

Simple Java

Contents

I	Preface	7
II	Java Questions	11
1	How to Check if an Array Contains a Value in Java Efficiently?	13
2	Top 10 Questions about Java Exceptions	16
3	Why Field Can't Be Overridden?	18
4	Constructors of Sub and Super Classes in Java?	20
5	Java Enum Examples	23
6	Java Access Level for Members: public, protected, private	24
7	The Interface and Class Hierarchy Diagram of Java Collections	25
8	Top 9 questions about Java Maps	28
9	Java equals() and hashCode() Contract	33
10	What does a Java array look like in memory?	35
11	The Introduction of Java Memory Leaks	38
12	Frequently Used Methods of Java HashMap	40
13	How Java Compiler Generate Code for Overloaded and Overridden Methods?	42
14	String is passed by "reference" in Java	43
15	FileOutputStream vs. FileWriter	46
16	HashSet vs. TreeSet vs. LinkedHashSet	46
17	How to Write a File Line by Line in Java?	52

18	What can we learn from Java HelloWorld?	54
19	Top 10 questions of Java Strings	58
20	How does Java handle aliasing?	61
21	How Static Type Checking Works in Java?	63
22	Interview Question - Use Java Thread to Do Math Calculation	64
23	Why String is immutable in Java ?	65
24	ArrayList vs. LinkedList vs. Vector	67
25	Java Varargs Examples	72
26	HashMap vs. TreeMap vs. Hashtable vs. LinkedHashMap	73
27	What is Instance Initializer in Java?	80
28	Top 10 Methods for Java Arrays	81
29	Java Type Erasure Mechanism	84
30	Simple Java - Foreword	86
31	Top 10 Mistakes Java Developers Make	86
32	How to make a method thread-safe in Java?	92
33	What Is Inner Interface in Java?	94
34	Top 10 questions about Java Collections	96
35	Set vs. Set<?>	101
36	How to Convert Array to ArrayList in Java?	103
37	Java read a file line by line - How Many Ways?	105
38	Yet Another "Java Passes By Reference or By Value"?	107
39	Monitors - The Basic Idea of Java Synchronization	109
40	The substring() Method in JDK 6 and JDK 7	111
41	2 Examples to Show How Java Exception Handling Works	114
42	Java Thread: an overriding example code	116

43 Comparable vs. Comparator in Java	117
44 Overriding vs. Overloading in Java	121
45 Why do we need Generic Types in Java?	124
46 Deep Understanding of Arrays.sort()	126
47 How to use java properties file?	130
48 Efficient Counter in Java	131
49 How Developers Sort in Java?	137
50 The Most Widely Used Java Libraries	139
51 Inheritance vs. Composition in Java	142
52 A simple TreeSet example	148
53 JVM Run-Time Data Areas	150
54 Diagram to show Java String's Immutability	151
55 Create Java String Using " " or Constructor?	153
56 What exactly is null in Java?	155
57 Diagram of Exception Hierarchy	156
58 java.util.ConcurrentModificationException	158
59 When to use private constructors in Java?	160
60 Top 10 Questions for Java Regular Expression	160
61 "Simple Java" PDF Download	165
62 Start from length & length() in Java	165
63 When and how a Java class is loaded and initialized?	167
64 Java Thread: notify() and wait() examples	170
65 Should .close() be put in finally block or not?	173
66 Java Serialization	175
67 Iteration vs. Recursion in Java	177

Part I.

Freface

The creation of Program Creek is inspired by the saying that "Every developer should have a blog." The word "creek" is picked because of the beautiful scenes of Arkansas which is a central state of America where I studied and worked 3 years. The blog has been used as my notes to track what I have done and my learning experience. Unexpectedly, millions of people have visited Program Creek since I wrote the first post 5 years ago.

The large amount of traffic indicates a more important fact other than that my writing skill is good(which is totally the opposite): Developers like to read simple learning materials and quick solutions. By analyzing the traffic data of blog posts, I learned which ways of explaining things are preferred by developers.

Many people believe in "no diagram no talk". While visualization is a good way to understand and remember things, there are other ways to enhance learning experience. One is by comparing different but related concepts. For example, by comparing ArrayList with LinkedList, one can better understand them and use them properly. Another way is to look at the frequently asked questions. For example, by reading "Top 10 methods for Java arrays", one can quickly remember some useful methods and use the methods used by the majority.

There are numerous blogs, books and tutorials available to learn Java. A lot of them receive large traffic by developers with a large variety of different interests. Program Creek is just one of them. This collection might be useful for two kinds of people: first, the regular visitors of Program Creek; second, developers who want to read something in a more readable format. Repetition is key of learning any programming languages. Hopefully, this contributes another non-boring repetition for you.

Since this collection is 100% from the blog, there is no good reason to keep two versions of it. The PDF book is converted automatically from the original blog posts. Every title in the book is linked back to the original blog. When it is clicked, it opens the original post in your browser. If you find any problem, please go to the post and leave your comment there. As it is an automatic conversion, there may be some format problem. Please leave your comment if you find one. You can also contact me by email: contact@programcreek.com. Thank you for downloading this PDF!

Christmas Day 2013

Part II.

Java Questions

1. How to Check if an Array Contains a Value in Java Efficiently?

How to check if an array (unsorted) contains a certain value? This is a very useful and frequently used operation in Java. It is also a top voted question on Stack Overflow. As shown in top voted answers, this can be done in several different ways, but the time complexity could be very different. In the following I will show the time cost of each method.

1.1. Four Different Ways to Check If an Array Contains a Value

1) Using List:

```
public static boolean useList(String[] arr, String targetValue) {  
    return Arrays.asList(arr).contains(targetValue);  
}
```

2) Using Set:

```
public static boolean useSet(String[] arr, String targetValue) {  
    Set<String> set = new HashSet<String>(Arrays.asList(arr));  
    return set.contains(targetValue);  
}
```

3) Using a simple loop:

```
public static boolean useLoop(String[] arr, String targetValue) {  
    for(String s: arr){  
        if(s.equals(targetValue))  
            return true;  
    }  
    return false;  
}
```

4) Using Arrays.binarySearch(): * The code below is wrong, it is listed here for completeness. binarySearch() can ONLY be used on sorted arrays. You will see the result is weird when running the code below.

```
public static boolean useArraysBinarySearch(String[] arr, String targetValue)  
{  
    int a = Arrays.binarySearch(arr, targetValue);
```

```
if(a > 0)
    return true;
else
    return false;
}
```

1.2. Time Complexity

The approximate time cost can be measured by using the following code. The basic idea is to search an array of size 5, 1k, 10k. The approach may not be precise, but the idea is clear and simple.

```
public static void main(String[] args) {
    String[] arr = new String[] { "CD", "BC", "EF", "DE", "AB"};

    //use list
    long startTime = System.nanoTime();
    for (int i = 0; i < 100000; i++) {
        useList(arr, "A");
    }
    long endTime = System.nanoTime();
    long duration = endTime - startTime;
    System.out.println("useList: " + duration / 1000000);

    //use set
    startTime = System.nanoTime();
    for (int i = 0; i < 100000; i++) {
        useSet(arr, "A");
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("useSet: " + duration / 1000000);

    //use loop
    startTime = System.nanoTime();
    for (int i = 0; i < 100000; i++) {
        useLoop(arr, "A");
    }
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("useLoop: " + duration / 1000000);

    //use Arrays.binarySearch()
    startTime = System.nanoTime();
    for (int i = 0; i < 100000; i++) {
        useArraysBinarySearch(arr, "A");
    }
    endTime = System.nanoTime();
}
```

```
duration = endTime - startTime;  
System.out.println("useArrayBinary: " + duration / 1000000);  
}
```

Result:

```
useList: 13  
useSet: 72  
useLoop: 5  
useArraysBinarySearch: 9
```

Use a larger array (1k):

```
String[] arr = new String[1000];  
  
Random s = new Random();  
for(int i=0; i< 1000; i++){  
    arr[i] = String.valueOf(s.nextInt());  
}
```

Result:

```
useList: 112  
useSet: 2055  
useLoop: 99  
useArrayBinary: 12
```

Use a larger array (10k):

```
String[] arr = new String[10000];  
  
Random s = new Random();  
for(int i=0; i< 10000; i++){  
    arr[i] = String.valueOf(s.nextInt());  
}
```

Result:

```
useList: 1590  
useSet: 23819  
useLoop: 1526  
useArrayBinary: 12
```

Clearly, using a simple loop method is more efficient than using any collection. A lot of developers use the first method, but it is inefficient. Pushing the array to another collection requires spin through all elements to read them in before doing anything with the collection type.

The array must be sorted, if `Arrays.binarySearch()` method is used. In this case, the array is not sorted, therefore, it should not be used.

Actually, if you really need to check if a value is contained in some array/collection efficiently, a sorted list or tree can do it in $O(\log(n))$ or hashset can do it in $O(1)$.

2. Top 10 Questions about Java Exceptions

This article summarizes the top 10 frequently asked questions about Java exceptions.

2.1. Checked vs. Unchecked

In brief, checked exceptions must be explicitly caught in a method or declared in the method's throws clause. Unchecked exceptions are caused by problems that can not be solved, such as dividing by zero, null pointer, etc. Checked exceptions are especially important because you expect other developers who use your API to know how to handle the exceptions.

For example, `IOException` is a commonly used checked exception and `RuntimeException` is an unchecked exception. You can check out the [Java Exception Hierarchy Diagram](#) before reading the rest.

2.2. Best practice for exception management

If an exception can be properly handled then it should be caught, otherwise, it should be thrown.

2.3. Why variables defined in try can not be used in catch or finally?

In the following code, the string `s` declared in try block can not be used in catch clause. The code does not pass compilation.

```
try {
    File file = new File("path");
    FileInputStream fis = new FileInputStream(file);
    String s = "inside";
} catch (FileNotFoundException e) {
    e.printStackTrace();
    System.out.println(s);
}
```

The reason is that you don't know where in the try block the exception would be thrown. It is quite possible that the exception is thrown before the object is declared. This is true for this particular example.

2.4. Why do Double.parseDouble(null) and Integer.parseInt(null) throw different exceptions?

They actually throw different exceptions. This is a problem of JDK. They are developed by different developers, so it does not worth too much thinking.

```
Integer.parseInt(null);  
// throws java.lang.NumberFormatException: null  
  
Double.parseDouble(null);  
// throws java.lang.NullPointerException
```

2.5. Commonly used runtime exceptions in Java

Here are just some of them. IllegalArgumentException ArrayIndexOutOfBoundsException

exception
They can be used in if statement when the condition is not satisfied as follows:

```
if (obj == null) {  
    throw new IllegalArgumentException("obj can not be null");  
}
```

2.6. Can we catch multiple exceptions in the same catch clause?

The answer is YES. As long as those exception classes can trace back to the same super class in the class inheritance hierarchy, you can use that super class only.

2.7. Can constructor throw exceptions in java?

The answer is YES. Constructor is a special kind of method. [Here](#) is a code example.

2.8. Throw exception in final clause

It is legal to do the following:

```
public static void main(String[] args) {  
    File file1 = new File("path1");  
    File file2 = new File("path2");  
    try {
```

```

        FileInputStream fis = new FileInputStream(file1);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        try {
            FileInputStream fis = new FileInputStream(file2);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

But to have better code readability, you should wrap the embedded try-catch block as a new method, and then put the method invocation in the finally clause.

```

public static void main(String[] args) {
    File file1 = new File("path1");
    File file2 = new File("path2");
    try {

        FileInputStream fis = new FileInputStream(file1);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        methodThrowException();
    }
}

```

2.9. Can return be used in finally block

Yes, it can.

2.10. . Why developers consume exception silently?

There are so many time code segments like the following occur. If properly handling exceptions are so important, why developers keep doing that?

```

try {
    ...
} catch (Exception e) {
    e.printStackTrace();
}

```

Ignoring is just easy. Frequent occurrence does not mean correctness.

3. Why Field Can't Be Overridden?

This article shows the basic object oriented concept in Java - Field Hiding.

3.1. Can Field Be Overridden in Java?

Let's first take a look at the following example which creates two Sub objects. One is assigned to a Sub reference, the other is assigned to a Super reference.

```
package oo;

class Super {
    String s = "Super";
}

class Sub extends Super {
    String s = "Sub";
}

public class FieldOverriding {
    public static void main(String[] args) {
        Sub c1 = new Sub();
        System.out.println(c1.s);

        Super c2 = new Sub();
        System.out.println(c2.s);
    }
}
```

What is the output?

Sub
Super

We did create two Sub objects, but why the second one prints out "Super"?

3.2. Hiding Fields Instead Of Overriding Them

In [1], there is a clear definition of Hiding Fields:

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super. Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

From this definition, member fields can not be overridden like methods. When a subclass defines a field with the same name, the subclass just declares a new field.

The field in the superclass is hidden. It is NOT overridden, so it can not be accessed polymorphically.

3.3. Ways to Access Hidden Fields

1). By using parenting reference type, the hidden parent fields can be access, like the example above. 2). By casting you can access the hidden member in the superclass.

```
System.out.println(((Super)c1).s);
```

4. Constructors of Sub and Super Classes in Java?

This post summarizes a commonly asked question about Java constructors.

4.1. Why creating an object of the sub class invokes also the constructor of the super class?

```
class Super {  
    String s;  
  
    public Super(){  
        System.out.println("Super");  
    }  
}  
  
public class Sub extends Super {  
  
    public Sub(){  
        System.out.println("Sub");  
    }  
  
    public static void main(String[] args){  
        Sub s = new Sub();  
    }  
}
```

It prints:

```
Super  
Sub
```

When inheriting from another class, `super()` has to be called first in the constructor. If not, the compiler will insert that call. This is why super constructor is also invoked when a Sub object is created.

This doesn't create two objects, only one Sub object. The reason to have super constructor called is that if super class could have private fields which need to be initialized by its constructor.

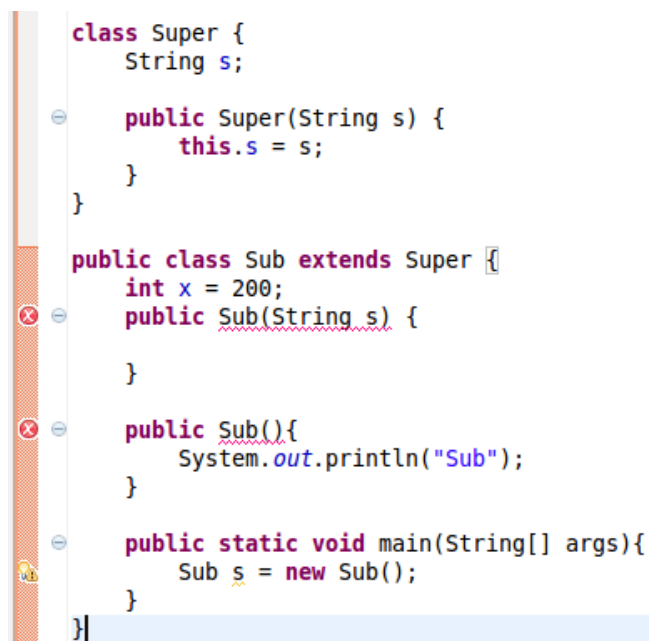
After compiler inserts the super constructor, the sub class constructor looks like the following:

```
public Sub(){
    super();
    System.out.println("Sub");
}
```

4.2. A Common Error Message: Implicit super constructor is undefined for default constructor

This is a compilation error message seen by a lot of Java developers:

"Implicit super constructor is undefined for default constructor. Must define an explicit constructor"



```
class Super {
    String s;

    public Super(String s) {
        this.s = s;
    }
}

public class Sub extends Super {
    int x = 200;
    public Sub(String s) {
    }

    public Sub(){
        System.out.println("Sub");
    }

    public static void main(String[] args){
        Sub s = new Sub();
    }
}
```

This compilation error occurs because the default super constructor is undefined. In Java, if a class does not define a constructor, compiler will insert a default no-argument constructor for the class by default. If a constructor is defined in Super class, in this

case `Super(String s)`, compiler will not insert the default no-argument constructor. This is the situation for the Super class above.

The constructors of the Sub class, either with-argument or no-argument, will call the no-argument Super constructor. Since compiler tries to insert `super()` to the 2 constructors in the Sub class, but the Super's default constructor is not defined, compiler reports the error message.

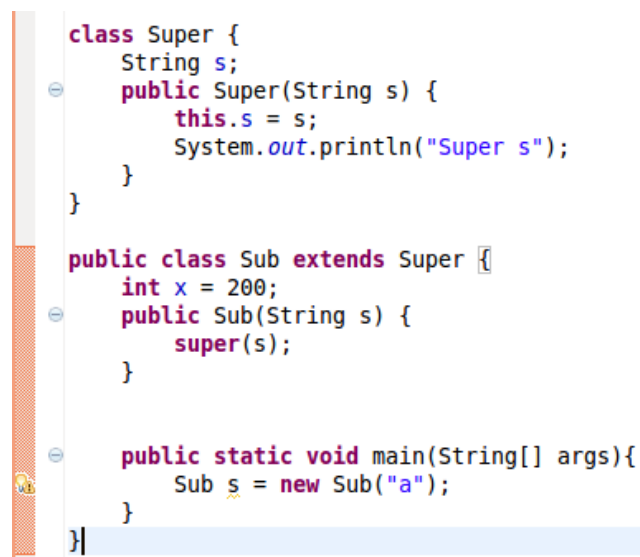
To fix this problem, simply 1) add a `Super()` constructor to the Super class like

```
public Super(){
    System.out.println("Super");
}
```

, or 2) remove the self-defined Super constructor, or 3) add `super(value)` to sub constructors.

4.3. Explicitly call super constructor in sub constructor

The following code is OK:



```
class Super {
    String s;
    public Super(String s) {
        this.s = s;
        System.out.println("Super s");
    }
}

public class Sub extends Super {
    int x = 200;
    public Sub(String s) {
        super(s);
    }

    public static void main(String[] args){
        Sub s = new Sub("a");
    }
}
```

The Sub constructor explicitly call the super constructor with parameter. The super constructor is defined, and good to invoke.

4.4. The Rule

In brief, the rules is: sub class constructor has to invoke super class instructor, either explicitly by programmer or implicitly by compiler. For either way, the invoked super constructor has to be defined.

4.5. The Interesting Question

Why Java doesn't provide default constructor, if class has a constructor with parameter(s)?

Some answers: <http://stackoverflow.com/q/16046200/127859>

5. Java Enum Examples

An enum in Java is just like any other class, with a predefined set of instances. Here are several examples to highlight how to use Java Enum.

5.1. Simple Example

```
public enum Color {  
    RED, YELLOW, BLUE; //each is an instance of Color  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        for(Color color: Color.values()){  
            System.out.println(color);  
        }  
    }  
}
```

Output:

```
RED  
YELLOW  
BLUE
```

5.2. With Constructor

```
public enum Color {  
    RED(1), YELLOW(2), BLUE(3); //each is an instance of Color  
  
    private int value;  
  
    private Color(){}  
  
    private Color(int i){  
        this.value = i;  
    }  
}
```

```

    }

    //define instance method
    public void printValue(){
        System.out.println(this.value);
    }
}

public class Test {
    public static void main(String[] args) {
        for(Color color: Color.values()){
            color.printValue();
        }
    }
}

```

```

1
2
3

```

5.3. When to Use Java Enum?

Recall the definition of Java Enum which is like any other class, with a predefined set of instances.

A good use case is preventing the possibility of an invalid parameter. For example, imagine the following method:

```
public void doSomethingWithColor(int color);
```

This is ambiguous, and other developers have no idea how value to use. If you have an enum Color with BLACK, RED, etc. the signature becomes:

```
public void doSomethingWithColor(Color color);
```

Code calling this method will be far more readable, and can't provide invalid data.

6. Java Access Level for Members: public, protected, private

Java access level contains two parts: 1) access level for classes and 2) access level for members.

For class access level, the keyword can be public or no explicit modifier(package-private). For member access level, the keyword can be public, protected, package-private (no explicit modifier), or private.

The following table summarizes the access level of different modifiers for members. Access level determines the accessibility of fields and methods. It has 4 levels: public, protected, package-private (no explicit modifier), or private.

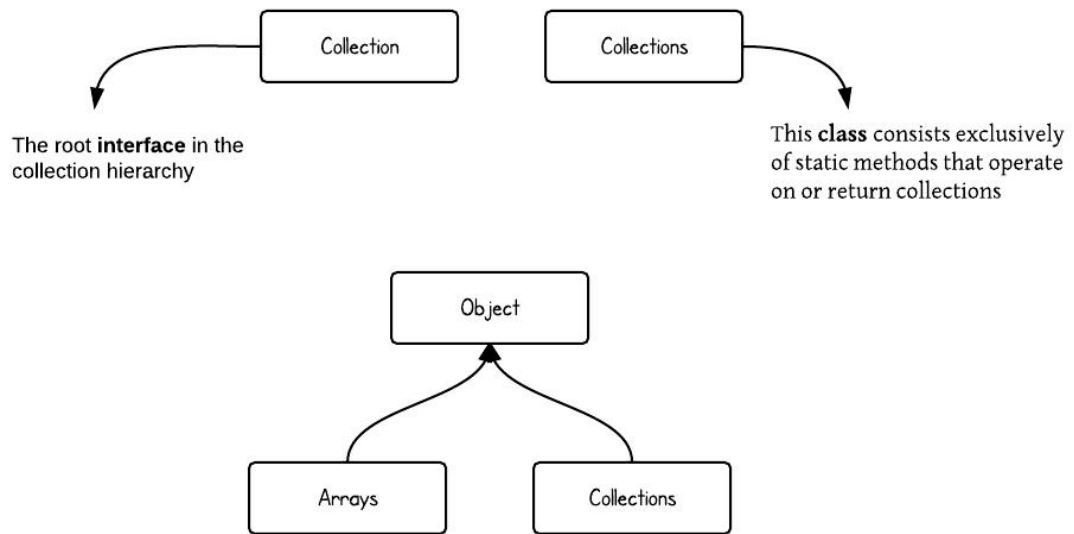
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	X
no modifier	Y	Y	X	X
private	Y	X	X	X

7. The Interface and Class Hierarchy

Diagram of Java Collections

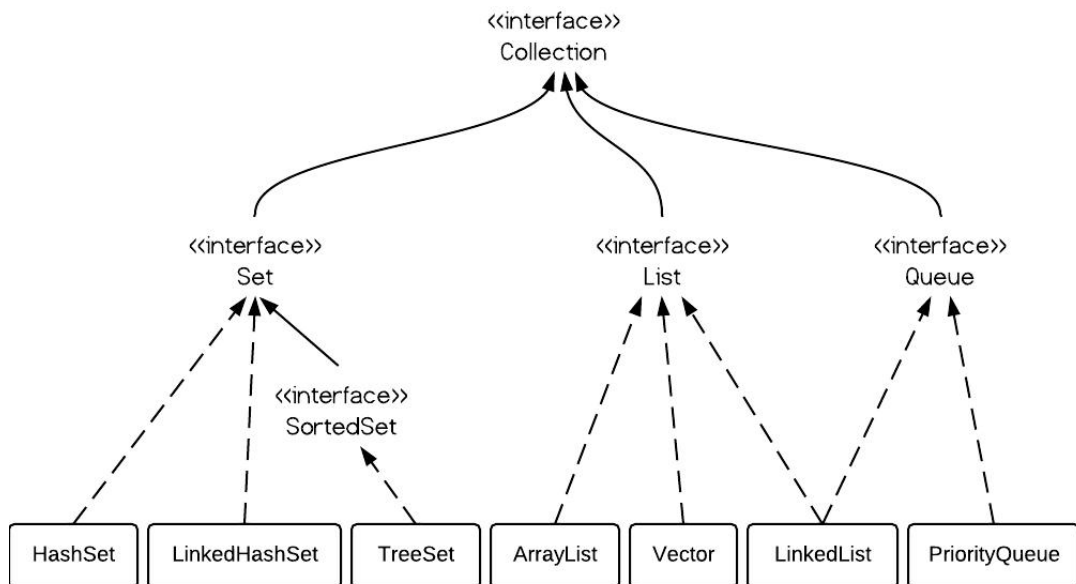
7.1. Collection vs Collections

First of all, "Collection" and "Collections" are two different concepts. As you will see from the hierarchy diagram below, "Collection" is a root interface in the Collection hierarchy but "Collections" is a class which provide static methods to manipulate on some Collection types.



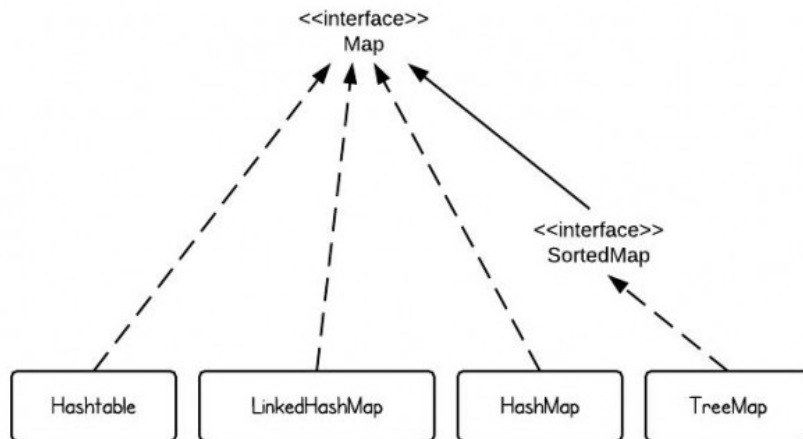
7.2. Class hierarchy of Collection

The following diagram demonstrates class hierarchy of Collection.



7.3. Class hierarchy of Map

Here is class hierarchy of Map.



7.4. Summary of classes

Interfaces	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

7.5. Code Example

The following is a simple example to illustrate some collection types:

```

List<String> a1 = new ArrayList<String>();
a1.add("Program");
a1.add("Creek");
a1.add("Java");
a1.add("Java");
System.out.println("ArrayList Elements");
System.out.print("\t" + a1 + "\n");

List<String> l1 = new LinkedList<String>();
l1.add("Program");
l1.add("Creek");
l1.add("Java");
l1.add("Java");
System.out.println("LinkedList Elements");
  
```

```

System.out.print("\t" + l1 + "\n");

Set<String> s1 = new HashSet<String>(); // or new TreeSet() will order the
    elements;
s1.add("Program");
s1.add("Creek");
s1.add("Java");
s1.add("Java");
s1.add("tutorial");
System.out.println("Set Elements");
System.out.print("\t" + s1 + "\n");

Map<String, String> m1 = new HashMap<String, String>(); // or new TreeMap()
    will order based on keys
m1.put("Windows", "2000");
m1.put("Windows", "XP");
m1.put("Language", "Java");
m1.put("Website", "programcreek.com");
System.out.println("Map Elements");
System.out.print("\t" + m1);

```

Output:

```

ArrayList Elements
    [Program, Creek, Java, Java]
LinkedList Elements
    [Program, Creek, Java, Java]
Set Elements
    [tutorial, Creek, Program, Java]
Map Elements
    {Windows=XP, Website=programcreek.com, Language=Java}

```

8. Top 9 questions about Java Maps

In general, [Map](#) is a data structure consisting of a set of key-value pairs, and each key can only appears once in the map. This post summarizes Top 9 FAQ of how to use Java [Map](#) and its implemented classes. For sake of simplicity, I will use [generics](#) in examples. Therefore, I will just write [Map](#) instead of specific [Map](#). But you can always assume that both the K and V are comparable, which means K extends Comparable and V extends Comparable.

8.1. Convert a Map to List

In Java, [Map](#) interface provides three collection views: key set, value set, and key-value set. All of them can be converted to [List](#) by using a constructor or `addAll()` method. The following snippet of code shows how to construct an [ArrayList](#) from a map.

```
// key list
List keyList = new ArrayList(map.keySet());
// value list
List valueList = new ArrayList(map.valueSet());
// key-value list
List entryList = new ArrayList(map.entrySet());
```

8.2. Iterate over each Entry in a Map

Iterating over every pair of key-value is the most basic operation to traverse a map. In Java, such pair is stored in the map entry called [Map.Entry](#). `Map.entrySet()` returns a key-value set, therefore the most efficient way of going through every entry of a map is

```
for(Entry entry: map.entrySet()) {
    // get key
    K key = entry.getKey();
    // get value
    V value = entry.getValue();
}
```

Iterator can also be used, especially before JDK 1.5

```
Iterator itr = map.entrySet().iterator();
while(itr.hasNext()) {
    Entry entry = itr.next();
    // get key
    K key = entry.getKey();
    // get value
    V value = entry.getValue();
}
```

8.3. Sort a Map on the keys

Sorting a map on the keys is another frequent operation. One way is to put [Map.Entry](#) into a list, and sort it using a comparator that sorts the value.

```
List list = new ArrayList(map.entrySet());
Collections.sort(list, new Comparator() {

    @Override
```

```
public int compare(Entry e1, Entry e2) {  
    return e1.getKey().compareTo(e2.getKey());  
}  
  
});
```

The other way is to use [SortedMap](#), which further provides a total ordering on its keys. Therefore all keys must either implement [Comparable](#) or be accepted by the comparator.

One implementing class of SortedMap is [TreeMap](#). Its constructor can accept a comparator. The following code shows how to transform a general map to a sorted map.

```
SortedMap sortedMap = new TreeMap(new Comparator() {  
  
    @Override  
    public int compare(K k1, K k2) {  
        return k1.compareTo(k2);  
    }  
  
});  
sortedMap.putAll(map);
```

8.4. Sort a Map on the values

Putting the map into a list and sorting it works on this case too, but we need to compare `Entry.getValue()` this time. The code below is almost same as before.

