What is JDK, JRE & JVM ?

"Without JRE, all Java programs whether in the form of applet or application will not execute. JRE is essential to run or execute the programs. JRE is essential for JDK, since one cannot test a program without executing it. JVM- stands for Java Virtual Machine. "

JDK (Java Development Kit):

JDK- stands for Java Development Kit. Java Developer Kit contains tools needed to develop the Java programs, and JRE to run the programs. The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc... Compiler converts java code into byte code. Java application launcher opens a JRE, loads the class, and invokes its main method. Java Development Kit (JDK) The JDK is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications. ** Classe loading mechanism handled by ClassLoader.

JRE (Java Runtime Environment):

Java Runtime Environment contains JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc. Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. In addition, two key deployment technologies are part of the JRE: Java Plug-in, which enables applets to run in popular browsers; and Java Web Start, which deploys standalone applications over a network.

JVM (Java Virtual Machine):

JVM- stands for Java Virtual Machine. Every platform has its own specific and dependent JVM. It provide the important feature- platform independence means our Java program is compiled into bytecode which can then be distributed. So, we can say that our Java Bytecode is platform independence due to JVM handle all type of platform dependency. JRE processes the bytecode and sends it to the JVM. JVM then creates the machine code based on the platform and execute the code that machine can understand. This executable is what we can call the running program. JVM comes with JRE and is different for different platforms. JVM is essential and required pre-installation to run Java programs. JVM is not Platform Independent.

The JVM performs following main tasks :

- Load code
- Verify code
- Execute code
- Provide runtime environment

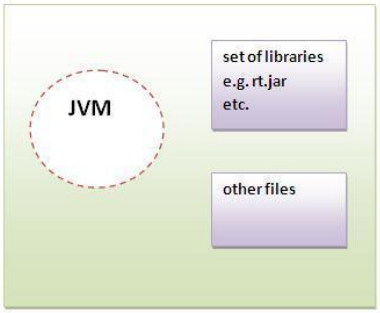
Difference between JDK,JRE and JVM ?

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. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.
- The JVM performs following main tasks:
    - Loads code
    - Verifies code
    - Executes code
    - Provides runtime environment

JRE

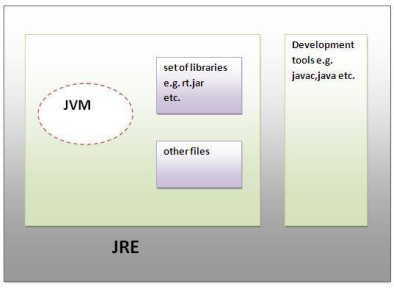
- JRE is an acronym for Java Runtime Environment.It is used to provide runtime environment.It is the implementation of JVM.It physically exists.It contains set of libraries + other files that JVM uses at runtime.
- Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

JDK

- JDK is an acronym for Java Development Kit.It physically exists.It contains JRE + development tools.



JDK

JIT Complier ?

The just-in-time (JIT) compiler is the heart of the Java Virtual Machine. A JIT compiler runs after the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set.

Classloader

- ClassLoader in Java is a class which is used to load class files in Java.
- Java code is compiled into class file by javac compiler and JVM executes Java program, by executing byte codes written in class file.
- ClassLoader is responsible for loading class files from file system, network or any other source.
- There are three default class loader used in Java, Bootstrap , Extension and System or Application class loader.
- Every class loader has a predefined location, from where they loads class files.

Bootstrap ClassLoader

- it is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java.
- Bootstrap class loader don't have any parents, if you call String.class.getClassLoader() it will return null and any code based on that may throw NullPointerException in Java.
- Bootstrap class loader is also known as Primordial ClassLoader in Java.

Extension ClassLoader

- It delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by sun.misc.Launcher$ExtClassLoader.

System or Application class loader

- Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from CLASSPATH environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR.
- Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class.
- Also, except Bootstrap class loader, which is implemented in native language mostly in C, all Java class loaders are implemented using java.lang.ClassLoader.

