15BECS404	Advanced Java Programming	L	T	P	C
		3	2	0	4

Course Objectives:

- Understand the concepts of object-oriented, event driven, and concurrent programming paradigms
- Develop skills in using these paradigms using Java.

Learning Outcomes:

To be able to describe & discuss advanced features of Java programming including:

- concurrent object-oriented programming in Java
- event-driven programming
- event handling in the context of Java GUI programming

UNIT I Interfaces and Packages 9

Defining Interfaces-Extending Interfaces-Implementing Interfaces-Accessing Interface Variables-Java API Packages-Using System Packages-Naming Conventions-Creating Packages-Accessing a Package-Using a Package-Adding a Class to a Package-Hiding Classes-Static Import

UNIT II Multithreaded Programming 9

Creating Threads-Extending the Thread Class-Stopping and Blocking a Thread-Life Cycle of a Thread-Using Thread Methods-Thread Exceptions-Thread Priority-Synchronization

UNIT III Managing Errors and Exceptions 9

Implementing the 'Runnable' Interface-Inter thread Communication-Types of Errors-Exceptions-Syntax of Exception Handling Code-Multiple Catch Statements-Using Finally Statement-Throwing Our Own Exceptions-Using Exceptions for Debugging

UNIT IV Applet and Graphics 9

How Applets Differ from Applications-Preparing to Write Applets-Building Applet Code-Applet Life Cycle-Creating an Executable Applet-Designing a Web Page-Applet Tag-Adding Applet to HTML File-Running the Applet-Getting Input from the User-Event Handling-The Graphics Class-Introduction to AWT Package-Introduction to Swings

UNIT V Managing Input/output Files in Java 9

Concept of Streams-Stream Classes-Byte Stream Classes-Character Stream Classes-Using Streams-Other Useful I/O Classes-Using the File Class-Input/Output Exceptions-Creation of Files-

Reading/Writing Characters-Reading/Writing Bytes-Handling Primitive Data Types-Concatenating and Buffering Files-
Random Access Files-Interactive Input and Output-Other Streamclasses

Total Hours: 45+15=60

Text Books:

1. E. Balagurusamy, “Programming with Java”, 4th Edition, Tata Mc Graw Hill, 2010
2. C. Thomas Wu, “An Introduction to Object-Oriented programming with Java”, 5th Edition Tata McGraw-Hill Publishing company Ltd 2010
3. Yashawant Kanetkar, “Let Us Java”, 1st Edition, PBP Publications, 2012

References:

1. Cay S. Horstmann and Gary Cornel, “Core Java: Volume I – Fundamentals”, 8th Edition, Sun Microsystems Press, 2011
2. Timothy Budd “Understanding Object-oriented programming with Java” Pearson Education,2nd edition, 2006
3. Herbert Schildt, “Java The Complete Reference”, Oracle Press, 8th edition, 2011

Websites:

1. http://java.sun.com.

KARPAGAM ACADEMY OF HIGHER EDUCATION
Faculty of Engineering

Department of Computer Science and Engineering

Lecture Plan

Subject Name: Subject Code:

S.No	Topic Name	No.of Periods	Supporting Materials	Teaching Aids
	UNIT- I Interfaces and Packages			
1	Defining Interfaces-Extending Interfaces	1	R[1]-1	BB
2	Implementing Interfaces	1	R[2]-1	BB
3	Accessing Interface Variables	1	R[1]-8	PPT
4	Java API Packages	1	T[1]-6	PPT
5	Using System Packages	1	R[2]-17	PPT
6	Naming Conventions	1	T[1]-45	PPT
7	Creating Packages	1	T[1]-56	BB
8	Accessing a Package	1	T[1]-85	BB
9	Using a Package	1	R[1] 201	PPT

