

Java Advanced

Part I

[Introduction](#)

[History](#)

[Versions](#)

[What's Java used for?](#)

[What technologies a Java developer must know?](#)

[Why is Java important?](#)

[Primary goals of Java language](#)

[Java Architecture](#)

[Garbage Collection](#)

[The Java Programmer Essentials](#)

[Apache Maven](#)

[Generate an initial project](#)

[Build and run the application](#)

[Using Maven](#)

[Import the project in your IDE](#)

[Constructors & Destructors](#)

[Visibility](#)

[Design Classes & Interfaces](#)

[Interfaces](#)

[Inheritance, multiple inheritance](#)

[Encapsulation](#)

[Methods](#)

[Construction patterns](#)

[Factory](#)

[Injection](#)

[Javadoc](#)

[Singleton](#)

[Testing with JUnit](#)

[Exceptions](#)

[Exceptions with JUnit](#)

[Generics](#)

[Arrays](#)

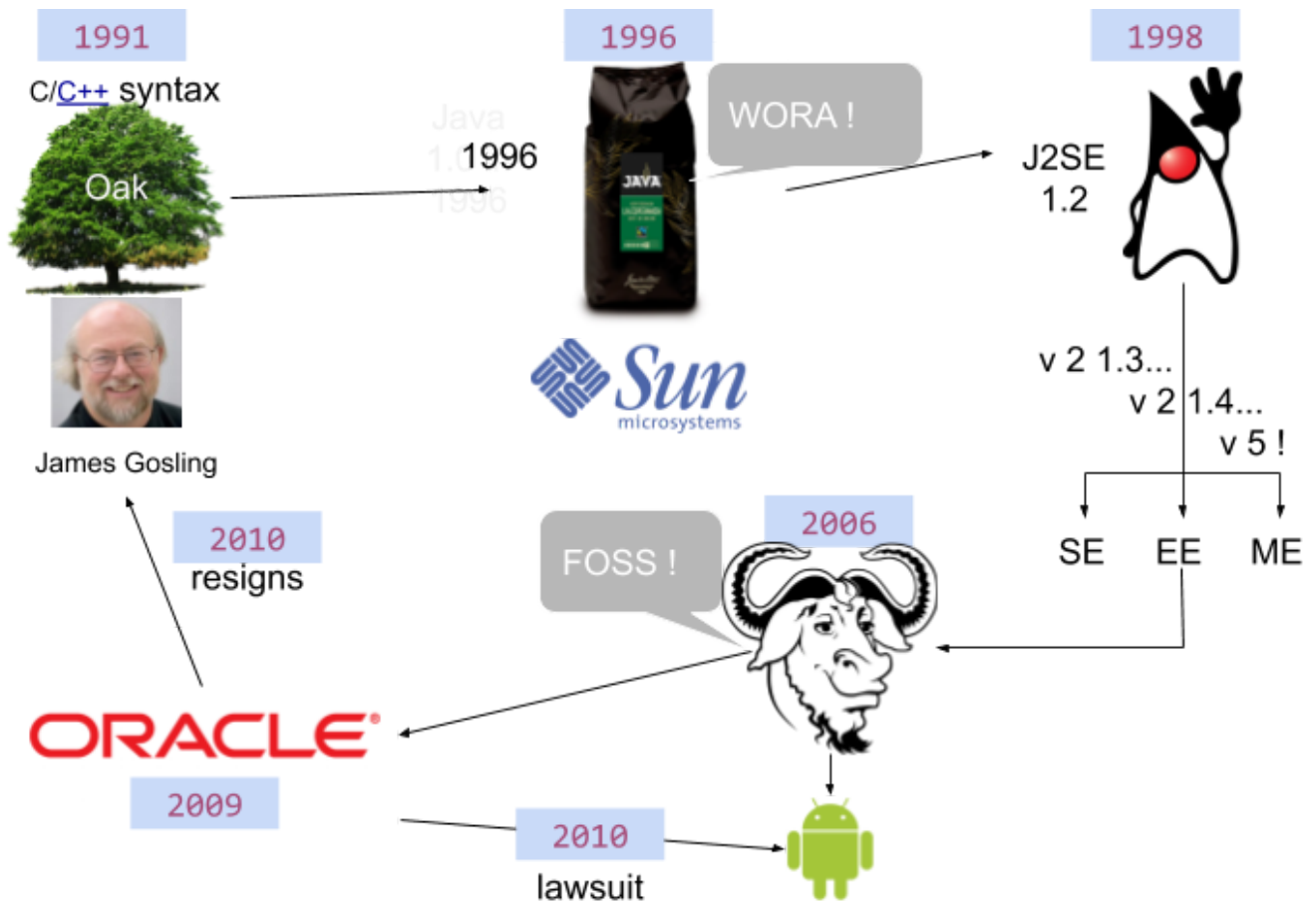
[I/O \(InputStream and OutputStream\)](#)