```
List list = new ArrayList(map.entrySet());  
Collections.sort(list, new Comparator() {  
  
    @Override  
    public int compare(Entry e1, Entry e2) {  
        return e1.getValue().compareTo(e2.getValue());  
    }  
  
});
```

We can still use a sorted map for this question, but only if the values are unique too. Under such condition, you can reverse the key=value pair to value=key. This solution has very strong limitation therefore is not really recommended by me.

8.5. Initialize a static/immutable Map

When you expect a map to remain constant, it's a good practice to copy it into an immutable map. Such defensive programming techniques will help you create not only safe for use but also safe for thread maps.

To initialize a static/immutable map, we can use a static initializer (like below). The problem of this code is that, although map is declared as static final, we can still operate it after initialization, like `Test.map.put(3,"three");`. Therefore it is not really immutable. To create an immutable map using a static initializer, we need an extra anonymous class and copy it into a unmodifiable map at the last step of initialization. Please see the second piece of code. Then, an `UnsupportedOperationException` will be thrown if you run `Test.map.put(3,"three");`.

```
public class Test {

    private static final Map map;
    static {
        map = new HashMap();
        map.put(1, "one");
        map.put(2, "two");
    }
}

public class Test {

    private static final Map map;
    static {
        Map aMap = new HashMap();
        aMap.put(1, "one");
        aMap.put(2, "two");
        map = Collections.unmodifiableMap(aMap);
    }
}
```

Guava libraries also support different ways of initializing a static and immutable collection. To learn more about the benefits of Guava's immutable collection utilities, see Immutable Collections Explained in [Guava User Guide](#).

8.6. Difference between HashMap, TreeMap, and Hashtable

There are three main implementations of `Map` interface in Java: `HashMap`, `TreeMap`, and `Hashtable`. The most important differences include:

- The order of iteration. `HashMap` and `Hashtable` make no guarantees as to the order of the map; in particular, they do not guarantee that the order will remain constant over time. But `TreeMap` will iterate the whole entries according the "natural ordering" of the keys or by a comparator.
- key-value permission. `HashMap` allows null key and null values (Only one null key is allowed because no two keys are allowed the same). `Hashtable` does not allow null key or null values. If `TreeMap` uses natural ordering or its comparator does not allow null keys, an exception will be thrown.
- Synchronized. Only `Hashtable` is synchronized, others are not. Therefore, "if a

thread-safe implementation is not needed, it is recommended to use `HashMap` in place of `Hashtable`."

A more complete comparison is

	HashMap	Hashtable	TreeMap

iteration order	no	no	yes
<code>null</code> key-value	yes	yes	no-no no-yes
<code>synchronized</code>	no	yes	no
time performance	$O(1)$	$O(1)$	$O(\log n)$
implementation	buckets	buckets	red-black tree

Read more about [HashMap vs. TreeMap vs. Hashtable vs. LinkedHashMap](#).

8.7. A Map with reverse view/lookup

Sometimes, we need a set of key-key pairs, which means the map's values are unique as well as keys (one-to-one map). This constraint enables to create an "inverse lookup/view" of a map. So we can lookup a key by its value. Such data structure is called [bidirectional map](#), which unfortunately is not supported by JDK.

8.8. Both Apache Common Collections and Guava provide implementation of bidirectional map, called `BidiMap` and `BiMap`, respectively. Both enforce the restriction that there is a 1:1 relation between keys and values.

7. Shallow copy of a Map

Most implementation of a map in java, if not all, provides a constructor of copy of another map. But the copy procedure is not synchronized. That means when one thread copies a map, another one may modify it structurally. To [prevent accidental unsynchronized copy, one should use `Collections.synchronizedMap()` in advance.

```
Map copiedMap = Collections.synchronizedMap(map);
```

Another interesting way of shallow copy is by using `clone()` method. However it is NOT even recommended by the designer of Java collection framework, Josh Bloch. In a conversation about "[Copy constructor versus cloning](#)", he said

I often provide a public clone method on concrete classes because people expect it. ... It's a shame that Cloneable is broken, but it happens. ... Cloneable is a weak spot, and I think people should be aware of its limitations.

8.9. For this reason, I will not even tell you how to use clone() method to copy a map. 8. Create an empty Map

If the map is immutable, use

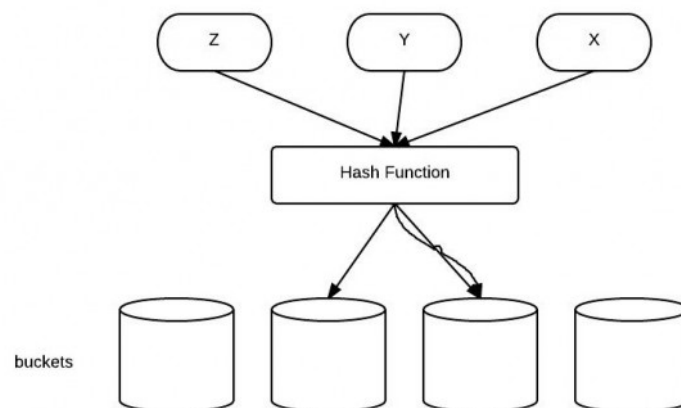
```
map = Collections.emptyMap();
```

Otherwise, use whichever implementation. For example

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

THE END

9. Java equals() and hashCode() Contract



The Java super class `java.lang.Object` has two very important methods defined:

```
public boolean equals(Object obj)
public int hashCode()
```

They have been proved to be extremely important to understand, especially when user-defined objects are added to Maps. However, even advanced-level developers sometimes can't figure out how they should be used properly. In this post, I will first show an example of a common mistake, and then explain how `equals()` and `hashCode()` contract works.

9.1. A common mistake

Common mistake is shown in the example below.

```
import java.util.HashMap;

public class Apple {
    private String color;

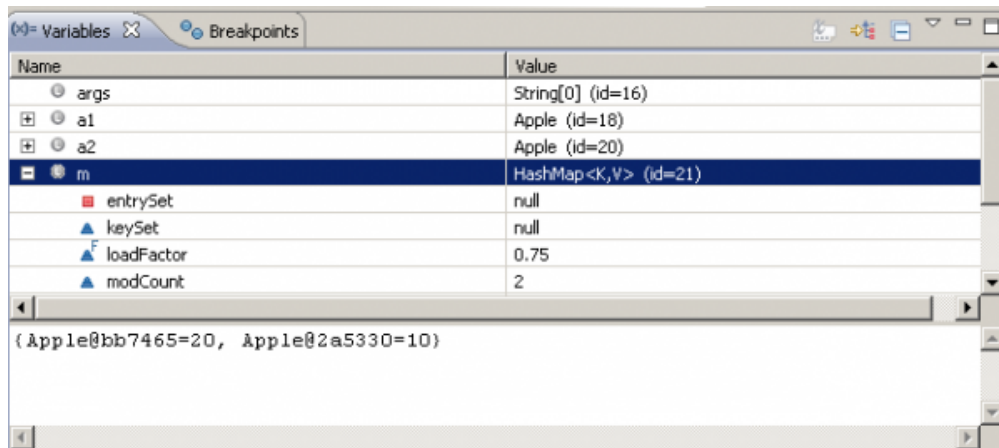
    public Apple(String color) {
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Apple))
            return false;
        if (obj == this)
            return true;
        return this.color.equals(((Apple) obj).color);
    }

    public static void main(String[] args) {
        Apple a1 = new Apple("green");
        Apple a2 = new Apple("red");

        //hashMap stores apple type and its quantity
        HashMap<Apple, Integer> m = new HashMap<Apple, Integer>();
        m.put(a1, 10);
        m.put(a2, 20);
        System.out.println(m.get(new Apple("green")));
    }
}
```

In this example, a green apple object is stored successfully in a hashMap, but when the map is asked to retrieve this object, the apple object is not found. The program above prints null. However, we can be sure that the object is stored in the hashMap by inspecting in the debugger:



9.2. Problem caused by hashCode()

The problem is caused by the un-overridden method "hashCode()". The contract between equals() and hashCode() is that: 1. If two objects are equal, then they must have the same hash code. 2. If two objects have the same hashcode, they may or may not be equal.

The idea behind a Map is to be able to find an object faster than a linear search. Using hashed keys to locate objects is a two-step process. Internally the Map stores objects as an array of arrays. The index for the first array is the hashCode() value of the key. This locates the second array which is searched linearly by using equals() to determine if the object is found.

The default implementation of hashCode() in Object class returns distinct integers for different objects. Therefore, in the example above, different objects(even with same type) have different hashCode.

Hash Code is like a sequence of garages for storage, different stuff can be stored in different garages. It is more efficient if you organize stuff to different place instead of the same garage. So it's a good practice to equally distribute the hashCode value. (Not the main point here though)

The solution is to add hashCode method to the class. Here I just use the color string's length for demonstration.

```
public int hashCode(){
    return this.color.length();
}
```

10. What does a Java array look like in memory?

Arrays in Java store one of two things: either primitive values (int, char, ...) or references (a.k.a pointers).

When an object is created by using "new", memory is allocated on the heap and a reference is returned. This is also true for arrays, since arrays are objects.

10.1. Single-dimension Array

```
int arr[] = new int[3];
```

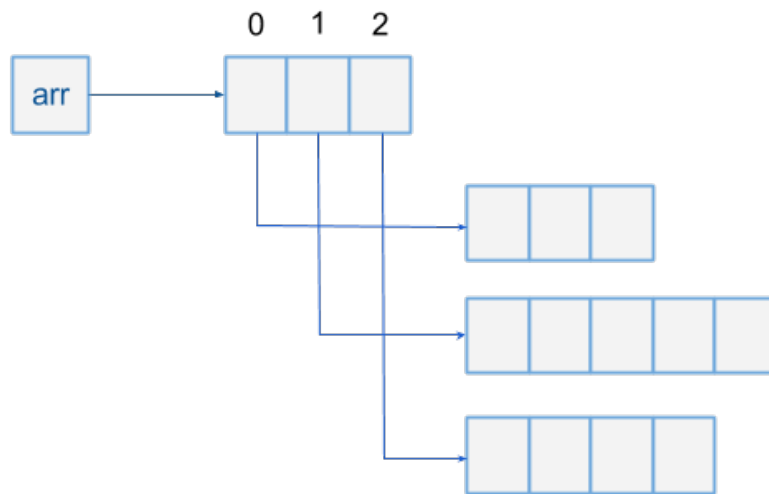
The `int[] arr` is just the reference to the array of 3 integers. If you create an array with 10 integers, it is the same - an array is allocated and a reference is returned.



10.2. Two-dimensional Array

How about 2-dimensional array? Actually, we can only have one dimensional arrays in Java. 2D arrays are basically just one dimensional arrays of one dimensional arrays.

```
int[ ][ ] arr = new int[3][ ];  
arr[0] = new int[3];  
arr[1] = new int[5];  
arr[2] = new int[4];
```



Multi-dimensional arrays use the name rules.

10.3. Where are they located in memory?

Arrays are also objects in Java, so how an object looks like in memory applies to an array.

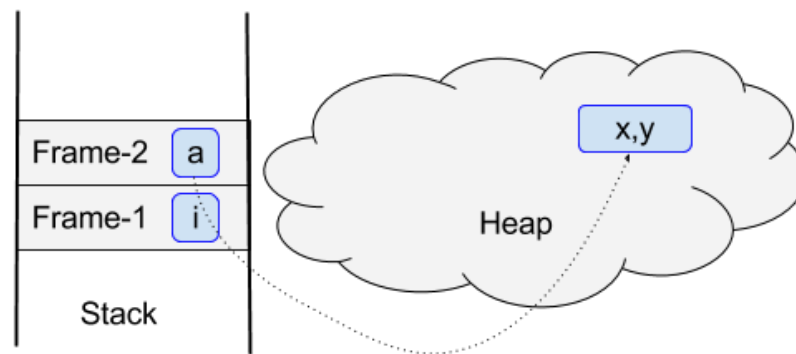
As we know that [JVM runtime data areas](#) include heap, JVM stack, and others. For a simple example as follows, let's see where the array and its reference are stored.

```
class A {  
    int x;  
    int y;  
}  
  
...  
  
public void m1() {  
    int i = 0;  
    m2();  
}  
  
public void m2() {  
    A a = new A();  
}  
  
...
```

With the above declaration, let's invoke `m1()` and see what happens:

- When `m1` is invoked, a new frame (Frame-1) is pushed into the stack, and local variable `i` is also created in Frame-1.

- Then m2 is invoked inside of m1, another new frame (Frame-2) is pushed into the stack. In m2, an object of class A is created in the heap and reference variable is put in Frame-2. Now, at this point, the stack and heap looks like the following:



Arrays are treated the same way like objects, so how array locates in memory is straight-forward.

11. The Introduction of Java Memory Leaks

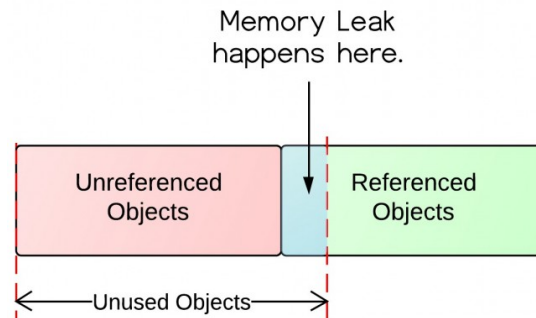
One of the most significant advantages of Java is its memory management. You simply create objects and Java Garbage Collector takes care of allocating and freeing memory. However, the situation is not as simple as that, because memory leaks frequently occur in Java applications.

This tutorial illustrates what is memory leak, why it happens, and how to prevent them.

11.1. What is Memory Leak?

Definition of Memory Leak: objects are no longer being used by the application, but Garbage Collector can not remove them because they are being referenced.

To understand this definition, we need to understand objects status in memory. The following diagram illustrates what is unused and what is unreferenced.

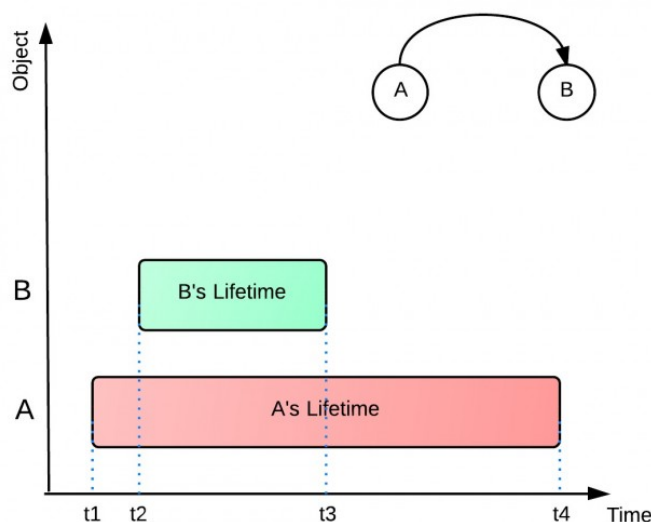


From the diagram, there are referenced objects and unreferenced objects. Unreferenced objects will be garbage collected, while referenced objects will not be garbage collected. Unreferenced objects are surely unused, because no other objects refer to it. However, unused objects are not all unreferenced. Some of them are being referenced! That's where the memory leaks come from.

11.2. Why Memory Leaks Happen?

Let's take a look at the following example and see why memory leaks happen. In the example below, object A refers to object B. A's lifetime ($t_1 - t_4$) is much longer than B's ($t_2 - t_3$). When B is no longer being used in the application, A still holds a reference to it. In this way, Garbage Collector can not remove B from memory. This would possibly cause out of memory problem, because if A does the same thing for more objects, then there would be a lot of objects that are uncollected and consume memory space.

It is also possible that B hold a bunch of references of other objects. Those objects referenced by B will not get collected either. All those unused objects will consume precious memory space.



11.3. How to Prevent Memory Leaks?

The following are some quick hands-on tips for preventing memory leaks.

- Pay attention to Collection classes, such as HashMap, ArrayList, etc., as they are common places to find memory leaks. When they are declared static, their life time is the same as the life time of the application.
- Pay attention to event listeners and callbacks. A memory leak may occur if a listener is registered but not unregistered when the class is not being used any longer.
- "If a class manages its own memory, the programmer should be alert for memory leaks." [1] Often times member variables of an object that point to other objects need to be null out.

11.4. A little Quiz: Why substring() method in JDK 6 can cause memory leaks?

To answer this question, you may want to read Substring() in JDK 6 and 7.

12. Frequently Used Methods of Java HashMap

HashMap is very useful when a counter is required.

```
HashMap<String, Integer> countMap = new HashMap<String, Integer>();

//.... a lot of a's like the following
if(countMap.keySet().contains(a)){
    countMap.put(a, countMap.get(a)+1);
}else{
    countMap.put(a, 1);
}
```

12.1. Loop Through HashMap

```
Iterator it = mp.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry pairs = (Map.Entry)it.next();
    System.out.println(pairs.getKey() + " = " + pairs.getValue());
}
```

```
Map<Integer, Integer> map = new HashMap<Integer, Integer>();
for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
    System.out.println("Key = " + entry.getKey() + ", Value = " +
        entry.getValue());
}
```

12.2. Print HashMap

```
public static void printMap(Map mp) {
    Iterator it = mp.entrySet().iterator();
    while (it.hasNext()) {
        Map.Entry pairs = (Map.Entry)it.next();
        System.out.println(pairs.getKey() + " = " + pairs.getValue());
        it.remove(); // avoids a ConcurrentModificationException
    }
}
```

12.3. Sort HashMap by Value

The following code example take advantage of a constructor of TreeMap here.

```
class ValueComparator implements Comparator<String> {

    Map<String, Integer> base;

    public ValueComparator(Map<String, Integer> base) {
        this.base = base;
    }

    public int compare(String a, String b) {
        if (base.get(a) >= base.get(b)) {
            return -1;
        } else {
            return 1;
        } // returning 0 would merge keys
    }
}
```

```
HashMap<String, Integer> countMap = new HashMap<String, Integer>();
//add a lot of entries
countMap.put("a", 10);
countMap.put("b", 20);

ValueComparator vc = new ValueComparator(countMap);
TreeMap<String,Integer> sortedMap = new TreeMap<String,Integer>(vc);
```

```
sortedMap.putAll(countMap);
```

```
printMap(sortedMap);
```

There are different ways of sorting HashMap, this way has been voted the most in [stackoverflow](#).

13. How Java Compiler Generate Code for Overloaded and Overridden Methods?

Here is a simple Java example showing Polymorphism: overloading and overriding.

Polymorphism means that functions assume different forms at different times. In case of compile time it is called function overloading. Overloading allows related methods to be accessed by use of a common name. It is sometimes called ad hoc polymorphism, as opposed to the parametric polymorphism.

```
class A {
    public void M(int i){
        System.out.println("int");
    }

    public void M(String s){
        //this is an overloading method
        System.out.println("string");
    }
}

class B extends A{
    public void M(int i){
        //this is overriding method
        System.out.println("overriden int");
    }
}
```

```
public static void main(String[] args) {
    A a = new A();
    a.M(1);
    a.M("abc");

    A b = new B();
    b.M(1234);
}
```

```
}
```

From the compiler perspective, how is code generated for the correct function calls?

Static overloading is not hard to implement. When processing the declaration of an overloaded function, a new binding maps it to a different implementation. During the type checking process, compiler analyzes the parameter's real type to determine which function to use.

Dynamic overloading allows different implementations of a function to be chosen on the run-time type of an actual parameter. It is a form of dynamic dispatch.

Dynamic dispatch is also used to implement method overriding. The overridden method are determined by real object type during run-time.

To understand dynamic dispatch, there is a post about [object layout in memory](#).

14. String is passed by “reference” in Java

This is a classic question of Java. Many similar questions have been asked on stack-overflow, and there are a lot of incorrect/incomplete answers. The question is simple if you don't think too much. But it could be very confusing, if you give more thought to it.

14.1. A code fragment that is interesting & confusing

```
public static void main(String[] args) {  
    String x = new String("ab");  
    change(x);  
    System.out.println(x);  
}  
  
public static void change(String x) {  
    x = "cd";  
}
```

It prints "ab".

In C++, the code is as follows:

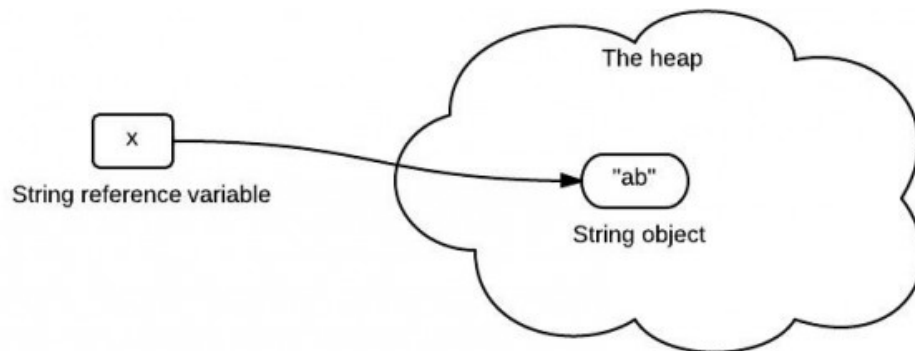
```
void change(string &x) {  
    x = "cd";  
}  
  
int main(){  
    string x = "ab";
```

```
change(x);  
cout << x << endl;  
}
```

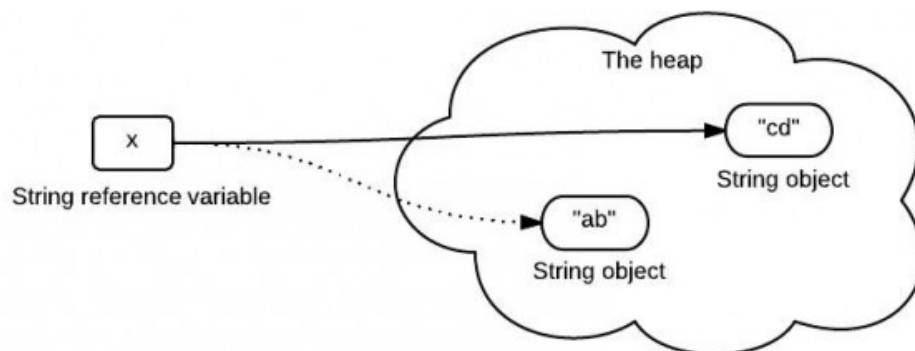
it prints "cd".

14.2. Common confusing questions

x stores the reference which points to the "ab" string in the heap. So when x is passed as a parameter to the change() method, it still points to the "ab" in the heap like the following:



Because java is pass-by-value, the value of x is the reference to "ab". When the method change() gets invoked, it creates a new "cd" object, and x now is pointing to "cd" like the following:



It seems to be a pretty reasonable explanation. They are clear that Java is always pass-by-value. But what is wrong here?

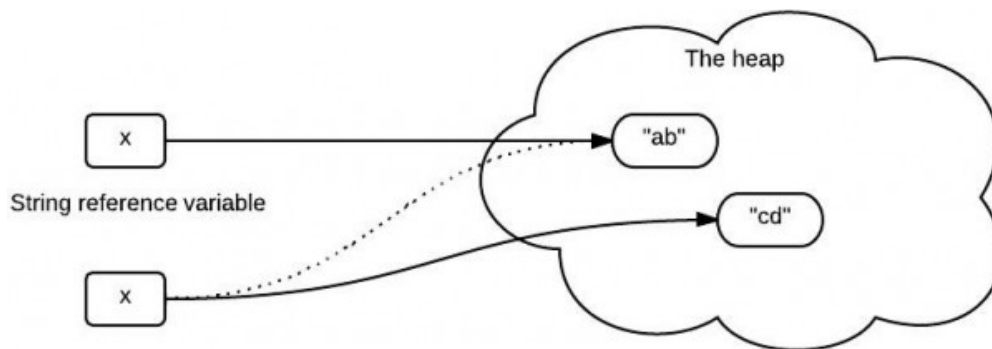
14.3. What the code really does?

The explanation above has several mistakes. To understand this easily, it is a good idea to briefly walk through the whole process.

When the string "ab" is created, Java allocates the amount of memory required to store the string object. Then, the object is assigned to variable x, the variable is actually assigned a reference to the object. This reference is the address of the memory location where the object is stored.

The variable x contains a reference to the string object. x is not a reference itself! It is a variable that stores a reference(memory address).

Java is pass-by-value ONLY. When x is passed to the change() method, a copy of value of x (a reference) is passed. The method change() creates another object "cd" and it has a different reference. It is the variable x that changes its reference(to "cd"), not the reference itself.



14.4. The wrong explanation

The problem raised from the first code fragment is nothing related with [string immutability](#). Even if String is replaced with StringBuilder, the result is still the same. The key point is that variable stores the reference, but is not the reference itself!

14.5. Solution to this problem

If we really need to change the value of the object. First of all, the object should be changeable, e.g., StringBuilder. Secondly, we need to make sure that there is no new object created and assigned to the parameter variable, because Java is passing-by-value only.

```
public static void main(String[] args) {
    StringBuilder x = new StringBuilder("ab");
    change(x);
    System.out.println(x);
}
```

```
public static void change(StringBuilder x) {  
    x.delete(0, 2).append("cd");  
}
```

15. FileOutputStream vs. FileWriter

When we use Java to write something to a file, we can do it in the following two ways. One uses `FileOutputStream`, the other uses `FileWriter`.

Using `FileOutputStream`:

```
File fout = new File(file_location_string);  
FileOutputStream fos = new FileOutputStream(fout);  
BufferedWriter out = new BufferedWriter(new OutputStreamWriter(fos));  
out.write("something");
```

Using `FileWriter`:

```
FileWriter fstream = new FileWriter(file_location_string);  
BufferedWriter out = new BufferedWriter(fstream);  
out.write("something");
```

Both will work, but what is the difference between `FileOutputStream` and `FileWriter`?

There are a lot of discussion on each of those classes, they both are good implements of file i/o concept that can be found in a general operating systems. However, we don't care how it is designed, but only how to pick one of them and why pick it that way.

From Java API Specification:

FileOutputStream is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using FileWriter.

If you are familiar with design patterns, `FileWriter` is a typical usage of Decorator pattern actually. I have use a simple [tutorial to demonstrate the Decorator pattern](#), since it is very important and very useful for many designs.

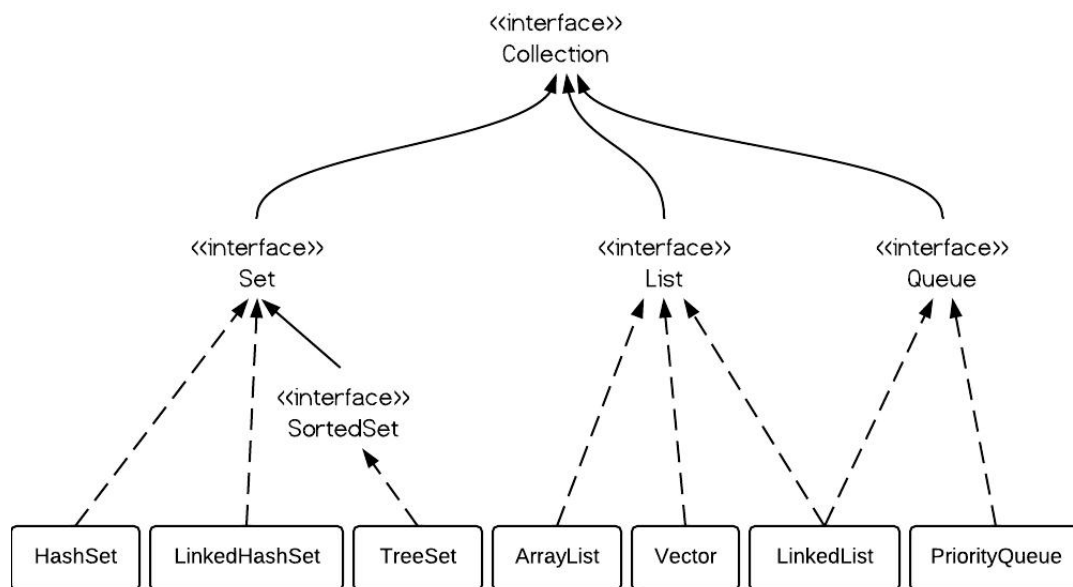
One application of `FileOutputStream` is [converting a file to a byte array](#).

16. HashSet vs. TreeSet vs. LinkedHashSet

A Set contains no duplicate elements. That is one of the major reasons to use a set. There are 3 commonly used implementations of Set: HashSet, TreeSet and LinkedHashSet. When and which to use is an important question. In brief, if you need a fast set, you should use HashSet; if you need a sorted set, then TreeSet should be used; if you need a set that can store the insertion order, LinkedHashSet should be used.

16.1. Set Interface

Set interface extends Collection interface. In a set, no duplicates are allowed. Every element in a set must be unique. You can simply add elements to a set, and duplicates will be removed automatically.



16.2. HashSet vs. TreeSet vs. LinkedHashSet

HashSet is Implemented using a hash table. Elements are not ordered. The add, remove, and contains methods have constant time complexity $O(1)$.

TreeSet is implemented using a tree structure(red-black tree in algorithm book). The elements in a set are sorted, but the add, remove, and contains methods has time complexity of $O(\log(n))$. It offers several methods to deal with the ordered set like `first()`, `last()`, `headSet()`, `tailSet()`, etc.

LinkedHashSet is between HashSet and TreeSet. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion. The time complexity of basic methods is $O(1)$.

16.3. TreeSet Example

```
TreeSet<Integer> tree = new TreeSet<Integer>();
tree.add(12);
tree.add(63);
tree.add(34);
tree.add(45);

Iterator<Integer> iterator = tree.iterator();
System.out.print("Tree set data: ");
while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
}
```

Output is sorted as follows:

```
Tree set data: 12 34 45 63
```

Now let's define a Dog class as follows:

```
class Dog {
    int size;

    public Dog(int s) {
        size = s;
    }

    public String toString() {
        return size + "";
    }
}
```

Let's add some dogs to TreeSet like the following:

```
import java.util.Iterator;
import java.util.TreeSet;

public class TestTreeSet {
    public static void main(String[] args) {
        TreeSet<Dog> dset = new TreeSet<Dog>();
        dset.add(new Dog(2));
        dset.add(new Dog(1));
        dset.add(new Dog(3));

        Iterator<Dog> iterator = dset.iterator();
```



```
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
```

Compile ok, but run-time error occurs:

```
Exception in thread "main" java.lang.ClassCastException: collection.Dog
    cannot be cast to java.lang.Comparable
    at java.util.TreeMap.put(Unknown Source)
    at java.util.TreeSet.add(Unknown Source)
    at collection.TestTreeSet.main(TestTreeSet.java:22)
```

Because TreeSet is sorted, the Dog object need to implement java.lang.Comparable's compareTo() method like the following:

```
class Dog implements Comparable<Dog>{
    int size;

    public Dog(int s) {
        size = s;
    }

    public String toString() {
        return size + "";
    }

    @Override
    public int compareTo(Dog o) {
        return size - o.size;
    }
}
```

The output is:

```
1 2 3
```

16.4. HashSet Example

```
HashSet<Dog> dset = new HashSet<Dog>();
dset.add(new Dog(2));
dset.add(new Dog(1));
dset.add(new Dog(3));
dset.add(new Dog(5));
dset.add(new Dog(4));
Iterator<Dog> iterator = dset.iterator();
```

```
while (iterator.hasNext()) {  
    System.out.print(iterator.next() + " ");  
}
```

Output:

5 3 2 1 4

Note the order is not certain.

16.5. LinkedHashSet Example

```
LinkedHashSet<Dog> dset = new LinkedHashSet<Dog>();  
dset.add(new Dog(2));  
dset.add(new Dog(1));  
dset.add(new Dog(3));  
dset.add(new Dog(5));  
dset.add(new Dog(4));  
Iterator<Dog> iterator = dset.iterator();  
while (iterator.hasNext()) {  
    System.out.print(iterator.next() + " ");  
}
```

The order of the output is certain and it is the insertion order:

2 1 3 5 4

16.6. Performance testing

The following method tests the performance of the three class on add() method.

