17
3.3 Marker Interfaces . . . 18
3.4 Functional interfaces, default and static methods . . . 19
3.5 Abstract classes . . . 20
3.6 Immutable classes . . . 20
3.7 Anonymous classes . . . 21
3.8 Visibility . . . 22
3.9 Inheritance . . . 22
3.10 Multiple inheritance . . . 24
3.11 Inheritance and composition . . . 25
3.12 Encapsulation . . . 26
3.13 Final classes and methods . . . 27
3.14 Download the Source Code . . . 27
3.15 What's next . . . 27

4 How and when to use Generics **28**

4.1 Introduction . . . 28
4.2 Generics and interfaces . . . 28
4.3 Generics and classes . . . 29
4.4 Generics and methods . . . 29
4.5 Limitation of generics . . . 30
4.6 Generics, wildcards and bounded types . . . 31
4.7 Generics and type inference . . . 32
4.8 Generics and annotations . . . 33
4.9 Accessing generic type parameters . . . 33
4.10 When to use generics . . . 34
4.11 Download the Source Code . . . 35
4.12 What's next . . . 35

5 How and when to use Enums and Annotations **36**

5.1 Introduction . . . 36
5.2 Enums as special classes . . . 36
5.3 Enums and instance fields . . . 37
5.4 Enums and interfaces . . . 38
5.5 Enums and generics . . . 39
5.6 Convenient Enums methods . . . 39
5.7 Specialized Collections: EnumSet and EnumMap . . . 40

5.8 When to use enums . . . 41
5.9 Annotations as special interfaces . . . 41
5.10 Annotations and retention policy . . . 42
5.11 Annotations and element types . . . 42
5.12 Annotations and inheritance . . . 43
5.13 Repeatable annotations . . . 44
5.14 Annotation processors . . . 44
5.15 Annotations and configuration over convention . . . 44
5.16 When to use annotations . . . 45
5.17 Download the Source Code . . . 46
5.18 What's next . . . 46

6 How to write methods efficiently **47**
6.1 Introduction . . . 47
6.2 Method signatures . . . 47
6.3 Method body . . . 48
6.4 Method overloading . . . 48
6.5 Method overriding . . . 49
6.6 Inlining . . . 50
6.7 Recursion . . . 50
6.8 Method References . . . 50
6.9 Immutability . . . 51
6.10 Method Documentation . . . 51
6.11 Method Parameters and Return Values . . . 53
6.12 Methods as API entry points . . . 53
6.13 Download the Source Code . . . 54
6.14 What's next . . . 54

7 General programming guidelines **55**
7.1 Introduction . . . 55
7.2 Variable scopes . . . 55
7.3 Class fields and local variables . . . 55
7.4 Method arguments and local variables . . . 56
7.5 Boxing and unboxing . . . 57
7.6 Interfaces . . . 57
7.7 Strings . . . 58
7.8 Naming conventions . . . 59
7.9 Standard Libraries . . . 60
7.10 Immutability . . . 60
7.11 Testing . . . 60
7.12 Download the Source Code . . . 61
7.13 What's next . . . 61

8 How and when to use Exceptions **62**

8.1 Introduction . . . 62
8.2 Exceptions and when to use them . . . 62
8.3 Checked and unchecked exceptions . . . 62
8.4 Using try-with-resources . . . 63
8.5 Exceptions and lambdas . . . 64
8.6 Standard Java exceptions . . . 65
8.7 Defining your own exceptions . . . 65
8.8 Documenting exceptions . . . 66
8.9 Exceptions and logging . . . 67
8.10 Download the Source Code . . . 67
8.11 What's next . . . 67

9 Concurrency best practices **68**

9.1 Introduction . . . 68
9.2 Threads and Thread Groups . . . 68
9.3 Concurrency, Synchronization and Immutability . . . 69
9.4 Futures, Executors and Thread Pools . . . 70
9.5 Locks . . . 71
9.6 Thread Schedulers . . . 73
9.7 Atomic Operations . . . 73
9.8 Concurrent Collections . . . 74
9.9 Explore Java standard library . . . 74
9.10 Using Synchronization Wisely . . . 75
9.11 Wait/Notify . . . 75
9.12 Troubleshooting Concurrency Issues . . . 76
9.13 Download . . . 76
9.14 What's next . . . 77

10 Built-in Serialization techniques **78**

10.1 Introduction . . . 78
10.2 Serializable interface . . . 78
10.3 Externalizable interface . . . 79
10.4 More about Serializable interface . . . 80
10.5 Serializability and Remote Method Invocation (RMI) . . . 81
10.6 JAXB . . . 82
10.7 JSON-P . . . 83
10.8 Cost of serialization . . . 84
10.9 Beyond Java standard library and specifications . . . 84
10.10 Download the Source code . . . 84
10.11 What's next . . . 84

11 How to use Reflection effectively **85**

11.1 Introduction . . . 85
11.2 Reflection API . . . 85
11.3 Accessing generic type parameters . . . 86
11.4 Reflection API and visibility . . . 87
11.5 Reflection API pitfalls . . . 87
11.6 Method Handles . . . 88
11.7 Method Argument Names . . . 88
11.8 Download the Source Code . . . 89
11.9 What's next . . . 89

12 Dynamic languages support **90**

12.1 Introduction . . . 90
12.2 Dynamic Languages Support . . . 90
12.3 Scripting API . . . 90
12.4 JavaScript on JVM . . . 91
12.5 Groovy on JVM . . . 91
12.6 Ruby on JVM . . . 93
12.7 Python on JVM . . . 93
12.8 Using Scripting API . . . 94
12.9 Download Code . . . 94
12.10 What's next . . . 95

13 Java Compiler API **96**

13.1 Introduction . . . 96
13.2 Java Compiler API . . . 96
13.3 Annotation Processors . . . 98
13.4 Element Scanners . . . 98
13.5 Java Compiler Tree API . . . 101
13.6 Download . . . 102
13.7 What's next . . . 102

14 Java Annotation Processors **103**

14.1 Introduction . . . 103
14.2 When to Use Annotation Processors . . . 103
14.3 Annotation Processing Under the Hood . . . 103
14.4 Writing Your Own Annotation Processor . . . 104
14.5 Running Annotation Processors . . . 107
14.6 Download the source code . . . 108
14.7 What's next . . . 108

15 Java Agents **109**

15.1 Introduction . . . 109

15.2 Java Agent Basics . . . 109

15.3 Java Agent and Instrumentation . . . 110

15.4 Writing Your First Java Agent . . . 110

15.5 Running Java Agents . . . 112

15.6 Download the source code . . . 113

15.7 What's next . . . 113

Chapter 1

How to create and destroy objects

1.1 Introduction

Java programming language, originated in Sun Microsystems and released back in 1995, is one of the most widely used programming languages in the world, according to TIOBE Programming Community Index. Java is a general-purpose programming language. It is attractive to software developers primarily due to its powerful library and runtime, simple syntax, rich set of supported platforms (Write Once, Run Anywhere - WORA) and awesome community.

In this tutorial we are going to cover advanced Java concepts, assuming that our readers already have some basic knowledge of the language. It is by no means a complete reference, rather a detailed guide to move your Java skills to the next level.

Along the course, there will be a lot of code snippets to look at. Where it makes sense, the same example will be presented using Java 7 syntax as well as Java 8 one.

1.2 Instance Construction

Java is object-oriented language and as such the creation of new class instances (objects) is, probably, the most important concept of it. Constructors are playing a central role in new class instance initialization and Java provides a couple of favors to define them.

1.2.1 Implicit (Generated) Constructor

Java allows to define a class without any constructors but it does not mean the class will not have any. For example, let us consider this class:

```
package com.javacodegeeks.advanced.construction;

public class NoConstructor {
}
```

This class has no constructor but Java compiler will generate one implicitly and the creation of new class instances will be possible using `new` keyword.

```
final NoConstructor noConstructorInstance = new NoConstructor();
```

1.2.2 Constructors without Arguments

The constructor without arguments (or no-arg constructor) is the simplest way to do Java compiler's job explicitly.

```
package com.javacodegeeks.advanced.construction;

public class NoArgConstructor {
    public NoArgConstructor() {
        // Constructor body here
    }
}
```

This constructor will be called once new instance of the class is created using the `new` keyword.

```
final NoArgConstructor noArgConstructor = new NoArgConstructor();
```

1.2.3 Constructors with Arguments

The constructors with arguments are the most interesting and useful way to parameterize new class instances creation. The following example defines a constructor with two arguments.

```
package com.javacodegeeks.advanced.construction;

public class ConstructorWithArguments {
    public ConstructorWithArguments(final String arg1,final String arg2) {
        // Constructor body here
    }
}
```

In this case, when class instance is being created using the `new` keyword, both constructor arguments should be provided.

```
final ConstructorWithArguments constructorWithArguments =
    new ConstructorWithArguments( "arg1", "arg2" );
```

Interestingly, the constructors can call each other using the special `this` keyword. It is considered a good practice to chain constructors in such a way as it reduces code duplication and basically leads to having single initialization entry point. As an example, let us add another constructor with only one argument.

```
public ConstructorWithArguments(final String arg1) {
    this(arg1, null);
}
```

1.2.4 Initialization Blocks

Java has yet another way to provide initialization logic using initialization blocks. This feature is rarely used but it is better to know it exists.

```
package com.javacodegeeks.advanced.construction;

public class InitializationBlock {
    {
        // initialization code here
    }
}
```

In a certain way, the initialization block might be treated as anonymous no-arg constructor. The particular class may have multiple initialization blocks and they all will be called in the order they are defined in the code. For example:

```
package com.javacodegeeks.advanced.construction;

public class InitializationBlocks {
```

```
    {
        // initialization code here
    }

    {
        // initialization code here
    }

}
```

Initialization blocks do not replace the constructors and may be used along with them. But it is very important to mention that initialization blocks are always called **before** any constructor.

```
package com.javacodegeeks.advanced.construction;

public class InitializationBlockAndConstructor {
    {
        // initialization code here
    }

    public InitializationBlockAndConstructor() {
    }
}
```

1.2.5 Construction guarantee

Java provides certain initialization guarantees which developers may rely on. Uninitialized instance and class (static) variables are automatically initialized to their default values.

Table 1.1: datasheet

Type	Default Value
boolean	False
byte	0
short	0
int	0
long	0L
char	u0000
float	0.0f
double	0.0d
object reference	null

Let us confirm that using following class as a simple example:

```
package com.javacodegeeks.advanced.construction;

public class InitializationWithDefaults {
    private boolean booleanMember;
    private byte byteMember;
    private short shortMember;
    private int intMember;
    private long longMember;
    private char charMember;
    private float floatMember;
    private double doubleMember;
    private Object referenceMember;
```

```
    public InitializationWithDefaults() {
        System.out.println( "booleanMember = " + booleanMember );
        System.out.println( "byteMember = " + byteMember );
        System.out.println( "shortMember = " + shortMember );
        System.out.println( "intMember = " + intMember );
        System.out.println( "longMember = " + longMember );
        System.out.println( "charMember = " +
            Character.codePointAt( new char[] { charMember }, 0  ) );
        System.out.println( "floatMember = " + floatMember );
        System.out.println( "doubleMember = " + doubleMember );
        System.out.println( "referenceMember = " + referenceMember );
    }
}
```

Once instantiated using `new` keyword:

```
final InitializationWithDefaults initializationWithDefaults = new  ↩
    InitializationWithDefaults(),
```

The following output will be shown in the console:

```
booleanMember = false
byteMember = 0
shortMember = 0
intMember = 0
longMember = 0
charMember = 0
floatMember = 0.0
doubleMember = 0.0
referenceMember = null
```

1.2.6 Visibility

Constructors are subject to Java visibility rules and can have access control modifiers which determine if other classes may invoke a particular constructor.

Table 1.2: datasheet

Modifier	Package	Subclass	Everyone Else
public	accessible	accessible	accessible
protected	accessible	accessible	**not accessible**
<no modifier>	accessible	**not accessible**	**not accessible**
private	**not accessible**	**not accessible**	**not accessible**

1.2.7 Garbage collection

Java (and JVM in particular) uses automatic garbage collection. To put it simply, whenever new objects are created, the memory is automatically allocated for them. Consequently, whenever the objects are not referenced anymore, they are destroyed and their memory is reclaimed.

Java garbage collection is generational and is based on assumption that most objects die young (not referenced anymore shortly after their creation and as such can be destroyed safely). Most developers used to believe that objects creation in Java is slow and instantiation of the new objects should be avoided as much as possible. In fact, it does not hold true: the objects creation in Java is quite cheap and fast. What is expensive though is an unnecessary creation of long-lived objects which eventually may fill up old generation and cause stop-the-world garbage collection.

1.2.8 Finalizers

So far we have talked about constructors and objects initialization but have not actually mentioned anything about their counterpart: objects destruction. That is because Java uses garbage collection to manage objects lifecycle and it is the responsibility of garbage collector to destroy unnecessary objects and reclaim the memory.

However, there is one particular feature in Java called **finalizers** which resemble a bit the destructors but serves the different purpose of performing resources cleanup. **Finalizers** are considered to be a dangerous feature (which leads to numerous side-effects and performance issues). Generally, they are not necessary and should be avoided (except very rare cases mostly related to native objects). A much better alternative to **finalizers** is the introduced by **Java 7** language construct called **try-with-resources** and AutoCloseable interface which allows to write clean code like this:

```
try ( final InputStream in = Files.newInputStream( path ) ) {
    // code here
}
```

1.3 Static initialization

So far we have looked through class instance construction and initialization. But Java also supports class-level initialization constructs called **static initializers**. There are very similar to the initialization blocks except for the additional `static` keyword. Please notice that static initialization is performed once per class-loader. For example:

```
package com.javacodegeeks.advanced.construction;

public class StaticInitializationBlock {
    static {
        // static initialization code here
    }
}
```

Similarly to initialization blocks, you may include any number of static initializer blocks in the class definition and they will be executed in the order in which they appear in the code. For example:

```
package com.javacodegeeks.advanced.construction;

public class StaticInitializationBlocks {
    static {
        // static initialization code here
    }

    static {
        // static initialization code here
    }
}
```

Because static initialization block can be triggered from multiple parallel threads (when the loading of the class happens in the first time), Java runtime guarantees that it will be executed only once and in thread-safe manner.

1.4 Construction Patterns

Over the years a couple of well-understood and widely applicable construction (or creation) patterns have emerged within Java community. We are going to cover the most famous of them: singleton, helpers, factory and dependency injection (also known as inversion of control).

1.4.1 Singleton

Singleton is one of the oldest and controversial patterns in software developer's community. Basically, the main idea of it is to ensure that only one single instance of the class could be created at any given time. Being so simple however, singleton raised a lot of the discussions about how to make it right and, in particular, thread-safe. Here is how a naive version of singleton class may look like:

```
package com.javacodegeeks.advanced.construction.patterns;

public class NaiveSingleton {
    private static NaiveSingleton instance;

    private NaiveSingleton() {
    }

    public static NaiveSingleton getInstance() {
        if( instance == null ) {
            instance = new NaiveSingleton();
        }

        return instance;
    }
}
```

At least one problem with this code is that it may create many instances of the class if called concurrently by multiple threads. One of the ways to design singleton properly (but in non-lazy fashion) is using the `static final` property of the class.

```
final property of the class.
package com.javacodegeeks.advanced.construction.patterns;

public class EagerSingleton {
    private static final EagerSingleton instance = new EagerSingleton();

    private EagerSingleton() {
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

If you do not want to waste your resources and would like your singletons to be lazily created when they are really needed, the explicit synchronization is required, potentially leading to lower concurrency in a multithreaded environments (more details about concurrency in Java will be discussing in **part 9** of the tutorial, **Concurrency best practices**).

```
package com.javacodegeeks.advanced.construction.patterns;

public class LazySingleton {
    private static LazySingleton instance;

    private LazySingleton() {
    }

    public static synchronized LazySingleton getInstance() {
        if( instance == null ) {
            instance = new LazySingleton();
        }

        return instance;
    }
}
```

Nowadays, singletons are not considered to be a good choice in most cases, primarily because they are making a code very hard to test. The domination of dependency injection pattern (please see the **Dependency Injection** section below) also makes singletons unnecessary.

1.4.2 Utility/Helper Class

The utility or helper classes are quite popular pattern used by many Java developers. Basically, it represents the non-instantiable class (with constructor declared as `private`), optionally declared as `final` (more details about declaring classes as `final` will be provided in **part 3** of the tutorial, **How to design Classes and Interfaces**) and contains `static` methods only. For example:

```
package com.javacodegeeks.advanced.construction.patterns;

public final class HelperClass {
    private HelperClass() {
    }

    public static void helperMethod1() {
        // Method body here
    }

    public static void helperMethod2() {
        // Method body here
    }
}
```

From seasoned software developer standpoint, such helpers often become containers for all kind of non-related methods which have not found other place to be put in but should be shared somehow and used by other classes. Such design decisions should be avoided in most cases: it is always possible to find another way to reuse the required functionality, keeping the code clean and concise.

1.4.3 Factory

Factory pattern is proven to be extremely useful technique in the hands of software developers. As such, it has several flavors in Java, ranging from **factory method** to **abstract factory**. The simplest example of factory pattern is a `static` method which returns new instance of a particular class (**factory method**). For example:

```
package com.javacodegeeks.advanced.construction.patterns;

public class Book {
    private Book( final String title) {
    }

    public static Book newBook( final String title ) {
        return new Book( title );
    }
}
```

The one may argue that it does not make a lot of sense to introduce the `newBook` **factory method** but using such a pattern often makes the code more readable. Another variance of factory pattern involves interfaces or abstract classes (**abstract factory**). For example, let us define a **factory interface**:

```
public interface BookFactory {
    Book newBook();
}
```

With couple of different implementations, depending on the library type:

```
public class Library implements BookFactory {
    @Override
    public Book newBook() {
        return new PaperBook();
    }
}

public class KindleLibrary implements BookFactory {
    @Override
    public Book newBook() {
        return new KindleBook();
    }
}
```

Now, the particular class of the `Book` is hidden behind `BookFactory` interface implementation, still providing the generic way to create books.

1.4.4 Dependency Injection

Dependency injection (also known as inversion of control) is considered as a good practice for class designers: if some class instance depends on the other class instances, those dependencies should be provided (injected) to it by means of constructors (or setters, strategies, etc.) but not created by the instance itself. Let us consider the following example:

```
package com.javacodegeeks.advanced.construction.patterns;

import java.text.DateFormat;
import java.util.Date;

public class Dependant {
    private final DateFormat format = DateFormat.getDateInstance();

    public String format( final Date date ) {
        return format.format( date );
    }
}
```

The class `Dependant` needs an instance of DateFormat and it just creates one by calling `DateFormat.getDateInstance()` at construction time. The better design would be to use constructor argument to do the same thing:

```
package com.javacodegeeks.advanced.construction.patterns;

import java.text.DateFormat;
import java.util.Date;

public class Dependant {
    private final DateFormat format;

    public Dependant( final DateFormat format ) {
        this.format = format;
    }

    public String format( final Date date ) {
        return format.format( date );
    }
}
```

In this case the class has all its dependencies provided from outside and it would be very easy to change date format and write test cases for it.

1.5 Download the Source Code

- You may download the source code here: advanced-java-part-1

1.6 What's next

In this part of the tutorial we have looked at classes and class instances construction and initialization techniques, along the way covering several widely used patterns. In the next part we are going to dissect the `Object` class and usage of its well-known methods: `equals`, `hashCode`, `toString` and `clone`.

Chapter 2

Using methods common to all objects

2.1 Introduction

From **part 1** of the tutorial, **How to create and destroy objects**, we already know that Java is an object-oriented language (however, not a pure object-oriented one). On top of the Java class hierarchy sits the Object class and every single class in Java implicitly is inherited from it. As such, all classes inherit the set of methods declared in `Object` class, most importantly the following ones:

Table 2.1: datasheet

Method	**Description**
`protected Object clone()`	Creates and returns a copy of this object.
`protected void finalize()`	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. We have discussed finalizers in the **part 1** of the tutorial, **How to create and destroy objects**.
`boolean equals(Object obj)`	Indicates whether some other object is "equal to" this one.
`int hashCode()`	Returns a hash code value for the object.
`String toString()`	Returns a string representation of the object.
`void notify()`	Wakes up a single thread that is waiting on this object's monitor. We are going to discuss this method in the **part 9** of the tutorial, **Concurrency best practices**.
`void notifyAll()`	Wakes up all threads that are waiting on this object's monitor. We are going to discuss this method in the **part 9** of the tutorial, **Concurrency best practices**.

`void wait()`
`void wait(long timeout)`
`void wait(long timeout, int nanos)`
	Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object. We are going to discuss these methods in the **part 9** of the tutorial, **Concurrency best practices**.

In this part of the tutorial we are going to look at `equals`,`hashCode`,`toString` and `clone` methods, their usage and important constraints to keep in mind.

2.2 Methods equals and hashCode

By default, any two object references (or class instance references) in Java are equal only if they are referring to the same memory location (reference equality). But Java allows classes to define their own equality rules by overriding the `equals()` method of the `Object` class. It sounds like a powerful concept, however the correct `equals()` method implementation should conform to a set of rules and satisfy the following constraints:

- **Reflexive**. Object **x** must be equal to itself and **equals(x)** must return **true**.
- **Symmetric**. If **equals(y)** returns **true** then **y.equals(x)** must also return **true**.
- **Transitive**. If **equals(y)** returns **true** and **y.equals(z)** returns **true**, then **x.equals(z)** must also return **true**.
- **Consistent**. Multiple invocation of **equals()** method must result into the same value, unless any of the properties used for equality comparison are modified.
- **Equals To Null**. The result of **equals(null)** must be always **false**.

Unfortunately, the Java compiler is not able to enforce those constraints during the compilation process. However, not following these rules may cause very weird and hard to troubleshoot issues. The general advice is this: if you ever are going to write your own `equals()` method implementation, think twice if you really need it. Now, armed with all these rules, let us write a simple implementation of the `equals()` method for the `Person` class.

```
package com.javacodegeeks.advanced.objects;

public class Person {
    private final String firstName;
    private final String lastName;
    private final String email;

    public Person( final String firstName, final String lastName, final String email ) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    // Step 0: Please add the @Override annotation, it will ensure that your
    // intention is to change the default implementation.
    @Override
    public boolean equals( Object obj ) {
        // Step 1: Check if the 'obj' is null
        if ( obj == null ) {
            return false;
        }

        // Step 2: Check if the 'obj' is pointing to the this instance
        if ( this == obj ) {
            return true;
        }
```

```
        // Step 3: Check classes equality. Note of caution here: please do not use the
        // 'instanceof' operator unless class is declared as final. It may cause
        // an issues within class hierarchies.
        if ( getClass() != obj.getClass() ) {
            return false;
        }

        // Step 4: Check individual fields equality
        final Person other = (Person) obj;
        if ( email == null ) {
            if ( other.email != null ) {
                return false;
            }
        } else if( !email.equals( other.email ) ) {
            return false;
        }

        if ( firstName == null ) {
            if ( other.firstName != null ) {
                return false;
            }
        } else if ( !firstName.equals( other.firstName ) ) {
            return false;
        }

        if ( lastName == null ) {
            if ( other.lastName != null ) {
                return false;
            }
        } else if ( !lastName.equals( other.lastName ) ) {
            return false;
        }

        return true;
    }
}
```

It is not by accident that this section also includes the `hashCode()` method in its title. The last, but not least, rule to remember: whenever you override `equals()` method, always override the `hashCode()` method as well. If for any two objects the `equals()` method returns **true**, then the `hashCode()` method on each of those two objects must return the same integer value (however the opposite statement is not as strict: if for any two objects the `equals()` method returns **false**, the `hashCode()` method on each of those two objects may or may not return the same integer value). Let us take a look on `hashCode()` method for the `Person` class.

```
// Please add the @Override annotation, it will ensure that your
// intention is to change the default implementation.
@Override
public int hashCode() {
    final int prime = 31;

    int result = 1;
    result = prime * result + ( ( email == null ) ? 0 : email.hashCode() );
    result = prime * result + ( ( firstName == null ) ? 0 : firstName.hashCode() );
    result = prime * result + ( ( lastName == null ) ? 0 : lastName.hashCode() );

    return result;
}
```

To protect yourself from surprises, whenever possible try to use `final` fields while implementing `equals()` and `hashCode()`. It will guarantee that behavior of those methods will not be affected by the field changes (however, in real-world projects

it is not always possible).

Finally, always make sure that the same fields are used within implementation of `equals()` and `hashCode()` methods. It will guarantee consistent behavior of both methods in case of any change affecting the fields in question.

2.3 Method toString

The `toString()` is arguably the most interesting method among the others and is being overridden more frequently. Its purpose is it to provide the string representation of the object (class instance). The properly written `toString()` method can greatly simplify debugging and troubleshooting of the issues in real-live systems.

The default `toString()` implementation is not very useful in most cases and just returns the full class name and object hash code, separated by `@`, f.e.:

```
com.javacodegeeks.advanced.objects.Person@6104e2ee
```

Let us try to improve the implementation and override the `toString()` method for our **Person** class example. Here is a one of the ways to make `toString()` more useful.

```
// Please add the @Override annotation, it will ensure that your
// intention is to change the default implementation.
@Override
public String toString() {
    return String.format( "%s[email=%s, first name=%s, last name=%s]",
        getClass().getSimpleName(), email, firstName, lastName );
}
```

Now, the `toString()` method provides the string version of the `Person` class instance with all its fields included. For example, while executing the code snippet below:

```
final Person person = new Person( "John", "Smith", "john.smith@domain.com" );
System.out.println( person.toString() );
```

The following output will be printed out in the console:

```
Person[email=john.smith@domain.com, first name=John, last name=Smith]
```

Unfortunately, the standard Java library has a limited support to simplify `toString()` method implementations, notably, the most useful methods are `Objects.toString()`, `Arrays.toString()` /`Arrays.deepToString()`. Let us take a look on the `Office` class and its possible `toString()` implementation.

```
package com.javacodegeeks.advanced.objects;

import java.util.Arrays;

public class Office {
    private Person[] persons;

    public Office( Person ... persons ) {
         this.persons = Arrays.copyOf( persons, persons.length );
    }

    @Override
    public String toString() {
        return String.format( "%s{persons=%s}",
            getClass().getSimpleName(), Arrays.toString( persons ) );
    }

    public Person[] getPersons() {
        return persons;
    }
}
```

The following output will be printed out in the console (as we can see the `Person` class instances are properly converted to string as well):

```
Office{persons=[Person[email=john.smith@domain.com, first name=John, last name=Smith]]}
```

The Java community has developed a couple of quite comprehensive libraries which help a lot to make `toString()` implementations painless and easy. Among those are Google Guava's Objects.toStringHelper and Apache Commons Lang ToStringBuilder.

2.4 Method clone

If there is a method with a bad reputation in Java, it is definitely `clone()`. Its purpose is very clear - return the exact copy of the class instance it is being called on, however there are a couple of reasons why it is not as easy as it sounds.

First of all, in case you have decided to implement your own `clone()` method, there are a lot of conventions to follow as stated in Java documentation. Secondly, the method is declared `protected` in `Object` class so in order to make it visible, it should be overridden as `public` with return type of the overriding class itself. Thirdly, the overriding class should implement the `Cloneable` interface (which is just a marker or mixin interface with no methods defined) otherwise CloneNotSupportedException exception will be raised. And lastly, the implementation should call `super.clone()` first and then perform additional actions if needed. Let us see how it could be implemented for our sample `Person` class.