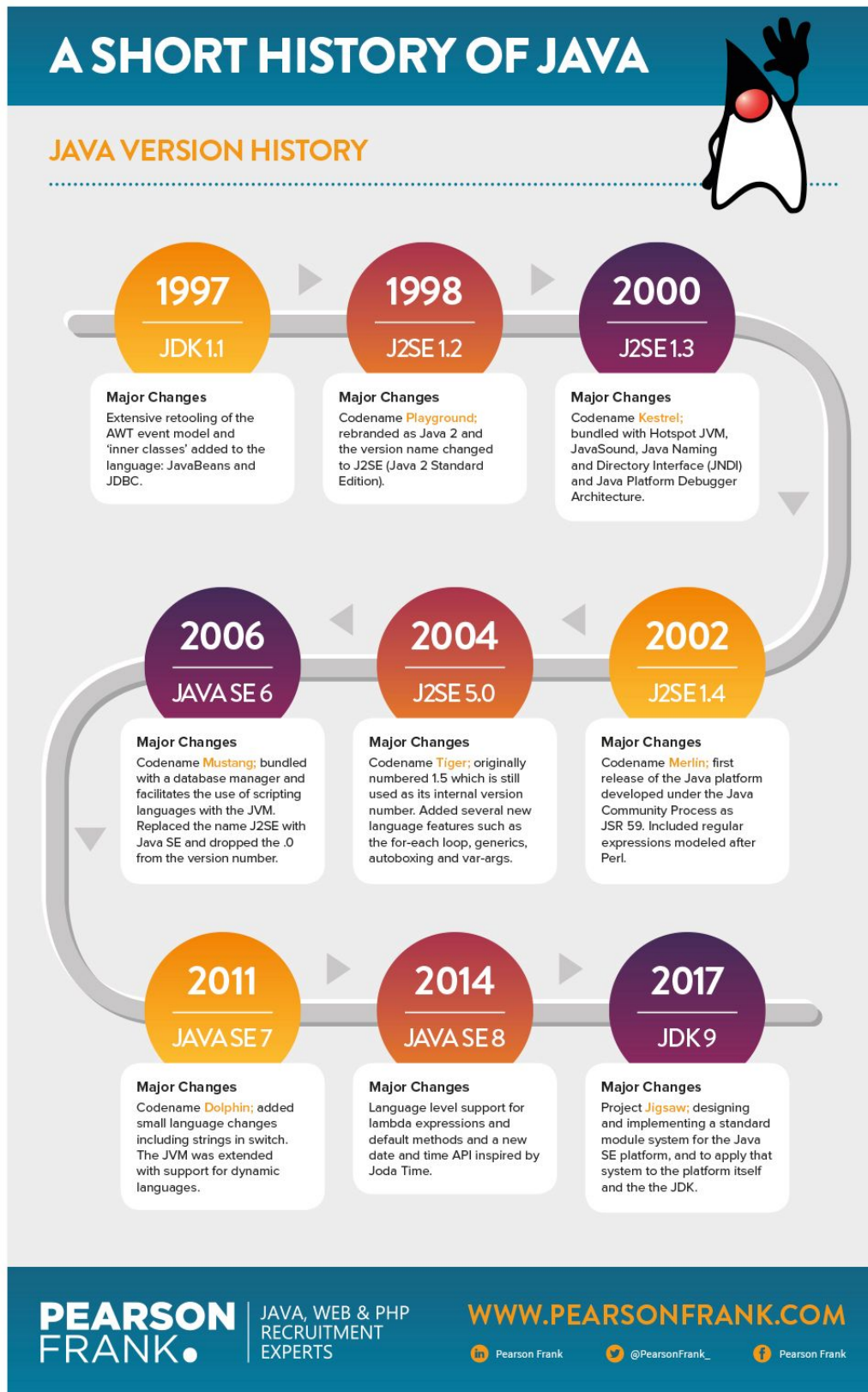
[References](#)



