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**}**.

2. **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 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.
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.

3. 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.

4. 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, 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**(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 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.

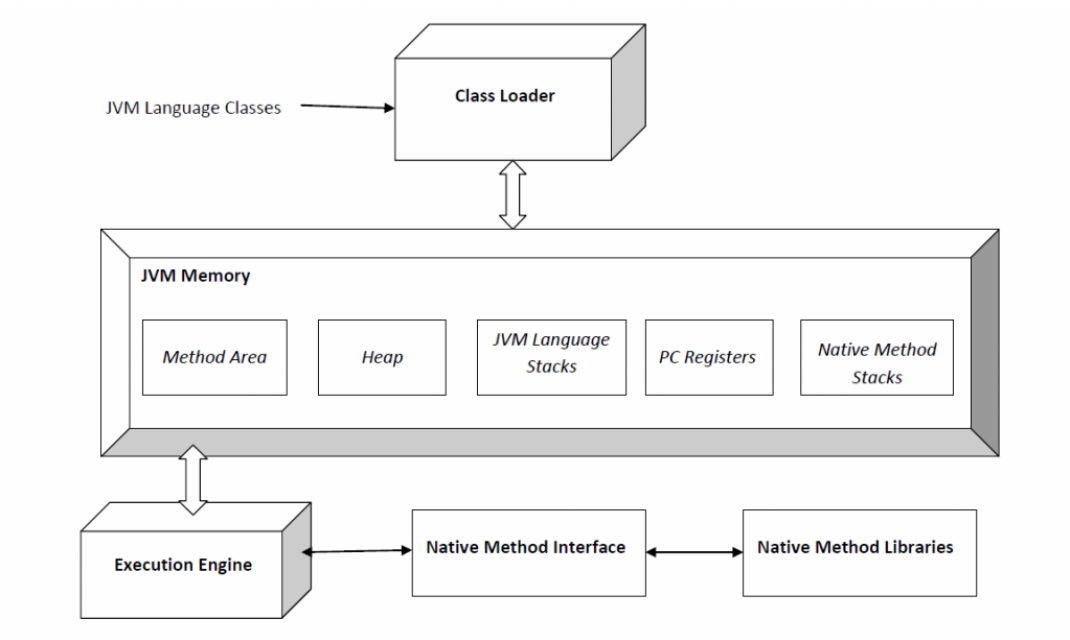


Image Source: https://en.wikipedia.org/wiki/Java_virtual_machine

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 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;
   }
}
```

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());
   }
}
```

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*.

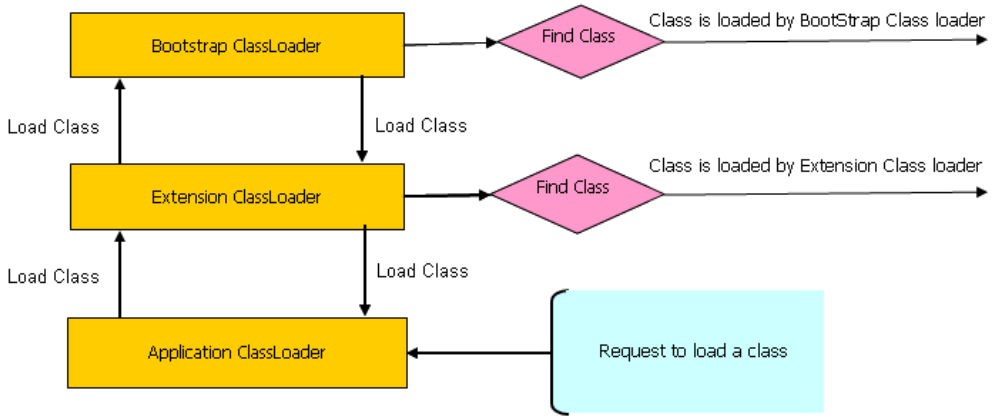


Image Source: http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html

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, its 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.

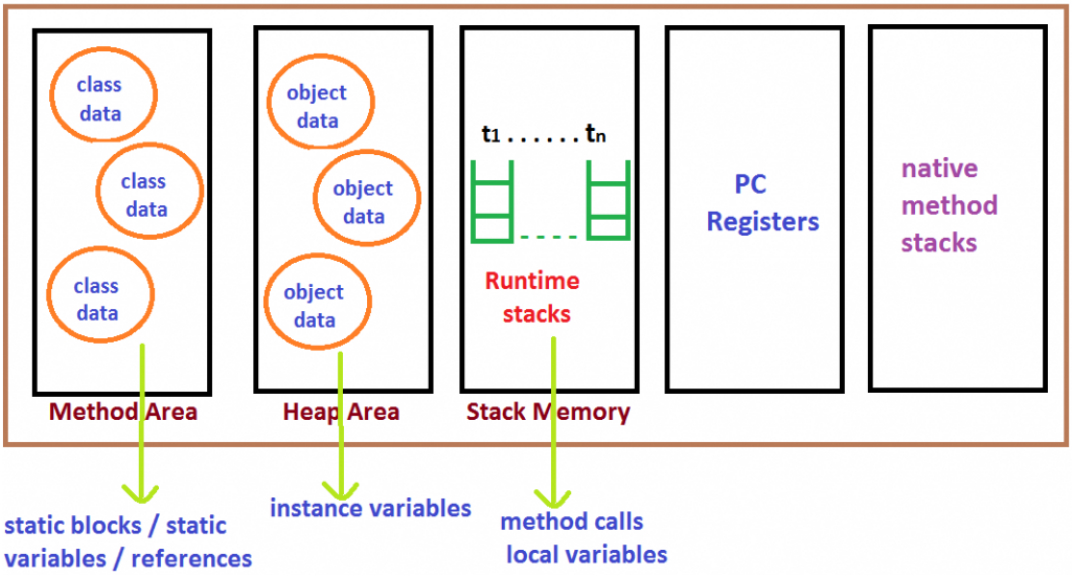


Image Source: http://java.scjp.jobs4times.com/fund/fund2.png

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

Java Native Interface (JNI):

It is an 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.

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*.

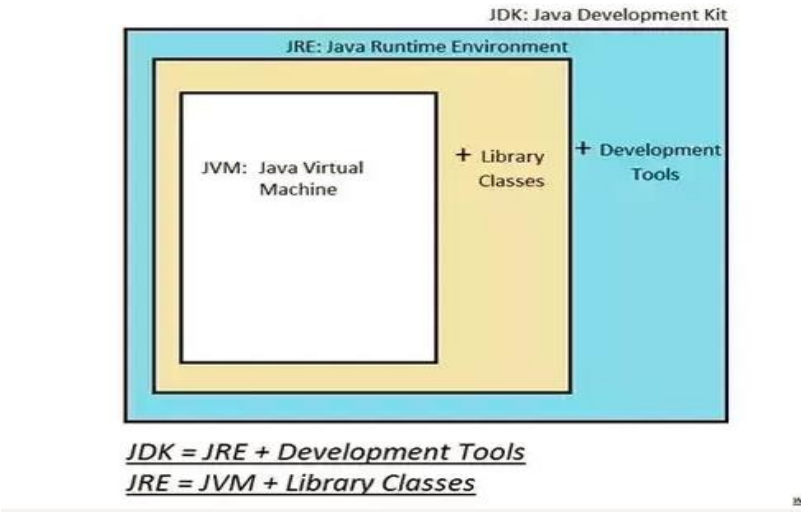
JAVA VIRTUAL MACHINE

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?

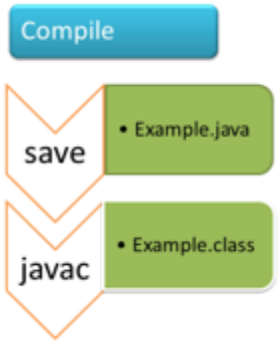
What does JRE consists of?

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.

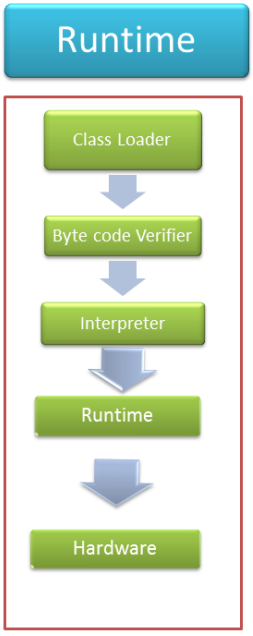
- **Class Loader**

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.

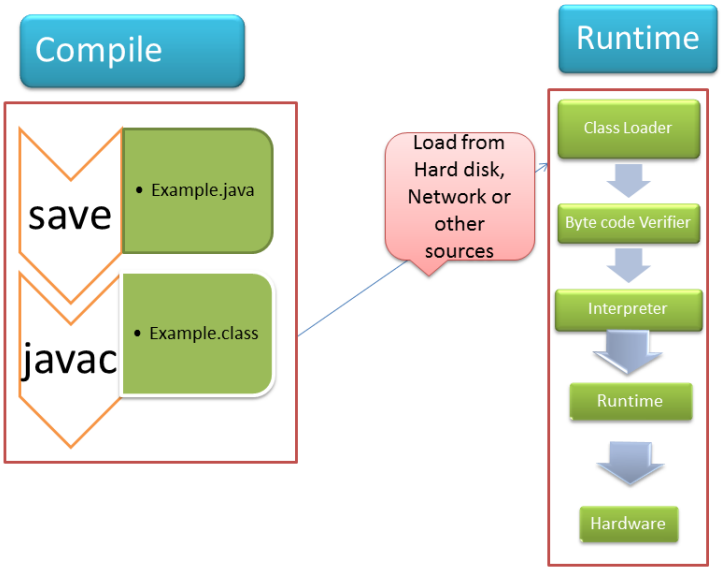


- **Intrepreter**

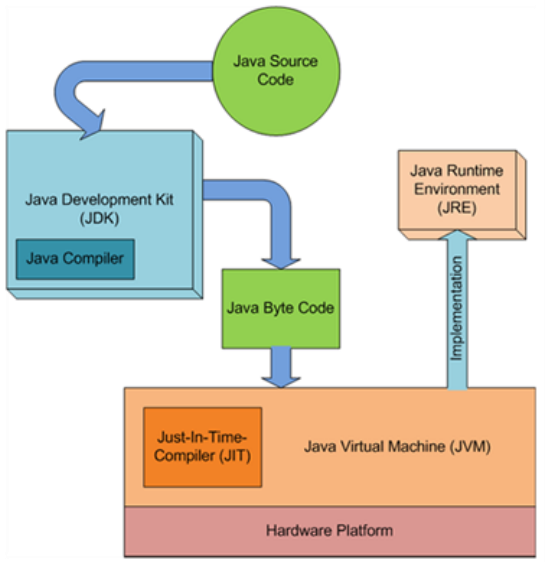
At runtime, the Byte Code is loaded, checked and run by the interpreter. The interpreter has the following two functions:

- 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, 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 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.

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 ma
in(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 comparison 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 object using new(), the object 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 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 initiated
   }
}
```

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 : 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 must use the newInstance 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: 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: 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)

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()
  }
}
```

Note: 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 allows Java to support overriding of methods which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allows subclasses to add its specific methods subclasses to define the specific implementation of some.

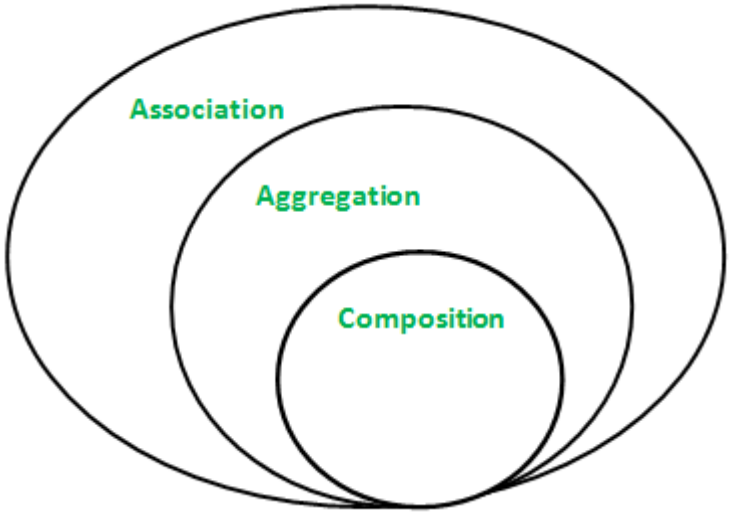
Static vs Dynamic binding

- 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.