```
public class Person implements Cloneable {
    // Please add the @Override annotation, it will ensure that your
    // intention is to change the default implementation.
    @Override
    public Person clone() throws CloneNotSupportedException {
        return ( Person )super.clone();
    }
}
```

The implementation looks quite simple and straightforward, so what could go wrong here? Couple of things, actually. While the cloning of the class instance is being performed, no class constructor is being called. The consequence of such a behavior is that unintentional data sharing may come out. Let us consider the following example of the `Office` class, introduced in previous section:

```
package com.javacodegeeks.advanced.objects;

import java.util.Arrays;

public class Office implements Cloneable {
    private Person[] persons;

    public Office( Person ... persons ) {
         this.persons = Arrays.copyOf( persons, persons.length );
    }

    @Override
    public Office clone() throws CloneNotSupportedException {
        return ( Office )super.clone();
    }

    public Person[] getPersons() {
        return persons;
    }
}
```

In this implementation, all the clones of the `Office` class instance will share the same persons array, which is unlikely the desired behavior. A bit of work should be done in order to make the `clone()` implementation to do the right thing.

```
@Override
public Office clone() throws CloneNotSupportedException {
    final Office clone = ( Office )super.clone();
    clone.persons = persons.clone();
    return clone;
}
```

It looks better now but even this implementation is very fragile as making the persons field to be `final` will lead to the same data sharing issues (as `final` cannot be reassigned).

By and large, if you would like to make exact copies of your classes, probably it is better to avoid `clone()` and `Cloneable` and use much simpler alternatives (for example, copying constructor, quite familiar concept to developers with C++ background, or factory method, a useful construction pattern we have discussed in **part 1** of the tutorial, **How to create and destroy objects**).

2.5 Method equals and == operator

There is an interesting relation between Java == operator and **equals()** method which causes a lot of issues and confusion. In most cases (except comparing primitive types), **==** operator performs referential equality: it returns **true** if both references point to the same object, and **false** otherwise. Let us take a look on a simple example which illustrates the differences:

```
final String str1 = new String( "bbb" );
System.out.println( "Using == operator: " + ( str1 == "bbb" ) );
System.out.println( "Using equals() method: " + str1.equals( "bbb" ) );
```

From the human being prospective, there are no differences between str1=="bbb" and str1.equals("bbb"): in both cases the result should be the same as str1 is just a reference to "bbb" string. But in Java it is not the case:

```
Using == operator: false
Using equals() method: true
```

Even if both strings look exactly the same, in this particular example they exist as two different string instances. As a rule of thumb, if you deal with object references, always use the `equals()` or `Objects.equals()` (see please next section Useful helper classes for more details) to compare for equality, unless you really have an intention to compare if object references are pointing to the same instance.

2.6 Useful helper classes

Since the release of Java 7, there is a couple of very useful helper classes included with the standard Java library. One of them is class Objects. In particular, the following three methods can greatly simplify your own `equals()` and `hashCode()` method implementations.

Table 2.2: datasheet

Method	Description
`static boolean equals(Object a, Object b)`	Returns **true** if the arguments are equal to each other and **false** otherwise.
`static int hash(Object...values)`	Generates a hash code for a sequence of input values.
`static int hashCode(Object o)`	Returns the hash code of a non-null argument and 0 for a null argument.

If we rewrite `equals()` and `hashCode()` method for our `Person`'s class example using these helper methods, the amount of the code is going to be significantly smaller, plus the code becomes much more readable.

```
@Override
public boolean equals( Object obj ) {
    if ( obj == null ) {
        return false;
    }

    if ( this == obj ) {
        return true;
    }

    if ( getClass() != obj.getClass() ) {
        return false;
    }

    final PersonObjects other = (PersonObjects) obj;
    if( !Objects.equals( email, other.email ) ) {
        return false;
    } else if( !Objects.equals( firstName, other.firstName ) ) {
        return false;
    } else if( !Objects.equals( lastName, other.lastName ) ) {
        return false;
    }

    return true;
}

@Override
public int hashCode() {
    return Objects.hash( email, firstName, lastName );
}
```

2.7 Download the Source Code

- You may download the source code here: advanced-java-part-2

2.8 What's next

In this section we have covered the `Object` class which is the foundation of object-oriented programming in Java. We have seen how each class may override methods inherited from `Object` class and impose its own equality rules. In the next section we are going to switch our gears from coding and discuss how to properly design your classes and interfaces.

Chapter 3

How to design Classes and Interfaces

3.1 Introduction

Whatever programming language you are using (and Java is not an exception here), following good design principles is a key factor to write clean, understandable, testable code and deliver long-living, easy to maintain solutions. In this part of the tutorial we are going to discuss the foundational building blocks which the Java language provides and introduce a couple of design principles, aiming to help you to make better design decisions.

More precisely, we are going to discuss **interfaces** and **interfaces with default methods** (new feature of Java 8), **abstract** and **final classes, immutable classes, inheritance, composition** and revisit a bit the **visibility** (or accessibility) rules we have briefly touched in **part 1** of the tutorial, **How to create and destroy objects**.

3.2 Interfaces

In object-oriented programming, the concept of interfaces forms the basics of contract-driven (or contract-based) development. In a nutshell, interfaces define the set of methods (contract) and every class which claims to support this particular interface must provide the implementation of those methods: a pretty simple, but powerful idea.

Many programming languages do have interfaces in one form or another, but Java particularly provides language support for that. Let take a look on a simple interface definition in Java.

```
package com.javacodegeeks.advanced.design;

public interface SimpleInterface {
    void performAction();
}
```

In the code snippet above, the interface which we named `SimpleInterface` declares just one method with name `performAction`. The principal differences of interfaces in respect to classes is that interfaces outline what the contact is (declare methods), but do not provide their implementations.

However, interfaces in Java can be more complicated than that: they can include nested interfaces, classes, enumerations, annotations (enumerations and annotations will be covered in details in **part 5** of the tutorial, **How and when to use Enums and Annotations**) and constants. For example:

```
package com.javacodegeeks.advanced.design;

public interface InterfaceWithDefinitions {
    String CONSTANT = "CONSTANT";

    enum InnerEnum {
        E1, E2;
```

```
    }

    class InnerClass {
    }

    interface InnerInterface {
        void performInnerAction();
    }

    void performAction();
}
```

With this more complicated example, there are a couple of constraints which interfaces implicitly impose with respect to the nested constructs and method declarations, and Java compiler enforces that. First and foremost, even if it is not being said explicitly, every declaration in the interface is **public** (and can be only **public**, for more details about visibility and accessibility rules, please refer to section Visibility). As such, the following method declarations are equivalent:

```
public void performAction();
void performAction();
```

Worth to mention that every single method in the interface is implicitly declared as **abstract** and even these method declarations are equivalent:

```
public abstract void performAction();
public void performAction();
void performAction();
```

As for the constant field declarations, additionally to being `public`, they are implicitly `static` and `final` so the following declarations are also equivalent:

```
String CONSTANT = "CONSTANT";
public static final String CONSTANT = "CONSTANT";
```

And finally, the nested classes, interfaces or enumerations, additionally to being `public`, are implicitly declared as `static`. For example, those class declarations are equivalent as well:

```
class InnerClass {
}

static class InnerClass {
}
```

Which style you are going to choose is a personal preference, however knowledge of those simple qualities of interfaces could save you from unnecessary typing.

3.3 Marker Interfaces

Marker interfaces are a special kind of interfaces which have no methods or other nested constructs defined. We have already seen one example of the marker interface in **part 2** of the tutorial **Using methods common to all objects**, the interface `Cloneable`. Here is how it is defined in the Java library:

```
public interface Cloneable {
}
```

Marker interfaces are not contracts per se but somewhat useful technique to “attach” or “tie” some particular trait to the class. For example, with respect to `Cloneable`, the class is marked as being available for cloning however the way it should or could be done is not a part of the interface. Another very well-known and widely used example of marker interface is `Serializable`:

```
public interface Serializable {
}
```

This interface marks the class as being available for serialization and deserialization, and again, it does not specify the way it could or should be done.

The marker interfaces have their place in object-oriented design, although they do not satisfy the main purpose of interface to be a contract.

3.4 Functional interfaces, default and static methods

With the release of Java 8, interfaces have obtained new very interesting capabilities: static methods, default methods and automatic conversion from lambdas (functional interfaces).

In section Interfaces we have emphasized on the fact that interfaces in Java can only declare methods but are not allowed to provide their implementations. With default methods it is not true anymore: an interface can mark a method with the `default` keyword and provide the implementation for it. For example:

```
package com.javacodegeeks.advanced.design;

public interface InterfaceWithDefaultMethods {
    void performAction();

    default void performDefaulAction() {
        // Implementation here
    }
}
```

Being an instance level, defaults methods could be overridden by each interface implementer, but from now, interfaces may also include `static` methods, for example:

```
package com.javacodegeeks.advanced.design;

public interface InterfaceWithDefaultMethods {
    static void createAction() {
        // Implementation here
    }
}
```

One may say that providing an implementation in the interface defeats the whole purpose of contract-based development, but there are many reasons why these features were introduced into the Java language and no matter how useful or confusing they are, they are there for you to use.

The functional interfaces are a different story and they are proven to be very helpful add-on to the language. Basically, the functional interface is the interface with just a single abstract method declared in it. The `Runnable` interface from Java standard library is a good example of this concept:

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

The Java compiler treats functional interfaces differently and is able to convert the lambda function into the functional interface implementation where it makes sense. Let us take a look on following function definition:

```
public void runMe( final Runnable r ) {
    r.run();
}
```

To invoke this function in Java 7 and below, the implementation of the `Runnable` interface should be provided (for example using Anonymous classes), but in Java 8 it is enough to pass `run()` method implementation using lambda syntax:

```
runMe( () -> System.out.println( "Run!" ) );
```

Additionally, the `@FunctionalInterface` annotation (annotations will be covered in details in **part 5** of the tutorial, **How and when to use Enums and Annotations**) hints the compiler to verify that the interface contains only one abstract method so any changes introduced to the interface in the future will not break this assumption.

3.5 Abstract classes

Another interesting concept supported by Java language is the notion of abstract classes. Abstract classes are somewhat similar to the interfaces in Java 7 and very close to interfaces with default methods in Java 8. By contrast to regular classes, abstract classes cannot be instantiated but could be subclassed (please refer to the section Inheritance for more details). More importantly, abstract classes may contain abstract methods: the special kind of methods without implementations, much like interfaces do. For example:

```
package com.javacodegeeks.advanced.design;

public abstract class SimpleAbstractClass {
    public void performAction() {
        // Implementation here
    }

    public abstract void performAnotherAction();
}
```

In this example, the class `SimpleAbstractClass` is declared as `abstract` and has one `abstract` method declaration as well. Abstract classes are very useful when most or even some part of implementation details could be shared by many subclasses. However, they still leave the door open and allow customizing the intrinsic behavior of each subclass by means of abstract methods.

One thing to mention, in contrast to interfaces which can contain only `public` declarations, abstract classes may use the full power of accessibility rules to control abstract methods visibility (please refer to the sections Visibility and Inheritance for more details).

3.6 Immutable classes

Immutability is becoming more and more important in the software development nowadays. The rise of multi-core systems has raised a lot of concerns related to data sharing and concurrency (in the **part 9**, **Concurrency best practices**, we are going to discuss in details those topics). But the one thing definitely emerged: less (or even absence of) mutable state leads to better scalability and simpler reasoning about the systems.

Unfortunately, the Java language does not provide strong support for class immutability. However using a combination of techniques it is possible to design classes which are immutable. First and foremost, all fields of the class should be `final`. It is a good start but does not guarantee immutability alone.

```
package com.javacodegeeks.advanced.design;

import java.util.Collection;

public class ImmutableClass {
    private final long id;
    private final String[] arrayOfStrings;
    private final Collection< String > collectionOfString;
}
```

Secondly, follow the proper initialization: if the field is the reference to a collection or an array, do not assign those fields directly from constructor arguments, make the copies instead. It will guarantee that state of the collection or array will not be changed from outside.

```
public ImmutableClass( final long id, final String[] arrayOfStrings,
        final Collection< String > collectionOfString) {
    this.id = id;
    this.arrayOfStrings = Arrays.copyOf( arrayOfStrings, arrayOfStrings.length );
    this.collectionOfString = new ArrayList<>( collectionOfString );
}
```

And lastly, provide the proper accessors (getters). For the collection, the immutable view should be exposed using `Collections.unmodifiableXxx` wrappers.

```
public Collection<String> getCollectionOfString() {
    return Collections.unmodifiableCollection( collectionOfString );
}
```

With arrays, the only way to ensure true immutability is to provide a copy instead of returning reference to the array. That might not be acceptable from a practical standpoint as it hugely depends on array size and may put a lot of pressure on garbage collector.

```
public String[] getArrayOfStrings() {
    return Arrays.copyOf( arrayOfStrings, arrayOfStrings.length );
}
```

Even this small example gives a good idea that immutability is not a first class citizen in Java yet. Things can get really complicated if an immutable class has fields referencing another class instances. Those classes should also be immutable however there is no simple way to enforce that.

There are a couple of great Java source code analyzers like **FindBugs**) and **PMD**) which may help a lot by inspecting your code and pointing to the common Java programming flaws. Those tools are great friends of any Java developer.

3.7 Anonymous classes

In the pre-Java 8 era, anonymous classes were the only way to provide in-place class definitions and immediate instantiations. The purpose of the anonymous classes was to reduce boilerplate and provide a concise and easy way to represent classes as expressions. Let us take a look on the typical old-fashioned way to spawn new thread in Java:

```
package com.javacodegeeks.advanced.design;

public class AnonymousClass {
    public static void main( String[] args ) {
        new Thread(
            // Example of creating anonymous class which implements
            // Runnable interface
            new Runnable() {
                @Override
                public void run() {
                    // Implementation here
                }
            }
        ).start();
    }
}
```

In this example, the implementation of the `Runnable` interface is provided in place as anonymous class. Although there are some limitations associated with anonymous classes, the fundamental disadvantages of their usage are a quite verbose syntax constructs which Java imposes as a language. Even the simplest anonymous class which does nothing requires at least 5 lines of code to be written every time.

```
    new Runnable() {
        @Override
        public void run() {
        }
    }
```

Luckily, with Java 8, lambdas and functional interfaces all this boilerplate is about to gone away, finally making the Java code to look truly concise.

```
package com.javacodegeeks.advanced.design;

public class AnonymousClass {
    public static void main( String[] args ) {
        new Thread( () -> { /* Implementation here */ } ).start();
    }
}
```

3.8 Visibility

We have already talked a bit about Java visibility and accessibility rules in **part 1** of the tutorial, **How to design Classes and Interfaces**. In this part we are going to get back to this subject again but in the context of subclassing.

Table 3.1: datasheet

Modifier	Package	Subclass	Everyone Else
public	accessible	accessible	Accessible
protected	accessible	accessible	**not accessible**
<no modifier>	accessible	**not accessible**	**not accessible**
private	**not accessible**	**not accessible**	**not accessible**

Different visibility levels allow or disallow the classes to see other classes or interfaces (for example, if they are in different packages or nested in one another) or subclasses to see and access methods, constructors and fields of their parents.

In next section, Inheritance, we are going to see that in action.

3.9 Inheritance

Inheritance is one of the key concepts of object-oriented programming, serving as a basis of building class relationships. Combined together with visibility and accessibility rules, inheritance allows designing extensible and maintainable class hierarchies.

Conceptually, inheritance in Java is implemented using subclassing and the `extends` keyword, followed by the parent class. The subclass inherits all of the public and protected members of its parent class. Additionally, a subclass inherits the package-private members of the parent class if both reside in the same package. Having said that, it is very important no matter what you are trying to design, to keep the minimal set of the methods which class exposes publicly or to its subclasses. For example, let us take a look on a class `Parent` and its subclass `Child` to demonstrate different visibility levels and their effect:

```
package com.javacodegeeks.advanced.design;

public class Parent {
    // Everyone can see it
    public static final String CONSTANT = "Constant";

    // No one can access it
```

```
    private String privateField;
    // Only subclasses can access it
    protected String protectedField;

    // No one can see it
    private class PrivateClass {
    }

    // Only visible to subclasses
    protected interface ProtectedInterface {
    }

    // Everyone can call it
    public void publicAction() {
    }

    // Only subclass can call it
    protected void protectedAction() {
    }

    // No one can call it
    private void privateAction() {
    }

    // Only subclasses in the same package can call it
    void packageAction() {
    }
}
```

```
package com.javacodegeeks.advanced.design;

// Resides in the same package as parent class
public class Child extends Parent implements Parent.ProtectedInterface {
    @Override
    protected void protectedAction() {
        // Calls parent's method implementation
        super.protectedAction();
    }

    @Override
    void packageAction() {
        // Do nothing, no call to parent's method implementation
    }

    public void childAction() {
        this.protectedField = "value";
    }
}
```

Inheritance is a very large topic by itself, with a lot of subtle details specific to Java. However, there are a couple of easy to follow rules which could help a lot to keep your class hierarchies concise. In Java, every subclass may override any inherited method of its parent unless it was declared as `final` (please refer to the section Final classes and methods).

However, there is no special syntax or keyword to mark the method as being overridden which may cause a lot of confusion. That is why the `@Override` annotation has been introduced: whenever your intention is to override the inherited method, please always use the `@Override` annotation to indicate that.

Another dilemma Java developers are often facing in design is building class hierarchies (with concrete or abstract classes) versus interface implementations. It is strongly advised to prefer interfaces to classes or abstract classes whenever possible. Interfaces are much more lightweight, easier to test (using mocks) and maintain, plus they minimize the side effects of implementation changes. Many advanced programming techniques like creating class proxies in standard Java library heavily rely on interfaces.

3.10 Multiple inheritance

In contrast to C++ and some other languages, Java does not support multiple inheritance: in Java every class has exactly one direct parent (with `Object` class being on top of the hierarchy as we have already known from **part 2** of the tutorial, **Using methods common to all objects**). However, the class may implement multiple interfaces and as such, stacking interfaces is the only way to achieve (or mimic) multiple inheritance in Java.

```
package com.javacodegeeks.advanced.design;

public class MultipleInterfaces implements Runnable, AutoCloseable {
    @Override
    public void run() {
        // Some implementation here
    }

    @Override
    public void close() throws Exception {
       // Some implementation here
    }
}
```

Implementation of multiple interfaces is in fact quite powerful, but often the need to reuse an implementation leads to deep class hierarchies as a way to overcome the absence of multiple inheritance support in Java.

```
public class A implements Runnable {
    @Override
    public void run() {
        // Some implementation here
    }
}
```

```
// Class B wants to inherit the implementation of run() method from class A.
public class B extends A implements AutoCloseable {
    @Override
    public void close() throws Exception {
       // Some implementation here
    }
}
```

```
// Class C wants to inherit the implementation of run() method from class A
// and the implementation of close() method from class B.
public class C extends B implements Readable {
    @Override
    public int read(java.nio.CharBuffer cb) throws IOException {
       // Some implementation here
    }
}
```

And so on... The recent Java 8 release somewhat addressed the problem with the introduction of default methods. Because of default methods, interfaces actually have started to provide not only contract but also implementation. Consequently, the classes which implement those interfaces are automatically inheriting these implemented methods as well. For example:

```
package com.javacodegeeks.advanced.design;

public interface DefaultMethods extends Runnable, AutoCloseable {
    @Override
    default void run() {
        // Some implementation here
    }
```

```
    @Override
    default void close() throws Exception {
        // Some implementation here
    }
}

// Class C inherits the implementation of run() and close() methods from the
// DefaultMethods interface.
public class C implements DefaultMethods, Readable {
    @Override
    public int read(java.nio.CharBuffer cb) throws IOException {
        // Some implementation here
    }
}
```

Be aware that multiple inheritance is a powerful, but at the same time a dangerous tool to use. The well known "Diamond of Death" problem is often cited as the fundamental flaw of multiple inheritance implementations, so developers are urged to design class hierarchies very carefully. Unfortunately, the Java 8 interfaces with default methods are becoming the victims of those flaws as well.

```
interface A {
    default void performAction() {
    }
}

interface B extends A {
    @Override
    default void performAction() {
    }
}

interface C extends A {
    @Override
    default void performAction() {
    }
}
```

For example, the following code snippet fails to compile:

```
// E is not compilable unless it overrides performAction() as well
interface E extends B, C {
}
```

At this point it is fair to say that Java as a language always tried to escape the corner cases of object-oriented programming, but as the language evolves, some of those cases are started to pop up.

3.11 Inheritance and composition

Fortunately, inheritance is not the only way to design your classes. Another alternative, which many developers consider being better than inheritance, is composition. The idea is very simple: instead of building class hierarchies, the classes should be composed from other classes.

Let us take a look on this example:

```
public class Vehicle {
    private Engine engine;
    private Wheels[] wheels;
    // ...
}
```

The `Vehicle` class is composed out of `engine` and `wheels` (plus many other parts which are left aside for simplicity). However, one may say that `Vehicle` class is also an engine and so could be designed using the inheritance.

```
public class Vehicle extends Engine {
    private Wheels[] wheels;
    // ...
}
```

Which design decision is right? The general guidelines are known as **IS-A** and **HAS-A** principles. **IS-A** is the inheritance relationship: the subclass also satisfies the parent class specification and a such **IS-A** variation of parent class. Consequently, **HAS-A** is the composition relationship: the class owns (or **HAS-A**) the objects which belong to it. In most cases, the **HAS-A** principle works better then **IS-A** for couple of reasons:

- The design is more flexible in a way it could be changed
- The model is more stable as changes are not propagating through class hierarchies
- The class and its composites are loosely coupled compared to inheritance which tightly couples parent and its subclasses
- The reasoning about class is simpler as all its dependencies are included in it, in one place

However, the inheritance has its own place, solves real design issues in different way and should not be neglected. Please keep those two alternatives in mind while designing your object-oriented models.

3.12 Encapsulation

The concept of encapsulation in object-oriented programming is all about hiding the implementation details (like state, internal methods, etc.) from the outside world. The benefits of encapsulation are maintainability and ease of change. The less intrinsic details classes expose, the more control the developers have over changing their internal implementation, without the fear to break the existing code (a real problem if you are developing a library or framework used by many people).

Encapsulation in Java is achieved using visibility and accessibility rules. It is considered a best practice in Java to never expose the fields directly, only by means of getters and setters (if the field is not declared as `final`). For example:

```
package com.javacodegeeks.advanced.design;

public class Encapsulation {
    private final String email;
    private String address;

    public Encapsulation( final String email ) {
        this.email = email;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getEmail() {
        return email;
    }
}
```

This example resembles what is being called JavaBeans in Java language: the regular Java classes written by following the set of conventions, one of those being allow the access to fields using getter and setter methods only.

As we already emphasized in the Inheritance section, please always try to keep the class public contract minimal, following the encapsulation principle. Whatever should not be `public`, should be `private` instead (or `protected` / `package private`, depending on the problem you are solving). In long run it will pay off, giving you the freedom to evolve your design without introducing breaking changes (or at least minimize them).

3.13 Final classes and methods

In Java, there is a way to prevent the class to be subclassed by any other class: it should be declared as `final`.

```
package com.javacodegeeks.advanced.design;

public final class FinalClass {
}
```

The same `final` keyword in the method declaration prevents the method in question to be overridden in subclasses.

```
package com.javacodegeeks.advanced.design;

public class FinalMethod {
    public final void performAction() {
    }
}
```

There are no general rules to decide if class or method should be `final` or not. Final classes and methods limit the extensibility and it is very hard to think ahead if the class should or should not be subclassed, or method should or should not be overridden. This is particularly important to library developers as the design decisions like that could significantly limit the applicability of the library.

Java standard library has some examples of `final` classes, with most known being `String` class. On an early stage, the decision has been taken to proactively prevent any developer's attempts to come up with own, "better" string implementations.

3.14 Download the Source Code

This was a lesson on How to design Classes and Interfaces.

- You may download the source code here: advanced-java-part-3

3.15 What's next

In this part of the tutorial we have looked at object-oriented design concepts in Java. We also briefly walked through contract-based development, touched some functional concepts and saw how the language evolved over time. In next part of the tutorial we are going to meet generics and how they are changing the way we approach type-safe programming.

Chapter 4

How and when to use Generics

4.1 Introduction

The idea of generics represents the abstraction over types (well-known to C++ developers as templates). It is a very powerful concept (which came up quite a while ago) that allows to develop abstract algorithms and data structures and to provide concrete types to operate on later. Interestingly, generics were not present in the early versions of Java and were added along the way only in Java 5 release. And since then, it is fair to say that generics revolutionized the way Java programs are being written, delivering much stronger type guaranties and making code significantly safer.

In this section we are going to cover the usage of generics everywhere, starting from interfaces, classes and methods. Providing a lot of benefits, generics however do introduce some limitations and side-effects which we also are going to cover.

4.2 Generics and interfaces

In contrast to regular interfaces, to define a generic interface it is sufficient to provide the type (or types) it should be parameterized with. For example:

```
package com.javacodegeeks.advanced.generics;

public interface GenericInterfaceOneType< T > {
    void performAction( final T action );
}
```

The `GenericInterfaceOneType` is parameterized with single type `T`, which could be used immediately by interface declarations. The interface may be parameterized with more than one type, for example:

```
package com.javacodegeeks.advanced.generics;

public interface GenericInterfaceSeveralTypes< T, R > {
    R performAction( final T action );
}
```

Whenever any class wants to implement the interface, it has an option to provide the exact type substitutions, for example the `ClassImplementingGenericInterface` class provides `String` as a type parameter `T` of the generic interface:

```
package com.javacodegeeks.advanced.generics;

public class ClassImplementingGenericInterface
        implements GenericInterfaceOneType< String > {
    @Override
    public void performAction( final String action ) {
```

```
        // Implementation here
    }
}
```

The Java standard library has a plenty of examples of the generic interfaces, primarily within collections library. It is very easy to declare and use generic interfaces however we are going to get back to them once again when discussing bounded types (wildcards and bounded types) and generic limitations (Limitation of generics).

4.3 Generics and classes

Similarly to interfaces, the difference between regular and generic classes is only the type parameters in the class definitions. For example:

```
package com.javacodegeeks.advanced.generics;

public class GenericClassOneType< T > {
    public void performAction( final T action ) {
        // Implementation here
    }
}
```

Please notice that any class (**concrete**, **abstract** or **final**) could be parameterized using generics. One interesting detail is that the class may pass (or may not) its generic type (or types) down to the interfaces and parent classes, without providing the exact type instance, for example:

```
package com.javacodegeeks.advanced.generics;

public class GenericClassImplementingGenericInterface< T >
        implements GenericInterfaceOneType< T > {
    @Override
    public void performAction( final T action ) {
        // Implementation here
    }
}
```

It is a very convenient technique which allows classes to impose additional bounds on generic type still conforming the interface (or parent class) contract, as we will see in section wildcards and bounded types.

4.4 Generics and methods

We have already seen a couple of generic methods in the previous sections while discussing classes and interfaces. However, there is more to say about them. Methods could use generic types as part of arguments declaration or return type declaration. For example:

```
public< T, R > R performAction( final T action ) {
    final R result = ...;
    // Implementation here
    return result;
}
```

There are no restrictions on which methods can use generic types, they could be concrete, **abstract**, **static** or **final**. Here is a couple of examples:

```
protected abstract< T, R > R performAction( final T action );

static< T, R > R performActionOn( final Collection< T > action ) {
    final R result = ...;
```

```
    // Implementation here
    return result;
}
```

If methods are declared (or defined) as part of generic interface or class, they may (or may not) use the generic types of their owner. They may define own generic types or mix them with the ones from their class or interface declaration. For example:

```
package com.javacodegeeks.advanced.generics;

public class GenericMethods< T > {
    public< R > R performAction( final T action ) {
        final R result = ...;
        // Implementation here
        return result;
    }

    public< U, R > R performAnotherAction( final U action ) {
        final R result = ...;
        // Implementation here
        return result;
    }
}
```

Class constructors are also considered to be kind of initialization methods, and as such, may use the generic types declared by their class, declare own generic types or just mix both (however they cannot return values so the return type parameterization is not applicable to constructors), for example:

```
public class GenericMethods< T > {
    public GenericMethods( final T initialAction ) {
        // Implementation here
    }

    public< J > GenericMethods( final T initialAction, final J nextAction ) {
        // Implementation here
    }
}
```

It looks very easy and simple, and it surely is. However, there are some restrictions and side-effects caused by the way generics are implemented in Java language and the next section is going to address that.

4.5 Limitation of generics

Being one of the brightest features of the language, generics unfortunately have some limitations, mainly caused by the fact that they were introduced quite late into already mature language. Most likely, more thorough implementation required significantly more time and resources so the trade-offs had been made in order to have generics delivered in a timely manner.

Firstly, primitive types (like `int`, `long`, `byte`, ...) are not allowed to be used in generics. It means whenever you need to parameterize your generic type with a primitive one, the respective class wrapper (`Integer`, `Long`, `Byte`, ...) has to be used instead.

