Java Tips and Best practices to avoid NullPointerException in Java Applications

A NullPointerException in Java application is best way to solve it and that is also key to write robust programs which can work smoothly. As it said "prevention is better than cure", same is true with nasty NullPointerException. Thankfully by applying somedefensive coding techniques and following contract between multiple part of application, you can avoid NullPointerException in Java to a good extent. By the way this is the second post on NullPointerException in Javarevisited, In last post we have discussed about common cause of NullPointerException in Javaand in this tutorial, we will learn some Java coding techniques andbest practices, which can be used to avoid NullPointerException in Java. Following these Java tips also minimize number of `!=null` check, which litter lot of Java code. As an experience Java programmer, you may be aware of some of these techniques and already following it in your project, but for freshers and intermediate developers, this can be good learning. By the way, if you know any other Java tips to avoid NullPointerException and reduce null checks in Java, then please share with us.

Java Tips and Best practices to avoid NullPointerException

These are simple techniques, which is very easy to follow, but has significant impact on code quality and robustness. In my experience, just first tip is resulted in significant improvement in code quality. As I said earlier, if you know any other Java tips or best practice, which can help to reduce null check, then you can share with us by commenting on this article.

1) Call `equals()` and `equalsIgnoreCase()` method on known String literal rather unknown object

Always call `equals()` method on known String which is not null. Since `equals()` method is symmetric, calling a.equals(b) is same as calling `b.equals(a)`, and that's why many programmer don't pay attention on object a and b. One side effect of this call can result in NullPointerException, if caller is null.

```
Object unknownObject = null;

//wrong way - may cause NullPointerException
if(unknownObject.equals("knownObject")){
   System.err.println("This may result in NullPointerException if unknownObject
is null");
}

//right way - avoid NullPointerException even if unknownObject is null
if("knownObject".equals(unknownObject)){
    System.err.println("better coding avoided NullPointerException");
}
```

This is the most easy Java tip or best practice to avoid NullPointerException, but results in tremendous improvement, because of `equals()` being a common method.

2) Prefer `valueOf()` over `toString()` where both return same result
Since calling toString() on null object throws `NullPointerException`, if we can get same value by calling valueOf() then prefer that, as passing null to `valueOf()` returns "`null`", specially in case of wrapper classes like `Integer`, `Float`,`Double` or `BigDecimal`.

```
BigDecimal bd = getPrice();
System.out.println(String.valueOf(bd)); //doesn’t throw NPE
System.out.println(bd.toString()); //throws "Exception in thread "main"
java.lang.NullPointerException"
```

Follow this Java tips, if you are unsure about object being null or not.

3) Using null safe methods and libraries
There are lot of open source library out there, which does the heavy lifting of checking null for you. One of the most common one is `StringUtils` from Apache commons. You can use `StringUtils.isBlank()`,`isNumeric()`,`isWhiteSpace()` and other utility methods without worrying of `NullPointerException`.

```
//StringUtils methods are null safe, they don't throw NullPointerException
System.out.println(StringUtils.isEmpty(null));
System.out.println(StringUtils.isBlank(null));
System.out.println(StringUtils.isNumeric(null));
System.out.println(StringUtils.isAllUpperCase(null));

Output:
true
true
false
false
```

But before reaching to any conclusion don't forget to read the documentation of Null safe methods and classes. This is another Java best practices, which doesn't require much effort, but result in great improvements.

4) Avoid returning null from method, instead return empty collection or empty array.
This Java best practice or tips is also mentioned by Joshua Bloch in his book Effective Java which is another good

source of better programming in Java. By returning empty collection or empty array you make sure that basic calls like `size()`, `length()` doesn't fail with `NullPointerException`. Collections class provides convenient empty `List`, `Set` and `Map` as`Collections.EMPTY_LIST`, `Collections.EMPTY_SET` and `Collections.EMPTY_MAP` which can be used accordingly. Here is code example

```
public List getOrders(Customer customer){
   List result = Collections.EMPTY_LIST;
   return result;
}
```

Similarly you can use `Collections.EMPTY_SET` and `Collections.EMPTY_MAP` instead of returning null.

5) Use of annotation `@NotNull` and `@Nullable`

While writing method you can define contracts about nullability, by declaring whether a method is null safe or not, by using annotations like `@NotNull` and `@Nullable`. Modern days compiler, IDE or tool can read this annotation and assist you to put a missing null check, or may inform you about an unnecessary null check, which is cluttering your code. `IntelliJ IDE` and `findbugs` already supports such annotation. These annotations are also part of JSR 305, but even in the absence of any tool or IDE support, this annotation itself work as documentation. By looking `@NotNull` and `@Nullable`, programmer can himself decide whether to check for null or not. By the way ,this is relatively new best practice for Java programmers and it will take some time to get adopted.

6) Avoid unnecessary autoboxing and unboxing in your code

Despite of other disadvantages like creating temporary object, autoboxing are also prone to `NullPointerException`, if the wrapper class object is null. For example, following code will fail with `NullPointerException` if person doesn't have phone number and instead return `null`.

```
Person ram = new Person("ram");
int phone = ram.getPhone();
```

Not just equality but < , > can also throw `NullPointerException` if used along autoboxing and unboxing. See this article to learn more pitfalls of autoboxing and unboxing in Java.

7) Follow Contract and define reasonable default value

One of the best way to *avoid NullPointerException in Java* is as simple as defining contracts and following them. Most of the `NullPointerException` occurs because Object is created with incomplete information or all required dependency is not provided. If you don't allow to create incomplete object and gracefully deny any such request you can prevent lots of`NullPointerException` down the road. Similarly if Object is allowed to be created, than you should work with reasonable default value. for example an `Employee` object can not be created without `id` and `name`, but can have an optional phone number. Now if `Employee` doesn't have phone number than instead of returning null, return default value e.g. zero, but that choice has to be carefully taken sometime checking for null is easy rather than calling an invalid number. One same note, by defining what can be null and what can not be null, caller can make an informed decision. Choice offailing fast or accepting null is also an important design decision you need to take and adhere consistently.

8) If you are using database for storing your domain object such as `Customers`, `Orders` etc than you should define your null-ability constraints on database itself. Since database can acquire data from multiple sources, having null-ability check in DB will ensure data integrity. Maintaining null constraints on database will also help in reducing *null check in Java code*. While loading objects from database you will be sure, which field can be null and which field is not null, this will minimize unnecessary `!= null` check in code.

9) Use Null Object Pattern
This is another way of avoiding `NullPointerExcpetion` in Java. If a method returns an object, on which caller, perform some operations e.g. `Collection.iterator()` method returns Iterator, on which caller performs traversal. Suppose if a caller doesn't have any Iterator, it can return Null object instead of null. Null object is a special object, which has different meaning in different context, for example, here an empty Iterator, calling `hasNext()` on which returns false, can be a null object. Similarly in case of method, which returns `Container` or `Collection` types, empty object should be used instead of returning null. I am planning to write a separate article on *Null Object pattern*, where I will share few more examples of NULL objects in Java.

That's all guys, these are couple of easy to follow Java tips and best practices to avoid NullPointerException. You would appreciate, how useful these tips can be, without too much of effort. If you are using any other tip to avoid this exception, which is not included in this list, than please share with us via comment, and I will include them here.