10	Adding a Class to a Package	1	R[2]101	BB
11	Hiding Classes, Static Import	1	T[2]-15	BB
Total		11		
	UNIT- II Multithreaded Programming			
12	Creating Threads	1	R[1]-156	PPT
13	Extending the Thread Class	1	Web	PPT
14	Stopping and Blocking a Thread	1	R[2]140	BB
15	Life Cycle of a Thread	1	R[2]156	PPT
16	Using Thread Methods	1	R[1]214	PPT
17	Thread Exceptions	1	R[2]135	PPT
18	Thread Priority	1	Web	PPT
19	Synchronization	1	R[2]214	BB
		8		
	UNIT- III Managing Errors and Exceptions			
20	Implementing the ‘Runnable’ Interface	1	Web	PPT
21	Inter thread Communication	1	Web	PPT
22	Types of Errors	1	Web	PPT
23	Exceptions	1	T[1]-242	BB
24	Syntax of Exception Handling Code	1	T[1]-245	BB
25	Multiple Catch Statements	1	T[1]-193	PPT
26	Using Finally Statement	1	T[2]-205	BB
27	Throwing Our Own Exceptions	1	R[1]-250	PPT
28	Using Exceptions for Debugging	1	R[1]-285	PPT
Total		9		
	UNIT- IV Applet and Graphics			
29	How Applets Differ from Applications-	1	R[1]-287	PPT
30	Preparing to Write Applets-	1	R[1]-289	PPT
31	Building Applet Code-	1	T[1]-250	PPT
32	Applet Life Cycle-	1	R[1]-291	BB
33	Creating an Executable Applet-	1	R[2]-189	BB
34	Designing a Web Page-	1	R[1]-293	PPT
35	Applet Tag-	1	T[1]-253	BB
36	Adding Applet to HTML File-	1	R[2]-191	BB
37	Running the Applet-	1	T[1]-293	BB
38	Getting Input from the User-	1	T[1]-300	BB
39	Event Handling-	1	T[1]-305	PPT

40	The Graphics Class	1	T[2]-208	PPT
41	-Introduction to AWT Package-	1	T[2]-210	BB
42	Introduction to Swings	1	T[2]-212	BB
Total		14		
	UNIT- V Managing Input/output Files in Java			
43	Concept of Streams	1	R[1]-248	PPT
44	Stream Classes, Byte Stream Classes, Character Stream Classes	1	R[1]-465	BB
45	Using Streams	1	R[1]-465	BB
46	Other Useful I/O Classes, Using the File Class	1	R[1]-255	PPT
47	Input/Output Exceptions	1	R[1]-248	PPT
48	Creation of Files	1	T[1]-519	PPT
49	Reading/Writing Characters, Reading/Writing Bytes, Handling Primitive Data Types	1	T[1]-524	PPT
50	Concatenating and Buffering Files, Random Access Files	1	T[1]-690	BB
51	Interactive Input and Output	1	R[1]-248	BB
52	Other Stream classes	1	R[1]-465	PPT
	Discussion on Previous University Question Papers			
Total		10		
	Total Hours	52		

LECTURE NOTES

UNIT 1

1.1 Introduction

Java programming language, originated in Sun Microsystems and released back in 1995, is one of the most widely used pro- gramming 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 sup- ported 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

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

should be provided.

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

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

one argument.

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

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

public class InitializationBlocks {
```

code. For example:

```
    {
        // 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 counter- part: 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-**

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

resources and AutoCloseable interface which allows to write clean code like this:

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

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

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

once per class-loader. For example:

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

```
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;
    }
}
```

like:

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

```
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;
    }
}
```

fashion) is using the `static final` property of the class.

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

```
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;
    }
}
```

discussing in **part 9** of the tutorial, **Concurrency best practices**).

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

```
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 );
    }
}
```

(**factory method**). For example:

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

```
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 );
    }
}
```

instance itself. Let us consider the following example:

The class `Dependant` needs an instance of DateFormat and it just creates one by calling `DateFormat.getDateInstanc e()` at construction time. The better design would be to use

```
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 );
    }
}
```

constructor argument to do the same thing:

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: com.javacodegeeks.advanced.java

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`.

UNIT II

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

```java
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;
        }
```

simple implementation of the `equals()` method for the `Person` class.

```
        // 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

```
// 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;
}
```

`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.

To protect yourself from surprises, whenever possible try to use `final` fields while implementing `equals()` and `hashCod e()`. 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()`,

```
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;
    }
}
```

`Arrays.toString() /Arrays.deepToString()`. Let us take a look on the `Office` class and its possible `toString()` implementation.

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()` implemen- tations 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 `Clo neable` 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

```
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();
    }
}
```

`super.clone()` first and then perform additional actions if needed. Let us see how it could be implemented for our sample `Person` class.

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.

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

Let us take a look on a simple example which illustrates the differences:

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"

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

string. But in Java it is not the case:

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.