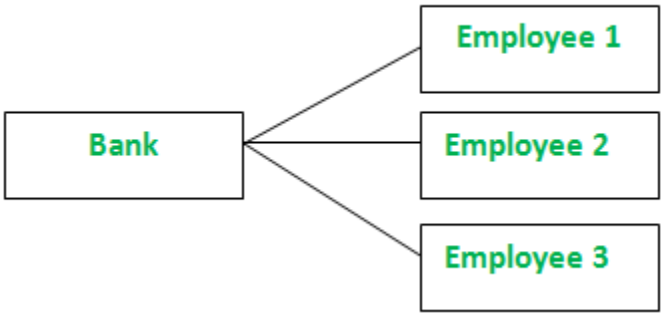

```
// Java program to illustrate the
// concept of Association
import java.io.*;

// 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.



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 associated 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;
```

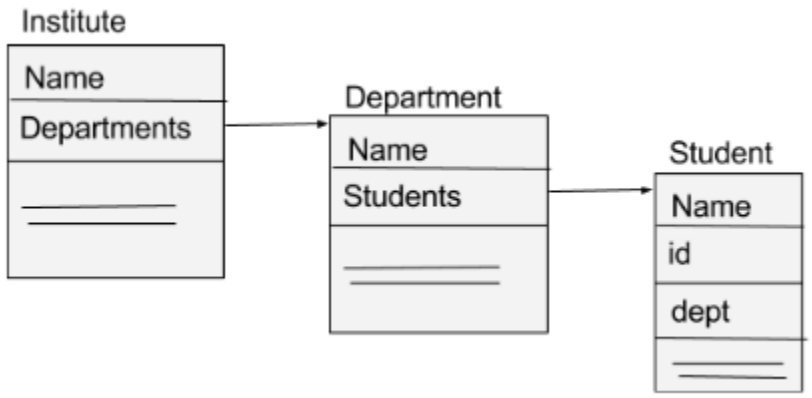
```
      for(Department dept : departments)
      {
         students = dept.getStudents();
         for(Student s : students)
         {
            noOfStudents++;
         }
      }
      return noOfStudents;
   }
 }

// 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 an 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.



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 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 independently of 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

- Encapsulation is data hiding (information hiding) while Abstraction is detail hiding (implementation hiding).
- 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

- It reduces the complexity of viewing the things.
- Avoids code duplication and increases reusability.
- 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 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 must 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 an 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.

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 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

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 converts 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 $4*16^0 + 6*16^1 = 100$

- **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
  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.
- **finalize() method** : This method is called just before an object is garbage collected. It is called by the Garbage Collector 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().
- The remaining three methods **wait()**, **notify() notifyAll()** are related to Concurrency. Refer Inter-thread Communication in 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). 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). The Object class 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. 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 -
    a. If the variable(var) is byte, char, short or int, then var_code = (int)var;
    b. If the variable(var) is long, then var_code = (int)(var ^ (var >>> 32));
    c. If the variable(var) is float, then var_code = Float.floatToIntBits(var);
    d. If the variable(var) is double, then -
    long bits = Double.doubleToLongBits(var);
    var_code = (int)(bits ^ (bits >>> 32));
    e. If the variable(var) is boolean, then var_code = var ? 1 : 0;
    f. 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

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

c. However, the equals method in java.lang.Object class only provides "shallow comparison", same as provided by the equality operator ==.
d. The equals method only takes Java objects as an argument, and not primitives; passing primitives will result in a compile time error.
e. Passing objects of different types to the equals method will never result in a compile time error or runtime error.
f. 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.
g. The class java.lang.StringBuffer does not override the equals method, and hence it inherits the implementation from java.lang.Object class.
h. 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.
i. If null is passed as an argument to the equals method, it will return false.
j. Equal hash codes do not imply that the objects are equal.
k. 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.
l. 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.
m. 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.
n. 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.

```
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();
   }
}
```

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();
   }
}
```

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");
  }
}
```

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");
   }
}
```

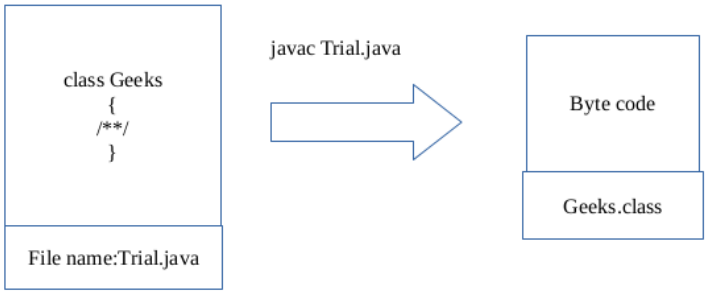
Run on IDE

```
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:



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 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));
    }
}
```

The output is

```
9GeeksforGeeks
GeeksforGeeks2016
8GeeksforGeeks2016
8GeeksforGeeks9
```

Explanation:

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);
  }
}
```

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 object 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));
    }
}
```

Output:

```
skeeGrofskeeG
```

Remove Trailing Zeros from String in Java

We use StringBuffer class as Strings are immutable.

- 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);
   }
}
```

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);
  }
}
```

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 like templates in C++. 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());
   }
}
```

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();
   }
}
```

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);
   }
}
```

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);
    }
}
```

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);
    }
}
```

Output:

```
15: error: no suitable method found for add(int)
        al.add(10);
         ^
```

3. 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);
   }
}
```

4. 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);
  }
}
```

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);
  }
}
```

Output:

```
[1, 2, 3]
[1.1, 2.2, 3.3]
```

Internal Working of HashMap in Java

In this article, we will see how hashmap's get and put method works internally. What operations are performed. How the hashing is done. How the value is fetched by key. How the key-value pair is stored.

HashMap contains an array of Node and Node can represent a class having following objects:

- int hash
- K key
- V value
- Node next

Now we will see how this works. First, we will see the hashing process.

Hashing

Hashing is a process of converting an object into integer form by using the method hashCode(). Its necessary to write hashCode() method properly for better performance of HashMap. Here I am taking key of my own class so that I can override hashCode() method to show different scenarios. My Key class is

```
//custom Key class to override hashCode()
// and equals() method
class Key
{
 String key;
 Key(String key)
 {
  this.key = key;
 }

 @Override
 public int hashCode()
 {
  return (int)key.charAt(0);
 }
```

```
 @Override
 public boolean equals(Object obj)
 {
  return key.equals((String)obj);
 }
}
```

Here overrided hashCode() method returns the first character's ASCII value as hash code. So, whenever the first character of key is same, the hash code will be same. You should not approach this criteria in your program. It is just for demo purpose. As HashMap also allows null key, so hash code of null will always be 0.

hashCode() method

hashCode() method is used to get the hash Code of an object. hashCode() method of object class returns the memory reference of object in integer form. Definition of hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because there is not any direct method in java to fetch the reference of object. It is possible to provide your own implementation of hashCode().

In HashMap, hashCode() is used to calculate the bucket and therefore calculate the index.

equals() method

equals method is used to check that 2 objects are equal or not. This method is provided by Object class. You can override this in your class to provide your own implementation.

HashMap uses equals() to compare the key whether they are equal or not. If equals() method return true, they are equal otherwise not equal.

Buckets

A bucket is one element of HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes. Buckets are different in capacity. A relation between bucket and capacity is as follows:

```
capacity = number of buckets * load factor
```

A single bucket can have more than one nodes, it depends on hashCode() method. The better your hashCode() method is, the better your buckets will be utilized.

Index Calculation in Hashmap

Hash code of key may be large enough to create an array. hash code generated may be in the range of integer and if we create arrays for such a range, then it will easily cause outOfMemoryException. So, we generate index to minimize the size of array. Basically, following operation is performed to calculate index.

```
index = hashCode(key) & (n-1).
```

where n is number of buckets or the size of array. In our example, I will consider n as default size that is 16.

- **Initially Empty hashMap:** Here, the hashmap is size is taken as 16.

```
HashMap map = new HashMap();
```

HashMap :

- **Inserting Key-Value Pair:** Putting one key-value pair in above HashMap

```
map.put(new Key("vishal"), 20);
```

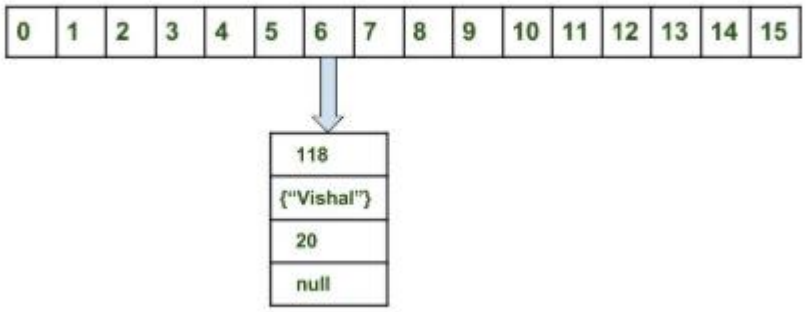
Steps:

1. Calculate hash code of Key {“vishal”}. It will be generated as 118.
2. Calculate index by using index method it will be 6.
3. Create a node object as:

```
{
 int hash = 118

 // {"vishal"} is not a string but
 // an object of class Key
 Key key = {"vishal"}

 Integer value = 20
 Node next = null
}
```

4. Place this object at index 6, if no other object is presented there.

Now HashMap becomes:



- **Inserting another Key-Value Pair:** Now, putting other pair that is,

```
map.put(new Key("sachin"), 30);
```

Steps:

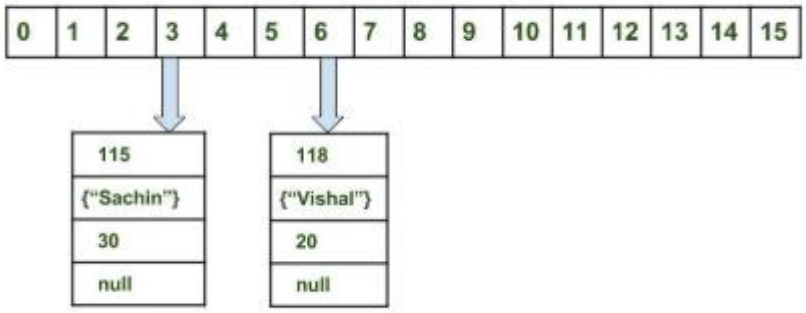
1. Calculate hashCode of Key {“sachin”}. It will be generated as 115.
2. Calculate index by using index method it will be 3.
3. Create a node object as :

```
{
```

```
 int hash = 115
 Key key = {"sachin"}
 Integer value = 30
 Node next = null
}
```

Place this object at index 3 if no other object is presented there.

Now HashMap becomes:



- **In Case of collision:** Now, putting another pair that is,

```
map.put(new Key("vaibhav"), 40);
```

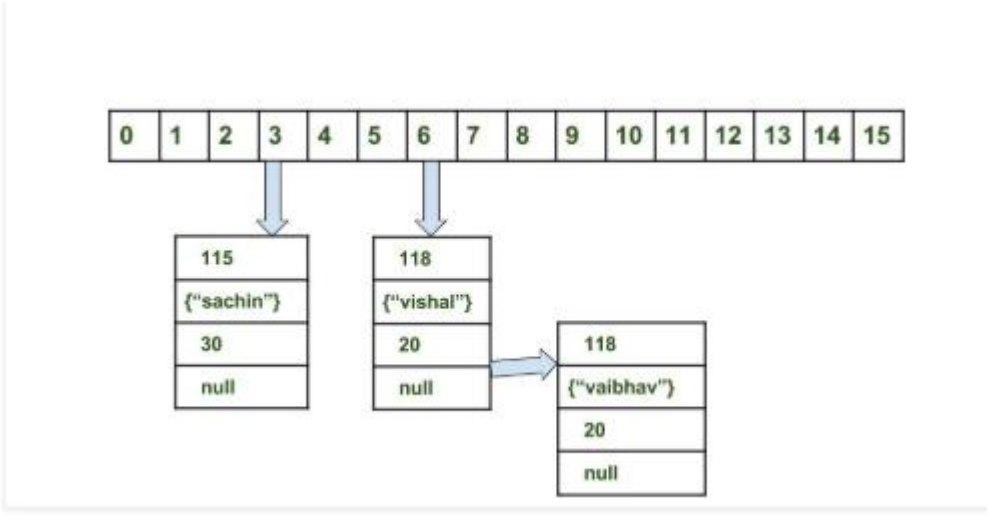
Steps:

1. Calculate hash code of Key {“vaibhav”}. It will be generated as 118.
2. Calculate index by using index method it will be 6.
3. Create a node object as:

```
{
 int hash = 118
 Key key = {"vaibhav"}
 Integer value = 40
 Node next = null
}
```

4. Place this object at index 6 if no other object is presented there.
5. In this case a node object is found at the index 6 – this is a case of collision.
6. In that case, check via hashCode() and equals() method that if both the keys are same.
7. If keys are same, replace the value with current value.
8. Otherwise connect this node object to the previous node object via linked list and both are stored at index 6.

Now HashMap becomes:



Using get method()

Now let's try some get method to get a value. get(K key) method is used to get a value by its key. If you don't know the key then it is not possible to fetch a value.

- **Fetch the data for key sachin:**

```
map.get(new Key("sachin"));
```

Steps:

1. Calculate hash code of Key {"sachin"}. It will be generated as 115.
2. Calculate index by using index method it will be 3.
3. Go to index 3 of array and compare first element's key with given key. If both are equals then return the value, otherwise check for next element if it exists.
4. In our case it is found as first element and returned value is 30.

