# Overview

## Setting up the environment in Java

Java is a general-purpose computer programming language that is concurrent, class-based,

Object-oriented etc.

Java applications are typically compiled to **bytecode** that can run on any Java virtual machine (JVM) regardless of computer architecture. The latest version is **Java 8**.

Below are the environment settings for both Linux and Windows. JVM, JRE and JDK all three are platform dependent because configuration of each Operating System is different. But, Java is platform independent.

There are few things which must be clear before setting up the environment

1. **JDK** (Java Development Kit): JDK is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.
2. **JRE** (Java Runtime Environment): JRE contains the parts of the Java libraries required to run Java programs and is intended for end users. JRE can be view as a subset of JDK.
3. **JVM:** JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms.

## Beginning Java programming with Hello World Example

The process of Java programming can be simplified in three steps:

* Create the program by typing it into a text editor and saving it to a file – HelloWorld.java.
* Compile it by typing “javac HelloWorld.java” in the terminal window.
* Execute (or run) it by typing “java HelloWorld” in the terminal window.

Below given program is the simplest program of Java printing “Hello World” to the screen. Let us try to understand every bit of code step by step.

|  |
| --- |
| /\* This is a simple Java program.     FileName : "HelloWorld.java". \*/  class HelloWorld  {      // Your program begins with a call to main().      // Prints "Hello, World" to the terminal window.      public static void main(String args[])      {          System.out.println("Hello World");      }  } |

Output:

Hello World

The “Hello World!” program consists of three primary components: the HelloWorld class definition, the main method and source code comments. Following explanation will provide you with a basic understanding of the code:

1. **Class definition:** This line uses the keyword **class**to declare that a new class is being defined.

class HelloWorld

**HelloWorld** is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace {and the closing curly brace**}**.

1. **main method:**In Java programming language, every application must contain a main method whose signature is:

public static void main(String[] args)

**public**: So that [JVM](http://www.geeksforgeeks.org/jvm-works-jvm-architecture/) can execute the method from anywhere.

**static**: Main method is to be called without object.

The modifiers public and static can be written in either order.

**void**: The main method doesn't return anything.

**main()**: Name configured in the [JVM](http://www.geeksforgeeks.org/jvm-works-jvm-architecture/).

**String[]**: The main method accepts a single argument: an array of elements of type String.

Like in C/C++, main method is the entry point for your application and will subsequently invoke all the other methods required by your program.

1. The next line of code is shown here. Notice that it occurs inside main( ).

System.out.println("Hello World");

This line outputs the string “Hello World” followed by a new line on the screen. Output is actually accomplished by the built-in println( ) method. **System** is a predefined class that provides access to the system, and **out** is the variable of type output stream that is connected to the console.

1. Comments: They can either be multi-line or single line comments.

/\* this is a simple Java program.

Call this file "HelloWorld.java". \*/

This is a multiline comment. This type of comment must begin with /\* and end with \*/. For single line you may directly use // as in C/C++.

**Important Points:**

* The name of the class defined by the program is HelloWorld, which is same as name of file (HelloWorld.java). This is not a coincidence. In Java, all codes must reside inside a class and there is at most one public class which contain main() method.
* By convention, the name of the main class (class which contain main method) should match the name of the file that holds the program.

**Compiling the program:**

* After successfully [setting up the environment](http://www.geeksforgeeks.org/setting-environment-java/), we can open terminal in both Windows/Unix and can go to directory where the file – HelloWorld.java is present.
* Now, to compile the HelloWorld program, execute the compiler – javac , specifying the name of the **source** file on the command line, as shown:
* javac HelloWorld.java
* The compiler creates a file called HelloWorld.class (in present working directory) that contains the bytecode version of the program. Now, to execute our program, [**JVM**](http://www.geeksforgeeks.org/jvm-works-jvm-architecture/)(Java Virtual Machine) needs to be called using java, specifying the name of the **class** file on the command line, as shown:
* java HelloWorld

This will print “Hello, World” to the terminal screen.

## How JVM Works – JVM Architecture?

JVM (Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE (Java Run Environment).

Java applications are called WORA (Write Once Run Everywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, a *.class* file (contains byte-code) with the same filename is generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.

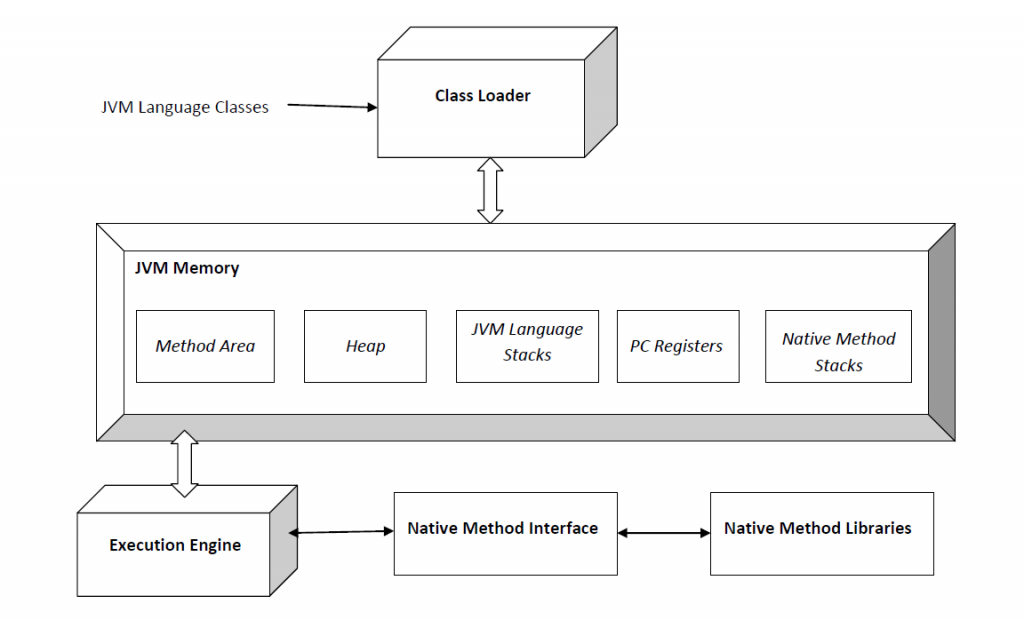
[](http://www.geeksforgeeks.org/wp-content/uploads/JVM.png)

Image Source: <https://en.wikipedia.org/wiki/Java_virtual_machine>

**Class Loader Subsystem**

It is mainly responsible for three activities.

* Loading
* Linking
* Initialization

**Loading:** The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class* file, JVM stores following information in method area.

* Fully qualified name of the loaded class and its immediate parent class.
* Whether *.class* file is related to Class or Interface or Enum.
* Modifier, Variables and Method information etc.

After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in *java.lang* package. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc. To get this object reference we can use *getClass()*method of [Object](http://www.geeksforgeeks.org/object-class-in-java/) class.

|  |
| --- |
| // A Java program to demonstrate working of a Class type  // object created by JVM to represent .class file in  // memory.  import java.lang.reflect.Field;  import java.lang.reflect.Method;    // Java code to demonstrate use of Class object  // created by JVM  public class Test  {      public static void main(String[] args)      {          Student s1 = new Student();            // Getting hold of Class object created          // by JVM.          Class c1 = s1.getClass();            // Printing type of object using c1.          System.out.println(c1.getName());            // getting all methods in an array          Method m[] = c1.getDeclaredMethods();          for (Method method : m)              System.out.println(method.getName());            // getting all fields in an array          Field f[] = c1.getDeclaredFields();          for (Field field : f)              System.out.println(field.getName());      }  }    // A sample class whose information is fetched above using  // its Class object.  class Student  {      private String name;      private int roll\_No;        public String getName()  {  return name;   }      public void setName(String name) { this.name = name; }      public int getRoll\_no()  { return roll\_No;  }      public void setRoll\_no(int roll\_no) {          this.roll\_No = roll\_no;      }  } |

Run on IDE

Output:

Student

getName

setName

getRoll\_no

setRoll\_no

name

roll\_No

**Note:** For every loaded *.class* file, only **one** object of Class is created.

Student s2 = new Student();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass();

System.out.println(c1==c2); // true

**Linking:** Performs verification, preparation, and (optionally) resolution.

* *Verification*: It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
* *Preparation*: JVM allocates memory for class variables and initializing the memory to default values.
* *Resolution*: It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

**Initialization:** In this phase, all static variables are assigned with their values defined in the code and static block (if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.  
In general there are three class loaders:

* *Bootstrap class loader*: Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in *JAVA\_HOME/jre/lib*directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
* *Extension class loader*: It is child of bootstrap class loader. It loads the classes present in the extensions directories *JAVA\_HOME/jre/lib/ext*(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.
* *System/Application class loader*: It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$AppClassLoader*class.

|  |
| --- |
| // Java code to demonstrate Class Loader subsystem  public class Test  {      public static void main(String[] args)      {          // String class is loaded by bootstrap loader, and          // bootstrap loader is not Java object, hence null          System.out.println(String.class.getClassLoader());            // Test class is loaded by Application loader          System.out.println(Test.class.getClassLoader());      }  } |

Run on IDE

Output:

null

sun.misc.Launcher$AppClassLoader@73d16e93

**Note:**JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*.

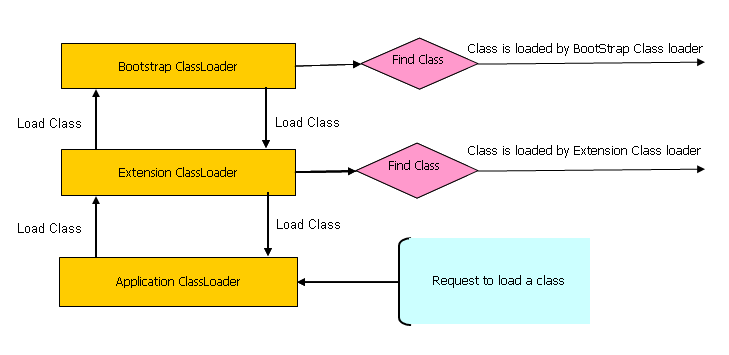
[](http://www.geeksforgeeks.org/wp-content/uploads/JVM1.png)

Image Source: <http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html>

**JVM Memory**

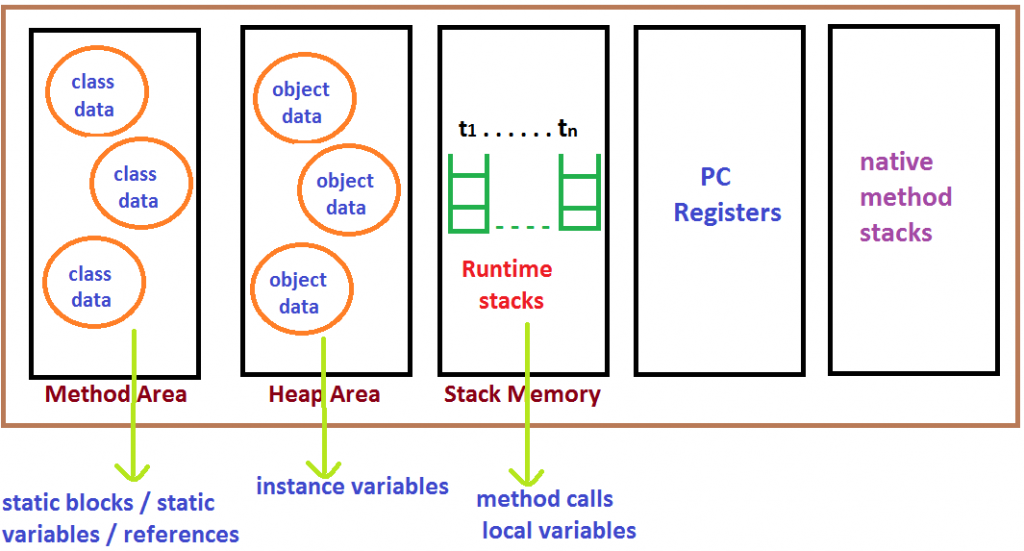
**Method area: In** method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

**Heap area: Information** of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

**Stack area: For** every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it’s run-time stack will be destroyed by JVM. It is not a shared resource.

**PC Registers: Store** address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

**Native method stacks: For** every thread, separate native stack is created. It stores native method information.

[](http://www.geeksforgeeks.org/wp-content/uploads/JVM2.png)  
Image Source: <http://java.scjp.jobs4times.com/fund/fund2.png>

**Execution Engine**

Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts:-

* **Interpreter:** It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
* **Just-In-Time Compiler (JIT):** It is used to increase efficiency of interpreter. It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls, JIT provide direct native code for that part so re-interpretation is not required, thus efficiency is improved.
* **Garbage Collector:** It destroy un-referenced objects. For more on Garbage Collector, refer [Garbage Collector](http://www.geeksforgeeks.org/garbage-collection-java/).

**Java Native Interface (JNI):**

It is a interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries:**

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

## Differences between JDK, JRE and JVM

**JAVA DEVELOPMENT KIT**

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

**JAVA RUNTIME ENVIRONMENT**

**JRE** stands for **“Java Runtime Environment”** and may also be written as **“Java RTE.”** The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the *Java Virtual Machine (JVM), core classes*, and *supporting files*.

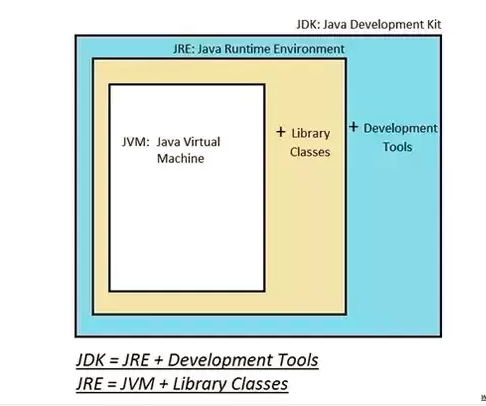
[**JAVA VIRTUAL MACHINE**](http://www.geeksforgeeks.org/jvm-works-jvm-architecture/)

It is:

* A **specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
* An **implementation** is a computer program that meets the requirements of the JVM specification
* **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

**Difference betweem JDK, JRE and JVM**

To understand the difference between these three, let us consider the following diagram.