```
final List< Long > longs = new ArrayList<>();
final Set< Integer > integers = new HashSet<>();
```

Not only that, because of necessity to use class wrappers in generics, it causes implicit boxing and unboxing of primitive values (this topic will be covered in details in the **part 7** of the tutorial, **General programming guidelines**), for example:

```
final List< Long > longs = new ArrayList<>();
longs.add( 0L ); // 'long' is boxed to 'Long'

long value = longs.get( 0 ); // 'Long' is unboxed to 'long'
// Do something with value
```

But primitive types are just one of generics pitfalls. Another one, more obscure, is type erasure. It is important to know that generics exist only at compile time: the Java compiler uses a complicated set of rules to enforce type safety with respect to generics and their type parameters usage, however the produced JVM bytecode has all concrete types erased (and replaced with the `Object` class). It could come as a surprise first that the following code does not compile:

```
void sort( Collection< String > strings ) {
    // Some implementation over strings heres
}

void sort( Collection< Number > numbers ) {
    // Some implementation over numbers here
}
```

From the developer's standpoint, it is a perfectly valid code, however because of type erasure, those two methods are narrowed down to the same signature and it leads to compilation error (with a weird message like **"Erasure of method sort(Collection<String>) is the same as another method ..."**):

```
void sort( Collection strings )
void sort( Collection numbers )
```

Another disadvantage caused by type erasure come from the fact that it is not possible to use generics' type parameters in any meaningful way, for example to create new instances of the type, or get the concrete class of the type parameter or use it in the `instanceof` operator. The examples shown below do no pass compilation phase:

```
public< T > void action( final T action ) {
    if( action instanceof T ) {
        // Do something here
    }
}

public< T > void action( final T action ) {
    if( T.class.isAssignableFrom( Number.class )  ) {
     // Do something here
    }
}
```

And lastly, it is also not possible to create the array instances using generics' type parameters. For example, the following code does not compile (this time with a clean error message **"Cannot create a generic array of T"**):

```
public< T > void performAction( final T action ) {
    T[] actions = new T[ 0 ];
}
```

Despite all these limitations, generics are still extremely useful and bring a lot of value. In the section Accessing generic type parameters we are going to take a look on the several ways to overcome some of the constraints imposed by generics implementation in Java language.

4.6 Generics, wildcards and bounded types

So far we have seen the examples using generics with unbounded type parameters. The extremely powerful ability of generics is imposing the constraints (or bounds) on the type they are parameterized with using the `extends` and `super` keywords.

The `extends` keyword restricts the type parameter to be a subclass of some other class or to implement one or more interfaces. For example:

```
public< T extends InputStream > void read( final T stream ) {
    // Some implementation here
}
```

The type parameter `T` in the `read` method declaration is required to be a subclass of the `InputStream` class. The same keyword is used to restrict interface implementations. For example:

```
public< T extends Serializable > void store( final T object ) {
    // Some implementation here
}
```

Method store requires its type parameter `T` to implement the `Serializable` interface in order for the method to perform the desired action. It is also possible to use other type parameter as a bound for `extends` keyword, for example:

```
public< T, J extends T > void action( final T initial, final J next ) {
     // Some implementation here
}
```

The bounds are not limited to single constraints and could be combined using the `&` operator. There could be multiple interfaces specified but only single class is allowed. The combination of class and interfaces is also possible, with a couple of examples show below:

```
public< T extends InputStream &amp; Serializable > void storeToRead( final T stream ) {
    // Some implementation here
}
public< T extends Serializable &amp; Externalizable &amp; Cloneable > void persist(
        final T object ) {
    // Some implementation here
}
```

Before discussing the `super` keyword, we need to get familiarized with the concepts of wildcards. If the type parameter is not of the interest of the generic class, interface or method, it could be replaced by the ? wildcard. For example:

```
public void store( final Collection< ? extends Serializable > objects ) {
    // Some implementation here
}
```

The method `store` does not really care what type parameters it is being called with, the only thing it needs to ensure that every type implements `Serializable` interface. Or, if this is not of any importance, the wildcard without bounds could be used instead:

```
public void store( final Collection< ? > objects ) {
    // Some implementation here
}
```

In contrast to `extends`, the `super` keyword restricts the type parameter to be a superclass of some other class. For example:

```
public void interate( final Collection< ? super Integer > objects ) {
    // Some implementation here
}
```

By using upper and lower type bounds (with `extends` and `super`) along with type wildcards, the generics provide a way to fine-tune the type parameter requirements or, is some cases, completely omit them, still preserving the generics type-oriented semantic.

4.7 Generics and type inference

When generics found their way into the Java language, they blew up the amount of the code developers had to write in order to satisfy the language syntax rules. For example:

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

```
for( final Map.Entry< String, Collection< String > > entry: map.entrySet() ) {
    // Some implementation here
}
```

The Java 7 release somewhat addressed this problem by making changes in the compiler and introducing the new diamond operator <>. For example:

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

The compiler is able to infer the generics type parameters from the left side and allows omitting them in the right side of the expression. It was a significant progress towards making generics syntax less verbose, however the abilities of the compiler to infer generics type parameters were quite limited. For example, the following code does not compile in Java 7:

```
public static < T > void performAction( final Collection< T > actions,
        final Collection< T > defaults ) {
    // Some implementation here
}

final Collection< String > strings = new ArrayList<>();
performAction( strings, Collections.emptyList() );
```

The Java 7 compiler cannot infer the type parameter for the `Collections.emptyList()` call and as such requires it to be passed explicitly:

```
performAction( strings, Collections.< String >emptyList() );
```

Luckily, the Java 8 release brings more enhancements into the compiler and, particularly, into the type inference for generics so the code snippet shown above compiles successfully, saving the developers from unnecessary typing.

4.8 Generics and annotations

Although we are going to discuss the annotations in the next part of the tutorial, it is worth mentioning that in the pre-Java 8 era the generics were not allowed to have annotations associated with their type parameters. But Java 8 changed that and now it becomes possible to annotate generics type parameters at the places they are declared or used. For example, here is how the generic method could be declared and its type parameter is adorned with annotations:

```
public< @Actionable T > void performAction( final T action ) {
    // Some implementation here
}
```

Or just another example of applying the annotation when generic type is being used:

```
final Collection< @NotEmpty String > strings = new ArrayList<>();
// Some implementation here
```

In the **part 4** of the tutorial, **How and when to use Enums and Annotations**, we are going to take a look on a couple of examples how the annotations could be used in order to associate some metadata with the generics type parameters. This section just gives you the feeling that it is possible to enrich generics with annotations.

4.9 Accessing generic type parameters

As you already know from the section Limitation of generics, it is not possible to get the class of the generic type parameter. One simple trick to work-around that is to require additional argument to be passed, `Class< T >`, in places where it is necessary to know the class of the type parameter `T`. For example:

```
public< T > void performAction( final T action, final Class< T > clazz ) {
    // Some implementation here
}
```

It might blow the amount of arguments required by the methods but with careful design it is not as bad as it looks at the first glance.

Another interesting use case which often comes up while working with generics in Java is to determine the concrete class of the type which generic instance has been parameterized with. It is not as straightforward and requires Java reflection API to be involved. We will take a look on complete example in the **part 11** of the tutorial, **Reflection and dynamic languages support** but for now just mention that the `ParameterizedType` instance is the central point to do the reflection over generics.

4.10 When to use generics

Despite all the limitations, the value which generics add to the Java language is just enormous. Nowadays it is hard to imagine that there was a time when Java had no generics support. Generics should be used instead of raw types (`Collection< T >` instead of `Collection`, `Callable< T >` instead of `Callable`, ...) or `Object` to guarantee type safety, define clear type constraints on the contracts and algorithms, and significantly ease the code maintenance and refactoring.

However, please be aware of the limitations of the current implementation of generics in Java, type erasure and the famous implicit boxing and unboxing for primitive types. Generics are not a silver bullet solving all the problems you may encounter and nothing could replace careful design and thoughtful thinking.

It would be a good idea to look on some real examples and get a feeling how generics make Java developer's life easier.

Example 1: Let us consider the typical example of the method which performs actions against the instance of a class which implements some interface (say, `Serializable`) and returns back the modified instance of this class.

```
class SomeClass implements Serializable {
}
```

Without using generics, the solution may look like this:

```
public Serializable performAction( final Serializable instance ) {
    // Do something here
    return instance;
}

final SomeClass instance = new SomeClass();
// Please notice a necessary type cast required
final SomeClass modifiedInstance = ( SomeClass )performAction( instance );
```

Let us see how generics improve this solution:

```
public< T extends Serializable > T performAction( final T instance ) {
    // Do something here
    return instance;
}

final SomeClass instance = new SomeClass();
final SomeClass modifiedInstance = performAction( instance );
```

The ugly type cast has gone away as compiler is able to infer the right types and prove that those types are used correctly.

Example 2: A bit more complicated example of the method which requires the instance of the class to implement two interfaces (say, `Serializable` and `Runnable`).

```
class SomeClass implements Serializable, Runnable {
    @Override
    public void run() {
```

```
        // Some implementation
    }
}
```

Without using generics, the straightforward solution is to introduce intermediate interface (or use the pure `Object` as a last resort), for example:

```
// The class itself should be modified to use the intermediate interface
// instead of direct implementations
class SomeClass implements SerializableAndRunnable {
    @Override
    public void run() {
        // Some implementation
    }
}

public void performAction( final SerializableAndRunnable instance ) {
    // Do something here
}
```

Although it is a valid solution, it does not look as the best option and with the growing number of interfaces it could get really nasty and unmanageable. Let us see how generics can help here:

```
public< T extends Serializable &amp;amp; Runnable > void performAction( final T instance ) {
    // Do something here
}
```

Very clear and concise piece of code, no intermediate interface or other tricks are required.

The universe of examples where generics make code readable and straightforward is really endless. In the next parts of the tutorial generics will be often used to demonstrate other features of the Java language.

4.11 Download the Source Code

- This was a lesson on How to design Classes and Interfaces. You may download the source code here: advanced-java-part-4

4.12 What's next

In this section we have covered one of very distinguishing features of Java language called generics. We have seen how generics make you code type-safe and concise by checking that the right types (with bounds) are being used everywhere. We also looked through some of the generics limitations and the ways to overcome them. In the next section we are going to discuss enumerations and annotations.

Chapter 5

How and when to use Enums and Annotations

5.1 Introduction

In this part of the tutorial we are going to cover yet another two great features introduced into the language as part of Java 5 release along with generics: enums (or enumerations) and annotations. Enums could be treated as a special type of classes and annotations as a special type of interfaces.

The idea of enums is simple, but quite handy: it represents a fixed, constant set of values. What it means in practice is that enums are often used to design the concepts which have a constant set of possible states. For example, the days of week are a great example of the enums: they are limited to Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday.

From the other side, annotations are a special kind of metadata which could be associated with different elements and constructs of the Java language. Interestingly, annotations have contributed a lot into the elimination of boilerplate XML descriptors used in Java ecosystem mostly everywhere. They introduced the new, type-safe and robust way of configuration and customization techniques.

5.2 Enums as special classes

Before enums had been introduced into the Java language, the regular way to model the set of fixed values in Java was just by declaring a number of constants. For example:

```
public class DaysOfTheWeekConstants {
    public static final int MONDAY = 0;
    public static final int TUESDAY = 1;
    public static final int WEDNESDAY = 2;
    public static final int THURSDAY = 3;
    public static final int FRIDAY = 4;
    public static final int SATURDAY = 5;
    public static final int SUNDAY = 6;
}
```

Although this approach kind of works, it is far from being the ideal solution. Primarily, because the constants themselves are just values of type `int` and every place in the code where those constants are expected (instead of arbitrary `int` values) should be explicitly documented and asserted all the time. Semantically, it is not a type-safe representation of the concept as the following method demonstrates.

```
public boolean isWeekend( int day ) {
    return( day == SATURDAY || day == SUNDAY );
}
```

From logical point of view, the day argument should have one of the values declared in the `DaysOfTheWeekConstants` class. However, it is not possible to guess that without additional documentation being written (and read afterwards by someone). For the Java compiler the call like `isWeekend *(100)*` looks absolutely correct and raises no concerns.

Here the enums come to the rescue. Enums allow to replace constants with the typed values and to use those types everywhere. Let us rewrite the solution above using enums.

```
public enum DaysOfTheWeek {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

What changed is that the `class` becomes `enum` and the possible values are listed in the enum definition. The distinguishing part however is that every single value is the instance of the enum class it is being declared at (in our example, `DaysOfTheWeek`). As such, whenever enum are being used, the Java compiler is able to do type checking. For example:

```
public boolean isWeekend( DaysOfTheWeek day ) {
    return( day == SATURDAY || day == SUNDAY );
}
```

Please notice that the usage of the uppercase naming scheme in enums is just a convention, nothing really prevents you from not doing that.

5.3 Enums and instance fields

Enums are specialized classes and as such are extensible. It means they can have instance fields, constructors and methods (although the only limitations are that the default no-args constructor cannot be declared and all constructors must be `private`). Let us add the property `isWeekend` to every day of the week using the instance field and constructor.

```
public enum DaysOfTheWeekFields {
    MONDAY( false ),
    TUESDAY( false ),
    WEDNESDAY( false ),
    THURSDAY( false ),
    FRIDAY( false ),
    SATURDAY( true ),
    SUNDAY( true );

    private final boolean isWeekend;

    private DaysOfTheWeekFields( final boolean isWeekend ) {
        this.isWeekend = isWeekend;
    }

    public boolean isWeekend() {
        return isWeekend;
    }
}
```

As we can see, the values of the enums are just constructor calls with the simplification that the `new` keyword is not required. The `isWeekend()` property could be used to detect if the value represents the week day or week-end. For example:

```
public boolean isWeekend( DaysOfTheWeek day ) {
    return day.isWeekend();
}
```

Instance fields are an extremely useful capability of the enums in Java. They are used very often to associate some additional details with each value, using regular class declaration rules.

5.4 Enums and interfaces

Another interesting feature, which yet one more time confirms that enums are just specialized classes, is that they can implement interfaces (however enums cannot extend any other classes for the reasons explained later in the Enums and generics section). For example, let us introduce the interface `DayOfWeek`.

```
interface DayOfWeek {
    boolean isWeekend();
}
```

And rewrite the example from the previous section using interface implementation instead of regular instance fields.

```
public enum DaysOfTheWeekInterfaces implements DayOfWeek {
    MONDAY() {
        @Override
        public boolean isWeekend() {
            return false;
        }
    },
    TUESDAY() {
        @Override
        public boolean isWeekend() {
            return false;
        }
    },
    WEDNESDAY() {
        @Override
        public boolean isWeekend() {
            return false;
        }
    },
    THURSDAY() {
        @Override
        public boolean isWeekend() {
            return false;
        }
    },
    FRIDAY() {
        @Override
        public boolean isWeekend() {
            return false;
        }
    },
    SATURDAY() {
        @Override
        public boolean isWeekend() {
            return true;
        }
    },
    SUNDAY() {
        @Override
        public boolean isWeekend() {
            return true;
        }
    };
}
```

The way we have implemented the interface is a bit verbose, however it is certainly possible to make it better by combining instance fields and interfaces together. For example:

```
public enum DaysOfTheWeekFieldsInterfaces implements DayOfWeek {
    MONDAY( false ),
    TUESDAY( false ),
    WEDNESDAY( false ),
    THURSDAY( false ),
    FRIDAY( false ),
    SATURDAY( true ),
    SUNDAY( true );

    private final boolean isWeekend;

    private DaysOfTheWeekFieldsInterfaces( final boolean isWeekend ) {
        this.isWeekend = isWeekend;
    }

    @Override
    public boolean isWeekend() {
        return isWeekend;
    }
}
```

By supporting instance fields and interfaces, enums can be used in a more object-oriented way, bringing some level of abstraction to rely upon.

5.5 Enums and generics

Although it is not visible from a first glance, there is a relation between enums and generics in Java. Every single enum in Java is automatically inherited from the generic `Enum< T >` class, where `T` is the enum type itself. The Java compiler does this transformation on behalf of the developer at compile time, expanding enum declaration `public enum DaysOfTheWeek` to something like this:

```
public class DaysOfTheWeek extends Enum< DaysOfTheWeek > {
    // Other declarations here
}
```

It also explains why enums can implement interfaces but cannot extend other classes: they implicitly extend `Enum< T >` and as we know from the **part 2** of the tutorial, **Using methods common to all objects**, Java does not support multiple inheritance.

The fact that every enum extends `Enum< T >` allows to define generic classes, interfaces and methods which expect the instances of enum types as arguments or type parameters. For example:

```
public< T extends Enum < ? > > void performAction( final T instance ) {
    // Perform some action here
}
```

In the method declaration above, the type `T` is constrained to be the instance of any enum and Java compiler will verify that.

5.6 Convenient Enums methods

The base `Enum< T >` class provides a couple of helpful methods which are automatically inherited by every enum instance.

Table 5.1: datasheet

Table 5.1: (continued)

Method	Description
`String name()`	Returns the name of this enum constant, exactly as declared in its enum declaration.
`int ordinal()`	Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).

Additionally, Java compiler automatically generates two more helpful `static `methods for every enum type it encounters (let us refer to the particular enum type as **T**).

Table 5.2: datasheet

Method	Description
`T[] values()`	Returns the all declared enum constants for the enum `T`.
`T valueOf(String name)`	Returns the enum constant `T` with the specified name.

Because of the presence of these methods and hard compiler work, there is one more benefit of using enums in your code: they can be used in `switch/case` statements. For example:

```
public void performAction( DaysOfTheWeek instance ) {
    switch( instance ) {
        case MONDAY:
            // Do something
            break;

        case TUESDAY:
            // Do something
            break;

        // Other enum constants here
    }
}
```

5.7 Specialized Collections: EnumSet and EnumMap

Instances of enums, as all other classes, could be used with the standard Java collection library. However, certain collection types have been optimized for enums specifically and are recommended in most cases to be used instead of general-purpose counterparts.

We are going to look on two specialized collection types: `EnumSet< T >` and `EnumMap< T, ?>`. Both are very easy to use and we are going to start with the `EnumSet< T >`.

The `EnumSet< T >` is the regular set optimized to store enums effectively. Interestingly, `EnumSet< T >` cannot be instantiated using constructors and provides a lot of helpful factory methods instead (we have covered factory pattern in the **part 1** of the tutorial, **How to create and destroy objects**).

For example, the `allOf` factory method creates the instance of the `EnumSet< T >` containing all enum constants of the enum type in question:

```
final Set< DaysOfTheWeek > enumSetAll = EnumSet.allOf( DaysOfTheWeek.class );
```

Consequently, the `noneOf` factory method creates the instance of an empty `EnumSet< T >` for the enum type in question:

```
final Set< DaysOfTheWeek > enumSetNone = EnumSet.noneOf( DaysOfTheWeek.class );
```

It is also possible to specify which enum constants of the enum type in question should be included into the `EnumSet< T >`, using the `of` factory method:

```
final Set< DaysOfTheWeek > enumSetSome = EnumSet.of(
    DaysOfTheWeek.SUNDAY,
    DaysOfTheWeek.SATURDAY
);
```

The `EnumMap< T, ?>` is very close to the regular map with the difference that its keys could be the enum constants of the enum type in question. For example:

```
final Map< DaysOfTheWeek, String > enumMap = new EnumMap<>( DaysOfTheWeek.class );
enumMap.put( DaysOfTheWeek.MONDAY, "Lundi" );
enumMap.put( DaysOfTheWeek.TUESDAY, "Mardi" );
```

Please notice that, as most collection implementations, `EnumSet< T >` and `EnumMap< T, ?>` are not thread-safe and cannot be used as-is in multithreaded environment (we are going to discuss thread-safety and synchronization in the **part 9** of the tutorial, **Concurrency best practices**).

5.8 When to use enums

Since Java 5 release enums are the only preferred and recommended way to represent and dial with the fixed set of constants. Not only they are strongly-typed, they are extensible and supported by any modern library or framework.

5.9 Annotations as special interfaces

As we mentioned before, annotations are the syntactic sugar used to associate the metadata with different elements of Java language.

Annotations by themselves do not have any direct effect on the element they are annotating. However, depending on the annotations and the way they are defined, they may be used by Java compiler (the great example of that is the `@Override` annotation which we have seen a lot in the **part 3** of the tutorial, **How to design Classes and Interfaces**), by annotation processors (more details to come in the Annotation processors section) and by the code at runtime using reflection and other introspection techniques (more about that in the **part 11** of the tutorial, **Reflection and dynamic languages support**).

Let us take a look at the simplest annotation declaration possible:

```
public @interface SimpleAnnotation {
}

The @interface keyword introduces new annotation type. That is why annotations could be ↩
    treated as specialized interfaces. Annotations may declare the attributes with or ↩
    without default values, for example:
public @interface SimpleAnnotationWithAttributes {
    String name();
    int order() default 0;
}
```

If an annotation declares an attribute without a default value, it should be provided in all places the annotation is being applied. For example:

```
@SimpleAnnotationWithAttributes( name = "new annotation" )
```

By convention, if the annotation has an attribute with the name `value` and it is the only one which is required to be specified, the name of the attribute could be omitted, for example:

```
public @interface SimpleAnnotationWithValue {
    String value();
}

It could be used like this:

@SimpleAnnotationWithValue( "new annotation" )
```

There are a couple of limitations which in certain use cases make working with annotations not very convenient. Firstly, annotations do not support any kind of inheritance: one annotation cannot extend another annotation. Secondly, it is not possible to create an instance of annotation programmatically using the `new` operator (we are going to take a look on some workarounds to that in the **part 11** of the tutorial, **Reflection and dynamic languages support**). And thirdly, annotations can declare only attributes of primitive types, `String` or `Class< ?>` types and arrays of those. No methods or constructors are allowed to be declared in the annotations.

5.10 Annotations and retention policy

Each annotation has the very important characteristic called **retention policy** which is an enumeration (of type RetentionPolicy) with the set of policies on how to retain annotations. It could be set to one of the following values.

Table 5.3: datasheet

Policy	Description
CLASS	Annotations are to be recorded in the class file by the compiler but need not be retained by the VM at run time
RUNTIME	Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively.
SOURCE	Annotations are to be discarded by the compiler.

Retention policy has a crucial effect on when the annotation will be available for processing. The retention policy could be set using `@Retention` annotation. For example:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention( RetentionPolicy.RUNTIME )
public @interface AnnotationWithRetention {
}
```

Setting annotation retention policy to `RUNTIME` will guarantee its presence in the compilation process and in the running application.

5.11 Annotations and element types

Another characteristic which each annotation must have is the element types it could be applied to. Similarly to the retention policy, it is defined as enumeration (ElementType) with the set of possible element types.

Table 5.4: datasheet

Table 5.4: (continued)

Element Type	Description
ANNOTATION_TYPE	Annotation type declaration
CONSTRUCTOR	Constructor declaration
FIELD	Field declaration (includes enum constants)
LOCAL_VARIABLE	Local variable declaration
METHOD	Method declaration
PACKAGE	Package declaration
PARAMETER	Parameter declaration
TYPE	Class, interface (including annotation type), or enum declaration

Additionally to the ones described above, Java 8 introduces two new element types the annotations can be applied to.

Table 5.5: datasheet

Element Type	Description
TYPE_PARAMETER	Type parameter declaration
TYPE_USE	Use of a type

In contrast to the retention policy, an annotation may declare multiple element types it can be associated with, using the `@Target` annotation. For example:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target( { ElementType.FIELD, ElementType.METHOD } )
public @interface AnnotationWithTarget {
}
```

Mostly all annotations you are going to create should have both retention policy and element types specified in order to be useful.

5.12 Annotations and inheritance

The important relation exists between declaring annotations and inheritance in Java. By default, the subclasses do not inherit the annotation declared on the parent class. However, there is a way to propagate particular annotations throughout the class hierarchy using the `@Inherited` annotation. For example:

```
@Target( { ElementType.TYPE } )
@Retention( RetentionPolicy.RUNTIME )
@Inherited
@interface InheritableAnnotation {
}

@InheritableAnnotation
public class Parent {
}

public class Child extends Parent {
}
```

In this example, the `@InheritableAnnotation` annotation declared on the `Parent` class will be inherited by the `Child` class as well.

5.13 Repeatable annotations

In pre-Java 8 era there was another limitation related to the annotations which was not discussed yet: the same annotation could appear only once at the same place, it cannot be repeated multiple times. Java 8 eased this restriction by providing support for repeatable annotations. For example:

```
@Target( ElementType.METHOD )
@Retention( RetentionPolicy.RUNTIME )
public @interface RepeatableAnnotations {
    RepeatableAnnotation[] value();
}

@Target( ElementType.METHOD )
@Retention( RetentionPolicy.RUNTIME )
@Repeatable( RepeatableAnnotations.class )
public @interface RepeatableAnnotation {
    String value();
};
@RepeatableAnnotation( "repeatition 1" )
@RepeatableAnnotation( "repeatition 2" )
public void performAction() {
    // Some code here
}
```

Although in Java 8 the repeatable annotations feature requires a bit of work to be done in order to allow your annotation to be repeatable (using `@Repeatable`), the final result is worth it: more clean and compact annotated code.

