9 Things about Null in Java

Java and null are uniquely bonded. There is hardly a Java programmer, who is not troubled by null pointer exception, it is the most infamous fact about. Even inventor of null concept has called it his billion dollar mistake, then why Java kept it? Null was there from long time and I believe Java designer knows that null creates more problem than it solves, but still they went with it. It surprise me even more because Java's design philosophy was to simplify things, that's why they didn't bothered with pointers, operator overloading and multiple inheritance of implementation, they why null. Well I really don't know the answer of that question, what I know is that, doesn't matter how much null is criticized by Java developers and open source community, we have to live with that. Instead of ruing about null it's better to learn more about it and make sure we use it correct. Why you should learn about null in Java? because If you don't pay attention to null, Java will make sure that you will suffer from dreaded`java.lang.NullPointerException` and you will learn your lesson hard way. Robust programming is an art and your team, customer and user will appreciate that. In my experience, one of the main reasons of NullPointerException are not enough knowledge about null in Java. Many of you already familiar with null but for those, who are not, can learn some old and new things about null keyword. Let's revisit or learn some important things about null in Java.

What is Null in Java

As I said, null is very very important concept in Java. It was originally invented to denote absence of something e.g. absence of user, a resource or anything, but over the year it has troubled Java programmer a lot with nasty null pointer exception. In this tutorial, we will learn basic facts about null keyword in Java and explore some techniques to minimize null checks and how to avoid nasty null pointer exceptions.

1) First thing, first, `null` is a keyword in Java, much like `public`, `static` or `final`. It's case sensitive, you cannot write null as `Null` or `NULL`, compiler will not recognize them and give error.

```
Object obj = NULL; // Not Ok
Object obj1 = null  //Ok
```

Programmer's which are coming from other language have this problem, but use of modern day IDE's has made it insignificant. Now days, IDE like Eclipse or Netbeans can correct this mistake, while you

type code, but in the era of notepad, Vim and Emacs, this was a common issue which could easily eat your precious time.

2) Just like every primitive has default value e.g. `int` has 0, `boolean` has `false`, `null`is the default value of any reference type, loosely spoken to all object as well. Just like if you create a `boolean` variable, it got default value as `false`, any reference variable in Java has default value `null`. This is true for all kind of variables e.g. member variable or local variable, instance variable or static variable, except that compiler will warn you if you use a local variable without initializing them. In order to verify this fact, you can see value of reference variable by creating a variable and them printing it's value, as shown in following code snippet :


```
private static Object myObj;
public static void main(String args[]){
    System.out.println("What is value of myObjc : " + myObj);
}
```

```
What is value of myObjc : null
```

This is true for both static and non-static object, as you can see here that I made `myObj` a static reference so that I can use it directly inside main method, which is static method and doesn't allow non-static variable inside.

3) 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, as shown below :

```
String str = null; // null can be assigned to String
Integer itr = null; // you can assign null to Integer also
Double dbl = null;  // null can also be assigned to Double

String myStr = (String) null; // null can be type cast to String
Integer myItr = (Integer) null; // it can also be type casted to Integer
Double myDbl = (Double) null; // yes it's possible, no error
```

You can see type casting null to any reference type is fine at both compile time and runtime, unlike many of you might have thought, it will also not throw`NullPointerException` at runtime.

4) null can only be assigned to reference type, you cannot assign null to primitive variables e.g. `int`, `double`, `float` or `boolean`. Compiler will complain if you do so, as shown below.

```
int i = null; // type mismatch : cannot convert from null to int
short s = null; //  type mismatch : cannot convert from null to short
```

```
byte b = null: // type mismatch : cannot convert from null to byte
double d = null; //type mismatch : cannot convert from null to double

Integer itr = null; // this is ok
int j = itr; // this is also ok, but NullPointerException at runtime
```

As you can see, when you directly assign null to primitive error it's compile time error, but if you assign null to a wrapper class object and then assign that object to respective primitive type, compiler doesn't complain, but you would be greeted by null pointer exception at runtime. This happens because of autoboxing in Java, and we will see it in next point.