* **JDK** – **Java Development Kit** (in short JDK) is Kit which provides the environment to **develop and execute (run)** the Java program. JDK is a kit(or package) which includes two things
  + 1. Development Tools(to provide an environment to develop your java programs)
    2. JRE (to execute your java program).

**Note:**JDK is only used by Java Developers.

* **JRE** – **Java Runtime Environment** (to say JRE) is an installation package which provides environment to **only run (not develop)** the java program (or application) onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
* **JVM** – **Java Virtual machine** (JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.

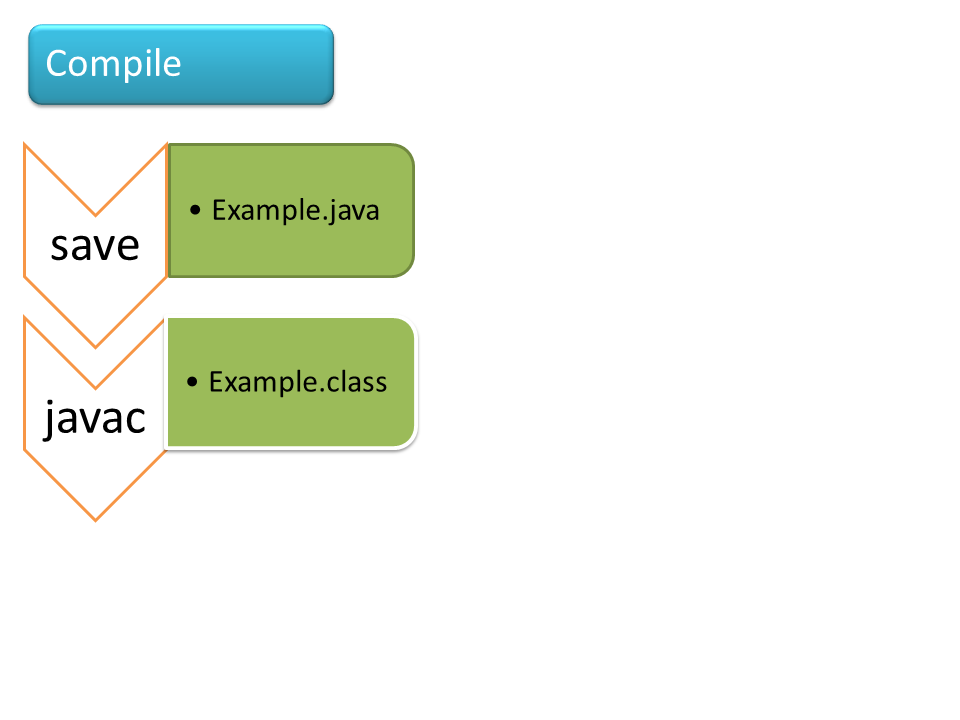
**How does JRE and JDK works?**

**What does JRE consists of?**

JRE consists of the following components:

* **Deployment technologies**, including deployment, Java Web Start and Java Plug-in.
* **User interface toolkits**, including *Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag and drop (DnD)* and *input methods*.
* **Integration libraries**, including *Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP)* and *scripting*.
* **Other base libraries**, including *international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization* and *Java for XML Processing (XML JAXP)*.
* **Lang and util base libraries**, including *lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects* and *Regular Expressions*.
* **Java Virtual Machine (JVM)**, including *Java HotSpot Client* and *Server Virtual Machines*.

**How does JRE works?**

To understand how the JRE works let us consider a Java source file saved as *Example.java*. The file is compiled into a set of Byte Code that is stored in a “*.class*” file. Here it will be “*Example.class*“.  
  
The following diagram depicts what is done at compile time.

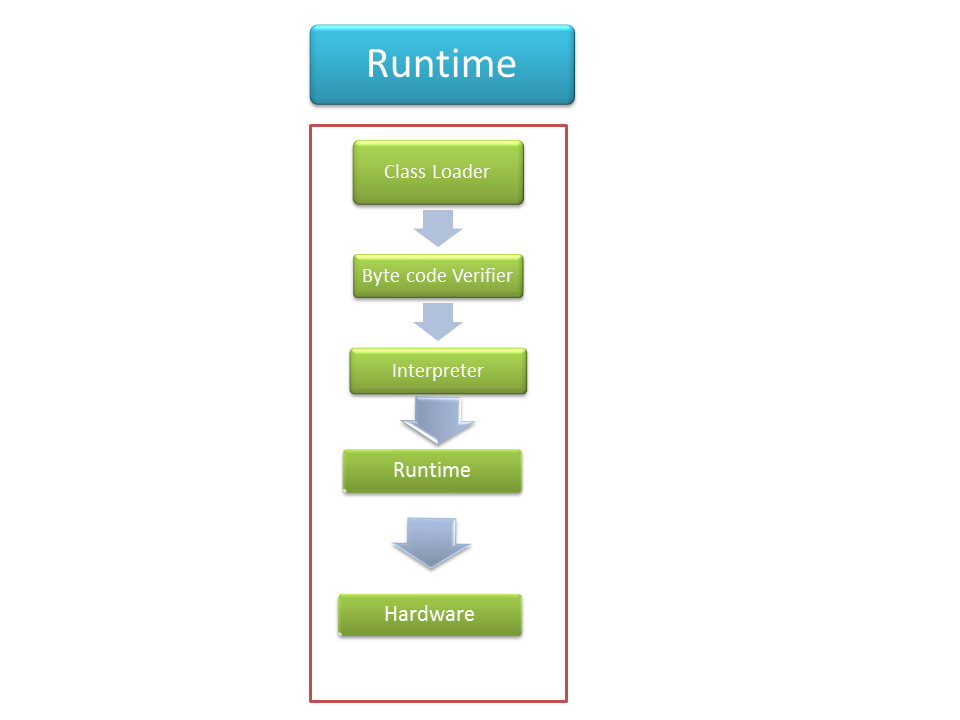
The following actions occur at runtime.

* **Class Loader**

The Class Loader loads all necessary classes needed for the execution of a program. It provides security by separating the namespaces of the local file system from that imported through the network. These files are loaded either from a hard disk, a network or from other sources.

* **Byte Code Verifier**

The JVM puts the code through the Byte Code Verifier that checks the format and checks for an illegal code. Illegal code, for example, is code that violates access rights on objects or violates the implementation of pointers.

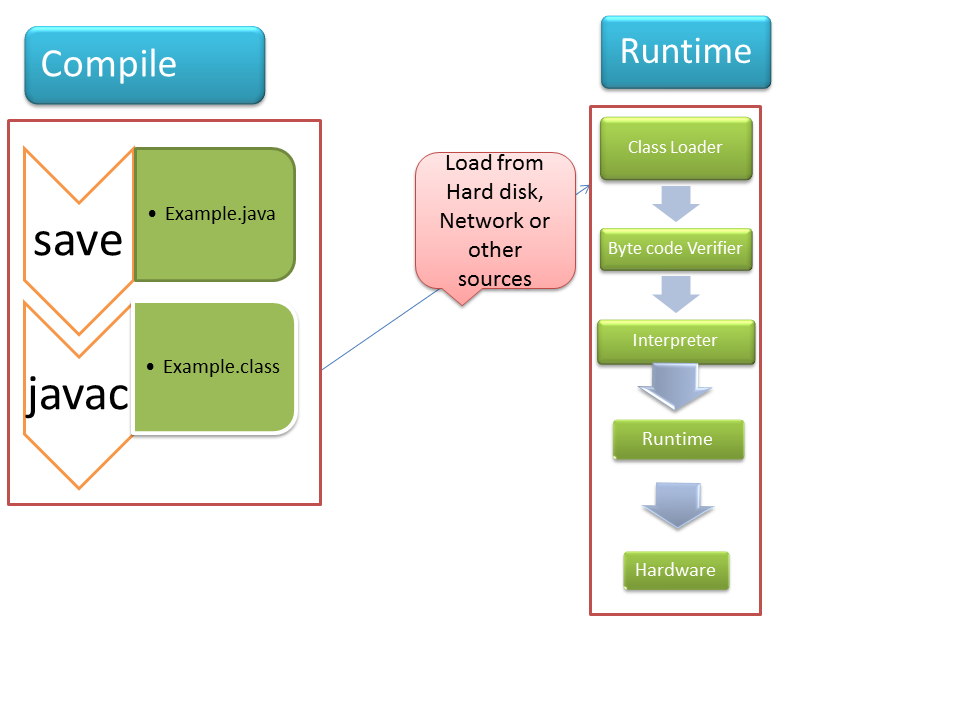
The Byte Code verifier ensures that the code adheres to the JVM specification and does not violate system integrity.  


* **Intrepreter**

At runtime the Byte Code is loaded, checked and run by the interpreter. The interpreter has the following two functions:

* + Execute the Byte Code
  + Make appropriate calls to the underlying hardware

Both operations can be shown as:

  
To understand the interactions between JDK and JRE consider the following diagram.



**How does JVM works?**

JVM becomes an instance of JRE at runtime of a Java program. It is widely known as a runtime interpreter.

JVM largely helps in the abstraction of inner implementation from the programmers who make use of libraries for their programmes from JDK.

## Does JVM create object of Main class (the class with main ())?

Consider following program.

|  |
| --- |
| class Main {      public static void main(String args[])      {          System.out.println("Hello");      }  } |

Output:

Hello

Does JVM create an object of class Main?

The answer is “No”. We have studied that the reason for main() static in Java is to make sure that the main() can be called without any instance. To justify the same, we can see that the following program compiles and runs fine.

|  |
| --- |
| // Not Main is abstract  abstract class Main {      public static void main(String args[])      {          System.out.println("Hello");      }  } |

Output:

Hello

Since we can’t create object of [abstract classes in Java](http://www.geeksforgeeks.org/abstract-classes-in-java/), it is guaranteed that object of class with main() is not created by JVM.

## Is main method compulsory in Java?

The answer to this question depends on version of java you are using. Prior to JDK 5, main method was not mandatory in a java program.

* You could write your full code under [static block](http://www.geeksforgeeks.org/g-fact-79/) and it ran normally.
  + The static block is first executed as soon as the class is loaded before the main(); method is invoked and therefore before the main() is called. main is usually declared as static method and hence [Java doesn’t need an object to call main method.](http://www.geeksforgeeks.org/jvm-create-object-main-class-class-contains-main/)

However, From JDK6 main method is mandatory. If your program doesn’t contain main method, then you will get a run-time error “main method not found in the class”. Note that your program will successfully compile in this case, but at run-time, it will throw error.

|  |
| --- |
| // This program will successfully run  // prior to JDK 5  public class Test  {      // static block      static      {          System.out.println("program is running without main() method");      }  } |

Output:

* If run prior to JDK 5

program is running without main() method

* If run on JDK 6,7,8

Error: Main method not found in class Test, please define the main method as: public static void main(String[] args)

# Basics

## Widening Primitive Conversion in Java

Here is a small code snippet given. Try to guess the output

|  |
| --- |
| public class Test  {      public static void main(String[] args)      {          System.out.print("Y" + "O");          System.out.print('L' + 'O');      }  } |

At first glance, we expect “YOLO” to be printed.

Actual Output:

“YO155”.

Explanation:  
When we use double quotes, the text is treated as a string and “YO” is printed, but when we use single quotes, the characters ‘L’ and ‘O’ are converted to int. This is called widening primitive conversion. After conversion to integer, the numbers are added (‘L’ is 76 and ‘O’ is 79) and 155 is printed.

Now try to guess the output of following:

|  |
| --- |
| public class Test  {      public static void main(String[] args)      {          System.out.print("Y" + "O");          System.out.print('L');          System.out.print('O');      }  } |

Output: YOLO

Explanation: This will now print “YOLO” instead of “YO7679”. It is because the widening primitive conversion happens only when ‘+’ operator is present.

Widening primitive conversion is applied to convert either or both operands as specified by the following rules. The result of adding Java chars, shorts or bytes is an int:

* If either operand is of type double, the other is converted to double.
* Otherwise, if either operand is of type float, the other is converted to float.
* Otherwise, if either operand is of type long, the other is converted to long.
* Otherwise, both operands are converted to type int

## Interesting facts about null in Java

Almost all the programming languages are bonded with null. There is hardly a programmer, who is not troubled by null.

In Java, null is associated java.lang.NullPointerException. As it is a class in java.lang package, it is called when we try to perform some operations with or without null and sometimes we don’t even know where it has happened.

Below are some important points about null in java which every Java programmer should know:

**1. null is Case sensitive:** null is literal in Java and because keywords are **case-sensitive** in java, we can’t write NULL or 0 as in C language.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| public class Test  {      public static void main (String[] args) throws java.lang.Exception      {          // compile-time error : can't find symbol 'NULL'          Object obj = NULL;          //runs successfully          Object obj1 = null;      }  }  Output:  5: error: cannot find symbol  can't find symbol 'NULL'  ^  variable NULL  class Test  1 error  **2. Reference Variable value:**Any reference variable in Java has default value null.   |  | | --- | | public class Test  {      private static Object obj;      public static void main(String args[])      {          // it will print null;          System.out.println("Value of object obj is : " + obj);      }  } |   **Output:**  Value of object obj is : null  **3. Type of null**: Unlike common misconception, null is not Object or neither a type. It’s just a special value, which can be assigned to any reference type and you can type cast null to any type.  **Examples:**  // null can be assigned to String  String str = null;  // you can assign null to Integer also  Integer itr = null;  // null can also be assigned to Double  Double dbl = null;  // null can be type cast to String  String myStr = (String) null;  // it can also be type casted to Integer  Integer myItr = (Integer) null;  // yes it's possible, no error  Double myDbl = (Double) null;    **4. Autoboxing and unboxing:** During auto-boxing and unboxing operations, compiler simply throws Nullpointer exception error if a null value is assigned to primitive boxed data type.   |  | | --- | | public class Test  {      public static void main (String[] args) throws java.lang.Exception      {              //An integer can be null, so this is fine              Integer i = null;                //Unboxing null to integer throws NullpointerException              int a = i;      }  } |   Output:  Exception in thread "main" java.lang.NullPointerException  at Test.main(Test.java:6)  **5. instanceof operator:**The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface). At run time, the result of the instanceof operator is true if the value of the Expression is not null.  This is an important property of instanceof operation which makes it useful for type casting checks.   |  | | --- | | public class Test  {      public static void main (String[] args) throws java.lang.Exception      {          Integer i = null;          Integer j = 10;            //prints false          System.out.println(i instanceof Integer);            //Compiles successfully          System.out.println(j instanceof Integer);      }  } |   Output:  false  true  **6. Static vs Non static Methods:**We cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but we can call static method with reference variables with null values. Since static methods are bonded using static binding, they won’t throw Null pointer Exception.   |  | | --- | | public class Test  {      public static void main(String args[])      {          Test obj= null;          obj.staticMethod();          obj.nonStaticMethod();      }        private static void staticMethod()      {          //Can be called by null reference          System.out.println("static method, can be called by null reference");        }        private void nonStaticMethod()      {          //Cannot be called by null reference          System.out.print(" Non-static method- ");          System.out.println("cannot be called by null reference");        }    } |   Output:  static method, can be called by null referenceException in thread "main"  java.lang.NullPointerException  at Test.main(Test.java:5)  **7. == and !=**The comparision and not equal to operators are allowed with null in Java. This can made useful in checking of null with objects in java.   |  | | --- | | public class Test  {      public static void main(String args[])      {        //return true;      System.out.println(null==null);        //return false;      System.out.println(null!=null);        }  } |   Output:  true  false |

# OOP Concepts

## How are Java objects stored in memory?

In Java, all objects are dynamically allocated on Heap. This is different from C++ where objects can be allocated memory either on Stack or on Heap. In C++, when we allocate abject using new(), the abject is allocated on Heap, otherwise on Stack if not global or static.