- **Fetch the data for key vaibahv:**

```
map.get(new Key("vaibhav"));
```

Steps:

1. Calculate hash code of Key {"vaibhav"}. It will be generated as 118.
2. Calculate index by using index method it will be 6.
3. Go to index 6 of array and compare first element's key with given key. If both are equals then return the value, otherwise check for next element if it exists.
4. In our case it is not found as first element and next of node object is not null.
5. If next of node is null then return null.
6. If next of node is not null traverse to the second element and repeat the process 3 until key is not found or next is not null.

```
// Java program to illustrate
// internal working of HashMap
import java.util.HashMap;

class Key {
   String key;
   Key(String key)
   {
      this.key = key;
```

```
    }

    @Override
    public int hashCode()
    {
        int hash = (int)key.charAt(0);
        System.out.println("hashCode for key: "
                           + key + " = " + hash);
        return hash;
    }

    @Override
    public boolean equals(Object obj)
    {
        return key.equals(((Key)obj).key);
    }
}

// Driver class
public class GFG {
    public static void main(String[] args)
    {
        HashMap map = new HashMap();
        map.put(new Key("vishal"), 20);
        map.put(new Key("sachin"), 30);
        map.put(new Key("vaibhav"), 40);

        System.out.println();
        System.out.println("Value for key sachin: " + map.get(new Key("sachin")));
        System.out.println("Value for key vaibhav: " + map.get(new Key("vaibhav")));
    }
}
```

Output:

```
hashCode for key: vishal = 118
hashCode for key: sachin = 115
hashCode for key: vaibhav = 118

hashCode for key: sachin = 115
Value for key sachin: 30
hashCode for key: vaibhav = 118
Value for key vaibhav: 40
```

HashMap Changes in Java 8

As we know now that in case of hash collision entry objects are stored as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a worst-case scenario the complexity becomes O(n). To address this issue, Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a hash becomes larger than a certain threshold, the hash will change from

using a **linked list to a balanced tree**, which will improve the worst case performance from O(n) to O(log n).

Important Points

- Time complexity is almost constant for put and get method until rehashing is not done.
- In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
- If key given already exist in HashMap, the value is replaced with new value.
- hash code of null key is 0.
- When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

When to use what (abstract class and interface)?

Consider using abstract classes if any of these statements apply to your situation:

- In java application, there are some related classes that need to share some lines of code then you can put these lines of code within abstract class and this abstract class should be extended by all these related classes.
- You can define non-static or non-final field(s) in abstract class, so that via a method you can access and modify the state of Object to which they belong.
- You can expect that the classes that extend an abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

Consider using interfaces if any of these statements apply to your situation:

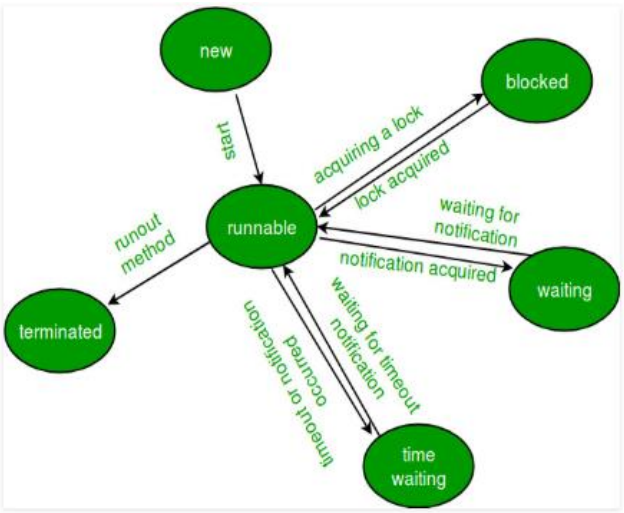
- It is total abstraction, all methods declared within an interface must be implemented by the class(es) that implements this interface.
- A class can implement more than one interface. It is called multiple inheritance.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.

Lifecycle and States of a Thread in Java

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated

The diagram shown below represent various states of a thread at any instant of time.



Life Cycle of a thread

New Thread: When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it's code is yet to be run and hasn't started to execute.

Runnable State: A thread that is ready to run is moved to runnable state. In this state, a thread might be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.

A multi-threaded program allocates a fixed amount of time to each individual thread. Each thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

Blocked/Waiting state: When a thread is temporarily inactive, then it's in one of the following states:

- Blocked
- Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

Timed Waiting: A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to timed waiting state.

Terminated State: A thread terminates because of either of the following reasons:

- Because it exists normally. This happens when the code of thread has entirely executed by the program.
- Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

A thread that lies in terminated state does no longer consumes any cycles of CPU.

https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples

Java Singleton Design Pattern

Java Singleton Pattern is one of the Gangs of Four Design patterns and comes in the Creational Design Pattern category. From the definition, it seems to be a very simple design pattern but when it comes to implementation, it comes with a lot of implementation concerns. The implementation of Java Singleton pattern has always been a controversial topic among developers. Here we will learn about Singleton design pattern principles, different ways to implement Singleton design pattern and some of the best practices for it's usage.

Java Singleton

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.
- The singleton class must provide a global access point to get the instance of the class.
- Singleton pattern is used for logging, drivers objects, caching and thread pool.
- Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc.
- Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

Java Singleton Pattern

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

In further sections, we will learn different approaches of Singleton pattern implementation and design concerns with the implementation.

1. Eager initialization
2. Static block initialization
3. Lazy Initialization
4. Thread Safe Singleton
5. Bill Pugh Singleton Implementation
6. Using Reflection to destroy Singleton Pattern
7. Enum Singleton
8. Serialization and Singleton

Eager initialization

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

Here is the implementation of static initialization singleton class.

```
public class EagerInitializedSingleton {

   private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();

   //private constructor to avoid client applications to use constructor
   private EagerInitializedSingleton(){}

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

If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc and we should avoid the instantiation until unless client calls the getInstance method. Also, this method doesn't provide any options for exception handling.

Static block initialization

Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.

```
public class StaticBlockSingleton {

   private static StaticBlockSingleton instance;

   private StaticBlockSingleton(){}

   //static block initialization for exception handling
   static{
      try{
         instance = new StaticBlockSingleton();
      }catch(Exception e){
         throw new RuntimeException("Exception occured in creating singleton instance");
      }
   }
```

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

Both eager initialization and static block initialization creates the instance even before it's being used and that is not the best practice to use. So, in further sections, we will learn how to create Singleton class that supports lazy initialization.

Lazy Initialization

Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this approach.

```
public class LazyInitializedSingleton {

   private static LazyInitializedSingleton instance;

   private LazyInitializedSingleton(){}

   public static LazyInitializedSingleton getInstance(){
      if(instance == null){
         instance = new LazyInitializedSingleton();
      }
      return instance;
   }
}
```

The above implementation works fine incase of single threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class. In next section, we will see different ways to create a thread-safe singleton class.

Thread Safe Singleton

The easier way to create a thread-safe singleton class is to make the global access method synchronized, so that only one thread can execute this method at a time. General implementation of this approach is like the below class.

```
public class ThreadSafeSingleton {

   private static ThreadSafeSingleton instance;

   private ThreadSafeSingleton(){}

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

}
```

Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances (Read: Java Synchronization). To avoid this extra overhead every time, double checked locking principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of singleton class is created.

Below code snippet provides the double-checked locking implementation.

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
   if(instance == null){
      synchronized (ThreadSafeSingleton.class) {
         if(instance == null){
            instance = new ThreadSafeSingleton();
         }
      }
   }
   return instance;
}
```

Bill Pugh Singleton Implementation

Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using a inner static helper class. The Bill Pugh Singleton implementation goes like this;

```
public class BillPughSingleton {

   private BillPughSingleton(){}

   private static class SingletonHelper{
      private static final BillPughSingleton INSTANCE = new BillPughSingleton();
   }

   public static BillPughSingleton getInstance(){
      return SingletonHelper.INSTANCE;
   }
}
```

Notice the private inner static class that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance method, this class gets loaded and creates the Singleton class instance.

This is the most widely used approach for Singleton class as it doesn't require synchronization. I am using this approach in many of my projects and it's easy to understand and implement also.

Using Reflection to destroy Singleton Pattern

Reflection can be used to destroy all the above singleton implementation approaches. Let's see this with an example class.

```
import java.lang.reflect.Constructor;

public class ReflectionSingletonTest {
```

```
  public static void main(String[] args) {
    EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();
    EagerInitializedSingleton instanceTwo = null;
    try {
      Constructor[] constructors = EagerInitializedSingleton.class.getDeclaredConstructors();
      for (Constructor constructor : constructors) {
        //Below code will destroy the singleton pattern
        constructor.setAccessible(true);
        instanceTwo = (EagerInitializedSingleton) constructor.newInstance();
        break;
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(instanceOne.hashCode());
    System.out.println(instanceTwo.hashCode());
  }

}
```

When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate, do check out Java Reflection Tutorial.

Enum Singleton

To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that **any enum value is instantiated only once in a Java program**. Since Java Enum values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

```
public enum EnumSingleton {

  INSTANCE;

  public static void doSomething(){
    //do something
  }
}
```

Serialization and Singleton

Sometimes in distributed systems, we need to implement Serializable interface in Singleton class so that we can store its state in file system and retrieve it at later point of time. Here is a small singleton class that implements Serializable interface also.

```
import java.io.Serializable;

public class SerializedSingleton implements Serializable{
private static final long serialVersionUID = -7604766932017737115L;

  private SerializedSingleton(){}
  private static class SingletonHelper{
    private static final SerializedSingleton instance = new SerializedSingleton();
  }
```

```
   public static SerializedSingleton getInstance(){
      return SingletonHelper.instance;
   }
}
```

The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Let's see it with a simple program.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

public class SingletonSerializedTest {

   public static void main(String[] args) throws FileNotFoundException, IOException,
ClassNotFoundException {
      SerializedSingleton instanceOne = SerializedSingleton.getInstance();
      ObjectOutput out = new ObjectOutputStream(new FileOutputStream(
            "filename.ser"));
      out.writeObject(instanceOne);
      out.close();

      //deserailize from file to object
      ObjectInput in = new ObjectInputStream(new FileInputStream(
            "filename.ser"));
      SerializedSingleton instanceTwo = (SerializedSingleton) in.readObject();
      in.close();

      System.out.println("instanceOne hashCode="+instanceOne.hashCode());
      System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());
   }
}
```

Output of the above program is;

```
instanceOne hashCode=2011117821
instanceTwo hashCode=109647522
```

So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.

```
protected Object readResolve() {
   return getInstance();
}
```

After this you will notice that hashCode of both the instances are same in test program.

JAVA 8 Features:

https://javapapers.com/java/java-8-features/

http://www.mkyong.com/tutorials/java-8-tutorials/

Stream in Java

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

The features of Java stream are –

A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.

Streams don't change the original data structure, they only provide the result as per the pipelined methods.

Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way like SQL statements. For example, consider the following SQL statement.

```
SELECT max(salary), employee_id, employee_name FROM Employee
```

The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end. Using collections framework in Java, a developer must use loops and make repeated checks. Another concern is efficiency; as multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone.

To resolve such issues, Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

- **Sequence of elements –** A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
- **Source –** Stream takes Collections, Arrays, or I/O resources as input source.
- **Aggregate operations –** Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
- **Pipelining –** Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
- **Automatic iterations –** Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Generating Streams

With Java 8, Collection interface has two methods to generate a Stream.

- stream() − Returns a sequential stream considering collection as its source.
- parallelStream() − Returns a parallel Stream considering collection as its source.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
```

forEach

Stream has provided a new method 'forEach' to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

map

The 'map' method is used to map each element to its corresponding result. The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

filter

The 'filter' method is used to eliminate elements based on a criterion. The following code segment prints a count of empty strings using filter.

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
//get count of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```

limit

The 'limit' method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

sorted

The 'sorted' method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

```
Random random = new Random();
random.ints().limit(10).sorted().forEach(System.out::println);
```

Parallel Processing

ParallelStream is the alternative of stream for parallel processing. Look at the following code segment that prints a count of empty strings using parallelStream.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
int count = strings.parallelStream().filter(string -> string.isEmpty()).count();
```

It is very easy to switch between sequential and parallel streams.

Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);
String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(",
"));
System.out.println("Merged String: " + mergedString);
```

Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

```
List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());
System.out.println("Lowest number in List : " + stats.getMin());
System.out.println("Sum of all numbers : " + stats.getSum());
System.out.println("Average of all numbers : " + stats.getAverage());
```

