**Reference types in JAVA**

**What is reference type?**

A reference type is a data type that’s based on a class rather than on one of the primitive types that are built in to the Java language. The class can be a class that’s provided as part of the Java API class library or a class that you write yourself.

Either way, when you create an object from a class, Java allocates the amount of memory the object requires to store the object. Then, if you assign the object to a variable, the variable is actually assigned a reference to the object, not the object itself. This reference is the address of the memory location where the object is stored.

To declare a variable using a reference type, you simply list the class name as the data type. For example, the following statement defines a variable that can reference objects created from a class named Ball:

**Ball b;**

To create a new instance of an object from a class, you use the new keyword along with the class name:

**Ball b = new Ball();**

One of the key concepts in working with reference types is the fact that a variable of a particular type doesn’t actually contain an object of that type. Instead, it contains a reference to an object of the correct type. An important side effect is that two variables can refer to the same object.

Consider these statements:

**Ball b1 = new Ball();**

**Ball b2 = b1;**

Here, both b1 and b2 refer to the same instance of the Ball class.

**TYPES OF REFERENCE TYPES:-**

Reference types are mainly divided into following four types. Depending upon how objects are garbage collected, references to those objects in java are grouped into 4 types. They are,

1) Strong References

2) Soft References

3) Weak References

4) Phantom References

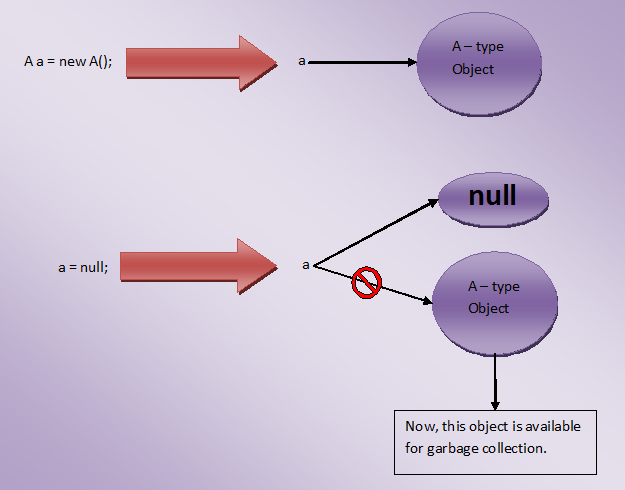
**1) Strong References**

These types of references we use daily while writing the code. Any object in the memory which has active **strong reference** is not eligible for garbage collection. For example, in the below program, reference variable **‘a’** is a strong reference which is pointing to class A-type object. At this point of time, this object can’t be garbage collected as it has strong reference.

|  |  |
| --- | --- |
| 1  3  4  5  6  7  8  9  10  11  12  13  14 | class A  {  //Class A  }  public class MainClass  {  public static void main(String[] args)  {  A a = new A();      //Strong Reference  a = null;    //Now, object to which 'a' is pointing earlier is eligible for garbage collection.  }  } |

If you make reference **‘a’** to point to null like in Line 12, then, object to which ‘a’ is pointing earlier will become eligible for garbage collection. Because, it will have no active references pointing to it. This object is most likely to be garbage collected when garbage collector decides to run.

Look at the below picture for more precise understanding.



2) Soft References

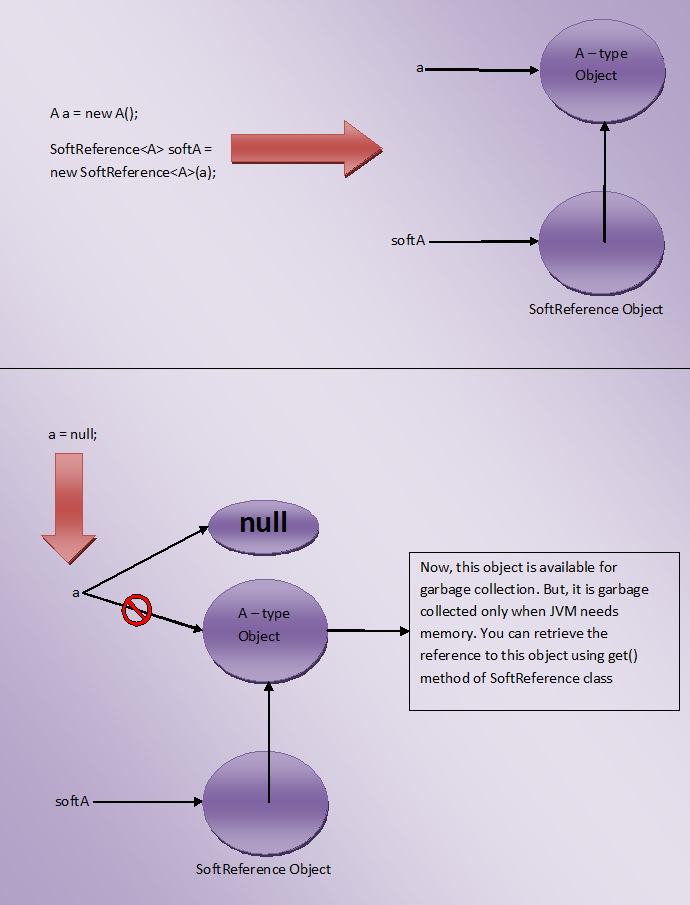
The objects which are softly referenced will not be garbage collected (even though they are available for garbage collection) until JVM badly needs memory. These objects will be cleared from the memory only if JVM runs out of memory. You can create a soft reference to an existing object by using  **java.lang.ref.SoftReference** class. Below is the code example on how to create a soft reference.