5.14 Annotation processors

The Java compiler supports a special kind of plugins called annotation processors (using the `-processor` command line argument) which could process the annotations during the compilation phase. Annotation processors can analyze the annotations usage (perform static code analysis), create additional Java source files or resources (which in turn could be compiled and processed) or mutate the annotated code.

The **retention policy** (see please Annotations and retention policy) plays a key role by instructing the compiler which annotations should be available for processing by annotation processors.

Annotation processors are widely used, however to write one it requires some knowledge of how Java compiler works and the compilation process itself.

5.15 Annotations and configuration over convention

Convention over configuration is a software design paradigm which aims to simplify the development process when a set of simple rules (or conventions) is being followed by the developers. For example, some **MVC** (model-view-controller) frameworks follow the convention to place controllers in the ***controller*** folder (or package). Another example is the **ORM** (object-relational mappers) frameworks which often follow the convention to look up classes in ***model*** folder (or package) and derive the relation table name from the respective class.

On the other side, annotations open the way for a different design paradigm which is based on explicit configuration. Considering the examples above, the `@Controller` annotation may explicitly mark any class as controller and `@Entity` may refer to relational database table. The benefits also come from the facts that annotations are extensible, may have additional attributes and are restricted to particular element types. Improper use of annotations is enforced by the Java compiler and reveals the misconfiguration issues very early (on the compilation phase).

5.16 When to use annotations

Annotations are literally everywhere: the Java standard library has a lot of them, mostly every Java specification includes the annotations as well. Whenever you need to associate an additional metadata with your code, annotations are straightforward and easy way to do so.

Interestingly, there is an ongoing effort in the Java community to develop common semantic concepts and standardize the annotations across several Java technologies (for more information, please take a look on JSR-250 specification). At the moment, following annotations are included with the standard Java library.

Table 5.6: datasheet

Annotation	Description
`@Deprecated`	Indicates that the marked element is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with this annotation.
`@Override`	Hints the compiler that the element is meant to override an element declared in a superclass.
`@SuppressWarnings`	Instructs the compiler to suppress specific warnings that it would otherwise generate.
`@SafeVarargs`	When applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its varargs parameter. When this annotation type is used, unchecked warnings relating to varargs usage are supressed (more details about varargs will be covered in the **part 6** of the tutorial, **How to write methods efficiently**).
`@Retention`	Specifies how the marked annotation is retained.
`@Target`	Specifies what kind of Java elements the marked annotation can be applied to.
`@Documented`	Indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool (by default, annotations are not included in Javadoc).
`@Inherited`	Indicates that the annotation type can be inherited from the super class (for more details please refer to Annotations and inheritance section).

And the Java 8 release adds a couple of new annotations as well.

Table 5.7: datasheet

Annotation	Description
`@FunctionalInterface`	Indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification (more details about functional interfaces are covered in the **part 3** of the tutorial, **How to design Classes and Interfaces**).
`@Repeatable`	Indicates that the marked annotation can be applied more than once to the same declaration or type use (for more details please refer to Repeatable annotations section).

5.17 Download the Source Code

This was a lesson on How to design Classes and Interfaces. You may download the source code here: advanced-java-part-5

5.18 What's next

In this section we have covered enums (or enumerations) which are used to represent the fixed set of constant values, and annotations which decorate elements of Java code with metadata. Although somewhat unrelated, both concepts are very widely used in the Java. Despite the fact that in the next part of the tutorial we are going to look on how to write methods efficiently, annotations will be often the part of the mostly every discussion.

Chapter 6

How to write methods efficiently

6.1 Introduction

In this section of the tutorial we are going to spend some time discussing different aspects related to designing and implementing methods in Java. As we have seen in previous parts of the tutorial, it is very easy to write methods in Java, however there are many things which could make your methods more readable and efficient.

6.2 Method signatures

As we already know very well, Java is an object-oriented language. As such, every method in Java belongs to some class instance (or a class itself in case of `static` methods), has visibility (or accessibility) rules, may be declared `abstract` or `final`, and so on. However, arguably the most important part of the method is its signature: the return type and arguments, plus the list of checked exceptions which method implementation may throw (but this part is used less and less often nowadays). Here is a small example to start with:

```
public static void main( String[] args ) {
    // Some implementation here
}
```

The method `main` accepts an array of strings as an only argument args and returns nothing. It would be very nice to keep all methods as simple as `main` is. But in reallity, a method signature may become unreadable. Let us take a look at the following example:

```
public void setTitleVisible( int lenght, String title, boolean visible ) {
    // Some implementation here
}
```

The first thing to notice here is that by convention, the method names in Java are written in camel case, for example: `setTitleVisible`. The name is well chosen and tries to describe what the method is supposed to do.

Secondly, the name of each argument says (or at least hints) about its purpose. It is very important to find the right, explanatory names for the method arguments, instead of `int i`, `String s`, `boolean f` (in very rare cases it makes sense however).

Thirdly, the method takes only three arguments. Although Java has a much higher limit for allowed number of arguments, it is highly recommended to keep this number below 6. Beyond this point method signature becomes hard to understand.

Since the Java 5 release, the methods can have variable list of arguments of the same type (called varargs) using a special syntax, for example:

```
public void find( String ... elements ) {
    // Some implementation here
}
```

Internally, the Java compiler converts varargs into an array of the respective type and that is how varargs can be accessed by the method implementation.

Interestingly, Java also allows to declare the **varargs** argument using generic type parameter. However, because the type of the argument is not known, the Java compiler wants to be sure that the **varargs** are used responsibly and advice the method to be `final` and annotated with the `@SafeVarargs` annotation (more details about annotations are covered in the **part 5** of the tutorial, **How and when to use Enums and Annotations**). For example:

```
@SafeVarargs
final public< T > void find( T ... elements ) {
    // Some implementation here
}
```

The other way around is by using the `@SuppressWarnings` annotation, for example:

```
@SuppressWarnings( "unchecked" )
public< T > void findSuppressed( T ... elements ) {
    // Some implementation here
}
```

The next example demonstrates the usage of the checked exceptions as part of the method signature. In recent years the checked exception has proved not to be as useful as they intended to be, causing more boilerplate code to be written than problems solved.

```
public void write( File file ) throws IOException {
    // Some implementation here
}
```

Last but not least, it is generally advised (but rarely used) to mark the method arguments as `final`. It helps to get rid of bad code practice when method arguments are reassigned with different values. Also, such method arguments could be used by anonymous classes (more details about anonymous classes are covered in **part 3** of the tutorial, **How to design Classes and Interfaces**), though Java 8 eased a bit this constraint by introducing effectively `final` variables.

6.3 Method body

Every method has its own implementation and purpose to exist. However, there are a couple of general guidelines which really help writing clean and understandable methods.

Probably the most important one is the single responsibility principle: try to implement the methods in such a way, that every single method does just one thing and does it well. Following this principle may blow up the number of class methods, so it is important to find the right balance.

Another important thing while coding and designing is to keep method implementations short (often just by following single responsibility principle you will get it for free). Short methods are easy to reason about, plus they usually fit into one screen so they could be understood by the reader of your code much faster.

The last (but not least) advice is related to using `return` statements. If a method returns some value, try to minimize the number of places where the `return` statement is being called (some people go even further and recommend to use just single return statement in all cases). More `return` statements method has, much harder it becomes to follow its logical flows and modify (or refactore) the implementation.

6.4 Method overloading

The technique of methods overloading is often used to provide the specialized version of the method for different argument types or combinations. Although the method name stays the same, the compiler picks the right alternative depending on the actual argument values at the invocation point (the best example of overloading is constructors in Java: the name is always the same but the set of arguments is different) or raises a compiler error if none found. For example:

```
public String numberToString( Long number ) {
    return Long.toString( number );
}

public String numberToString( BigDecimal number ) {
    return number.toString();
}
```

Method overloading is somewhat close to generics (more details about generics are covered in **part 4** of the tutorial, **How and when to use Generics**), however it is used in cases where the generic approach does not work well and each (or most) generic type arguments require their own specialized implementations. Nevertheless, combining both generics and overloading could be very powerful, but often not possible in Java, because of type erasure (for more details please refer to **part 4** of the tutorial, **How and when to use Generics**). Let us take a look on this example:

```
public< T extends Number > String numberToString( T number ) {
    return number.toString();
}

public String numberToString( BigDecimal number ) {
    return number.toPlainString();
}
```

Although this code snippet could be written without using generics, it is not importan for our demonstration purposes. The interesting part is that the method `numberToString` is overloaded with a specialized implementation for `BigDecimal` and a generic version is provided for all other numbers.

6.5 Method overriding

We have talked a lot about method overriding in **part 3** of the tutorial, **How to design Classes and Interfaces**. In this section, when we already know about method overloading, we are going to show off why using `@Override` annotation is so important. Our example will demonstrate the subtle difference between method overriding and overloading in the simple class hierarchy.

```
public class Parent {
    public Object toObject( Number number ) {
        return number.toString();
    }
}
```

The class Parent has just one method `toObject`. Let us subclass this class and try to come up with the method version to convert numbers to strings (instead of raw objects).

```
public class Child extends Parent {
    @Override
    public String toObject( Number number ) {
        return number.toString();
    }
}
```

Nevertheless the signature of the `toObject` method in the `Child` class is a little bit different (please see Covariant method return types for more details), it does override the one from the superclass and Java compiler has no complaints about that. Now, let us add another method to the `Child` class.

```
public class Child extends Parent {
    public String toObject( Double number ) {
        return number.toString();
    }
}
```

Again, there is just a subtle difference in the method signature (`Double` instead of `Number`) but in this case it is an overloaded version of the method, it does not override the parent one. That is when the help from Java compiler and `@Override` annotations pay off: annotating the method from last example with `@Override` raises the compiler error.

6.6 Inlining

Inlining is an optimization performed by the Java JIT (just-in-time) compiler in order to eliminate a particular method call and replace it directly with method implementation. The heuristics JIT compiler uses are depending on both how often a method is being invoked and also on how large it is. Methods that are too large cannot be inlined effectively. Inlining may provide significant performance improvements to your code and is yet another benefit of keeping methods short as we already discussed in the section Method body.

6.7 Recursion

Recursion in Java is a technique where a method calls itself while performing calculations. For example, let us take a look on the following example which sums the numbers of an array:

```
public int sum( int[] numbers ) {
    if( numbers.length == 0 ) {
        return 0;
    } if( numbers.length == 1 ) {
        return numbers[ 0 ];
    } else {
        return numbers[ 0 ] + sum( Arrays.copyOfRange( numbers, 1, numbers.length ) );
    }
}
```

It is a very ineffective implementation, however it demonstrates recursion well enough. There is one well-known issue with recursive methods: depending how deep the call chain is, they can blow up the stack and cause a `StackOverflowError` exception. But things are not as bad as they sound because there is a technique which could eliminate stack overflows called tail call optimization. This could be applied if the method is tail-recursive (tail-recursive methods are methods in which all recursive calls are tail calls). For example, let us rewrite the previous algorithm in a tail-recursive manner:

```
public int sum( int initial, int[] numbers ) {
    if( numbers.length == 0 ) {
        return initial;
    } if( numbers.length == 1 ) {
        return initial + numbers[ 0 ];
    } else {
        return sum( initial + numbers[ 0 ],
            Arrays.copyOfRange( numbers, 1, numbers.length ) );
    }
}
```

Unfortunately, at the moment the Java compiler (as well as JVM JIT compiler) does not support **tail call optimization** but still it is a very useful technique to know about and to consider whenever you are writing the recursive algorithms in Java.

6.8 Method References

Java 8 made a huge step forward by introducing functional concepts into the Java language. The foundation of that is treating the methods as data, a concept which was not supported by the language before (however, since Java 7, the JVM and the Java standard library have already some features to make it possible). Thankfully, with method references, it is now possible.

Table 6.1: datasheet

Type of the reference	Example
Reference to a static method	`SomeClass::staticMethodName`
Reference to an instance method of a particular object	`someInstance::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`SomeType::methodName`
Reference to a constructor	`SomeClass::new`

Let us take a look on a quick example on how methods could be passed around as arguments to other methods.

```
public class MethodReference {
    public static void println( String s ) {
        System.out.println( s );
    }

    public static void main( String[] args ) {
        final Collection< String > strings = Arrays.asList( "s1", "s2", "s3" );
        strings.stream().forEach( MethodReference::println );
    }
}
```

The last line of the `main` method uses the reference to `println` method to print each element from the collection of strings to the console and it is passed as an argument to another method, `forEach`.

6.9 Immutability

Immutability is taking a lot of attention these days and Java is not an exception. It is well-known that immutability is hard in Java but this does not mean it should be ignored.

In Java, immutability is all about changing internal state. As an example, let us take a look on the **JavaBeans** specification (http://docs.oracle.com/javase/tutorial/javabeans/). It states very clearly that setters may modify the state of the containing object and that is what every Java developer expects.

However, the alternative approach would be not to modify the state, but return a new one every time. It is not as scary as it sounds and the new Java 8 Date/Time API (developed under JSR 310: Date and Time API umbrella) is a great example of that. Let us take a look on the following code snippet:

```
final LocalDateTime now = LocalDateTime.now();
final LocalDateTime tomorrow = now.plusHours( 24 );

final LocalDateTime midnight = now
    .withHour( 0 )
    .withMinute( 0 )
    .withSecond( 0 )
    .withNano( 0 );
```

Every call to the `LocalDateTime` instance which needs to modify its state returns the new `LocalDateTime` instance and keeps the original one unchanged. It is a big shift in API design paradigm comparing to old `Calendar` and `Date` ones (which mildly speaking were not very pleasant to use and caused a lot of headaches).

6.10 Method Documentation

In Java, specifically if you are developing some kind of library or framework, all public methods should be documented using the **Javadoc** tool (http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html). Strictly speaking, nothing enforces you

to do that, but good documentation helps other developers to understand what a particular method is doing, what arguments it requires, which assumptions or constraints its implementation has, what types of exceptions and when could be raised and what the return value (if any) could be (plus many more things).

Let us take a look on the following example:

```
/**
 * The method parses the string argument as a signed decimal integer.
 * The characters in the string must all be decimal digits, except
 * that the first character may be a minus sign {@code '-'} or plus
 * sign {@code '+'}.
 *
 * <p>An exception of type {@code NumberFormatException} is thrown if
 * string is {@code null} or has length of zero.
 *
 * <p>Examples:
 * <blockquote><pre>
 * parse( "0" ) returns 0
 * parse( "+42") returns 42
 * parse( "-2" ) returns -2
 * parse( "string" ) throws a NumberFormatException
 * </pre></blockquote>
 *
 * @param str a {@code String} containing the {@code int} representation to be parsed
 * @return the integer value represented by the string
 * @exception NumberFormatException if the string does not contain a valid integer value
 */
public int parse( String str ) throws NumberFormatException {
    return Integer.parseInt( str );
}
```

It is quite a verbose documentation for such a simple method as `parse`, is but it shows up a couple of useful capabilities the **Javadoc** tool provides, including references to other classes, sample snippets and advanced formatting. Here is how this method documentation is reflected by Eclipse, one of the popular Java **IDEs**.

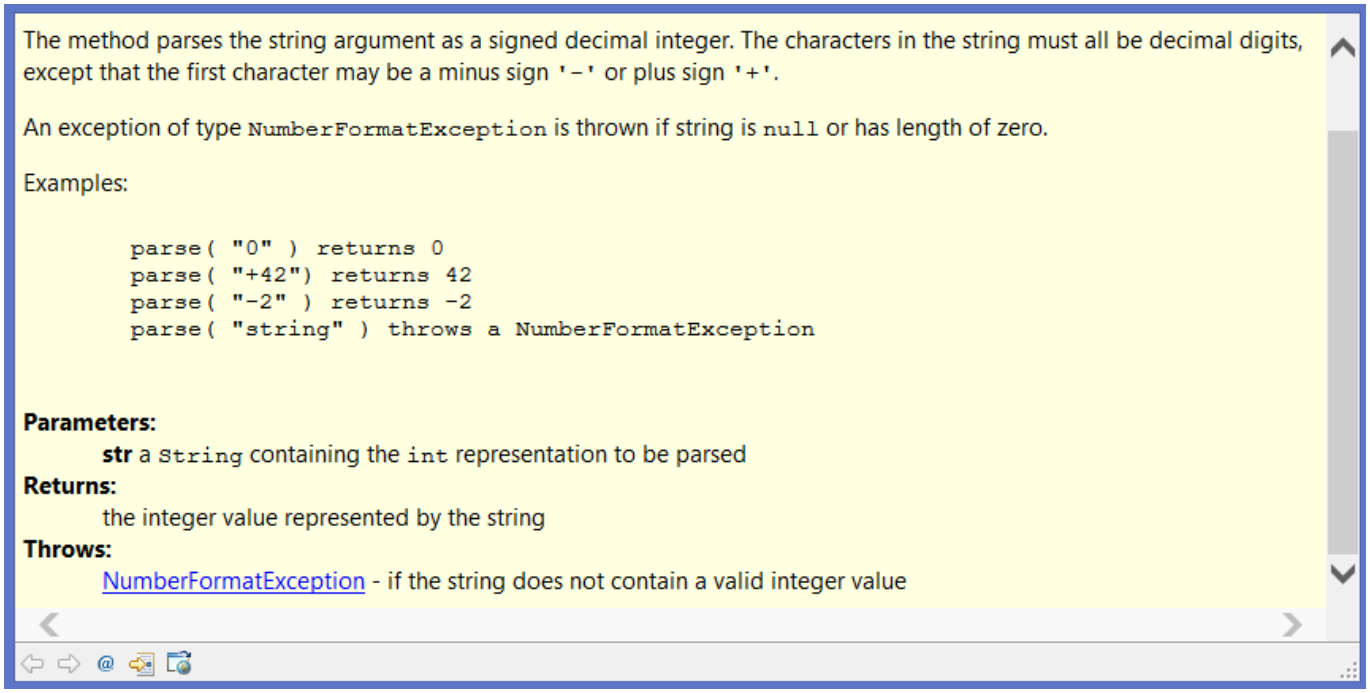

Figure 6.1: 6.Javadoc.Eclipse

Just by looking on the image above, any Java developer from junior to senior level can understand the purpose of the method and the proper way to use it.

6.11 Method Parameters and Return Values

Documenting your methods is a great thing, but unfortunately it does not prevent the use cases when a method is being called using incorrect or unexpected argument values. Because of that, as a rule of thumb all public methods should validate their arguments and should never believe that they are going to be specified with the correct values all the time (the pattern better known as sanity checks).

Returning back to our example from the previous section, the method `parse` should perform the validation of its only argument before doing anything with it:

```
public int parse( String str ) throws NumberFormatException {
    if( str == null ) {
        throw new IllegalArgumentException( "String should not be null" );
    }

    return Integer.parseInt( str );
}
```

Java has another option to perform validation and sanity checks using the `assert` statements. However, those could be turned off at runtime and may not be executed. It is preferred to always perform such checks and raise the relevant exceptions.

Even having the methods documented and validating their arguments, there are still couple of notes to mention related to the values they could return. Before Java 8, the simplest way for a method to say "I have no value to return at this time" was just by returning `null`. That is why Java is so infamous for `NullPointerException` exceptions. Java 8 tries to address this issue with introduction of `Optional < T >` class. Let us take a look on this example:

```
public< T > Optional< T > find( String id ) {
    // Some implementation here
}
```

`Optional < T >` provides a lot of useful methods and completely eliminates the need for the method to return `null` and pollute your code with `null` checks everywhere. The only exception probably is collections. Whenever method returns the collection, it is always better to return the empty one instead of `null` (and even `Optional < T >`), for example:

```
public&amp;lt; T &amp;gt; Collection&amp;lt; T &amp;gt; find( String id ) {
return Collections.emptyList();
}
```

6.12 Methods as API entry points

Even if you are just a developer building applications within your organization or a contributor to one of the popular Java framework or library, the design decisions you are taking play a very important role in a way how your code is going to be used.

While the API design guidelines are worth of several books, this part of the tutorial touches many of them (as methods become the API entry points) so a quick summary would be very helpful:

- Use meaningful names for methods and their arguments (Method signatures)
- Try to keep the number of arguments to be less than 6 (section Method signatures)
- Keep your methods short and readable (section Method body and Inlining)
- Always document your public methods, including preconditions and examples if it makes sense (section Method Documentation)

- Always perform argument validation and sanity checks (section Method Parameters and Return Values)
- Try to escape `null` as method return value (section Method Parameters and Return Values)
- Whenever it makes sense, try to design immutable methods (which do not affect the internal state, section Immutability)
- Use visibility and accessibility rules to hide the methods which should not be public (**part 3** of the tutorial, **How to design Classes and Interfaces**)

6.13 Download the Source Code

This was a lesson on How to write methods efficiently. You may download the source code here: advanced-java-part-6

6.14 What's next

This part of the tutorial was talking a bit less about Java as a language, but more about how to use Java as a language effectively, in particular by writing readable, clean, documented and effective methods. In the next section we are going to continue with the same basic idea and discuss general programming guidelines which are intended to help you to be better Java developer.

Chapter 7

General programming guidelines

7.1 Introduction

In this part of the tutorial we are going to continue discussing general principles of good programming style and robust design in Java. Some of those principles we have already seen in the previous parts of the tutorial, however a lot of new practical advices will be introduced along the way aiming to improve your skills as a Java developer.

7.2 Variable scopes

In the **part 3** of the tutorial, **How to design Classes and Interfaces**, we have discussed how the visibility and accessibility could be applied to class and interface members, limiting their scope. However we have not discussed yet the local variables, which are used within method implementations.

In the Java language, every local variable, once declared, has a scope. The variable becomes visible from the place it is declared to the end of the method (or code block) it is declared in. As such, there is only one single rule to follow: declare the local variable as close to the place where it is used as possible. Let us take a look on some typical examples:

```
for( final Locale locale: Locale.getAvailableLocales() ) {
    // Some implementation here
}

try( final InputStream in = new FileInputStream( "file.txt" ) ) {
    // Some implementation here
}
```

In both code snippets the scope of the local variables is limited to the execution blocks they are declared in. Once the block ends, the local variable goes out of scope and is not visible anymore. It looks clean and concise however with the release of Java 8 and introduction of lambdas, many well-known idioms of using local variables are becoming obsolete. Let us rewrite the **`for-each`** loop from the previous example to use lambdas instead:

```
Arrays.stream( Locale.getAvailableLocales() ).forEach( ( locale ) -> {
        // Some implementation here
    }
);
```

The local variable became the argument of the function which itself is passed as the argument of the `forEach` method.

7.3 Class fields and local variables

Every method in Java belongs to some class (or some interface in case of Java 8 and the method is declared as `default`). As such, there is a probability of name conflicts between local variables used in method implementations and class members.

The Java compiler is able to pick the right variable from the scope although it might not be the one developer intended to use. The modern Java IDEs do a tremendous work in order to hint to the developers when such conflicts are taking place (warnings, highlightings, ...) but it is still better to think about that while developing. Let us take a look on this example:

```
public class LocalVariableAndClassMember {
     private long value;

     public long calculateValue( final long initial ) {
         long value = initial;

         value *= 10;
         value += value;

         return value;
     }
 }
```

The example looks quite simple however there is a catch. The method `calculateValue` introduces a local variable with name `value` and by doing that hides the class member with the same name. The line `08` should have summed the class member and the local variable but instead it does something very different. The correct version may look like that (using the keyword `this`):

```
public class LocalVariableAndClassMember {
     private long value;

     public long calculateValue( final long initial ) {
         long value = initial;

         value *= 10;
         value += this.value;

         return value;
     }
 }
```

Somewhat naive implementation but nevertheless it highlights the important issue which in some cases could take hours to debug and troubleshoot.

7.4 Method arguments and local variables

Another trap, which quite often inexperienced Java developers fall into, is using method arguments as local variables. Java allows to reassign non-`final` method arguments with a different value (however it has no effect on the original value whatsoever). For example:

```
public String sanitize( String str ) {
    if( !str.isEmpty() ) {
        str = str.trim();
    }

    str = str.toLowerCase();
    return str;
}
```

It is not a beautiful piece of code but it is good enough to reveal the problem: method argument `str` is reassigned to another value (and basically is used as a local variable). In all cases (with no exception) this pattern could and should be avoided (for example, by declaring method arguments as `final`). For example:

```
public String sanitize( final String str ) {
    String sanitized = str;
```

```
    if( !str.isEmpty() ) {
        sanitized = str.trim();
    }

    sanitized = sanitized.toLowerCase();
    return sanitized;
}
```

The code which does follow this simple rule is much easier to follow and reason about, even at the price of introducing local variables.

7.5 Boxing and unboxing

Boxing and unboxing are all the names of the same technique used in Java language to convert between primitive types (like `int`, `long`, `double`) to respective primitive type wrappers (like `Integer`, `Long`, `Double`). In the **part 4** of the tutorial, **How and when to use Generics**, we have already seen it in action while talking about primitive type wrappers as generic type parameters.

Although the Java compiler tries to do its best to hide those conversions by performing autoboxing, sometimes it make things worse and leads to unexpected results. Let us take a look on this example:

```
public static void calculate( final long value ) {
    // Some implementation here
}
```

```
final Long value = null;
 calculate( value );
```

The code snippet above compiles perfectly fine, however it throws `NullPointerException` on line `02` when the conversion between `Long` and `long` happens. The advice here would be to prefer using primitive type (however as we already know, it is not always possible).

7.6 Interfaces

In the **part 3** of the tutorial, **How to design Classes and Interfaces**, we have discussed interfaces and contract-based development, emphasizing a lot on the fact that interfaces should be preferred to concrete classes wherever possible. The intent of this section is to convince you one more time to consider interfaces first by showing off real examples.

Interfaces are not tied to any particular implementation (with default methods being an exception). They are just contracts and as such they provide a lot of freedom and flexibility in the way contracts could be fulfilled. This flexibility becomes increasingly important when the implementation involves external systems or services. Let us take a look on the following simple interface and its possible implementation:

```
public interface TimezoneService {
    TimeZone getTimeZone( final double lat, final double lon ) throws IOException;
}
```

```
public class TimezoneServiceImpl implements TimezoneService {
    @Override
    public TimeZone getTimeZone(final double lat, final double lon) throws IOException {
        final URL url = new URL( String.format(
            "http://api.geonames.org/timezone?lat=%.2f&amp;lng=%.2f&amp;username=demo",
            lat, lon
        ) );
        final HttpURLConnection connection = ( HttpURLConnection )url.openConnection();
```

```
        connection.setRequestMethod( "GET" );
        connection.setConnectTimeout( 1000 );
        connection.setReadTimeout( 1000 );
        connection.connect();

        int status = connection.getResponseCode();
        if (status == 200) {
            // Do something here
        }

        return TimeZone.getDefault();
    }
}
```

The code snippet above demonstrates a typical interface / implementation pattern. The implementation uses external HTTP service (http://api.geonames.org/) to retrieve the time zone for a particular location. However, because the contact is driven by the interface, it is very easy to introduce yet another implementation using, for example, database or even flat file. With that, interfaces greatly help to design testable code. For example, it is not always practical to call external service on each test run so it makes sense to provide the alternative, dummy implementation (also known as stub or mock) instead:

```
public class TimezoneServiceTestImpl implements TimezoneService {
    @Override
    public TimeZone getTimeZone(final double lat, final double lon) throws IOException {
        return TimeZone.getDefault();
    }
}
```

This implementation could be used in every place where `TimezoneService` interface is required, isolating the test scenario from dependency on external components.