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So the object is always allocated memory on heap (See [this](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html) for more details).

For example, following program fails in compilation. Compiler gives error “Error here because t is not initiated”.

|  |
| --- |
| class Test {      // class contents      void show() {          System.out.println("Test::show() called");      }  }    public class Main {      public static void main(String[] args) {          Test t;          t.show(); // Error here because t is not initialed      }  } |

Allocating memory using new() makes above program work.

|  |
| --- |
| class Test {      // class contents      void show() {          System.out.println("Test::show() called");      }  }    public class Main {      public static void main(String[] args) {          Test t = new Test(); //all objects are dynamically allocated          t.show(); // No error      }  } |

## Different ways to create objects in Java

As you all know, in Java, a class provides the blueprint for objects, you create an object from a class. There are many different ways to create objects in Java.

**Following are some ways in which you can create objects in Java:**

**1) Using new Keyword :** Using new keyword is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way. By using this method we can call any constructor we want to call (no argument or parameterized constructors).

|  |
| --- |
| // Java program to illustrate creation of Object  // using new keyword  public class NewKeywordExample  {      String name = "GeeksForGeeks";      public static void main(String[] args)      {          // Here we are creating Object of          // NewKeywordExample using new keyword          NewKeywordExample obj = new NewKeywordExample();          System.out.println(obj.name);      }  } |

Output:

GeeksForGeeks

**2) Using**[**New Instance**](http://www.geeksforgeeks.org/new-operator-vs-newinstance-method-java/)**:**If we know the name of the class & if it has a public default constructor we can create an object –**Class.forName**. We can use it to create the Object of a Class. Class.forName actually loads the Class in Java but doesn’t create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.

|  |
| --- |
| // Java program to illustrate creation of Object  // using new Instance  public class NewInstanceExample  {      String name = "GeeksForGeeks";      public static void main(String[] args)      {          try          {              Class cls = Class.forName("NewInstanceExample");              NewInstanceExample obj =                      (NewInstanceExample) cls.newInstance();              System.out.println(obj.name);          }          catch (ClassNotFoundException e)          {              e.printStackTrace();          }          catch (InstantiationException e)          {              e.printStackTrace();          }          catch (IllegalAccessException e)          {              e.printStackTrace();          }      }  } |

Output:

GeeksForGeeks

**3) Using**[**clone() method:**](http://www.geeksforgeeks.org/clone-method-in-java-2/) Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.  
To use clone() method on an object we need to implement **Cloneable** and define the clone() method in it.

|  |
| --- |
| // Java program to illustrate creation of Object  // using clone() method  public class CloneExample implements Cloneable  {      @Override      protected Object clone() throws CloneNotSupportedException      {          return super.clone();      }      String name = "GeeksForGeeks";        public static void main(String[] args)      {          CloneExample obj1 = new CloneExample();          try          {              CloneExample obj2 = (CloneExample) obj1.clone();              System.out.println(obj2.name);          }          catch (CloneNotSupportedException e)          {              e.printStackTrace();          }      }  } |

Output:

GeeksForGeeks

**Note :**

* Here we are creating the clone of an existing Object and not any new Object.
* Class need to implement Cloneable Interface otherwise it will throw **CloneNotSupportedException**.

**4) Using**[**deserialization**](http://www.geeksforgeeks.org/serialization-in-java/)**:** Whenever we serialize and then deserialize an object, JVM creates a separate object. In **deserialization**, JVM doesn’t use any constructor to create the object.

To deserialize an object we need to implement the Serializable interface in the class.

**Serializing an Object :**

|  |
| --- |
| // Java program to illustrate Serializing  // an Object.  import java.io.\*;    class DeserializationExample implements Serializable  {      private String name;      DeserializationExample(String name)      {          this.name = name;      }        public static void main(String[] args)      {          try          {              DeserializationExample d =                      new DeserializationExample("GeeksForGeeks");              FileOutputStream f = new FileOutputStream("file.txt");              ObjectOutputStream oos = new ObjectOutputStream(f);              oos.writeObject(d);              oos.close();              f.close();          }          catch (Exception e)          {              e.printStackTrace();          }      }  } |

Object of DeserializationExample class is serialized using writeObject() method and written to file.txt file.

**Deserialization of Object :**

|  |
| --- |
| // Java program to illustrate creation of Object  // using Deserialization.  import java.io.\*;    public class DeserializationExample  {      public static void main(String[] args)      {          try          {              DeserializationExample d;              FileInputStream f = new FileInputStream("file.txt");              ObjectOutputStream oos = new ObjectOutputStream(f);              d = (DeserializationExample)oos.readObject();          }          catch (Exception e)          {              e.printStackTrace();          }          System.out.println(d.name);      }  } |

Output:

GeeksForGeeks

**5) Using newInstance() method of Constructor class :** This is similar to the newInstance() method of a class. There is one newInstance() method in the **java.lang.reflect.Constructor**class which we can use to create objects. It can also call parameterized constructor, and private constructor by using this newInstance() method.

Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class internally uses newInstance() method of Constructor class.

|  |
| --- |
| // Java program to illustrate creation of Object  // using newInstance() method of Constructor class  import java.lang.reflect.\*;    public class ReflectionExample  {      private String name;      ReflectionExample()      {      }      public void setName(String name)      {          this.name = name;      }      public static void main(String[] args)      {          try          {              Constructor<ReflectionExample> constructor                  = ReflectionExample.class.getDeclaredConstructor();              ReflectionExample r = constructor.newInstance();              r.setName("GeeksForGeeks");              System.out.println(r.name);          }          catch (Exception e)          {              e.printStackTrace();          }      }  } |

Output:

GeeksForGeeks

## Shadowing of static functions in Java [(Also called Method Hiding)](https://www.geeksforgeeks.org/g-fact-63/)

In Java, if name of a derived class static function is same as base class static function then the derived class static function shadows (or conceals) the base class static function. For example, the following Java code prints “A.fun()”

|  |
| --- |
| // file name: Main.java  class A {     static void fun() {        System.out.println("A.fun()");     }  }    class B extends A {     static void fun() {        System.out.println("B.fun()");     }  }    public class Main {     public static void main(String args[]) {        A a = new B();        a.fun();  // prints A.fun()     }  } |

If we make both A.fun() and B.fun() as non-static then the above program would print “B.fun()”.

## Runtime Polymorphism with Data Members

In Java, we can override methods only, not the variables (data members), so runtime polymorphism cannot be achieved by data members.

 For example:

|  |
| --- |
| // Java program to illustrate the fact that  // runtime polymorphism cannot be achieved  // by data members    // class A  class A  {      int x = 10;  }    // class B  class B extends A  {      int x = 20;  }    // Driver class  public class Test  {      public static void main(String args[])      {          A a = new B(); // object of type B            // Data member of class A will be accessed          System.out.println(a.x);      }  } |

Output:

10

**Explanation**: In above program, both the class A(super class) and B(sub class) have a common variable ‘x’. Now we make object of class B, referred by ‘a’ which is of type of class A. Since variables are not overridden, so the statement “a.x” will always refer to data member of super class.

**Advantages of Dynamic Method Dispatch**

1. Dynamic method dispatch allow Java to support [overriding of methods](http://www.geeksforgeeks.org/overriding-in-java/) which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

[**Static vs Dynamic binding**](http://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/)

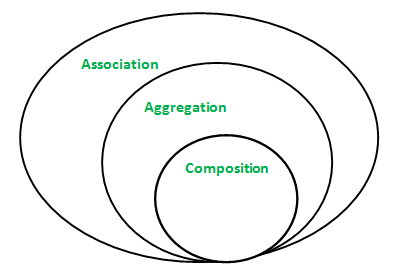
* Static binding is done during compile-time while dynamic binding is done during run-time.
* private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

## Association, Composition and Aggregation in Java

**Association**

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.

In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. Composition and Aggregation are the two forms of association.

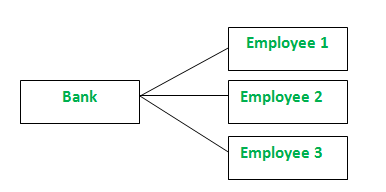
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/AssociationAggregation-and-Composition.png)

|  |
| --- |
| // Java program to illustrate the  // concept of Association  import java.io.\*;    // class bank  class Bank  {      private String name;        // bank name      Bank(String name)      {          this.name = name;      }        public String getBankName()      {          return this.name;      }  }    // employee class  class Employee  {      private String name;        // employee name      Employee(String name)      {          this.name = name;      }        public String getEmployeeName()      {          return this.name;      }  }    // Association between both the  // classes in main method  class Association  {      public static void main (String[] args)      {          Bank bank = new Bank("Axis");          Employee emp = new Employee("Neha");            System.out.println(emp.getEmployeeName() +                 " is employee of " + bank.getBankName());      }  } |

Output:

Neha is employee of Axis

In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, so it is a one-to-many relationship.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Association-in-Java.png)

**Aggregation**

It is a special form of Association where:

* It represents Has-A relationship.
* It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation, both the entries can survive individually which means ending one entity will not affect the other entity.

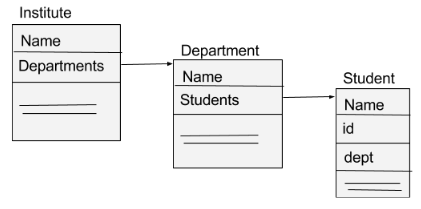
|  |
| --- |
| // Java program to illustrate  //the concept of Aggregation.  import java.io.\*;  import java.util.\*;    // student class  class Student  {      String name;      int id ;      String dept;        Student(String name, int id, String dept)      {            this.name = name;          this.id = id;          this.dept = dept;        }  }    /\* Department class contains list of student  Objects. It is associated with student  class through its Object(s). \*/  class Department  {        String name;      private List<Student> students;      Department(String name, List<Student> students)      {            this.name = name;          this.students = students;        }        public List<Student> getStudents()      {          return students;      }  }    /\* Institute class contains list of Department  Objects. It is asoociated with Department  class through its Object(s).\*/  class Institute  {        String instituteName;      private List<Department> departments;        Institute(String instituteName, List<Department> departments)      {          this.instituteName = instituteName;          this.departments = departments;      }        // count total students of all departments      // in a given institute      public int getTotalStudentsInInstitute()      {          int noOfStudents = 0;          List<Student> students;          for(Department dept : departments)          {              students = dept.getStudents();              for(Student s : students)              {                  noOfStudents++;              }          }          return noOfStudents;      }    }    // main method  class GFG  {      public static void main (String[] args)      {          Student s1 = new Student("Mia", 1, "CSE");          Student s2 = new Student("Priya", 2, "CSE");          Student s3 = new Student("John", 1, "EE");          Student s4 = new Student("Rahul", 2, "EE");            // making a List of          // CSE Students.          List <Student> cse\_students = new ArrayList<Student>();          cse\_students.add(s1);          cse\_students.add(s2);            // making a List of          // EE Students          List <Student> ee\_students = new ArrayList<Student>();          ee\_students.add(s3);          ee\_students.add(s4);            Department CSE = new Department("CSE", cse\_students);          Department EE = new Department("EE", ee\_students);            List <Department> departments = new ArrayList<Department>();          departments.add(CSE);          departments.add(EE);            // creating an instance of Institute.          Institute institute = new Institute("BITS", departments);            System.out.print("Total students in institute: ");          System.out.print(institute.getTotalStudentsInInstitute());      }  } |

Output:

Total students in institute: 4

In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).

**It represents a Has-A relationship.**

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Aggregation_1.png)

When do we use Aggregation?

Code reuse is best achieved by aggregation.

**Composition**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents part-of relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object cannot exist without the other entity.

Let’s take example of Library.

|  |
| --- |
| // Java program to illustrate  // the concept of Composition  import java.io.\*;  import java.util.\*;    // class book  class Book  {        public String title;      public String author;        Book(String title, String author)      {            this.title = title;          this.author = author;      }  }    // Libary class contains  // list of books.  class Library  {        // reference to refer to list of books.      private final List<Book> books;        Library (List<Book> books)      {          this.books = books;      }        public List<Book> getTotalBooksInLibrary(){           return books;      }    }    // main method  class GFG  {      public static void main (String[] args)      {            // Creating the Objects of Book class.          Book b1 = new Book("EffectiveJ Java", "Joshua Bloch");          Book b2 = new Book("Thinking in Java", "Bruce Eckel");          Book b3 = new Book("Java: The Complete Reference", "Herbert Schildt");            // Creating the list which contains the          // no. of books.          List<Book> books = new ArrayList<Book>();          books.add(b1);          books.add(b2);          books.add(b3);            Library library = new Library(books);            List<Book> bks = library.getTotalBooksInLibrary();          for(Book bk : bks){                System.out.println("Title : " + bk.title + " and "              +" Author : " + bk.author);          }      }  } |

Output

Title : EffectiveJ Java and Author : Joshua Bloch

Title : Thinking in Java and Author : Bruce Eckel

Title : Java: The Complete Reference and Author : Herbert Schildt

In above example a library can have no. of books on same or different subjects. So, if Library gets destroyed then All books within that particular library will be destroyed. I.e. book cannot exist without library. That’s why it is composition.