```
public static void main(String[] args) {  
  
    Random r = new Random();  
  
    HashSet<Dog> hashSet = new HashSet<Dog>();  
    TreeSet<Dog> treeSet = new TreeSet<Dog>();  
    LinkedHashSet<Dog> linkedSet = new LinkedHashSet<Dog>();  
  
    // start time  
    long startTime = System.nanoTime();  
  
    for (int i = 0; i < 1000; i++) {  
        int x = r.nextInt(1000 - 10) + 10;  
        hashSet.add(new Dog(x));  
    }  
    // end time
```

```

    long endTime = System.nanoTime();
    long duration = endTime - startTime;
    System.out.println("HashSet: " + duration);

    // start time
    startTime = System.nanoTime();
    for (int i = 0; i < 1000; i++) {
        int x = r.nextInt(1000 - 10) + 10;
        treeSet.add(new Dog(x));
    }
    // end time
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("TreeSet: " + duration);

    // start time
    startTime = System.nanoTime();
    for (int i = 0; i < 1000; i++) {
        int x = r.nextInt(1000 - 10) + 10;
        linkedSet.add(new Dog(x));
    }
    // end time
    endTime = System.nanoTime();
    duration = endTime - startTime;
    System.out.println("LinkedHashSet: " + duration);
}

```

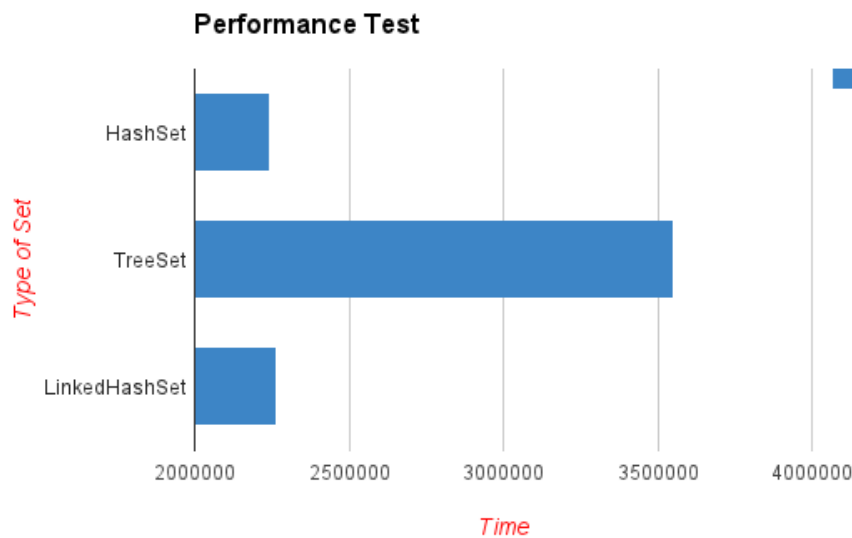
From the output below, we can clearly see that HashSet is the fastest one.

```

HashSet: 2244768
TreeSet: 3549314
LinkedHashSet: 2263320

```

* The test is not precise, but can reflect the basic idea that TreeSet is much slower because it is sorted.



Read: [ArrayList vs. LinkedList vs. Vector](#)

17. How to Write a File Line by Line in Java?

This is Java code for writing something to a file. Every time after it runs, a new file is created, and the previous one is gone. This is different from appending content to a file.

```
public static void writeFile1() throws IOException {
    File fout = new File("out.txt");
    FileOutputStream fos = new FileOutputStream(fout);

    BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));

    for (int i = 0; i < 10; i++) {
        bw.write("something");
        bw.newLine();
    }

    bw.close();
}
```

This example use `FileOutputStream`, instead you can use `FileWriter` or `PrintWriter` which is normally good enough for a text file operations.

Use FileWriter:

```
public static void writeFile2() throws IOException {
    FileWriter fw = new FileWriter("out.txt");

    for (int i = 0; i < 10; i++) {
        fw.write("something");
    }

    fw.close();
}
```

Use PrintWriter:

```
public static void writeFile3() throws IOException {
    PrintWriter pw = new PrintWriter(new FileWriter("out.txt"));

    for (int i = 0; i < 10; i++) {
        pw.write("something");
    }

    pw.close();
}
```

Use OutputStreamWriter:

```
public static void writeFile4() throws IOException {
    File fout = new File("out.txt");
    FileOutputStream fos = new FileOutputStream(fout);

    OutputStreamWriter osw = new OutputStreamWriter(fos);

    for (int i = 0; i < 10; i++) {
        osw.write("something");
    }

    osw.close();
}
```

From Java Doc:

FileWriter is a convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.

PrintWriter prints formatted representations of objects to a text-output stream. This class implements all of the print methods found in PrintStream. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

The main difference is that PrintWriter offers some additional methods for formatting such as println and printf. In addition, FileWriter throws IOException in case

of any I/O failure. `PrintWriter` methods do not throw `IOException`, instead they set a boolean flag which can be obtained using `checkError()`. `PrintWriter` automatically invokes `flush` after every byte of data is written. In case of `FileWriter`, caller has to take care of invoking `flush`.

18. What can we learn from Java HelloWorld?

This is the program every Java programmer knows. It is simple, but a simple start can lead to deep understanding of more complex concepts. In this post I will explore what can be learned from this simple program. Please leave your comments if hello world means more to you.

HelloWorld.java

```
public class HelloWorld {  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("Hello World");  
    }  
}
```

18.1. Why everything starts with a class?

Java programs are built from classes, every method and field has to be in a class. This is due to its object-oriented feature: everything is an object which is an instance of a class. Object-oriented programming languages have a lot of advantages over functional programming languages such as better modularity, extensibility, etc.

18.2. Why there is always a "main" method?

The "main" method is the program entrance and it is static. "static" means that the method is part of its class, not part of objects.

Why is that? Why don't we put a non-static method as program entrance?

If a method is not static, then an object needs to be created first to use the method. Because the method has to be invoked on an object. For the entrance purpose, this is not realistic. We can not get an egg without a chicken. Therefore, program entrance method is static.

The parameter "String[] args" indicates that an array of strings can be sent to the program to help with program initialization.

18.3. Bytecode of HelloWorld

To execute the program, Java file is first compiled to java byte code stored in the .class file. What does the byte code look like? The byte code itself is not readable. If we use a hex editor, it looks like the following:

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000000	CA	FE	BA	BE	00	00	00	32	00	2C	07	00	02	01	00	092,.....
000010	54	65	73	74	41	72	72	61	79	07	00	04	01	00	10	6A	TestArray.....j
000020	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	ava/lang/Object.
000030	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	00	..<init>...()V..
000040	04	43	6F	64	65	0A	00	03	00	09	0C	00	05	00	06	01	.Code.....
000050	00	0F	4C	69	6E	65	4E	75	6D	62	65	72	54	61	62	6C	..LineNumberTabl
000060	65	01	00	12	4C	6F	63	61	6C	56	61	72	69	61	62	6C	e...LocalVariabl
000070	65	54	61	62	6C	65	01	00	04	74	68	69	73	01	00	0B	eTable...this...
000080	4C	54	65	73	74	41	72	72	61	79	3B	01	00	04	6D	61	LTestArray;...ma
000090	69	6E	01	00	16	28	5B	4C	6A	61	76	61	2F	6C	61	6E	in...([Ljava/lang
0000A0	67	2F	53	74	72	69	6E	67	3B	29	56	07	00	11	01	00	g/String;)V....
0000B0	02	5B	49	09	00	13	00	15	07	00	14	01	00	10	6A	61	.[I.....ja
0000C0	76	61	2F	6C	61	6E	67	2F	53	79	73	74	65	6D	0C	00	va/lang/System..
0000D0	16	00	17	01	00	03	6F	75	74	01	00	15	4C	6A	61	76out...Ljav
0000E0	61	2F	69	6F	2F	50	72	69	6E	74	53	74	72	65	61	6D	a/io/PrintStream
0000F0	3B	08	00	19	01	00	07	65	6C	65	6D	65	6E	74	0A	00	;.....element..
000100	1B	00	1D	07	00	1C	01	00	13	6A	61	76	61	2F	69	6Fjava/io
000110	2F	50	72	69	6E	74	53	74	72	65	61	6D	0C	00	1E	00	/PrintStream....
000120	1F	01	00	07	70	72	69	6E	74	6C	6E	01	00	15	28	4Cprintln...(L
000130	6A	61	76	61	2F	6C	61	6E	67	2F	53	74	72	69	6E	67	java/lang/String

We can see a lot of opcode(e.g. CA, 4C, etc) in the bytecode above, each of them has a corresponding mnemonic code (e.g., aload_0 in the example below). The opcode is not readable, but we can use javap to see the mnemonic form of a .class file.

"javap -c" prints out disassembled code for each method in the class. Disassembled code means the instructions that comprise the Java bytecodes.

```
javap -classpath . -c HelloWorld
```

Compiled from "HelloWorld.java"

```
public class HelloWorld extends java.lang.Object{
public HelloWorld();
```

Code:

```
0: aload_0
```

```
1: invokespecial #1; //Method java/lang/Object."<init>":()V
```

```
4: return
```

```
public static void main(java.lang.String[]);
```

Code:

```
0: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc #3; //String Hello World
5: invokevirtual #4; //Method
    java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
}
```

The code above contains two methods: one is the default constructor, which is inferred by the compiler; the other is the main method.

Below each method, there are a sequence of instructions, such as `aload_0`, `invokeSpecial #1`, etc. What each instruction does can be looked up in [Java bytecode instruction listings](#). For instance, `aload_0` loads a reference onto the stack from local variable 0, `getstatic` fetches a static field value of a class. Notice the "#2" after `getstatic` instruction points to the run-time constant pool. Constant pool is one of the [JVM run-time data areas](#). This leads us to take a look at the constant pool, which can be done by using "javap -verbose" command.

In addition, each instruction starts with a number, such as 0, 1, 4, etc. In the .class file, each method has a corresponding bytecode array. These numbers correspond to the index of the array where each opcode and its parameters are stored. Each opcode is 1 byte long and instructions can have 0 or multiple parameters. That's why these numbers are not consecutive.

Now we can use "javap -verbose" to take a further look of the class.

```
javap -classpath . -verbose HelloWorld
```

```
Compiled from "HelloWorld.java"
public class HelloWorld extends java.lang.Object
  SourceFile: "HelloWorld.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method #6.#15; // java/lang/Object."<init>":()V
const #2 = Field #16.#17; // java/lang/System.out:Ljava/io/PrintStream;
const #3 = String #18; // Hello World
const #4 = Method #19.#20; //
    java/io/PrintStream.println:(Ljava/lang/String;)V
const #5 = class #21; // HelloWorld
const #6 = class #22; // java/lang/Object
const #7 = Asciz <init>;
const #8 = Asciz ()V;
const #9 = Asciz Code;
const #10 = Asciz LineNumberTable;
const #11 = Asciz main;
const #12 = Asciz ([Ljava/lang/String;)V;
const #13 = Asciz SourceFile;
const #14 = Asciz HelloWorld.java;
const #15 = NameAndType #7:#8; // "<init>":()V
const #16 = class #23; // java/lang/System
```



```

const #17 = NameAndType #24:#25;// out:Ljava/io/PrintStream;
const #18 = Asciz Hello World;
const #19 = class #26; // java/io/PrintStream
const #20 = NameAndType #27:#28;// println:(Ljava/lang/String;)V
const #21 = Asciz HelloWorld;
const #22 = Asciz java/lang/Object;
const #23 = Asciz java/lang/System;
const #24 = Asciz out;
const #25 = Asciz Ljava/io/PrintStream;;
const #26 = Asciz java/io/PrintStream;
const #27 = Asciz println;
const #28 = Asciz (Ljava/lang/String;)V;

{
public HelloWorld();
  Code:
    Stack=1, Locals=1, Args_size=1
    0: aload_0
    1: invokespecial #1; //Method java/lang/Object."<init>":()V
    4: return
  LineNumberTable:
    line 2: 0

public static void main(java.lang.String[]);
  Code:
    Stack=2, Locals=1, Args_size=1
    0: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
    3: ldc #3; //String Hello World
    5: invokevirtual #4; //Method
      java/io/PrintStream.println:(Ljava/lang/String;)V
    8: return
  LineNumberTable:
    line 9: 0
    line 10: 8
}

```

From [JVM specification](#): The run-time constant pool serves a function similar to that of a symbol table for a conventional programming language, although it contains a wider range of data than a typical symbol table.

The "#1" in the "invokespecial #1" instruction points to #1 constant in the constant pool. The constant is "Method #6.#15;". From the number, we can get the final constant recursively.

LineNumberTable provides information to a debugger to indicate which line of Java source code corresponds to which byte code instruction. For example, line 9 in the Java source code corresponds to byte code 0 in the main method and line 10 corresponds to byte code 8.

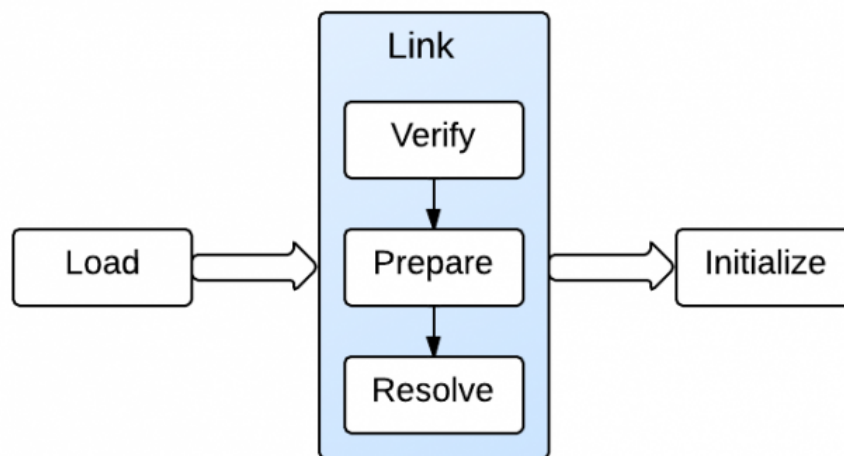
If you want to know more about bytecode, you can create and compile a more

complicated class to take a look. HelloWorld is really a start point of doing this.

18.4. How is it executed in JVM?

Now the question is how JVM loads the class and invoke the main method?

Before the main method is executed, JVM needs to 1) load, 2) link, and 3) initialize the class. 1) Loading brings binary form for a class/interface into JVM. 2) Linking incorporates the binary type data into the run-time state of JVM. Linking consists of 3 steps: verification, preparation, and optional resolution. Verification ensures the class/interface is structurally correct; preparation involves allocating memory needed by the class/interface; resolution resolves symbolic references. And finally 3) initialization assigns the class variables with proper initial values.



This loading job is done by Java Classloaders. When the JVM is started, three class loaders are used:

- Bootstrap class loader: loads the core Java libraries located in the `/jre/lib` directory. It is a part of core JVM, and is written in native code.
- Extensions class loader: loads the code in the extension directories(e.g., `/jar/lib/ext`).
- System class loader: loads code found on CLASSPATH.

So HelloWorld class is loaded by system class loader. When the main method is executed, it will trigger **loading, linking, and initialization of other dependent classes** if they exist.

Finally, the `main()` frame is pushed into the JVM stack, and program counter(PC) is set accordingly. PC then indicates to push `println()` frame to the JVM stack. When the `main()` method completes, it will popped up from the stack and execution is done.

19. Top 10 questions of Java Strings

The following are top 10 frequently asked questions about Java Strings.

19.1. How to compare strings? Use "==" or use equals()?

In brief, "==" tests if references are equal and equals() tests if values are equal. Unless you want to check if two strings are the same object, you should always use equals(). It would be better if you know the concept of [string interning](#).

19.2. Why is char[] preferred over String for security sensitive information?

Strings are [immutable](#), which means once they are created, they will stay unchanged until Garbage Collector kicks in. With an array, you can explicitly change its elements. In this way, security sensitive information(e.g. password) will not be present anywhere in the system.

19.3. Can we use string for switch statement?

Yes to version 7. From [JDK 7](#), we can use string as switch condition. Before version 6, we can not use string as switch condition.

```
// java 7 only!
switch (str.toLowerCase()) {
    case "a":
        value = 1;
        break;
    case "b":
        value = 2;
        break;
}
```

19.4. How to convert string to int?

```
int n = Integer.parseInt("10");
```

Simple, but so frequently used and sometimes ignored.

19.5. How to split a string with white space characters?

```
String[] strArray = aString.split("\\s+");
```

19.6. What substring() method really does?

In JDK 6, the substring() method gives a window to an array of chars which represents the existing String, but do not create a new one. To create a new string represented by a new char array, you can do add an empty string like the following:

```
str.substring(m, n) + ""
```

This will create a new char array that represents the new string. The above approach sometimes can make your code faster, because Garbage Collector can collect the unused large string and keep only the sub string.

In Oracle JDK 7, substring() creates a new char array, not uses the existing one. Check out the diagram for showing substring() difference between JDK 6 and JDK 7.

19.7. String vs StringBuilder vs StringBuffer

String vs StringBuilder: StringBuilder is mutable, which means you can modify it after its creation. StringBuilder vs StringBuffer: StringBuffer is synchronized, which means it is thread-safe but slower than StringBuilder.

19.8. How to repeat a string?

In Python, we can just multiply a number to repeat a string. In Java, we can use the repeat() method of StringUtils from Apache Commons Lang package.

```
String str = "abcd";  
String repeated = StringUtils.repeat(str,3);  
//abcdabcdabcd
```

19.9. How to convert string to date?

```
String str = "Sep 17, 2013";  
Date date = new SimpleDateFormat("MMMM d, yy", Locale.ENGLISH).parse(str);  
System.out.println(date);  
//Tue Sep 17 00:00:00 EDT 2013
```

19.10. . How to count # of occurrences of a character in a string?

Use StringUtils from apache commons lang.

```
int n = StringUtils.countMatches("11112222", "1");  
System.out.println(n);
```

19.11. One more Do you know How to detect if a string contains only uppercase letter?

20. How does Java handle aliasing?

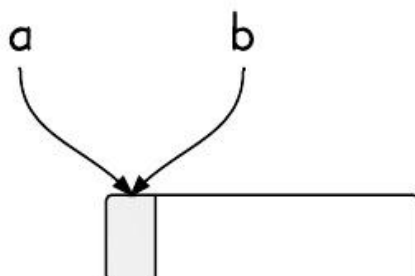
20.1. What is Java aliasing?

Aliasing means there are multiple aliases to a location that can be updated, and these aliases have different types.

In the following example, a and b are two variable names that have two different types A and B. B extends A.

```
B[] b = new B[10];  
A[] a = b;  
  
a[0] = new A();  
b[0].methodParent();
```

In memory, they both refer to the same location.



The pointed memory location are pointed by both a and b. During run-time, the

actual object stored determines which method to call.

20.2. How does Java handle aliasing problem?

If you copy this code to your eclipse, there will be no compilation errors.

```
class A {
    public void methodParent() {
        System.out.println("method in Parent");
    }
}

class B extends A {
    public void methodParent() {
        System.out.println("override method in Child");
    }

    public void methodChild() {
        System.out.println("method in Child");
    }
}

public class Main {

    public static void main(String[] args) {

        B[] b = new B[10];
        A[] a = b;

        a[0] = new A();
        b[0].methodParent();
    }
}
```

But if you run the code, the output would be:

```
Exception in thread "main" java.lang.ArrayStoreException: aliasingtest.A
at aliasingtest.Main.main(Main.java:26)
```

The reason is that Java handles aliasing during run-time. During run-time, it knows that the first element should be a B object, instead of A.

Therefore, it only runs correctly if it is changed to:

```
B[] b = new B[10];
A[] a = b;

a[0] = new B();
b[0].methodParent();
```

and the output is:

override method in Child

21. How Static Type Checking Works in Java?

From Wiki:

Static type-checking is the process of verifying the type safety of a program based on analysis of a program's source code.

Dynamic type-checking is the process of verifying the type safety of a program at runtime

Java uses static type checking to analyze the program during compile-time to prove the absence of type errors. The basic idea is never let bad things happen at runtime. By understanding the following example, you should have a good understanding of how static type checking works in Java.

21.1. Code Example

Suppose we have the following classes, A and B. B extends A.

```
class A {
    A me() {
        return this;
    }

    public void doA() {
        System.out.println("Do A");
    }
}

class B extends A {
    public void doB() {
        System.out.println("Do B");
    }
}
```

First of all, what does "new B().me()" return? An A object or a B object?

The me() method is declared to return an A, so during compile time, compiler only sees it return an A object. However, it actually returns a B object during run-time, since B inherits A's methods and return this(itself).

21.2. How Static Type Checking Works?

The following line will be illegal, even though the object is being invoked on is a B object. The problem is that its reference type is A. Compiler doesn't know its real type during compile-time, so it sees the object as type A.

```
//illegal
new B().me().doB();
```

So only the following method can be invoked.

```
//legal
new B().me().doA();
```

However, we can cast the object to type B, like the following:

```
//legal
((B) new B().me()).doB();
```

If the following C class is added,

```
class C extends A{
    public void doBad() {
        System.out.println("Do C");
    }
}
```

then the following statement is legal and can pass static type checking:

```
//legal
((C) new B().me()).beBad();
```

Compiler does not know it's real time, but runtime will throw a cast exception since B can not be casted to C:

```
java.lang.ClassCastException: B cannot be cast to C
```

22. Interview Question - Use Java Thread to Do Math Calculation

This is an example for showing how to use join(). Interview question: Use Java multi-threading to calculate the expression $1^2/(1+2)$.

Solution:

Use one thread to do addition, one thread to do multiplication, and a main thread to do the division. Since there is no need to communicate data between threads, so only need to consider the order of thread execution.

In the main thread, let addition and multiplication join the main thread. The join() method is used when we want the parent thread waits until the threads which call join() ends. In this case, we want addition and multiplication complete first and then do the division.

```
class Add extends Thread {
    int value;

    public void run() {
        value = 1 + 2;
    }
}

class Mul extends Thread {
    int value;

    public void run() {
        value = 1 * 2;
    }
}

public class Main{
    public static void main(String[] args){
        Add t1 = new Add();
        Mul t2 = new Mul();

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        double n = ((double)t2.value/t1.value);

        System.out.println(n);
    }
}
```

23. Why String is immutable in Java ?

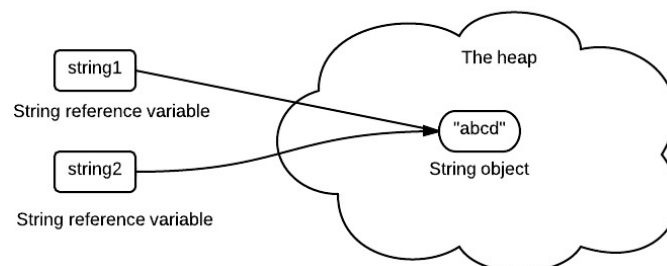
String is an immutable class in Java. An immutable class is simply a class whose instances cannot be modified. All information in an instance is initialized when the instance is created and the information can not be modified. There are many advantages of immutable classes. This article summarizes why [String is designed to be immutable](#). A good answer depends on deep understanding of memory, synchronization, data structures, etc.

23.1. Requirement of String Pool

String pool (String intern pool) is a special storage area in [Method Area](#). 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";
```



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

23.2. Caching Hashcode

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

In String class, it has the following code:

```
private int hash; // this is used to cache hash code.
```

23.3. Facilitating the Use of Other Objects

To make this concrete, consider the following program:

```
HashSet<String> set = new HashSet<String>();
set.add(new String("a"));
set.add(new String("b"));
set.add(new String("c"));

for(String a: set)
    a.value = "a";
```

In this example, if String is mutable, its value can be changed which would violate the design of set (set contains unduplicated elements). This example is designed for simplicity sake, in the real String class there is no value field.