Introduction

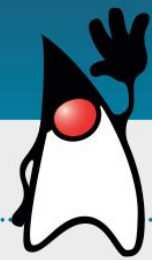
History






What's Java used for?

A SHORT HISTORY OF JAVA




WHAT JAVA IS USED FOR

Java is used in many applications that you are likely to use on a day-to-day basis, from scientific applications to financial applications, from video games to desktop applications.




Mobile Phones

If you have an Android phone you use Java every day! Android apps – and indeed the Android operating system! – are written in Java, with Google's API, which is similar to JDK.




Point of Sale Systems

Java is also used in the creation of PoS systems, helping businesses exchange goods or services for money from their customers.




Video Games

One of the most popular games of all time, Minecraft, was written in Java by Mojang. Minecraft is a sandbox construction game, where you can build anything you can imagine.



Trading Applications

Several third-party trading applications use Java. Murex, which is used by many banks for front to back connectivity, is also written in Java.





Big Data Technologies


The Java platform is very popular in writing high-performance systems. Hadoop and ElasticSearch are both written in Java and are often used in Big Data projects.

PEARSON FRANK. | JAVA, WEB & PHP RECRUITMENT EXPERTS

WWW.PEARSONFRANK.COM

 Pearson Frank

 @PearsonFrank_

 Pearson Frank

([source](#))

Things Java Programmers Should Learn in 2020

- Unit Testing
- RESTful Web Service
- Learn Java Performance Tuning
- Concurrency
- Participate in Open source projects
- Learn Network Programming in Java
- Spring + Spring Security 5.0
- Android
- Git
- Docker and Kubernetes
- Microservices
- Cloud
-

([source](#))

What technologies a Java developer must know?