Many excellent examples of appropriate use of the interfaces are encapsulated inside the Java standard collection library. `Collection`, `List`, `Set`, all those interfaces are backed by several implementations which could be replaced seamlessly and interchangeably when contracts are favored, for example:

```
public static< T > void print( final Collection< T > collection ) {
    for( final T element: collection ) {
        System.out.println( element );
    }
}
```

```
print( new HashSet< Object >( /* ... */ ) );
print( new ArrayList< Integer >( /* ... */ ) );
print( new TreeSet< String >( /* ... */ ) );
print( new Vector< Long >( /* ... */ ) );
```

7.7 Strings

Strings are one of the most widely used types in Java and, arguably, in most of the programming languages. The Java language simplifies a lot the routine operations over strings by natively supporting the concatenations and comparison. Additionally, the Java standard library provides many different classes to make strings operations efficient and that is what we are going to discuss in this section.

In Java, strings are immutable objects, represented in UTF-16 format. Every time you concatenate the strings (or perform any operation which modifies the original string) the new instance of the `String` class is created. Because of this fact, the concatenation operations may become very ineffective, causing the creation of many intermediate string instances (generally speaking, generating garbage).

But the Java standard library provides two very helpful classes which aim to facilitate string manipulations: `StringBuilder` and `StringBuffer` (the only difference between those is that `StringBuffer` is thread-safe while `StringBuilder` is not). Let us take a look on couple of examples using one of these classes:

```
final StringBuilder sb = new StringBuilder();

for( int i = 1; i <= 10; ++i ) {
    sb.append( " " );
    sb.append( i );
}

sb.deleteCharAt( 0 );
sb.insert( 0, "[" );
sb.replace( sb.length() - 3, sb.length(), "]" );
```

Though using the `StringBuilder` / `StringBuffer` is the recommended way to manipulate strings, it may look over-killing in simple scenario of concatenating two or three strings so the regular + operator could be used instead, for example:

```
String userId = "user:" + new Random().nextInt( 100 );
```

The often better alternative to straightforward concatenation is to use string formatting and Java standard library is also here to help by providing the static helper method `String.format`. It supports a rich set of format specifiers, including numbers, characters, dates/times, etc. (for the complete reference please visit the official documentation). Let us explore the power of formatting by example:

```
String.format( "%04d", 1 );                    -> 0001
String.format( "%.2f", 12.324234d );           -> 12.32
String.format( "%tR", new Date() );            -> 21:11
String.format( "%tF", new Date() );            -> 2014-11-11
String.format( "%d%%", 12 );                   -> 12%
```

The `String.format` method provides a clean and easy approach to construct strings from different data types. It worth to mention that some modern Java IDEs are able to analyze the format specification against the arguments passed to the `String.format` method and warn developers in case of any mismatches detected.

7.8 Naming conventions

Java as a language does not force the developers to strictly follow any naming conventions, however the community has developed a set of easy to follow rules which make the Java code looking uniformly across standard library and any other Java project in the wild.

- **package names** are typed in **lower case**: `org.junit`, `com.fasterxml.jackson`, `javax.json`
- **class, enum, interface** or **annotation names** are typed in **capitalized case**: `StringBuilder`, `Runnable`, `@Override`
- **method** or **field names** (except `static final`) are typed in **camel case**: `isEmpty`, `format`, `addAll`
- **static final field** or **enumeration constant names** are typed in **upper case**, s*eparated by underscore* **`_' :`** `LOG`, `MIN_RADIX`, `INSTANCE`
- **local variable** and **method arguments names** are typed in **camel case**: `str`, `newLength`, `minimumCapacity`
- **generic type parameter names** are usually represented as **one character in upper case**: `T`, `U`, `E`

By following these simple conventions the code you are writing will look concise and indistinguishable from any other library or framework, giving the impression it was authored by the same person (one of those rare cases when conventions are really working).

7.9 Standard Libraries

No matter what kind of Java projects you are working on, the Java standard libraries are your best friends. Yes, it is hard to disagree that they have some rough edges and strange design decisions, nevertheless in 99% it is a high quality code written by experts. It is worth learning.

Every Java release brings a lot of new features to existing libraries (with some possible deprecation of the old ones) as well as adds many new libraries. Java 5 brought new concurrency library reconciled under `java.util.concurrent` package. Java 6 delivered (a bit less known) scripting support (`javax.script` package) and the Java compiler API (under `javax.tools` package). Java 7 brought a lot of improvements into `java.util.concurrent`, introduced new I/O library under `java.nio.file` package and dynamic languages support with `java.lang.invoke` package. And finally, Java 8 delivered a long-awaited date/time API hosted under `java.time` package.

Java as a platform is evolving and it is extremely important to keep up with this evolution. Whenever you are considering to introduce third-party library or framework into your project, make sure the required functionality is not already present in the Java standard libraries (indeed, there are many specialized and high-performance algorithm implementations which outperforms the ones from standard libraries but in most of the cases you do not really need them).

7.10 Immutability

Immutability is all over the tutorial and in this part it stays as a reminder: please take immutability seriously. If a class you are designing or a method you are implementing could provide the immutability guarantee, it could be used mostly everywhere without the fear of concurrent modifications. It will make your life as a developer easier (and hopefully lives of your teammates as well).

7.11 Testing

Test-driven development (TDD) practices are extremely popular in Java community, raising the quality bar of the code being written. With all those benefits which TDD brings on the table, it is sad to observe that Java standard library does not include any test framework or scaffolding as of today.

Nevertheless, testing becomes a necessary part of the modern Java development and in this section we are going to cover some basics using excellent JUnit framework. Essentially, in JUnit, each test is a set of assertions about the expect object state or behavior.

The secret of writing great tests is to keep them short and simple, testing one thing at a time. As an exercise, let us write a set of tests to verify that `String.format` function from the section Strings returns the desired results.

```
package com.javacodegeeks.advanced.generic;

import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.equalTo;

import org.junit.Test;

public class StringFormatTestCase {
    @Test
    public void testNumberFormattingWithLeadingZeros() {
        final String formatted = String.format( "%04d", 1 );
        assertThat( formatted, equalTo( "0001" ) );
    }

    @Test
    public void testDoubleFormattingWithTwoDecimalPoints() {
        final String formatted = String.format( "%.2f", 12.324234d );
        assertThat( formatted, equalTo( "12.32" ) );
    }
}
```

The tests look very readable and their execution is instantiations. Nowadays, the average Java project contains hundreds of test cases, giving the developer a quick feedback on regressions or features under development.

7.12 Download the Source Code

This was a lesson on the General programming guidelines, lesson of Advanced Java course. You may download the source code here: AdvancedJavaPart7

7.13 What's next

This part of the tutorial finishes the series of discussions related to Java programming practices and guidelines. In the next part we are going to return back to the language features by exploring the world of Java exceptions, their types, how and when to use them.

Chapter 8

How and when to use Exceptions

8.1 Introduction

Exceptions in Java are an important instrument to signal abnormal (or exceptional) conditions in the program flow which may prevent it to make a further progress. By nature, those exceptional conditions may be fatal (the program is not able to function anymore and should be terminated) or recoverable (the program may continue to run though some functionality may not be available).

In this section of the tutorial we are going to walk through the typical scenario of using exceptions in Java, discuss the checked and unchecked exceptions, and touch some corner cases and useful idioms.

8.2 Exceptions and when to use them

In a nutshell, exceptions are some kind of events (or signals) which occur during program execution and interrupt the regular execution flow. The idea which led to introduction of the exceptions was born as a replacement to the error codes and status checking techniques used back in the days. Since then, exceptions gained widespread acceptance as the standard way to deal with error conditions in many programming languages, including Java.

There is only one important rule related to exceptions handling (not only in Java): never ignore them! Every exception should be a least logged (please see Exceptions and logging) but not ignored, ever. Nonetheless, there are those rare circumstances when exception could be safely ignored because really not much could be done about it (please refer to the Using try-with-resources section for the example).

And one more thing, in the **part 6** of the tutorial, **How to write methods efficiently**, we have discussed argument validation and sanity checks. Exceptions are a crucial part of these practices: every public method should verify all required preconditions before doing any real work and raise an appropriate exception if some of those have not been met.

8.3 Checked and unchecked exceptions

The exceptions management in the Java language differs from other programming languages. This is primarily because there are two classes of exceptions in Java: **checked** exceptions and **unchecked** exceptions. Interestingly, those two classes are somewhat artificial and are imposed by Java language rules and its compiler (but JVM makes no difference between them).

As a rule of thumb, unchecked exceptions are used to signal about erroneous conditions related to program logic and assumptions being made (invalid arguments, null pointers, unsupported operations, ...). Any unchecked exception is a subclass of `RuntimeException` and that is how Java compiler understands that a particular exception belongs to the class of unchecked ones.

Unchecked exceptions are not required to be caught by the caller or to be listed as a part of the method signature (using throws keyword). The `NullPointerException` is the most known member of unchecked exceptions and here is its declaration from the Java standard library:

```
public class NullPointerException extends RuntimeException {
    public NullPointerException() {
        super();
    }

    public NullPointerException(String s) {
        super(s);
    }
}
```

Consequently, checked exceptions represent invalid conditions in the areas which are outside of the immediate control of the program (like memory, network, file system, ...). Any checked exception is a subclass of Exception. In contrast to the unchecked exceptions, checked exceptions must be either caught by the caller or be listed as a part of the method signature (using `throws` keyword). The `IOException` is, probably, the most known one among checked exceptions:

```
public class IOException extends Exception {
    public IOException() {
        super();
    }

    public IOException(String message) {
        super(message);
    }

    public IOException(String message, Throwable cause) {
        super(message, cause);
    }

    public IOException(Throwable cause) {
        super(cause);
    }
}
```

The separation to checked and unchecked exceptions sounded like a good idea at the time, however over the years it turned out that it has introduced more boilerplate and not so pretty code patterns than solved the real problems. The typical (and unfortunately quite cumbersome) pattern which emerged within Java ecosystem is to hide (or wrap) the checked exception within unchecked one, for example:

```
try {
    // Some I/O operation here
} catch( final IOException ex ) {
    throw new RuntimeException( "I/O operation failed", ex );
}
```

It is not the best option available, however with a careful design of own exception hierarchies it may reduce a lot the amount of boilerplate code developers need to write.

It is worth to mention that there is another class of exceptions in Java which extends the Error class (for example, `OutOfMemoryError` or `StackOverflowError`). These exceptions usually indicate the fatal execution failure which leads to immediate program termination as recovering from such error conditions is not possible.

8.4 Using try-with-resources

Any exception thrown causes some, so called, stack unwinding and changes in the program execution flow. The results of that are possible resource leaks related to unclosed native resources (like file handles and network sockets). The typical well-behaved I/O operation in Java (up until version 7) required to use a mandatory `finally` block to perform the cleanup and usually was looking like that:

```
public void readFile( final File file ) {
    InputStream in = null;

    try {
        in = new FileInputStream( file );
        // Some implementation here
    } catch( IOException ex ) {
        // Some implementation here
    } finally {
        if( in != null ) {
            try {
                in.close();
            } catch( final IOException ex ) {
                /* do nothing */
            }
        }
    }
}
```

Nevertheless the `finally` block looks really ugly (unfortunately, not too much could be done here as calling close method on the input stream could also result into `IOException` exception), whatever happens the attempt to close input stream (and free up the operation system resources behind it) will be performed. In the section Exceptions and when to use them we emphasized on the fact that exceptions should be never ignored, however the ones thrown by close method are arguably the single exclusion from this rule.

Luckily, since Java 7 there is a new construct introduced into the language called **try-with-resources** which significantly simplified overall resource management. Here is the code snippet above rewritten using **try-with-resources**:

```
public void readFile( final File file ) {
    try( InputStream in = new FileInputStream( file ) ) {
        // Some implementation here
    } catch( final IOException ex ) {
        // Some implementation here
    }
}
```

The only thing which the resource is required to have in order to be used in the **try-with-resources** blocks is implementation of the interface `AutoCloseable`. Behind the scene Java compiler expands this construct to something more complex but for developers the code looks very readable and concise. Please use this very convenient technique where appropriate.

8.5 Exceptions and lambdas

In the **part 3** of the tutorial, **How to design Classes and Interfaces**, we already talked about latest and greatest Java 8 features, in particular lambda functions. However we have not looked deeply into many practical use cases and exceptions are one of them.

With no surprise, unchecked exceptions work as expected, however Java's lambda functions syntax does not allow to specify the checked exceptions (unless those are defined by `@FunctionalInterface` itself) which may be thrown. The following code snippet will not compile with a compilation error *"Unhandled exception type IOException" * (which could be thrown at line `03`):

```
public void readFile() {
     run( () -> {
         Files.readAllBytes( new File( "some.txt" ).toPath() );
     } );
 }

 public void run( final Runnable runnable ) {
     runnable.run();
 }
```

The only solution right now is to catch the `IOException` exception inside lambda function body and re-throw the appropriate `RuntimeException` exception (not forgetting to pass the original exception as a cause), for example:

```
public void readFile() {
    run( () -> {
        try {
            Files.readAllBytes( new File( "some.txt" ).toPath() );
        } catch( final IOException ex ) {
            throw new RuntimeException( "Error reading file", ex );
        }
    } );
}
```

Many functional interfaces are declared with the ability to throw any Exception from its implementation but if not (like Runnable), wrapping (or catching) the checked exceptions into unchecked ones is the only way to go.

8.6 Standard Java exceptions

The Java standard library provides a plenty on exception classes which are designated to cover most of the generic errors happening during program execution. The most widely used are presented in the table below, please consider those before defining your own.

Exception Class	Purpose
`NullPointerException`	Attempts to use `null` in a case where an object is required.
`IllegalArgumentException`	Method has been passed an illegal or inappropriate argument.
`IllegalStateException`	Method has been invoked at an illegal or inappropriate time.
`IndexOutOfBoundsException`	Index of some sort (such as to an array, to a string, or to a vector) is out of range.
`UnsupportedOperationException`	Requested operation is not supported.
`ArrayIndexOutOfBoundsException`	An array has been accessed with an illegal index.
`ClassCastException`	Code has attempted to cast an object to a subclass of which it is not an instance.
`EnumConstantNotPresentException`	Attempt to access an `enum` constant by name and the `enum` type contains no constant with the specified name (`enums` have been on covered in the **part 5** of the tutorial, **How and when to use Enums and Annotations**).
`NumberFormatException`	Attempts to convert a string to one of the numeric types, but that the string does not have the appropriate format.
`StringIndexOutOfBoundsException`	Index is either negative or greater than the size of the string.
`IOException`	I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.

Figure 8.1: advanced-java-standard-java-exceptions

8.7 Defining your own exceptions

The Java language makes it very easy to define own exception classes. Carefully designed exception hierarchies allow to implement detailed and fine-grained erroneous conditions management and reporting. As always, finding the right balance is very important: too many exception classes may complicate the development and blow the amount of the code involved in catching exception or propagating them down the stack.

It is strongly advised that all user-defined exceptions should be inherited from `RuntimeException` class and fall into the class of unchecked exceptions (however, there are always exclusions from the rule). For example, let us defined exception to dial with authentication:

```
public class NotAuthenticatedException extends RuntimeException {
    private static final long serialVersionUID = 2079235381336055509L;

    public NotAuthenticatedException() {
        super();
    }

    public NotAuthenticatedException( final String message ) {
        super( message );
    }

    public NotAuthenticatedException( final String message, final Throwable cause ) {
        super( message, cause );
    }
}
```

The purpose of this exception is to signal about non-existing or invalid user credentials during sing-in process, for example:

```
public void signin( final String username, final String password ) {
    if( !exists( username, password ) ) {
        throw new NotAuthenticatedException(
            "User / Password combination is not recognized" );
    }
}
```

It is always a good idea to pass the informative message along with the exception as it helps a lot to troubleshoot production systems. Also, if the exception was re-thrown as the result of another exceptional condition, the initial exception should be preserved using the `cause` constructor argument. It will help to figure out the real source of the problem.

8.8 Documenting exceptions

In the **part 6** of the tutorial, **How to write methods efficiently**, we have covered the proper documentation of the methods in Java. In this section we are going to spend a bit more time discussing how to make exceptions to be a part of the documentation as well.

If the method as a part of its implementation may throw the checked exception, it must become a part of the method signature (using `throws` declaration). Respectively, Java documentation tool has the `@throws` tag for describing those exceptions. For example:

```
/**
 * Reads file from the file system.
 * @throws IOException if an I/O error occurs.
 */
public void readFile() throws IOException {
    // Some implementation here
}
```

In contrast, as we know from section Checked and unchecked exceptions, the unchecked exception usually are not declared as part of method signature. However it is still a very good idea to document them so the caller of the method will be aware of possible exceptions which may be thrown (using the same `@throws` tag). For example:

```
/**
 * Parses the string representation of some concept.
 * @param str String to parse
 * @throws IllegalArgumentException if the specified string cannot be parsed properly
 * @throws NullPointerException if the specified string is null
```

```
 */
public void parse( final String str ) {
    // Some implementation here
}
```

Please always document the exceptions which your methods could throw. It will help other developers to implement proper exception handling and recovering (fallback) logic from the beginning, rescuing them from troubleshooting issues in production systems.

8.9 Exceptions and logging

Logging (http://en.wikipedia.org/wiki/Logfile) is an essential part of any more or less complex Java application, library or framework. It is a journal of the important events happening in the application and exceptions are the crucial part of this flow. Later in the tutorial we may cover a bit the logging subsystem provided by Java standard library however please remember that exceptions should be properly logged and analyzed later on in order to discover problems in the applications and troubleshoot critical issues.

8.10 Download the Source Code

This was a lesson on how and when to use Exceptions. You may download the source code here: advanced-java-part-8

8.11 What's next

In this part of the tutorial we have covered exceptions, a very important feature of the Java language. We have seen that exceptions are the foundation of the errors management in Java. Exceptions make handling and signaling erroneous conditions quite an easy job and, in contrast to error codes, flags and statuses, once occurred, exceptions cannot be ignored. In the next part we are going to address a very hot and complicated topic: concurrency and multithreaded programming in Java.

Chapter 9

Concurrency best practices

9.1 Introduction

The multiprocessor and multicore hardware architectures greatly influence the design and execution model of applications which run on them nowadays. In order to utilize the full power of available computational units, the applications should be ready to support multiple execution flows which are running concurrently and competing for resources and memory. Concurrent programming brings a lot of challenges related to data access and non-deterministic flow of events which can lead to unexpected crashes and strange failures.

In this part of the tutorial we are going to look at what Java can offer to the developers in order to help them to write robust and safe applications in concurrent world.

9.2 Threads and Thread Groups

Threads are the foundational building blocks of concurrent applications in Java. Threads are sometimes called **lightweight processes** and they allow multiple execution flows to proceed concurrently. Every single application in Java has at least one thread called the **main thread**. Every Java thread exists inside JVM only and may not reflect any operating system thread.

Threads in Java are instances of class `Thread`. Typically, it is not recommended to directly create and manage threads using the instances of Thread class (executors and thread pools covered in section Futures and Executors provide a better way to do that), however it is very easy to do:

```
public static void main(String[] args) {
    new Thread( new Runnable() {
        @Override
        public void run() {
            // Some implementation here
        }
    } ).start();
}
```

Or the same example using Java 8 lambda functions:

```
public static void main(String[] args) {
    new Thread( () -> { /* Some implementation here */ } ).start();
}
```

Nevertheless creating a new thread in Java looks very simple, threads have a complex lifecycle and can be in one of the following states (a thread can be in only one state at a given point in time).

Thread State	Description
NEW	A thread that has not yet started is in this state.
RUNNABLE	A thread executing in the Java virtual machine is in this state.
BLOCKED	A thread that is blocked waiting for a monitor lock is in this state.
WAITING	A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TIMED_WAITING	A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TERMINATED	A thread that has exited is in this state.

Figure 9.1: Thread States

Not all thread states are clear right now but later in the tutorial we will go over most of them and discuss what kind of events cause the thread to be in one state or another.

Threads could be assembled into groups. A thread group represents a set of threads and can also include other thread groups (thus forming a tree). Threads groups intended to be a nice feature however they are not recommended for use nowadays as executors and thread pools (please see Futures) are much better alternatives.

9.3 Concurrency, Synchronization and Immutability

In mostly every Java application multiple running threads need to communicate with each other and access shared data. Reading this data is not so much of a problem, however uncoordinated modification of it is a straight road to disaster (so called racing conditions). That is the point where **synchronization** kicks in. **Synchronization** is a mechanism to ensure that several concurrently running threads will not execute the specifically guarded (synchronized) block of application code at the same time. If one of the threads has begun to execute a synchronized block of the code, any other thread trying to execute the same block must wait until the first one finishes.

Java language has **synchronization** support built-in in the form of `synchronized` keyword. This keyword can be applied to instance methods, static methods or used around arbitrary execution blocks and guarantees that only one thread at a time will be able to invoke it. For example:

```
public synchronized void performAction() {
    // Some implementation here
}

public static synchronized void performClassAction() {
    // Some implementation here
}
```

Or, alternatively, the example which uses the synchronized with a code block:

```
public void performActionBlock() {
    synchronized( this ) {
        // Some implementation here
    }
}
```

There is another very important effect of `synchronized` keyword: it automatically establishes a **happens-before** relationship (http://en.wikipedia.org/wiki/Happened-before) with any invocation of a `synchronized` method or code block for the same object. That guarantees that changes to the state of the object are visible to all threads.

Please notice that constructors cannot be synchronized (using the `synchronized` keyword with a constructor raises compiler error) because only the thread which creates an instance has access to it while instance is being constructed.

In Java, synchronization is built around an internal entity known as **monitor** (or intrinsic/monitor lock, http://en.wikipedia.org/-wiki/Monitor_(synchronization)). Monitor enforces exclusive access to an object's state and establishes **happens-before** relationships. When any thread invokes a `synchronized` method, it automatically acquires the intrinsic (monitor) lock for that method's instance (or class in case of static methods) and releases it once the method returns.

Lastly, the synchronization is Java is **reentrant**: it means that the thread can acquire a lock which it already owns. Reentrancy significantly simplifies the programming model of the concurrent applications as the threads have fewer chances to block themselves.

As you can see, concurrency introduces a lot of complexity into the Java applications. However, there is a cure: **immutability**. We have talked about that many times already, but it is really very important for multithreaded applications in particular: immutable objects do not need the synchronization as they are never being updated by more than one threads.

9.4 Futures, Executors and Thread Pools

Creating new threads in Java is easy, but managing them is really tough. Java standard library provides extremely useful abstractions in the form of executors and thread pools targeted to simplify threads management.

Essentially, in its simplest implementation, thread pool creates and maintains a list of threads, ready to be used right away. Applications, instead of spawning new thread every time, just borrows the one (or as many as needed) from the pool. Once borrowed thread finishes its job, it is returned back to the pool, and becomes available to pick up next task.

Though it is possible to use thread pools directly, Java standard library provides an executors faÃ§ade which has a set of factory method to create commonly used thread pool configurations. For example, the code snippet below creates a thread pool with fixed number of threads (10):

```
ExecutorService executor = Executors.newFixedThreadPool( 10 );
```

Executors could be used to offload any task so it will be executed in the separate thread from the thread pool (as a note, it is not recommended to use executors for long-running tasks). The executors' facade allows customizing the behavior of the underlying thread pool and supports following configurations:

Thread State	Description
NEW	A thread that has not yet started is in this state.
RUNNABLE	A thread executing in the Java virtual machine is in this state.
BLOCKED	A thread that is blocked waiting for a monitor lock is in this state.
WAITING	A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TIMED_WAITING	A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TERMINATED	A thread that has exited is in this state.

Figure 9.2: Thread States

In some cases, the result of the execution is not very important so executors support **fire-and-forget** semantic, for example:

```
executor.execute( new Runnable() {
    @Override
    public void run() {
        // Some implementation here
    }
} );
```

The equivalent Java 8 example is much more concise:

```
executor.execute( () -> {
    // Some implementation here
} );
```

But if the result of the execution is important, Java standard library provides another abstraction to represent the computation which will happen at some point in the future, called `Future<T>`. For example:

```
Future< Long > result = executor.submit( new Callable< Long >() {
     @Override
     public Long call() throws Exception {
        // Some implementation here
        return ...;
     }
} );
```

The result of the `Future<T>` might not be available immediately so the application should wait for it using a family of `get(...)` methods. For example:

```
Long value = result.get( 1, TimeUnit.SECONDS );
```

If result of the computation is not available within specified timeout, the `TimeoutException` exception will be raised. There is an overloaded version of **`get()`** which waits forever but please prefer to use the one with timeout.

Since the Java 8 release, developers have yet another version of the `Future<T>`, `CompletableFuture<T>`, which supports addition functions and actions that trigger upon its completion. Not only that, with introduction of streams, Java 8 introduces a simple and very straightforward way to perform parallel collection processing using **`parallelStream()`** method, for example:

```
final Collection< String > strings = new ArrayList<>();
// Some implementation here

final int sumOfLengths = strings.parallelStream()
    .filter( str -> !str.isEmpty() )
    .mapToInt( str -> str.length() )
    .sum();
```

The simplicity, which executors and parallel streams brought to the Java platform, made the concurrent and parallel programming in Java much easier. But there is a catch: uncontrolled creation of thread pools and parallel streams could kill application performance so it is important to manage them accordingly.

9.5 Locks

Additionally to the monitors, Java has support of the reentrant mutual exclusion locks (with the same basic behavior and semantics as the monitor lock but with more capabilities). Those locks are available through **`ReentrantLock`** class from **`java.util.concurrent.locks`** package. Here is a typical lock usage idiom:

```
private final ReentrantLock lock = new ReentrantLock();

public void performAction() {
    lock.lock();

    try {
        // Some implementation here
    } finally {
        lock.unlock();
    }
}
```

Please notice that any lock must be explicitly released by calling the **`unlock()`** method (for `synchronized` methods and execution blocks Java compiler under the hood emits the instructions to release the monitor lock). If the locks require writing more code, why they are better than monitors? Well, for couple of reason but most importantly, locks could use timeouts while waiting for acquisition and fail fast (monitors always wait indefinitely and do not have a way to specify the desired timeout). For example:

```
public void performActionWithTimeout() throws InterruptedException {
    if( lock.tryLock( 1, TimeUnit.SECONDS ) ) {
        try {
            // Some implementation here
        } finally {
            lock.unlock();
        }
    }
}
```

Now, when we have enough knowledge about monitors and locks, let us discuss how their usage affects thread states.

When any thread is waiting for the lock (acquired by another thread) using **`lock()`** method call, it is in a `WAITING` state. However, when any thread is waiting for the lock (acquired by another thread) using **`tryLock()`** method call with timeout, it is in a `TIMED_WAITING` state. In contrast, when any thread is waiting for the monitor (acquired by another thread) using `synchronized` method or execution block, it is in a `BLOCKED` state.