```java
@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.

UNIT III

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 `perfo rmAction`. 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, an- notations (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

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

to section Visibility). As such, the following method declarations are equivalent:

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

```
public interface Cloneable {
}
```

library:

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

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

public interface InterfaceWithDefaultMethods {
    void performAction();

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

example:

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

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

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

    public abstract void performAnotherAction();
}
```

like interfaces do. For example:

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

```
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 `Collecti ons.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

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

on array size and may put a lot of pressure on garbage collector.

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

```
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();
    }
}
```

old-fashioned way to spawn new thread in Java:

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. Com- bined 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.

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

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

    // No one can access it
```

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;

// 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

```
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
    }
}
```

achieve (or mimic) multiple inheritance in Java.

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

```
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
    }
}
```

support in Java.

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

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

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

automatically inheriting these implemented methods as well. For example:

```
    @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

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

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

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

well.

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

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

designed using the inheritance.

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:

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 priv ate`, 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

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

public final class FinalClass {
}
```

`final`.

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

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

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

subclasses.

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.

UNIT IV

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 dec- larations. The interface may be parameterized with more than one type,

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

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

for example:

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

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

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

generic interface:

```
        // 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

```
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;
    }
}
```

their class or interface declaration. For example:

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

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

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

could come as a surprise first that the following code does not compile:

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 )
```

```
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
    }
}
```

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:

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**

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

a generic array of T"):

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

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

example:

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

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

for `extends` keyword, for example:

The bounds are not limited to single constraints and could be combined using the `&amp;` 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

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

replaced by the ? wildcard. For example:

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

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

class. For example:

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

```
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() );
```

limited. For example, the following code does not compile in Java 7:

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

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

generic method could be declared and its type parameter is adorned with annotations:

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

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

modified instance of this class.

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; 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.

UNIT V

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

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

demonstrates.

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,

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

the Java compiler is able to do type checking. For example:

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.

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

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

represents the week day or week-end. For example:

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

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

`DayOfWeek`.

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

instance fields.

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

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

this:

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 in- stances 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 instan- tiated 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 annota- tions 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 de- tails 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, anno- tations 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 appli- cation.

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.

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

@InheritableAnnotation
public class Parent {
}

public class Child extends Parent {
}
```

For example:

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 an- notations 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.

KARPAGAM ACADEMY OF HIGHER EDUCATION
COIMBATORE-21
Faculty of Engineering
Department of Computer Science and Engineering
UNIVERSITY EXAMINATION

Subject Code	**: 15BECS404**
Name of the Course	**: II B.E CSE**
Title of the paper	**: Advanced Java Programming**

Part-A
Answer All Questions (9*2=18)

1. What is origin of Java?
2. What is the command javac is used for?
3. What is a Java Servlet?
4. Which is the root class of all AWT events?
5. State some OOP features.
6. What is Polymorphism?
7. State difference between inheritance and interface.
8. What are the types of inheritance? Which inheritance is not supported by Java?
9. Which method will a web browser call on a new applet?

Part-B
Answer ALL Questions (3*14=42)

10. Explain the concepts of Object Oriented Programming.

OR

11. Compare the lifestyle of CMP and BMP Entity Beans.

12. Explain Android Operating System architecture. Discuss Android features.

OR

13. Explain Struts Validation Framework with a diagram.

14. Write down the steps and code snippets for creating a simple struts application for logging on the site.

OR

15. Explain Android architecture.

ONLINE 1 MARK QUESTIONS

1. Java programs are
A) Faster than others
B) Platform independent
C) Not reusable
D) Not scalable

2. Java has its origin in
A) C programming language
B) PERRL
C) COBOL
D) Oak programming language

3. Which one of the following is true for Java
A) Java is object-oriented and interpreted
B) Java is efficient and faster than C
C) Java is the choice of everyone.
D) Java is not robust.