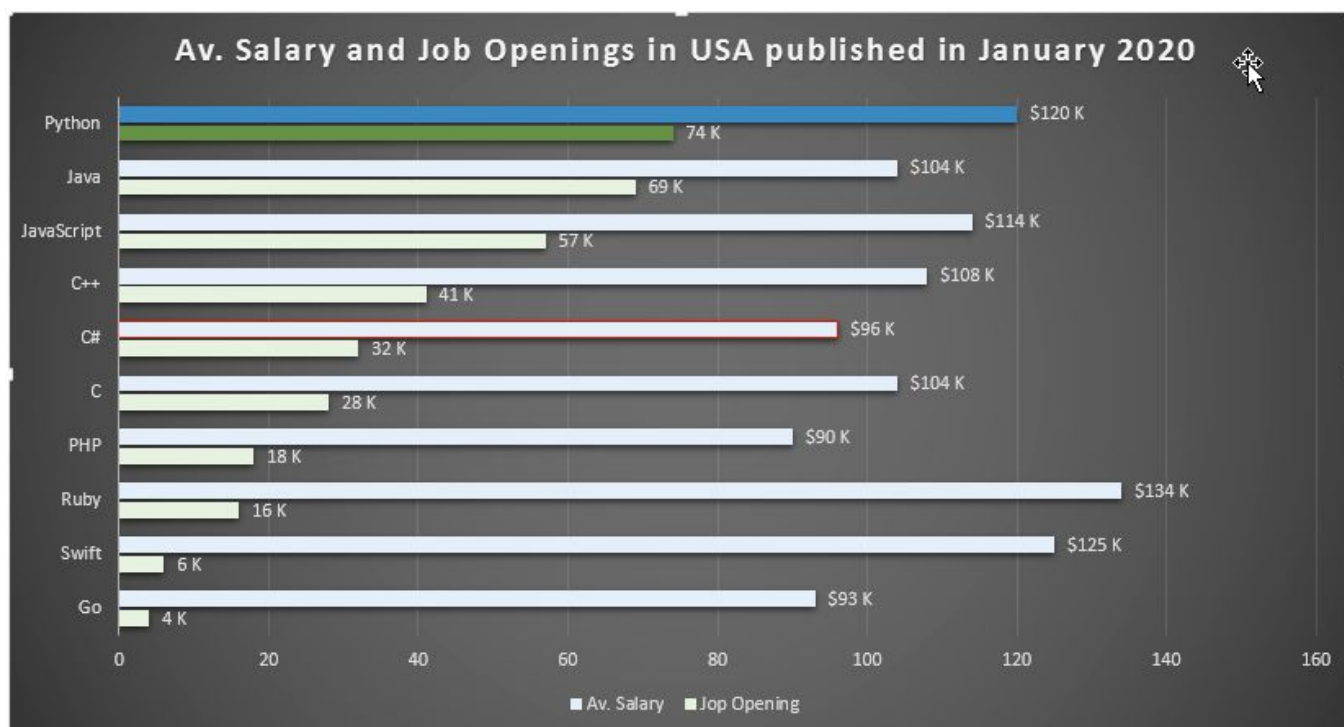
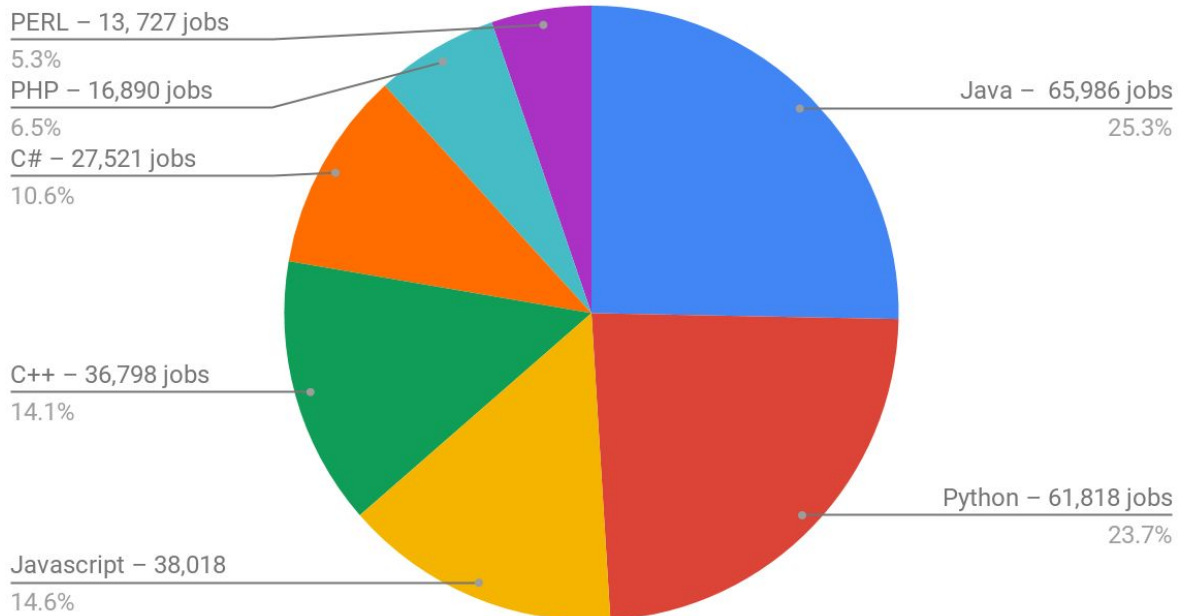
1. At least one MVC Framework like JSF, Struts, or Spring MVC
2. Hibernate or JPA
3. Dependency Injection (as demonstrated in Spring or Java EE through @Resource)
4. SOAP based Web Services (JAX-WS)
5. Some build tool (Ant, Maven, etc.)
6. JUnit (or other Unit Testing framework)
7. Version control - Git
8. JSTL (web dev only)
9. Application server/container configuration management and application deployment (whether it is WebSphere, Tomcat, JBoss, etc. you need to know where your application runs and how to improve its execution).
10. AJAX

([source](#))

Why is Java important?