The examples we have seen so far are quite simple but lock management is really hard and full of pitfalls. The most infamous of them is **deadlock:** a situation in which two or more competing threads are waiting for each other to proceed and thus neither ever does so. Deadlocks usually occur when more than one lock or monitor lock are involved. JVM often is able to detect the deadlocks in the running applications and warn the developers (see please Troubleshooting Concurrency Issues section). The canonical example of the deadlock looks like this:

```
private final ReentrantLock lock1 = new ReentrantLock();
private final ReentrantLock lock2 = new ReentrantLock();

public void performAction() {
    lock1.lock();

    try {
        // Some implementation here
        try {
            lock2.lock();
            // Some implementation here
        } finally {
            lock2.unlock();
        }
        // Some implementation here
    } finally {
        lock1.unlock();
    }
}

public void performAnotherAction() {
    lock2.lock();

    try {
        // Some implementation here
        try {
            lock1.lock();
            // Some implementation here
        } finally {
            lock1.unlock();
        }
        // Some implementation here
    } finally {
```

```
            lock2.unlock();
        }
    }
```

The `performAction()` method tries to acquire `lock1` and then `lock2`, while the method `performAnotherAction()` does it in the different order, `lock2` and then `lock1`. If by program execution flow those two methods are being called on the same class instance in two different threads, the deadlock is very likely to happen: the first thread will be waiting indefinitely for the `lock2` acquired by the second thread, while the second thread will be waiting indefinitely for the `lock1` acquired by the first one.

9.6 Thread Schedulers

In JVM, the thread scheduler determines which thread should be run and for how long. All threads created by Java applications have the priority which basically influences the thread scheduling algorithm when it makes a decision when thread should be scheduled and its time quantum. However this feature has a reputation of being non-portable (as mostly every trick which relies on specific behavior of the thread scheduler).

The **`Thread`** class also provide another way to intervene into thread scheduling implementation by using a **`yield()`** method. It hints the thread scheduler that the current thread is willing to yield its current use of a processor time (and also has a reputation of being non-portable).

By and large, relying on the Java thread scheduler implementation details is not a great idea. That is why the executors and thread pools from the Java standard library (see please Futures section) try to not expose those non-portable details to the developers (but still leaving a way to do so if it is really necessary). Nothing works better than careful design which tries to take into account the real hardware the application is running on (f.e., number of available CPUs and cores could be retrieved easily using **`Runtime`** class).

9.7 Atomic Operations

In multithread world there is a particular set of instructions called compare-and-swap (CAS). Those instructions compare their values to a given ones and, only if they are the same, set a new given values. This is done as a single atomic operation which is usually lock-free and highly efficient.

Java standard library has a large list of classes supporting atomic operation, all of them located under **`java.util.concurrent.atomic`** package.

Class	Description
`AtomicBoolean`	A **boolean** value that may be updated atomically
`AtomicInteger`	An **int** value that may be updated atomically.
`AtomicIntegerArray`	A **long** value that may be updated atomically.
`AtomicLongArray`	A **long** array in which elements may be updated atomically.
`AtomicReference<V>`	An object reference that may be updated atomically.
`AtomicReferenceArray<E>`	An array of object references in which elements may be updated atomically.

Figure 9.3: Atomic Classes

The Java 8 release extends the **`java.util.concurrent.atomic`** with a new set of atomic operations (accumulators and adders).

Class	Description
`DoubleAccumulator`	One or more variables that together maintain a running double value updated using a supplied function.
`DoubleAdder`	One or more variables that together maintain an initially zero double sum.
`LongAccumulator`	One or more variables that together maintain a running long value updated using a supplied function.
`LongAdder`	One or more variables that together maintain an initially zero long sum.

Figure 9.4: Atomic Classes Java 8

9.8 Concurrent Collections

Shared collections, accessible and modifiable by multiple threads, are rather a rule than an exception. Java standard library provides a couple of helpful static methods in the class **`Collections`** which make any existing collection thread-safe. For example:

```
final Set< String > strings =
    Collections.synchronizedSet( new HashSet< String >() );

final Map< String, String > keys =
    Collections.synchronizedMap( new HashMap< String, String >() );
```

However the returned general-purpose collection wrappers are thread-safe, it is often not the best option as they provide quite a mediocre performance in real-world applications. That is why Java standard library includes a rich set of collection classes tuned for concurrency. Below is just a list of most widely used ones, all hosted under **`java.util.concurrent`** package.

Class	Description
`CountDownLatch`	A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
`CyclicBarrier`	A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
`Exchanger<V>`	A synchronization point at which threads can pair and swap elements within pairs.
`Phaser`	A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.
`Semaphore`	A counting semaphore.
`ThreadLocalRandom`	A random number generator isolated to the current thread
`ReentrantReadWriteLock`	An implementation of read/write lock

Figure 9.5: Atomic Classes Java 8

Those classes are specifically designed to be used in the multithreaded applications. They exploit a lot of techniques to make the concurrent access to the collection as efficient as possible and are the recommended replacement to `synchronized` collection wrappers.

9.9 Explore Java standard library

The **`java.util.concurrent`** and **`java.util.concurrent.locks`** packages are real gems for the Java developers who are writing concurrent applications. As there are a lot of the classes there, in this section we are going to cover most useful

of them, but please do not hesitate to consult Java official documentation and explore them all.

Class	Description
`CountDownLatch`	A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
`CyclicBarrier`	A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
`Exchanger<V>`	A synchronization point at which threads can pair and swap elements within pairs.
`Phaser`	A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.
`Semaphore`	A counting semaphore.
`ThreadLocalRandom`	A random number generator isolated to the current thread
`ReentrantReadWriteLock`	An implementation of read/write lock

Figure 9.6: Atomic Classes Java 8

Unfortunately, the Java implementation of **`ReentrantReadWriteLock`** was not so great and as of Java 8, there is new kind of lock:

`StampedLock`: A capability-based lock with three modes for controlling read/write access.

9.10 Using Synchronization Wisely

Locking and `synchronized` keyword are powerful instruments which help a lot to keep the data model and program state consistent in multithreaded applications. However, using them unwisely causes threads contention and could dramatically decrease application performance. From the other side, not using the synchronization primitives may (and will) lead to a weird program state and corrupted data which eventually causes application to crash. So the balance is important.

The advice is to try to use locks or/and `synchronized` where it is really necessary. While doing so, make sure that the locks are released as soon as possible, and the execution blocks which require locking or synchronization are kept minimal. Those techniques at least should help to reduce the contention but will not eliminate it.

In the recent years, a lot of so called lock-free algorithms and data structure have emerged (f.e., atomic operations in Java from Atomic Operations section). They provide much better performance comparing to the equivalent implementations which are built using synchronization primitives.

It is good to know that JVM has a couple of runtime optimizations in order to eliminate the locking when it may be not necessary. The most known is **biased locking**: an optimization that improves uncontended synchronization performance by eliminating operations associated with the Java synchronization primitives (for more details, please refer to http://www.oracle.com/technetwork/java/6-performance-137236.html#2.1.1).

Nevertheless JVM does its best, eliminating the unnecessary synchronization in the application is much better option. Excessive use of synchronization has a negative impact on application performance as threads will be wasting expensive CPU cycles competing for resources instead of doing the real work.

9.11 Wait/Notify

Prior to the introduction of the concurrency utilities in the Java standard library (**`java.util.concurrent`**), the usage of **`Object`**'s **`wait()/notify()/notifyAll()`** methods was the way to establish communication between threads in Java. **All those methods must be called only if the thread owns the monitor on the object in question**. For example:

```
private Object lock = new Object();

public void performAction() {
    synchronized( lock ) {
        while( <condition> ) {
            // Causes the current thread to wait until
            // another thread invokes the notify() or notifyAll() methods.
            lock.wait();
        }

        // Some implementation here
    }
}
```

Method **`wait()`** releases the monitor lock the current thread holds because the condition it is waiting for is not met yet (**wait() method must be called in a loop and should never be called outside of a loop**). Consequently, another thread waiting on the same monitor gets a chance to run. When this thread is done, it should call one of **`notify()/notifyAll()`** methods to wake up the thread (or threads) waiting for the monitor lock. For example:

```
public void performAnotherAction() {
    synchronized( lock ) {
        // Some implementation here

        // Wakes up a single thread that is waiting on this object's monitor.
        lock.notify();
    }
}
```

The difference between **`notify()`** and **`notifyAll()`** is that the first wakes up a single thread while second wakes up all waiting threads (which start to contend for the monitor lock).

The **`wait()/notify()`** idiom is not advisable to be used in the modern Java applications. Not only is it complicated, it also requires following a set of mandatory rules. As such, it may cause subtle bugs in the running program which will be very hard and time-consuming to investigate. The `java.util.concurrent` has a lot to offer to replace the `wait()/notify()` with much simpler alternatives (which very likely will have much better performance in the real-world scenario).

9.12 Troubleshooting Concurrency Issues

Many, many things could go wrong in multithreaded applications. Reproducing issues becomes a nightmare. Debugging and troubleshooting can take hours and even days or weeks. Java Development Kit (JDK) includes a couple of tools which at least are able to provide some details about application threads and their states, and diagnose deadlock conditions (see please Threads and thread groups and Locks sections). It is a good point to start with. Those tools are (but not limited to):

- **JVisualVM**(http://docs.oracle.com/javase/7/docs/technotes/tools/share/jvisualvm.html)
- **Java Mission Control** (http://docs.oracle.com/javacomponents/jmc.htm)
- **jstack** (https://docs.oracle.com/javase/7/docs/technotes/tools/share/jstack.html)

9.13 Download

You can download the source code of this lesson here: **advanced-java-part-9**

9.14 What's next

In this part we have looked through very important aspect of modern software and hardware platforms - concurrency. In particular, we have seen what kind of instruments Java as a language and its standard library offer to developers to help them dealing with concurrency and asynchronous executions. In the next part of the tutorial we are going to cover serialization techniques in Java.

Chapter 10

Built-in Serialization techniques

10.1 Introduction

This part of the tutorial is going to be solely devoted to **serialization**: the process of translating Java objects into a format that can be used to store and be reconstructed later in the same (or another) environment (http://en.wikipedia.org/wiki/Serialization). **Serialization** not only allows saving and loading Java objects to/from the persistent storage, but is also a very important component of modern distributed systems communication.

Serialization is not easy, but effective **serialization** is even harder. Besides the Java standard library, there are many serialization techniques and frameworks available: some of them are using compact binary representation, others put the readability on the first place. Although we are going to mention many alternatives along the way, our attention will be concentrated on the ones from Java standard library (and latest specifications): `Serializable`, `Externalizable`, Java Architecture for XML Binding (**JAXB**, JSR-222) and Java API for JSON Processing (**JSON-P**, JSR-353).

10.2 Serializable interface

Arguably, the easiest way in Java to mark the class as available for serialization is by implementing the `java.io.Serializable` interface. For example:

```
public class SerializableExample implements Serializable {
}
```

The serialization runtime associates with each serializable class a special version number, called a **serial version UID**, which is used during **deserialization** (the process opposite to **serialization**) to make sure that the loaded classes for the serialized object are compatible. In case the compatibility has been compromised, the `InvalidClassException` will be raised.

A serializable class may introduce its own **serial version UID** explicitly by declaring a field with name `serialVersionUID` that must be `static`, `final`, and of type `long`. For example:

```
public class SerializableExample implements Serializable {
    private static final long serialVersionUID = 8894f47504319602864L;
}
```

However, if a serializable class does not explicitly declare a `serialVersionUID` field, then the serialization runtime will generate a default `serialVersionUID` field for that class. It is worth to know that it is strongly recommended by all classes implementing `Serializable` to explicitly declare the `serialVersionUID` field, because the default `serialVersionUID` generation heavily relies on intrinsic class details and may vary depending on Java compiler implementation and its version. As such, to guarantee a consistent behavior, a serializable class must always declare an explicit `serialVersionUID` field.

Once the class becomes serializable (implements `Serializable` and declares `serialVersionUID`), it could be stored and retrieved using, for example, `ObjectOutputStream` / `ObjectInputStream`:

```
final Path storage = new File( "object.ser" ).toPath();

try( final ObjectOutputStream out =
        new ObjectOutputStream( Files.newOutputStream( storage ) ) ) {
    out.writeObject( new SerializableExample() );
}
```

Once stored, it could be retrieved in a similar way, for example:

```
try( final ObjectInputStream in =
        new ObjectInputStream( Files.newInputStream( storage ) ) ) {
    final SerializableExample instance = ( SerializableExample )in.readObject();
    // Some implementation here
}
```

As we can see, the `Serializable` interface does not provide a lot of control over what should be serialized and how (with exception of `transient` keyword which marks the fields as non-serializable). Moreover, it limits the flexibility of changing the internal class representation as it could break the serialization / deserialization process. That is why another interface, `Externalizable`, has been introduced.

10.3 Externalizable interface

In contrast to `Serializable` interface, `Externalizable` delegates to the class the responsibility of how it should be serialized and deserialized. It has only two methods and here is its declaration from the Java standard library:

```
public interface Externalizable extends java.io.Serializable {
    void writeExternal(ObjectOutput out) throws IOException;
    void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
}
```

In turn, every class which implements `Externalizable` interface should provide the implementation of these two methods. Let us take a look on the example:

```
public class ExternalizableExample implements Externalizable {
    private String str;
    private int number;
    private SerializableExample obj;

    @Override
    public void readExternal(final ObjectInput in)
            throws IOException, ClassNotFoundException {
        setStr(in.readUTF());
        setNumber(in.readInt());
        setObj(( SerializableExample )in.readObject());
    }

    @Override
    public void writeExternal(final ObjectOutput out)
            throws IOException {
        out.writeUTF(getStr());
        out.writeInt(getNumber());
        out.writeObject(getObj());
    }
}
```

Similarly to the classes implementing `Serializable`, the classes implementing `Externalizable` could be stored and retrieved using, for example, `ObjectOutputStream` / `ObjectInputStream`:

```
final Path storage = new File( "extobject.ser" ).toPath();

final ExternalizableExample instance = new ExternalizableExample();
instance.setStr( "Sample String" );
instance.setNumber( 10 );
instance.setObj( new SerializableExample() );

try( final ObjectOutputStream out =
        new ObjectOutputStream( Files.newOutputStream( storage ) ) ) {
    out.writeObject( instance );
}

try( final ObjectInputStream in =
        new ObjectInputStream( Files.newInputStream( storage ) ) ) {
    final ExternalizableExample obj = ( ExternalizableExample )in.readObject();
    // Some implementation here
}
```

The `Externalizable` interface allows a fine-grained serialization / deserialization customization in the cases when the simpler approach with `Serializable` interface does not work well.

10.4 More about Serializable interface

In the previous section we mentioned that the `Serializable` interface does not provide a lot of control over what should be serialized and how. In fact, it is not completely true (at least when `ObjectOutputStream` / `ObjectInputStream` are used). There are some special methods which any serializable class can implement in order to control the default serialization and deserialization.

```
private void writeObject(ObjectOutputStream out) throws IOException;
```

This method is responsible for writing the state of the object for its particular class so that the corresponding `readObject` method can restore it (the default mechanism for saving the Object's fields can be invoked by calling `out.defaultWriteObject`).

```
private void readObject(ObjectInputStream in) throws IOException,ClassNotFoundException;
```

This method is responsible for reading from the stream and restoring the state of the object (the default mechanism for restoring the Object's fields can be invoked by calling `in.defaultReadObject`).

```
private void readObjectNoData() throws ObjectStreamException;
```

This method is responsible for initializing the state of the object in the case when the serialization stream does not list the given class as a superclass of the object being deserialized.

```
Object writeReplace() throws ObjectStreamException;
```

This method is used when serializable classes need to designate an alternative object to be used when writing an object to the stream.

```
Object readResolve() throws ObjectStreamException;
```

And lastly, this method is used when serializable classes need to designate a replacement when an instance of it is read from the stream.

The default serialization mechanism (using `Serializable` interface) could get really cumbersome in Java once you know the intrinsic implementation details and those special methods to use. More code you are writing to support serialization, more likely more bugs and vulnerabilities will show off.

However, there is a way to reduce those risks by employing quite simple pattern named **Serialization Proxy**, which is based on utilizing `writeReplace` and `readResolve` methods. The basic idea of this pattern is to introduce dedicated companion class for serialization (usually as `private static` inner class), which complements the class required to be serialized. Let us take a look on this example:

```
public class SerializationProxyExample implements Serializable {
    private static final long serialVersionUID = 6163321482548364831L;

    private String str;
    private int number;

    public SerializationProxyExample( final String str, final int number) {
        this.setStr(str);
        this.setNumber(number);
    }

    private void readObject(ObjectInputStream stream) throws InvalidObjectException {
        throw new InvalidObjectException( "Serialization Proxy is expected" );
    }

    private Object writeReplace() {
        return new SerializationProxy( this );
    }

    // Setters and getters here
}
```

When the instances of this class are being serialized, the class `SerializationProxyExample` implementation provides the replacement object (instance of the `SerializationProxy` class) instead. It means that instances of the `SerializationProxyExample` class will never be serialized (and deserialized) directly. It also explains why the `readObject` method raises an exception in case a deserialization attempt somehow happens. Now, let us take a look on the companion `SerializationProxy` class:

```
private static class SerializationProxy implements Serializable {
    private static final long serialVersionUID = 8368440585226546959L;

    private String str;
    private int number;

    public SerializationProxy( final SerializationProxyExample instance ) {
        this.str = instance.getStr();
        this.number = instance.getNumber();
    }

    private Object readResolve() {
        return new SerializationProxyExample(str, number); // Uses public constructor
    }
}
```

In our somewhat simplified case, the `SerializationProxy` class just duplicates all the fields of the `SerializationProxyExample` (but it could be much complicated than that). Consequently, when the instances of this class are being deserialized, the `readResolve` method is called and `SerializationProxy` provides the replacement as well, this time in a shape of `SerializationProxyExample` instance. As such, the `SerializationProxy` class serves as a serialization proxy for `SerializationProxyExample` class.

10.5 Serializability and Remote Method Invocation (RMI)

For quite some time, Java Remote Method Invocation (**RMI**) was the only mechanism available for building distributed applications on Java platform. **RMI** provides all the heavy lifting and makes it possible to transparently invoke the methods of remote

Java objects from other JVMs on the same host or on different physical (or virtual) hosts. In the foundation of **RMI** lays object serialization, which is used to marshal (serialize) and unmarshal (deserialize) method parameters.

RMI is still being used in many Java applications nowadays, but it lesser and lesser becomes a choice because of its complexity and communication restrictions (most of the firewalls do block **RMI** ports). To get more details about **RMI** please refer to official documentation.

10.6 JAXB

Java Architecture for XML Binding, or just **JAXB**, is probably the oldest alternative serialization mechanism available to Java developers. Underneath, it uses XML as the serialization format, provides a wide range of customization options and includes a lot of annotations which makes **JAXB** very appealing and easy to use (annotations are covered in **part 5** of the tutorial, **How and when to use Enums and Annotations**).

Let us take a look on a quite simplified example of the plain old Java class (POJO) annotated with **JAXB** annotations:

```
import java.math.BigDecimal;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType( XmlAccessType.FIELD )
@XmlRootElement( name = "example" )
public class JaxbExample {
    @XmlElement(required = true) private String str;
    @XmlElement(required = true) private BigDecimal number;

    // Setters and getters here
}
```

To serialize the instance of this class into XML format using **JAXB** infrastructure, the only thing needed is the instance of the marshaller (or serializer), for example:

```
final JAXBContext context = JAXBContext.newInstance( JaxbExample.class );
final Marshaller marshaller = context.createMarshaller();

final JaxbExample example = new JaxbExample();
example.setStr( "Some string" );
example.setNumber( new BigDecimal( 12.33d, MathContext.DECIMAL64 ) );

try( final StringWriter writer = new StringWriter() ) {
    marshaller.marshal( example, writer );
}
```

Here is the XML representation of the `JaxbExample` class instance from the example above:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<example>
    <str>Some string</str>
    <number>12.33000000000000</number>
</example>
```

Following the same principle, the instances of the class could be deserialized back from XML representation into the Java objects using the instance of the unmarshaller (or deserializer), for example:

```
final JAXBContext context = JAXBContext.newInstance( JaxbExample.class );

final String xml = "" +
    "<?xml version=\\"1.0\\" encoding=\\"UTF-8\\" standalone=\\"yes\\"?>" +
```

```
    "<example>" +
    "    <str>Some string</str>" +
    "    <number>12.33000000000000</number>" +
    "</example>";

final Unmarshaller unmarshaller = context.createUnmarshaller();
try( final StringReader reader = new StringReader( xml ) ) {
    final JaxbExample example = ( JaxbExample )unmarshaller.unmarshal( reader );
    // Some implementaion here
}
```

As we can see, **JAXB** is quite easy to use and the XML format is still quite popular choice nowadays. However, one of the fundamental pitfalls of XML is verbosity: quite often the necessary XML structural elements significantly surpass the effective data payload.

10.7 JSON-P

Since 2013, Java developers are able to use **JSON** as the serialization format, by virtue of newly introduced Java API for JSON Processing (**JSON-P**).

As of now, **JSON-P** is not a part of Java standard library although there are many discussions to include native **JSON** support into the language in the upcoming **Java 9** release (http://openjdk.java.net/jeps/198). Nevertheless, it is there and available as part of Java **JSON Processing Reference Implementation** (https://jsonp.java.net/).

In contrast to **JAXB**, there is nothing required to be added to the class to make it suitable for **JSON** serialization, for example:

```
public class JsonExample {
    private String str;
    private BigDecimal number;
    // Setters and getters here
}
```

The serialization is not as transparent as with **JAXB**, and requires a bit of code to be written for each class intended to be serialized into **JSON**, for example:

```
final JsonExample example = new JsonExample();
example.setStr( "Some string" );
example.setNumber( new BigDecimal( 12.33d, MathContext.DECIMAL64 ) );

try( final StringWriter writer = new StringWriter() ) {
    Json.createWriter(writer).write(
        Json.createObjectBuilder()
            .add("str", example.getStr() )
            .add("number", example.getNumber() )
            .build()
        );
}
```

And here is the **JSON** representation of the `JsonExample` class instance from the example above:

```
{
    "str":"Some string",
    "number":12.33000000000000
}
```

The deserialization process goes in the same vein:

```
final String json = "{\\"str\\":\\"Some string\\",\\"number\\":12.33000000000000}";

try( final StringReader reader = new StringReader( json ) ) {
```

```
    final JsonObject obj = Json.createReader( reader ).readObject();
    final JsonExample example = new JsonExample();
    example.setStr( obj.getString( "str" ) );
    example.setNumber( obj.getJsonNumber( "number" ).bigDecimalValue() );
}
```

It is fair to say that at the moment **JSON** support in Java is pretty basic. Nonetheless it is a great thing to have and Java community is working on enriching the **JSON** support by introducing **Java API for JSON Binding** (JSON-B, JSR-367). With this API the serialization and deserialization of the Java objects to/from **JSON** should be as transparent as **JAXB** has.

10.8 Cost of serialization

It is very important to understand that though serialization / deserialization looks simple in Java, it is not free and depending on the data model and data access patterns may consume quite a lot of network bandwidth, memory and CPU resources. More to that, nevertheless Java has some kind of versioning support for the serializable classes (using **serial version UID** as we have seen in the section Serializable interface), it does make the development process much harder as developers are on their own to figure out how to manage data model evolution.

To add another point, Java serialization does not work well outside of JVM world. It is a significant constraint for the modern distributed applications which are built using multiple programming languages and runtimes.

That explains why many alternative serialization frameworks and solutions have emerged and became very popular choice in the Java ecosystem.

10.9 Beyond Java standard library and specifications

In this section we are going to look on alternative solutions for painless and effective Java serialization, starting from the **Fast-serialization** project (http://ruedigermoeller.github.io/fast-serialization/): the fast Java serialization drop in-replacement. The usage of **Fast-serialization** is not much different from what Java standard library provides but it claims to be much faster and more effective.

Another set of frameworks has a different take on the problem. They are based on structured data definition (or protocol) and serialize data into compact binary representation (the corresponding data model could be even generated from the definition). Aside from that, those frameworks are going far beyond just Java platform and can be used for cross-language / cross-platform serialization. The most known Java libraries in this space are **Google Protocol Buffers** (https://developers.google.com/protocol-buffers/), **Apache Avro** (http://avro.apache.org/) and **Apache Thrift** (https://thrift.apache.org/).

10.10 Download the Source code

You can dowload the source code of this course here: **advanced-java-part-10**

10.11 What's next

In this part of the tutorial we have discussed the built-in serialization techniques provided by the Java language and its runtime. We have seen how important serialization is today, when mostly every single application being built is a part of larger distributed system and needs to communicate with the rest of it (or with other external systems). In the next part of the tutorial we are going to talk about reflection and dynamic languages support in Java.

Chapter 11

How to use Reflection effectively

11.1 Introduction

In this part of the tutorial we are going to briefly look through a very interesting subject called **reflection**. **Reflection** is the ability of the program to examine or introspect itself at runtime. **Reflection** is an extremely useful and powerful feature which significantly expands the capabilities of the program to perform its own inspections, modifications or transformations during its execution, without a single line of code change. Not every programming language implementation has this feature supported, but luckily Java has adopted it since its beginning.

11.2 Reflection API

The **Reflection API**, which is part of the Java standard library, provides a way to explore intrinsic class details at runtime, dynamically create new class instances (without the explicit usage of the `new` operator), dynamically invoke methods, introspect annotations (annotations have been covered in **part 5** of the tutorial, **How and when to use Enums and Annotations**), and much, much more. It gives the Java developers a freedom to write the code which could adapt, verify, execute and even modify itself while it is being running.

The **Reflection API** is designed in quite intuitive way and is hosted under the **`java.lang.reflect`** package. Its structure follows closely the Java language concepts and has all the elements to represent classes (including generic versions), methods, fields (members), constructors, interfaces, parameters and annotations. The entry point for the **Reflection API** is the respective instance of the **`Class< ?>`** class. For example, the simplest way to list all public methods of the class String is by using the `getMethods()` method call:

```
final Method[] methods = String.class.getMethods();
for( final Method method: methods ) {
    System.out.println( method.getName() );
}
```

Following the same principle, we can list all public fields of the class String is by using the `getFields()` method call, for example:

```
final Field[] fields = String.class.getFields();
for( final Field field: fields ) {
    System.out.println( field.getName() );
}
```

Continuing the experimentation with the String class using reflection, let us try to create a new instance and call the `length()` method on it, all that by using the **reflection API** only.