23.4. 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);
}
```

23.5. Immutable objects are naturally thread-safe

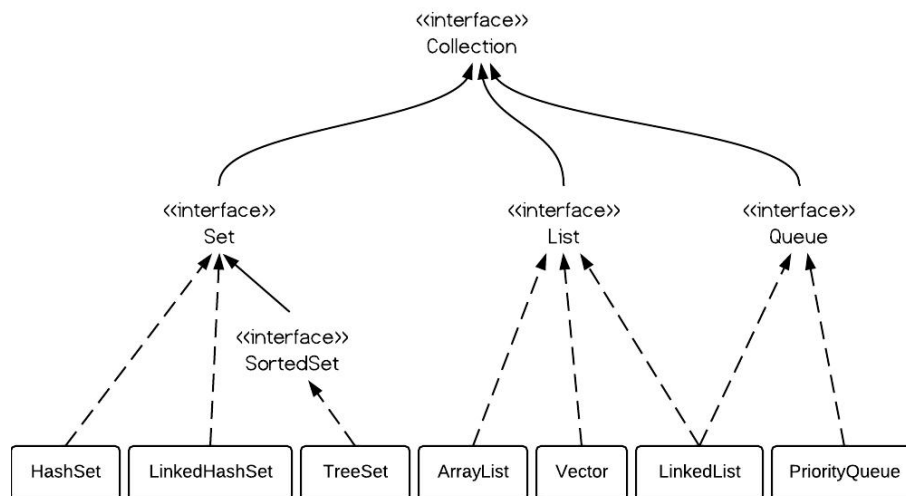
Because immutable objects can not be changed, they can be shared among multiple threads freely. This eliminates the requirements of doing synchronization.

In summary, String is designed to be immutable for the sake of efficiency and security. This is also the reason why immutable classes are preferred in general.

24. ArrayList vs. LinkedList vs. Vector

24.1. List Overview

List, as its name indicates, is an ordered sequence of elements. When we talk about List, it is a good idea to compare it with Set which is a set of unique and unordered elements. The following is the class hierarchy diagram of Collection. From that you can get a general idea of Java Collections.



24.2. ArrayList vs. LinkedList vs. Vector

From the hierarchy diagram, they all implement List interface. They are very similar to use. Their main difference is their implementation which causes different performance for different operations.

`ArrayList` is implemented as a resizable array. As more elements are added to `ArrayList`, its size is increased dynamically. Its elements can be accessed directly by using the `get` and `set` methods, since `ArrayList` is essentially an array.

`LinkedList` is implemented as a double linked list. Its performance on `add` and `remove` is better than `ArrayList`, but worse on `get` and `set` methods.

`Vector` is similar with `ArrayList`, but it is synchronized.

`ArrayList` is a better choice if your program is thread-safe. `Vector` and `ArrayList` require more space as more elements are added. `Vector` each time doubles its array size, while `ArrayList` grow 50

Note: The default initial capacity of an `ArrayList` is pretty small. It is a good habit to construct the `ArrayList` with a higher initial capacity. This can avoid the resizing cost.

24.3. ArrayList example

```
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(3);
al.add(2);
al.add(1);
al.add(4);
al.add(5);
al.add(6);
al.add(6);

Iterator<Integer> iter1 = al.iterator();
while(iter1.hasNext()){
    System.out.println(iter1.next());
}
```

24.4. LinkedList example

```
LinkedList<Integer> ll = new LinkedList<Integer>();
ll.add(3);
ll.add(2);
ll.add(1);
ll.add(4);
ll.add(5);
ll.add(6);
ll.add(6);

Iterator<Integer> iter2 = ll.iterator();
while(iter2.hasNext()){
    System.out.println(iter2.next());
}
```

As shown in the examples above, they are similar to use. The real difference is their underlying implementation and their operation complexity.

24.5. Vector

Vector is almost identical to ArrayList, and the difference is that Vector is synchronized. Because of this, it has an overhead than ArrayList. Normally, most Java programmers use ArrayList instead of Vector because they can synchronize explicitly by themselves.

24.6. Performance of ArrayList vs. LinkedList

	ArrayList	LinkedList
get()	O(1)	O(n)
add()	O(1)	O(1) amortized
remove()	O(n)	O(n)

* add() in the table refers to add(E e), and remove() refers to remove(int index)

- ArrayList has O(n) time complexity for arbitrary indices of add/remove, but O(1) for the operation at the end of the list.
- LinkedList has O(n) time complexity for arbitrary indices of add/remove, but O(1) for operations at end/beginning of the List.

I use the following code to test their performance:

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
LinkedList<Integer> linkedList = new LinkedList<Integer>();

// ArrayList add
long startTime = System.nanoTime();

for (int i = 0; i < 100000; i++) {
    arrayList.add(i);
}
long endTime = System.nanoTime();
long duration = endTime - startTime;
System.out.println("ArrayList add: " + duration);

// LinkedList add
startTime = System.nanoTime();

for (int i = 0; i < 100000; i++) {
    linkedList.add(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("LinkedList add: " + duration);

// ArrayList get
startTime = System.nanoTime();

for (int i = 0; i < 10000; i++) {
    arrayList.get(i);
}
endTime = System.nanoTime();
```

```
duration = endTime - startTime;
System.out.println("ArrayList get: " + duration);

// LinkedList get
startTime = System.nanoTime();

for (int i = 0; i < 10000; i++) {
    linkedList.get(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("LinkedList get: " + duration);

// ArrayList remove
startTime = System.nanoTime();

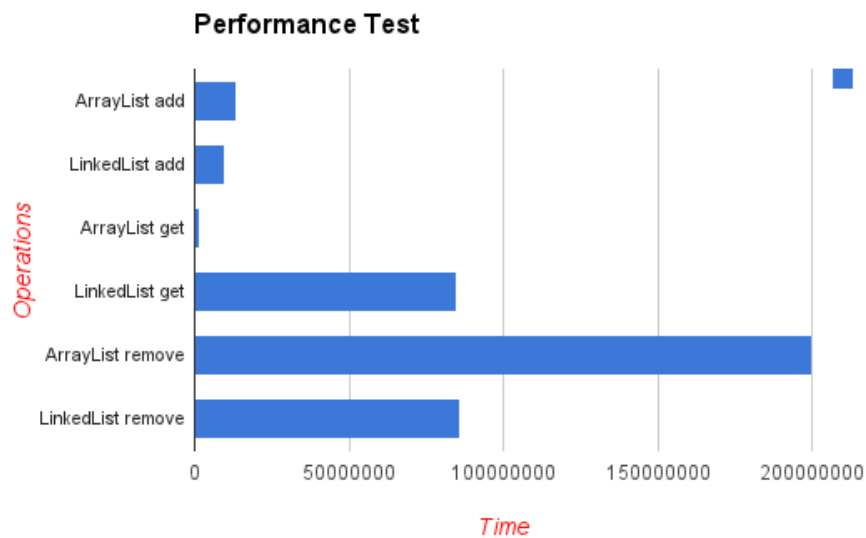
for (int i = 9999; i >=0; i--) {
    arrayList.remove(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("ArrayList remove: " + duration);

// LinkedList remove
startTime = System.nanoTime();

for (int i = 9999; i >=0; i--) {
    linkedList.remove(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("LinkedList remove: " + duration);
```

And the output is:

```
ArrayList add: 13265642
LinkedList add: 9550057
ArrayList get: 1543352
LinkedList get: 85085551
ArrayList remove: 199961301
LinkedList remove: 85768810
```



The difference of their performance is obvious. LinkedList is faster in add and remove, but slower in get. Based on the complexity table and testing results, we can figure out when to use ArrayList or LinkedList. In brief, LinkedList should be preferred if:

- there are no large number of random access of element
- there are a large number of add/remove operations

25. Java Varargs Examples

25.1. What is Varargs in Java?

Varargs (variable arguments) is a feature introduced in Java 1.5. It allows a method take an arbitrary number of values as arguments.

```
public static void main(String[] args) {  
    print("a");  
    print("a", "b");  
    print("a", "b", "c");  
}  
  
public static void print(String ... s){  
    for(String a: s)
```



```
    System.out.println(a);  
}
```

25.2. How Varargs Work?

When varargs facility is used, it actually first creates an array whose size is the number of arguments passed at the call site, then puts the argument values into the array, and finally passes the array to the method.

25.3. When to Use Varargs?

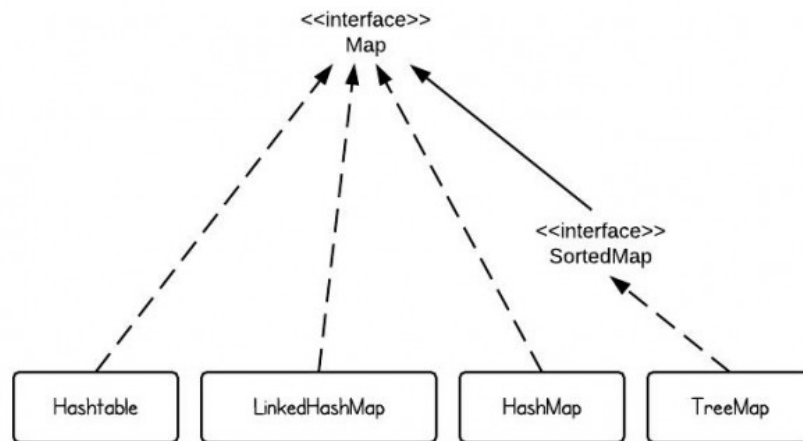
As its definition indicates, varargs is useful when a method needs to deal with an arbitrary number of objects. One good example from Java SDK is `String.format(String format, Object... args)`. The string can format any number of parameters, so varargs is used.

```
String.format("An integer: %d", i);  
String.format("An integer: %d and a string: %s", i, s);
```

26. HashMap vs. TreeMap vs. Hashtable vs. LinkedHashMap

Map is one of the most important data structures. In this tutorial, I will show you how to use different maps such as HashMap, TreeMap, Hashtable and LinkedHashMap.

26.1. Map Overview



There are 4 commonly used implementations of Map in Java SE - HashMap, TreeMap, Hashtable and LinkedHashMap. If we use one sentence to describe each implementation, it would be the following:

- HashMap is implemented as a hash table, and there is no ordering on keys or values.
- TreeMap is implemented based on red-black tree structure, and it is ordered by the key.
- LinkedHashMap preserves the insertion order
- Hashtable is synchronized, in contrast to HashMap.

26.2. HashMap

If key of the HashMap is self-defined objects, then equals() and hashCode() contract need to be followed.

```
class Dog {
    String color;

    Dog(String c) {
        color = c;
    }
    public String toString(){
        return color + " dog";
    }
}
```

```

public class TestHashMap {
    public static void main(String[] args) {
        HashMap<Dog, Integer> hashMap = new HashMap<Dog, Integer>();
        Dog d1 = new Dog("red");
        Dog d2 = new Dog("black");
        Dog d3 = new Dog("white");
        Dog d4 = new Dog("white");

        hashMap.put(d1, 10);
        hashMap.put(d2, 15);
        hashMap.put(d3, 5);
        hashMap.put(d4, 20);

        //print size
        System.out.println(hashMap.size());

        //loop HashMap
        for (Entry<Dog, Integer> entry : hashMap.entrySet()) {
            System.out.println(entry.getKey().toString() + " - " +
                               entry.getValue());
        }
    }
}

```

Output:

```

4
white dog - 5
black dog - 15
red dog - 10
white dog - 20

```

Note here, we add "white dogs" twice by mistake, but the HashMap takes it. This does not make sense, because now we are confused how many white dogs are really there.

The Dog class should be defined as follows:

```

class Dog {
    String color;

    Dog(String c) {
        color = c;
    }

    public boolean equals(Object o) {
        return ((Dog) o).color.equals(this.color);
    }

    public int hashCode() {
        return color.length();
    }
}

```

```
}

public String toString(){
    return color + " dog";
}
}
```

Now the output is:

```
3
red dog - 10
white dog - 20
black dog - 15
```

The reason is that HashMap doesn't allow two identical elements. By default, the hashCode() and equals() methods implemented in Object class are used. The default hashCode() method gives distinct integers for distinct objects, and the equals() method only returns true when two references refer to the same object. Check out the hashCode() and equals() contract if this is not obvious to you.

Check out the [most frequently used methods for HashMap](#), such as iteration, print, etc.

26.3. TreeMap

A TreeMap is sorted by keys. Let's first take a look at the following example to understand the "sorted by keys" idea.

```
class Dog {
    String color;

    Dog(String c) {
        color = c;
    }
    public boolean equals(Object o) {
        return ((Dog) o).color.equals(this.color);
    }

    public int hashCode() {
        return color.length();
    }
    public String toString(){
        return color + " dog";
    }
}

public class TestTreeMap {
    public static void main(String[] args) {
        Dog d1 = new Dog("red");
        Dog d2 = new Dog("black");
    }
}
```

```

Dog d3 = new Dog("white");
Dog d4 = new Dog("white");

TreeMap<Dog, Integer> treeMap = new TreeMap<Dog, Integer>();
treeMap.put(d1, 10);
treeMap.put(d2, 15);
treeMap.put(d3, 5);
treeMap.put(d4, 20);

for (Entry<Dog, Integer> entry : treeMap.entrySet()) {
    System.out.println(entry.getKey() + " - " + entry.getValue());
}
}
}

```

Output:

```

Exception in thread "main" java.lang.ClassCastException: collection.Dog
    cannot be cast to java.lang.Comparable
    at java.util.TreeMap.put(Unknown Source)
    at collection.TestHashMap.main(TestHashMap.java:35)

```

Since TreeMaps are sorted by keys, the object for key has to be able to compare with each other, that's why it has to implement Comparable interface. For example, you use String as key, because String implements Comparable interface.

Let's change the Dog, and make it comparable.

```

class Dog implements Comparable<Dog>{
    String color;
    int size;

    Dog(String c, int s) {
        color = c;
        size = s;
    }

    public String toString(){
        return color + " dog";
    }

    @Override
    public int compareTo(Dog o) {
        return o.size - this.size;
    }
}

public class TestTreeMap {
    public static void main(String[] args) {
        Dog d1 = new Dog("red", 30);
        Dog d2 = new Dog("black", 20);
    }
}

```

```

Dog d3 = new Dog("white", 10);
Dog d4 = new Dog("white", 10);

TreeMap<Dog, Integer> treeMap = new TreeMap<Dog, Integer>();
treeMap.put(d1, 10);
treeMap.put(d2, 15);
treeMap.put(d3, 5);
treeMap.put(d4, 20);

for (Entry<Dog, Integer> entry : treeMap.entrySet()) {
    System.out.println(entry.getKey() + " - " + entry.getValue());
}
}
}

```

Output:

```

red dog - 10
black dog - 15
white dog - 20

```

It is sorted by key, i.e., dog size in this case.

If "Dog d4 = new Dog("white", 10);" is replaced with "Dog d4 = new Dog("white", 40);", the output would be:

```

white dog - 20
red dog - 10
black dog - 15
white dog - 5

```

The reason is that TreeMap now uses compareTo() method to compare keys. Different sizes make different dogs!

26.4. Hashtable

From Java Doc: The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.

26.5. LinkedHashMap

LinkedHashMap is a subclass of HashMap. That means it inherits the features of HashMap. In addition, the linked list preserves the insertion-order.

Let's replace the HashMap with LinkedHashMap using the same code used for HashMap.

```

class Dog {
    String color;

```

```

Dog(String c) {
    color = c;
}

public boolean equals(Object o) {
    return ((Dog) o).color.equals(this.color);
}

public int hashCode() {
    return color.length();
}

public String toString(){
    return color + " dog";
}
}

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

        Dog d1 = new Dog("red");
        Dog d2 = new Dog("black");
        Dog d3 = new Dog("white");
        Dog d4 = new Dog("white");

        LinkedHashMap<Dog, Integer> linkedHashMap = new LinkedHashMap<Dog,
            Integer>();
        linkedHashMap.put(d1, 10);
        linkedHashMap.put(d2, 15);
        linkedHashMap.put(d3, 5);
        linkedHashMap.put(d4, 20);

        for (Entry<Dog, Integer> entry : linkedHashMap.entrySet()) {
            System.out.println(entry.getKey() + " - " + entry.getValue());
        }
    }
}

```

Output is:

```

red dog - 10
black dog - 15
white dog - 20

```

The difference is that if we use HashMap the output could be the following - the insertion order is not preserved.

```

red dog - 10
white dog - 20
black dog - 15

```

Read: [ArrayList](#) vs. [LinkedList](#) vs. [Vector](#)

27. What is Instance Initializer in Java?

In this post, I will first use an example to show what are instance variable initializer, instance initializer and static initializer and then illustrate how the instance initializer works in Java.

27.1. Execution Order

Look at the following class, do you know which one gets executed first?

```
public class Foo {  
  
    //instance variable initializer  
    String s = "abc";  
  
    //constructor  
    public Foo() {  
        System.out.println("constructor called");  
    }  
  
    //static initializer  
    static {  
        System.out.println("static initializer called");  
    }  
  
    //instance initializer  
    {  
        System.out.println("instance initializer called");  
    }  
  
    public static void main(String[] args) {  
        new Foo();  
        new Foo();  
    }  
}
```

Output:

```
static initializer called  
instance initializer called  
constructor called  
instance initializer called  
constructor called
```

27.2. How does Java instance initializer work?

The instance initializer above contains a `println` statement. To understand how it works, we can treat it as a variable assignment statement, e.g., `b = 0`. This can make it more obvious to understand.

Instead of

```
int b = 0
```

, you could write

```
int b;  
b = 0;
```

Therefore, instance initializers and instance variable initializers are pretty much the same.

27.3. When are instance initializers useful?

The use of instance initializers are rare, but still it can be a useful alternative to instance variable initializers if:

(1) initializer code must handle exceptions (2) perform calculations that can't be expressed with an instance variable initializer.

Of course, such code could be written in constructors. But if a class had multiple constructors, you would have to repeat the code in each constructor.

With an instance initializer, you can just write the code once, and it will be executed no matter what constructor is used to create the object. (I guess this is just a concept, and it is not used often.)

Another case in which instance initializers are useful is anonymous inner classes, which can't declare any constructors at all. (Will this be a good place to place a logging function?)

Thanks to Derhein.

Also note that Anonymous classes that implement interfaces [1] have no constructors. Therefore instance initializers are needed to execute any kinds of expressions at construction time.

28. Top 10 Methods for Java Arrays

The following are top 10 methods for Java Array. They are the most voted questions from stackoverflow.

28.1. Declare an array

```
String[] aArray = new String[5];
String[] bArray = {"a", "b", "c", "d", "e"};
String[] cArray = new String[]{"a", "b", "c", "d", "e"};
```

28.2. Print an array in Java

```
int[] intArray = { 1, 2, 3, 4, 5 };
String intArrayString = Arrays.toString(intArray);

// print directly will print reference value
System.out.println(intArray);
// [I@7150bd4d

System.out.println(intArrayString);
// [1, 2, 3, 4, 5]
```

28.3. Create an ArrayList from an array

```
String[] stringArray = { "a", "b", "c", "d", "e" };
ArrayList<String> arrayList = new
    ArrayList<String>(Arrays.asList(stringArray));
System.out.println(arrayList);
// [a, b, c, d, e]
```

28.4. Check if an array contains a certain value

```
String[] stringArray = { "a", "b", "c", "d", "e" };
boolean b = Arrays.asList(stringArray).contains("a");
System.out.println(b);
// true
```

28.5. Concatenate two arrays

```
int[] intArray = { 1, 2, 3, 4, 5 };
int[] intArray2 = { 6, 7, 8, 9, 10 };
// Apache Commons Lang library
int[] combinedIntArray = ArrayUtils.addAll(intArray, intArray2);
```

28.6. Declare an array inline

```
method(new String[]{"a", "b", "c", "d", "e"});
```

28.7. Joins the elements of the provided array into a single String

```
// containing the provided list of elements
// Apache common lang
String j = StringUtils.join(new String[] { "a", "b", "c" }, ", ");
System.out.println(j);
// a, b, c
```

28.8. Convert an ArrayList to an array

```
String[] stringArray = { "a", "b", "c", "d", "e" };
ArrayList<String> arrayList = new
    ArrayList<String>(Arrays.asList(stringArray));
String[] stringArr = new String[arrayList.size()];
arrayList.toArray(stringArr);
for (String s : stringArr)
    System.out.println(s);
```

28.9. Convert an array to a set

```
Set<String> set = new HashSet<String>(Arrays.asList(stringArray));
System.out.println(set);
//[d, e, b, c, a]
```

28.10. Reverse an array

```
int[] intArray = { 1, 2, 3, 4, 5 };
ArrayUtils.reverse(intArray);
System.out.println(Arrays.toString(intArray));
//[5, 4, 3, 2, 1]
```

28.11. . Remove element of an array

```
int[] intArray = { 1, 2, 3, 4, 5 };
int[] removed = ArrayUtils.removeElement(intArray, 3); //create a new array
System.out.println(Arrays.toString(removed));
```

28.12. One more - convert int to byte array

```
byte[] bytes = ByteBuffer.allocate(4).putInt(8).array();

for (byte t : bytes) {
    System.out.format("0x%x ", t);
}
```

29. Java Type Erasure Mechanism

Java Generics is a feature introduced from JDK 5. It allows us to use type parameter when defining class and interface. It is extensively used in Java Collection framework. The type erasure concept is one of the most confusing part about Generics. This article illustrates what it is and how to use it.

29.1. A Common Mistake

In the following example, the method `accept` accepts a list of `Object` as its parameter. In the main method, it is called by passing a list of `String`. Does this work?

```
public class Main {
    public static void main(String[] args) throws IOException {
        ArrayList<String> al = new ArrayList<String>();
        al.add("a");
        al.add("b");

        accept(al);
    }

    public static void accept(ArrayList<Object> al){
        for(Object o: al)
            System.out.println(o);
    }
}
```

It seems fine since Object is a super type of String obviously. However, that will not work. Compilation will not pass, and give you an error at the line of accept(al);

The method accept(ArrayList < Object >) in the type Main is not applicable
for the arguments
(ArrayList < String >)

29.2. List <Object >vs. List <String >

The reason is type erasure. REMEMBER: Java generics is implemented on the compilation level. The byte code generated from compiler does not contain type information of generic type for the run-time execution.

After compilation, both List of Object and List of String become List, and the Object/String type is not visible for JVM. During compilation stage, compiler finds out that they are not the same, then gives a compilation error.

29.3. Wildcards and Bounded Wildcards

List<? >- List can contain any type

```
public static void main(String args[]) {
    ArrayList<Object> al = new ArrayList<Object>();
    al.add("abc");
    test(al);
}

public static void test(ArrayList<?> al){
    for(Object e: al){//no matter what type, it will be Object
        System.out.println(e);
    }
}
// in this method, because we don't know what type ? is, we can not add
// anything to al.
```

Always remember that Generic is a concept of compile-time. In the example above, since we don't know ?, we can not add anything to al. To make it work, you can use wildcards.

List< Object > - List can contain Object or it's subtype

List< ? extends Number > - List can contain Number or its subtypes.
List< ? super Number > - List can contain Number or its supertypes.

29.4. Comparisons

Now we know that `ArrayList<String>` is NOT a subtype of `ArrayList<Object>`. As a comparison, you should know that if two generic types have the same parameter, their inheritance relation is true for the types. For example, `ArrayList<String>` is subtype of `Collection<String>`.

Arrays are different. They know and enforce their element types at runtime. This is called reification. For example, `Object[] objArray` is a super type of `String[] strArr`. If you try to store a `String` into an array of integer, you will get an `ArrayStoreException` during run-time.

30. Simple Java - Foreword



Many people believe in "no diagram no talk". While visualization is a good way to understand and remember things, there are also other ways to enhance learning experience. One is by comparing different but related concepts. For example, by comparing `ArrayList` with `LinkedList`, one can better understand them and use them properly. Another way is to look at the frequently asked questions. For example, by reading "Top 10 methods for Java arrays", one can quickly remember some useful methods and use the methods used by the majority.

There are numerous blogs, books and tutorials available to learn Java. A lot of them receive large traffic from developers with a wide variety of interests. Program Creek is just one of them. This collection might be useful for two kinds of developers: first, the regular visitors of Program Creek; second, developers who want to read something in a more readable format. Repetition is key of learning any programming languages. Hopefully, this contributes another non-boring repetition for you.

Since this collection is 100

31. Top 10 Mistakes Java Developers Make

This list summarizes the top 10 mistakes that Java developers frequently make.

31.1. #1. Convert Array to ArrayList

To convert an array to an ArrayList, developers often do this:

```
List<String> list = Arrays.asList(arr);
```

Arrays.asList() will return an ArrayList which is a private static class inside Arrays, it is not the java.util.ArrayList class. The java.util.Arrays.ArrayList class has set(), get(), contains() methods, but does not have any methods for adding elements, so its size is fixed. To create a real ArrayList, you should do:

```
ArrayList<String> arrayList = new ArrayList<String>(Arrays.asList(arr));
```

The constructor of ArrayList can accept a Collection type, which is also a super type for java.util.Arrays.ArrayList.

31.2. #2. Check If an Array Contains a Value

Developers often do:

```
Set<String> set = new HashSet<String>(Arrays.asList(arr));  
return set.contains(targetValue);
```

The code works, but there is no need to convert a list to set first. Converting a list to a set requires extra time. It can be as simple as:

```
Arrays.asList(arr).contains(targetValue);
```

or

```
for(String s: arr){  
    if(s.equals(targetValue))  
        return true;  
}  
return false;
```

The first one is more readable than the second one.

31.3. #3. Remove an Element from a List Inside a Loop

Consider the following code which removes elements during iteration:

```
ArrayList<String> list = new ArrayList<String>(Arrays.asList("a", "b", "c",  
    "d"));  
for (int i = 0; i < list.size(); i++) {  
    list.remove(i);  
}  
System.out.println(list);
```

The output is:

[b, d]

There is a serious problem in this method. When an element is removed, the size of the list shrinks and the index changes. So if you want to delete multiple elements inside a loop by using the index, that will not work properly.

You may know that using iterator is the right way to delete elements inside loops, and you know foreach loop in Java works like an iterator, but actually it is not. Consider the following code:

```
ArrayList<String> list = new ArrayList<String>(Arrays.asList("a", "b", "c",  
    "d"));  
  
for (String s : list) {  
    if (s.equals("a"))  
        list.remove(s);  
}
```

It will throw out [ConcurrentModificationException](#).

Instead the following is OK:

```
ArrayList<String> list = new ArrayList<String>(Arrays.asList("a", "b", "c",  
    "d"));  
Iterator<String> iter = list.iterator();  
while (iter.hasNext()) {  
    String s = iter.next();  
  
    if (s.equals("a")) {  
        iter.remove();  
    }  
}
```

`.next()` must be called before `.remove()`. In the foreach loop, compiler will make the `.next()` called after the operation of removing element, which caused the `ConcurrentModificationException`. You may want to take a look at the source code of [ArrayList.iterator\(\)](#).

31.4. #4. Hashtable vs HashMap

By conventions in algorithm, Hashtable is the name of the data structure. But in Java, the data structure's name is HashMap. One of the key differences between Hashtable and HashMap is that Hashtable is synchronized. So very often you don't need Hashtable, instead HashMap should be used.

[HashMap vs. TreeMap vs. Hashtable vs. LinkedHashMap Top 10 questions about Map](#)

31.5. #5. Use Raw Type of Collection

In Java, raw type and unbounded wildcard type are easy to mixed together. Take Set for example, Set is raw type, while Set<?> is unbounded wildcard type.

Consider the following code which uses a raw type List as a parameter:

```
public static void add(List list, Object o){
    list.add(o);
}
public static void main(String[] args){
    List<String> list = new ArrayList<String>();
    add(list, 10);
    String s = list.get(0);
}
```

This code will throw an exception:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer
    cannot be cast to java.lang.String
    at ...
```

Using raw type collection is dangerous as the raw type collections skip the generic type checking and not safe. There are huge differences between Set, Set<?>, and Set<Object>. Check out [Raw type vs. Unbounded wildcard](#) and [Type Erasure](#).

31.6. #6. Access Level

Very often developers use public for class field. It is easy to get the field value by directly referencing, but this is a very bad design. The rule of thumb is giving access level for members as low as possible.

[public, default, protected, and private](#)

31.7. #7. ArrayList vs. LinkedList

When developers do not know the difference between ArrayList and LinkedList, they often use ArrayList, because it looks familiar. However, there is a huge performance difference between them. In brief, LinkedList should be preferred if there are a large

number of add/remove operations and there are not a lot of random access operations. Check out [ArrayList vs. LinkedList](#) to get more information about their performance if this is new to you.

31.8. #8. Mutable vs. Immutable

Immutable objects have many advantages such simplicity, safety, etc. But it requires a separate object for each distinct value, and too many objects might cause high cost of garbage collection. There should be a balance when choosing between mutable and immutable.

In general, mutable objects are used to avoid producing too many intermediate objects. One classic example is concatenating a large number of strings. If you use an immutable string, you would produce a lot of objects that are eligible for garbage collection immediately. This wastes time and energy on the CPU, using a mutable object the right solution (e.g. `StringBuilder`).

```
String result="";
for(String s: arr){
    result = result + s;
}
```

There are other situations when mutable objects are desirable. For example passing mutable objects into methods lets you collect multiple results without jumping through too many syntactic hoops. Another example is sorting and filtering: of course, you could make a method that takes the original collection, and returns a sorted one, but that would become extremely wasteful for larger collections. (From [dasblinkenlight's answer](#) on Stack Overflow)

[Why String is Immutable??](#)

31.9. #9. Constructor of Super and Sub

```
class Super {
    String s;

    public Super(String s) {
        this.s = s;
    }
}

public class Sub extends Super {
    int x = 200;
    public Sub(String s) {
    }

    public Sub(){
        System.out.println("Sub");
    }

    public static void main(String[] args){
        Sub s = new Sub();
    }
}
```

This compilation error occurs because the default super constructor is undefined. In Java, if a class does not define a constructor, compiler will insert a default no-argument constructor for the class by default. If a constructor is defined in Super class, in this case Super(String s), compiler will not insert the default no-argument constructor. This is the situation for the Super class above.

The constructors of the Sub class, either with-argument or no-argument, will call the no-argument Super constructor. Since compiler tries to insert super() to the 2 constructors in the Sub class, but the Super's default constructor is not defined, compiler reports the error message.

To fix this problem, simply 1) add a Super() constructor to the Super class like

```
public Super(){
    System.out.println("Super");
}
```

, or 2) remove the self-defined Super constructor, or 3) add super(value) to sub constructors.

[Constructor of Super and Sub](#)

31.10. #10. "" or Constructor?

String can be created by two ways:

```
//1. use double quotes
```

```
String x = "abc";  
//2. use constructor  
String y = new String("abc");
```

What is the difference?

The following examples can provide a quick answer:

```
String a = "abcd";  
String b = "abcd";  
System.out.println(a == b); // True  
System.out.println(a.equals(b)); // True
```

```
String c = new String("abcd");  
String d = new String("abcd");  
System.out.println(c == d); // False  
System.out.println(c.equals(d)); // True
```

For more details about how they are allocated in memory, check out [Create Java String Using " " or Constructor??.](#)

31.11. Future Work

The list is based on my analysis of a large number of open source projects on GitHub, Stack Overflow questions, and popular Google queries. There is no evaluation to prove that they are precisely the top 10, but definitely they are very common. Please leave your comment, if you don't agree with any part. I would really appreciate it if you could point out some other mistakes that are more common.

32. How to make a method thread-safe in Java?

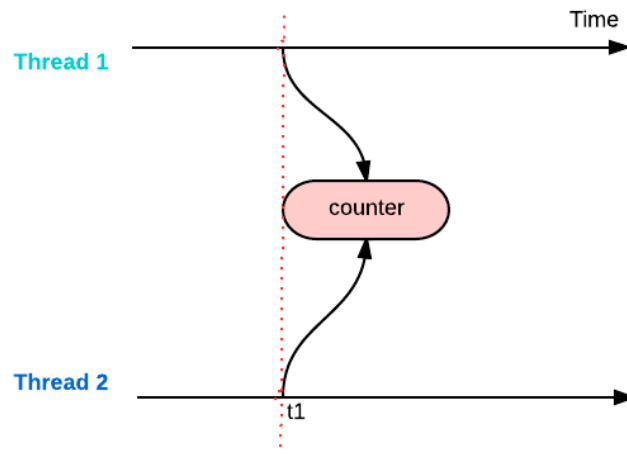
Interview Question:

Is the following method thread-safe? How to make it thread-safe?