Top 7 programming languages with most job posting on [Indeed](https://www.indeed.com) as of January 2019:

Jobs



Primary goals of Java language

It must be "simple, object-oriented, and familiar".
It must be "robust and secure".
It must be "architecture-neutral and portable".
It must execute with "high performance".
It must be "interpreted, threaded, and dynamic".

The 5 primary goals in the creation of the Java language

Syntax, Packages, Classes, Objects, Main.

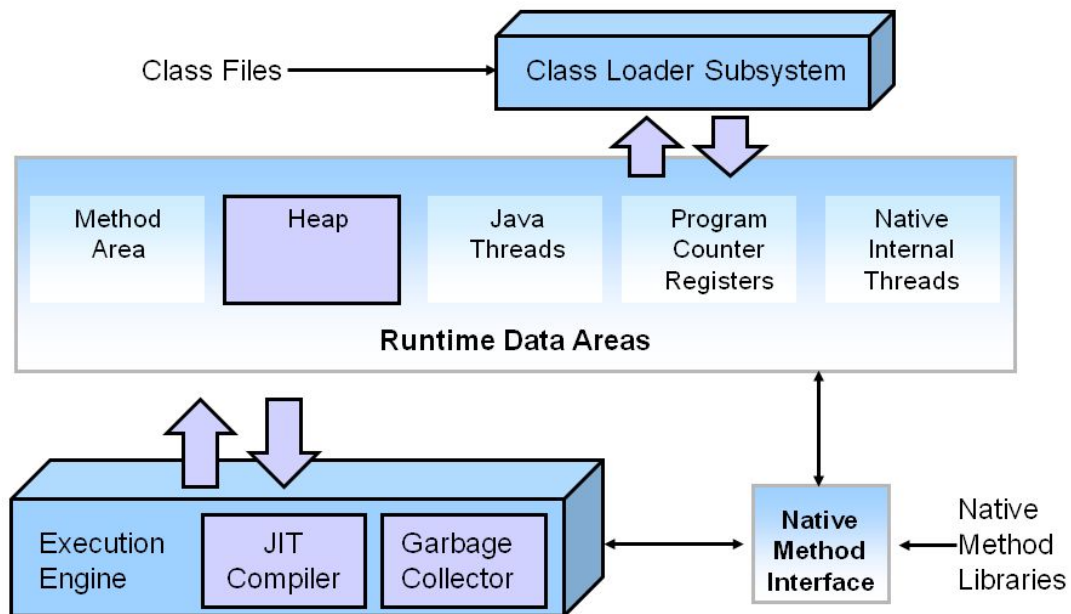
```
package myPackage;

import java.util.Random; // Single type declaration

public class ImportsTest {
    public static void main(String[] args) {
        /* The following line is equivalent to
         * java.util.Random random = new java.util.Random();
         * It would've been incorrect without the import declaration */
        Random random = new Random();
    }
}
```


Java Architecture

Key HotSpot JVM Components



Garbage Collection

Memory allocation and restitution in Java.

Open this link and parse the page with attendees:

<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

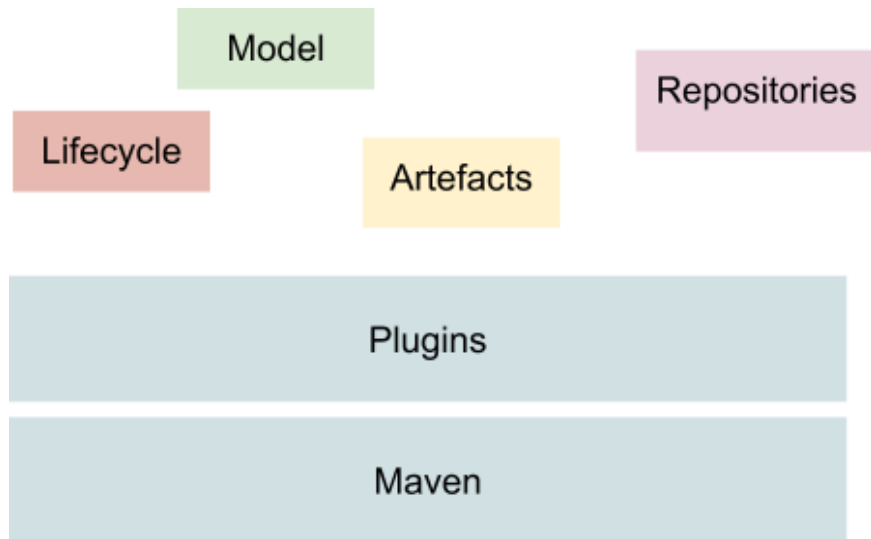
The Java Programmer Essentials

Stackoverflow: <https://stackoverflow.com/questions/tagged/java>

Oracle Javadoc: <https://docs.oracle.com/javase/7/docs/api/>

Apache Maven

Maven: Convention over configuration



Maven uses convention over configuration, this means that you only need to tell Maven the things that are different from the defaults.

For example, the default packaging is jar, so conveniently for us here, as we want to build a .jar file, we don't have to tell Maven what packaging to use.

Generate an initial project

Generate the source files of your project using maven archetypes. Pay attention to your alias, no space, no special characters. Use the first letter of your first name, followed by your name (Example: "John Doe" alias is jdoe).

Follow the instructions on the [Maven in 5 minutes](#) page to install Java and Maven.

Run the command to generate your java project:

```
mvn archetype:generate -DgroupId=isen.m1.[youralias] -DartifactId=my-app  
-DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4  
-DinteractiveMode=false
```

Build and run the application

Using Maven

Go to the newly **my-app** created folder and run the maven build in order to generate the jar application

```
mvn clean package
```

Check that the build is a success:

```
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----
```

The generated jar file should be in the target directory. Now execute the newly created application.

```
java -cp target/my-app-1.0-SNAPSHOT.jar isen.m1.[youralias].App
```

You should see “Hello World!” popping-up.

Publish a documentation of your project:

```
mvn site
```

Then open the generated page located **target/site/index.html** folder.

Import the project in your IDE

Imported the generated maven project into your favorite IDE, unless you don't use an IDE.

In Eclipse: File > Import > Existing maven project

Constructors & Destructors

Default, Arguments, Call to another constructor with **this**. Default values of members.

Exercise: Code a class named App that displays the default value of types: boolean, byte, short, int, long, float, double, object reference.

Resolution: Code written together with the professor.

Visibility

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Design Classes & Interfaces

Interfaces

Interfaces declarations, methods visibility, *default* and *static*.

Default: implements the default behavior of a method. Can be overridden.

```
1 package com.javacodegeeks.advanced.design;
2
3 public interface InterfaceWithDefaultMethods {
4     void performAction();
5
6     default void performDefaultAction() {
7         // Implementation here
8     }
9 }
```

Static: method declared at class level, thus cannot be overridden.

```
1 package com.javacodegeeks.advanced.design;
2
3 public interface InterfaceWithDefaultMethods {
4     void performAction();
5
6     default void performDefaultAction() {
7         // Implementation here
8     }
9 }
```

Abstract classes: cannot be instantiated and may contain abstract methods without implementation.

```

1 package com.javacodegeeks.advanced.design;
2
3 public abstract class SimpleAbstractClass {
4     public void performAction() {
5         // Implementation here
6     }
7
8     public abstract void performAnotherAction();
9 }

```

Inheritance, multiple inheritance

```

public interface OperateCar {

    // constant declarations, if any

    // method signatures

    // An enum with values RIGHT, LEFT
    int turn(Direction direction,
             double radius,
             double startSpeed,
             double endSpeed);
    int changeLanes(Direction direction,
                   double startSpeed,
                   double endSpeed);
    int signalTurn(Direction direction,
                  boolean signalOn);
    int getRadarFront(double distanceToCar,
                    double speedOfCar);
    int getRadarRear(double distanceToCar,
                   double speedOfCar);
    // .....
    // more method signatures
}