**Aggregation vs Composition**

1. Dependency: Aggregation implies a relationship where the child can exist independentlyof the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don’t exist separate to a Human
2. Type of Relationship: Aggregation relation is “has-a” and composition is “part-of”relation.
3. Type of association: Composition is a strong Association whereas Aggregation is a weakAssociation.

|  |
| --- |
| // Java program to illustrate the difference between Aggregation and Composition.    import java.io.\*;    // Engine class which will be used by car. So 'Car' class will have a field of Engine type.  class Engine  {      // starting an engine.      public void work()      {           System.out.println("Engine of car has been started ");         }  }    // Engine class  final class Car  {       // For a car to move, it need to have an engine.      private final Engine engine; // Composition      //private Engine engine;     // Aggregation        Car(Engine engine)      {          this.engine = engine;      }        // car start moving by starting engine      public void move()      {        //if(engine != null)          {              engine.work();              System.out.println("Car is moving ");          }      }  }    class GFG  {      public static void main (String[] args)      {           // making an engine by creating an instance of Engine class.          Engine engine = new Engine();            // Making a car with engine. so we are passing a engine instance as an argument while          // creating instance of Car.          Car car = new Car(engine);          car.move();        }  } |

Output:

Engine of car has been started

Car is moving

In case of aggregation, the Car also performs its functions through an Engine. But the Engine is not always an internal part of the Car. An engine can be swapped out or even can be removed from the car. That’ why we make The Engine type field non-final.

**Encapsulation vs Data Abstraction**

1. [Encapsulation](http://contribute.geeksforgeeks.org/encapsulation-in-java/) is data hiding (information hiding) while Abstraction is detail hiding (implementation hiding).
2. While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

**Advantages of Abstraction**

1. It reduces the complexity of viewing the things.
2. Avoids code duplication and increases reusability.
3. Helps to increase security of an application or program as only important details are provided to the user.

## Covariant return types in Java

Before JDK 5.0, it was not possible to [override](https://www.geeksforgeeks.org/overriding-in-java/) a method by changing the return type. When we override a parent class method, the name, argument types and return type of the overriding method in child class has to be exactly same as that of parent class method. Overriding method was said to be invariant with respect to return type.

Covariant return types

Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child’s return type should be sub-type of parent’s return type. Overriding method becomes variant with respect to return type.

Co-variant return type is based on [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle).

Below is the simple example to understand the co-variant return type with method overriding.

|  |
| --- |
| // Java program to demonstrate that we can have  // different return types if return type in  // overridden method is sub-type    // Two classes used for return types.  class A {}  class B extends A {}    class Base  {      A fun()      {          System.out.println("Base fun()");          return new A();      }  }    class Derived extends Base  {      B fun()      {          System.out.println("Derived fun()");          return new B();      }  }    public class Main  {      public static void main(String args[])      {         Base base = new Base();         base.fun();           Derived derived = new Derived();         derived.fun();      }  } |

Output:

Base fun()

Derived fun()

Note: If we swap return types of Base and Derived, then above program would not work. Please see [this](https://ide.geeksforgeeks.org/mqAFFv) program for example.

**Advantages:**

* It helps to avoid confusing type casts present in the class hierarchy and thus making the code readable, usable and maintainable.
* We get a liberty to have more specific return types when overriding methods.
* Help in preventing run-time ClassCastExceptions on returns.

## Object class in Java

**Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

**Using Object class methods**

There are 12 methods in **Object** class:

* **toString()**: toString() provides String representation of an Object and used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@’, and the unsigned hexadecimal representation of the hash code of the object. In other words, it is defined as:

// Default behavior of toString() is to print class name, then

// @, then unsigned hexadecimal representation of the hash code

// of the object

public String toString()

{

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

It is always recommended to override **toString()** method to get our own String representation of Object. For more on override of toString() method refer – [Overriding toString() in Java](https://www.geeksforgeeks.org/overriding-tostring-method-in-java/)

**Note :** Whenever we try to print any Object reference, then internally toString() method is called.

Student s = new Student();

// Below two statements are equivalent

System.out.println(s);

System.out.println(s.toString());

* **hashCode()**: For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that hashCode() method returns the address of object, which is not correct. It convert the internal address of object to an integer by using an algorithm. The hashCode() method is **native** because in Java it is impossible to find address of an object, so it uses native languages like C/C++ to find address of the object.

**Use of hashCode() method :** Returns a hash value that is used to search object in a collection. JVM (Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable etc.

The main advantage of saving objects based on Hash code is that searching becomes easy.

**Note:** Override of **hashCode()** method needs to be done such that for every object we generate a unique number. For example,for a Student class we can return roll no. of student from hashCode() method as it is unique.

|  |
| --- |
| // Java program to demonstrate working of  // hasCode() and toString()  public class Student  {      static int last\_roll = 100;      int roll\_no;        // Constructor      Student()      {          roll\_no = last\_roll;          last\_roll++;      }        // Overriding hashCode()      @Override      public int hashCode()      {          return roll\_no;      }        // Driver code      public static void main(String args[])      {          Student s = new Student();            // Below two statements are equivalent          System.out.println(s);          System.out.println(s.toString());      }  } |

Output:

Student@64

Student@64

Note that 4\*160 + 6\*161 = 100

* **equals(Object obj)**: Compares the given object to “this” object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override **equals(Object obj)** method to get our own equality condition on Objects. For more on override of equals(Object obj) method refer – [Overriding equals method in Java](https://www.geeksforgeeks.org/overriding-equals-method-in-java/)

**Note :** It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

* **getClass()** : Returns the class object of “this” object and used to get actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don’t override it.

|  |
| --- |
| // Java program to demonstrate working of getClass()  public class Test  {      public static void main(String[] args)      {          Object obj = new String("GeeksForGeeks");          Class c = obj.getClass();          System.out.println("Class of Object obj is : "                             + c.getName());      }  } |

* Output:

Class of Object obj is : java.lang.String

* **Note :**After loading a .class file, JVM will create an object of the type java.lang.Class in the Heap area. We can use this class object to get Class level information. It is widely used in[Reflection](https://www.geeksforgeeks.org/reflection-in-java/)
* **finalize()** method : This method is called just before an object is garbage collected. It is called by the [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/) on an object when garbage collector determines that there are no more references to the object. We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks. For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.  
  **Note: finalize** method is called just **once** on an object even though that object is eligible for garbage collection multiple times.

|  |
| --- |
| // Java program to demonstrate working of finalize()  public class Test  {      public static void main(String[] args)      {          Test t = new Test();          System.out.println(t.hashCode());            t = null;            // calling garbage collector          System.gc();            System.out.println("end");      }        @Override      protected void finalize()      {          System.out.println("finalize method called");      }  } |
|  |

Output:

366712642

end

finalize method called

* **clone()** : It returns a new object that is exactly the same as this object. For clone() method refer [Clone()](https://www.geeksforgeeks.org/clone-method-in-java-2/).
* The remaining three methods **wait()**, **notify()** **notifyAll()** are related to Concurrency. Refer

[Inter-thread Communication in Java](https://www.geeksforgeeks.org/inter-thread-communication-java/)for details.

## Overriding equals method in Java

Consider the following Java program:

|  |
| --- |
| class Complex {      private double re, im;        public Complex(double re, double im) {          this.re = re;          this.im = im;      }  }    // Driver class to test the Complex class  public class Main {      public static void main(String[] args) {          Complex c1 = new Complex(10, 15);          Complex c2 = new Complex(10, 15);          if (c1 == c2) {              System.out.println("Equal ");          } else {              System.out.println("Not Equal ");          }      }  } |

Output:

Not Equal

The reason for printing “Not Equal” is simple: when we compare c1 and c2, it is checked whether both c1 and c2 refer to same object or not ([object variables are always references in Java](https://www.geeksforgeeks.org/archives/8926)). c1 and c2 refer to two different objects, hence the value (c1 == c2) is false. If we create another reference say c3 like following, then (c1 == c3) will give true.

|  |
| --- |
| Complex c3 = c1;  // (c3 == c1) will be true |

So, how do we check for equality of values inside the objects? All classes in Java inherit from the Object class, directly or indirectly (See point 1 of [this](https://www.geeksforgeeks.org/archives/15055)). The [Object class](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html) has some basic methods like clone(), toString(), equals(),.. etc. We can override the equals method in our class to check whether two objects have same data or not.

|  |
| --- |
| class Complex {        private double re, im;        public Complex(double re, double im) {          this.re = re;          this.im = im;      }        // Overriding equals() to compare two Complex objects      @Override      public boolean equals(Object o) {            // If the object is compared with itself then return true          if (o == this) {              return true;          }            /\* Check if o is an instance of Complex or not            "null instanceof [type]" also returns false \*/          if (!(o instanceof Complex)) {              return false;          }            // typecast o to Complex so that we can compare data members          Complex c = (Complex) o;            // Compare the data members and return accordingly          return Double.compare(re, c.re) == 0                  && Double.compare(im, c.im) == 0;      }  }    // Driver class to test the Complex class  public class Main {        public static void main(String[] args) {          Complex c1 = new Complex(10, 15);          Complex c2 = new Complex(10, 15);          if (c1.equals(c2)) {              System.out.println("Equal ");          } else {              System.out.println("Not Equal ");          }      }  } |

Output:

Equal

As a side note, when we override equals(), it is recommended to also override the hashCode() method. If we don’t do so, equal objects may get different hash-values; and hash based collections, including HashMap, HashSet, and Hashtable do not work properly.

### Introduction

The Java super class java.lang.Object has two very important methods defined in it. They are -

* public boolean equals(Object obj)
* public int hashCode()

These methods prove very important when user classes are confronted with other Java classes, when objects of such classes are added to collections etc. These two methods have become part of Sun Certified Java Programmer 1.4 exam (SCJP 1.4) objectives. This article intends to provide the necessary information about these two methods that would help the SCJP 1.4 exam aspirants. Moreover, this article hopes to help you understand the mechanism and general contracts of these two methods; irrespective of whether you are interested in taking the SCJP 1.4 exam or not. This article should help you while implementing these two methods in your own classes.

**public boolean equals(Object obj)**

This method checks if some other object passed to it as an argument is equal to the object on which this method is invoked. The default implementation of this method in Object class simply checks if two object references x and y refer to the same object. i.e. It checks if x == y. This particular comparison is also known as "shallow comparison". However, the classes providing their own implementations of the equals method are supposed to perform a "deep comparison"; by actually comparing the relevant data members. Since Object class has no data members that define its state, it simply performs shallow comparison.  
  
This is what the JDK 1.4 API documentation says about the equals method of Object class- 

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation:

* It is reflexive: for any reference value x, x.equals(x) should return true.
* It is symmetric: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
* It is transitive: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* It is consistent: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
* For any non-null reference value x, x.equals(null) should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any reference values x and y, this method returns true if and only if x and y refer to the same object (x==y has the value true).  
  
Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

The contract of the equals method precisely states what it requires. Once you understand it completely, implementation becomes relatively easy, moreover it would be correct. Let's understand what each of this really means.

1. **Reflexive -** It simply means that the object must be equal to itself, which it would be at any given instance; unless you intentionally override the equals method to behave otherwise.
2. **Symmetric -** It means that if object of one class is equal to another class object, the other class object must be equal to this class object. In other words, one object can not unilaterally decide whether it is equal to another object; two objects, and consequently the classes to which they belong, must bilaterally decide if they are equal or not. They BOTH must agree.  
   Hence, it is improper and incorrect to have your own class with equals method that has comparison with an object of java.lang.String class, or with any other built-in Java class for that matter. It is very important to understand this requirement properly, because it is quite likely that a naive implementation of equals method may violate this requirement which would result in undesired consequences.
3. **Transitive -** It means that if the first object is equal to the second object and the second object is equal to the third object; then the first object is equal to the third object. In other words, if two objects agree that they are equal, and follow the symmetry principle, one of them can not decide to have a similar contract with another object of different class. All three must agree and follow symmetry principle for various permutations of these three classes.  
   Consider this example - A, B and C are three classes. A and B both implement the equals method in such a way that it provides comparison for objects of class A and class B. Now, if author of class B decides to modify its equals method such that it would also provide equality comparison with class C; he would be violating the transitivity principle. Because, no proper equals comparison mechanism would exist for class A and class C objects.
4. **Consistent -** It means that if two objects are equal, they must remain equal as long as they are not modified. Likewise, if they are not equal, they must remain non-equal as long as they are not modified. The modification may take place in any one of them or in both of them.
5. **null comparison -** It means that any instantiable class object is not equal to null, hence the equals method must return false if a null is passed to it as an argument. You have to ensure that your implementation of the equals method returns false if a null is passed to it as an argument.
6. **Equals & Hash Code relationship -** The last note from the API documentation is very important, it states the relationship requirement between these two methods. It simply means that if two objects are equal, then they must have the same hash code, however the opposite is NOT true. This is discussed in details later in this article.

The details about these two methods are interrelated and how they should be overridden correctly is discussed later in this article.

**public int hashCode()**

This method returns the hash code value for the object on which this method is invoked. This method returns the hash code value as an integer and is supported for the benefit of hashing based collection classes such as Hashtable, HashMap, HashSet etc. This method must be overridden in every class that overrides the equals method.  
  
This is what the JDK 1.4 API documentation says about the hashCode method of Object class- 

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.

The general contract of hashCode is:

* Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
* If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
* It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

As compared to the general contract specified by the equals method, the contract specified by the hashCode method is relatively simple and easy to understand. It simply states two important requirements that must be met while implementing the hashCode method. The third point of the contract, in fact is the elaboration of the second point. Let's understand what this contract really means.