Stream Example

```
Java8Tester.java
 Live Demo
import java.util.ArrayList;
import java.util.Arrays;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.Random;
```

```
import java.util.stream.Collectors;
import java.util.Map;

public class Java8Tester {

  public static void main(String args[]) {
    System.out.println("Using Java 7: ");

    // Count empty strings
    List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
    System.out.println("List: " +strings);
    long count = getCountEmptyStringUsingJava7(strings);

    System.out.println("Empty Strings: " + count);
    count = getCountLength3UsingJava7(strings);

    System.out.println("Strings of length 3: " + count);

    //Eliminate empty string
    List<String> filtered = deleteEmptyStringsUsingJava7(strings);
    System.out.println("Filtered List: " + filtered);

    //Eliminate empty string and join using comma.
    String mergedString = getMergedStringUsingJava7(strings,", ");
    System.out.println("Merged String: " + mergedString);
    List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

    //get list of square of distinct numbers
    List<Integer> squaresList = getSquares(numbers);
    System.out.println("Squares List: " + squaresList);
    List<Integer> integers = Arrays.asList(1,2,13,4,15,6,17,8,19);

    System.out.println("List: " +integers);
    System.out.println("Highest number in List : " + getMax(integers));
    System.out.println("Lowest number in List : " + getMin(integers));
    System.out.println("Sum of all numbers : " + getSum(integers));
    System.out.println("Average of all numbers : " + getAverage(integers));
    System.out.println("Random Numbers: ");

    //print ten random numbers
    Random random = new Random();

    for(int i = 0; i < 10; i++) {
      System.out.println(random.nextInt());
    }

    System.out.println("Using Java 8: ");
    System.out.println("List: " +strings);

    count = strings.stream().filter(string->string.isEmpty()).count();
    System.out.println("Empty Strings: " + count);
```

```
    count = strings.stream().filter(string -> string.length() == 3).count();
    System.out.println("Strings of length 3: " + count);

    filtered = strings.stream().filter(string ->!string.isEmpty()).collect(Collectors.toList());
    System.out.println("Filtered List: " + filtered);

    mergedString = strings.stream().filter(string ->!string.isEmpty()).collect(Collectors.joining(", "));
    System.out.println("Merged String: " + mergedString);

    squaresList = numbers.stream().map( i ->i*i).distinct().collect(Collectors.toList());
    System.out.println("Squares List: " + squaresList);
    System.out.println("List: " +integers);

    IntSummaryStatistics stats = integers.stream().mapToInt((x) ->x).summaryStatistics();

    System.out.println("Highest number in List : " + stats.getMax());
    System.out.println("Lowest number in List : " + stats.getMin());
    System.out.println("Sum of all numbers : " + stats.getSum());
    System.out.println("Average of all numbers : " + stats.getAverage());
    System.out.println("Random Numbers: ");

    random.ints().limit(10).sorted().forEach(System.out::println);

    //parallel processing
    count = strings.parallelStream().filter(string -> string.isEmpty()).count();
    System.out.println("Empty Strings: " + count);
  }

  private static int getCountEmptyStringUsingJava7(List<String> strings) {
    int count = 0;

    for(String string: strings) {

      if(string.isEmpty()) {
        count++;
      }
    }
    return count;
  }

  private static int getCountLength3UsingJava7(List<String> strings) {
    int count = 0;

    for(String string: strings) {

      if(string.length() == 3) {
        count++;
      }
    }
    return count;
```

```
}

private static List<String> deleteEmptyStringsUsingJava7(List<String> strings) {
  List<String> filteredList = new ArrayList<String>();

  for(String string: strings) {

    if(!string.isEmpty()) {
       filteredList.add(string);
    }
  }
  return filteredList;
}

private static String getMergedStringUsingJava7(List<String> strings, String separator) {
  StringBuilder stringBuilder = new StringBuilder();

  for(String string: strings) {

    if(!string.isEmpty()) {
      stringBuilder.append(string);
      stringBuilder.append(separator);
    }
  }
  String mergedString = stringBuilder.toString();
  return mergedString.substring(0, mergedString.length()-2);
}

private static List<Integer> getSquares(List<Integer> numbers) {
  List<Integer> squaresList = new ArrayList<Integer>();

  for(Integer number: numbers) {
    Integer square = new Integer(number.intValue() * number.intValue());

    if(!squaresList.contains(square)) {
      squaresList.add(square);
    }
  }
  return squaresList;
}

private static int getMax(List<Integer> numbers) {
  int max = numbers.get(0);

  for(int i = 1;i < numbers.size();i++) {

    Integer number = numbers.get(i);

    if(number.intValue() > max) {
      max = number.intValue();
    }
```

```
    }
    return max;
  }

  private static int getMin(List<Integer> numbers) {
    int min = numbers.get(0);

    for(int i= 1;i < numbers.size();i++) {
      Integer number = numbers.get(i);

      if(number.intValue() < min) {
        min = number.intValue();
      }
    }
    return min;
  }

  private static int getSum(List numbers) {
    int sum = (int)(numbers.get(0));

    for(int i = 1;i < numbers.size();i++) {
      sum += (int)numbers.get(i);
    }
    return sum;
  }

  private static int getAverage(List<Integer> numbers) {
    return getSum(numbers) / numbers.size();
  }
}
```

Verify the Result
Compile the class using javac compiler as follows −

```
C:\JAVA>javac Java8Tester.java
```

Now run the Java8Tester as follows −

```
C:\JAVA>java Java8Tester
```

It should produce the following result −

Using Java 7:

```
List: [abc, , bc, efg, abcd, , jkl]
Empty Strings: 2
Strings of length 3: 3
Filtered List: [abc, bc, efg, abcd, jkl]
Merged String: abc, bc, efg, abcd, jkl
Squares List: [9, 4, 49, 25]
List: [1, 2, 13, 4, 15, 6, 17, 8, 19]
Highest number in List : 19
Lowest number in List : 1
Sum of all numbers : 85
Average of all numbers : 9
Random Numbers:
-1279735475
```

```
903418352
-1133928044
-1571118911
628530462
18407523
-881538250
-718932165
270259229
421676854
Using Java 8:
List: [abc, , bc, efg, abcd, , jkl]
Empty Strings: 2
Strings of length 3: 3
Filtered List: [abc, bc, efg, abcd, jkl]
Merged String: abc, bc, efg, abcd, jkl
Squares List: [9, 4, 49, 25]
List: [1, 2, 13, 4, 15, 6, 17, 8, 19]
Highest number in List : 19
Lowest number in List : 1
Sum of all numbers : 85
Average of all numbers : 9.444444444444445
Random Numbers:
-1009474951
-551240647
-2484714
181614550
933444268
1227850416
1579250773
1627454872
1683033687
1798939493
Empty Strings: 2
```

https://dzone.com/articles/why-string-immutable-java

Why String is Immutable in Java

This is an old yet still popular question. There are multiple reasons that String is designed to be immutable in Java. A good answer depends on good understanding of memory, synchronization, data structures, etc.

Requirement of String Pool

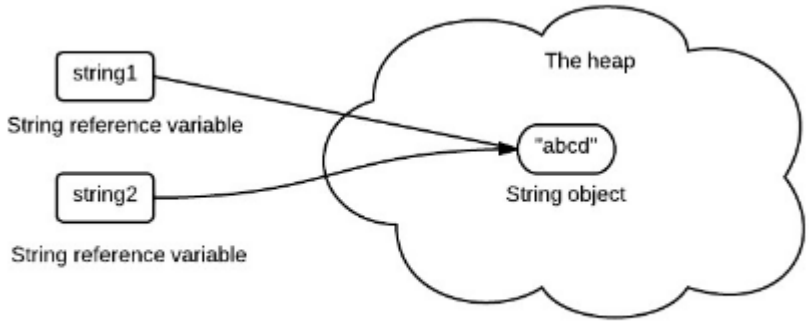
String pool (String intern pool) is a special storage area in Java heap. When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object and returning its reference.

The following code will create only one string object in the heap.

```
String string1 = "abcd";
```

```
String string2 = "abcd";
```

Here is how it looks:



If string is not immutable, changing the string with one reference will lead to the wrong value for the other references.

Allow String to Cache its Hashcode

The hashcode of string is frequently used in Java. For example, in a HashMap. Being immutable guarantees that hashcode will always the same, so that it can be cashed without worrying the changes. That means, there is no need to calculate hashcode every time it is used. This is more efficient.

Security

String is widely used as parameter for many java classes, e.g. network connection, opening files, etc. Were String not immutable, a connection or file would be changed and lead to serious security threat. The method thought it was connecting to one machine, but was not. Mutable strings could cause security problem in Reflection too, as the parameters are strings.

Here is a code example:

```
boolean connect(string s){
   if (!isSecure(s)) {
throw new SecurityException();
}
   //here will cause problem, if s is changed before this by using other references.
   causeProblem(s);
}
```

In summary, the reasons include design, efficiency, and security. This is also true for many other “why” questions in a Java interview.

https://javaconceptoftheday.com/example-to-prove-strings-are-immutable/

An Example To Prove Strings Are Immutable

One more interesting thing about String objects in java is that they are immutable. That means once you create a string object, you can’t modify the contents of that object. If you try to modify the contents of string object, a new string object is created with modified content.

In this article, we will discuss the examples which prove that strings are immutable.

An Example To Prove Strings Are Immutable :

First, create one string object ‘s1’ using string literal “JAVA”.

```
String s1 = "JAVA";
```

Create one more string object ‘s2’ using the same string literal “JAVA”.

```
String s2 = "JAVA";
```

We have seen in the previous article that string objects created using string literal are stored in the String Constant Pool and any two objects in the pool can’t have same content. Here s1 and s2 are created using same literal. Therefore, they will be pointing to same object in the pool. Then s1 == s2 should return true.

```
System.out.println(s1 == s2);     //Output : true
```

Now, I want to make little modification to this object through ‘s1’ reference. I want to append “J2EE” at end of this string through ‘s1’. That can be done like below,

```
s1 =s1 + "J2EE";
```

This statement appends “J2EE” to the object to which s1 is pointing and re-assigns reference of that object back to s1.

Now, compare physical address of s1 and s2 using “==” operator. This time it will return false.

```
System.out.println(s1 == s2);     //Output : false
```

That means now both s1 and s2 are pointing to two different objects in the pool. Before modifications they are pointing to same object. Once we tried to change the content of the object using ‘s1’, a new object is created in the pool with “JAVAJ2EE” as it’s content and its reference is assigned to s1. If the strings are mutable, both s1 and s2 should point to same object even after modification. That never happened here. That proves the string objects are immutable in java.

The whole program can be written like this,

```
public class StringExamples
{
   public static void main(String[] args)
   {
      String s1 = "JAVA";
       String s2 = "JAVA";
       System.out.println(s1 == s2);        //Output : true
       s1 = s1 + "J2EE";
       System.out.println(s1 == s2);        //Output : false
   }
}
```

is new String() also immutable?

After seeing the above example, one more question may have left in your mind. Are string objects created using new operator also immutable? The answer is Yes. String objects created using new operator are also immutable although they are stored in the heap memory. This can be also proved with help of an example.

```
public class StringExamples
{
   public static void main(String[] args)
   {
      String s1 = new String("JAVA");
```

```
        System.out.println(s1);        //Output : JAVA
        s1.concat("J2EE");
        System.out.println(s1);        //Output : JAVA
    }
}
```

In this example, a string object is created with “JAVA” as it’s content using new operator and it’s reference is assigned to s1. I have tried to change the contents of this object using concat() method. But, these changes are not reflected in the object as seen in Line 11. Even after the concatenation, content of the object is same as before. This is because the strings are immutable. Once I tried to concatenate “J2EE” to an existing string “JAVA”, a new string object is created with “JAVAJ2EE” as it’s content. But we don’t have reference to that object in this program.

Conclusion:

Immutability is the fundamental property of string objects. In whatever way, you create the string objects, either using string literals or using new operator, they are immutable.

https://www.journaldev.com/129/how-to-create-immutable-class-in-java

Immutable Class in Java

Immutable class is good for caching purpose because you don’t need to worry about the value changes. Other benefit of immutable class is that it is **inherently thread-safe**, so you don’t need to worry about thread safety in case of multi-threaded environment.

To create immutable class in java, you must do following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables
4. Make all mutable fields final so that it’s value can be assigned only once.
5. Initialize all the fields via a constructor performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

To understand points 4 and 5, let’s run the sample Final class that works well and values doesn’t get altered after instantiation.

```
package com.journaldev.java;
import java.util.HashMap;
import java.util.Iterator;

public final class FinalClassExample {

        private final int id;
        private final String name;
        private final HashMap<String,String> testMap;

        public int getId() {
                return id;
        }
        public String getName() {
```

```
		return name;
	}
	/**
	 * Accessor function for mutable objects
	 */
	public HashMap<String, String> getTestMap() {
		//return testMap;
		return (HashMap<String, String>) testMap.clone();
	}
	/**
	 * Constructor performing Deep
	 * @param i
	 * @param n
	 * @param hm
	 */

	public FinalClassExample(int i, String n, HashMap<String,String> hm){
		System.out.println("Performing Deep  for Object initialization");
		this.id=i;
		this.name=n;
		HashMap<String,String> tempMap=new HashMap<String,String>();
		String key;
		Iterator<String> it = hm.keySet().iterator();
		while(it.hasNext()){
			key=it.next();
			tempMap.put(key, hm.get(key));
		}
		this.testMap=tempMap;
	}
	/**
	 * Constructor performing Shallow
	 * @param i
	 * @param n
	 * @param hm
	 */
	/**
	public FinalClassExample(int i, String n, HashMap<String,String> hm){
		System.out.println("Performing Shallow  for Object initialization");
		this.id=i;
		this.name=n;
		this.testMap=hm;
	}
	*/

	/**
	 * To test the consequences of Shallow  and how to avoid it with Deep  for creating
immutable classes
	 * @param args
	 */
	public static void main(String[] args) {
		HashMap<String, String> h1 = new HashMap<String,String>();
```

```
                        h1.put("1", "first");
                        h1.put("2", "second");
                        String s = "original";
                        int i=10;
                        FinalClassExample ce = new FinalClassExample(i,s,h1);

                        //Lets see whether its copy by field or reference
                        System.out.println(s==ce.getName());
                        System.out.println(h1 == ce.getTestMap());
                        //print the ce values
                        System.out.println("ce id:"+ce.getId());
                        System.out.println("ce name:"+ce.getName());
                        System.out.println("ce testMap:"+ce.getTestMap());
                        //change the local variable values
                        i=20;
                        s="modified";
                        h1.put("3", "third");
                        //print the values again
                        System.out.println("ce id after local variable change:"+ce.getId());
                        System.out.println("ce name after local variable change:"+ce.getName());
                        System.out.println("ce testMap after local variable change:"+ce.getTestMap());

                        HashMap<String, String> hmTest = ce.getTestMap();
                        hmTest.put("4", "new");
                        System.out.println("ce testMap after changing variable from accessor
methods:"+ce.getTestMap());
                }

}
```

Output of the above immutable class in java example program is:

```
Performing Deep  for Object initialization
true
false
ce id:10
ce name:original
ce testMap:{2=second, 1=first}
ce id after local variable change:10
ce name after local variable change:original
ce testMap after local variable change:{2=second, 1=first}
ce testMap after changing variable from accessor methods:{2=second, 1=first}
```

Now let's comment the constructor providing deep copy and uncomment the constructor providing shallow copy. Also uncomment the return statement in getTestMap() method that returns the actual object reference and then execute the program once again.

```
Performing Shallow  for Object initialization
true
true
ce id:10
ce name:original
ce testMap:{2=second, 1=first}
```

```
ce id after local variable change:10
ce name after local variable change:original
ce testMap after local variable change:{3=third, 2=second, 1=first}
ce testMap after changing variable from accessor methods:{3=third, 2=second, 1=first, 4=new}
```

As you can see from the output, HashMap values got changed because of shallow copy in the constructor and providing direct reference to the original object in the getter function.

http://java-performance.info/memory-consumption-of-java-data-types-2/

https://dzone.com/articles/an-interview-question-on-spring-singletons

Spring Singletons

The Java singleton is scoped by the Java class loader, the Spring singleton is scoped by the container context.