```

```

public class OperateBMW760i implements
OperateCar {

    // the OperateCar method signatures,
    with implementation --
    // for example:
    int signalTurn(Direction direction,
                  boolean signalOn) {
        // code to turn BMW's LEFT turn
        indicator lights on
        // code to turn BMW's LEFT turn
        indicator lights off
        // code to turn BMW's RIGHT turn
        indicator lights on
        // code to turn BMW's RIGHT turn
        indicator lights off
    }

    // other members, as needed -- for
    example, helper classes not
    // visible to clients of the interface
}

```

Challenge: What is the output of the following main method ?

```
class A {  
    int num = 1;  
    void sayHello(){  
        System.out.println("Hello, I am A");  
    }  
}  
  
class B extends A {  
    int num = 2;  
    void sayHello(){  
        System.out.println("Hello, I am B");  
    }  
}  
  
public class TestInheritance {  
    public static void main(String[] args) {  
        A ref = new A();  
        ref.sayHello();  
        System.out.println(ref.num);  
  
        ref = new B();  
        ref.sayHello();  
        System.out.println(ref.num);  
    }  
}
```

Why ?

Because variables in Java do not follow polymorphism and overriding is only applicable to methods but not to variables. And when an instance variable in a child class has the same name as an instance variable in a parent class, then the instance variable is chosen from the reference type.

([source](#))

Encapsulation

Encapsulation is the fact of embedding attributes inside a class, without directly exposing the attribute itself.

Methods

Method overloading: provides a specialized method with different arguments.

```
public void setIsbn(String isbn) {  
    this.isbn = isbn;  
}  
  
public void setIsbn(long isbn) {  
    setIsbn(Long.toString(isbn));  
}
```

Method overriding: overwrites a method definition, using the parent version is possible with the keyword *super* (Exercise on overriding will be covered in the Exceptions chapter).

Exercise:

Prerequisite: generate a basic java maven project and import it to your IDE

1/ Under your name package, write the class `Book`, having the following attributes:

`String isbn`, `String title`, `double price`, `java.util.Date issueDate`

Hide the class members from the outside using the appropriate visibility attribute and encapsulate the members by writing the proper set and get methods (please follow java camel case convention).

2/ Write two public constructors for this class, one with no argument, the other having the `title` as an argument.

3/ Write the classes `PaperBook` and `KindleBook` that extends the `Book` class. Implement their two constructors using the `super` keyword.

4/ Write a Class named `BookMain` with a `main` method creating a book by calling the constructor method of Paper and Kindle books with a title, then call `System.out.println` on the book title like this:

```
System.out.println("Book title: "+book.getTitle());
```

Execute the `BookMain` class, see the result in the console output.

Construction patterns

Factory

What is a Factory: a Factory is a class that's purpose is to return new instances of a particular class. The benefits are that:

- 1) The code is more readable
- 2) It allows abstraction of the actual type of the newly created object

```
01 package com.javacodegeeks.advanced.construction.patterns;
02
03 public class Book {
04     private Book( final String title) {
05     }
06
07     public static Book newBook( final String title ) {
08         return new Book( title );
09     }
10 }
```

Exercise:

1/ Based on the above examples, write the interface `BookFactory` having 2 methods:

```
public Book newBook();
public Book newBook(String title);
```

Write 2 implementations of this interface:

`Library` and `KindleLibrary` respectively returning instances of `PaperBook` or `KindleBook`, both classes inheriting from `Book`.

2/ Update the `BookMain` class with the `main` method now creating a book by calling the `newBook` method of `Library` class with a title.

Injection

Dependency injection: When an instance of a class depends on another instance of a class, a good practice is that this dependency is not created by the instance itself. This mechanism is also called "inversion of control".

```
01 package com.javacodegeeks.advanced.construction.patterns;
02
03 import java.text.DateFormat;
04 import java.util.Date;
05
06 public class Dependant {
07     private final DateFormat format;
08
09     public Dependant( final DateFormat format ) {
10         this.format = format;
11     }
12
13     public String format( final Date date ) {
14         return format.format( date );
15     }
16 }
```

Javadoc

Explaining Javadoc.

Showing the usage.

Write some javadoc comments.

Exercise: Method deprecation: adding a `@Deprecated` annotation to the `newBook()` method in `BookFactory` interface in order to force the usage of the `newBook(String)` method, using javadoc to explain the method deprecation.