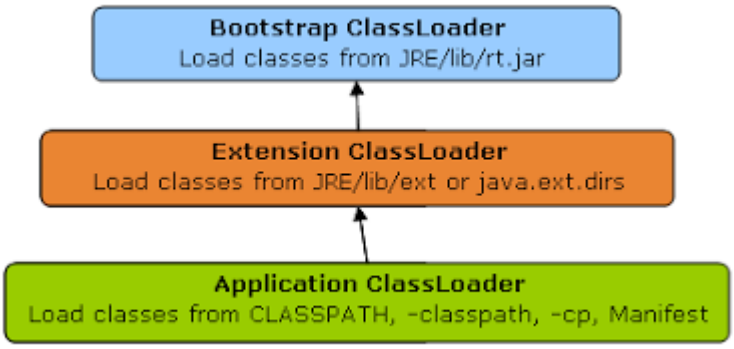
Class Loader in Java BootStrap Extension and Application

In short here is the location from which Bootstrap, Extension and Application ClassLoader load Class files.

1) Bootstrap ClassLoader - JRE/lib/rt.jar

2) Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs

3) Application ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.



How ClassLoader Works in Java

- Java class loaders are used to load classes at runtime.
- ClassLoader in Java works on three principle: **delegation, visibility and uniqueness**.
- Delegation principle forward request of class loading to parent class loader and only loads the class, if parent is not able to find or load class.
- Visibility principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader can not see classes loaded by child.
- Uniqueness principle allows to load a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent.
- Correct understanding of class loader is must to resolve issues like NoClassDefFoundError in Java and java.lang.ClassNotFoundException, which are related to class loading.
- ClassLoader is also an important topic in advanced Java Interviews, where good knowledge of working of Java ClassLoader and How classpath works in Java is expected from Java programmer.

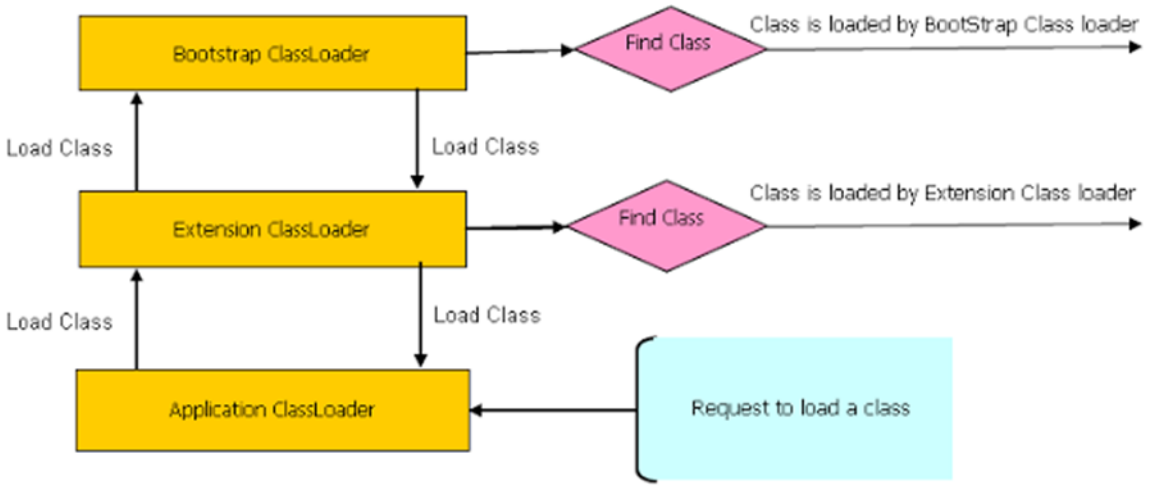
Delegation principles

- As discussed on when a class is loaded and initialized in Java, a class is loaded in Java, when its needed.
- Suppose you have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader.
- Primordial will look for that class in rt.jar and since that class is not there, request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there than Extension class loader will load

that class and Application class loader will never load that class but if its not loaded by extension class-loader than Application class loader loads it from Classpath in Java.