1. **Consistency during same execution -** Firstly, it states that the hash code returned by the hashCode method must be consistently the same for multiple invocations during the same execution of the application as long as the object is not modified to affect the equals method.
2. **Hash Code & Equals relationship -** The second requirement of the contract is the hashCode counterpart of the requirement specified by the equals method. It simply emphasizes the same relationship - equal objects must produce the same hash code. However, the third point elaborates that unequal objects need not produce distinct hash codes.

After reviewing the general contracts of these two methods, it is clear that the relationship between these two methods can be summed up in the following statement –

**Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.**

The rest of the requirements specified in the contracts of these two methods are specific to those methods and are not directly related to the relationship between these two methods. Those specific requirements are discussed earlier. This relationship also enforces that whenever you override the equals method, you must override the hashCode method as well. Failing to comply with this requirement usually results in undetermined, undesired behavior of the class when confronted with Java collection classes or any other Java classes.

**Correct Implementation Example**

The following code exemplifies how all the requirements of equals and hashCode methods should be fulfilled so that the class behaves correctly and consistently with other Java classes. This class implements the equals method in such a way that it only provides equality comparison for the objects of the same class, similar to built-in Java classes like String and other wrapper classes.

public class Test

{

private int num;

private String data;

public boolean equals(Object obj)

{

if(this == obj)

return true;

if((obj == null) || (obj.getClass() != this.getClass()))

return false;

// object must be Test at this point

Test test = (Test)obj;

return num == test.num &&

(data == test.data || (data != null && data.equals(test.data)));

}

public int hashCode()

{

int hash = 7;

hash = 31 \* hash + num;

hash = 31 \* hash + (null == data ? 0 : data.hashCode());

return hash;

}

// other methods

}

Now, let's examine why this implementation is the correct implementation. The class Test has two member variables - num and data. These two variables define state of the object and they also participate in the equals comparison for the objects of this class. Hence, they should also be involved in calculating the hash codes of this class objects.  
  
Consider the equals method first. We can see that at line 8, the passed object reference is compared with this object itself, this approach usually saves time if both the object references are referring to the same object on the heap and if the equals comparison is expensive. Next, the if condition at line 10 first checks if the argument is null, if not, then (due to the short-circuit nature of the OR || operator) it checks if the argument is of type Test by comparing the classes of the argument and this object. This is done by invoking the getClass() method on both the references. If either of these conditions fails, then false is returned. This is done by the following code -  
if((obj == null) || (obj.getClass() != this.getClass())) return false; // prefer  
This conditional check should be preferred instead of the conditional check given by -  
if(!(obj instanceof Test)) return false; // avoid  
This is because, the first condition (code in blue) ensures that it will return false if the argument is a subclass of the class Test. However, in case of the second condition (code in red) it fails. The instanceofoperator condition fails to return false if the argument is a subclass of the class Test. Thus, it might violate the symmetry requirement of the contract. The instanceof check is correct only if the class is final, so that no subclass would exist. The first condition will work for both, final and non-final classes. Note that, both these conditions will return false if the argument is null. The instanceof operator returns false if the left hand side (LHS) operand is null, irrespective of the operand on the right hand side (RHS) as specified by [JLS 15.20.2](http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#80289). However, the first condition should be preferred for better type checking.  
  
This class implements the equals method in such a way that it provides equals comparison only for the objects of the same class. Note that, this is not mandatory. But, if a class decides to provide equals comparison for other class objects, then the other class (or classes) must also agree to provide the same for this class so as to fulfill the symmetry and reflexivity requirements of the contract. This particular equals method implementation does not violate both these requirements. The lines 14 and 15 actually perform the equality comparison for the data members, and return true if they are equal. Line 15 also ensures that invoking the equals method on String variable data will not result in a NullPointerException.  
While implementing the equals method, primitives can be compared directly with an equality operator (==) after performing any necessary conversions (Such as float to Float.floatToIntBits or double to Double.doubleToLongBits). Whereas, object references can be compared by invoking their equals method recursively. You also need to ensure that invoking the equals method on these object references does not result in a NullPointerException.  
  
Here are some useful guidelines for implementing the equals method correctly.

1. Use the equality == operator to check if the argument is the reference to this object, if yes. return true. This saves time when actual comparison is costly.
2. Use the following condition to check that the argument is not null and it is of the correct type, if not then return false.  
   if((obj == null) || (obj.getClass() != this.getClass())) return false;  
   Note that, correct type does not mean the same type or class as shown in the example above. It could be any class or interface that one or more classes agree to implement for providing the comparison.
3. Cast the method argument to the correct type. Again, the correct type may not be the same class. Also, since this step is done after the above type-check condition, it will not result in a ClassCastException.
4. Compare significant variables of both, the argument object and this object and check if they are equal. If \*all\* of them are equal then return true, otherwise return false. Again, as mentioned earlier, while comparing these class members/variables; primitive variables can be compared directly with an equality operator (==) after performing any necessary conversions (Such as float to Float.floatToIntBitsor double to Double.doubleToLongBits). Whereas, object references can be compared by invoking their equals method recursively. You also need to ensure that invoking equals method on these object references does not result in a NullPointerException, as shown in the example above (Line 15).  
   It is neither necessary, nor advisable to include those class members in this comparison which can be calculated from other variables, hence the word "significant variables". This certainly improves the performance of the equals method. Only you can decide which class members are significant and which are not.
5. Do not change the type of the argument of the equals method. It takes a java.lang.Object as an argument, do not use your own class instead. If you do that, you will not be overriding the equals method, but you will be overloading it instead; which would cause problems. It is a very common mistake, and since it does not result in a compile time error, it becomes quite difficult to figure out why the code is not working properly.
6. Review your equals method to verify that it fulfills all the requirements stated by the general contract of the equals method.
7. Lastly, do not forget to override the hashCode method whenever you override the equals method, that's unpardonable. ;)

Now, let's examine the hashCode method of this example. At line 20, a non-zero constant value 7 (arbitrary) is assigned to an int variable hash. Since the class members/variables num and data do participate in the equals method comparison, they should also be involved in the calculation of the hash code. Though, this is not mandatory. You can use subset of the variables that participate in the equals method comparison to improve performance of the hashCode method. Performance of the hashCode method indeed is very important. But, you have to be very careful while selecting the subset. The subset should include those variables which are most likely to have the greatest diversity of the values. Sometimes, using all the variables that participate in the equals method comparison for calculating the hash code makes more sense.  
This class uses both the variables for computing the hash code. Lines 21 and 22 calculate the hash code values based on these two variables. Line 22 also ensures that invoking hashCode method on the variable data does not result in a NullPointerException if data is null. This implementation ensures that the general contract of the hashCode method is not violated. This implementation will return consistent hash code values for different invocations and will also ensure that equal objects will have equal hash codes.  
While implementing the hashCode method, primitives can be used directly in the calculation of the hash code value after performing any necessary conversions, such as float to Float.floatToIntBits or double to Double.doubleToLongBits. Since return type of the hashCode method is int, long values must to be converted to the integer values. As for hash codes of the object references, they should be calculated by invoking their hashCode method recursively. You also need to ensure that invoking the hashCode method on these object references does not result in a NullPointerException.  
  
Writing a very good implementation of the hashCode method which calculates hash code values such that the distribution is uniform is not a trivial task and may require inputs from mathematicians and theoretical computer scientist. Nevertheless, it is possible to write a decent and correct implementation by following few simple rules.  
  
Here are some useful guidelines for implementing the hashCode method correctly.

1. Store an arbitrary non-zero constant integer value (say 7) in an int variable, called hash.
2. Involve significant variables of your object in the calculation of the hash code, all the variables that are part of equals comparison should be considered for this. Compute an individual hash code int var\_code for each variable var as follows -
   1. If the variable(var) is byte, char, short or int, then var\_code = (int)var;
   2. If the variable(var) is long, then var\_code = (int)(var ^ (var >>> 32));
   3. If the variable(var) is float, then var\_code = Float.floatToIntBits(var);
   4. If the variable(var) is double, then -  
      long bits = Double.doubleToLongBits(var);  
      var\_code = (int)(bits ^ (bits >>> 32));
   5. If the variable(var) is boolean, then var\_code = var ? 1 : 0;
   6. If the variable(var) is an object reference, then check if it is null, if yes then var\_code = 0; otherwise invoke the hashCode method recursively on this object reference to get the hash code. This can be simplified and given as -  
      var\_code = (null == var ? 0 : var.hashCode());
3. Combine this individual variable hash code var\_code in the original hash code hash as follows -   
   hash = 31 \* hash + var\_code;
4. Follow these steps for all the significant variables and in the end return the resulting integer hash.
5. Lastly, review your hashCode method and check if it is returning equal hash codes for equal objects. Also, verify that the hash codes returned for the object are consistently the same for multiple invocations during the same execution.

The guidelines provided here for implementing equals and hashCode methods are merely useful as guidelines, these are not absolute laws or rules. Nevertheless, following them while implementing these two methods will certainly give you correct and consistent results. 

**Summary & Miscellaneous Tips**

* 1. Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.
  2. The equals method provides "deep comparison" by checking if two objects are logically equal as opposed to the "shallow comparison" provided by the equality operator ==.
  3. However, the equals method in java.lang.Object class only provides "shallow comparison", same as provided by the equality operator ==.
  4. The equals method only takes Java objects as an argument, and not primitives; passing primitives will result in a compile time error.
  5. Passing objects of different types to the equals method will never result in a compile time error or runtime error.
  6. For standard Java wrapper classes and for java.lang.String, if the equals argument type (class) is different from the type of the object on which the equals method is invoked, it will return false.
  7. The class java.lang.StringBuffer does not override the equals method, and hence it inherits the implementation from java.lang.Object class.
  8. The equals method must not provide equality comparison with any built in Java class, as it would result in the violation of the symmetry requirement stated in the general contract of the equals method.
  9. If null is passed as an argument to the equals method, it will return false.
  10. Equal hash codes do not imply that the objects are equal.
  11. return 1; is a legal implementation of the hashCode method, however it is a very bad implementation. It is legal because it ensures that equal objects will have equal hash codes, it also ensures that the hash code returned will be consistent for multiple invocations during the same execution. Thus, it does not violate the general contract of the hashCode method. It is a bad implementation because it returns same hash code for all the objects. This explanation applies to all implementations of the hashCode method which return same constant integer value for all the objects.
  12. In standard JDK 1.4, the wrapper classes java.lang.Short, java.lang.Byte, java.lang.Character and java.lang.Integer simply return the value they represent as the hash code by typecasting it to an int.
  13. Since JDK version 1.3, the class java.lang.String caches its hash code, i.e. it calculates the hash code only once and stores it in an instance variable and returns this value whenever the hashCode method is called. It is legal because java.lang.String represents an immutable string.
  14. It is incorrect to involve a random number directly while computing the hash code of the class object, as it would not consistently return the same hash code for multiple invocations during the same execution.

## **Instance Variable Hiding in Java**

In Java, if there a local variable in a method with same name as instance variable, then the local variable hides the instance variable. If we want to reflect the change made over to the instance variable, this can be achieved with the help of [this reference](http://geeksquiz.com/this-reference-in-java/).

|  |
| --- |
| class Test  {      // Instance variable or member variable      private int value = 10;        void method()      {          // This local variable hides instance variable          int value = 40;            System.out.println("Value of Instance variable :"                              + this.value);          System.out.println("Value of Local variable :"                              + value);      }  }    class UseTest  {      public static void main(String args[])      {          Test obj1 = new Test();          obj1.method();      }  } |

Output:

Value of Instance variable :10

Value of Local variable :40

## Static vs Dynamic Binding in Java

**Static Binding:**The binding which can be resolved at compile time by compiler is known as static or early binding. Binding of all the static, private and final methods is done at compile-time.

**Why binding of static, final and private methods is always a static binding?**

Static binding is better performance wise (no extra overhead is required). Compiler knows that all such methods **cannot be overridden** and will always be accessed by object of local class. Hence compiler doesn’t have any difficulty to determine object of class (local class for sure). That’s the reason binding for such methods is static.

Let’s see by an example:

|  |
| --- |
| public class NewClass  {      public static class superclass      {          static void print()          {              System.out.println("print in superclass.");          }      }      public static class subclass extends superclass      {          static void print()          {              System.out.println("print in subclass.");          }      }        public static void main(String[] args)      {          superclass A = new superclass();          superclass B = new subclass();          A.print();          B.print();      }  } |

Run on IDE

Before scrolling further down, guess the output of the above program?

**Output**:

print in superclass.

print in superclass.

As you can see, in both cases print method of superclass is called. Let’s see how this happens

* We have created one object of subclass and one object of superclass with the reference of the superclass.
* Since the print method of superclass is static, compiler knows that it will not be overridden in subclasses and hence compiler knows during compile time which print method to call and hence no ambiguity.

As an exercise, reader can change the reference of object B to subclass and then check the output.

**Dynamic Binding:**In Dynamic binding compiler doesn’t decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have same method.

Let’s see by an example

|  |
| --- |
| public class NewClass  {      public static class superclass      {          void print()          {              System.out.println("print in superclass.");          }      }        public static class subclass extends superclass      {          @Override          void print()          {              System.out.println("print in subclass.");          }      }        public static void main(String[] args)      {          superclass A = new superclass();          superclass B = new subclass();          A.print();          B.print();      }  } |

Run on IDE

**Output:**

print in superclass.

print in subclass.

Here the output differs. But why? Let’s break down the code and understand it thoroughly.

* Methods are not static in this code.
* During compilation, the compiler has no idea as to which print has to be called since compiler goes only by referencing variable not by type of object and therefore the binding would be delayed to runtime and therefore the corresponding version of print will be called based on type on object.

**Important Points**

* private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.
* Static binding uses Type information for binding while Dynamic binding uses Objects to resolve binding.
* Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e., at run time.