```
class MyCounter {  
    private static int counter = 0;  
  
    public static int getCount() {  
        return counter++;  
    }  
}
```

This post explains a general interview question that has been asked by Google and a lot of companies. It's low-level and not about how to design concurrent program.

First of all, the answer is NO. The method is not thread-safe, because the counter++ operation is not atomic, which means it consists more than one atomic operations. In this case, one is accessing value and the other is increasing the value by one.



When Thread 1 accesses the method at t1, Thread 2 may not be done with the method. So the value returned to Thread 1 is the value that has not been increased.

32.1. Make a method thread-safe - Method 1

Adding synchronized to this method will makes it thread-safe. When synchronized is added to a static method, the Class object is the object which is locked.

Is marking it synchronized enough? The answer is YES.

```
class MyCounter {  
    private static int counter = 0;  
  
    public static synchronized int getCount() {  
        return counter++;  
    }  
}
```

If the method is not static, then adding synchronized keyword will synchronize the instance of the class, not the Class object.

32.2. Make a method thread-safe - Method 2

In this particular counter example, we actually can make count++ atomic by using AtomicInteger from the package "java.util.concurrent.atomic".

```
import java.util.concurrent.atomic.AtomicInteger;  
  
public class MyCounter {
```

```
private static AtomicInteger counter = new AtomicInteger(0);

public static int getCount() {
    return counter.getAndIncrement();
}
}
```

32.3. Some other useful facts about thread-safe

Local variables are thread safe in Java.

Each thread has its own stack. Two different threads never shares the same stack. All local variables defined in a method will be allocated memory in stack. As soon as method execution is completed by the current thread, stack frame will be removed.

33. What Is Inner Interface in Java?

33.1. What is Inner Interface in Java?

Inner interface is also called nested interface, which means declare an interface inside of another interface. For example, the Entry interface is declared in the Map interface.

```
public interface Map {
    interface Entry{
        int getKey();
    }

    void clear();
}
```

33.2. Why Use Inner Interface?

There are several compelling reasons for using inner interface:

- It is a way of logically grouping interfaces that are only used in one place.
- It increases encapsulation.
- Nested interfaces can lead to more readable and maintainable code.

One example of inner interface used in java standard library is `java.util.Map` and `Java.util.Map.Entry`. Here `java.util.Map` is used also as a namespace. `Entry` does not

belong to the global scope, which means there are many other entities that are Entries and are not necessary Map's entries. This indicates that Entry represents entries related to the Map.

33.3. How Inner Interface Works?

To figure out how inner interface works, we can compare it with nested classes. Nested classes can be considered as a regular method declared in outer class. Since a method can be declared as static or non-static, similarly nested classes can be static and non-static. Static class is like a static method, it can only access outer class members through objects. Non-static class can access any member of the outer class.

```
public class OuterClass {
    private int x;

    static class StaticInnerClass {
        void InnerMethod() {
            // like a static method,
            // can access variables in outer class through object
            System.out.println(new OuterClass().x);
        }
    }

    class NonStaticInnerClass {
        void InnerMethod() {
            // like a non-static method,
            // can access variables in outer class directly
            System.out.println(x);
        }
    }
}

public class OuterClass2 {
    private int x;

    static void Method1() {
        System.out.println(new OuterClass2().x);
    }

    void Method2() {
        System.out.println(x);
    }
}
```

```
graph TD
    OuterClass --> StaticInnerClass
    OuterClass --> NonStaticInnerClass
    OuterClass2 --> Method1
    OuterClass2 --> Method2
```

Because an interface can not be instantiated, the inner interface only makes sense if it is static. Therefore, by default inner interface is static, no matter you manually add

static or not.

33.4. A Simple Example of Inner Interface?

Map.java

```
public interface Map {  
    interface Entry{  
        int getKey();  
    }  
  
    void clear();  
}
```

MapImpl.java

```
public class MapImpl implements Map {  
  
    class ImplEntry implements Map.Entry{  
        public int getKey() {  
            return 0;  
        }  
    }  
  
    @Override  
    public void clear() {  
        //clear  
    }  
}
```

34. Top 10 questions about Java Collections

The following are the most popular questions of Java collections asked and discussed on Stackoverflow. Before you look at those questions, it's a good idea to see the [class hierarchy diagram](#).

34.1. When to use LinkedList over ArrayList?

[ArrayList](#) is essentially an array. Its elements can be accessed directly by index. But if the array is full, a new larger array is needed to allocate and moving all elements to

the new array will take $O(n)$ time. Also adding or removing an element needs to move existing elements in an array. This might be the most disadvantage to use [ArrayList](#).

[LinkedList](#) is a double linked list. Therefore, to access an element in the middle, it has to search from the beginning of the list. On the other hand, adding and removing an element in [LinkedList](#) is quicker, because it only changes the list locally.

In summary, the worst case of time complexity comparison is as follows:

	ArrayList	LinkedList

get(index)	$O(1)$	$O(n)$
add(E)	$O(n)$	$O(1)$
add(E, index)	$O(n)$	$O(n)$
remove(index)	$O(n)$	$O(n)$
Iterator.remove()	$O(n)$	$O(1)$
Iterator.add(E)	$O(n)$	$O(1)$

Despite the running time, memory usage should be considered too especially for large lists. In [LinkedList](#), every node needs at least two extra pointers to link the previous and next nodes; while in [ArrayList](#), only an array of elements is needed.

[More comparisons between list.](#)

34.2. Efficient equivalent for removing elements while iterating the Collection

The only correct way to modify a collection while iterating is using [Iterator.remove\(\)](#). For example,

```
Iterator<Integer> itr = list.iterator();
while(itr.hasNext()) {
    // do something
    itr.remove();
}
```

One most frequent incorrect code is

```
for(Integer i: list) {
    list.remove(i);
}
```

You will get a [ConcurrentModificationException](#) by running the above code. This is because an iterator has been generated (in for statement) to traverse over the list, but at the same time the list is changed by [Iterator.remove\(\)](#). In Java, "it is not generally permissible for one thread to modify a collection while another thread is iterating over it."

34.3. How to convert List to int[]?

The easiest way might be using [ArrayUtils](#) in [Apache Commons Lang](#) library.

```
int[] array = ArrayUtils.toPrimitive(list.toArray(new Integer[0]));
```

In JDK, there is no short-cut. Note that you can not use `List.toArray()`, because that will convert List to `Integer[]`. The correct way is following,

```
int[] array = new int[list.size()];
for(int i=0; i < list.size(); i++) {
    array[i] = list.get(i);
}
```

34.4. How to convert int[] into List?

The easiest way might still be using [ArrayUtils](#) in [Apache Commons Lang](#) library, like below.

```
List list = Arrays.asList(ArrayUtils.toObject(array));
```

In JDK, there is no short-cut either.

```
int[] array = {1,2,3,4,5};
List<Integer> list = new ArrayList<Integer>();
for(int i: array) {
    list.add(i);
}
```

34.5. What is the best way to filter a Collection?

Again, you can use third-party package, like [Guava](#) or [Apache Commons Lang](#) to fulfil this function. Both provide a `filter()` method (in [Collections2](#) of [Guava](#) and in [CollectionUtils](#) of [Apache](#)). The `filter()` method will return elements that match a given Predicate.

In JDK, things become harder. A good news is that in Java 8, [Predicate](#) will be added. But for now you have to use [Iterator](#) to traverse the whole collection.

```
Iterator<Integer> itr = list.iterator();
while(itr.hasNext()) {
    int i = itr.next();
    if (i > 5) { // filter all ints bigger than 5
        itr.remove();
    }
}
```

Of course you can mimic the way of what Guava and Apache did, by introducing a new interface Predicate. That might also be what most advanced developers will do.

```
public interface Predicate<T> {
    boolean test(T o);
}

public static <T> void filter(Collection<T> collection, Predicate<T>
    predicate) {
    if ((collection != null) && (predicate != null)) {
        Iterator<T> itr = collection.iterator();
        while(itr.hasNext()) {
            T obj = itr.next();
            if (!predicate.test(obj)) {
                itr.remove();
            }
        }
    }
}
```

Then we can use the following code to filter a collection:

```
filter(list, new Predicate<Integer>() {
    public boolean test(Integer i) {
        return i <= 5;
    }
});
```

34.6. Easiest way to convert a List to a Set?

There are two ways to do so, depending on how you want equal defined. The first piece of code puts a list into a [HashSet](#). Duplication is then identified mostly by `hashCode()`. In most cases, it will work. But if you need to specify the way of comparison, it is better to use the second piece of code where you can define your own comparator.

```
Set<Integer> set = new HashSet<Integer>(list);
```

```
Set<Integer> set = new TreeSet<Integer>(aComparator);
set.addAll(list);
```

34.7. How do I remove repeated elements from ArrayList?

This question is quite related to the above one. If you don't care the ordering of the elements in the [ArrayList](#), a clever way is to put the list into a set to remove duplication, and then to move it back to the list. Here is the code

```
ArrayList<> list = ... // initial a list with duplicate elements
Set<Integer> set = new HashSet<Integer>(list);
list.clear();
list.addAll(set);
```

If you DO care about the ordering, order can be preserved by putting a list into a `LinkedHashSet` which is in the standard JDK.

34.8. Sorted collection

There are a couple of ways to maintain a sorted collection in Java. All of them provide a collection in natural ordering or by the specified comparator. By natural ordering, you also need to implement the `Comparable` interface in the elements.

- `Collections.sort()` can sort a `List`. As specified in the javadoc, this sort is stable, and guarantee $n \log(n)$ performance.
- `PriorityQueue` provides an ordered queue. The difference between `PriorityQueue` and `Collections.sort()` is that, `PriorityQueue` maintain an order queue at all time, but you can only get the head element from the queue. You can not randomly access its element like `PriorityQueue.get(4)`.
- If there is no duplication in the collection, `TreeSet` is another choice. Same as `PriorityQueue`, it maintains the ordered set at all time. You can get the lowest and highest elements from the `TreeSet`. But you still cannot randomly access its element.

In a short, `Collections.sort()` provides a one-time ordered list. `PriorityQueue` and `TreeSet` maintain ordered collections at all time, in the cost of no indexed access of elements.

34.9. `Collections.emptyList()` vs new instance

The same question applies to `emptyMap()` and `emptySet()`.

Both methods return an empty list, but `Collections.emptyList()` returns a list that is immutable. This mean you cannot add new elements to the "empty" list. At the background, each call of `Collections.emptyList()` actually won't create a new instance of an empty list. Instead, it will reuse the existing empty instance. If you are familiar `Singleton` in the design pattern, you should know what I mean. So this will give you better performance if called frequently.

34.10. `Collections.copy`

There are two ways to copy a source list to a destination list. One way is to use `ArrayList` constructor

```
ArrayList<Integer> dstList = new ArrayList<Integer>(srcList);
```

The other is to use `Collections.copy()` (as below). Note the first line, we allocate a list at least as long as the source list, because in the javadoc of `Collections`, it says The destination list must be at least as long as the source list.

```
ArrayList<Integer> dstList = new ArrayList<Integer>(srcList.size());  
Collections.copy(dstList, srcList);
```

Both methods are shallow copy. So what is the difference between these two methods?

- First, `Collections.copy()` won't reallocate the capacity of `dstList` even if `dstList` does not have enough space to contain all `srcList` elements. Instead, it will throw an `IndexOutOfBoundsException`. One may question if there is any benefit of it. One reason is that it guarantees the method runs in linear time. Also it makes suitable when you would like to reuse arrays rather than allocate new memory in the constructor of `ArrayList`.
- `Collections.copy()` can only accept `List` as both source and destination, while `ArrayList` accepts `Collection` as the parameter, therefore more general.

35. Set vs. Set<?>

You may know that an unbounded wildcard `Set<?>` can hold elements of any type, and a raw type `Set` can also hold elements of any type. What is the difference between them?

35.1. Two facts about Set<?>

There are two facts about `Set<?>`: Item 1: Since the question mark `?` stands for any type. `Set<?>` is capable of holding any type of elements. Item 2: Because we don't know the type of `?`, we can't put any element into `Set<?>`

So a `Set<?>` can hold any type of element (Item 1), but we can't put any element into it (Item 2). Do the two statements conflict to each other? Of course they are not. This can be clearly illustrated by the following two examples:

Item 1 means the following situation:

```
//Legal Code  
public static void main(String[] args) {  
    HashSet<Integer> s1 = new HashSet<Integer>(Arrays.asList(1, 2, 3));  
    printSet(s1);  
}
```

```

HashSet<String> s2 = new HashSet<String>(Arrays.asList("a", "b", "c"));
printSet(s2);
}

public static void printSet(Set<?> s) {
    for (Object o : s) {
        System.out.println(o);
    }
}

```

Since Set<?> can hold any type of elements, we simply use Object in the loop.
Item 2 means the following situation which is illegal:

```

//Illegal Code
public static void printSet(Set<?> s) {
    s.add(10); //this line is illegal
    for (Object o : s) {
        System.out.println(o);
    }
}

```

Because we don't know the type of <?> exactly, we can not add any thing to it other than null. For the same reason, we can not initialize a set with Set<?>. The following is illegal:

```

//Illegal Code
Set<?> set = new HashSet<?>();

```

35.2. Set vs. Set<?>

What's the difference between raw type Set and unbounded wildcard Set<?>?

This method declaration is fine:

```

public static void printSet(Set s) {
    s.add("2");
    for (Object o : s) {
        System.out.println(o);
    }
}

```

because raw type has no restrictions. However, this will easily corrupt the invariant of collection.

In brief, wildcard type is safe and the raw type is not. We can not put any element into a Set<?>.

35.3. When Set<?> is useful?

When you want to use a generic type, but you don't know or care what the actual type the parameter is, you can use <?>[1]. It can only be used as parameters for a method.

For example:

```
public static void main(String[] args) {
    HashSet<Integer> s1 = new HashSet<Integer>(Arrays.asList(1,2,3));
    HashSet<Integer> s2 = new HashSet<Integer>(Arrays.asList(4,2,3));

    System.out.println(getUnion(s1, s2));
}

public static int getUnion(Set<?> s1, Set<?> s2){
    int count = s1.size();
    for(Object o : s2){
        if(!s1.contains(o)){
            count++;
        }
    }
    return count;
}
```

36. How to Convert Array to ArrayList in Java?

This article analyzes answers for a top-voted questions on Stack Overflow. The person who asked this question got a lot of reputation points, which could grant him permissions to do a lot of things on Stack Overflow. This does not make sense to me, but let's take a look at the question first.

The question is "how to convert the following array to an ArrayList?".

```
Element[] array = {new Element(1),new Element(2),new Element(3)};
```

36.1. Most popular and accepted answer

The most popular and the accepted answer is the following:

```
ArrayList<Element> arrayList = new ArrayList<Element>(Arrays.asList(array));
```

First, let's take a look at the Java Doc for the constructor method of ArrayList.

ArrayList(Collection <? extends E >c) : Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

So what the constructor does is the following: 1. Convert the collection c to an array
2. Copy the array to ArrayList's own back array called "elementData"

Here is the source code of Constructor of ArrayList.

```
public ArrayList(Collection<? extends E> c) {
    elementData = c.toArray();
    size = elementData.length;

    if (elementData.getClass() != Object[].class)
        elementData = Arrays.copyOf(elementData, size, Object[].class);
}
```

36.2. Next popular answer

The next popular answer is:

```
List<Element> list = Arrays.asList(array);
```

It is not the best, because the size of the list returned from asList() is fixed. Actually the list returned is not java.util.ArrayList, but a private static class defined inside java.util.Arrays. We know ArrayList is essentially implemented as an array, and the list returned from asList() is a fixed-size list backed by the original array. In this way, if add or remove elements from the returned list, an UnsupportedOperationException will be thrown.

```
list.add(new Element(4));
```

```
Exception in thread "main" java.lang.ClassCastException:
    java.util.Arrays$ArrayList cannot be cast to java.util.ArrayList
    at collection.ConvertArray.main(ConvertArray.java:22)
```

36.3. Another Solution

This solution is from Otto's comment below.

```
Element[] array = {new Element(1), new Element(2)};
List<element> list = new ArrayList<element>(array.length);
Collections.addAll(list, array);
```

36.4. Indications of the question

The problem is not hard, but interesting. Every Java programmer knows ArrayList, but it's easy to make such a mistake. I guess that is why this question is so popular. If a similar question asked about a Java library in a specific domain, it would be less likely to become so popular.

There are several answers that provide the same solution. This is also true for a lot of other questions on Stack Overflow, I guess people just don't care what others say if they would like to answer a question!

37. Java read a file line by line - How Many Ways?

The number of total classes of Java I/O is large, and it is easy to get confused when to use which. The following are two methods for reading a file line by line.

Method 1:

```
private static void readFile1(File fin) throws IOException {
    FileInputStream fis = new FileInputStream(fin);

    //Construct BufferedReader from InputStreamReader
    BufferedReader br = new BufferedReader(new InputStreamReader(fis));

    String line = null;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }

    br.close();
}
```

Method 2:

```
private static void readFile2(File fin) throws IOException {
    // Construct BufferedReader from FileReader
    BufferedReader br = new BufferedReader(new FileReader(fin));

    String line = null;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }

    br.close();
}
```

Use the following code:

```
//use . to get current directory
File dir = new File(".");
File fin = new File(dir.getCanonicalPath() + File.separator + "in.txt");

readFile1(fin);
readFile2(fin);
```

Both works for reading a text file line by line.

The difference between the two methods is what to use to construct a `BufferedReader`. Method 1 uses `InputStreamReader` and Method 2 uses `FileReader`. What's the difference between the two classes?

From Java Doc, "An `InputStreamReader` is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset." `InputStreamReader` can handle other input streams than files, such as network connections, classpath resources, ZIP files, etc.

`FileReader` is "Convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate." `FileReader` does not allow you to specify an encoding other than the platform default encoding. Therefore, it is not a good idea to use it if the program will run on systems with different platform encoding.

In summary, `InputStreamReader` is always a safer choice than `FileReader`.

It is worth to mention here that instead of using a concrete / or for a path, you should always use `File.separator` which can ensure that the separator is always correct for different operating systems. Also the path used should be relative, and that ensures the path is always correct.

Update: You can also use the following method which is available since Java 1.7. Essentially, it is the same with Method 1.

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s\n", x);
}
```

The `newBufferedReader` method does the following:

```
public static BufferedReader newBufferedReader(Path path, Charset cs){
    CharsetDecoder decoder = cs.newDecoder();
    Reader reader = new InputStreamReader(newInputStream(path), decoder);
    return new BufferedReader(reader);
}
```

Reading the class hierarchy diagram is also very helpful for understanding those inputstream and reader related concept: <http://www.programcreek.com/2012/05/java-io-class-hierarchy-diagram/>.

38. Yet Another “Java Passes By Reference or By Value”?

This is a classic interview question which confuses novice Java developers. In this post I will use an example and some diagram to demonstrate that: Java is pass-by-value.

38.1. Some Definitions

Pass by value: make a copy in memory of the actual parameter's value that is passed in. Pass by reference: pass a copy of the address of the actual parameter.

Java is always pass-by-value. Primitive data types and object reference are just values.

38.2. Passing Primitive Type Variable

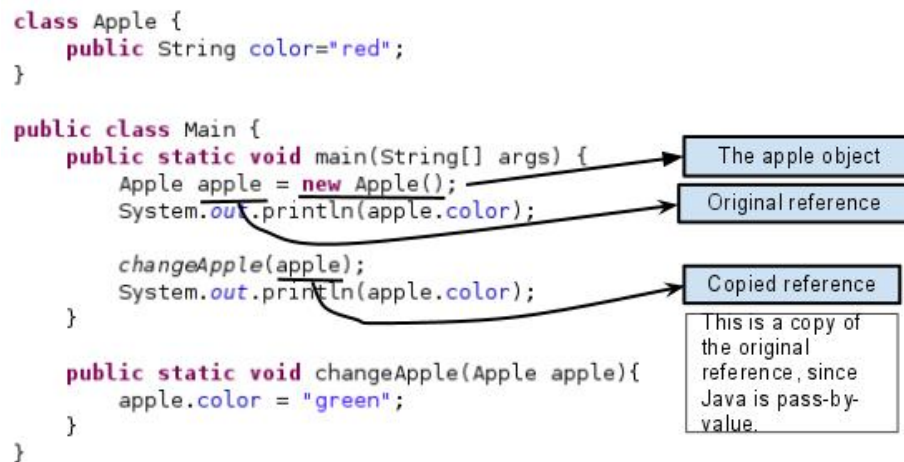
Since Java is pass-by-value, it's not hard to understand the following code will not swap anything.

```
swap(Type arg1, Type arg2) {  
    Type temp = arg1;  
    arg1 = arg2;  
    arg2 = temp;  
}
```

38.3. Passing Object Variable

Java manipulates objects by reference, and all object variables are references. However, Java doesn't pass method arguments by reference, but by value.

Question is: why the member value of the object can get changed?



Code:

```

class Apple {
    public String color="red";
}

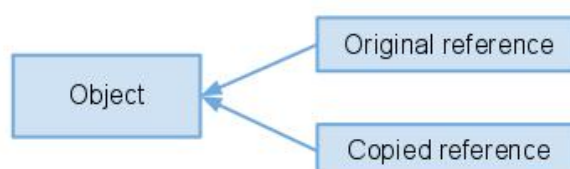
public class Main {
    public static void main(String[] args) {
        Apple apple = new Apple();
        System.out.println(apple.color);

        changeApple(apple);
        System.out.println(apple.color);
    }

    public static void changeApple(Apple apple){
        apple.color = "green";
    }
}

```

Since the original and copied reference refer the same object, the member value gets changed.



Output:

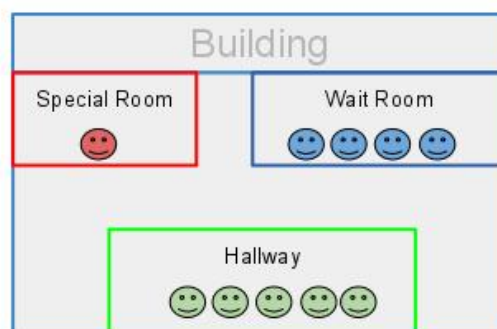
red
green

39. Monitors - The Basic Idea of Java Synchronization

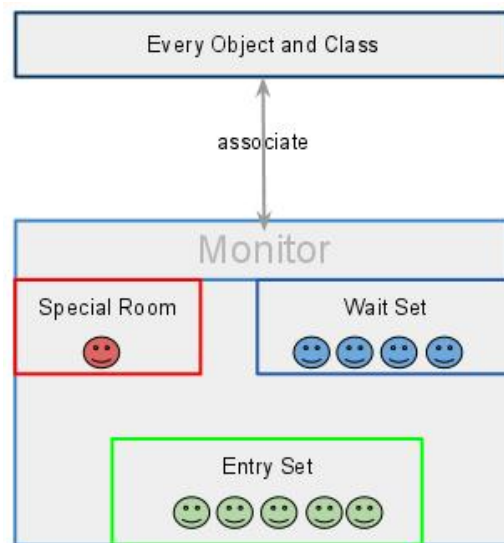
If you took operating system course in college, you might remember that monitor is an important concept of synchronization in operating systems. It is also used in Java synchronization. This post uses an analogy to explain the basic idea of "monitor".

39.1. What is a Monitor?

A monitor can be considered as a building which contains a special room. The special room can be occupied by only one customer(thread) at a time. The room usually contains some data and code.



If a customer wants to occupy the special room, he has to enter the Hallway(Entry Set) to wait first. Scheduler will pick one based on some criteria(e.g. FIFO). If he is suspended for some reason, he will be sent to the wait room, and be scheduled to reenter the special room later. As it is shown in the diagram above, there are 3 rooms in this building.



In brief, a monitor is a facility which monitors the threads' access to the special room. It ensures that only one thread can access the protected data or code.

39.2. How is it implemented in Java?

In the Java virtual machine, every object and class is logically associated with a monitor. To implement the mutual exclusion capability of monitors, a lock (sometimes called a mutex) is associated with each object and class. This is called a semaphore in operating systems books, mutex is a binary semaphore.

If one thread owns a lock on some data, then no others can obtain that lock until the thread that owns the lock releases it. It would be not convenient if we need to write a semaphore all the time when we do multi-threading programming. Luckily, we don't need to since JVM does that for us automatically.

To claim a monitor region which means data not accessible by more than one thread, Java provide synchronized statements and synchronized methods. Once the code is embedded with synchronized keyword, it is a monitor region. The locks are implemented in the background automatically by JVM.

39.3. In Java synchronization code, which part is monitor?

We know that each object/class is associated with a Monitor. I think it is good to say that each object has a monitor, since each object could have its own critical section, and capable of monitoring the thread sequence.

To enable collaboration of different threads, Java provide wait() and notify() to suspend a thread and to wake up another thread that are waiting on the object respectively. In addition, there are 3 other versions:

```
wait(long timeout, int nanos)
```

```
wait(long timeout) notified by other threads or notified by timeout.  
notify(all)
```

Those methods can only be invoked within a synchronized statement or synchronized method. The reason is that if a method does not require mutual exclusion, there is no need to monitor or collaborate between threads, every thread can access that method freely.

[Here](#) are some synchronization code examples.

40. The substring() Method in JDK 6 and JDK 7

The substring(int beginIndex, int endIndex) method in JDK 6 and JDK 7 are different. Knowing the difference can help you better use them. For simplicity reasons, in the following substring() represent the substring(int beginIndex, int endIndex) method.

40.1. What substring() does?

The substring(int beginIndex, int endIndex) method returns a string that starts with beginIndex and ends with endIndex-1.

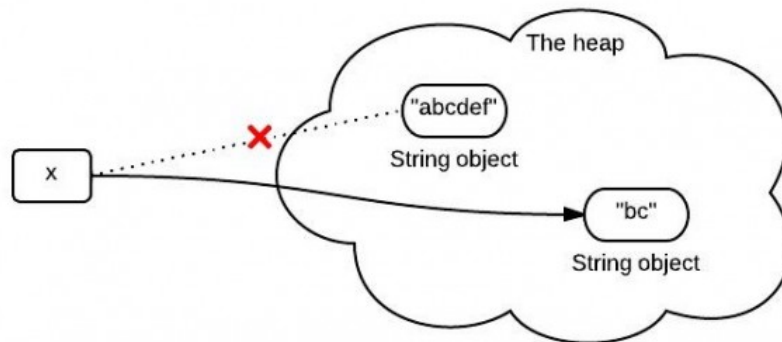
```
String x = "abcdef";  
x = x.substring(1,3);  
System.out.println(x);
```

Output:

bc

40.2. What happens when substring() is called?

You may know that because x is immutable, when x is assigned with the result of x.substring(1,3), it points to a totally new string like the following:

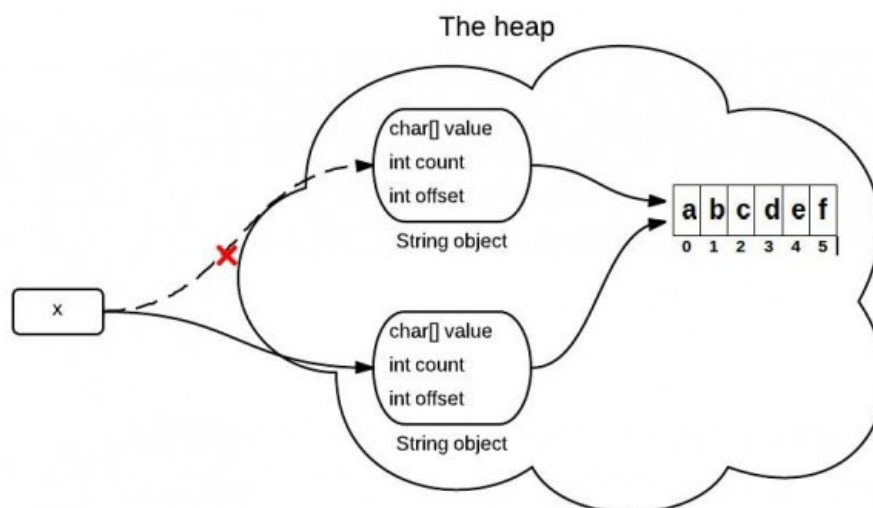


However, this diagram is not exactly right or it represents what really happens in the heap. What really happens when `substring()` is called is different between JDK 6 and JDK 7.

40.3. `substring()` in JDK 6

String is supported by a char array. In JDK 6, the String class contains 3 fields: `char[] value`, `int offset`, `int count`. They are used to store real character array, the first index of the array, the number of characters in the String.

When the `substring()` method is called, it creates a new string, but the string's value still points to the same array in the heap. The difference between the two Strings is their count and offset values.