- Remember Classpath is used to load class files while PATH is used to locate executable like javac or java command.

ClassLoader load class in Java using delegation.



Visibility Principle

- According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either java.lang.ClassNotFoundException. as shown in below Example

```
package test;

import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to demonstrate How ClassLoader works in Java,
 * in particular about visibility principle of ClassLoader.
 *
 * @author Javin Paul
 */

public class ClassLoaderTest {

    public static void main(String args[]) {
        try {
            //printing ClassLoader of this class
            System.out.println("ClassLoaderTest.getClass().getClassLoader() : "
                    + ClassLoaderTest.class.getClassLoader());
```

```
            //trying to explicitly load this class again using Extension class loader
            Class.forName("test.ClassLoaderTest", true
                        , ClassLoaderTest.class.getClassLoader().getParent());
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(ClassLoaderTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

```
Output:
ClassLoaderTest.getClass().getClassLoader() :
sun.misc.Launcher$AppClassLoader@601bb1
16/08/2012 2:43:48 AM test.ClassLoaderTest main
SEVERE: null
java.lang.ClassNotFoundException: test.ClassLoaderTest
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
        at sun.misc.Launcher$ExtClassLoader.findClass(Launcher.java:229)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:247)
        at test.ClassLoaderTest.main(ClassLoaderTest.java:29)
```

Uniqueness Principle

- According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again.
- Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial. You should follow all class loader principle while writing your own ClassLoader.

How to load class explicitly in Java

- Java provides API to explicitly load a class by Class.forName(classname) and Class.forName(classname, initialized, classloader), remember JDBC code which is used to load JDBC drives we have seen in Java program to Connect Oracle database.
- As shown in above example you can pass name of ClassLoader which should be used to load that particular class along with binary name of class.
- Class is loaded by calling loadClass() method of java.lang.ClassLoader class which calls findClass() method to locate bytecodes for corresponding class.
- In this example Extension ClassLoader uses java.net.URLClassLoader which search for class files and resources in JAR and directories.
- any search path which is ended using "/" is considered directory. If findClass() does not found the class than it throws java.lang.ClassNotFoundException and if it finds it calls defineClass() to convert bytecodes into a .class instance which is returned to the caller.

Where to use ClassLoader in Java

- ClassLoader in Java is a powerful concept and used at many places.
- One of the popular example of ClassLoader is AppletClassLoader which is used to load class by Applet, since Applets are mostly loaded from internet rather than local file system, By using separate ClassLoader you can also loads same class from multiple sources and they will be treated as different class in JVM. J2EE uses multiple class loaders to load class from different location like classes from WAR file will be loaded by Web-app ClassLoader while classes bundled in EJB-JAR is loaded by another class loader.
- Some web server also supports hot deploy functionality which is implemented using ClassLoader. You can also use ClassLoader to load classes from database or any other persistent store.

Bytecode Verification

When a class loader presents the bytecodes of a newly loaded Java platform class to the virtual machine, these bytecodes are first inspected by a verifier. The verifier checks that the instructions cannot perform actions that are obviously damaging. All classes except for system classes are verified. You can, however, deactivate verification with the undocumented -noverify option.

For example,
java -noverify SampleClass

SampleClass is Class for verification

Program Counter Register :

PC (program counter) registers. It contains the address of the Java virtual machine instruction currently being executed.

Native Method Stack :

It contains all the native methods used in the application.

Execution Engine :

It contains:

1). A virtual processor

2). Interpreter:Read bytecode stream then execute the instructions.

3). Just-In-Time(JIT) compiler:It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.Here the term - compiler refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

Memory areas in JVM.

1. Stack Memory
2. Heap Memory
3. Non-heap Memory
4. Method Area
5. Memory Pool
6. Runtime Constant Pool
7. Memory Generations

Stack Memory :

- Java Stack memory frames. Stack frames holds local variables and partial results, and plays a part in method invocation and return.

Heap Memory :

- It is the runtime data area in which objects are allocated.

- The JVM has a heap that is the runtime objects area from which memory for all class instances and arrays are allocated. It is created at the JVM start-up.
- The heap size may be configured with the following VM options:
- -Xmx<size> - to set the maximum Java heap size
- -Xms<size> - to set the initial Java heap size