```
final Constructor< String > constructor = String.class.getConstructor( String.class );
final String str = constructor.newInstance( "sample string" );
```

```
final Method method = String.class.getMethod( "length" );
final int length = ( int )method.invoke( str );
// The length of the string is 13 characters
```

Probably the most demanded use cases for **reflection** revolve around annotation processing. Annotations by themselves (excluding the ones from Java standard library) do not have any effect on the code. However, Java applications can use **reflection** to inspect the annotations present on the different Java elements of their interest at runtime and apply some logic depending on annotation and its attributes. For example, let us take a look on the way to introspect if the specific annotation is present on a class definition:

```
@Retention( RetentionPolicy.RUNTIME )
@Target( ElementType.TYPE )
public @interface ExampleAnnotation {
    // Some attributes here
}

@ExampleAnnotation
public class ExampleClass {
    // Some getter and setters here
}
```

Using the **reflection API**, it could be easily done using the `getAnnotation()` method call. The returned non-null value indicates that annotation is present, for example:

```
final ExampleAnnotation annotation =
    ExampleClass.class.getAnnotation( ExampleAnnotation.class );

if( annotation != null ) {
    // Some implementation here
}
```

Most of the Java APIs nowadays are including annotations to facilitate their usage and integration for developers. The recent versions of the very popular Java specifications like **Java API for RESTful Web Services** (https://jcp.org/en/jsr/detail?id=339), **Bean Validation** (https://jcp.org/en/jsr/detail?id=349), **Java Temporary Caching API** (https://jcp.org/en/jsr/detail?id=107), **Java Message Service** (https://jcp.org/en/jsr/detail?id=343), **Java Persistence** (https://jcp.org/en/jsr/detail?id=338) and many, many more are built on top of annotations and their implementations usually heavily use the **Reflection API** to gather metadata about the application being run.

11.3 Accessing generic type parameters

Since the introduction of generics (generics are covered in **part 4** of the tutorial, **How and When To Use Generics**), the **Reflection API** has been extended to support the introspection of generic types. The use case which often pops up in many different applications is to figure out the type of the generic parameters that particular class, method or other element has been declared with. Let us take a look on the example class declaration:

```
public class ParameterizedTypeExample {
    private List< String > strings;

    public List< String > getStrings() {
        return strings;
    }
}
```

Now, while inspecting the class using **Reflection** it would be very handy to know that the `strings` property is declared as generic type `List` with `String` type parameter. The code snippet below illustrates how it could be done:

```
final Type type = ParameterizedTypeExample.class
    .getDeclaredField( "strings" ).getGenericType();
```

```
if( type instanceof ParameterizedType ) {
    final ParameterizedType parameterizedType = ( ParameterizedType )type;
    for( final Type typeArgument: parameterizedType.getActualTypeArguments() ) {
        System.out.println( typeArgument );
    }
}
```

The following generic type parameter will be printed on the console:

```
class java.lang.String
```

11.4 Reflection API and visibility

In the **part 1** of the tutorial, **How to create and destroy objects**, we have met first time the accessibility an visibility rules which are supported by the Java language. It may come up as a surprise, but the **reflection API** is able to modify, in a certain way, the visibility rules on a given class member.

Let us take a look on the following example of the class with a single private field name. The getter to this filed is provided, but the setter is not, and this is by intention.

```
public static class PrivateFields {
    private String name;

    public String getName() {
        return name;
    }
}
```

Obviously, it is apparent for any Java developer that the `name` field cannot be set using Java language syntax constructs as the class does not provide the way to do that. **Reflection API** on the rescue, let see how it could be done by changing field's visibility and accessibility scope.

```
final PrivateFields instance = new PrivateFields();
final Field field = PrivateFields.class.getDeclaredField( "name" );
field.setAccessible( true );
field.set( instance, "sample name" );
System.out.println( instance.getName() );
```

The following output will be printed on the console:

```
sample name
```

Please notice that without the `field.setAccessible( true )` call, the exception will be raised at runtime saying that the member of class with modifiers `private` cannot be accessed.

This feature of the **reflection API** is often used by test scaffolding or dependency injection frameworks in order to get access to the intrinsic (or not exposable) implementation details. Please, try to avoid using it in your applications unless you really have no other choice.

11.5 Reflection API pitfalls

Also, be aware that even though the **reflection API** is very powerful, it has a few pitfalls. First of all, it is a subject of security permissions and may not be available in all environments your code is running on. Secondly, it could have a performance impact on your applications. From execution prospective, the calls to **reflection API** are quite expensive.

Lastly, **reflection API** does not provide enough type safety guarantees, forcing developers to use Object instances in most of the places, and is quite limited in transforming constructor / method arguments or method return values.

Since the Java 7 release, there are a couple of useful features which could provide much faster, alternative way to access some functionality used to be available only through **reflection** calls. The next section will introduce you to them.

11.6 Method Handles

The Java 7 release introduced a new very important feature into the JVM and Java standard library - method handles. Method handle is a typed, directly executable reference to an underlying method, constructor or field (or similar low-level operation) with optional transformations of arguments or return values. They are in many respects better alternative to method invocations performed using the **Reflection API**. Let us take a look on a code snippet which uses method handles to dynamically invoke the method `length()` on the `String` class.

```
final MethodHandles.Lookup lookup = MethodHandles.lookup();
final MethodType methodType = MethodType.methodType( int.class );
final MethodHandle methodHandle =
    lookup.findVirtual( String.class, "length", methodType );
final int length = ( int )methodHandle.invokeExact( "sample string" );
// The length of the string is 13 characters
```

The example above is not very complicated and just outlines the basic idea of what method handles are capable of. Please compare it with the same implementation which uses the **Reflection API** from Reflection API section. However it does look a bit more verbose but from the performance and type safety prospective method handles are better alternative.

Method handles are very powerful tool and they build a necessary foundation for effective implementation of dynamic (and scripting) languages on the JVM platform. In the **part 12** of the tutorial, **Dynamic languages support**, we are going to look on couple of those languages.

11.7 Method Argument Names

The well-known issue which Java developers have been facing for years is the fact that method argument names are not preserved at runtime and were wiped out completely. Several community projects, like for example **Paranamer** (https://github.com/paul-hammant/paranamer), tried to solve this issue by injecting some additional metadata into the generated byte code. Luckily, Java 8 changed that by introducing new compiler argument `-parameters` which injects the exact method argument names into the byte code. Let us take a look on the following method:

```
public static void performAction( final String action, final Runnable callback ) {
        // Some implementation here
}
```

In the next step, let us use the **Reflection API** to inspect method argument names for this method and make sure they are preserved:

```
final Method method = MethodParameterNamesExample.class
    .getDeclaredMethod( "performAction", String.class, Runnable.class );
Arrays.stream( method.getParameters() )
    .forEach( p -> System.out.println( p.getName() ) );
```

With the `-parameters` compiler option specified, the following argument names will be printed on the console:

```
action
callback
```

This very long-awaited feature is really a great relief for developers of many Java libraries and frameworks. From now on, much more pieces of useful metadata could be extracted using just pure Java **Reflection API** without a need to introduce any additional workarounds (or hacks).

11.8 Download the Source Code

This was a lesson of **Reflection**, part 11 of **Advanced Java course**. You may download the source code here: **advanced-java-part-11**

11.9 What's next

In this part of the tutorial we have covered **reflection API**, which is the way to inspect your code, extract useful metadata out of it or even modify it. Despite all its drawbacks, **reflection API** is very widely used in most (if not all) Java applications these days. In next part of the tutorial we are going to talk about scripting and dynamic languages support in Java.

Chapter 12

Dynamic languages support

12.1 Introduction

In this part of the tutorial our attention will be fully concentrated on the scripting and dynamic languages support in Java. Since Java 7, the JVM has a direct support of modern dynamic (also often called scripting) languages and the Java 8 release delivered even more enhancements into this space.

One of the strength of the dynamic languages is that the behavior of the program is defined at runtime, rather than at compile time. Among those languages, **Ruby** (https://www.ruby-lang.org/en/), **Python** (https://www.python.org/) and **JavaScript** (http://en.wikipedia.org/wiki/JavaScript) have gained a lot of popularity and are the most widely used ones at the moment. We are going to take a look on how Java scripting API opens a way to integrate those languages into existing Java applications.

12.2 Dynamic Languages Support

As we already know very well, Java is a statically typed language. This means that all typed information for class, its members, method parameters and return values is available at compile time. Using all this details, the Java compiler emits strongly typed byte code which can then be efficiently interpreted by the JVM at runtime.

However, dynamic languages perform type checking at runtime, rather than compile time. The challenge of dealing with dynamically languages is how to implement a runtime system that can choose the most appropriate implementation of a method to call after the program has been compiled.

For a long time, the JVM had had no special support for dynamically typed languages. But Java 7 release introduced the new **invokedynamic** instruction that enabled the runtime system (JVM) to customize the linkage between a call site (the place where method is being called) and a method implementation. It really opened a door for effective dynamic languages support and implementations on JVM platform.

12.3 Scripting API

As part of the Java 6 release back in 2006, the new scripting API has been introduced under the `javax.script` package. This extensible API was designed to plug in mostly any scripting language (which provides the script engine implementation) and run it on JVM platform.

Under the hood, the Java scripting API is really small and quite simple. The initial step to begin working with scripting API is to create new instance of the `ScriptEngineManager` class. ScriptEngineManager provides the capability to discover and retrieve available scripting engines by their names from the running application classpath.

Each scripting engine is represented using a respective `ScriptEngine` implementation and essentially provides the ability to execute the scripts using `eval()` functions family (which has multiple overloaded versions). Quite a number of popular scripting (dynamic) languages already provide support of the Java scripting API and in the next sections of this tutorial we will see how nice this API works in practice by playing with **JavaScript**, **Groovy**, **Ruby/JRuby** and **Python/Jython**.

12.4 JavaScript on JVM

It is not by accident that we are going to start our journey with the JavaScript language as it was the one of the first scripting languages supported by the Java standard library scripting API. And also because, by and large, it is the single programming language every web browser understands.

In its simplest form, the `eval()` function executes the script, passed to it as a plain Java string. The script has no state shared with the evaluator (or caller) and is self-contained piece of code. However, in typical real-world applications it is quite rare and more often than not some variables or properties are required to be provided to the script in order to perform some meaningful calculations or actions. With that being said, let us take a look on a quick example evaluating real JavaScript function call using simple variable bindings:

```
final ScriptEngineManager factory = new ScriptEngineManager();
final ScriptEngine engine = factory.getEngineByName( "JavaScript" );

final Bindings bindings = engine.createBindings();
bindings.put( "str", "Calling JavaScript" );
bindings.put( "engine", engine );

engine.eval( "print(str + ' from ' + engine.getClass().getSimpleName() )", bindings );
```

Once executed, the following output will be printed on the console:

```
Calling JavaScript from RhinoScriptEngine
```

For quite a while, **Rhino** used to be the single JavaScript scripting engine, available on JVM. But the Java 8 release brought a brand-new implementation of JavaScript scripting engine called **Nashorn** (http://www.oracle.com/technetwork/articles/java/-jf14-nashorn-2126515.html).

From the API standpoint, there are not too many differences however the internal implementation differs significantly, promising much better performance. Here is the same example rewritten to use **Nashorn** JavaScript engine:

```
final ScriptEngineManager factory = new ScriptEngineManager();
final ScriptEngine engine = factory.getEngineByName( "Nashorn" );

final Bindings bindings = engine.createBindings();
bindings.put( "engine", engine );

engine.eval( "print(str + ' from ' + engine.getClass().getSimpleName() )", bindings );
```

The following output will be printed on the console (please notice a different script engine implementation this time):

```
Calling JavaScript from NashornScriptEngine
```

Nonetheless, the examples of JavaScript code snippets we have looked at are quite trivial. You could actually evaluate whole JavaScript files using overloaded eval() function call and implement quite sophisticated algorithms, purely in JavaScript. In the next sections we are going to see such examples while exploring other scripting languages.

12.5 Groovy on JVM

Groovy (http://groovy.codehaus.org) is one of the most successful dynamic languages for the JVM platform. It is often used side by side with Java, however it also provides the Java scripting API engine implementation and could be used in a similar way as a JavaScript one.

Let us make this **Groovy** example a bit more meaningful and interesting by developing a small standalone script which prints out on the console some details about every book from the collection shared with it by calling Java application.

The Book class is quite simple and has only two properties, author and title:

```
public class Book {
    private final String author;
    private final String title;

    public Book(final String author, final String title) {
        this.author = author;
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public String getTitle() {
        return title;
    }
}
```

The **Groovy** script (named just **script.groovy**) uses some nifty language features like closures and string interpolation to output the book properties to the console:

```
books.each {
    println "Book '$it.title' is written by $it.author"
}

println "Executed by ${engine.getClass().simpleName}"
println "Free memory (bytes): " + Runtime.getRuntime().freeMemory()
```

Now let us execute this **Groovy** script using Java scripting API and predefined collection of books (surely, all about **Groovy**):

```
final ScriptEngineManager factory = new ScriptEngineManager();
final ScriptEngine engine = factory.getEngineByName( "Groovy" );

final Collection< Book > books = Arrays.asList(
        new Book( "Venkat Subramaniam", "Programming Groovy 2" ),
        new Book( "Ken Kousen", "Making Java Groovy" )
    );

final Bindings bindings = engine.createBindings();
bindings.put( "books", books );
bindings.put( "engine", engine );

try( final Reader reader = new InputStreamReader(
        Book.class.getResourceAsStream("/script.groovy" ) ) ) {
    engine.eval( reader, bindings );
}
```

Please notice that the **Groovy** scripting engine has a full access to Java standard library and does not require any addition bindings. To confirm that, the last line from the **Groovy** script above accesses current runtime environment by calling the `Runtime.getRuntime()` static method and prints out the amount of free heap available to running JVM (in bytes). The following sample output is going to appear on the console:

```
Book 'Programming Groovy 2' is written by Venkat Subramaniam
Book 'Making Java Groovy' is written by Ken Kousen
Executed by GroovyScriptEngineImpl
Free memory (bytes): 153427528
```

It has been 10 years since **Groovy** was introduced. It quickly became very popular because of the innovative language features, similar to Java syntax and great interoperability with existing Java code. It may look like introduction of lambdas and Stream API in Java 8 has made **Groovy** a bit less appealing choice, however it is still widely used by Java developers.

12.6 Ruby on JVM

Couple of years ago **Ruby** (https://www.ruby-lang.org/en/) was the most popular dynamic language used for web application development. Even though its popularity has somewhat shaded away nowadays, **Ruby** and its ecosystem brought a lot of innovations into modern web applications development, inspiring the creation and evolution of many other programming languages and frameworks.

JRuby (http://jruby.org/) is an implementation of the **Ruby** programming language for JVM platform. Similarly to **Groovy**, it also provides great interoperability with existing Java code preserving the beauty of the **Ruby** language syntax.

Let us rewrite the **Groovy** script from the Groovy on JVM section in **Ruby** language (with name **script.jruby**) and evaluate it using the Java scripting API.

```
$books.each do |it|
    java.lang.System.out.println( "Book '" + it.title + "' is written by " + it.author )
end

java.lang.System.out.println( "Executed by " + $engine.getClass().simpleName )
java.lang.System.out.println( "Free memory (bytes): " +
        java.lang.Runtime.getRuntime().freeMemory().to_s )
```

The script evaluation codes stays mostly the same, except different scripting engine and the sample books collection, which is now all about **Ruby**.

```
final ScriptEngineManager factory = new ScriptEngineManager();
final ScriptEngine engine = factory.getEngineByName( "jruby" );

final Collection< Book > books = Arrays.asList(
        new Book( "Sandi Metz", "Practical Object-Oriented Design in Ruby" ),
        new Book( "Paolo Perrotta", "Metaprogramming Ruby 2" )
    );

final Bindings bindings = engine.createBindings();
bindings.put( "books", books );
bindings.put( "engine", engine );

try( final Reader reader = new InputStreamReader(
        Book.class.getResourceAsStream("/script.jruby" ) ) ) {
    engine.eval( reader, bindings );
}
```

The following sample output is going to appear on the console:

```
Book 'Practical Object-Oriented Design in Ruby' is written by Sandi Metz
Book 'Metaprogramming Ruby 2' is written by Paolo Perrotta
Executed by JRubyEngine
Free memory (bytes): 142717584
```

As we can figure out from the **JRuby** code snippet above, using the classes from standard Java library is a bit verbose and have to be prefixed by package name (there are some tricks to get rid of that but we are not going in such specific details).

12.7 Python on JVM

Our last but not least example is going to showcase the **Python** (https://www.python.org/) language implementation on JVM platform, which is called **Jython** (http://www.jython.org/).

The **Python** language has gained a lot of traction recently and its popularity is growing every day. It is widely used by the scientific community and has a large set of libraries and frameworks, ranging from web development to natural language processing.

Following the same path as with **Ruby**, we are going to rewrite the example script from Groovy on JVM section using **Python** language (with name **script.py**) and evaluate it using the Java scripting API.

```
from java.lang import Runtime

for it in books:
    print "Book '%s' is written by %s" % (it.title, it.author)

print "Executed by " + engine.getClass().simpleName
print "Free memory (bytes): " + str( Runtime.getRuntime().freeMemory() )
```

Let us instantiate the **Jython** scripting engine and execute the **Python** script above using already familiar Java scripting API.

```
final ScriptEngineManager factory = new ScriptEngineManager();
final ScriptEngine engine = factory.getEngineByName( "jython" );

final Collection< Book > books = Arrays.asList(
        new Book( "Mark Lutz", "Learning Python" ),
        new Book( "Jamie Chan", "Learn Python in One Day and Learn It Well" )
    );

final Bindings bindings = engine.createBindings();
bindings.put( "books", books );
bindings.put( "engine", engine );

try( final Reader reader = new InputStreamReader(
        Book.class.getResourceAsStream("/script.py" ) ) ) {
    engine.eval( reader, bindings );
}
```

The following sample output will be printed out on the console:

```
Book 'Learning Python' is written by Mark Lutz
Book 'Learn Python in One Day and Learn It Well' is written by Jamie Chan
Executed by PyScriptEngine
Free memory (bytes): 132743352
```

The power of **Python** as a programming language is in its simplicity and steep learning curve. With an army of **Python** developers out there, the ability to integrate the **Python** scripting language into your Java applications as some kind of extensibility mechanism may sound like an interesting idea.

12.8 Using Scripting API

The Java **scripting API** is a great way to enrich your Java applications with extensible scripting support, just pick your language. It is also the simplest way to plug in **domain-specific languages** (DSLs) and allows the business experts to express their intentions in the most convenient manner.

The latest changes in the JVM itself (see please Dynamic Languages Support section) made it much friendlier runtime platform for different dynamic (scripting) languages implementations. No doubts, more and more scripting language engines will be available in the future, opening the door to seamless integration with new and existing Java applications.

12.9 Download Code

This was a lesson of **Dynamic Language Support**, part 12 of **Advanced Java course**. You may download the source code here: **advanced-java-part-12**

12.10 What's next

Beginning from this part we are really starting the discussions about advanced concepts of Java as a language and JVM as excellent runtime execution platform. In the next part of the tutorial we are going to look at the Java Compiler API and the Java Compiler Tree API to learn how to manipulate Java sources at runtime.

Chapter 13

Java Compiler API

13.1 Introduction

In this part of the tutorial we are going to take 10000 feet view of the Java Compiler API. This API provides programmatic access to the Java compiler itself and allows developers to compile Java classes from source files on the fly from application code.

More interestingly, we also are going to walk through the Java Compiler Tree API, which provides access to Java syntax parser functionality. By using this API, Java developers have the ability to directly plug into syntax parsing phase and post-analyze Java source code being compiled. It is a very powerful API which is heavily utilized by many static code analysis tools.

The Java Compiler API also supports annotation processing (for more details please refer to **part 5** of the tutorial, **How and when to use Enums and Annotations**, more to come in **part 14** of the tutorial, **Annotation Processors**) and is split between three different packages, shown below.

- **javax.annotation.processing** :Annotation processing.
- **javax.lang.model** : Language model used in annotation processing and Compiler Tree API (including Java language elements, types and utility classes).
- **javax.tools** : Java Compiler API itself.

On the other side, the Java Compiler Tree API is hosted under the `com.sun.source package` and, following Java standard library naming conventions, is considered to be non-standard (proprietary or internal). In general, these APIs are not very well documented or supported and could change any time. Moreover, they are tied to the particular JDK/JRE version and may limit the portability of the applications which use them.

13.2 Java Compiler API

Our exploration will start from the Java Compiler API, which is quite well documented and easy to use. The entry point into Java Compiler API is the ToolProvider class, which allows to obtain the Java compiler instance available in the system (the official documentation is a great starting point to get familiarized with the typical usage scenarios). For example:

```
final JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
for( final SourceVersion version: compiler.getSourceVersions() ) {
    System.out.println( version );
}
```

This small code snippet gets the Java compiler instances and prints out on the console a list of supported Java source versions. For **Java 7** compiler, the output looks like this:

```
RELEASE_3
RELEASE_4
RELEASE_5
RELEASE_6
RELEASE_7
```

It corresponds to more well-known Java version scheme: **1.3**, **1.4**, **5**, **6** and **7**. For **Java 8** compiler, the list of supported versions looks a bit longer:

```
RELEASE_3
RELEASE_4
RELEASE_5
RELEASE_6
RELEASE_7
RELEASE_8
```

Once the instance of Java compiler is available, it could be used to perform different compilation tasks over the set of Java source files. But before that, the set of compilation units and diagnostics collector (to collect all encountered compilation errors) should be prepared. To experiment with, we are going to compile this simple Java class stored in `SampleClass.java` source file:

```
public class SampleClass {
    public static void main(String[] args) {
        System.out.println( "SampleClass has been compiled!" );
    }
}
```

Having this source file created, let us instantiate the diagnostics collector and configure the list of source files (which includes only `SampleClass.java`) to compile.

```
final DiagnosticCollector< JavaFileObject > diagnostics = new DiagnosticCollector<>();
final StandardJavaFileManager manager = compiler.getStandardFileManager(
    diagnostics, null, null );

final File file = new File(
    CompilerExample.class.getResource("/SampleClass.java").toURI() );

final Iterable< ? extends JavaFileObject > sources =
    manager.getJavaFileObjectsFromFiles( Arrays.asList( file ) );
```

Once the preparation is done, the last step is basically to invoke Java compiler task, passing the diagnostics collector and list of source files to it, for example:

```
final CompilationTask task = compiler.getTask( null, manager, diagnostics,
    null, null, sources );
task.call();
```

That is, basically, it. After the compilation task finishes, the `SampleClass.class` should be available in the **target/classes** folder. We can run it to make sure compilation has been performed successfully:

```
java -cp target/classes SampleClass
```

The following output will be shown on the console, confirming that the source file has been properly compiled into byte code:

```
SampleClass has been compiled!
```

In case of any errors encountered during the compilation process, they will become available through the diagnostics collector (by default, any additional compiler output will be printed into `System.err` too). To illustrate that, let us try to compile the sample Java source file which by intention contains some errors (`SampleClassWithErrors.java`):

```
private class SampleClassWithErrors {
    public static void main(String[] args) {
        System.out.println( "SampleClass has been compiled!" );
    }
}
```

The compilation process should fail and error message (including line number and source file name) could be retrieved from diagnostics collector, for example:

```
for( final Diagnostic< ? extends JavaFileObject > diagnostic:
            diagnostics.getDiagnostics() ) {

        System.out.format("%s, line %d in %s",
            diagnostic.getMessage( null ),
            diagnostic.getLineNumber(),
            diagnostic.getSource().getName() );
}
```

Invoking the compilation task on the `SampleClassWithErrors.java` source file will print out on the console the following sample error description:

```
modifier private not allowed here, line 1 in SampleClassWithErrors.java
```

Last but not least, to properly finish up working with the Java Compiler API, please do not forget to close file manager:

```
manager.close();
```

Or even better, always use `try-with-resources` construct (which has been covered in **part 8** of the tutorial, **How and when to use Exceptions**):

```
try( final StandardJavaFileManager manager =
        compiler.getStandardFileManager( diagnostics, null, null ) ) {
    // Implementation here
}
```

In a nutshell, those are typical usage scenarios of Java Compiler API. When dealing with more complicated examples, there is a couple of subtle but quite important details which could speed up the compilation process significantly. To read more about that please refer to the official documentation.

13.3 Annotation Processors

Fortunately, the compilation process is not limited to compilation only. Java Compiler supports annotation processors which could be thought as a compiler plugins. As the name suggests, annotation processors could perform addition processing (usually driven by annotations) over the code being compiled.

In the **part 14** of the tutorial, **Annotation Processors**, we are going to have much more comprehensive coverage and examples of annotation processors. For the moment, please refer to official documentation to get more details.

13.4 Element Scanners

Sometimes it becomes necessary to perform shallow analysis across all language elements (classes, methods/constructors, fields, parameters, variables, ...) during the compilation process. Specifically for that, the Java Compiler API provides the concept of element scanners. The element scanners are built around visitor pattern and basically require the implementation of the single scanner (and visitor). To simplify the implementation, a set of base classes is kindly provided.

The example we are going to develop is simple enough to show off the basic concepts of element scanners usage and is going to count all classes, methods and fields across all compilation units. The basic scanner / visitor implementation extends ElementScanner7 class and overrides only the methods it is interested in:

```
public class CountClassesMethodsFieldsScanner extends ElementScanner7< Void, Void > {
    private int numberOfClasses;
    private int numberOfMethods;
    private int numberOfFields;

    public Void visitType( final TypeElement type, final Void p ) {
        ++numberOfClasses;
        return super.visitType( type, p );
    }

    public Void visitExecutable( final ExecutableElement executable, final Void p ) {
        ++numberOfMethods;
        return super.visitExecutable( executable, p );
    }

    public Void visitVariable( final VariableElement variable, final Void p ) {
        if ( variable.getEnclosingElement().getKind() == ElementKind.CLASS ) {
            ++numberOfFields;
        }

        return super.visitVariable( variable, p );
    }
}
```

Quick note on the element scanners: the family of **ElementScannerX** classes corresponds to the particular Java version. For instance, ElementScanner8 corresponds to **Java 8**, ElementScanner7 corresponds to **Java 7**, ElementScanner6 corresponds to **Java 6**, and so forth. All those classes do have a family of `visitXxx` methods which include:

- `visitPackage` : Visits a package element.
- `visitType` : Visits a type element.
- `visitVariable` : Visits a variable element.
- `visitExecutable` : Visits an executable element.
- `visitTypeParameter` : Visits a type parameter element.

One of the ways to invoke the scanner (and visitors) during the compilation process is by using the annotation processor. Let us define one by extending AbstractProcessor class (please notice that annotation processors are also tight to particular Java version, in our case **Java 7**):

```
@SupportedSourceVersion( SourceVersion.RELEASE_7 )
@SupportedAnnotationTypes( "*" )
public class CountElementsProcessor extends AbstractProcessor {
    private final CountClassesMethodsFieldsScanner scanner;

    public CountElementsProcessor( final CountClassesMethodsFieldsScanner scanner ) {
        this.scanner = scanner;
    }

    public boolean process( final Set< ? extends TypeElement > types,
            final RoundEnvironment environment ) {

        if( !environment.processingOver() ) {
            for( final Element element: environment.getRootElements() ) {
                scanner.scan( element );
            }
        }

        return true;
```

```
    }
}
```

Basically, the annotation processor just delegates all the hard work to the scanner implementation we have defined before (in the **part 14** of the tutorial, **Annotation Processors**, we are going to have much more comprehensive coverage and examples of annotation processors).

The `SampleClassToParse.java` file is the example which we are going to compile and count all classes, methods/constructors and fields in:

```
public class SampleClassToParse {
    private String str;

    private static class InnerClass {
        private int number;

        public void method() {
            int i = 0;

            try {
                // Some implementation here
            } catch( final Throwable ex ) {
                // Some implementation here
            }
        }
    }

    public static void main( String[] args ) {
        System.out.println( "SampleClassToParse has been compiled!" );
    }
}
```

The compilation procedure looks exactly like we have seen in the Java Compiler API section. The only difference is that compilation task should be configured with annotation processor instance(s). To illustrate that, let us take a look on the code snippet below:

```
final CountClassesMethodsFieldsScanner scanner = new CountClassesMethodsFieldsScanner();
final CountElementsProcessor processor = new CountElementsProcessor( scanner );

final CompilationTask task = compiler.getTask( null, manager, diagnostics,
    null, null, sources );
task.setProcessors( Arrays.asList( processor ) );
task.call();

System.out.format( "Classes %d, methods/constructors %d, fields %d",
    scanner.getNumberOfClasses(),
    scanner.getNumberOfMethods(),
    scanner.getNumberOfFields() );
```

Executing the compilation task against `SampleClassToParse.java` source file will output the following message in the console:

```
Classes 2, methods/constructors 4, fields 2
```

It makes sense: there are two classes declared, `SampleClassToParse` and `InnerClass`. `SampleClassToParse` class has `default constructor` (defined implicitly), method `main` and field `str`. In turn, `InnerClass` class also has `default constructor` (defined implicitly), method `method` and field `number`.

This example is very naive but its goal is not to demonstrate something fancy but rather to introduce the foundational concepts (the **part 14** of the tutorial, **Annotation Processors**, will include more complete examples).

13.5 Java Compiler Tree API

Element scanners are quite useful but the details they provide access to are quite limited. Once in a while it becomes necessary to parse Java source files into abstract syntax trees (or AST) and perform more deep analysis. Java Compiler Tree API is the tool we need to make it happen. Java Compiler Tree API works closely with Java Compiler API and uses `javax.lang.model` package.

The usage of Java Compiler Tree API is very similar to the element scanners from section Element Scanners and is built following the same patterns. Let us reuse the sample source file `SampleClassToParse.java` from Element Scanners section and count how many empty `try/catch` blocks are present across all compilation units. To do that, we have to define tree path scanner (and visitor), similarly to element scanner (and visitor) by extending TreePathScanner base class.