The following code is simplified and only contains the key point for explain this

problem.

```
//JDK 6
String(int offset, int count, char value[]) {
    this.value = value;
    this.offset = offset;
    this.count = count;
}

public String substring(int beginIndex, int endIndex) {
    //check boundary
    return new String(offset + beginIndex, endIndex - beginIndex, value);
}
```

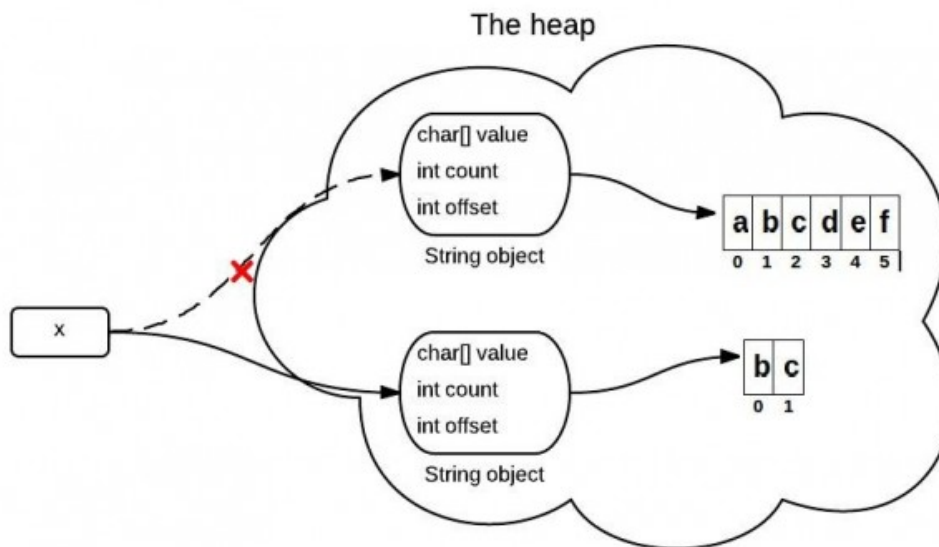
40.4. A problem caused by substring() in JDK 6

If you have a VERY long string, but you only need a small part each time by using substring(). This will cause a performance problem, since you need only a small part, you keep the whole thing. For JDK 6, the solution is using the following, which will make it point to a real sub string:

```
x = x.substring(x, y) + ""
```

40.5. substring() in JDK 7

This is improved in JDK 7. In JDK 7, the substring() method actually create a new array in the heap.



```
//JDK 7
public String(char value[], int offset, int count) {
    //check boundary
    this.value = Arrays.copyOfRange(value, offset, offset + count);
}

public String substring(int beginIndex, int endIndex) {
    //check boundary
    int subLen = endIndex - beginIndex;
    return new String(value, beginIndex, subLen);
}
```

[Top 10 questions about Java String.](#)

41. 2 Examples to Show How Java Exception Handling Works

There are 2 examples below. One shows all caller methods also need to handle exceptions thrown by the callee method. The other one shows the super class can be used to catch or handle subclass exceptions.

41.1. Caller method must handle exceptions thrown by the callee method

Here is a program which handles exceptions. Just test that, if an exception is thrown in one method, not only that method, but also all methods which call that method have to declare or throw that exception.

```
public class exceptionTest {
    private static Exception exception;

    public static void main(String[] args) throws Exception {
        callDoOne();
    }

    public static void doOne() throws Exception {
        throw exception;
    }

    public static void callDoOne() throws Exception {
        doOne();
    }
}
```

41.2. The super class can be used to catch or handle subclass exceptions

The following is also OK, because the super class can be used to catch or handle subclass exceptions:

```
class myException extends Exception{
}

public class exceptionTest {
    private static Exception exception;
    private static myException myexception;

    public static void main(String[] args) throws Exception {
        callDoOne();
    }

    public static void doOne() throws myException {
        throw myexception;
    }

    public static void callDoOne() throws Exception {
        doOne();
    }
}
```

```
        throw exception;
    }
}
```

This is the reason that only one parent class in the catch clause is syntactically safe.

42. Java Thread: an overriding example code

```
class A implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}

class B implements Runnable {

    public void run() {
        new A().run();
        new Thread(new A(), "name_thread2").run();
        new Thread(new A(), "name_thread3").start();
    }
}

public class Main {

    public static void main(String[] args) {
        new Thread(new B(), "name_thread1").start();
    }
}
```

What is the output?

```
=====
name_thread1
name_thread1
name_thread3
=====
```

The difference between "new Thread(new A(),"name_thread2").run();" and "new Thread(new A(),"name_thread3").start();" is that the start() method creates a new Thread and executes the run method in that thread. If you invoke the run() method directly, the code in run method will execute in the current thread. This explains why the code

prints two lines with the same thread name.

43. Comparable vs. Comparator in Java

Comparable and Comparator are two interfaces provided by Java Core API. From their names, we can tell they may be used for comparing stuff in some way. But what exactly are they and what is the difference between them? The following are two examples for answering this question. The simple examples compare two HDTV's size. How to use Comparable vs. Comparator is obvious after reading the code.

43.1. Comparable

Comparable is implemented by a class in order to be able to comparing object of itself with some other objects. The class itself must implement the interface in order to be able to compare its instance(s). The method required for implementation is compareTo(). Here is an example:

```
class HDTV implements Comparable<HDTV> {
    private int size;
    private String brand;

    public HDTV(int size, String brand) {
        this.size = size;
        this.brand = brand;
    }

    public int getSize() {
        return size;
    }

    public void setSize(int size) {
        this.size = size;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    @Override
    public int compareTo(HDTV tv) {
```

```

        if (this.getSize() > tv.getSize())
            return 1;
        else if (this.getSize() < tv.getSize())
            return -1;
        else
            return 0;
    }
}

public class Main {
    public static void main(String[] args) {
        HDTV tv1 = new HDTV(55, "Samsung");
        HDTV tv2 = new HDTV(60, "Sony");

        if (tv1.compareTo(tv2) > 0) {
            System.out.println(tv1.getBrand() + " is better.");
        } else {
            System.out.println(tv2.getBrand() + " is better.");
        }
    }
}

```

Sony is better.

43.2. Comparator

In some situations, you may not want to change a class and make it comparable. In such cases, Comparator can be used if you want to compare objects based on certain attributes/fields. For example, 2 persons can be compared based on 'height' or 'age' etc. (this can not be done using comparable.)

The method required to implement is compare(). Now let's use another way to compare those TV by size. One common use of Comparator is sorting. Both Collections and Arrays classes provide a sort method which use a Comparator.

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class HDTV {
    private int size;
    private String brand;

    public HDTV(int size, String brand) {
        this.size = size;
        this.brand = brand;
    }
}

```

```
public int getSize() {
    return size;
}

public void setSize(int size) {
    this.size = size;
}

public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}
}

class SizeComparator implements Comparator<HDTV> {
    @Override
    public int compare(HDTV tv1, HDTV tv2) {
        int tv1Size = tv1.getSize();
        int tv2Size = tv2.getSize();

        if (tv1Size > tv2Size) {
            return 1;
        } else if (tv1Size < tv2Size) {
            return -1;
        } else {
            return 0;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        HDTV tv1 = new HDTV(55, "Samsung");
        HDTV tv2 = new HDTV(60, "Sony");
        HDTV tv3 = new HDTV(42, "Panasonic");

        ArrayList<HDTV> al = new ArrayList<HDTV>();
        al.add(tv1);
        al.add(tv2);
        al.add(tv3);

        Collections.sort(al, new SizeComparator());
        for (HDTV a : al) {
            System.out.println(a.getBrand());
        }
    }
}
```

Output:

```
Panasonic
Samsung
Sony
```

Often we may use `Collections.reverseOrder()` method to get a descending order Comparator. Like the following:

```
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(3);
al.add(1);
al.add(2);
System.out.println(al);
Collections.sort(al);
System.out.println(al);

Comparator<Integer> comparator = Collections.reverseOrder();
Collections.sort(al,comparator);
System.out.println(al);
```

Output:

```
[3,1,2]
[1,2,3]
[3,2,1]
```

43.3. When to use Which?

In brief, a class that implements `Comparable` will be comparable, which means its instances can be compared with each other.

A class that implements `Comparator` will be used in mainly two situations: 1) It can be passed to a sort method, such as `Collections.sort()` or `Arrays.sort()`, to allow precise control over the sort order and 2) It can also be used to control the order of certain data structures, such as sorted sets (e.g. `TreeSet`) or sorted maps (e.g., `TreeMap`).

For example, to create a `TreeSet`. We can either pass the constructor a comparator or make the object class comparable.

Approach 1 - `TreeSet(Comparator comparator)`

```
class Dog {
    int size;

    Dog(int s) {
        size = s;
    }
}

class SizeComparator implements Comparator<Dog> {
```



```

@Override
public int compare(Dog d1, Dog d2) {
    return d1.size - d2.size;
}
}

public class ImpComparable {
    public static void main(String[] args) {
        TreeSet<Dog> d = new TreeSet<Dog>(new SizeComparator()); // pass
        comparator
        d.add(new Dog(1));
        d.add(new Dog(2));
        d.add(new Dog(1));
    }
}

```

Approach 2 - Implement Comparable

```

class Dog implements Comparable<Dog>{
    int size;

    Dog(int s) {
        size = s;
    }

    @Override
    public int compareTo(Dog o) {
        return o.size - this.size;
    }
}

public class ImpComparable {
    public static void main(String[] args) {
        TreeSet<Dog> d = new TreeSet<Dog>();
        d.add(new Dog(1));
        d.add(new Dog(2));
        d.add(new Dog(1));
    }
}

```

44. Overriding vs. Overloading in Java

Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }
    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

Overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name,
Different Parameter

Overriding and Overloading are two very important concepts in Java. They are confusing for Java novice programmers. This post illustrates their differences by using two simple examples.

44.1. Definitions

Overloading occurs when two or more methods in one class have the same method name but different parameters.

Overriding means having two methods with the same method name and parameters (i.e., method signature). One of the methods is in the parent class and the other is in the child class. Overriding allows a child class to provide a specific implementation of a method that is already provided its parent class.

44.2. Overriding vs. Overloading

Here are some important facts about Overriding and Overloading:

1. Real object type, not the reference variable's type, determines which overridden method is used at runtime. In contrast, reference type determines which overloaded method will be used at compile time.
2. Polymorphism applies to overriding, not to overloading.
3. Overriding is a run-time concept while overloading is a compile-time concept.

44.3. An Example of Overriding

Here is an example of overriding. After reading the code, guess the output.

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}

public class OverridingTest{
    public static void main(String [] args){
        Dog dog = new Hound();
        dog.bark();
    }
}
```

Output:

bowl

In the example above, the dog variable is declared to be a Dog. During compile time, the compiler checks if the Dog class has the bark() method. As long as the Dog class has the bark() method, the code compiles. At run-time, a Hound is created and assigned to dog. The JVM knows that dog is referring to the object of Hound, so it calls the bark() method of Hound. This is called Dynamic Polymorphism.

44.4. An Example of Overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

In this overloading example, the two bark method can be invoked by using different parameters. Compiler know they are different because they have different method signature (method name and method parameter list).

45. Why do we need Generic Types in Java?

Generic types are extensively used in Java collections. Why do we need Generic types in Java? Understanding this question can help us better understand a lot of related concepts. In this article, I will use a very short example to illustrate why Generic is useful.

45.1. Overview of Generics

The goal of implementing Generics is finding bugs in compile-time, other than in run-time. Finding bugs in compile-time can save time for debugging java program, because compile-time bugs are much easier to find and fix. Generic types only exist in compile-time. This fact is the most important thing to remember for learning Java Generics.

45.2. What if there is no Generics?

In the following program, the "Room" class defines a member object. We can pass any object to it, such as String, Integer, etc.

```
class Room {  
  
    private Object object;  
  
    public void add(Object object) {  
        this.object = object;  
    }  
  
    public Object get() {  
        return object;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Room room = new Room();  
        room.add(60);  
        //room.add("60"); //this will cause a run-time error  
    }  
}
```

```
        Integer i = (Integer)room.get();
        System.out.println(i);
    }
}
```

The program runs totally fine when we add an integer and cast it. But if a user accidentally add a string "60" to it, compiler does not know it is a problem. When the program is run, it will get a `ClassCastException`.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
    cannot be cast to java.lang.Integer
    at collection.Main.main(Main.java:21)
```

You may wonder why not just declare the field type to be `Integer` instead of `Object`. If so, then the room is not so much useful because it can only store one type of thing.

45.3. When generics is used

If generic type is used, the program becomes the following.

```
class Room<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}

public class Main {
    public static void main(String[] args) {
        Room<Integer> room = new Room<Integer>();
        room.add(60);

        Integer i = room.get();
        System.out.println(i);
    }
}
```

Now if someone adds `room.add("60")`, a compile-time error will be shown like the following:

```

public static void main(String[] args) {
    Room<Integer> room = new Room<Integer>();
    room.add("60");

    Integer i = room.get();
    System.out.println(i);
}

```

We can easily see how this works. In addition, there is no need to cast the result any more from `room.get()` since compile knows `get()` will return an `Integer`.

45.4. Summary

To sum up, the reasons to use Generics are as follows:

- Stronger type checking at compile time.
- Elimination of explicit cast.
- Enabling better code reusability such as implementation of generic algorithms

Java Generic type is only a compile-time concept. During run-time, all types information is erased, and this is call Type Erasure. [Here](#) is an interesting example to show how to avoid the common mistakes of Type Erasure.

46. Deep Understanding of Arrays.sort()

`Arrays.sort(T[], Comparator <? super T >c)` is a method for sorting user-defined object array. The official Java Doc briefly describe what it does, but not much for deep understanding. In this post, I will walk through the key information for deeper understanding of this method.

46.1. How to Use Arrays.sort(): A Simple Example

By reading the following example, you can quickly get an idea of how to use this method correctly. A `Comparator` is defined for comparing Dogs by size and then the `Comparator` is used as a parameter for the sort method.

```

import java.util.Arrays;
import java.util.Comparator;

class Dog{
    int size;
    public Dog(int s){
        size = s;
    }
}

```

```

    }
}

class DogSizeComparator implements Comparator<Dog>{

    @Override
    public int compare(Dog o1, Dog o2) {
        return o1.size - o2.size;
    }
}

public class ArraySort {

    public static void main(String[] args) {
        Dog d1 = new Dog(2);
        Dog d2 = new Dog(1);
        Dog d3 = new Dog(3);

        Dog[] dogArray = {d1, d2, d3};
        printDogs(dogArray);

        Arrays.sort(dogArray, new DogSizeComparator());
        printDogs(dogArray);
    }

    public static void printDogs(Dog[] dogs){
        for(Dog d: dogs)
            System.out.print(d.size + " " );

        System.out.println();
    }
}

```

Output:

```

2 1 3
1 2 3

```

46.2. The Strategy Pattern Used in Arrays.sort()

As this is a perfect example of [Strategy pattern](#), it is worth to mention here why strategy pattern is good for this situation. In brief, [Strategy pattern](#) enables different algorithms get selected at run-time. In this case, by passing different Comparator, different algorithms can get selected. Based on the example above and now assuming you have another Comparator which compares Dogs by weight instead of by size, you can simply create a new Comparator like the following.

```

class Dog{

```

```
int size;
int weight;

public Dog(int s, int w){
    size = s;
    weight = w;
}
}

class DogSizeComparator implements Comparator<Dog>{

    @Override
    public int compare(Dog o1, Dog o2) {
        return o1.size - o2.size;
    }
}

class DogWeightComparator implements Comparator<Dog>{

    @Override
    public int compare(Dog o1, Dog o2) {
        return o1.weight - o2.weight;
    }
}

public class ArraySort {

    public static void main(String[] args) {
        Dog d1 = new Dog(2, 50);
        Dog d2 = new Dog(1, 30);
        Dog d3 = new Dog(3, 40);

        Dog[] dogArray = {d1, d2, d3};
        printDogs(dogArray);

        Arrays.sort(dogArray, new DogSizeComparator());
        printDogs(dogArray);

        Arrays.sort(dogArray, new DogWeightComparator());
        printDogs(dogArray);
    }

    public static void printDogs(Dog[] dogs){
        for(Dog d: dogs)
            System.out.print("size="+d.size + " weight=" + d.weight + " ");

        System.out.println();
    }
}
```

```
size=2 weight=50 size=1 weight=30 size=3 weight=40
size=1 weight=30 size=2 weight=50 size=3 weight=40
size=1 weight=30 size=3 weight=40 size=2 weight=50
```

Comparator is just an interface. Any Comparator that implements this interface can be used during run-time. This is the key idea of Strategy design pattern.

46.3. Why Use "super"?

It is straightforward if "Comparator <T>c" is the parameter, but the second parameter is "Comparator<? super T>c". <? super T> means the type can be T or its super types. Why it allows super types? The answer is: This approach allows using same comparator for all sub classes. This is almost obvious in the following example.

```
import java.util.Arrays;
import java.util.Comparator;

class Animal{
    int size;
}

class Dog extends Animal{
    public Dog(int s){
        size = s;
    }
}

class Cat extends Animal{
    public Cat(int s){
        size = s;
    }
}

class AnimalSizeComparator implements Comparator<Animal>{

    @Override
    public int compare(Animal o1, Animal o2) {
        return o1.size - o2.size;
    }
    //in this way, all sub classes of Animal can use this comparator.
}

public class ArraySort {

    public static void main(String[] args) {
        Dog d1 = new Dog(2);
        Dog d2 = new Dog(1);
        Dog d3 = new Dog(3);
```

```

Dog[] dogArray = {d1, d2, d3};
printDogs(dogArray);

Arrays.sort(dogArray, new AnimalSizeComparator());
printDogs(dogArray);

System.out.println();

//when you have an array of Cat, same Comparator can be used.
Cat c1 = new Cat(2);
Cat c2 = new Cat(1);
Cat c3 = new Cat(3);

Cat[] catArray = {c1, c2, c3};
printDogs(catArray);

Arrays.sort(catArray, new AnimalSizeComparator());
printDogs(catArray);
}

public static void printDogs(Animal[] animals){
    for(Animal a: animals)
        System.out.print("size="+a.size + " ");
    System.out.println();
}
}

```

```

size=2 size=1 size=3
size=1 size=2 size=3

```

```

size=2 size=1 size=3
size=1 size=2 size=3

```

46.4. Summary

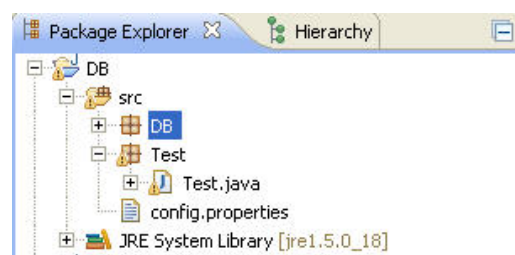
To summarize, the takeaway messages from `Arrays.sort()`:

- generic - super
- strategy pattern
- merge sort - $n \log(n)$ time complexity
- `Java.util.Collections#sort(List <T>list, Comparator <? super T>c)` has similar idea with `Arrays.sort`.

47. How to use java properties file?

For configuration purposes, using properties file is a good way of reusing. In this way, when the code is packaged to a jar file, other users can just put the different configurations in the config.properties file. The following is a simple example of using properties file.

1. create the file hierarchy like the following. Mainly remember to put the config.properties file under src package. Other testing code and database class are put in different package under src.



2. The following is the code.

```
package Test;

import java.io.IOException;
import java.util.Properties;

public class Test {

    public static void main(String[] args) {
        Properties configFile = new Properties();
        try {
            configFile.load(Test.class.getClassLoader().getResourceAsStream("config.properties"));
            String name = configFile.getProperty("name");
            System.out.println(name);
        } catch (IOException e) {

            e.printStackTrace();
        }
    }
}
```

3. The content in the configuration file following the format of "key=value".

48. Efficient Counter in Java

You may often need a counter to understand the frequency of something (e.g., words) from a database or text file. A counter can be easily implemented by using a `HashMap` in Java. This article compares different approaches to implement a counter. Finally, an efficient one will be concluded.

UPDATE: Check out [Java 8 counter](#), writing a counter is just 2 simple lines now.

48.1. The Naive Counter

Naively, it can be implemented as the following:

```
String s = "one two three two three three";
String[] sArr = s.split(" ");

//naive approach
HashMap<String, Integer> counter = new HashMap<String, Integer>();

for (String a : sArr) {
    if (counter.containsKey(a)) {
        int oldValue = counter.get(a);
        counter.put(a, oldValue + 1);
    } else {
        counter.put(a, 1);
    }
}
```

In each loop, you check if the key exists or not. If it does, increment the old value by 1, if not, set it to 1. This approach is simple and straightforward, but it is not the most efficient approach. This method is considered less efficient for the following reasons:

- `containsKey()`, `get()` are called twice when a key already exists. That means searching the map twice.
- Since `Integer` is immutable, each loop will create a new one for increment the old value

48.2. The Better Counter

Naturally we want a mutable integer to avoid creating many `Integer` objects. A mutable integer class can be defined as follows:

```
class MutableInteger {

    private int val;

    public MutableInteger(int val) {
```

```

        this.val = val;
    }

    public int get() {
        return val;
    }

    public void set(int val) {
        this.val = val;
    }

    //used to print value conveniently
    public String toString(){
        return Integer.toString(val);
    }
}

```

And the counter is improved and changed to the following:

```

HashMap<String, MutableInteger> newCounter = new HashMap<String,
    MutableInteger>();

for (String a : sArr) {
    if (newCounter.containsKey(a)) {
        MutableInteger oldValue = newCounter.get(a);
        oldValue.set(oldValue.get() + 1);
    } else {
        newCounter.put(a, new MutableInteger(1));
    }
}

```

This seems better because it does not require creating many Integer objects any longer. However, the search is still twice in each loop if a key exists.

48.3. The Efficient Counter

The `HashMap.put(key, value)` method returns the key's current value. This is useful, because we can use the reference of the old value to update the value without searching one more time!

```

HashMap<String, MutableInteger> efficientCounter = new HashMap<String,
    MutableInteger>();

for (String a : sArr) {
    MutableInteger initValue = new MutableInteger(1);
    MutableInteger oldValue = efficientCounter.put(a, initValue);

    if (oldValue != null){
        initValue.set(oldValue.get() + 1);
    }
}

```

```
}  
}
```

48.4. Performance Difference

To test the performance of the three different approaches, the following code is used. The performance test is on 1 million times. The raw results are as follows:

```
Naive Approach : 222796000  
Better Approach: 117283000  
Efficient Approach: 96374000
```

The difference is significant - 223 vs. 117 vs. 96. There is huge difference between Naive and Better, which indicates that creating objects are expensive!

```
String s = "one two three two three three";  
String[] sArr = s.split(" ");  
  
long startTime = 0;  
long endTime = 0;  
long duration = 0;  
  
// naive approach  
startTime = System.nanoTime();  
HashMap<String, Integer> counter = new HashMap<String, Integer>();  
  
for (int i = 0; i < 1000000; i++)  
    for (String a : sArr) {  
        if (counter.containsKey(a)) {  
            int oldValue = counter.get(a);  
            counter.put(a, oldValue + 1);  
        } else {  
            counter.put(a, 1);  
        }  
    }  
  
endTime = System.nanoTime();  
duration = endTime - startTime;  
System.out.println("Naive Approach : " + duration);  
  
// better approach  
startTime = System.nanoTime();  
HashMap<String, MutableInteger> newCounter = new HashMap<String,  
    MutableInteger>();  
  
for (int i = 0; i < 1000000; i++)  
    for (String a : sArr) {  
        if (newCounter.containsKey(a)) {
```

```

        MutableInteger oldValue = newCounter.get(a);
        oldValue.set(oldValue.get() + 1);
    } else {
        newCounter.put(a, new MutableInteger(1));
    }
}

endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("Better Approach: " + duration);

// efficient approach
startTime = System.nanoTime();

HashMap<String, MutableInteger> efficientCounter = new HashMap<String,
    MutableInteger>();

for (int i = 0; i < 1000000; i++)
    for (String a : sArr) {
        MutableInteger initValue = new MutableInteger(1);
        MutableInteger oldValue = efficientCounter.put(a, initValue);

        if (oldValue != null) {
            initValue.set(oldValue.get() + 1);
        }
    }

endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("Efficient Approach: " + duration);

```

When you use a counter, you probably also need a function to sort the map by value. You can check out [the frequently used method of HashMap](#).

48.5. Solutions from Keith

Added a couple tests: 1) Refactored "better approach" to just call get instead of containsKey. Usually, the elements you want are in the HashMap so that reduces from two searches to one. 2) Added a test with AtomicInteger, which michal mentioned. 3) Compared to singleton int array, which uses less memory according to <http://amzn.com/0748614079>

I ran the test program 3x and took the min to remove variance from other programs. Note that you can't do this within the program or the results are affected too much, probably due to gc.

```

Naive: 201716122
Better Approach: 112259166
Efficient Approach: 93066471

```

Better Approach (without containsKey): 69578496

Better Approach (without containsKey, with AtomicInteger): 94313287

Better Approach (without containsKey, with `int[]`): 65877234

Better Approach (without containsKey):

```
HashMap<String, MutableInteger> efficientCounter2 = new HashMap<String,
    MutableInteger>();
for (int i = 0; i < NUM_ITERATIONS; i++) {
    for (String a : sArr) {
        MutableInteger value = efficientCounter2.get(a);

        if (value != null) {
            value.set(value.get() + 1);
        } else {
            efficientCounter2.put(a, new MutableInteger(1));
        }
    }
}
```

Better Approach (without containsKey, with AtomicInteger):

```
HashMap<String, AtomicInteger> atomicCounter = new HashMap<String,
    AtomicInteger>();
for (int i = 0; i < NUM_ITERATIONS; i++) {
    for (String a : sArr) {
        AtomicInteger value = atomicCounter.get(a);

        if (value != null) {
            value.incrementAndGet();
        } else {
            atomicCounter.put(a, new AtomicInteger(1));
        }
    }
}
```

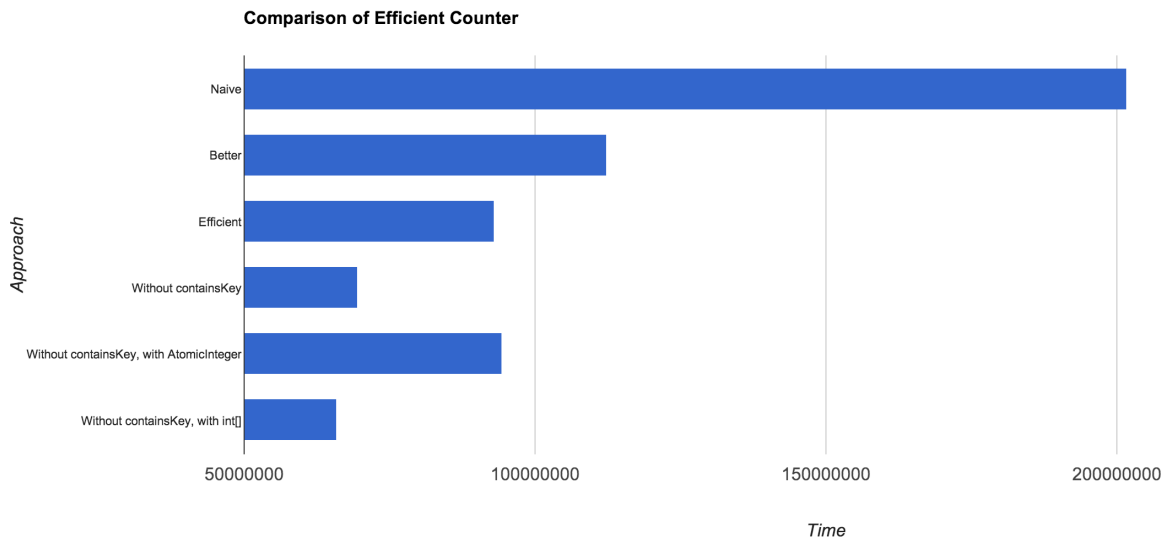
Better Approach (without containsKey, with `int[]`):

```
HashMap<String, int[]> intCounter = new HashMap<String, int[]>();
for (int i = 0; i < NUM_ITERATIONS; i++) {
    for (String a : sArr) {
        int[] valueWrapper = intCounter.get(a);

        if (valueWrapper == null) {
            intCounter.put(a, new int[] { 1 });
        } else {
            valueWrapper[0]++;
        }
    }
}
```

Guava's MultiSet is probably faster still.

48.6. Conclusion



The winner is the last one which uses int arrays.

49. How Developers Sort in Java?

While analyzing source code of a large number of open source Java projects, I found Java developers frequently sort in two ways. One is using the sort() method of Collections or Arrays, and the other is using sorted data structures, such as TreeMap and TreeSet.

49.1. Using sort() Method

If it is a collection, use Collections.sort() method.

```
// Collections.sort
List<ObjectName> list = new ArrayList<ObjectName>();
Collections.sort(list, new Comparator<ObjectName>() {
    public int compare(ObjectName o1, ObjectName o2) {
        return o1.toString().compareTo(o2.toString());
    }
});
```

If it is an array, use Arrays.sort() method.

```
// Arrays.sort
ObjectName[] arr = new ObjectName[10];
Arrays.sort(arr, new Comparator<ObjectName>() {
    public int compare(ObjectName o1, ObjectName o2) {
        return o1.toString().compareTo(o2.toString());
    }
});
```

This is very convenient if a collection or an array is already set up.

49.2. Using Sorted Data Structures

If it is a list or set, use TreeSet to sort.

```
// TreeSet
Set<ObjectName> sortedSet = new TreeSet<ObjectName>(new
    Comparator<ObjectName>() {
        public int compare(ObjectName o1, ObjectName o2) {
            return o1.toString().compareTo(o2.toString());
        }
    });
sortedSet.addAll(unsortedSet);
```

If it is a map, use TreeMap to sort. TreeMap is sorted by key.

```
// TreeMap - using String.CASE_INSENSITIVE_ORDER which is a Comparator that
// orders Strings by compareToIgnoreCase
Map<String, Integer> sortedMap = new TreeMap<String,
    Integer>(String.CASE_INSENSITIVE_ORDER);
sortedMap.putAll(unsortedMap);
```

```
//TreeMap - In general, defined comparator
Map<ObjectName, String> sortedMap = new TreeMap<ObjectName, String>(new
    Comparator<ObjectName>() {
        public int compare(ObjectName o1, ObjectName o2) {
            return o1.toString().compareTo(o2.toString());
        }
    });
sortedMap.putAll(unsortedMap);
```

This approach is very useful, if you would do a lot of search operations for the collection. The sorted data structure will give time complexity of $O(\log n)$, which is lower than $O(n)$.

49.3. Bad Practices

There are still bad practices, such as using self-defined sorting algorithm. Take the code below for example, not only the algorithm is not efficient, but also it is not readable. This happens a lot in different forms of variations.