Which basically means that, in Java, you can be sure a singleton is a truly a singleton only within the context of the class loader which loaded it. Other class loaders should be capable of creating another instance of it (provided the class loaders are not in the same class loader hierarchy), despite of all your efforts in code to try to prevent it.

In Spring, if you could load your singleton class in two different contexts and then again, we can break the singleton concept.

So, in summary, Java considers something a singleton if it cannot create more than one instance of that class within a given class loader, whereas Spring would consider something a singleton if it cannot create more than one instance of a class within a given container/context.

Spring Singleton is “per container per bean”.

While interviewing for positions using Spring Core, I often ask a certain question, "What do you mean by Spring Singleton scope?"

Most of the time, I get an answer like, "Spring Singleton scope manages only one object in the container."

After getting this answer, I ask the next question, "Please tell me what the output of the following program would be."

Spring.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
   <bean id="scopeTest" class="com.example.scope.Scope" scope="singleton">
      <property name="name" value="Shamik Mitra"/>
   </bean>
   <bean id="scopeTestDuplicate" class="com.example.scope.Scope" scope="singleton">
```

```
    <property name="name" value="Samir Mitra"/>
  </bean>
</beans>
```

Scope.java:

```
package com.example.scope;
public class Scope {
  private String name;
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  @Override
  public String toString() {
    return "Scope [name=" + name + "]";
  }
}
```

Main class:

```
package com.example.scope;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Main {
  public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext(
      "configFiles/Scope.xml");
    Scope scope = (Scope) ctx.getBean("scopeTest");
    Scope scopeDuplicate = (Scope) ctx.getBean("scopeTestDuplicate");
    System.out.println(scope == scopeDuplicate);
    System.out.println(scope + "::" + scopeDuplicate);
  }
}
```

Here, I create two beans of the Scope class and make Spring Scope a singleton, now checking the references.

This is where interviewees sometimes get confused. Usually, I will get three types of answers:

- This code will not compile. It will throw an error at runtime, as you cannot define two Spring beans of the same class with Singleton Scope in XML. (Very rare)
- The reference check will return true, as the container maintains one object. Both bean definitions will return the same object, so the memory location would be the same. (Often)
- The reference check will return false, which means Spring Singletons don't work like they said earlier. (A few)

The third answer is the correct answer. A Spring Singleton does not work like a Java Singleton.

If we see the output of the program, we will understand that it will return two different instances, so in a container, there may be more than one object in spite of the fact that the Scope is the singleton.

Output:

Reference Check ::false
Scope [name=Shamik Mitra]::Scope [name=Samir Mitra]

So, let's ask the question again. "What do you mean by Spring Singleton Scope?"

According to the Spring documentation:

"When a bean is a singleton, only one shared instance of the bean will be managed, and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, then the Spring IoC container will create exactly one instance of the object defined by that bean definition. This single instance will be stored in a cache of such singleton beans, and all subsequent requests and references for that named bean will result in the cached object being returned."

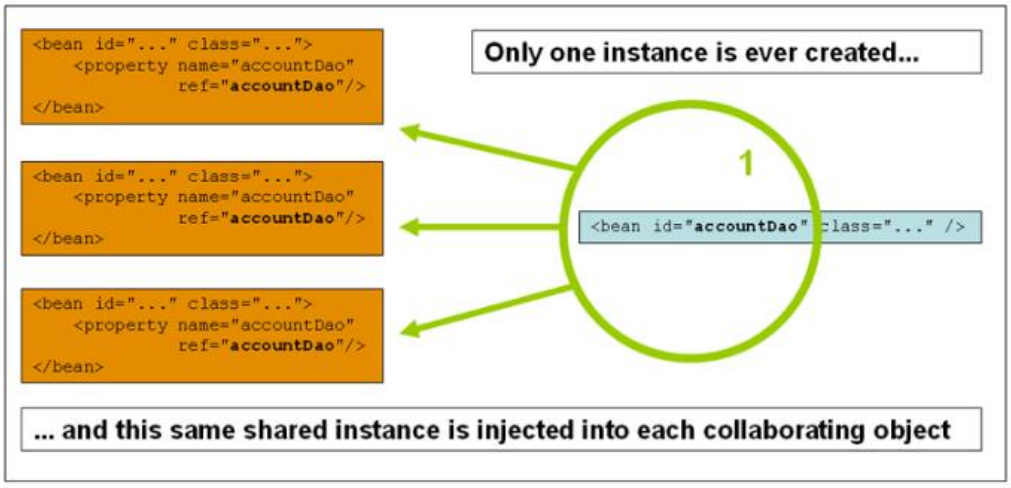
So, for a given id, a Spring container maintains only one shared instance in a singleton cache.

In my example, I use two different ids (scopeTest and ScopeTestDuplicate), so the Spring container creates two instances of the same class and binds them with respective ids, then stores them in a Singleton cache.

You can think of a Spring container as managing a key-value pair, where the key is the id or name of the bean and the value is the bean itself. So, for a given key, it maintains a Singleton. So, if we use that key as a reference to or of other beans, the same bean will be injected to those other beans.

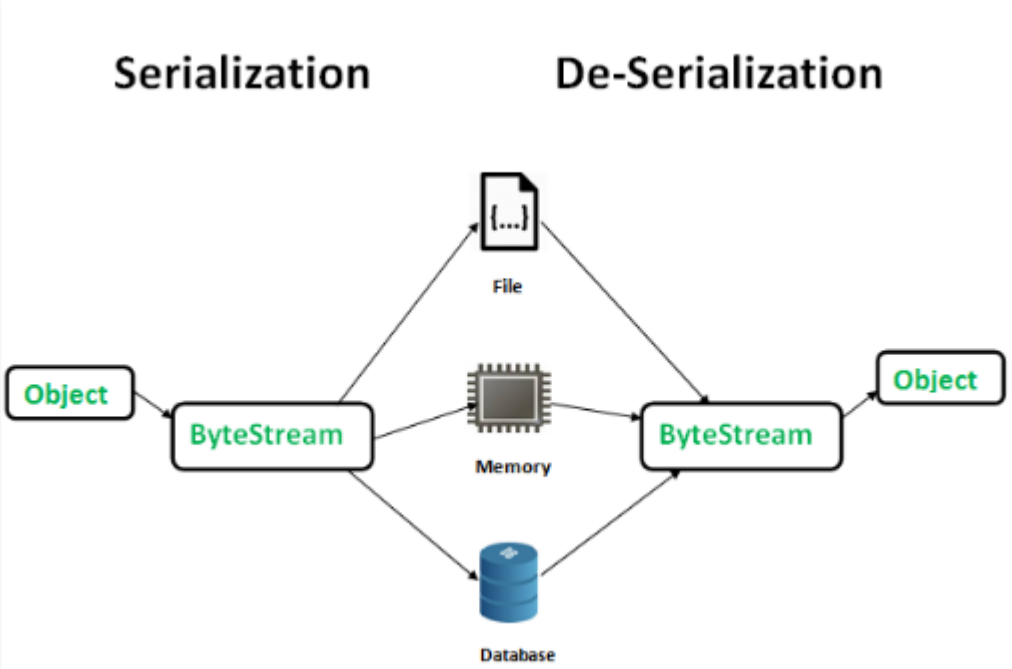
In summation, Spring guarantees exactly one shared bean instance for the given id per IoC container, unlike Java Singletons, where the Singleton hardcodes the scope of an object such that one and only one instance of a class will ever be created per ClassLoader.

And we'll close on this picture taken from the Spring docs.



Serialization and Deserialization in Java with Example

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the java.io.Serializable interface.
The ObjectOutputStream class contains writeObject() method for serializing an Object.

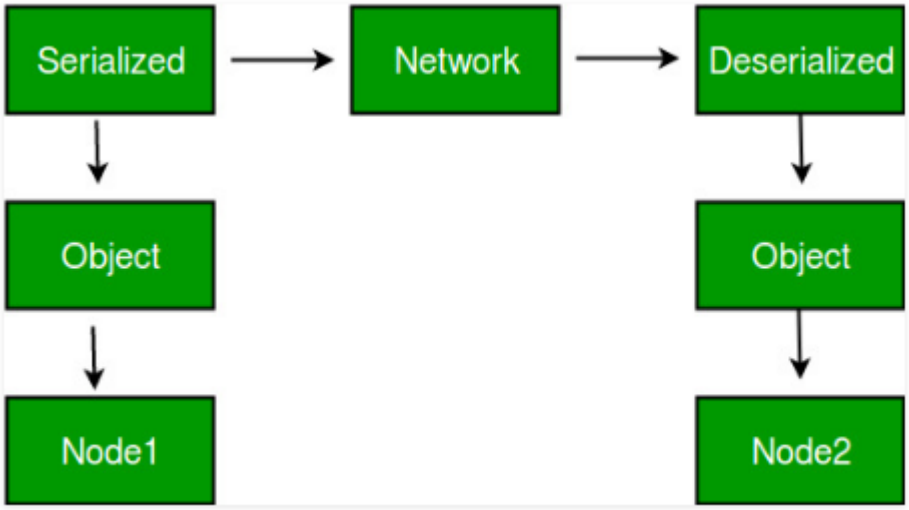
```
public final void writeObject(Object obj)
                    throws IOException
```

The ObjectInputStream class contains readObject() method for deserializing an object.

```
public final Object readObject()
                throws IOException,
              ClassNotFoundException
```

Advantages of Serialization

1. To save/persist state of an object.
2. To travel an object across a network.



Only the objects of those classes can be serialized which are implementing java.io.Serializableinterface.
Serializable is a marker interface (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

Points to remember

- If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.
- Only non-static data members are saved via Serialization process.
- Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.
- Constructor of object is never called when an object is deserialized.
- Associated objects must be implementing Serializable interface.

Example:

```
class A implements Serializable{
// B also implements Serializable
// interface.
B ob=new B();
}
```

SerialVersionUID

The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the reciever has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an InvalidClassException.

A Serializable class can declare its own UID explicitly by declaring a field name.
It must be static, final and of type long.

i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

It is also recommended to use private modifier for UID since it is not useful as inherited member.

Serialver

The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.
You can run the following command to get serialVersionUID

serialver [-classpath classpath] [-show] [classname…]

Example 1:

```
// Java code for serialization and deserialization
// of a Java object
import java.io.*;

class Demo implements java.io.Serializable
{
   public int a;
   public String b;

   // Default constructor
   public Demo(int a, String b)
   {
      this.a = a;
      this.b = b;
   }

}

class Test
{
   public static void main(String[] args)
   {
      Demo object = new Demo(1, "geeksforgeeks");
      String filename = "file.ser";

      // Serialization
      try
      {
         //Saving of object in a file
         FileOutputStream file = new FileOutputStream(filename);
         ObjectOutputStream out = new ObjectOutputStream(file);

         // Method for serialization of object
         out.writeObject(object);
```

```
            out.close();
            file.close();

            System.out.println("Object has been serialized");

        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }

        Demo object1 = null;

        // Deserialization
        try
        {
            // Reading the object from a file
            FileInputStream file = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(file);

            // Method for deserialization of object
            object1 = (Demo)in.readObject();

            in.close();
            file.close();

            System.out.println("Object has been deserialized ");
            System.out.println("a = " + object1.a);
            System.out.println("b = " + object1.b);
        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }

        catch(ClassNotFoundException ex)
        {
            System.out.println("ClassNotFoundException is caught");
        }

    }
}
```

Output :

```
Object has been serialized
Object has been deserialized
a = 1
b = geeksforgeeks
```

Fail Fast and Fail-Safe Iterators in Java

In this article, I am going to explain how those collections behave which doesn't iterate as fail-fast. First, there is no term as fail-safe given in many places as Java SE specifications does not use this term. I am using fail safe to segregate between Fail fast and Non fail-fast iterators.

Concurrent Modification: Concurrent Modification in programming means to modify an object concurrently when another task is already running over it. For example, in Java to modify a collection when another thread is iterating over it. Some Iterator implementations (including those of all the general-purpose collection implementations provided by the JRE) may choose to throw **ConcurrentModificationException** if this behavior is detected.

Fail Fast and Fail-Safe Iterators in Java

Iterators in java are used to iterate over the Collection objects. Fail-Fast iterators immediately throw ConcurrentModificationException if there is structural modification of the collection. Structural modification means adding, removing or updating any element from collection while a thread is iterating over that collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.

Fail-Safe iterators don't throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that's why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.

How Fail Fast Iterator works?

To know whether the collection is structurally modified or not, fail-fast iterators use an internal flag called modCount which is updated each time a collection is modified. Fail-fast iterators checks the modCount flag whenever it gets the next value (i.e. using next() method), and if it finds that the modCount has been modified after this iterator has been created, it throws ConcurrentModificationException.