4. The command javac is used to
A) debug a java program
B) compile a java program
C) interpret a java program
D) execute a java program

5. Java servlets are an efficient and powerful solution for creating ………….. for the web.
A) Dynamic content
B) Static content
C) Hardware
D) Both a and b

6. Filters were officially introduced in the Servlet ……………… specification.
A) 2.1
B) 2.3
C) 2.2
D) 2.4

7. Which is the root class of all AWT events

A) java.awt.ActionEvent
B) java.awt.AWTEvent
C) java.awt.event.AWTEvent
D) java.awt.event.Event

8. OOP features are
i) Increasing productivity ii) Reusability
iii) Decreasing maintenance cost iv) High vulnerability
A) 1,2 & 4
B) 1,2 & 3
C) 1, 2 & 4
D) none of the above

9. break statement is used to
i) get out of method ii) end a program
iii) get out of a loop iv) get out of the system
A) 1 & 2
B) 1,2 & 3
C) 1 & 3
D) 3

10. Native-protocol pure Java converts ……….. into the ………… used by DBMSs directly.
A) JDBC calls, network protocol
B) ODBC class, network protocol
C) ODBC class, user call
D) JDBC calls, user call

11. All Java classes are derived from
A) java.lang.Class
B) java.util.Name
C) java.lang.Object
D) java.awt.Window

12. The jdb is used to
A) Create a jar archive
B) Debug a java program
C) Create a C header file
D) Generate java documentation

13. What would happen if "String[]args" is not included as an argument in the main method?
A) No error
B) Compilation error
C) The program won't run
D) Program exit

14. For the execution of DELETE SQL query in JDBC, …………. method must be used.
A) executeQuery()
B) executeDeleteQuery()
C) executeUpdate()
D) executeDelete()

15. Which method will a web browser call on a new applet?
A) main method
B) destroy method
C) execute method

D) init method

16. Which of the following is not mandatory in a variable declaration?
A) a semicolon
B) an identifier
C) an assignment
D) a data type

17. When a programming class implements an interface, it must provide behavior for
A) two methods defined in that interface
B) any methods in a class
C) only certain methods in that interface
D) all methods defined in that interface

18. In order to run JSP ……………….. is required.
A) Mail Server
B) Applet viewer
C) Java Web Server
D) Database connection

19. State true or false.
i) AWT is an extended version of swing
ii) Paint( ) of Applet class cannot be overridden
A) i-false, ii-false
B) i-false,ii-true
C) i-true, ii-false
D) i-true, ii-true

20. Prepared Statement object in JDBC used to execute……….. queries.
A) Executable
B) Simple
C) High level
D) Parameterized

21. In Java variables, if first increment of the variable takes place and then the assignment occurs. This operation is also called…………………………
A) pre-increment
B) post-increment
C) incrementation
D) pre incrementation

22. When the operators are having the same priority, they are evaluated from …………….. …………. in the order they appear in the expression.

A) right to left
B) left to right
C) any of the order
D) depends on the compiler

23. In Java, …………. can only test for equality, whereas ………… can evaluate any type of Boolean expression.
A) switch, if
B) if, switch
C) if, break
D) continue, if

24. The ………………….. looks only for a match between the value of the expression and one of its case constants.
A) if
B) match
C) switch
D) None of the above

25. System.in.read() is being used, the program must specify the …………….. clause.
A) throws.java.out.IOException
B) throws.java.in.IOException
C) throws.java.io.IOException
D) throws.java.io.InException

26. By using ………………. you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
A) Break
B) Continue
C) Terminate
D) Loop Close

27. The out object is an object encapsulated inside the …………….. class and represents the standard output device.
A) standard
B) local
C) global
D) system

28. The third type of comment is used by a tool called …………….. for automatic generation of documentation.
A) Java commenting
B) Java generator
C) Java doc
D) Java loc

29. In the second type, the information written in java after // is ignored by the …………………..
A) Interpreter
B) Compiler
C) Programmer
D) All of the above

30. The compiled Java program can run on any ………………… platform having Java Virtual Machine (JVM) installed on it.
A) program
B) java
C) hardware
D) nonjava