```
double t;
for (int i = 0; i < 2; i++)
    for (int j = i + 1; j < 3; j++)
        if (r[j] < r[i]) {
            t = r[i];
            r[i] = r[j];
            r[j] = t;
        }
```

50. The Most Widely Used Java Libraries

A typical Java project relies on third-party libraries. This article summarizes the most popular and widely used Java libraries for a variety of different applications. A simple example is also provided for some of them, if it can be found on ProgramCreek.

[Java SDK](#) is surely the #1 widely used library. So the focus of this list is the popular third-party libraries. The list may not be perfect, so leave your comment if you think others should be included.

50.1. Core

- Apache Commons Lang - Apache's library that provides a host of helper utilities for the java.lang API, such as String manipulation, object creation, etc.
- Google Guava - Google's Core library for collections, caching, primitives support, etc. (example)

50.2. HTML, XML Parser

- Jsoup - a convenient library to manipulate HTML. (example)
- STaX - Process XML code. (example)

50.3. Web Frameworks

- Spring - an open source application framework and inversion of control container for the Java platform. (example)

- Struts 2 - most popular web framework from Apache. (example)
- Google Web Toolkit - a development toolkit from Google for building and optimizing complex browser-based applications. (example)
- Stripes - a presentation framework for building web applications using the latest Java technologies.
- Tapestry - component oriented framework for creating dynamic, robust, highly scalable web applications in Java.

50.4. Chart, Report, Graph

- JFreeChart - creates charts such as bar charts, line charts, pie charts, etc.
- JFreeReport - creates PDF reports.
- JGraphT - create graph that contains a set of nodes connected by edges.

50.5. Windowing Libraries

- Swing - a GUI library from SDK. (example)
- SWT - a GUI library from eclipse. SWT vs. Swing

50.6. GUI Frameworks

- Eclipse RCP. (example)

50.7. Natural Language Processing

- OpenNLP - a library from Apache. (example)
- Stanford Parser - a library from Stanford University. (example)

If you are an expert of NLP, [here](#) are more tools.

50.8. Static Analysis

- Eclipse JDT - a library from IBM which can manipulate Java source code. (example)
- WALA - a library that can process .jar file, i.e., bytecode. (example)

50.9. JSON

- Jackson - a multi-purpose Java library for processing JSON data format. Jackson aims to be the best possible combination of fast, correct, lightweight, and ergonomic for developers.
- XStream - a simple library to serialize objects to XML and back again.
- Google Gson - a Java library that can be used to convert Java Objects into their JSON representation. (example)
- JSON-lib - a java library for transforming beans, maps, collections, java arrays and XML to JSON and back again to beans and DynaBeans.

50.10. . Math

- Apache Commons Math - provide functions for math and statistics.

50.11. . Logging

- Apache Log4j - most popular logging library. (example)
- Logback - a successor to the popular log4j project.

50.12. . Office-Complicant

- Apache POI - APIs for manipulating various file formats based upon Microsoft's OLE 2 Compound Document format using pure Java.
- Docx4j - a Java library for creating and manipulating Microsoft Open XML (Word docx, Powerpoint pptx, and Excel xlsx) files.

— More from comments —

50.13. . Date and Time

- Joda-Time - a quality replacement for the Java date and time classes.

50.14. . Database

- Hibernate / EclipseLink / JPA
- JDO
- jOOQ
- SpringJDBC / Spring Data

- Apache DbUtils

50.15. Development Tools

- Lombok - a Java library meant to simplify the development of Java code writing

* 1) The list above are based on my own survey combined with personal experience. It is possible that they are not precisely THE MOST popular, but at least well-known.

* 2) I will keep updating this list to make it more complete and accurate. Thanks for your comments.

51. Inheritance vs. Composition in Java

This article illustrates the concepts of inheritance vs. composition in Java. It first shows an example of inheritance, and then shows how to improve the inheritance design by using composition. How to choose between them is summarized at the end.

51.1. Inheritance

Let's suppose we have an Insect class. This class contains two methods: 1) move() and 2) attack().

```
class Insect {
    private int size;
    private String color;

    public Insect(int size, String color) {
        this.size = size;
        this.color = color;
    }

    public int getSize() {
        return size;
    }

    public void setSize(int size) {
        this.size = size;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
```

```
        this.color = color;
    }

    public void move() {
        System.out.println("Move");
    }

    public void attack() {
        move(); //assuming an insect needs to move before attacking
        System.out.println("Attack");
    }
}
```

Now you want to define a Bee class, which is a type of Insect, but have different implementations of attack() and move(). This can be done by using an inheritance design like the following:

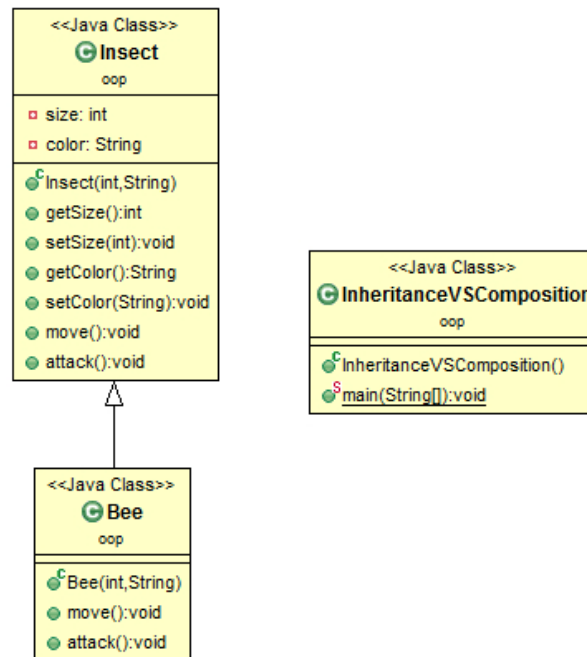
```
class Bee extends Insect {
    public Bee(int size, String color) {
        super(size, color);
    }

    public void move() {
        System.out.println("Fly");
    }

    public void attack() {
        move();
        super.attack();
    }
}
```

```
public class InheritanceVSComposition {
    public static void main(String[] args) {
        Insect i = new Bee(1, "red");
        i.attack();
    }
}
```

The class hierarchy diagram is as simple as:



Output:

Fly
Fly
Attack

"Fly" was printed twice, which indicates move() is called twice. But it should be called only ONCE.

The problem is caused by the super.attack() method. The attack() method of Insect invokes move() method. When the subclass calls super.attack(), it also invokes the overridden move() method.

To fix the problem, we can:

- eliminate the subclass's attack() method. This will make the subclass depends on the superclass's implementation of attack(). If the attack() method in the superclass is changed later (which is out of your control), e.g., the superclass's attack() method use another method to move, the subclass will need to be changed too. This is bad encapsulation.
- rewrite the attack() method like the following: public void attack() move(); System.out.println("Attack"); This would guarantee the correct result, because the subclass is not dependent on the superclass any more. However, the code is the duplicate of the superclass. (Image attack() method does complex things other than just printing a string) This does not following software engineering rule of reusing.

This inheritance design is bad, because the subclass depends on the implementation details of its superclass. If the superclass changes, the subclass may break.

51.2. Composition

Instead of inheritance, composition can be used in this case. Let's first take a look at the composition solution.

The attack function is abstracted as an interface.

```
interface Attack {  
    public void move();  
    public void attack();  
}
```

Different kinds of attack can be defined by implementing the Attack interface.

```
class AttackImpl implements Attack {  
    private String move;  
    private String attack;  
  
    public AttackImpl(String move, String attack) {  
        this.move = move;  
        this.attack = attack;  
    }  
  
    @Override  
    public void move() {  
        System.out.println(move);  
    }  
  
    @Override  
    public void attack() {  
        move();  
        System.out.println(attack);  
    }  
}
```

Since the attack function is extracted, Insect does not do anything related with attack any longer.

```
class Insect {  
    private int size;  
    private String color;  
  
    public Insect(int size, String color) {  
        this.size = size;  
        this.color = color;  
    }  
}
```

```
public int getSize() {
    return size;
}

public void setSize(int size) {
    this.size = size;
}

public String getColor() {
    return color;
}

public void setColor(String color) {
    this.color = color;
}
}
```

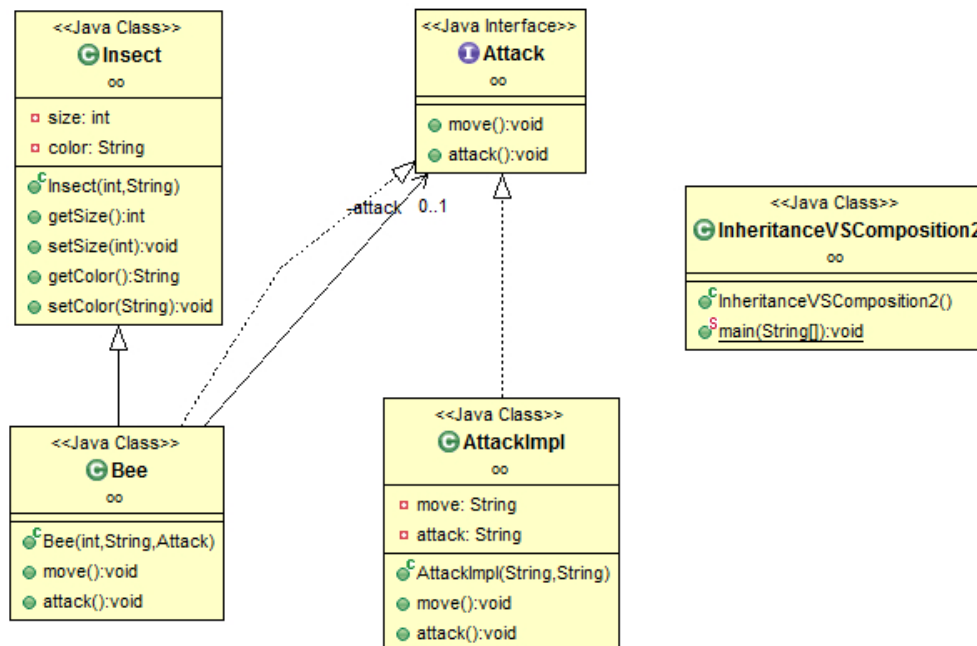
Bee is a type of Insect, it can attack.

```
// This wrapper class wrap an Attack object
class Bee extends Insect implements Attack {
    private Attack attack;

    public Bee(int size, String color, Attack attack) {
        super(size, color);
        this.attack = attack;
    }

    public void move() {
        attack.move();
    }

    public void attack() {
        attack.attack();
    }
}
```



```

public class InheritanceVSComposition2 {
    public static void main(String[] args) {
        Bee a = new Bee(1, "black", new AttackImpl("fly", "move"));
        a.attack();

        // if you need another implementation of move()
        // there is no need to change Insect, we can quickly use new method to
        // attack

        Bee b = new Bee(1, "black", new AttackImpl("fly", "sting"));
        b.attack();
    }
}

```

```

fly
move
fly
sting

```

51.3. When to Use Which?

The following two items can guide the selection between inheritance and composition:

- If there is an IS-A relation, and a class wants to expose all the interface to another class, inheritance is likely to be preferred.

- If there is a HAS-A relationship, composition is preferred.

In summary, Inheritance and composition both have their uses, and it pays to understand their relative merits.

52. A simple TreeSet example

The following is a very simple TreeSet example. From this simple example, you will see:

- TreeSet is sorted
- How to iterate a TreeSet
- How to check empty
- How to retrieve first/last element
- How to remove an element

If you want to know more about Java Collection, check out the [Java Collection hierarchy diagram](#).

```
import java.util.Iterator;
import java.util.TreeSet;

public class TreeSetExample {

    public static void main(String[] args) {
        System.out.println("Tree Set Example!\n");
        TreeSet<Integer> tree = new TreeSet<Integer>();
        tree.add(12);
        tree.add(63);
        tree.add(34);
        tree.add(45);

        // here it test it's sorted, 63 is the last element. see output below
        Iterator<Integer> iterator = tree.iterator();
        System.out.print("Tree set data: ");

        // Displaying the Tree set data
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println();

        // Check empty or not
        if (tree.isEmpty()) {
```

```

        System.out.print("Tree Set is empty.");
    } else {
        System.out.println("Tree Set size: " + tree.size());
    }

    // Retrieve first data from tree set
    System.out.println("First data: " + tree.first());

    // Retrieve last data from tree set
    System.out.println("Last data: " + tree.last());

    if (tree.remove(45)) { // remove element by value
        System.out.println("Data is removed from tree set");
    } else {
        System.out.println("Data doesn't exist!");
    }
    System.out.print("Now the tree set contain: ");
    iterator = tree.iterator();

    // Displaying the Tree set data
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }
    System.out.println();
    System.out.println("Now the size of tree set: " + tree.size());

    // Remove all
    tree.clear();
    if (tree.isEmpty()) {
        System.out.print("Tree Set is empty.");
    } else {
        System.out.println("Tree Set size: " + tree.size());
    }
}
}

```

Output:

Tree Set Example!

Tree set data: 12 34 45 63

Tree Set size: 4

First data: 12

Last data: 63

Data is removed from tree set

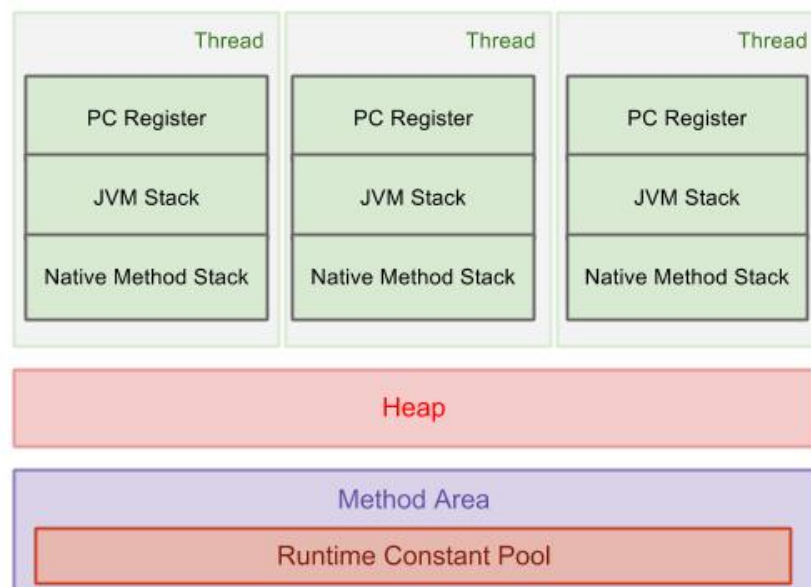
Now the tree set contain: 12 34 63

Now the size of tree set: 3

Tree Set is empty.

53. JVM Run-Time Data Areas

This is my note of reading JVM specification. I draw a diagram which helps me understand.



53.1. Data Areas for Each Individual Thread (not shared)

Data Areas for each individual thread include program counter register, JVM Stack, and Native Method Stack. They are all created when a new thread is created.

Program Counter Register - used to control each execution of each thread. JVM Stack - contains frames which is demonstrated in the diagram below. Native Method Stack - used to support native methods, i.e., non-Java language methods. [U+3000]

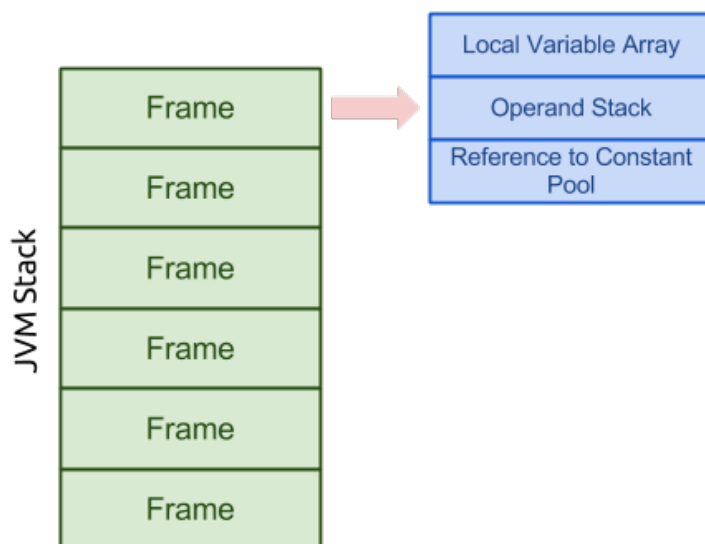
53.2. Data Areas Shared by All Threads

All threads share Heap and Method Area.

Heap: it is the area that we most frequently deal with. It stores arrays and objects, created when JVM starts up. Garbage Collection works in this area.

Method Area: it stores run-time constant pool, field and method data, and methods and constructors code.

Runtime Constant Pool: It is a per-class or per-interface run-time representation of the constant_pool table in a class file. It contains several kinds of constants, ranging from numeric literals known at compile-time to method and field references that must be resolved at run-time.



Stack contains Frames, and a frame is pushed to the stack when a method is invoked. A frame contains local variable array, Operand Stack, Reference to Constant Pool.

For more information, please go to the official JVM specification site.

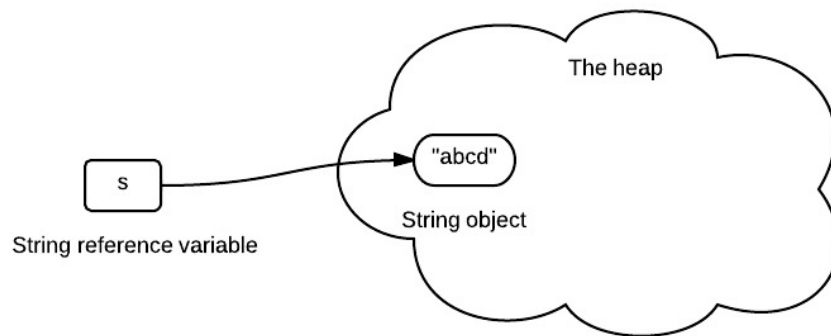
54. Diagram to show Java String's Immutability

Here are a set of diagrams to illustrate Java String's immutability.

54.1. Declare a string

```
String s = "abcd";
```

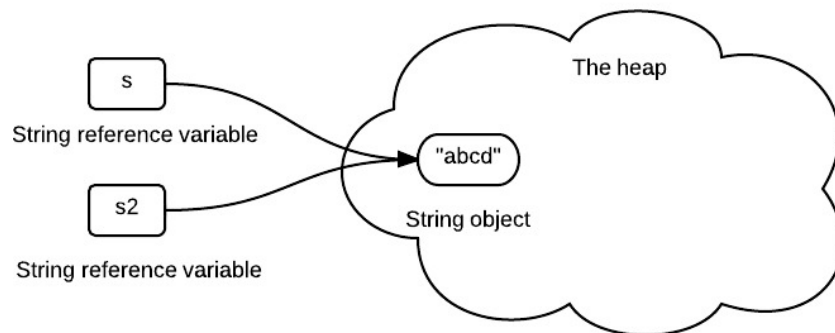
s stores the reference of the string object. The arrow below should be interpreted as "store reference of".



54.2. Assign one string variable to another string variable

```
String s2 = s;
```

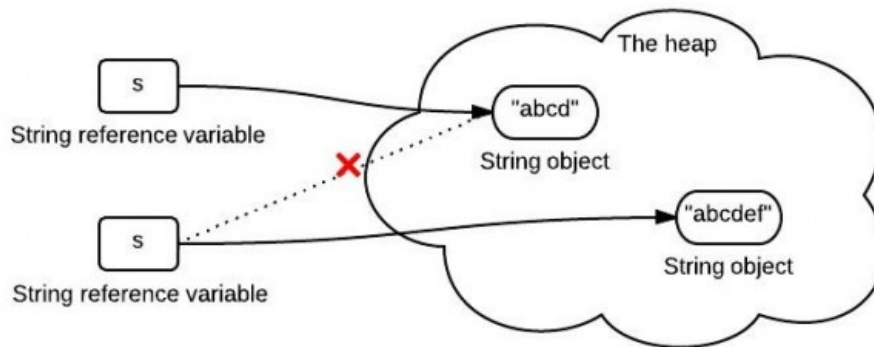
s2 stores the same reference value, since it is the same string object.



54.3. Concat string

```
s = s.concat("ef");
```

s now stores the reference of newly created string object.



54.4. Summary

Once a string is created in [memory](#)(heap), it can not be changed. We should note that all methods of String do not change the string itself, but rather return a new String.

If we need a string that can be modified, we will need StringBuffer or StringBuilder. Otherwise, there would be a lot of time wasted for Garbage Collection, since each time a new String is created. [Here](#) is an example of StringBuilder usage.

55. Create Java String Using " " or Constructor?

In Java, a string can be created by using two methods:

```
String x = "abc";
String y = new String("abc");
```

What is the difference between using double quotes and using constructor?

55.1. Double Quotes vs. Constructor

This question can be answered by using two simple code examples.

Example 1:

```
String a = "abcd";
String b = "abcd";
System.out.println(a == b); // True
System.out.println(a.equals(b)); // True
```

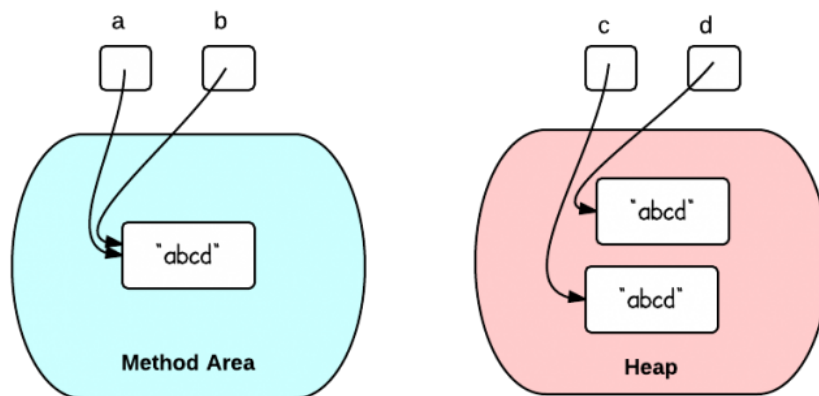
a==b is true because a and b are referring to the same string literal in the [method area](#). The memory references are the same.

When the same string literal is created more than once, only one copy of each distinct string value is stored. This is called "string interning". All compile-time constant strings in Java are automatically interned.

Example 2:

```
String c = new String("abcd");  
String d = new String("abcd");  
System.out.println(c == d); // False  
System.out.println(c.equals(d)); // True
```

c==d is false because c and d refer to two different objects in the heap. Different objects always have different memory references.



55.2. Run-Time String Interning

Thanks to LukasEder (his comment below):

String interning can still be done at run-time, even if two strings are constructed with constructors:

```
String c = new String("abcd").intern();  
String d = new String("abcd").intern();  
System.out.println(c == d); // Now true  
System.out.println(c.equals(d)); // True
```

55.3. When to Use Which

Because the literal "abcd" is already of type String, using constructor will create an extra unnecessary object. Therefore, double quotes should be used if you just need to create a String.

If you do need to create a new object in the heap, constructor should be used. [Here](#) is a use case.

56. What exactly is null in Java?

Let's start from the following statement:

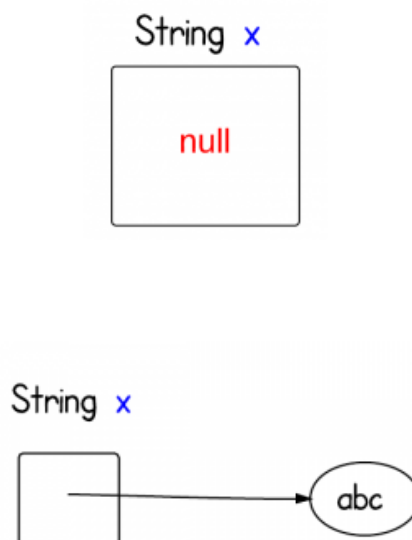
```
String x = null;
```

56.1. What exactly does this statement do?

Recall what is a variable and what is a value. A common metaphor is that a variable is similar to a box. Just as you can use a box to store something, you can use a variable to store a value. When declaring a variable, we need to set its type.

There are two major categories of types in Java: primitive and reference. Variables declared of a primitive type store values; variables declared of a reference type store references. In this case, the initialization statement declares a variable "x". "x" stores String reference. It is null here.

The following visualization gives a better sense about this concept.



56.2. What exactly is null in memory?

What exactly is null in memory? Or What is the null value in Java?

First of all, null is not a valid object instance, so there is no memory allocated for it. It is simply a value that indicates that the object reference is not currently referring to an object.

From [JVM Specifications](#):

The Java Virtual Machine specification does not mandate a concrete value encoding null.

I would assume it is all zeros or something similar like it is on other C-like languages.

56.3. What exactly is x in memory?

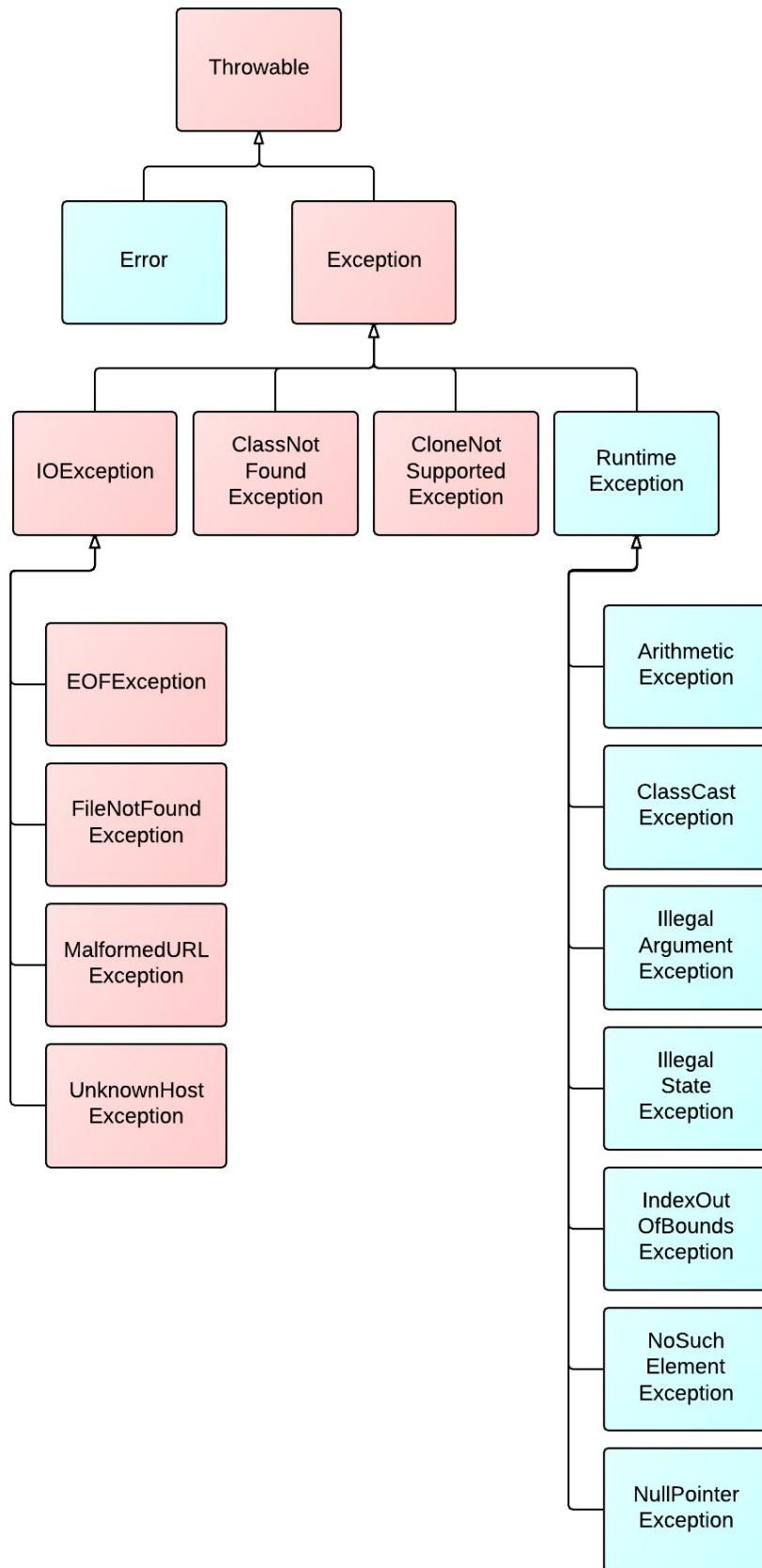
Now we know what null is. And we know a variable is a storage location and an associated symbolic name (an identifier) which contains some value. Where exactly x is in memory?

From [the diagram of JVM run-time data areas](#), we know that since each method has a private stack frame within the thread's stack, the local variables are located on that frame.

57. Diagram of Exception Hierarchy

In Java, exceptions can be checked or unchecked. They both fit into a class hierarchy. The following diagram shows the Java Exception classes hierarchy.

Red-colored are checked exceptions. Any checked exceptions that may be thrown in a method must either be caught or declared in the method's throws clause. Checked exceptions must be caught at compile time. Checked exceptions are so called because both the Java compiler and the Java virtual machine check to make sure this rule is obeyed. Green-colored are unchecked exceptions. They are exceptions that are not expected to be recovered, such as null pointer, divide by 0, etc.



Check out [top 10 questions about Java exceptions](#).

58. java.util.ConcurrentModificationException

This post shows how to solve the problem of java.util.ConcurrentModificationException for ArrayList.

The error message looks like the following:

```
Exception in thread "main" java.util.ConcurrentModificationException
  at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
  at java.util.ArrayList$Itr.next(Unknown Source)
  ...
  ...
```

58.1. The Problem

You may want to iterate through an ArrayList, and delete some element under some condition. For example, the following code looks reasonable:

```
import java.util.ArrayList;
import java.util.List;

public class AddRemoveListElement {

    public static void main(String args[]) {
        List<String> list = new ArrayList<String>();
        list.add("A");
        list.add("B");

        for (String s : list) {
            if (s.equals("B")) {
                list.remove(s);
            }
        }
    }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
  at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
  at java.util.ArrayList$Itr.next(Unknown Source)
  at arraylist.AddRemoveListElement.main(AddRemoveListElement.java:17)
```

58.2. Solution 1

Iterator can be used to solve this problem. Iterators allow the caller to remove elements from the underlying collection during the iteration.