- By default, the maximum heap size is 64 Mb.

Non-heap Memory :

- At runtime objects are stored in heap memory. Non-Heap Memory, which is used by Java to store loaded classes and other meta-data.
- Non-heap memory is all the memory the JVM allocated for purposes other than the heap.
- Non-heap memory includes:
    - stores per-class structures.
    - code for methods and constructors.
    - runtime constant pool, field and method data the call stacks.
    - memory allocated by native code (e.g. for off-heap caching);
    - memory used by the JIT compiler (compiled native code).
    - in HotSpot 8, the Meta-space (replacement for the Permanent Generation);

Method Areas

- The method area is also known as the permanent generation space (PermGen). All class data are loaded into this memory space. This includes the field and method data and the code for the methods and constructors.

- Java virtual machine must store the following kinds of information in the method area:
- The fully qualified name of the type's direct superclass (unless the type is an interface or class java.lang.Object, neither of which have a superclass) Whether or not the type is a class or an interface The type's modifiers (some subset of` public,abstract, final) An ordered list of the fully qualified names of any direct super-interfaces.

Memory Pool

- Memory pools are created by JVM memory managers during runtime. Memory pool may belong to either heap or non-heap memory.

The Constant Pool

- For each type it loads, a Java virtual machine must store a constant pool. A constant pool is an ordered set of constants used by the type, including literals (string, integer, and floating point constants) and symbolic references to types, fields, and methods. Entries in the constant pool are referenced by index, much like the elements of an array. Because it holds symbolic references to all types, fields, and methods used by a type, the constant pool plays a central role in the dynamic linking of Java programs.

Runtime Constant Pool

- A run time constant pool is a per-class or per-interface run time representation of the constant_pool table in a class file. Each runtime constant pool is allocated from the Java virtual machine's method area.

Memory Generations :

- HotSpot VM's garbage collector uses generational garbage collection. It separates the JVM's memory into and they are called young generation and old generation.

Young Generation :

- Young generation memory consists of two parts, Eden space and survivor space. Shortlived objects will be available in Eden space.
- Every object starts its life from Eden space. When GC happens, if an object is still alive and it will be moved to survivor space and other dereferenced objects will be removed.

Old Generation – Tenured and PermGen :

- Old generation memory has two parts, tenured generation and permanent generation (PermGen). PermGen is a popular term. We used to error like PermGen space not sufficient. GC moves live objects from survivor space to tenured generation. The permanent generation contains meta data of the virtual machine, class and method objects.

How Memory Manegment handled in Java

- When we use a literal value or create a variable or constant and assign it a value, the value is stored in the memory of the computer.

How Local Variables and Instance Variables Stored in memory

- The following figure illustrates that local variables are stored separately (on the stack) from the attribute variables on the heap.

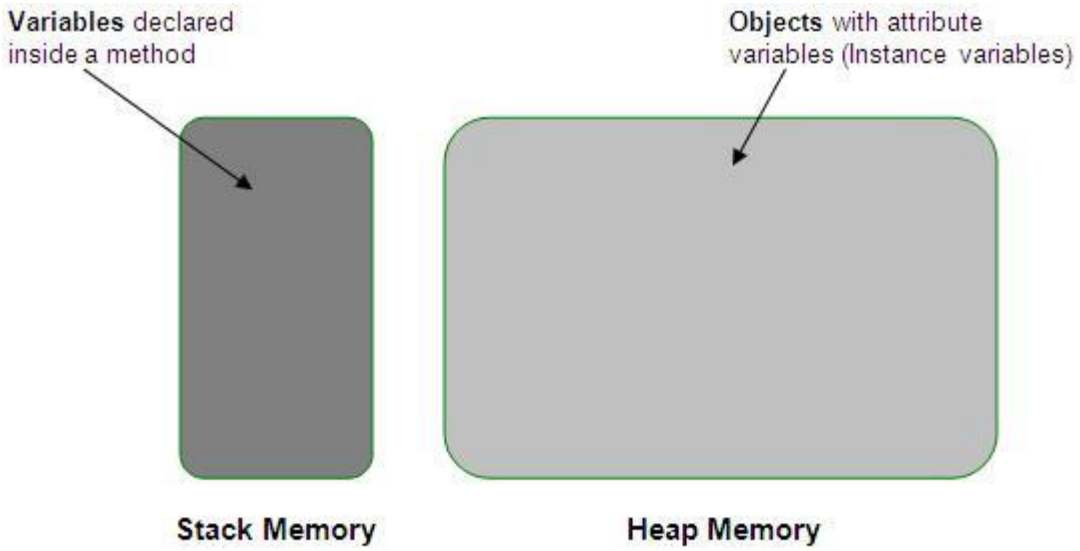


Figure - Java Memory Management

Where local Variables stored in memory:

- Local Variables (that are defined in method or block) are usually stored in stack memory. Stack Memory stores items that are only used for a brief period of time (shorter than the life of an object), such as variables declared inside of a method. Operating System is responsible of destroying local variables (when lifetime of local variables are complete) in Stack Memory.

Where Instance Variables stored in memory:

- Objects and their attributes variables and methods are usually stored in Heap Memory. Heap memory is dynamically allocated memory chunks containing information used to hold objects (including their attribute variables and methods) while they are needed by your program. Garbage Collector is responsible of destroying Objects and their attributes variables (when lifetime of instance variables are complete) in Heap Memory.

What are the Object Reference Variable :

- Object Reference Variables are variables containing an address to an object in heap memory. Declaring and initializing a reference variable is very similar to declaring and initializing a primitive type variable. The only difference is that you must create an object instance (from a class) for the reference variable to point to before you can initialize the object reference.
- To declare, an instantiate, and initialize an object reference variable, we must follow the given instructions:
    - Declare a reference to the object by specifying its identifier and the type of object that the reference points to (the class of the object).
    - Create the object instance using the new keyword.
    - Initialize the object reference variable by assigning the object to the object reference variable.

We will understand all concepts with the help of following example:

```
//Example of Shirt Class
public class Shirt
{
 int shirtId;
 float price;
 char colorCode;

 public void getShirtInformation(int id,float p,char cCode)
 {
  shirtId=id;
  price=p;
  colorCode=cCode;
 }

 public void displayShirtInformation()
 {
  System.out.println("Shirt id :"+shirtId);
  System.out.println("Shirt price:"+price);
  System.out.println("Shirt ColorCode:"+colorCode);
 }
```

```
 public static void main(String ar[])
 {
  //Declare an Object Reference Variable
  Shirt myShirt;

 //Instantiating an object & initializing the object ref. variable
  myShirt=new Shirt();


//We can perform above steps in one line by using :
 //Shirt myShirt=new Shirt();

  myShirt.getShirtInformation(11,799.0F,'U');

  myShirt.displayShirtInformation();
 }
}
```

Example Explanation :

How to Declare Object Reference Variable

- To declare an object reference variable, state the class that you want to create object from, then select the name you want to use to refer to the object.
- Syntax for Declaring the Object Reference Variable
    - Classname identifier;
- Here identifier is an Object Reference Variable.
- The Classname is the class or type of the object referred to with the object reference.
- The identifier is the name you assigned to the variable of type Classname.
- In above example :
    - Shirt myShirt;
- Shirt class creates an object reference variable called myShirt. By Default, object references that are member variables are initialized to null. Because the variable myShirt is a local variable declared in Example of Shirt Class, it is not initialized.

Instantiating an Object

- After we declare the object reference, we can create the object that you refer to.

Syntax for instantiating an object is:

```
new Classname();
//The new keyword creates an object instance of class.
//The Classname is the class or type of object being created
Creating Object Reference Variable Using One Line of Code
Shirt myShirt=new Shirt();
Creating Object Reference Variable Using Two Lines of Code
```

```
Shirt myShirt;
myShirt=new Shirt();
```

Storing Object Reference Variables in Memory

- While primitive variables hold values, object reference variables hold the location (memory address) of objects in memory.
- The following figure shows how the primitive variables and object reference variables are stored differently.