```
/ Java code to illustrate
// Fail Fast Iterator in Java
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class FailFastExample {
   public static void main(String[] args)
   {
      Map<String, String> cityCode = new HashMap<String, String>();
      cityCode.put("Delhi", "India");
      cityCode.put("Moscow", "Russia");
      cityCode.put("New York", "USA");

      Iterator iterator = cityCode.keySet().iterator();

      while (iterator.hasNext()) {
         System.out.println(cityCode.get(iterator.next()));

         // adding an element to Map
         // exception will be thrown on next call
         // of next() method.
         cityCode.put("Istanbul", "Turkey");
      }
   }
}
```

Output:

```
India
Exception in thread "main" java.util.ConcurrentModificationException
   at java.util.HashMap$HashIterator.nextNode(HashMap.java:1442)
   at java.util.HashMap$KeyIterator.next(HashMap.java:1466)
   at FailFastExample.main(FailFastExample.java:18)
```

Important points of fail-fast iterators:

- These iterators throw ConcurrentModificationException if a collection is modified while iterating over it.
- They use original collection to traverse over the elements of the collection.
- These iterators don't require extra memory.
- Ex: Iterators returned by ArrayList, Vector, HashMap.

Note 1(from java-docs): The fail-fast behavior of an iterator cannot be guaranteed as it is impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

Note 2: If you remove an element via Iterator remove() method, exception will not be thrown. However, in case of removing via a particular collection remove() method ,

ConcurrentModificationException will be thrown.

Below code snippet will demonstrate this:

```
// Java code to demonstrate remove
// case in Fail-fast iterators

import java.util.ArrayList;
import java.util.Iterator;

public class FailFastExample {
   public static void main(String[] args)
   {
      ArrayList<Integer> al = new ArrayList<>();
      al.add(1);
      al.add(2);
      al.add(3);
      al.add(4);
      al.add(5);

      Iterator<Integer> itr = al.iterator();
      while (itr.hasNext()) {
         if (itr.next() == 2) {
            // will not throw Exception
            itr.remove();
         }
      }

      System.out.println(al);

      itr = al.iterator();
      while (itr.hasNext()) {
         if (itr.next() == 3) {
            // will throw Exception on
            // next call of next() method
            al.remove(3);
         }
      }
   }
}
```

Output :

```
[1, 3, 4, 5]
Exception in thread "main" java.util.ConcurrentModificationException
   at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
   at java.util.ArrayList$Itr.next(ArrayList.java:851)
   at FailFastExample.main(FailFastExample.java:28)
```

Fail Safe Iterator

First, there is no term as fail-safe given in many places as Java SE specifications does not use this term. I am using this term to demonstrate the difference between Fail Fast and Non-Fail Fast Iterator. These iterators make a copy of the internal collection (object array) and iterates over the

copied collection. Any structural modification done to the iterator affects the copied collection, not original collection. So, original collection remains structurally unchanged.

- Fail-safe iterators allow modifications of a collection while iterating over it.
- These iterators don't throw any Exception if a collection is modified while iterating over it.
- They use copy of original collection to traverse over the elements of the collection.
- These iterators require extra memory for cloning of collection. Ex : ConcurrentHashMap, CopyOnWriteArrayList

Example of Fail Safe Iterator in Java:

```
// Java code to illustrate
// Fail Safe Iterator in Java
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.Iterator;

class FailSafe {
   public static void main(String args[])
   {
      CopyOnWriteArrayList<Integer> list
         = new CopyOnWriteArrayList<Integer>(new Integer[] { 1, 3, 5, 8 });
      Iterator itr = list.iterator();
      while (itr.hasNext()) {
         Integer no = (Integer)itr.next();
         System.out.println(no);
         if (no == 8)

            // This will not print,
            // hence it has created separate copy
            list.add(14);
      }
   }
}
```

Output:

```
1
3
5
8
```

Also, those collections which don't use fail-fast concept may not necessarily create clone/snapshot of it in memory to avoid ConcurrentModificationException. For example, in case of ConcurrentHashMap, it does not operate on a separate copy although it is not fail-fast. Instead, it has semantics that is described by the official specification as weakly consistent(memory consistency properties in Java). Below code snippet will demonstrate this:

Example of Fail-Safe Iterator which does not create separate copy

```
// Java program to illustrate
// Fail-Safe Iterator which
// does not create separate copy
import java.util.concurrent.ConcurrentHashMap;
import java.util.Iterator;

public class FailSafeItr {
   public static void main(String[] args)
   {

      // Creating a ConcurrentHashMap
      ConcurrentHashMap<String, Integer> map
         = new ConcurrentHashMap<String, Integer>();

      map.put("ONE", 1);
      map.put("TWO", 2);
      map.put("THREE", 3);
      map.put("FOUR", 4);

      // Getting an Iterator from map
      Iterator it = map.keySet().iterator();

      while (it.hasNext()) {
         String key = (String)it.next();
         System.out.println(key + " : " + map.get(key));

         // This will reflect in iterator.
         // Hence, it has not created separate copy
         map.put("SEVEN", 7);
      }
   }
}
```

Output

```
ONE : 1
FOUR : 4
TWO : 2
THREE : 3
SEVEN : 7
```

Note (from java-docs): The iterators returned by ConcurrentHashMap is weakly consistent. This means that this iterator can tolerate concurrent modification, traverse's elements as they existed when iterator was constructed and may (but not guaranteed to) reflect modifications to the collection after the construction of the iterator.

Difference between Fail Fast Iterator and Fail-Safe Iterator

The major difference is fail-safe iterator doesn't throw any Exception, contrary to fail-fast Iterator.This is because they work on a clone of Collection instead of the original collection and that's why they are called as the fail-safe iterator.

https://www.journaldev.com/2623/spring-autowired-annotation

Spring Boot: Creating Microservices on Java

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 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 an 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 be 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 many 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 changes 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 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 its 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.

Microservices: Spring Cloud Eureka Server Configuration

What Is the Eureka Server?

The Eureka server is nothing but a service discovery pattern implementation, where every microservice is registered and a client microservice looks up the Eureka server to get a dependent microservice to get the job done.

The Eureka Server is a Netflix OSS product, and Spring Cloud offers a declarative way to register and invoke services by Java annotation.

Why Is the Eureka Server?

To understand why the Eureka server is needed in microservice architecture, let's understand how one service calls another service REST endpoint for communication.

Say we need to call the employee payroll service to get payroll information for an employee. The payroll service is deployed on the localhost 8080 port (we got that information by passing the employee primary key) so we just call the following:

http://localhost:8080/payroll/1 or http://127.0.0.1/payroll/1

Where localhost/127.0.0.1 is the hostname/IP address and payroll is the payroll service context, 1 is the employee primary key.

But this is only possible when you know the hostname/IP addresses beforehand, then you can configure your URL. So here the hostname or IP address is a constraint or a pain point.

If the IP address of a server/container is fixed, then you can use that approach, but what happens when your IP addresses and hostname are unpredictable?

Nowadays, on a cloud platform, it is obvious that all the servers or containers use dynamic IPs for autoscaling. And the interesting thing is that in microservice architecture, the key principle is that your service can autoscaled as per load, so cloud platforms are ideal for microservices.

What I am trying to say here is that we can not predict the IP addresses of the container/server beforehand, so putting dependent services IP addresses in the config file is not a solution. We need a more sophisticated technique to identify the service, and Eureka server steps in here.

Eureka Server/Client Communication

Every microservice registers itself in the Eureka server when bootstrapped, generally using the {ServiceId} it registers into the Eureka server, or it can use the hostname or any public IP (if those are fixed). After registering, every 30 seconds, it pings the Eureka server to notify it that the service itself is available. If the Eureka server not getting any pings from a service for a quite long time, this

service is unregistered from the Eureka server automatically and the Eureka server notifies the new state of the registry to all other services. I will write this mechanism elaborately in the next article.

Now one question may pop up our mind: what is the Eureka server itself?

The Eureka server is nothing but another microservice which treats itself as a Eureka client.

What I mean is that the Eureka server has to be highly available as every service communicates it to discover other services. So it is recommended that it should not be a single point of failure. To overcome it, we need multiple Eureka server instances running behind a load balancer. Now when there are multiple Eureka servers, each server needs to have synchronized registry details so that every server knows about the current state of every microservice registered in the Eureka server registry.

The Eureka server communicates its peer server as a client and clones the updated state of the registry, so the Eureka server itself acts as a client. We can perform this by just configuring the eureka.client.serviceUrl.defaultZone property.

The Eureka server works in two modes:

- Standalone: in local, we configure a stand-alone mode where we have only one Eureka server (localhost) and the same cloning property from itself.
- Clustered: we have multiple Eureka servers, each cloning its states from its peer.

The Eureka server can have a properties file and communicate with a config server as other microservices do.

PFB link of Standalone Setup of Eureka Server

Link : https://dzone.com/articles/microservice-spring-cloud-eureka-server-configurat

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:

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.

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.

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 or a normal java class. For our factory design pattern example, we have abstract super class with overridden **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;
	}

}
```

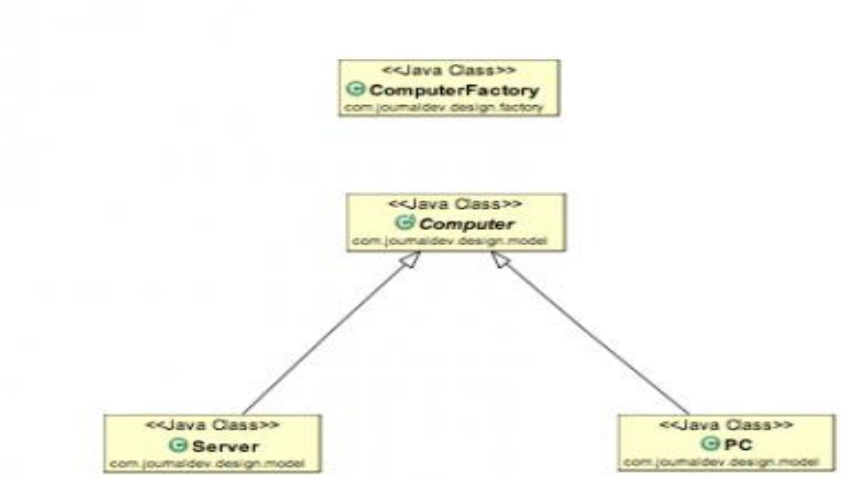
Factory Class

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.
- Notice that based on the input parameter, different subclass is created and returned. getComputer is the factory method.



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, Bridge Pattern, Composite Pattern) and uses abstract classes or interface with composition to implement.

We use inheritance 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 or 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, furthermore 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 furthermore 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 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();
}
```

2. **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.");
	}

}
```

3. **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.

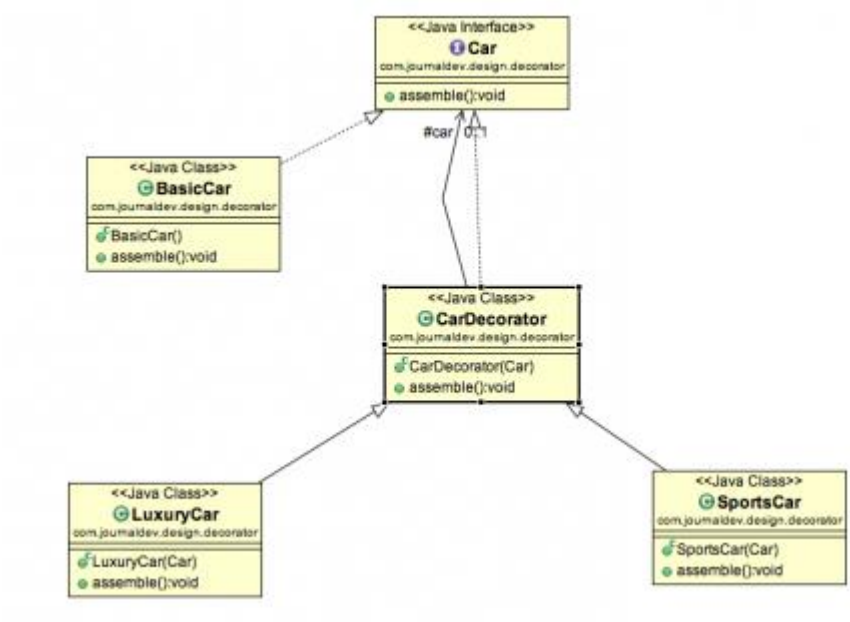
```
package com.journaldev.design.decorator;

public class CarDecorator implements Car {

	protected Car car;

	public CarDecorator(Car c){
		this.car=c;
	}

	@Override
	public void assemble() {
		this.car.assemble();
	}

}
```

4. **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



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. It's 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 classes, such as FileReader, BufferedReader etc.

https://www.journaldev.com/2696/spring-interview-questions-and-answers

Spring Interview Questions and Answers

What is Spring Framework?