```
public class EmptyTryBlockScanner extends TreePathScanner< Object, Trees > {
    private int numberOfEmptyTryBlocks;

    @Override
    public Object visitTry(final TryTree tree, Trees trees) {
        if( tree.getBlock().getStatements().isEmpty() ){
            ++numberOfEmptyTryBlocks;
        }

        return super.visitTry( tree, trees );
    }

    public int getNumberOfEmptyTryBlocks() {
        return numberOfEmptyTryBlocks;
    }
}
```

The number of `visitXxx` methods is significantly richer (around 50 methods) comparing to element scanners and covers all Java language syntax constructs. As with element scanners, one of the ways to invoke tree path scanners is also by defining dedicate annotation processor, for example:

```
@SupportedSourceVersion( SourceVersion.RELEASE_7 )
@SupportedAnnotationTypes( "*" )
public class EmptyTryBlockProcessor extends AbstractProcessor {
    private final EmptyTryBlockScanner scanner;
    private Trees trees;

    public EmptyTryBlockProcessor( final EmptyTryBlockScanner scanner ) {
        this.scanner = scanner;
    }

    @Override
    public synchronized void init( final ProcessingEnvironment processingEnvironment ) {
        super.init( processingEnvironment );
        trees = Trees.instance( processingEnvironment );
    }

    public boolean process( final Set< ? extends TypeElement > types,
            final RoundEnvironment environment ) {

        if( !environment.processingOver() ) {
            for( final Element element: environment.getRootElements() ) {
                scanner.scan( trees.getPath( element ), trees );
            }
        }

        return true;
    }
}
```

The initialization procedure became a little bit more complex as we have to obtain the instance of Trees class and convert each element into tree path representation. At this moment, the compilation steps should look very familiar and be clear enough. To make it a little bit more interesting, let us run it against all source files we have experimenting with so far: `SampleClassToParse.java` and `SampleClass.java`.

```
final EmptyTryBlockScanner scanner = new EmptyTryBlockScanner();
final EmptyTryBlockProcessor processor = new EmptyTryBlockProcessor( scanner );

final Iterable<? extends JavaFileObject> sources = manager.getJavaFileObjectsFromFiles(
    Arrays.asList(
        new File(CompilerExample.class.getResource("/SampleClassToParse.java").toURI()),
        new File(CompilerExample.class.getResource("/SampleClass.java").toURI())
    )
);

final CompilationTask task = compiler.getTask( null, manager, diagnostics,
    null, null, sources );
task.setProcessors( Arrays.asList( processor ) );
task.call();

System.out.format( "Empty try/catch blocks: %d", scanner.getNumberOfEmptyTryBlocks() );
```

Once run against multiple source files, the code snippet above is going to print the following output in the console:

```
Empty try/catch blocks: 1
```

The Java Compiler Tree API may look a little bit low-level and it certainly is. Plus, being an internal API, it does not have well-supported documentation available. However, it gives the full access to the abstract syntax trees and it is a life-saver when you need to perform deep source code analysis and post-processing.

13.6 Download

This was a lesson for the Java Compiler API, part 13 of Advanced Java Course. You can download the source code of the lesson here: **advanced-java-part-13**

13.7 What's next

In this part of the tutorial we have looked at programmatic access to Java Compiler API from within the Java applications. We also dug deeper, touched annotation processors and uncovered Java Compiler Tree API which provides the full access to abstract syntax trees of the Java source files being compiled (compilation units). In the next part of the tutorial we are going to continue in the same vein and take a closer look on annotation processors and their applicability.

Chapter 14

Java Annotation Processors

14.1 Introduction

In this part of the tutorial we are going to demystify the magic of annotation processing, which is often used to inspect, modify or generate source code, driven only by annotations. Essentially, annotation processors are some kind of plugins of the Java compiler. Annotation processors used wisely could significantly simplify the life of Java developers so that is why they are often bundled with many popular libraries and frameworks.

Being compiler plugins also means that annotation processors are a bit low-level and highly depend on the version of Java. However, the knowledge about annotations from the **part 5** of the tutorial **How and when to use Enums and Annotations** and Java Compiler API from the **part 13** of the tutorial, **Java Compiler API**, is going to be very handy in the understanding of intrinsic details of how the annotation processors work.

14.2 When to Use Annotation Processors

As we briefly mentioned, annotations processors are typically used to inspect the codebase against the presence of particular annotations and, depending on use case, to:

- generate a set of source or resource files
- mutate (modify) the existing source code
- analyze the exiting source code and generate diagnostic messages

The usefulness of annotation processors is hard to overestimate. They can significantly reduce the amount of code which developers have to write (by generating or modifying one), or, by doing static analysis, hint the developers if the assumptions expressed by a particular annotation are not being hold.

Being somewhat invisible to the developers, annotation processors are supported in full by all modern Java IDEs and popular building tools and generally do not require any particular intrusion. In the next section of the tutorial we are going to build own somewhat naÃ¯ve annotation processors which nonetheless will show off the full power of this Java compiler feature.

14.3 Annotation Processing Under the Hood

Before diving into implementation of our own annotation processors, it is good to know the mechanics of that. Annotation processing happens in a sequence of rounds. On each round, an annotation processor might be asked to process a subset of the annotations which were found on the source and class files produced by a prior round.

Please notice that, if an annotation processor was asked to process on a given round, it will be asked to process on subsequent rounds, including the last round, even if there are no annotations for it to process.

In essence, any Java class could become a full-blow annotation processor just by implementing a single interface: javax.annotation.proces
However, to become really usable, each implementation of the javax.annotation.processing.Processor must provide a public no-argument constructor (for more details, please refer to the **part 1** of the tutorial, **How to create and destroy objects**) which could be used to instantiate the processor. The processing infrastructure will follow a set of rules to interact with an annotation processor and the processor must respect this protocol:

- the instance of the annotation processor is created using the no-argument constructor of the processor class
- the `init` method is being called with an appropriate javax.annotation.processing.ProcessingEnvironment instance being passed
- the `getSupportedAnnotationTypes`, `getSupportedOptions`, and `getSupportedSourceVersion` methods are being called (these methods are only called once per run, not on each round)
- and lastly, as appropriate, the process method on the javax.annotation.processing.Processor is being called (please take into account that a new annotation processor instance is not going to be created for each round)

The Java documentation emphasizes that if annotation processor instance is created and used without the above protocol being followed then the processor's behavior is not defined by this interface specification.

14.4 Writing Your Own Annotation Processor

We are going to develop several kinds of annotation processors, starting from the simplest one, immutability checker. Let us define a simple annotation `Immutable` which we are going to use in order to annotate the class to ensure it does not allow to modify its state.

```
@Target( ElementType.TYPE )
@Retention( RetentionPolicy.CLASS )
public @interface Immutable {
}
```

Following the retention policy, the annotation is going to be retained by Java compiler in the class file during the compilation phase however it will not be (and should not be) available at runtime.

As we already know from **part 3** of the tutorial, **How to design Classes and Interfaces**, immutability is really hard in Java. To keep things simple, our annotation processor is going to verify that all fields of the class are declared as final. Luckily, the Java standard library provides an abstract annotation processor, javax.annotation.processing.AbstractProcessor, which is designed to be a convenient superclass for most concrete annotation processors. Let us take a look on **SimpleAnnotationProcessor** annotation processor implementation.

```
@SupportedAnnotationTypes( "com.javacodegeeks.advanced.processor.Immutable" )
@SupportedSourceVersion( SourceVersion.RELEASE_7 )
public class SimpleAnnotationProcessor extends AbstractProcessor {
  @Override
  public boolean process(final Set< ? extends TypeElement > annotations,
      final RoundEnvironment roundEnv) {

    for( final Element element: roundEnv.getElementsAnnotatedWith( Immutable.class ) ) {
      if( element instanceof TypeElement ) {
        final TypeElement typeElement = ( TypeElement )element;

        for( final Element eclosedElement: typeElement.getEnclosedElements() ) {
           if( eclosedElement instanceof VariableElement ) {
           final VariableElement variableElement = ( VariableElement )eclosedElement;

           if( !variableElement.getModifiers().contains( Modifier.FINAL ) ) {
             processingEnv.getMessager().printMessage( Diagnostic.Kind.ERROR,
               String.format( "Class '%s' is annotated as @Immutable,
                 but field '%s' is not declared as final",
```

```
                typeElement.getSimpleName(), variableElement.getSimpleName()
              )
            );
          }
        }
      }
    }

    // Claiming that annotations have been processed by this processor
    return true;
  }
}
```

The `SupportedAnnotationTypes` annotation is probably the most important detail which defines what kind of annotations this annotation processor is interested in. It is possible to use * here to handle all available annotations.

Because of the provided scaffolding, our `SimpleAnnotationProcessor` has to implement only a single method, `process`. The implementation itself is pretty straightforward and basically just verifies if class being processed has any field declared without `final` modifier. Let us take a look on an example of the class which violates this naÃ¯ve immutability contract.

```
@Immutable
public class MutableClass {
    private String name;

    public MutableClass( final String name ) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Running the `SimpleAnnotationProcessor` against this class is going to output the following error on the console:

```
Class 'MutableClass' is annotated as @Immutable, but field 'name' is not declared as final
```

Thus confirming that the annotation processor successfully detected the misuse of `Immutable` annotation on a mutable class.

By and large, performing some introspection (and code generation) is the area where annotation processors are being used most of the time. Let us complicate the task a little bit and apply some knowledge of Java Compiler API from the **part 13** of the tutorial, **Java Compiler API**. The annotation processor we are going to write this time is going to mutate (or modify) the generated bytecode by adding the `final` modifier directly to the class field declaration to make sure this field will not be reassigned anywhere else.

```
@SupportedAnnotationTypes( "com.javacodegeeks.advanced.processor.Immutable" )
@SupportedSourceVersion( SourceVersion.RELEASE_7 )
public class MutatingAnnotationProcessor extends AbstractProcessor {
  private Trees trees;

  @Override
  public void init (ProcessingEnvironment processingEnv) {
    super.init( processingEnv );
    trees = Trees.instance( processingEnv );
  }

  @Override
  public boolean process( final Set< ? extends TypeElement > annotations,
      final RoundEnvironment roundEnv) {

    final TreePathScanner< Object, CompilationUnitTree > scanner =
      new TreePathScanner< Object, CompilationUnitTree >() {
```

```
        @Override
         public Trees visitClass(final ClassTree classTree,
           final CompilationUnitTree unitTree) {

         if (unitTree instanceof JCCompilationUnit) {
           final JCCompilationUnit compilationUnit = ( JCCompilationUnit )unitTree;

           // Only process on files which have been compiled from source
           if (compilationUnit.sourcefile.getKind() == JavaFileObject.Kind.SOURCE) {
             compilationUnit.accept(new TreeTranslator() {
               public void visitVarDef( final JCVariableDecl tree ) {
                 super.visitVarDef( tree );

                 if ( ( tree.mods.flags &amp;amp; Flags.FINAL ) == 0 ) {
                   tree.mods.flags |= Flags.FINAL;
                 }
               }
             });
           }
         }

        return trees;
      }
    };

    for( final Element element: roundEnv.getElementsAnnotatedWith( Immutable.class ) ) {
      final TreePath path = trees.getPath( element );
      scanner.scan( path, path.getCompilationUnit() );
    }

    // Claiming that annotations have been processed by this processor
    return true;
  }
}
```

The implementation became more complex, however many classes (like `TreePathScanner`, `TreePath`) should be already familiar. Running the annotation processor against the same `MutableClass` class will generate following byte code (which could be verified by executing `javap -p MutableClass.class` command):

```
public class com.javacodegeeks.advanced.processor.examples.MutableClass {
  private final java.lang.String name;
  public com.javacodegeeks.advanced.processor.examples.MutableClass(java.lang.String);
  public java.lang.String getName();
}
```

Indeed, the `name` field has `final` modifier present nonetheless it was omitted in the original Java source file. Our last example is going to show off the code generation capabilities of annotation processors (and conclude the discussion). Continuing in the same vein, let us implement an annotation processor which will generate new source file (and new class respectively) by appending `Immutable` suffix to class name annotated with `Immutable` annotation.

```
@SupportedAnnotationTypes( "com.javacodegeeks.advanced.processor.Immutable" )
@SupportedSourceVersion( SourceVersion.RELEASE_7 )
public class GeneratingAnnotationProcessor extends AbstractProcessor {
  @Override
  public boolean process(final Set< ? extends TypeElement > annotations,
      final RoundEnvironment roundEnv) {

    for( final Element element: roundEnv.getElementsAnnotatedWith( Immutable.class ) ) {
      if( element instanceof TypeElement ) {
        final TypeElement typeElement = ( TypeElement )element;
        final PackageElement packageElement =
          ( PackageElement )typeElement.getEnclosingElement();
```

```
        try {
          final String className = typeElement.getSimpleName() + "Immutable";
          final JavaFileObject fileObject = processingEnv.getFiler().createSourceFile(
            packageElement.getQualifiedName() + "." + className);

          try( Writer writter = fileObject.openWriter() ) {
            writter.append( "package " + packageElement.getQualifiedName() + ";" );
            writter.append( "\\n\\n");
            writter.append( "public class " + className + " {" );
            writter.append( "\\n");
            writter.append( "}");
          }
        } catch( final IOException ex ) {
          processingEnv.getMessager().printMessage(Kind.ERROR, ex.getMessage());
        }
      }
    }

    // Claiming that annotations have been processed by this processor
    return true;
  }
}
```

As the result of injecting this annotation processor into compilation process of the `MutableClass` class, the following file will be generated:

```
package com.javacodegeeks.advanced.processor.examples;

public class MutableClassImmutable {
}
```

Nevertheless the source file and its class have been generated using primitive string concatenations (and it fact, this class is really very useless) the goal was to demonstrate how the code generation performed by annotation processors works so more sophisticated generation techniques may be applied.

14.5 Running Annotation Processors

The Java compiler makes it easy to plug any number of annotation processors into the compilation process by supporting **-processor** command line argument. For example, here is one way of running `MutatingAnnotationProcessor` by passing it as an argument of `javac` tool during the compilation of `MutableClass.java` source file:

```
javac -cp processors/target/advanced-java-part-14-java7.processors-0.0.1-SNAPSHOT.jar
  -processor com.javacodegeeks.advanced.processor.MutatingAnnotationProcessor
  -d examples/target/classes
  examples/src/main/java/com/javacodegeeks/advanced/processor/examples/MutableClass.java
```

Compiling just one file does not look very complicated but real-life projects contain thousands of Java source files and using `javac` tool from command line to compile those is just overkill. Likely, the community has developed a lot of great build tools (like Apache Maven, Gradle, sbt, Apache Ant, ...), which take care of invoking Java compiler and doing a lot of other things, so nowadays most of Java project out there use at least one of them. Here, for example, is the way to invoke `MutatingAnnotationProcessor` from Apache Maven build file (**pom.xml**):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>1.7</source>
```

```
    <target>1.7</target>
    <annotationProcessors>
<proc>com.javacodegeeks.advanced.processor.MutatingAnnotationProcessor</proc>
    </annotationProcessors>
  </configuration>
</plugin>
```

14.6 What's next

In this part of the tutorial we have taken a deep look on annotation processors and the ways they help to inspect the source code, mutate (modify) resulting bytecode or generate new Java source files or resources. Annotation processors are very often used to free up Java developers from writing tons of boilerplate code by deriving it from annotations spread across codebase. In the next section of the tutorial we are going to take a look on Java agents and the way to manipulate how JVM interprets bytecode at runtime.

Chapter 15

Java Agents

15.1 Introduction

In this last part of the tutorial we are going to talk about Java agents, a real black magic for regular Java developers out there. Java agents are able to "intrude" into the execution of Java applications running on the JVM at runtime by performing the direct modifications of the bytecode. Java agents are extremely as powerful as dangerous: they can do mostly everything however if something goes wrong, they can easily crash the JVM.

The goal of this part is to demystify Java agents by explaining how they work, how to run them and showing off some simple examples where Java agents come as a clear advantage.

15.2 Java Agent Basics

In its essence, a Java agent is a regular Java class which follows a set of strict conventions. The agent class must implement a `public static void premain(String agentArgs, Instrumentation inst)` method which becomes an agent entry point (similar to the `main` method for regular Java applications).

Once the Java Virtual Machine (JVM) has initialized, each such `premain(String agentArgs, Instrumentation inst)` method of every agent will be called in the order the agents were specified on JVM start. When this initialization step is done, the real Java application `main` method will be called.

However, if the class does not implement `public static void premain(String agentArgs, Instrumentation inst)` method, the JVM will try to look up and invoke another, overloaded version, `public static void premain(String agentArgs)`. Please notice that each `premain` method must return in order for the startup phase to proceed.

Last but not least, the Java agent class may also have a `public static void agentmain(String agentArgs, Instrumentation inst)` or `public static void agentmain(String agentArgs)` methods which are used when the agent is started after JVM startup.

It looks simple on the first glance but there is one more thing which Java agent implementation should provide as part of its packaging: manifest. Manifest files, usually located in the **META-INF** folder and named **MANIFEST.MF**, contain a various metadata related to package distribution.

We have not talked about manifests along this tutorial because most of the time they are not required, however this is not the case with Java agents. The following attributes are defined for Java agents who are packaged as a Java archive (or simply JAR) files:

Manifest Attribute	Description
Premain-Class	When an agent is specified at JVM launch time this attribute defines the Java agent class: the class containing the `premain` method. When an agent is specified at JVM launch time this attribute is required. If the attribute is not present the JVM will abort.
Agent-Class	If an implementation supports a mechanism to start Java agents sometime after the JVM has started then this attribute specifies the agent class: the class containing the `agentmain` method. This attribute is required and the agent will not be started if this attribute is not present.
Boot-Class-Path	A list of paths to be searched by the bootstrap class loader. Paths represent directories or libraries.
Can-Redefine-Classes	A value of `true` or `false`, case-insensitive and defines if the ability to redefine classes needed by this agent. This attribute is optional, the default is `false`.
Can-Retransform-Classes	A value of `true` or `false`, case-insensitive and defines if the ability to retransform classes needed by this agent. This attribute is optional, the default is `false`.
Can-Set-Native-Method-Prefix	A value of `true` or `false`, case-insensitive and defines if the ability to set native method prefix needed by this agent. This attribute is optional, the default is `false`.

Figure 15.1: Java Agent Attributes

For more details please do not hesitate to consult the official documentation dedicated to Java agents and instrumentation.

15.3 Java Agent and Instrumentation

The instrumentation capabilities of Java agents are truly unlimited. Most noticeable ones include but are not limited to:

- Ability to redefine classes at run-time. The redefinition may change method bodies, the constant pool and attributes. The redefinition must not add, remove or rename fields or methods, change the signatures of methods, or change inheritance.
- Ability to retransform classes at run-time. The retransformation may change method bodies, the constant pool and attributes. The retransformation must not add, remove or rename fields or methods, change the signatures of methods, or change inheritance.
- Ability to modify the failure handling of native method resolution by allowing retry with a prefix applied to the name.

Please notice that retransformed or redefined class bytecode is not checked, verified and installed just after the transformations or redefinitions have been applied. If the resulting bytecode is erroneous or not correct, the exception will be thrown and that may crash JVM completely.

15.4 Writing Your First Java Agent

In this section we are going to write a simple Java agent by implementing our own class transformer. With that being said, the only disadvantage working with Java agents is that in order to accomplish any more or less useful transformation, direct bytecode manipulation skills are required. And, unfortunately, theJava standard library does not provide any API (at least, documented ones) to make those bytecode manipulations possible.

To fill this gap, creative Java community came up with a several excellent and at this moment very mature libraries, such as Javassist and ASM, just to name a few. Of those two, Javassist is much simpler to use and that is why it became the one we are going to employ as a bytecode manipulation solution. So far this is a first time we were not able to find the appropriate API in Java standard library and had no choice other than to use the one provided by the community.

The example we are going to work on is rather simple but it is taken from real-world use case. Let us say we would like to capture URL of every HTTP connection opened by the Java applications. There are many ways to do that by directly modifying the Java source code but let us assume that the source code is not available due to license policies or whatnot. The typical example of the class which opens the HTTP connection may look like that:

```
public class SampleClass {
    public static void main( String[] args ) throws IOException {
        fetch("http://www.google.com");
        fetch("http://www.yahoo.com");
    }

    private static void fetch(final String address)
            throws MalformedURLException, IOException {

        final URL url = new URL(address);
        final URLConnection connection = url.openConnection();

        try( final BufferedReader in = new BufferedReader(
                new InputStreamReader( connection.getInputStream() ) ) ) {

            String inputLine = null;
            final StringBuffer sb = new StringBuffer();
            while ( ( inputLine = in.readLine() ) != null) {
                sb.append(inputLine);
            }

            System.out.println("Content size: " + sb.length());
        }
    }
}
```

Java agents fit very well into solving such kind of challenges. We just need to define the transformer which will slightly modify `sun.net.www.protocol.http.HttpURLConnection` constructors by injecting the code to produce output to the console. Sounds scary but with `ClassFileTransformer` and Javassist it is very simple. Let us take a look on such transformer implementation:

```
public class SimpleClassTransformer implements ClassFileTransformer {
  @Override
  public byte[] transform(
      final ClassLoader loader,
      final String className,
      final Class<?> classBeingRedefined,
      final ProtectionDomain protectionDomain,
      final byte[] classfileBuffer ) throws IllegalClassFormatException {

    if (className.endsWith("sun/net/www/protocol/http/HttpURLConnection")) {
      try {
        final ClassPool classPool = ClassPool.getDefault();
        final CtClass clazz =
          classPool.get("sun.net.www.protocol.http.HttpURLConnection");

        for (final CtConstructor constructor: clazz.getConstructors()) {
          constructor.insertAfter("System.out.println(this.getURL());");
        }

        byte[] byteCode = clazz.toBytecode();
        clazz.detach();

        return byteCode;
      } catch (final NotFoundException | CannotCompileException | IOException ex) {
        ex.printStackTrace();
      }
    }

    return null;
  }
```

```
}
```

The `ClassPool` and all `CtXxx` classes (`CtClass`, `CtConstructor`) came from Javassist library. The transformation we have done is quite naÃ¯ve but it is here for demonstrational purposes. Firstly, because we were interested in HTTP communications only, the `sun.net.www.protocol.http.HttpURLConnection` is the class from standard Java library being responsible for that.

Please notice that instead of "." separator, the `className` has the "/" one. Secondly, we looked for `HttpURLConnection` class and modified all its constructors by injecting the `System.out.println(this.getURL());` statement at the end. And lastly, we returned the new bytecode of the transformed version of the class so it is going to be used by JVM instead of original one.

With that, the role of Java agent `premain` method would be just to add the instance of `SimpleClassTransformer` class to the instrumentation context:

```
public class SimpleAgent {
    public static void premain(String agentArgs, Instrumentation inst) {
        final SimpleClassTransformer transformer = new SimpleClassTransformer();
        inst.addTransformer(transformer);
    }
}
```

That's it. It looks quite easy and somewhat frightening at the same time. To finish up with Java agent, we have to supply the proper **MANIFEST.MF** so the JVM will be able to pick the right class. Here is the respective minimum set of the required attributes (please refer to Java Agent Basics section for more details):

```
Manifest-Version: 1.0
Premain-Class: com.javacodegeeks.advanced.agent.SimpleAgent
```

With that, out first Java agents is ready for a real battle. In the next section of the tutorial we are going to cover one of the ways to run Java agent along with your Java applications.

15.5 Running Java Agents

When running from the command line, the Java agent could be passed to JVM instance using `-javaagent` argument which has following semantic:

```
-javaagent:<path-to-jar>[=options]
```

Where `<path-to-jar>` is the path to locate Java agent JAR archive, and options holds additional options which could be passed to the Java agent, more precisely through `agentArgs` argument. For example, the command line for running our Java agent from the section Writing Your First Java Agent (using Java 7 version of it) will look like that (assuming that the agent JAR file is located in the current folder):

```
java -javaagent:advanced-java-part-15-java7.agents-0.0.1-SNAPSHOT.jar
```

When running the `SampleClass` class along with the `advanced-java-part-15-java7.agents-0.0.1-SNAPSHOT.jar` Java agent, the application is going to print on the console all the URLs (Google and Yahoo!) which were attempted to be accessed using HTTP protocol (followed by the content size of the Google and Yahoo! search home web pages respectively):

```
http://www.google.com
Content size: 20349
http://www.yahoo.com
Content size: 1387
```

Running the same `SampleClass` class without Java agent specified is going to output on the console only content size, no URLs (please notice the content size may vary):

```
Content size: 20349
Content size: 1387
```

JVM makes it simple to run Java agents. However, please be warned, any mistakes or inaccurate bytecode generation may crash JVM, possibly losing important data your applications may hold at this moment.

15.6 Download the source code

You can download the source code of this lesson here: **advanced-java-part-15**

15.7 What's next

With this part coming to the end, the advanced Java tutorial is over as well. Hopefully you found it to be useful, practical and entertaining. There are many topics which have not been covered in it but you are very welcome to continue this deep dive into the wonderful world of Java language, platform, ecosystem and incredible community. Good luck!