Singleton

What is a Singleton: The singleton pattern ensures only one single instance of the class can be created at any given time.

Exercise: Based on the definition, build a class that is a Singleton.

With the help of the professor, explain the 3 different implementations below, from naive to safe.

Naive implementation:

```
01 package com.javacodegeeks.advanced.construction.patterns;
02
03 public class NaiveSingleton {
04     private static NaiveSingleton instance;
05
06     private NaiveSingleton() {
07     }
08
09     public static NaiveSingleton getInstance() {
10         if( instance == null ) {
11             instance = new NaiveSingleton();
12         }
13
14         return instance;
15     }
16 }
```

Safe but Non-Lazy implementation:

```
01 final property of the class.
02 package com.javacodegeeks.advanced.construction.patterns;
03
04 public class EagerSingleton {
05     private static final EagerSingleton instance = new EagerSingleton();
06
07     private EagerSingleton() {
08     }
09
10     public static EagerSingleton getInstance() {
11         return instance;
12     }
13 }
```

Safe & Lazy implementation:

```
01 package com.javacodegeeks.advanced.construction.patterns;
02
03 public class LazySingleton {
04     private static LazySingleton instance;
05
06     private LazySingleton() {
07     }
08
09     public static synchronized LazySingleton getInstance() {
10         if( instance == null ) {
11             instance = new LazySingleton();
12         }
13
14         return instance;
15     }
16 }
```

Exercise: Make the Library and KindleLibrary classes lazy and safe Singletons.

As you can see, after your modification of the Library class, the BookMain class should not compile.

Make the appropriate changes to fix the problems, then run the BookMain class to test that everything is working fine.

Testing with JUnit

Introduction to JUnit.

Test classes location in maven project: `src/test/java`

Defining a Test

```
import org.junit.Test;
...
@Test
Public void testMethod() {...
```

Assertions

```
import static org.junit.Assert.*;
```

Exercise:

1/ Update your maven pom.xml with the proper JUnit dependency:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

2/ Write a `BookTest` class, with the following tests (execute the tests using your IDE):

- When a book is created, its isbn is `null`
- When `newBook(String)` method is called on the Library, the `title` of the book is properly set (Test all the set methods on every attribute of the book instance)
- A book's title can't be null or empty. The title must also be trimmed.
- When `newBook()` is called on a Factory (Library or Kindle), the `Book` returned type is the right one (use `instanceof`)

Tip: Always put the definition of the test in the javadoc part of your test function.

Exercise: Write the following test case in the BookTest class:

```
@Test
public void testDateFormat() {

    Book book = Library.getInstance().newBook(title);
    book.setIsbn(isbn);
    book.setIssueDate(issueDate);
    assertTrue(book.toString().contains("20 octobre 1955"));

    Library.getInstance().setDateFormat(DateFormat.getDateInstance(DateFormat.LONG, Locale.US));
    book = Library.getInstance().newBook(title);
    book.setIsbn(isbn);
    book.setIssueDate(issueDate);
    assertTrue(book.toString().contains("October 20, 1955"));
}
```

Add the appropriate global variables, also modify the BookFactory interface the Library and KindleLibrary class for the test to compile.

Make the appropriate changes to the Library class and the Book class in order to make the test pass successfully.

Add the following attribute to the Book class:

```
DateFormat dateFormat;
```

Override the toString method of the Book class. The method must display the book's isbn, title and it's issuing date like this:

"isbn: xxxxxxx, title: *title*, date: *date*"

Tips

- dateFormat.format(issueDate) formats a date to a readable string where dateFormat is an instance of class DateFormat
- new GregorianCalendar(1955, Calendar.OCTOBER, 20).getTime() creates a Date instance at the specified time (here it's October, 20th 1955)
- DateFormat.getDateInstance(DateFormat.LONG, Locale.FRANCE) creates a DateFormat instance with the France local set.

For quick testing, in the class BookMain set a Date to your book in your main method, then call System.out.println on the book instance. Execute the BookMain class, see the result in the console output.

Tip:

- new Date() creates a Date instance set to the date of the current day