Spring is one of the most widely used Java EE framework. Spring framework core concepts are "Dependency Injection" and "Aspect Oriented Programming".

Spring framework can be used in normal java applications also to achieve loose coupling between different components by implementing dependency injection and we can perform cross cutting tasks such as logging and authentication using spring support for aspect oriented programming.

I like spring because it provides a lot of features and different modules for specific tasks such as Spring MVC and Spring JDBC. Since it's an open source framework with a lot of online resources and active community members, working with Spring framework is easy and fun at same time.

What are some of the important features and advantages of Spring Framework?

Spring Framework is built on top of two design concepts – Dependency Injection and Aspect Oriented Programming.

Some of the features of spring framework are:

- Lightweight and very little overhead of using framework for our development.
- Dependency Injection or Inversion of Control to write components that are independent of each other, spring container takes care of wiring them together to achieve our work.
- Spring IoC container manages Spring Bean life cycle and project specific configurations such as JNDI lookup.
- Spring MVC framework can be used to create web applications as well as restful web services capable of returning XML as well as JSON response.

- Support for transaction management, JDBC operations, File uploading, Exception Handling etc with very little configurations, either by using annotations or by spring bean configuration file.

Some of the advantages of using Spring Framework are:

- Reducing direct dependencies between different components of the application, usually Spring IoC container is responsible for initializing resources or beans and inject them as dependencies.
- Writing unit test cases are easy in Spring framework because our business logic doesn't have direct dependencies with actual resource implementation classes. We can easily write a test configuration and inject our mock beans for testing purposes.
- Reduces the amount of boiler-plate code, such as initializing objects, open/close resources. I like JdbcTemplate class a lot because it helps us in removing all the boiler-plate code that comes with JDBC programming.
- Spring framework is divided into several modules, it helps us in keeping our application lightweight. For example, if we don't need Spring transaction management features, we don't need to add that dependency in our project.
- Spring framework support most of the Java EE features and even much more. It's always on top of the new technologies, for example there is a Spring project for Android to help us write better code for native android applications. This makes spring framework a complete package and we don't need to look after different framework for different requirements.

What do you understand by Dependency Injection?

Dependency Injection design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable. We can implement dependency injection pattern to move the dependency resolution from compile-time to runtime.

Some of the benefits of using Dependency Injection are: Separation of Concerns, Boilerplate Code reduction, Configurable components and easy unit testing.

Read more at Dependency Injection Tutorial. We can also use Google Guice for Dependency Injectionto automate the process of dependency injection. But in most of the cases we are looking for more than just dependency injection and that's why Spring is the top choice for this.

How do we implement DI in Spring Framework?

We can use Spring XML based as well as Annotation based configuration to implement DI in spring applications. For better understanding, please read Spring Dependency Injection example where you can learn both the ways with JUnit test case. The post also contains sample project zip file, that you can download and play around to learn more.

What are the benefits of using Spring Tool Suite?

We can install plugins into Eclipse to get all the features of Spring Tool Suite. However, STS comes with Eclipse with some other important stuffs such as Maven support, Templates for creating different types of Spring projects and tc server for better performance with Spring applications.

I like STS because it highlights the Spring components and if you are using AOP pointcuts and advices, then it clearly shows which methods will come under the specific pointcut. So rather than installing everything on our own, I prefer using STS when developing Spring based applications.

Name some of the important Spring Modules?

Some of the important Spring Framework modules are:

- Spring Context – for dependency injection.
- Spring AOP – for aspect oriented programming.
- Spring DAO – for database operations using DAO pattern
- Spring JDBC – for JDBC and DataSource support.
- Spring ORM – for ORM tools support such as Hibernate
- Spring Web Module – for creating web applications.
- Spring MVC – Model-View-Controller implementation for creating web applications, web services etc.

What do you understand by Aspect Oriented Programming?

Enterprise applications have some common cross-cutting concerns that is applicable for different types of Objects and application modules, such as logging, transaction management, data validation, authentication etc. In Object Oriented Programming, modularity of application is achieved by Classes whereas in AOP application modularity is achieved by Aspects and they are configured to cut across different classes methods.

AOP takes out the direct dependency of cross-cutting tasks from classes that is not possible in normal object-oriented programming. For example, we can have a separate class for logging but again the classes will have to call these methods for logging the data. Read more about Spring AOP support at Spring AOP Example.

What is Aspect, Advice, Pointcut, JointPoint and Advice Arguments in AOP?

- **Aspect:** Aspect is a class that implements cross-cutting concerns, such as transaction management. Aspects can be a normal class configured and then configured in Spring Bean configuration file or we can use Spring AspectJ support to declare a class as Aspect using @Aspect annotation.
- **Advice:** Advice is the action taken for a particular join point. In terms of programming, they are methods that gets executed when a specific join point with matching pointcut is reached in the application. You can think of Advices as Spring interceptors or Servlet Filters.
- **Pointcut:** Pointcut are regular expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points. Spring framework uses the AspectJ pointcut expression language for determining the join points where advice methods will be applied.
- **Join Point:** A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.
- **Advice Arguments:** We can pass arguments in the advice methods. We can use args() expression in the pointcut to be applied to any method that matches the argument pattern. If we use this, then we need to use the same name in the advice method from where argument type is determined.

  These concepts seems confusing at first, but if you go through Spring Aspect, Advice Example then you can easily relate to them.

What is the difference between Spring AOP and AspectJ AOP?

AspectJ is the industry-standard implementation for Aspect Oriented Programming whereas Spring implements AOP for some cases. Main differences between Spring AOP and AspectJ are:

- Spring AOP is simpler to use than AspectJ because we don't need to worry about the weaving process.
- Spring AOP supports AspectJ annotations, so if you are familiar with AspectJ then working with Spring AOP is easier.
- Spring AOP supports only proxy-based AOP, so it can be applied only to method execution join points. AspectJ support all kinds of pointcuts.
- One of the shortcoming of Spring AOP is that it can be applied only to the beans created through Spring Context.

What is Spring IoC Container?

Inversion of Control (IoC) is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, the objects define their dependencies that are being injected by other assembler objects. Spring IoC container is the program that injects dependencies into an object and make it ready for our use.

Spring Framework IoC container classes are part of **org.springframework.beans** and **org.springframework.context packages** and provides us different ways to decouple the object dependencies.

Some of the useful ApplicationContext implementations that we use are;

- **AnnotationConfigApplicationContext:** For standalone java applications using annotations based configuration.
- **ClassPathXmlApplicationContext:** For standalone java applications using XML based configuration.
- **FileSystemXmlApplicationContext:** Like ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
- **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

What is a Spring Bean?

Any normal java class that is initialized by Spring IoC container is called Spring Bean. We use Spring **ApplicationContext** to get the Spring Bean instance.

Spring IoC container manages the life cycle of Spring Bean, bean scopes and injecting any required dependencies in the bean.

What is the importance of Spring bean configuration file?

We use Spring Bean configuration file to define all the beans that will be initialized by Spring Context. When we create the instance of Spring ApplicationContext, it reads the spring bean xml file and initialize all of them. Once the context is initialized, we can use it to get different bean instances.

Apart from Spring Bean configuration, this file also contains spring MVC interceptors, view resolvers and other elements to support annotations based configurations.

What are different ways to configure a class as Spring Bean?

There are three different ways to configure Spring Bean.

- XML Configuration: This is the most popular configuration and we can use bean element in context file to configure a Spring Bean. For example:

```
<bean name="myBean" class="com.journaldev.spring.beans.MyBean"></bean>
```

- Java Based Configuration: If you are using only annotations, you can configure a Spring bean using **@Bean** annotation. This annotation is used with **@Configuration** classes to configure a spring bean. Sample configuration is:

```
@Configuration
@ComponentScan(value="com.journaldev.spring.main")
public class MyConfiguration {
	@Bean
	public MyService getService(){
		return new MyService();
	}
}
```

To get this bean from spring context, we need to use following code snippet:

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(
			MyConfiguration.class);
MyService service = ctx.getBean(MyService.class);
```

Annotation Based Configuration: We can also use @Component, @Service, @Repository and @Controller annotations with classes to configure them to be as spring bean. For these, we would need to provide base package location to scan for these classes. For example:

```
<context:component-scan base-package="com.journaldev.spring" />
```

What are different scopes of Spring Bean?

There are five scopes defined for Spring Beans.

- **singleton:** Only one instance of the bean will be created for each container. This is the default scope for the spring beans. While using this scope, make sure spring bean doesn't have shared instance variables otherwise it might lead to data inconsistency issues because it's not thread-safe.
- **prototype:** A new instance will be created every time the bean is requested.
- **request:** This is same as prototype scope, however it's meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
- **session:** A new bean will be created for each HTTP session by the container.
- **global-session:** This is used to create global session beans for Portlet applications.

Spring Framework is extendable and we can create our own scopes too, however most of the times we are good with the scopes provided by the framework.

To set spring bean scopes we can use "scope" attribute in bean element or @Scope annotation for annotation based configurations.

What is Spring Bean life cycle?

Spring Beans are initialized by Spring Container and all the dependencies are also injected. When context is destroyed, it also destroys all the initialized beans. This works well in most of the cases but sometimes we want to initialize other resources or do some validation before making our beans ready to use. Spring framework provides support for post-initialization and pre-destroy methods in spring beans.

We can do this by two ways – by implementing **InitializingBean** and **DisposableBean** interfaces or using init-method and destroy-method attribute in spring bean configurations. For more details, please read Spring Bean Life Cycle Methods.

How to get ServletContext and ServletConfig object in a Spring Bean?

There are two ways to get Container specific objects in the spring bean.

- Implementing Spring *Aware interfaces, for these ServletContextAware and ServletConfigAware interfaces, for complete example of these aware interfaces, please read Spring Aware Interfaces
- Using **@Autowired** annotation with bean variable of type **ServletContext** and **ServletConfig**. They will work only in servlet container specific environment only though.

```
@Autowired
ServletContext servletContext;
```

What is Bean wiring and @Autowired annotation?

The process of injection spring bean dependencies while initializing it called Spring Bean Wiring.

Usually it's best practice to do the explicit wiring of all the bean dependencies, but spring framework also supports autowiring. We can use **@Autowired** annotation with fields or methods for **autowiring** byType. For this annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done by context:annotation-config element.

For more details about **@Autowired** annotation, please read Spring Autowire Example.

What are different types of Spring Bean autowiring?

There are four types of autowiring in Spring framework.

- autowire byName
- autowire byType
- autowire by constructor
- autowiring by **@Autowired** and **@Qualifier** annotations

Prior to Spring 3.1, autowire by autodetect was also supported that was similar to autowire by constructor or byType. For more details about these options, please read Spring Bean Autowiring.

Does Spring Bean provide thread safety?

The default scope of Spring bean is singleton, so there will be only one instance per context. That means that all the having a class level variable that any thread can update will lead to inconsistent data. Hence in default mode spring beans are not thread-safe.

However we can change spring bean scope to request, prototype or session to achieve thread-safety at the cost of performance. It's a design decision and based on the project requirements.

What is a Controller in Spring MVC?

Just like MVC design pattern, Controller is the class that takes care of all the client requests and send them to the configured resources to handle it. In Spring MVC, **org.springframework.web.servlet.DispatcherServlet** is the front controller class that initializes the context based on the spring beans configurations.

A Controller class is responsible to handle different kind of client requests based on the request mappings. We can create a controller class by using **@Controller** annotation. Usually it's used with **@RequestMapping** annotation to define handler methods for specific URI mapping.

What's the difference between @Component, @Controller, @Repository & @Service annotations in Spring?

@Component is used to indicate that a class is a component. These classes are used for auto detection and configured as bean, when annotation based configurations are used.

@Controller is a specific type of component, used in MVC applications and mostly used with RequestMapping annotation.

@Repository annotation is used to indicate that a component is used as repository and a mechanism to store/retrieve/search data. We can apply this annotation with DAO pattern implementation classes.

@Service is used to indicate that a class is a Service. Usually the business facade classes that provide some services are annotated with this.

We can use any of the above annotations for a class for auto-detection but different types are provided so that you can easily distinguish the purpose of the annotated classes.

What is DispatcherServlet and ContextLoaderListener?

DispatcherServlet is the front controller in the Spring MVC application and it loads the spring bean configuration file and initialize all the beans that are configured. If annotations are enabled, it also scans the packages and configure any bean annotated with **@Component**, **@Controller**, **@Repository** or **@Service** annotations.

ContextLoaderListener is the listener to start up and shut down Spring's root **WebApplicationContext**. It's important functions are to tie up the lifecycle of **ApplicationContext** to the lifecycle of the **ServletContext** and to automate the creation of **ApplicationContext**. We can use it to define shared beans that can be used across different spring contexts.

What is ViewResolver in Spring?

ViewResolver implementations are used to resolve the view pages by name. Usually we configure it in the spring bean configuration file. For example:

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
```

```
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

InternalResourceViewResolver is one of the implementation of **ViewResolver** interface and we are providing the view pages directory and suffix location through the bean properties. So if a controller handler method returns “home”, view resolver will use view page located at /WEB-INF/views/home.jsp.

What is a MultipartResolver and when its used?

MultipartResolver interface is used for uploading files
– **CommonsMultipartResolver** and **StandardServletMultipartResolver** are two implementations provided by spring framework for file uploading. By default there are no multipart resolvers configured but to use them for uploading files, all we need to define a bean named “multipartResolver” with type as MultipartResolver in spring bean configurations.

Once configured, any multipart request will be resolved by the configured MultipartResolver and pass on a wrapped HttpServletRequest. Then it’s used in the controller class to get the file and process it. For a complete example, please read Spring MVC File Upload Example.

How to handle exceptions in Spring MVC Framework?

Spring MVC Framework provides following ways to help us achieving robust exception handling.

- **Controller Based –** We can define exception handler methods in our controller classes. All we need is to annotate these methods with @ExceptionHandler annotation.
- **Global Exception Handler –** Exception Handling is a cross-cutting concern and Spring provides @ControllerAdvice annotation that we can use with any class to define our global exception handler.
- **HandlerExceptionResolver implementation –** For generic exceptions, most of the times we serve static pages. Spring Framework provides HandlerExceptionResolver interface that we can implement to create global exception handler. The reason behind this additional way to define global exception handler is that Spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.

For a complete example, please read Spring Exception Handling Example.

How to create ApplicationContext in a Java Program?

There are following ways to create spring context in a standalone java program.

AnnotationConfigApplicationContext: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.

ClassPathXmlApplicationContext: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.

FileSystemXmlApplicationContext: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.

Can we have multiple Spring configuration files?

For Spring MVC applications, we can define multiple spring context configuration files through **contextConfigLocation**. This location string can consist of multiple locations separated by any number of commas and spaces. For example;