## Myth about the file name and class name in Java

The first lecture note given during java class is “In java file name and class name should be the same”. When the above law is violated a compiler error message will appear as below

|  |
| --- |
| /\*\*\*\*\* File name: Trial.java \*\*\*\*\*\*/  public class Geeks  {     public static void main(String[] args) {          System.out.println("Hello world");     }  } |

Run on IDE

Output:

javac Trial.java

Trial.java:9: error: class Geeks is public, should be

declared in a file named Geeks.java

public class Geeks

^

1 error

But the myth can be violated in such a way to compile the above file.

|  |
| --- |
| /\*\*\*\*\* File name: Trial.java \*\*\*\*\*\*/  class Geeks  {      public static void main(String[] args) {          System.out.println("Hello world");      }  } |

Run on IDE

Step 1: javac Trial.java

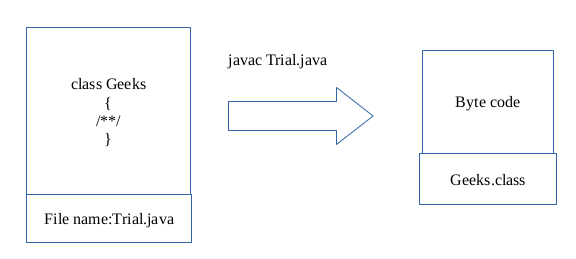
Step1 will create a Geeks.class (byte code) without any error message since the class is not public.

Step 2: java Geeks

Now the output will be **Hello world**

The myth about the file name and class name should be same only when the class is declared in  
**public**.

The above program works as follows:

[](https://www.geeksforgeeks.org/wp-content/uploads/javaex.png)

Now this .class file can be executed. By the above features some more miracles can be done. It is possible to have many classes in a java file. For debugging purposes this approach can be used. Each class can be executed separately to test their functionalities (only on one condition: Inheritance concept should not be used).

But in general it is good to follow the myth.

For example:

|  |
| --- |
| /\*\*\* File name: Trial.java \*\*\*/  class ForGeeks  {     public static void main(String[] args){        System.out.println("For Geeks class");     }  }    class GeeksTest  {     public static void main(String[] args){        System.out.println("Geeks Test class");     }  } |

Run on IDE

When the above file is compiled as **javac Trial.java**will create two **.class** files as **ForGeeks.class and GeeksTest.class**.

Since each class has separate main() stub they can be tested individually.

When **java ForGeeks** is executed the output is **For Geeks class.**   
When **java GeeksTest** is executed the output is **Geeks Test class**.

## **Why Java is not a purely Object-Oriented Language?**

Pure Object Oriented Language or Complete Object Oriented Language are Fully Object Oriented Language which supports or have features which treats everything inside program as objects. It doesn’t support primitive datatype(like int, char, float, bool, etc.). There are seven qualities to be satisfied for a programming language to be pure Object Oriented. They are:

1. Encapsulation/Data Hiding
2. Inheritance
3. Polymorphism
4. Abstraction
5. All predefined types are objects
6. All user defined types are objects
7. All operations performed on objects must be only through methods exposed at the objects.

Example: Smalltalk

**Why Java is not a Pure Object Oriented Language?**

Java supports property 1, 2, 3, 4 and 6 but fails to support property 5 and 7 given above. Java language is not a Pure Object Oriented Language as it contain these properties:

* **Primitive Data Type ex. int, long, bool, float, char, etc as Objects:** Smalltalk is a “pure” object-oriented programming language unlike Java and C++ as there is no difference between values which are objects and values which are primitive types. In Smalltalk, primitive values such as

integers, booleans and characters are also objects.

In Java, we have predefined types as non-objects (primitive types).

int a = 5;

System.out.print(a);

* **The static keyword:**When we declares a class as static then it can be used without the use of an object in Java. If we are using static function or static variable then we can’t call that function or variable by using dot(.) or class object defying object oriented feature.
* **Wrapper Class:** Wrapper class provides the mechanism to convert primitive into object and object into primitive. In Java, you can use Integer, Float etc. instead of int, float etc. We can communicate with objects without calling their methods. Ex. using arithmetic operators.

String s1 = "ABC" + "A" ;

Even using Wrapper classes does not make Java a pure OOP language, as internally it will use the operations like Unboxing and Autoboxing. So if you create instead of int Integer and do any mathematical operation on it, under the hoods Java is going to use primitive type int only.

|  |
| --- |
| public class BoxingExample  {      public static void main(String[] args)      {              Integer i = new Integer(10);              Integer j = new Integer(20);              Integer k = new Integer(i.intValue() + j.intValue());              System.out.println("Output: "+ k);      }  } |

**In the above code, there are 2 problems where Java fails to work as pure OOP:**

* 1. While creating Integer class you are using primitive type “int” i.e. numbers 10, 20.
  2. While doing addition Java is using primitive type “int”.

# Operators

## **Comparison of Autoboxed Integer objects in Java**

When we assign an integer value to an Integer object, the value is [autoboxed](http://docs.oracle.com/javase/tutorial/java/data/autoboxing.html)into an Integer object. For example the statement “Integer x = 10” creates an object ‘x’ with value 10.

Following are some interesting output questions based on comparison of Autoboxed Integer objects.

Predict the output of following Java Program

|  |
| --- |
| // file name: Main.java  public class Main {      public static void main(String args[]) {           Integer x = 400, y = 400;           if (x == y)              System.out.println("Same");           else              System.out.println("Not Same");      }  } |

Output:

Not Same

Since x and y refer to different objects, we get the output as “Not Same”

The output of following program is a surprise from Java.

|  |
| --- |
| // file name: Main.java  public class Main {      public static void main(String args[]) {           Integer x = 40, y = 40;           if (x == y)              System.out.println("Same");           else              System.out.println("Not Same");      }  } |

Output:

Same

In Java, values from -128 to 127 are cached, so the same objects are returned. The implementation of valueOf() uses cached objects if the value is between -128 to 127.

If we explicitly create Integer objects using new operator, we get the output as “Not Same”. See the following Java program. In the following program, valueOf() is not used.

|  |
| --- |
| // file name: Main.java  public class Main {      public static void main(String args[]) {            Integer x = new Integer(40), y = new Integer(40);           if (x == y)              System.out.println("Same");           else              System.out.println("Not Same");      }  } |

Output:

Not Same

Predict the output of the following program. This example is contributed by *Bishal Dubey*.

|  |
| --- |
| class GFG  {      public static void main(String[] args)      {      Integer X = new Integer(10);      Integer Y = 10;        // Due to auto-boxing, a new Wrapper object      // is created which is pointed by Y      System.out.println(X == Y);      }  } |

Output:

false

**Explanation:**Two objects will be created here. First object which is pointed by X due to calling of new operator and second object will be created because of Auto-boxing.

## Addition and Concatenation in Java

**Try to predict the output of following code:**

|  |
| --- |
| public class Geeksforgeeks  {      public static void main(String[] args)      {          System.out.println(2+0+1+6+"GeeksforGeeks");          System.out.println("GeeksforGeeks"+2+0+1+6);          System.out.println(2+0+1+5+"GeeksforGeeks"+2+0+1+6);          System.out.println(2+0+1+5+"GeeksforGeeks"+(2+0+1+6));      }  } |

**Explanation:**

The output is

9GeeksforGeeks  
GeeksforGeeks2016  
8GeeksforGeeks2016  
8GeeksforGeeks9

This unpredictable output is due the fact that the compiler evaluates the given expression from left to right given that the operators have same precedence. Once it encounters the String, it considers the rest of the expression as of a String (again based on the precedence order of the expression).

* **System.out.println(2 + 0 + 1 + 6 + “GeeksforGeeks”);**  *// It prints the addition of 2,0,1 and 6 which is equal to 9*
* **System.out.println(“GeeksforGeeks” + 2 + 0 + 1 + 6);**  *//It prints the concatenation of 2,0,1 and 6 which is 2016  since it encounters the string initially. Basically, Strings take precedence because they have a higher casting priority than integers do.*
* **System.out.println(2 + 0 + 1 + 5 + “GeeksforGeeks” + 2 + 0 + 1 + 6);***//It prints the addition of 2,0,1 and 5 while the concatenation of 2,0,1 and 6 based on the above given examples.*
* **System.out.println(2 + 0 + 1 + 5 + “GeeksforGeeks” + (2 + 0 + 1 + 6));** *//It prints the addition of  both 2,0,1 and 5  and 2,0,1 and 6 based due the precedence of ( ) over +. Hence expression in ( ) is calculated first and then the further evaluation takes place.*

# Strings in Java

## Swap two Strings without using third user defined variable in Java

Given two string variables a and b, swap these variables without using temporary or third variable in Java. Use of library methods is allowed.

**Algorithm:**

1) Append second string to first string and

store in first string:

a = a + b

2) call the method substring(int beginindex, int endindex)

by passing beginindex as 0 and endindex as,

a.length() - b.length():

b = substring(0,a.length()-b.length());

3) call the method substring(int beginindex) by passing

b.length() as argument to store the value of initial

b string in a

a = substring(b.length());

|  |
| --- |
| // Java program to swap two strings without using a temporary  // variable.  import java.util.\*;    class Swap  {      public static void main(String args[])      {          // Declare two strings          String a = "Hello";          String b = "World";            // Print String before swapping          System.out.println("Strings before swap: a = " +                                         a + " and b = "+b);            // append 2nd string to 1st          a = a + b;            // store intial string a in string b          b = a.substring(0,a.length()-b.length());            // store initial string b in string a          a = a.substring(b.length());            // print String after swapping          System.out.println("Strings after swap: a = " +                                       a + " and b = " + b);      }  } |

Run on IDE

Output:

Strings before swap: a = Hello and b = World

Strings after swap: a = World and b = Hello

* Objects of String are immutable, and objects of StringBuffer and StringBuilder are mutable.
* StringBuffer and StringBuilder are similar, but StringBuilder is faster and preferred over StringBuffer for single threaded program. If thread safety is needed, then StringBuffer is used.

## Reverse a string in Java

**Converting String into Bytes:**getBytes() method is used to convert the input string into bytes[].  
**Method:**

1. Create a temporary byte[] of length equal

to the length of the input string.

2. Store the bytes (which we get by using

getBytes() method) in reverse order into

the temporary byte[] .

3. Create a new String abject using byte[] to

store result.

|  |
| --- |
| // Java program to ReverseString using ByteArray.  import java.lang.\*;  import java.io.\*;  import java.util.\*;    // Class of ReverseString  class ReverseString  {      public static void main(String[] args)      {          String input = "GeeksforGeeks";            // getBytes() method to convert string          // into bytes[].          byte [] strAsByteArray = input.getBytes();            byte [] result =                     new byte [strAsByteArray.length];            // Store result in reverse order into the          // result byte[]          for (int i = 0; i<strAsByteArray.length; i++)              result[i] =               strAsByteArray[strAsByteArray.length-i-1];            System.out.println(new String(result));      }  } |

Run on IDE

Output:

skeeGrofskeeG

## Remove Trailing Zeros from String in Java

We use [StringBuffer](https://www.geeksforgeeks.org/g-fact-27-string-vs-stringbuilder-vs-stringbuffer/) class as [Strings are immutable](https://www.geeksforgeeks.org/string-class-in-java/).

* Count trailing zeros.
* Use StringBuffer replace function to remove characters equal to above count.

|  |
| --- |
| // Java program to remove trailing/preceding zeros  // from a given string  import java.util.Arrays;  import java.util.List;    /\* Name of the class to remove trailing/preceding zeros \*/  class RemoveZero  {      public static String removeZero(String str)      {          // Count trailing zeros          int i = 0;          while (str.charAt(i) == '0')              i++;            // Convert str into StringBuffer as Strings          // are immutable.          StringBuffer sb = new StringBuffer(str);            // The  StringBuffer replace function removes          // i characters from given index (0 here)          sb.replace(0, i, "");            return sb.toString();  // return in String      }        // Driver code      public static void main (String[] args)      {          String str = "00000123569";          str = removeZero(str);          System.out.println(str);      }  } |

Run on IDE

Output:

123569

## Counting number of lines, words, characters and paragraphs in a text file using Java

Counting the number of characters is important because almost all the text boxes that rely on user input have certain limit on the number of characters that can be inserted. For example, the character limit on a Facebook post is 63, 206 characters. Whereas, for a tweet on Twitter the character limit is 140 characters and the character limit is 80 per post for Snapchat.

Determining character limits become crucial when the tweet and Facebook post updates are being done through api’s.

**Note**: This program would not run on online compilers. Please make a txt file on your system and give its path to run this program on your system.

|  |
| --- |
| // Java program to count the  // number of charaters in a file  import java.io.\*;    public class Test  {      public static void main(String[] args) throws IOException      {          File file = new File("C:\\Users\\Mayank\\Desktop\\1.txt");          FileInputStream fileStream = new FileInputStream(file);          InputStreamReader input = new InputStreamReader(fileStream);          BufferedReader reader = new BufferedReader(input);            String line;            // Initializing counters          int countWord = 0;          int sentenceCount = 0;          int characterCount = 0;          int paragraphCount = 1;          int whitespaceCount = 0;            // Reading line by line from the          // file until a null is returned          while((line = reader.readLine()) != null)          {              if(line.equals(""))              {                  paragraphCount++;              }              if(!(line.equals("")))              {                    characterCount += line.length();                    // \\s+ is the space delimiter in java                  String[] wordList = line.split("\\s+");                    countWord += wordList.length;                  whitespaceCount += countWord -1;                    // [!?.:]+ is the sentence delimiter in java                  String[] sentenceList = line.split("[!?.:]+");                    sentenceCount += sentenceList.length;              }          }            System.out.println("Total word count = " + countWord);          System.out.println("Total number of sentences = " + sentenceCount);          System.out.println("Total number of characters = " + characterCount);          System.out.println("Number of paragraphs = " + paragraphCount);          System.out.println("Total number of whitespaces = " + whitespaceCount);      }  } |

Run on IDE

Output:

Total word count = 5

Total number of sentences = 3

Total number of characters = 21

Number of paragraphs = 2

Total number of whitespaces = 7

# Useful and/or Advanced Features

## Generics in Java

Generics in Java is similar to [templates in C++](http://geeksquiz.com/templates-cpp/). The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

**Generic Class**

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

// To create an instance of generic class

BaseType <Type> obj = new BaseType <Type>()

**Note:** In Parameter type we can not use primitives like

'int','char' or 'double'.

|  |
| --- |
| // A Simple Java program to show working of user defined  // Generic classes    // We use < > to specify Parameter type  class Test<T>  {      // An object of type T is declared      T obj;      Test(T obj) {  this.obj = obj;  }  // constructor      public T getObject()  { return this.obj; }  }    // Driver class to test above  class Main  {      public static void main (String[] args)      {          // instance of Integer type          Test <Integer> iObj = new Test<Integer>(15);          System.out.println(iObj.getObject());            // instance of String type          Test <String> sObj =                            new Test<String>("GeeksForGeeks");          System.out.println(sObj.getObject());      }  } |