5) Any wrapper class with value null will throw `java.lang.NullPointerException`when Java unbox them into primitive values. Some programmer makes wrong assumption that, auto boxing will take care of converting null into default values for respective primitive type e.g. 0 for `int`, false for `boolean` etc, but that's not true, as seen below.

```
Integer iAmNull = null;
int i = iAmNull; // Remember - No Compilation Error
```

but when you run above code snippet you will see Exception in thread "main"`java.lang.NullPointerException` in your console. This happens a lot while working with HashMap and Integer key values. Code like shown below will break as soon as you run.

```
import java.util.HashMap;
import java.util.Map;


/**
 * An example of Autoboxing and NullPointerExcpetion
 *
 * @author WINDOWS 8
 */
public class Test {

    public static void main(String args[]) throws InterruptedException {

      Map numberAndCount = new HashMap<>();

      int[] numbers = {3, 5, 7,9, 11, 13, 17, 19, 2, 3, 5, 33, 12, 5};

      for(int i : numbers){
         int count = numberAndCount.get(i);
         numberAndCount.put(i, count++); // NullPointerException here
      }
    }

}

Output:
Exception in thread "main" java.lang.NullPointerException
 at Test.main(Test.java:25)
```

This code looks very simple and innocuous. All you are doing is finding how many times a number has appeared in a array, classic technique to find duplicates in Java array. Developer is getting the previous count, increasing it by one and putting it back into Map. He might have thought that auto-boxing will take care of converting Integer to int , as it doing while calling put method, but he forget that when there is no count exist for a number, get() method of HashMap will return null, not zero because default value of Integer is null not 0, and auto boxing will throw null pointer exception while trying to convert it into an int variable. Imagine if this code is inside an if loop and doesn't run in QA environment but as soon as you put into production, BOOM :-)

6)`instanceof` operator will return false if used against any reference variable with `null`value or `null` literal itself, e.g.

```
Integer iAmNull = null;
if(iAmNull instanceof Integer){
   System.out.println("iAmNull is instance of Integer");

}else{
   System.out.println("iAmNull is NOT an instance of Integer");
}
```

Output : iAmNull is NOT an instance of Integer

This is an important property of `instanceof` operation which makes it useful for type casting checks.

7) You may know that you cannot call a non-static method on a reference variable with`null` value, it will throw `NullPointerException`, but you might not know that, you*can* call static method with reference variables with `null` values. Since static methods are bonded using static binding, they won't throw NPE. Here is an example :

```
public class Testing {
   public static void main(String args[]){
      Testing myObject = null;
      myObject.iAmStaticMethod();
      myObject.iAmNonStaticMethod();
   }

   private static void iAmStaticMethod(){
        System.out.println("I am static method, can be called by null reference");
   }

   private void iAmNonStaticMethod(){
 System.out.println("I am NON static method, don't date to call me by null");
   }

}
```

```
Output:
I am static method, can be called by null reference
Exception in thread "main" java.lang.NullPointerException
                at Testing.main(Testing.java:11)
```

8) You can pass `null` to methods, which accepts any reference type e.g. `public void print(Object obj)` can be called as `print(null)`. This is Ok from compiler's point of view, but behavior is entirely depends upon method. Null safe method, doesn't throw `NullPointerException` in such case, they just exit gracefully. It is recommended to write null safe method if business logic allows.

9) You can compare null value using `==` (equal to) operator and `!=` (not equal to) operator, but cannot use it with other arithmetic or logical operator e.g. less than or greater than. Unlike in SQL, in Java `null == null` will return true, as shown below :

```
public class Test {

    public static void main(String args[]) throws InterruptedException {

       String abc = null;
       String cde = null;

       if(abc == cde){
           System.out.println("null == null is true in Java");
       }

       if(null != null){
           System.out.println("null != null is false in Java");
       }

       // classical null check
       if(abc == null){
           // do something
       }

       // not ok, compile time error
       if(abc > null){

       }
    }
}
```

```
Output:
null == null is true in Java
```

That's all about null in Java. By some experience in Java coding and by using simple tricks to avoid NullPointerExcpetion, you can make your code null safe. Since null can be treated as empty or uninitialized value it's often source of confusion, that's why it's more important to document behavior of a method for null input. Always remember, null is default value of any reference variable and you cannot call any instance method, or access an instance variable using null reference in Java.