```
Iterator<String> iter = list.iterator();
while(iter.hasNext()){
    String str = iter.next();
    if( str.equals("B") )
    {
        iter.remove();
    }
}
```

58.3. Solution 2

Instead of ArrayList, CopyOnWriteArrayList can be used to solve the problem. CopyOnWriteArrayList is a thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.

```
public static void main(String args[]) {
    List<String> list = new CopyOnWriteArrayList<String>();
    list.add("A");
    list.add("B");

    for (String s : list) {
        if (s.equals("B")) {
            list.remove(s);
        }
    }
}
```

58.4. How about other Collection types?

```
public static void main(String args[]) {
    Set<String> set = new HashSet<String>();
    set.add("A");
    set.add("B");

    for (String s : set) {
        if (s.equals("B")) {
            set.remove(s);
        }
    }
}
```

```
public static void main(String args[]) {
    LinkedList<String> llist = new LinkedList<String>();
    llist.add("A");
    llist.add("B");

    for (String s : llist) {
        if (s.equals("B")) {
            llist.remove(s);
        }
    }
}
```

The above code is fine, because they do not use array as the underlining data structure.

59. When to use private constructors in Java?

If a method is private, it means that it can not be accessed from any class other than itself. This is the access control mechanism provided by Java. When it is used appropriately, it can produce security and functionality. Constructors, like regular methods, can also be declared as private. You may wonder why we need a private constructor since it is only accessible from its own class. When a class needs to prevent the caller from creating objects. Private constructors are suitable. Objects can be constructed only internally.

One application is in [the singleton design pattern](#). The policy is that only one object of that class is supposed to exist. So no other class than itself can access the constructor. This ensures the single instance existence of the class. Private constructors have been widely used in JDK, the following code is part of Runtime class.

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    public static Runtime getRuntime() {
        return currentRuntime;
    }

    // Don't let anyone else instantiate this class
    private Runtime() {
    }
}
```

60. Top 10 Questions for Java Regular Expression

This post summarizes the top questions asked about Java regular expressions. As they are most frequently asked, you may find that they are also very useful.

1. How to extract numbers from a string?

One common question of using regular expression is to extract all the numbers into an array of integers.

In Java, `\d` means a range of digits (0-9). Using the predefined classes whenever possible will make your code easier to read and eliminate errors introduced by malformed character classes. Please refer to [Predefined character classes](#) for more details. Please note the first backslash in `\d`. If you are using an escaped construct within a string literal, you must precede the backslash with another backslash for the string to compile. That's why we need to use `\\d`.

```
List<Integer> numbers = new LinkedList<Integer>();
Pattern p = Pattern.compile("\\d+");
Matcher m = p.matcher(str);
while (m.find()) {
    numbers.add(Integer.parseInt(m.group()));
}
```

2. How to split Java String by newlines?

There are at least three different ways to enter a new line character, dependent on the operating system you are working on.

`\r` represents CR (Carriage Return), which is used in Unix
`\n` means LF (Line Feed), used in Mac OS
`\r\n` means CR + LF, used in Windows

Therefore the most straightforward way to split string by new lines is

```
String lines[] = String.split("\\r?\\n");
```

But if you don't want empty lines, you can use, which is also my favourite way:

```
String.split("[\\r\\n]+")
```

A more robust way, which is really system independent, is as follows. But remember, you will still get empty lines if two newline characters are placed side by side.

```
String.split(System.getProperty("line.separator"));
```

3. Importance of Pattern.compile()

A regular expression, specified as a string, must first be compiled into an instance of

`Pattern` class. `Pattern.compile()` method is the only way to create an instance of object. A typical invocation sequence is thus

```
Pattern p = Pattern.compile("a*b");
Matcher matcher = p.matcher("aaaaab");
assert matcher.matches() == true;
```

Essentially, `Pattern.compile()` is used to transform a regular expression into a Finite state machine (see *Compilers: Principles, Techniques, and Tools* (2nd Edition)). But all of the states involved in performing a match reside in the matcher. By this way, the `Pattern` `p` can be reused. And many matchers can share the same pattern.

```
Matcher anotherMatcher = p.matcher("aab");
assert anotherMatcher.matches() == true;
```

`Pattern.matches()` method is defined as a convenience for when a regular expression is used just once. This method still uses `compile()` to get the instance of a `Pattern` implicitly, and matches a string. Therefore,

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

is equivalent to the first code above, though for repeated matches it is less efficient since it does not allow the compiled pattern to be reused.

4. How to escape text for regular expression?

In general, regular expression uses `"\"` to escape constructs, but it is painful to precede the backslash with another backslash for the Java string to compile. There is another way for users to pass string literals to the `Pattern`, like `"$5"`. Instead of writing `$5` or `[\backslash]5, we can type`

```
Pattern.quote("$5");
```

5. Why does `String.split()` need pipe delimiter to be escaped?

`String.split()` splits a string around matches of the given regular expression. Java expression supports special characters that affect the way a pattern is matched, which is called `metacharacter`. `|` is one `metacharacter` which is used to match a single regular expression out of several possible regular expressions. For example, `A|B` means either `A` or `B`. Please refer to [Alternation with The Vertical Bar or Pipe Symbol](#) for more details. Therefore, to use `|` as a literal, you need to escape it by adding `\` in front of it, like

`|`.

6. How can we match `anbn` with Java regex?

This is the language of all non-empty strings consisting of some number of `a`'s followed by an equal number of `b`'s, like `ab`, `aabb`, and `aaabbb`. This language can be shown to be context-free grammar $S \rightarrow aSb \mid ab$, and therefore a non-regular language.

However, Java regex implementations can recognize more than just regular languages. That is, they are not "regular" by formal language theory definition. Using lookahead and self-reference matching will achieve it. Here I will give the final regular

expression first, then explain it a little bit. For a comprehensive explanation, I would refer you to read [How can we match \$a^n b^n\$ with Java regex.](#)

```
Pattern p = Pattern.compile("(?x)(?:a(?:a*(\\1?+b)))+\\1");
// true
System.out.println(p.matcher("aaabbb").matches());
// false
System.out.println(p.matcher("aaaabbb").matches());
// false
System.out.println(p.matcher("aaabbbb").matches());
// false
System.out.println(p.matcher("caaabbb").matches());
```

Instead of explaining the syntax of this complex regular expression, I would rather say a little bit how it works.

- In the first iteration, it stops at the first a then looks ahead (after skipping some as by using a^*) whether there is a b. This was achieved by using $(?:a(?:a^*(1?+b)))$. If it matches, 1 , the self-reference matching, will match the very inner parenthesized elements, which is one single b in the first iteration.
- In the second iteration, the expression will stop at the second a, then it looks ahead (again skipping as) to see if there will be b. But this time, $1+b$ is actually equivalent to bb , therefore two bs have to be matched. If so, 1 will be changed to bb after the second iteration.
- In the n th iteration, the expression stops at the n th a and see if there are n bs ahead.

By this way, the expression can count the number of as and match if the number of bs followed by a is same.

7. How to replace 2 or more spaces with single space in string and delete leading spaces only?

`String.replaceAll()` replaces each substring that matches the given regular expression with the given replacement. "2 or more spaces" can be expressed by regular expression `[]+`. Therefore, the following code will work. Note that, the solution won't ultimately remove all leading and trailing whitespaces. If you would like to have them deleted, you can use `String.trim()` in the pipeline.

```
String line = " aa bbbbbb ccc d ";
// " aa bbbbbb ccc d "
System.out.println(line.replaceAll("[\\s]+", " "));
```

8. How to determine if a number is a prime with regex?

```
public static void main(String[] args) {
```

```

// false
System.out.println(prime(1));
// true
System.out.println(prime(2));
// true
System.out.println(prime(3));
// true
System.out.println(prime(5));
// false
System.out.println(prime(8));
// true
System.out.println(prime(13));
// false
System.out.println(prime(14));
// false
System.out.println(prime(15));
}

public static boolean prime(int n) {
    return !new String(new char[n]).matches(".*?(..+?)\\1+");
}

```

The function first generates `n` number of characters and tries to see if that string matches `.*?(..+?)`

`1+`. If it is prime, the expression will return false and the `!` will reverse the result.

The first part `.*?` just tries to make sure `1` is not primer. The magic part is the second part where backreference is used. `(..+?)`

`1+` first try to matches `n` length of characters, then repeat it several times by

`1+`.

By definition, a **prime number** is a natural number greater than `1` that has no positive divisors other than `1` and itself. That means if $a=n*m$ then `a` is not a prime. $n*m$ can be further explained "repeat `n` `m` times", and that is exactly what the regular expression does: matches `n` length of characters by using `(..+?)`, then repeat it `m` times by using `1+`. Therefore, if the pattern matches, the number is not prime, otherwise it is. Remind that `!` will reverse the result.

9. How to split a comma-separated string but ignoring commas in quotes?

You have reached the point where regular expressions break down. It is better and more neat to write a simple splitter, and handles special cases as you wish.

Alternative, you can mimic the operation of finite state machine, by using a switch statement or if-else. Attached is a snippet of code.

```

public static void main(String[] args) {
    String line = "aaa,bbb,\"c,c\",dd;dd,\"e,e";
    List<String> toks = splitComma(line);
    for (String t : toks) {
        System.out.println("> " + t);
    }
}

```

```

private static List<String> splitComma(String str) {
    int start = 0;
    List<String> toks = new ArrayList<String>();
    boolean withinQuote = false;
    for (int end = 0; end < str.length(); end++) {
        char c = str.charAt(end);
        switch(c) {
            case ',':
                if (!withinQuote) {
                    toks.add(str.substring(start, end));
                    start = end + 1;
                }
                break;
            case '"':
                withinQuote = !withinQuote;
                break;
        }
    }
    if (start < str.length()) {
        toks.add(str.substring(start));
    }
    return toks;
}

```

60.1. . How to use backreferences in Java Regular Expressions

Backreferences is another useful feature in Java regular expression.

61. “Simple Java” PDF Download

Simple Java Download Link (Old Version)

Simple Java Download Link (Latest Version)

62. Start from length & length() in Java

First of all, can you quickly answer the following question?

Without code autocompletion of any IDE, how to get the length of an array? And how to get the length of a String?

I asked this question to developers of different levels: entry and intermediate. They can not answer the question correctly or confidently. While IDE provides convenient code autocompletion, it also brings the problem of "surface understanding". In this post, I will explain some key concepts about Java arrays.

The answer:

```
int[] arr = new int[3];
System.out.println(arr.length);//length for array

String str = "abc";
System.out.println(str.length());//length() for string
```

The question is why array has the length field but string does not? Or why string has the length() method while array does not?

62.1. Q1. Why arrays have length property?

First of all, an array is a container object that holds a fixed number of values of a single type. After an array is created, its length never changes[1]. The array's length is available as a final instance variable length. Therefore, length can be considered as a defining attribute of an array.

An array can be created by two methods: 1) an array creation expression and 2) an array initializer. When it is created, the size is specified.

An array creation expression is used in the example above. It specifies the element type, the number of levels of nested arrays, and the length of the array for at least one of the levels of nesting.

This declaration is also legal, since it specifies one of the levels of nesting.

```
int[][] arr = new int[3][];
```

An array initializer creates an array and provides initial values for all its components. It is written as a comma-separated list of expressions, enclosed by braces and .

For example,

```
int[] arr = {1,2,3};
```

62.2. Q2. Why there is not a class "Array" defined similarly like "String"?

Since an array is an object, the following code is legal.

```
Object obj = new int[10];
```

An array contains all the members inherited from class Object(except clone). Why there is not a class definition of an array? We can not find an Array.java file. A rough

explanation is that they're hidden from us. You can think about the question - if there IS a class Array, what would it look like? It would still need an array to hold the array data, right? Therefore, it is not a good idea to define such a class.

Actually we can get the class of an array by using the following code:

```
int[] arr = new int[3];  
System.out.println(arr.getClass());
```

Output:

```
class [I
```

"class [I" stands for the run-time type signature for the class object "array with component type int".

62.3. Q3. Why String has length() method?

The backup data structure of a String is a char array. There is no need to define a field that is not necessary for every application. Unlike C, an Array of characters is not a String in Java.

63. When and how a Java class is loaded and initialized?

In Java, you first write a .java file which is then compiled to .class file during compile time. Java is capable of loading classes at run time. The confusion is what is the difference between "load" and "initialize". When and how is a Java class loaded and initialized? It can be clearly illustrated by using a simple example below.

63.1. What does it mean by saying "load a class"?

C/C++ is compiled to native machine code first and then it requires a linking step after compilation. What the linking does is combining source files from different places and form an executable program. Java does not do that. The linking-like step for Java is done when they are loaded into JVM.

Different JVMs load classes in different ways, but the basic rule is only loading classes when they are needed. If there are some other classes that are required by the loaded class, they will also be loaded. The loading process is recursive.

63.2. When and how is a Java class loaded?

In Java, loading policies is handled by a ClassLoader. The following example shows how and when a class is loaded for a simple program.

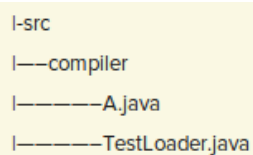
TestLoader.java

```
package compiler;
public class TestLoader {
    public static void main(String[] args) {
        System.out.println("test");
    }
}
```

A.java

```
package compiler;
public class A {
    public void method(){
        System.out.println("inside of A");
    }
}
```

Here is the directory hierarchy in eclipse:



```
I-src
I--compiler
I-----A.java
I-----TestLoader.java
```

By running the following command, we can get information about each class loaded. The "-verbose:class" option displays information about each class loaded.

```
java -verbose:class -classpath /home/ron/workspace/UltimateTest/bin/
compiler.TestLoader
```

Part of output:

```
[Loaded sun.misc.JavaSecurityProtectionDomainAccess from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded java.security.ProtectionDomain$2 from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded java.security.ProtectionDomain$Key from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded java.security.Principal from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded compiler.TestLoader from
  file:/home/xiwang/workspace/UltimateTest/bin/]
test
```



```
[Loaded java.lang.Shutdown from /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded java.lang.Shutdown$Lock from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
```

Now If we change TestLoader.java to:

```
package compiler;
public class TestLoader {
    public static void main(String[] args) {
        System.out.println("test");
        A a = new A();
        a.method();
    }
}
```

And run the same command again, the output would be:

```
[Loaded sun.misc.JavaSecurityProtectionDomainAccess from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded java.security.ProtectionDomain$2 from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded java.security.ProtectionDomain$Key from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded java.security.Principal from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded compiler.TestLoader from
  file:/home/xiwang/workspace/UltimateTest/bin/]
test
[Loaded compiler.A from file:/home/xiwang/workspace/UltimateTest/bin/]
inside of A
[Loaded java.lang.Shutdown from /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
[Loaded java.lang.Shutdown$Lock from
  /usr/local/java/jdk1.6.0_34/jre/lib/rt.jar]
```

We can see the difference highlighted in red. A.class is loaded only when it is used. In summary, a class is loaded:

- when the new bytecode is executed. For example, `SomeClass f = new SomeClass();`
- when the bytecodes make a static reference to a class. For example, `System.out`.

63.3. When and how is a Java class initialized?

A class is initialized when a symbol in the class is first used. When a class is loaded it is not initialized.

JVM will initialize superclass and fields in textual order, initialize static, final fields first, and give every field a default value before initialization.

[Java Class Instance Initialization](#) is an example that shows the order of execution for field, static field and constructor.

64. Java Thread: notify() and wait() examples

This article contains two code examples to demonstrate [Java concurrency](#). They stand for very typical usage. By understanding them, you will have a better understanding about notify() and wait().

64.1. Some background knowledge

- synchronized keyword is used for exclusive accessing.
- To make a method synchronized, simply add the synchronized keyword to its declaration. Then no two invocations of synchronized methods on the same object can interleave with each other.
- Synchronized statements must specify the object that provides the intrinsic lock. When synchronized(this) is used, you have to avoid to synchronizing invocations of other objects' methods.
- wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- notify() wakes up the first thread that called wait() on the same object.

64.2. notify() and wait() - example 1

```
public class ThreadA {
    public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();

        synchronized(b){
            try{
                System.out.println("Waiting for b to complete...");
                b.wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }

            System.out.println("Total is: " + b.total);
        }
    }
}
```

```

    }
}

class ThreadB extends Thread{
    int total;
    @Override
    public void run(){
        synchronized(this){
            for(int i=0; i<100 ; i++){
                total += i;
            }
            notify();
        }
    }
}

```

In the example above, an object, b, is synchronized. b completes the calculation before Main thread outputs its total value.

Output:

```

Waiting for b to complete...
Total is: 4950

```

If b is not synchronized like the code below:

```

public class ThreadA {
    public static void main(String[] args) {
        ThreadB b = new ThreadB();
        b.start();

        System.out.println("Total is: " + b.total);
    }
}

class ThreadB extends Thread {
    int total;

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            total += i;
        }
    }
}

```

The result would be 0, 10, etc. Because sum is not finished before it is used.

64.3. notify() and wait() - example 2

The second example is more complex, see the comments.

```
import java.util.Vector;

class Producer extends Thread {

    static final int MAXQUEUE = 5;
    private Vector messages = new Vector();

    @Override
    public void run() {
        try {
            while (true) {
                putMessage();
                //sleep(5000);
            }
        } catch (InterruptedException e) {
        }
    }

    private synchronized void putMessage() throws InterruptedException {
        while (messages.size() == MAXQUEUE) {
            wait();
        }
        messages.addElement(new java.util.Date().toString());
        System.out.println("put message");
        notify();
        //Later, when the necessary event happens, the thread that is running
        //it calls notify() from a block synchronized on the same object.
    }

    // Called by Consumer
    public synchronized String getMessage() throws InterruptedException {
        notify();
        while (messages.size() == 0) {
            wait(); //By executing wait() from a synchronized block, a thread
            //gives up its hold on the lock and goes to sleep.
        }
        String message = (String) messages.firstElement();
        messages.removeElement(message);
        return message;
    }
}

class Consumer extends Thread {

    Producer producer;
```

```

Consumer(Producer p) {
    producer = p;
}

@Override
public void run() {
    try {
        while (true) {
            String message = producer.getMessage();
            System.out.println("Got message: " + message);
            //sleep(200);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void main(String args[]) {
    Producer producer = new Producer();
    producer.start();
    new Consumer(producer).start();
}
}

```

A possible output sequence:

```

Got message: Fri Dec 02 21:37:21 EST 2011
put message
put message
put message
put message
put message
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
put message
put message
put message
put message
put message
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011

```

65. Should `.close()` be put in finally block or not?

The following are 3 different ways to close a output writer. The first one puts `close()` method in try clause, the second one puts `close` in finally clause, and the third one uses a try-with-resources statement. Which one is the right or the best?

`//close() is in try clause`

```
try {
    PrintWriter out = new PrintWriter(
        new BufferedWriter(
            new FileWriter("out.txt", true)));
    out.println("the text");
    out.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

`//close() is in finally clause`

```
PrintWriter out = null;
try {
    out = new PrintWriter(
        new BufferedWriter(
            new FileWriter("out.txt", true)));
    out.println("the text");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (out != null) {
        out.close();
    }
}
```

`//try-with-resource statement`

```
try (PrintWriter out2 = new PrintWriter(
    new BufferedWriter(
        new FileWriter("out.txt", true)))) {
    out2.println("the text");
} catch (IOException e) {
    e.printStackTrace();
}
```

65.1. Answer

Because the Writer should be closed in either case (exception or no exception), close() should be put in finally clause.

From Java 7, we can use `try-with-resources` statement.

66. Java Serialization

66.1. What is Serialization?

In Java, object serialization means representing an object as a sequence of bytes. The bytes includes the object's data and information. A serialized object can be written into a file/database, and read from the file/database and deserialized. The bytes that represent the object and its data can be used to recreate the object in memory.

66.2. Why Do We Need Serialization?

Serialization is usually used when you need to send object over network or stored in files. Network infrastructure and hard disk can only understand bits and bytes but not Java objects. Serialization translate Java objects to bytes and send it over network or save it.

Why we want to store or transfer an object? In my programming experience, I have the following reasons that motivate me to use serializable objects.

- An object creation depends on a lot of context. Once created, its methods as well as its fields are required for other components.
- When an object is created and it contains many fields, we are not sure what to use. So store it to database for later data analysis.

66.3. Java Serialization Example

The following example shows how to make a class serializable and serialize & deserialize it.

```
package serialization;

import java.io.Serializable;

public class Dog implements Serializable {
    private static final long serialVersionUID = -5742822984616863149L;

    private String name;
```

```

private String color;
private transient int weight;

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getColor() {
    return color;
}

public void setColor(String color) {
    this.color = color;
}

public int getWeight() {
    return weight;
}

public void setWeight(int weight) {
    this.weight = weight;
}

public void introduce() {
    System.out.println("I have a " + color + " " + name + ".");
}
}

```

```

package serialization;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializeDemo {
    public static void main(String[] args) {
        //create an object
        Dog e = new Dog();
        e.setName("bulldog");
        e.setColor("white");
        e.setWeight(5);

        //serialize
    }
}

```



```

try {
    FileOutputStream fileOut = new FileOutputStream("./dog.ser");
    ObjectOutputStream out = new ObjectOutputStream(fileOut);
    out.writeObject(e);
    out.close();
    fileOut.close();
    System.out.printf("Serialized dog is saved in ./dog.ser");
} catch (IOException i) {
    i.printStackTrace();
}

e = null;

//Deserialize
try {
    FileInputStream fileIn = new FileInputStream("./dog.ser");
    ObjectInputStream in = new ObjectInputStream(fileIn);
    e = (Dog) in.readObject();
    in.close();
    fileIn.close();
} catch (IOException i) {
    i.printStackTrace();
    return;
} catch (ClassNotFoundException c) {
    System.out.println("Dog class not found");
    c.printStackTrace();
    return;
}

System.out.println("\nDeserialized Dog ...");
System.out.println("Name: " + e.getName());
System.out.println("Color: " + e.getColor());
System.out.println("Weight: " + e.getWeight());

e.introduce();

}
}

```

Output:

```

Serialized dog is saved in ./dog.ser
Deserialized Dog...
Name: bulldog
Color: white
Weight: 0
I have a white bulldog.

```

67. Iteration vs. Recursion in Java

67.1. Recursion

Consider the factorial function: $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$

There are many ways to compute factorials. One way is that $n!$ is equal to $n \cdot (n-1)!$. Therefore the program can be directly written as:

Program 1:

```
int factorial (int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n*factorial(n-1);  
    }  
}
```

In order to run this program, the computer needs to build up a chain of multiplications: $\text{factorial}(n) \rightarrow \text{factorial}(n-1) \rightarrow \text{factorial}(n-2) \rightarrow \dots \rightarrow \text{factorial}(1)$. Therefore, the computer has to keep track of the multiplications to be performed later on. This type of program, characterized by a chain of operations, is called recursion. Recursion can be further categorized into linear and tree recursion. When the amount of information needed to keep track of the chain of operations grows linearly with the input, the recursion is called linear recursion. The computation of $n!$ is such a case, because the time required grows linearly with n . Another type of recursion, tree recursion, happens when the amount of information grows exponentially with the input. But we will leave it undiscussed here and go back shortly afterwards.

67.2. Iteration

A different perspective on computing factorials is by first multiplying 1 by 2, then multiplying the result by 3, then by 4, and so on until n . More formally, the program can use a counter that counts from 1 up to n and compute the product simultaneously until the counter exceeds n . Therefore the program can be written as:

Program 2:

```
int factorial (int n) {  
    int product = 1;  
    for(int i=2; i<n; i++) {  
        product *= i;  
    }  
    return product;  
}
```

This program, by contrast to program 1, does not build a chain of multiplication. At each step, the computer only needs to keep track of the current values of the product

and i. This type of program is called iteration, whose state can be summarized by a fixed number of variables, a fixed rule that describes how the variables should be updated, and an end test that specifies conditions under which the process should terminate. Same as recursion, when the time required grows linearly with the input, we call the iteration linear recursion.

67.3. Recursion vs Iteration

Compared the two processes, we can find that they seem almost same, especially in term of mathematical function. They both require a number of steps proportional to n to compute $n!$. On the other hand, when we consider the running processes of the two programs, they evolve quite differently.

In the iterative case, the program variables provide a complete description of the state. If we stopped the computation in the middle, to resume it only need to supply the computer with all variables. However, in the recursive process, information is maintained by the computer, therefore "hidden" to the program. This makes it almost impossible to resume the program after stopping it.

67.4. Tree recursion

As described above, tree recursion happens when the amount of information grows exponentially with the input. For instance, consider the sequence of Fibonacci numbers defined as follows:

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise} \end{cases}$$

By the definition, Fibonacci numbers have the following sequence, where each number is the sum of the previous two: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

A recursive program can be immediately written as:

Program 3:

```
int fib (int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Therefore, to compute `fib(5)`, the program computes `fib(4)` and `fib(3)`. To compute `fib(4)`, it computes `fib(3)` and `fib(2)`. Notice that the `fib` procedure calls itself twice at the last line. Two observations can be obtained from the definition and the program:

- The *i*th Fibonacci number `Fib(i)` is equal to $\text{phi}(i)/\text{rootsquare}(5)$ rounded to the nearest integer, which indicates that Fibonacci numbers grow exponentially.
- This is a bad way to compute Fibonacci numbers because it does redundant computation. Computing the running time of this procedure is beyond the scope of this article, but one can easily find that in books of algorithms, which is $O(\text{phi}(n))$. Thus, the program takes an amount of time that grows exponentially with the input.

On the other hand, we can also write the program in an iterative way for computing the Fibonacci numbers. Program 4 is a linear iteration. The difference in time required by Program 3 and 4 is enormous, even for small inputs.

Program 4:

```
int fib (int n) {
    int fib = 0;
    int a = 1;
    for(int i=0; i<n; i++) {
        fib = fib + a;
        a = fib;
    }
    return fib;
}
```

However, one should not think tree-recursive programs are useless. When we consider programs that operate on hierarchically data structures rather than numbers, tree-recursion is a natural and powerful tool. It can help us understand and design programs. Compared with Program 3 and 4, we can easily tell Program 3 is more straightforward, even if less efficient. After that, we can most likely reformulate the program into an iterative way.

68. 4 types of Java inner classes

There are 4 different types of inner classes you can use in Java. The following gives their name and examples.

68.1. Static Nested Classes

```
class Outer {
    static class Inner {
```

```

        void go() {
            System.out.println("Inner class reference is: " + this);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outer.Inner n = new Outer.Inner();
        n.go();
    }
}

```

```

Inner class reference is: Outer$Inner@19e7ce87

```

68.2. Member Inner Class

Member class is instance-specific. It has access to all methods, fields, and the Outer's this reference.

```

public class Outer {
    private int x = 100;

    public void makeInner(){
        Inner in = new Inner();
        in.seeOuter();
    }

    class Inner{
        public void seeOuter(){
            System.out.println("Outer x is " + x);
            System.out.println("Inner class reference is " + this);
            System.out.println("Outer class reference is " + Outer.this);
        }
    }

    public static void main(String [] args){
        Outer o = new Outer();
        Inner i = o.new Inner();
        i.seeOuter();
    }
}

```

```

Outer x is 100
Inner class reference is Outer$Inner@4dfd9726
Outer class reference is Outer@43ce67ca

```

68.3. Method-Local Inner Classes

```
public class Outer {  
    private String x = "outer";  
  
    public void doStuff() {  
        class MyInner {  
            public void seeOuter() {  
                System.out.println("x is " + x);  
            }  
        }  
  
        MyInner i = new MyInner();  
        i.seeOuter();  
    }  
  
    public static void main(String[] args) {  
        Outer o = new Outer();  
        o.doStuff();  
    }  
}
```

x is outer

```
public class Outer {  
    private static String x = "static outer";  
  
    public static void doStuff() {  
        class MyInner {  
            public void seeOuter() {  
                System.out.println("x is " + x);  
            }  
        }  
  
        MyInner i = new MyInner();  
        i.seeOuter();  
    }  
  
    public static void main(String[] args) {  
        Outer.doStuff();  
    }  
}
```

x is static outer

68.4. Anonymous Inner Classes

This is frequently used when you add an action listener to a widget in a GUI application.

```
button.addActionListener(new ActionListener(){  
    public void actionPerformed(ActionEvent e){  
        comp.setText("Button has been clicked");  
    }  
});
```