```
public static void main (String args[]) {
int counter;
counter = 10;
Shirt myShirt = new Shirt();
Shirt yourShirt=new Shirt();
}
```

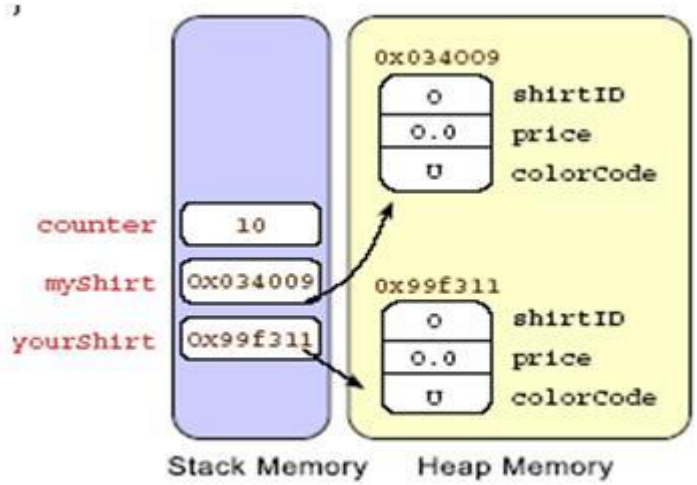


Figure - How Reference Variables are Stored in Memory

Assigning a Reference from One Variable to Another

- The address in reference variable yourShirt, such as 0x99f311, is assigned to the variable myShirt. Both variables now point to the same object, even though the other object, the one that the variable myShirt once pointed to, still exists.
- Unless another reference variable was pointing to the second Shirt object, the object is garbage collected.

```
Shirt myShirt = new Shirt( );
Shirt yourShirt = new Shirt( );
myShirt = yourShirt;
```

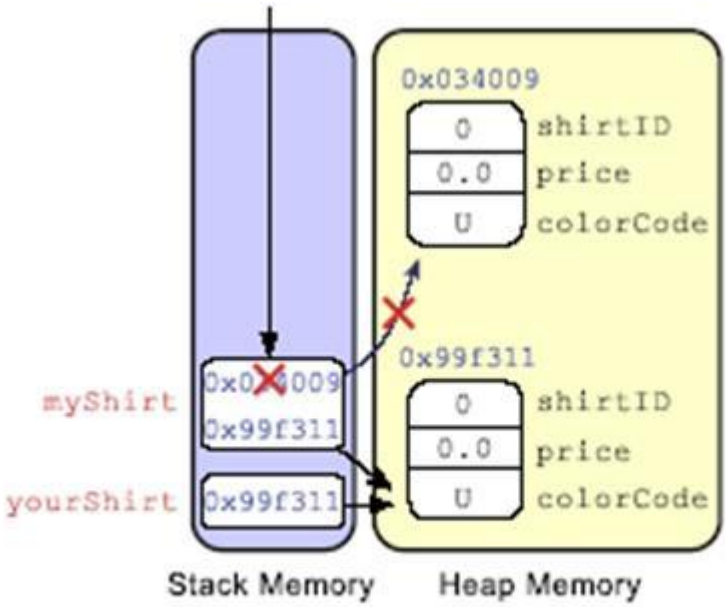


Figure - Assigning an Object Reference from One Variable to Another

- The address in reference variable yourShirt, such as 0x99f311, is assigned to the variable myShirt. Both variables now point to the same object, even though the other object, the one that the variable myShirt once pointed to, still exists.
- Unless another reference variable was pointing to the second Shirt object, the object is garbage collected.