Run on IDE

Output:

15

GeeksForGeeks

We can also pass multiple Type parameters in Generic classes.

|  |
| --- |
| // A Simple Java program to show multiple  // type parameters in Java Generics    // We use < > to specify Parameter type  class Test<T, U>  {      T obj1;  // An object of type T      U obj2;  // An object of type U        // constructor      Test(T obj1, U obj2)      {          this.obj1 = obj1;          this.obj2 = obj2;      }        // To print objects of T and U      public void print()      {          System.out.println(obj1);          System.out.println(obj2);      }  }    // Driver class to test above  class Main  {      public static void main (String[] args)      {          Test <String, Integer> obj =              new Test<String, Integer>("GfG", 15);            obj.print();      }  } |

Run on IDE

Output:

GfG

15

**Generic Functions:**

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

|  |
| --- |
| // A Simple Java program to show working of user defined  // Generic functions    class Test  {      // A Generic method example      static <T> void genericDisplay (T element)      {          System.out.println(element.getClass().getName() +                             " = " + element);      }        // Driver method      public static void main(String[] args)      {           // Calling generic method with Integer argument          genericDisplay(11);            // Calling generic method with String argument          genericDisplay("GeeksForGeeks");            // Calling generic method with double argument          genericDisplay(1.0);      }  } |

Run on IDE

Output:

java.lang.Integer = 11

java.lang.String = GeeksForGeeks

java.lang.Double = 1.0

**Advantages of Generics:**

Programs that uses Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use for any type we want.
2. Type Safety: Generics make errors to appear compile time than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

|  |
| --- |
| // A Simple Java program to demonstrate that NOT using  // generics can cause run time exceptions  import java.util.\*;    class Test  {      public static void main(String[] args)      {          // Creatinga an ArrayList without any type specified          ArrayList al = new ArrayList();            al.add("Sachin");          al.add("Rahul");          al.add(10); // Compiler allows this            String s1 = (String)al.get(0);          String s2 = (String)al.get(1);            // Causes Runtime Exception          String s3 = (String)al.get(2);      }  } |

Run on IDE

Output :

Exception in thread "main" java.lang.ClassCastException:

java.lang.Integer cannot be cast to java.lang.String

at Test.main(Test.java:19)

**How generics solve this problem?**

At the time of defining ArrayList, we can specify that this list can take only String objects.

|  |
| --- |
| // Using generics converts run time exceptions into  // compile time exception.  import java.util.\*;    class Test  {      public static void main(String[] args)      {          // Creating a an ArrayList with String specified          ArrayList <String> al = new ArrayList<String> ();            al.add("Sachin");          al.add("Rahul");            // Now Compiler doesn't allow this          al.add(10);            String s1 = (String)al.get(0);          String s2 = (String)al.get(1);          String s3 = (String)al.get(2);      }  } |

Run on IDE

Output:

15: error: no suitable method found for add(int)

al.add(10);

^

1. Individual Type Casting is not needed: If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

|  |
| --- |
| // We don't need to typecast individual members of ArrayList  import java.util.\*;    class Test  {      public static void main(String[] args)      {          // Creating a an ArrayList with String specified          ArrayList <String> al = new ArrayList<String> ();            al.add("Sachin");          al.add("Rahul");            // Typecasting is not needed          String s1 = al.get(0);          String s2 = al.get(1);      }  } |

1. Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.

## Wildcards in Java

The question mark (?) is known as the wildcard in generic programming . It represents an unknown type. The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type. Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly. This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.

**Types of wildcards in Java:**

**1. Upper Bounded Wildcards:**These wildcards can be used when you want to relax the restrictions on a variable. For example, say you want to write a method that works on List < integer >, List < double >, and List < number > , you can do this  using an upper bounded wildcard.  
To declare an upper-bounded wildcard, use the wildcard character (‘?’), followed by the extends keyword, followed by its upper bound.

public static void add(List<? extends Number> list)

**Implementation:**

|  |
| --- |
| //Java program to demonstrate Upper Bounded Wildcards  import java.util.Arrays;  import java.util.List;    class WildcardDemo  {      public static void main(String[] args)      {            //Upper Bounded Integer List          List<Integer> list1= Arrays.asList(4,5,6,7);            //printing the sum of elements in list          System.out.println("Total sum is:"+sum(list1));            //Double list          List<Double> list2=Arrays.asList(4.1,5.1,6.1);            //printing the sum of elements in list          System.out.print("Total sum is:"+sum(list2));      }        private static double sum(List<? extends Number> list)      {          double sum=0.0;          for (Number i: list)          {              sum+=i.doubleValue();          }            return sum;      }  } |

**Output:**

Total sum is:22.0

Total sum is:15.299999999999999

In the above program, list1 and list2 are objects of the List class. list1 is a collection of Integer and list2 is a collection of Double. Both of them are being passed to method sum which has a wildcard that extends Number. This means that list being passed can be of any field or subclass of that field. Here, Integer and Double are subclasses of class Number.

**2.Lower Bounded Wildcards:**It is expressed using the wildcard character (‘?’), followed by the super keyword, followed by its lower bound: <? super A>.

Syntax: Collectiontype <? super A>

**Implementation:**

|  |
| --- |
| //Java program to demonstrate Lower Bounded Wildcards  import java.util.Arrays;  import java.util.List;    class WildcardDemo  {      public static void main(String[] args)      {          //Lower Bounded Integer List          List<Integer> list1= Arrays.asList(4,5,6,7);            //Integer list object is being passed          printOnlyIntegerClassorSuperClass(list1);            //Number list          List<Number> list2= Arrays.asList(4,5,6,7);            //Integer list object is being passed          printOnlyIntegerClassorSuperClass(list2);      }        public static void printOnlyIntegerClassorSuperClass(List<? super Integer> list)      {          System.out.println(list);      }  } |

Run on IDE

**Output:**

[4, 5, 6, 7]

[4, 5, 6, 7]

Here arguments can be Integer or superclass of Integer(which is Number). The method printOnlyIntegerClassorSuperClass will only take Integer or its superclass objects. However if we pass list of type Double then we will get compilation error. It is because only the Integer field or its superclass can be passed. Double is not the superclass of Integer.

Use extend wildcard when you want to get values out of a structure and super wildcard when you put values in a structure. Don’t use wildcard when you get and put values in a structure.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

**3.Unbounded Wildcard:** This wildcard type is specified using the wildcard character (?), for example, List. This is called a list of unknown type. These are useful in the following cases

* When writing a method which can be employed using functionality provided in Object class.
* When the code is using methods in the generic class that don’t depend on the type parameter

**Implementation:**

|  |
| --- |
| //Java program to demonstrate Unbounded wildcard  import java.util.Arrays;  import java.util.List;    class unboundedwildcardemo  {      public static void main(String[] args)      {            //Integer List          List<Integer> list1= Arrays.asList(1,2,3);            //Double list          List<Double> list2=Arrays.asList(1.1,2.2,3.3);            printlist(list1);            printlist(list2);      }        private static void printlist(List<?> list)      {            System.out.println(list);      }  } |

Run on IDE

Output:

[1, 2, 3]

[1.1, 2.2, 3.3]

# Spring Boot: Creating Microservices on Java

Welcome, reader. In this article, we will talk about an interesting architectural model, microservices architecture, in addition to studying one of the new features of Spring 4.0, Spring Boot. But after all, what are microservices?

**Microservices**

In the development of large systems, it is common to develop various components and libraries that implement various functions, ranging from the implementation of business requirements to technical tasks, such as an XML parser, for example. In these scenarios, several components are reused by different interfaces and / or systems. Imagine, for example, a component that implements a register of customers and we package this component in a java project, which generates his deliverable as a .jar.

In this scenario, we could have several interfaces to use this component, such as web applications, mobile, EJBs, etc. In the traditional form of Java implementation, we would package this jar in several other packages, such as EAR files, WAR, etc. Imagine now that a problem in the customer register is found. In this scenario, we have a considerable operational maintenance work, since as well as the correction on the component, we would have to make the redeploy of all consumer applications due to the component to be packaged inside the other deployment packages.

In order to propose a solution to this issue, was born microservices architecture model. In this architectural model, rather than package the jar files into consumer systems, the components are independently exposed in the form of remote accessible APIs, consumed using protocols such as HTTP, for example.

An important point to note in the above explanations, is that although we are exemplifying the model using the Java world, the same principles can be applied to other technologies such as C #.

Spring Boot

Among the new features in version 4.0 of the Spring Framework, a new project that has arisen is the Spring Boot.O goal of Spring Boot is to provide a way to provide Java applications quickly and simply, through an embedded server - by default it uses an embedded version of tomcat - thus eliminating the need of Java EE containers. With Spring Boot, we can expose components such as REST services independently, exactly as proposed in microservices architecture, so that in any maintenance of the components, we no longer make the redeploy of all the system.

So without further delay, let's begin our hands-on. For this lab, we will use the Eclipse Luna and Maven 3.

To illustrate the concept of microservices, we will create 3 Maven projects in this hands-on: each of them will symbolize back-end functionality, i.e. reusable APIs, and one of them held a composition, that is, will be a consumer of the other 2.

To begin, let's create 3 simple Maven projects without defined archetype, and let's call them Product-backend, Customer-backend and Order-backend. In the poms of the 3 projects, we will add the dependencies for the creation of our REST services and startup Spring Boot, as we can see below:

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.2.0.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jersey</artifactId>

</dependency>

</dependencies>

With the dependencies established, we start coding. The first class that we create, that we call Application, will be identical in all three projects, because it only works as an initiator to Spring Boot - as defined by @SpringBootApplication annotation - rising a Spring context and the embedded server:

package br.com.alexandreesl.handson;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Aplication.class, args);

}

}

The next class we will see is the ApplicationConfig. In this class, which uses the @Configuration Spring annotation to indicate to the framework that it is a resource configuration class, we set the Jersey, which is our ResourceManager responsible for exposing REST services for the consumers.

In a real application, this class would be also creating datasources for access to databases and other resources, but in order to keep it simple enough to be able to focus on the Spring Boot, we will use mocks to represent the data access.

package br.com.alexandreesl.handson;

import javax.inject.Named;

import org.glassfish.jersey.server.ResourceConfig;

import org.springframework.context.annotation.Configuration;

@Configuration

public class ApplicationConfig {

@Named

static class JerseyConfig extends ResourceConfig {

public JerseyConfig() {

this.packages(“br.com.alexandreesl.handson.rest”);

}

}

}

The above class will be used identically in the projects relating to customers and products. For the orders, however, since it will be a consumer of other services, we will use this class with a slight difference, as we will also instantiate a RestTemplate. This class, one of the new features in the Spring Framework, is a standardized and very simple interface that facilitates the consumption of REST services. The class to use in the Order-backend project can be seen below:

package br.com.alexandreesl.handson;

import javax.inject.Named;

import org.glassfish.jersey.server.ResourceConfig;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class ApplicationConfig {

@Named

static class JerseyConfig extends ResourceConfig {

public JerseyConfig() {

this.packages(“br.com.alexandreesl.handson.rest”);

}

}

@Bean

public RestTemplate restTemplate() {

RestTemplate restTemplate = new RestTemplate();

return restTemplate;

}

}

Finally, we will start the implementation of the REST services themselves. In the project responsible for customer features (Customer-backend), we create a class of DTO and a REST service. The class, which is a simple POJO:

package br.com.alexandreesl.handson.rest;

public class Customer {

private long id;

private String name;

private String email;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setNome(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

The REST service, in turn, has only 2 capabilities, a search of all customers and other that query a customer from his id:

package br.com.alexandreesl.handson.rest;

import java.util.ArrayList;

import java.util.List;

import javax.inject.Named;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.QueryParam;

import javax.ws.rs.core.MediaType;

@Named

@Path(“/”)

public class CustomerRest {

private static List<Customer> Customers = new ArrayList<Customer>();

static {

Customer customer1 = new Customer();

customer1.setId(1);

customer1.setNome(“Customer 1″);

customer1.setEmail(“customer1@gmail.com”);

Customer customer2 = new Customer();

customer2.setId(2);

customer2.setNome(“Customer 2″);

customer2.setEmail(“Customer2@gmail.com”);

Customer customer3 = new Customer();

customer3.setId(3);

customer3.setNome(“Customer 3″);

customer3.setEmail(“Customer3@gmail.com”);

Customer customer4 = new Customer();

customer4.setId(4);

customer4.setNome(“Customer 4″);

customer4.setEmail(“Customer4@gmail.com”);

Customer customer5 = new Customer();

customer5.setId(5);

customer5.setNome(“Customer 5″);

customer5.setEmail(“Customer5@gmail.com”);

customers.add(customer1);

customers.add(customer2);

customers.add(customer3);

customers.add(customer4);

Customers.add(customer5);

}

@GET

@Produces(MediaType.APPLICATION\_JSON)

public List<Customer> getCustomers() {

return customers;

}

@GET

@Path(“customer”)

@Produces(MediaType.APPLICATION\_JSON)

public Customer getCustomer(@QueryParam(“id”) long id) {

Customer cli = null;

for (Customer c : customers) {

if (c.getId() == id)

cli = c;

}

return cli;

}

}

And that concludes our REST customers service. For products, analogous to customers, we have the methods to search all products or a product through one of his ids and finally we have the orders service, which through a submitOrder method gets the data of a product and a customer - whose keys are passed as parameters to the method - and returns a order header. The classes that make up our services are the following:

package br.com.alexandreesl.handson.rest;

public class Product {

private long id;

private String sku;

private String description;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getSku() {

return sku;

}

public void setSku(String sku) {

this.sku = sku;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

}

package br.com.alexandreesl.handson.rest;

import java.util.ArrayList;

import java.util.List;

import javax.inject.Named;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.QueryParam;

import javax.ws.rs.core.MediaType;

@Named

@Path(“/”)

public class ProductRest {

private static List<Product> products = new ArrayList<Product>();

static {

Product product1 = new Product();

product1.setId(1);

product1.setSku(“abcd1″);

product1.setDescricao(“Product1″);

Product product2 = new Product();

product2.setId(2);

product2.setSku(“abcd2″);

product2.setDescricao(“Product2″);