```
<servlet>
		<servlet-name>appServlet</servlet-name>
		<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<init-param>
	<param-name>contextConfigLocation</param-name>
		  <param-value>/WEB-INF/spring/appServlet/servlet-context.xml,/WEB-
INF/spring/appServlet/servlet-jdbc.xml</param-value>
		</init-param>
		<load-on-startup>1</load-on-startup>
</servlet>
```

We can also define multiple root level spring configurations and load it through context-param. For example;

```
<context-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/spring/root-context.xml /WEB-INF/spring/root-
security.xml</param-value>
</context-param>
```

Another option is to use import element in the context configuration file to import other configurations, for example:

```
<beans:import resource="spring-jdbc.xml"/>
```

What is ContextLoaderListener?

ContextLoaderListener is the listener class used to load root context and define spring bean configurations that will be visible to all other contexts. It's configured in web.xml file

```
<context-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>

<listener>
		<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

What are the minimum configurations needed to create Spring MVC application?

For creating a simple Spring MVC application, we would need to do following tasks.

- Add **spring-context** and **spring-webmvc** dependencies in the project.
- Configure **DispatcherServlet** in the web.xml file to handle requests through spring container.
- Spring bean configuration file to define beans, if using annotations then it has to be configured here. Also we need to configure view resolver for view pages.
- Controller class with request mappings defined to handle the client requests.

Above steps should be enough to create a simple Spring MVC Hello World application.

How would you relate Spring MVC Framework to MVC architecture?

As the name suggests Spring MVC is built on top of **Model-View-controller** architecture. **DispatcherServlet** is the Front Controller in the Spring MVC application that takes care of all the incoming requests and delegate it to different controller handler methods.

Model can be any Java Bean in the Spring Framework, just like any other MVC framework Spring provides automatic binding of form data to java beans. We can set model beans as attributes to be used in the view pages.

View Pages can be JSP, static HTMLs etc. and view resolvers are responsible for finding the correct view page. Once the view page is identified, control is given back to the DispatcherServlet controller. DispatcherServlet is responsible for rendering the view and returning the final response to the client.

How to achieve localization in Spring MVC applications?

Spring provides excellent support for localization or i18n through resource bundles. Basis steps needed to make our application localized are:

- Creating message resource bundles for different locales, such as messages_en.properties, messages_fr.properties etc.
- Defining messageSource bean in the spring bean configuration file of type **ResourceBundleMessageSource** or **ReloadableResourceBundleMessageSource**.
- For change of locale support, define localeResolver bean of type CookieLocaleResolver and configure LocaleChangeInterceptor interceptor. Example configuration can be like below:

```
<beans:bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
<beans:property name="basename" value="classpath:messages" />
<beans:property name="defaultEncoding" value="UTF-8" />
</beans:bean>

<beans:bean id="localeResolver"
   class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
   <beans:property name="defaultLocale" value="en" />
   <beans:property name="cookieName" value="myAppLocaleCookie"></beans:property>
   <beans:property name="cookieMaxAge" value="3600"></beans:property>
</beans:bean>

<interceptors>
   <beans:bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
      <beans:property name="paramName" value="locale" />
   </beans:bean>
</interceptors>
```

- Use **spring:message** element in the view pages with key names, DispatcherServlet picks the corresponding value and renders the page in corresponding locale and return as response.

For a complete example, please read Spring Localization Example.

How can we use Spring to create Restful Web Service returning JSON response?

We can use Spring Framework to create Restful web services that returns JSON data. Spring provides integration with Jackson JSON API that we can use to send JSON response in restful web service.

We would need to do following steps to configure our Spring MVC application to send JSON response:

- 1. Adding Jackson JSON dependencies, if you are using Maven it can be done with following code:

```
<!-- Jackson -->
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>${jackson.databind-version}</version>
</dependency>
```

- Configure **RequestMappingHandlerAdapter** bean in the spring bean configuration file and set the messageConverters property to MappingJackson2HttpMessageConverter bean. Sample configuration will be:

```
<!-- Configure to plugin JSON as request and response in method handler -->
<beans:bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"
>
   <beans:property name="messageConverters">
      <beans:list>
         <beans:ref bean="jsonMessageConverter"/>
      </beans:list>
   </beans:property>
</beans:bean>

<!-- Configure bean to convert JSON to POJO and vice versa -->
<beans:bean id="jsonMessageConverter"
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
</beans:bean>
```

- In the controller handler methods, return the Object as response using **@ResponseBody** annotation. Sample code:

```
@RequestMapping(value = EmpRestURIConstants.GET_EMP, method = RequestMethod.GET)
public @ResponseBody Employee getEmployee(@PathVariable("id") int empId) {
   logger.info("Start getEmployee. ID="+empId);
   return empData.get(empId);
}
```

- You can invoke the rest service through any API, but if you want to use Spring then we can easily do it using RestTemplate class.

For a complete example, please read Spring Restful Webservice Example.

What are some of the important Spring annotations you have used?

Some of the Spring annotations that I have used in my project are:

- **@Controller –** for controller classes in Spring MVC project.
- **@RequestMapping –** for configuring URI mapping in controller handler methods. This is a very important annotation, so you should go through Spring MVC RequestMapping Annotation Examples
- **@ResponseBody –** for sending Object as response, usually for sending XML or JSON data as response.
- **@PathVariable –** for mapping dynamic values from the URI to handler method arguments.
- **@Autowired –** for autowiring dependencies in spring beans.
- **@Qualifier –** with @Autowired annotation to avoid confusion when multiple instances of bean type is present.
- **@Service –** for service classes.
- **@Scope –** for configuring scope of the spring bean.
- **@Configuration**, **@ComponentScan** and **@Bean** – for java based configurations.
- AspectJ annotations for configuring aspects and advices, **@Aspect**, **@Before**, **@After**, **@Around**, **@Pointcut** etc.

Can we send an Object as the response of Controller handler method?

Yes, we can, using **@ResponseBody** annotation. This is how we send JSON or XML based response in restful web services.

How to upload file in Spring MVC Application?

Spring provides built-in support for uploading files through **MultipartResolver** interface implementations. It's very easy to use and requires only configuration changes to get it working. Obviously, we would need to write controller handler method to handle the incoming file and process it. For a complete example, please refer Spring File Upload Example.

How to validate form data in Spring Web MVC Framework?

Spring supports JSR-303 annotation based validations as well as provide Validator interface that we can implement to create our own custom validator. For using JSR-303 based validation, we need to annotate bean variables with the required validations.

For custom validator implementation, we need to configure it in the controller class. For a complete example, please read Spring MVC Form Validation Example.

What is Spring MVC Interceptor and how to use it?

Spring MVC Interceptors are like Servlet Filters and allow us to intercept client request and process it. We can intercept client request at three places
– **preHandle**, **postHandle** and **afterCompletion**.

We can create spring interceptor by implementing HandlerInterceptor interface or by extending abstract class **HandlerInterceptorAdapter**.

We need to configure interceptors in the spring bean configuration file. We can define an interceptor to intercept all the client requests or we can configure it for specific URI mapping too. For a detailed example, please refer Spring MVC Interceptor Example.

What is Spring JdbcTemplate class and how to use it?

Spring Framework provides excellent integration with JDBC API and provides JdbcTemplate utility class that we can use to avoid bolier-plate code from our database operations logic such as Opening/Closing Connection, ResultSet, PreparedStatement etc.

For JdbcTemplate example, please refer Spring JDBC Example.

How to use Tomcat JNDI DataSource in Spring Web Application?

For using servlet container configured JNDI DataSource, we need to configure it in the spring bean configuration file and then inject it to spring beans as dependencies. Then we can use it with **JdbcTemplate** to perform database operations.

Sample configuration would be:

```
<beans:bean id="dbDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
   <beans:property name="jndiName" value="java:comp/env/jdbc/MyLocalDB"/>
</beans:bean>
```

For complete example, please refer Spring Tomcat JNDI Example.

How would you achieve Transaction Management in Spring?

Spring framework provides transaction management support through Declarative Transaction Management as well as programmatic transaction management. Declarative transaction management is most widely used because it's easy to use and works in most of the cases.

We use annotate a method with **@Transactional** annotation for Declarative transaction management. We need to configure transaction manager for the DataSource in the spring bean configuration file.

```
<bean id="transactionManager"
   class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource" />
</bean>
```

What is Spring DAO?

Spring DAO support is provided to work with data access technologies like JDBC, Hibernate in a consistent and easy way. For example we have **JdbcDaoSupport**, **HibernateDaoSupport**, **JdoDaoSupport** and **JpaDaoSupport** for respective technologies.

Spring DAO also provides consistency in exception hierarchy and we don't need to catch specific exceptions.

How to integrate Spring and Hibernate Frameworks?

We can use Spring ORM module to integrate Spring and Hibernate frameworks, if you are using Hibernate 3+ where SessionFactory provides current session, then you should avoid using **HibernateTemplate** or **HibernateDaoSupport** classes and better to use DAO pattern with dependency injection for the integration.

Also, Spring ORM provides support for using Spring declarative transaction management, so you should utilize that rather than going for hibernate boiler-plate code for transaction management.

For better understanding you should go through following tutorials:

- Spring Hibernate Integration Example
- Spring MVC Hibernate Integration Example

What is Spring Security?

Spring security framework focuses on providing both authentication and authorization in java applications. It also takes care of most of the common security vulnerabilities such as CSRF attack.

It's very beneficial and easy to use Spring security in web applications, through the use of annotations such as **@EnableWebSecurity**. You should go through following posts to learn how to use Spring Security framework.

- Spring Security in Servlet Web Application
- Spring MVC and Spring Security Integration Example

How to inject a java.util.Properties into a Spring Bean?

We need to define propertyConfigurer bean that will load the properties from the given property file. Then we can use Spring EL support to inject properties into other bean dependencies. For example;

```
<bean id="propertyConfigurer"
  class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
    <property name="location" value="/WEB-INF/application.properties" />
</bean>
```

```
<bean class="com.journaldev.spring.EmployeeDaoImpl">
   <property name="maxReadResults" value="${results.read.max}"/>
</bean>
```

If you are using annotation to configure the spring bean, then you can inject property like below.

```
@Value("${maxReadResults}")
private int maxReadResults;
```

Name some of the design patterns used in Spring Framework?

Spring Framework is using a lot of design patterns, some of the common ones are:

- Singleton Pattern: Creating beans with default scope.
- Factory Pattern: Bean Factory classes
- Prototype Pattern: Bean scopes
- Adapter Pattern: Spring Web and Spring MVC
- Proxy Pattern: Spring Aspect Oriented Programming support
- Template Method Pattern: JdbcTemplate, HibernateTemplate etc
- Front Controller: Spring MVC DispatcherServlet
- Data Access Object: Spring DAO support
- Dependency Injection and Aspect Oriented Programming

What are some of the best practices for Spring Framework?

Some of the best practices for Spring Framework are:

- Avoid version numbers in schema reference, to make sure we have the latest configs.
- Divide spring bean configurations based on their concerns such as spring-jdbc.xml, spring-security.xml.
- For spring beans that are used in multiple contexts in Spring MVC, create them in the root context and initialize with listener.
- Configure bean dependencies as much as possible, try to avoid autowiring as much as possible.
- For application level properties, best approach is to create a property file and read it in the spring bean configuration file.
- For smaller applications, annotations are useful but for larger applications annotations can become a pain. If we have all the configuration in xml files, maintaining it will be easier.
- Use correct annotations for components for understanding the purpose easily. For services use @Service and for DAO beans use @Repository.
- Spring framework has a lot of modules, use what you need. Remove all the extra dependencies that gets usually added when you create projects through Spring Tool Suite templates.
- If you are using Aspects, make sure to keep the join pint as narrow as possible to avoid advice on unwanted methods. Consider custom annotations that are easier to use and avoid any issues.

- Use dependency injection when there is actual benefit, just for the sake of loose-coupling don’t use it because it’s harder to maintain.