|  |  |
| --- | --- |
|  | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating Soft Reference to A-type object to which 'a' is also pointing            SoftReference<A> softA = new SoftReference<A>(a);            a = null;    //Now, A-type object to which 'a' is pointing earlier is eligible for garbage collection. But, it will be  garbage collected only when JVM needs memory.            a = softA.get();    //You can retrieve back the object which has been softly referenced      }  } |
|  |  |

**2. SOFT REFERENCE**

In the above example, you create two strong references – ‘**a**‘ and ‘**softA**‘. ‘a’ is pointing to A-type object and ‘softA’ is pointing to SoftReference type object. This SoftReference type object is internally referring to A-type object to which ‘a’ is also pointing. When ‘a’ is made to point to null, object to which ‘a’ is pointing earlier becomes eligible for garbage collection. But, it will be garbage collected only when JVM needs memory. Because, it is softly referenced by ‘softA’ object.

Look at the below picture for more clarity.



One more use of SoftReference class is that you can retrieve back the object which has been softly referenced. It will be done by using **get()**method. This method returns reference to the object if object is not cleared from the memory. If object is cleared from the memory, it will return null.

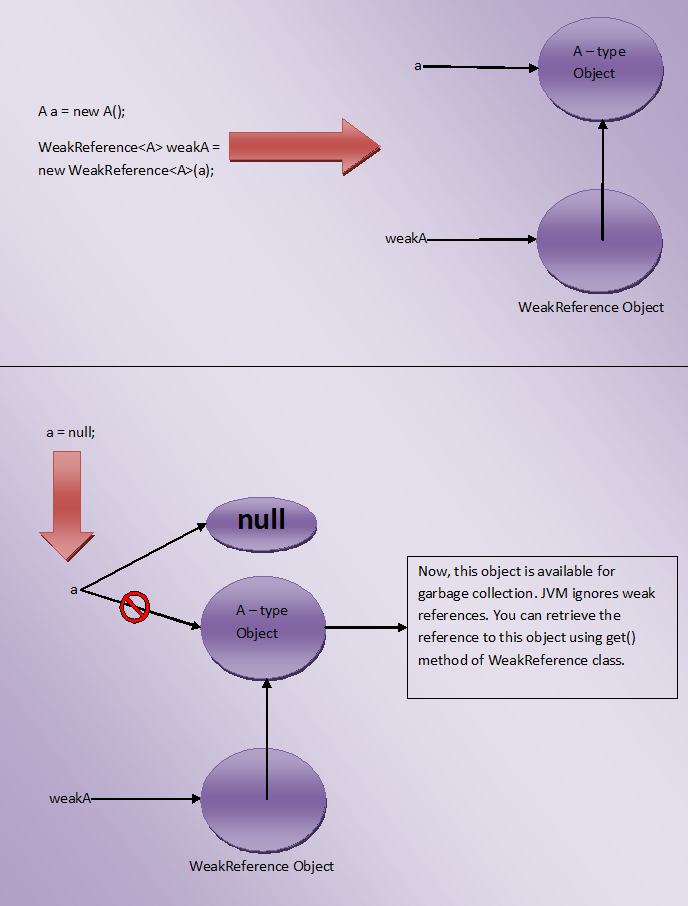
3) Weak References

JVM ignores the **weak references**. That means objects which has only week references are eligible for garbage collection. They are likely to be garbage collected when JVM runs garbage collector thread. JVM doesn’t show any regard for weak references.

Below is the code which shows how to create weak references.

|  |  |
| --- | --- |
|  | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating Weak Reference to A-type object to which 'a' is also pointing.            WeakReference<A> weakA = new WeakReference<A>(a);            a = null;    //Now, A-type object to which 'a' is pointing earlier is available for garbage collection.            a = weakA.get();    //You can retrieve back the object which has been weakly referenced.      }  } |

Look at the below picture for more clear understanding.



You may think that what is the use of creating weak references if they are ignored by the JVM, Use of weak reference is that you can retrieve back the weakly referenced object if it is not yet removed from the memory. This is done using get() method of WeakReference class. It will return reference to the object if object is not yet removed from the memory.

4) Phantom References

The objects which are being referenced by **phantom references** are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called **‘reference queue’**. They are put in a reference queue after calling finalize() method on them. You can’t retrieve back the objects which are being phantom referenced. That means calling get() method on phantom reference always returns null.

class A

{

    //A Class

}

public class MainClass

{

    public static void main(String[] args)

    {

        A a = new A();      //Strong Reference

        //Creating ReferenceQueue

        ReferenceQueue<A> refQueue = new ReferenceQueue<A>();

        //Creating Phantom Reference to A-type object to which 'a' is also pointing

        PhantomReference<A> phantomA = new PhantomReference<A>(a, refQueue);

        a = null;    //Now, A-type object to which 'a' is pointing earlier is available for garbage collection. But, this object is kept in 'refQueue' before removing it from the memory.

        a = phantomA.get();    //it always returns null

    }

}