Product product3 = new Product();

product3.setId(3);

product3.setSku(“abcd3″);

product3.setDescricao(“Product3″);

Product product4 = new Product();

product4.setId(4);

product4.setSku(“abcd4″);

product4.setDescricao(“Product4″);

products.add(product1);

products.add(product2);

products.add(product3);

products.add(product4);

}

@GET

@Produces(MediaType.APPLICATION\_JSON)

public List<Product> getProducts() {

return products;

}

@GET

@Path(“product”)

@Produces(MediaType.APPLICATION\_JSON)

public Product getProduct(@QueryParam(“id”) long id) {

Product prod = null;

for (Product p : products) {

if (p.getId() == id)

prod = p;

}

return prod;

}

}

Finally, the classes that make up our aforementioned order service in the Order-backend project are:

package br.com.alexandreesl.handson.rest;

import java.util.Date;

public class Order {

private long id;

private long amount;

private Date dateOrder;

private Customer customer;

private Product product;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public long getAmount() {

return amount;

}

public void setAmount(long amount) {

this.amount = amount;}

public Date getDateOrder() {

return dateOrder;

}

public void setDateOrder(Date dateOrder) {

this.dateOrder = dateOrder;

}

public Customer getCustomer() {

return customer;

}

public void setCustomer(Customer customer) {

this.customer = customer;

}

public Product getProduct() {

return product;

}

public void setProduct(Product product) {

this.product = product;

}

}

package br.com.alexandreesl.handson.rest;

import java.util.Date;

import javax.inject.Inject;

import javax.inject.Named;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.QueryParam;

import javax.ws.rs.core.MediaType;

import org.springframework.web.client.RestTemplate;

@Named

@Path(“/”)

public class OrderRest {

private long id = 1;

@Inject

private RestTemplate restTemplate;

@GET

@Path(“order”)

@Produces(MediaType.APPLICATION\_JSON)

public Order submitOrder(@QueryParam(“idCustomer”) long idCustomer,

@QueryParam(“idProduct”) long idProduct,

@QueryParam(“amount”) long amount) {

Order order = new Order();

Customer customer = restTemplate.getForObject(

“http://localhost:8081/customer?id={id}”, Customer.class,

idCustomer);

Product product = restTemplate.getForObject(

“http://localhost:8082/product?id={id}”, Product.class,

idProduct);

order.setCustomer(customer);

order.setProduct(product);

order.setId(id);

order.setAmount(amount);

order.setDataOrder(new Date());

id++;

return order;

}

}

The reader should note the use of product and customer classes in our order service. Such classes, however, are not direct references to the ones implemented in other projects, but classes "cloned" of the original, within the order project. This apparent duplication of code in the DTO classes, sure to be a negative aspect of the solution, which can be seen as similar as the stubs classes we see in JAX-WS clients, must be measured carefully as it can be considered a small price to pay, compared to the impact we see if we make the coupling of the projects.

A half solution that can minimize this problem is to create a unique project for the domain classes, which would be imported by all other projects, as the domain classes must undergo a number of much lower maintenance than the services. I leave it to the reader to assess the best option, according to the characteristics of their projects.

Good, but after all this coding, let’s get down to, which is to test our services!

To begin, let's start our REST services. For this, we create run configurations in Eclipse where we will add a system property, which specify the port where the spring boot will start the services. In my environment, I started the customer service on port 8081, the products in 8082 and the orders on port 8083, but the reader is free to use the most appropriate ports for his environment. The property to be used to configure the port is:

-Dserver.port=8081

**NOTE:** If the reader change the ports, it must correct the ports of the calls on the order service code.

With properly configured run configurations, we will start processing and test the calls to our REST. Simply click the icon and select to run each run configuration created, one at a time, which will generate 3 different consoles running in the Eclipse console window. As the reader can see, when we start a project, Spring Boot generates a boot log, where you can see the embedded tomcat and its associated resources, such as Jersey, being initialized:

To make just one example, let’s call the Order Service. If we call the following URL:

http://localhost:8083/order?idCustomer=2&idProduct=3&amount=4

We produce the following JSON, representing the header of a order:

{"id":1,"amount":4,"dateOrder":1419187358576,"customer":{"id":2,"name":"Customer 2","email":"customer2@gmail.com"},"product":{"id":3,"sku":"abcd3","description":"Product3"}}

At this point, the reader may notice a bug in our order service: subsequent calls will generate the same ID order! This is due to our mock variable that generates the ids be declared as a global variable that is recreated every new instance of the class. As REST services have request scope, every request generates a new instance, which means that the variable is never incremented through the calls. One of the simplest ways of fixing this bug is declaring the variable as static, but before we do that, let's take a moment to think about the fact that we have implemented our projects as microservices - yes, they are microservices! - Can help us in our maintenance:

- If we were in a traditional implementation, each of these components would be a jar file encapsulated within a client application such as a web application (WAR);

- Thus, for fixing this bug, not only we would have to correct the order project, but we would also redeploy the product project, the customer project and the web application itself! The advantages become even more apparent if we consider that the application would have many more features in addition to the problematic feature, so to correct one feature, we would have to perform the redeploy of all others, causing a complete unavailability of our system during reimplantation;

So, having realized the advantages of our construction format, we will initiate the maintenance. During the procedure, we will make the restart of our order service in order to demonstrate how microservices do not affect each other's availability.

To begin our maintenance, we will terminate the Spring Boot of the order process. To do this, we simply select the corresponding console window and terminate. After the stop, if we call the URL of the order service, we have the following error message, indicating the unavailability.

However, if we try to make the product and customer service calls, we see that both are operational, proving the independence.

Then we make the maintenance, changing the variable to the static type:

private static long id = 1;

Finally, we perform a restart of the order service with the implemented correction. If we run several calls to the URL, we see that the service is generating orders with different IDs, proving that the fix was a success:

{"id":9,"amount":4,"dateOrder":1419187358576,"customer":{"id":2,"name":"Customer 2","email":"customer2@gmail.com"},"product":{"id":3,"sku":"abcd3","description":"Product3"}}

We realize, with this simple example, that the independence of such implementations with microservices brought us a powerful architecture: you can undeploy, correct / evolve and deploy new versions of parts of a system, without thereby requiring the redeployment of the whole system and it’s totally unfeasibility.

**Conclusion**

And so we conclude our hands-on. With a simple implementation, but powerful, Spring Boot is a good option to implement a microservices architecture and it must be evaluated throughout java architects or developers who wants to promote this model in their demands. Thanks to everyone who supported me in this hands-on, until next time.

# Design Patterns

Design pattern is a general reusable solution or template to a commonly occurring problem in software design. The patterns typically show relationships and interactions between classes or objects. The idea is to speed up the development process by providing tested, proven development paradigm.

**Goal:**  
•Understand the problem and matching it with some pattern.

• Reusage of old interface or making the present design reusable for the future usage.

**Types of Design Patterns**

There are mainly three types of design patterns:

1. **Creational**

These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are Factory Method, Abstract Factory, Builder, Singleton, Object Pool, Prototype and Singleton.

1. **Structural**

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

1. **Behavioural**

Behavioural patterns are about identifying common communication patterns between objects and realize these patterns. Behavioural patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor

## Factory Design Pattern in Java

**Factory Pattern** is one of the **Creational Design pattern** and it’s widely used in JDK as well as frameworks like Spring and Struts.

Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class.

**Factory Design Pattern Super Class**

Super class in factory design pattern can be an interface, [abstract class](https://www.journaldev.com/1582/abstract-class-in-java) or a normal java class. For our factory design pattern example, we have abstract super class with [overridden](https://www.journaldev.com/817/java-override-annotation) **toString()** method for testing purpose.

package com.journaldev.design.model;

public abstract class Computer {

public abstract String getRAM();

public abstract String getHDD();

public abstract String getCPU();

@Override

public String toString(){

return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

}

}

**Factory Design Pattern Sub Classes**

Let’s say we have two sub-classes PC and Server with below implementation.

package com.journaldev.design.model;

public class PC extends Computer {

private String ram;

private String hdd;

private String cpu;

public PC(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

Notice that both the classes are extending **Computer** super class.

package com.journaldev.design.model;

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

**Factory Class**

Now that we have super classes and sub-classes ready, we can write our factory class. Here is the basic implementation.

package com.journaldev.design.factory;

import com.journaldev.design.model.Computer;

import com.journaldev.design.model.PC;

import com.journaldev.design.model.Server;

public class ComputerFactory {

public static Computer getComputer(String type, String ram, String hdd, String cpu){

if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);

else if("Server".equalsIgnoreCase(type)) return new Server(ram, hdd, cpu);

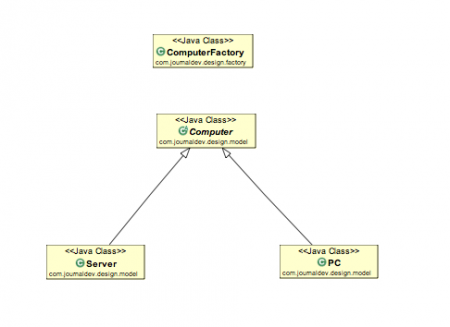
return null;

}

}

**Some important points about Factory Design Pattern method are;**

* We can keep Factory class Singleton or we can keep the method that returns the subclass as [static](https://www.journaldev.com/1365/static-keyword-in-java).
* Notice that based on the input parameter, different subclass is created and returned. getComputer is the factory method.

[](https://cdn.journaldev.com/wp-content/uploads/2013/05/factory-pattern-java.png)

Here is a simple test client program that uses above factory design pattern implementation.

package com.journaldev.design.test;

import com.journaldev.design.factory.ComputerFactory;

import com.journaldev.design.model.Computer;

public class TestFactory {

public static void main(String[] args) {

Computer pc = ComputerFactory.getComputer("pc","2 GB","500 GB","2.4 GHz");

Computer server = ComputerFactory.getComputer("server","16 GB","1 TB","2.9 GHz");

System.out.println("Factory PC Config::"+pc);

System.out.println("Factory Server Config::"+server);

}

}

**Output of above program is:**

Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

**Factory Design Pattern Advantages**

* Factory design pattern provides approach to code for interface rather than implementation.
* Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
* Factory pattern provides abstraction between implementation and client classes through inheritance.

**Factory Design Pattern Examples in JDK**

* java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern.
* valueOf() method in wrapper classes like Boolean, Integer etc.

## Decorator Design Pattern

**Decorator design pattern** is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behaviour. Decorator design pattern is one of the structural design pattern (such as [Adapter Pattern](https://www.journaldev.com/1487/adapter-design-pattern-java), [Bridge Pattern](https://www.journaldev.com/1491/bridge-design-pattern-java), [Composite Pattern](https://www.journaldev.com/1535/composite-design-pattern-in-java)) and uses abstract classes or interface with [composition](https://www.journaldev.com/1325/composition-in-java-example) to implement.

We use [inheritance](https://www.journaldev.com/644/inheritance-java-example) or composition to extend the behaviour of an object but this is done at compile time and it’s applicable to all the instances of the class. We can’t add any new functionality of remove any existing behaviour at runtime – this is when Decorator pattern comes into picture.

Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.

But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now imagine if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern in java.

We need to have following types to implement decorator design pattern.

1. **Component Interface** – The interface or [abstract class](https://www.journaldev.com/1582/abstract-class-in-java) defining the methods that will be implemented. In our case Car will be the component interface.

package com.journaldev.design.decorator;

public interface Car {

public void assemble();

}

1. **Component Implementation** – The basic implementation of the component interface. We can have BasicCar class as our component implementation.

package com.journaldev.design.decorator;

public class BasicCar implements Car {

@Override

public void assemble() {

System.out.print("Basic Car.");

}

}

1. Decorator**– Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.**

package com.journaldev.design.decorator;

public class CarDecorator implements Car {

protected Car car;

public CarDecorator(Car c){

this.car=c;

}

@Override

public void assemble() {

this.car.assemble();

}

}

1. **Concrete Decorators** – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.

package com.journaldev.design.decorator;

public class SportsCar extends CarDecorator {

public SportsCar(Car c) {

super(c);

}

@Override

public void assemble(){

super.assemble();

System.out.print(" Adding features of Sports Car.");

}

}

package com.journaldev.design.decorator;

public class LuxuryCar extends CarDecorator {

public LuxuryCar(Car c) {

super(c);

}

@Override

public void assemble(){

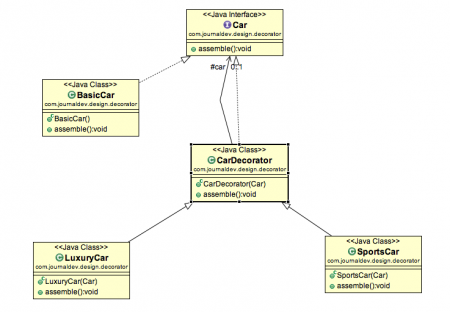
super.assemble();

System.out.print(" Adding features of Luxury Car.");

}

}

**Decorator Design Pattern – Class Diagram**

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/decorator-pattern.png)

**Decorator Design Pattern Test Program**

package com.journaldev.design.test;

import com.journaldev.design.decorator.BasicCar;

import com.journaldev.design.decorator.Car;

import com.journaldev.design.decorator.LuxuryCar;

import com.journaldev.design.decorator.SportsCar;

public class DecoratorPatternTest {

public static void main(String[] args) {

Car sportsCar = new SportsCar(new BasicCar());

sportsCar.assemble();

System.out.println("\n\*\*\*\*\*");

Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));

sportsLuxuryCar.assemble();

}

}

Notice that client program can create different kinds of Object at runtime and they can specify the order of execution too.

Output of above test program is:

Basic Car. Adding features of Sports Car.

\*\*\*\*\*

Basic Car. Adding features of Luxury Car. Adding features of Sports Car.

**Decorator Design Pattern – Important Points**

* Decorator design pattern is helpful in providing runtime modification abilities and hence more flexible. Its easy to maintain and extend when the number of choices are more.
* The disadvantage of decorator design pattern is that it uses a lot of similar kind of objects (decorators).
* Decorator pattern is used a lot in [Java IO](https://www.journaldev.com/942/java-io-tutorial) classes, such as [FileReader, BufferedReader](https://www.journaldev.com/867/java-read-text-file) etc.