Simple Java

X Wang

Version 1.0
Published In The Wild

CONTENTS

Part I

FREFACE

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

The large amount of traffic indicates a more important fact than that my writing skills are good(which is not the case): 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 developers prefer.

Many people believe in that diagrams are easier to understand things. While visualization is a good way to understand and remember things, there are other ways to enhance a 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 will find a convenient collection of most popular posts; second, developers who want to read something that is more than words. Repetition is the key of learning any programming language. 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 was converted automatically from the original blog posts. Every title in the book is linked back to the original blog. When the title is clicked, it opens the original post in your browser. If you find any problems, please go to the post and leave your comment there. As it is an automatic conversion, there may be some formatting problems. Please leave a comment if you find one. You can also contact me by email: contact@programcreek.com. Thank you for downloading this PDF!

Chrismas Day 2013

Part II

JAVA QUESTIONS

1

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 stuff. 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

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

1.1   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.

1.2   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 an entrance, this is not realistic. 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.

1.3   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 09   .......2.,......
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/lan
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 76   ......out...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 6F   .........java/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 4C   ....println...(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 compiler; the other is main method.

Below each method, there are a sequence of instructions, such as aload_0, invoke-special #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.

1.4   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.
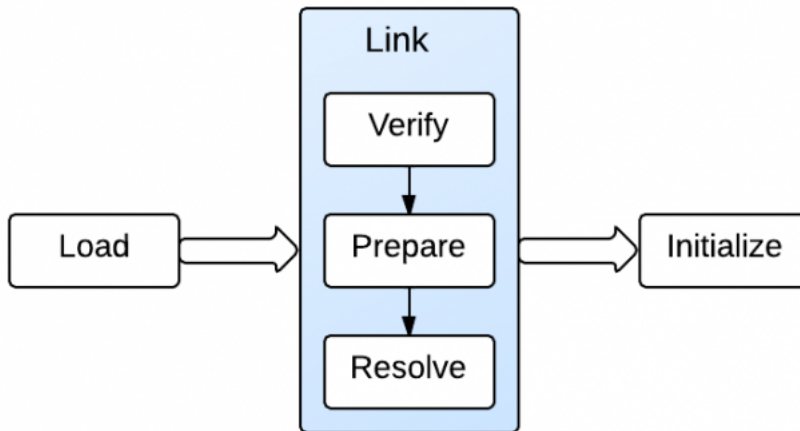
This loading job is done by Java Classloaders. When the JVM is started, three class loaders are used:

- 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.

2

HOW TO BUILD YOUR OWN JAVA LIBRARY?

Code reuse is one of the most important factors in software development. It is a VERY good idea to put frequently-used functions together and build a library for yourself. Whenever some method is used, just simply make a method invocation. For Java, it's straightforward to manage such a library. Here a simple example in Eclipse. The library will contain only one "add" method for demo purpose.

Step 1: Create a "Java Project" named as "MyMath", and a simple "add" method under "Simple" class.

Package structure is as follows:

Simple.java

```java
public class Simple {
        public static int add(int a, int b){
                return a+b;
        }
}
```

Step 2: Export as a .jar file.

Right Click the project and select "export", a window is show as follows.
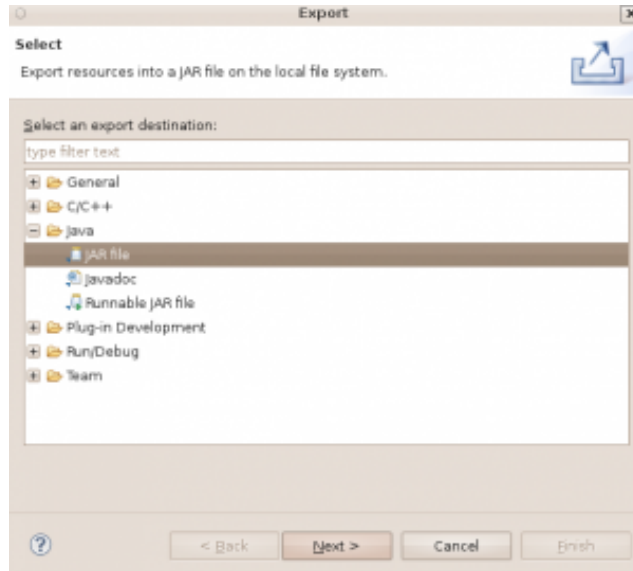
Following the wizard, to get the .jar file.

Step 3: Use the jar file.

Right click the target project, and select "Build Path"->"Add External Archives"->following wizard to add the jar file.

Now you can make a simple method call.

```java
public class Main {
        public static void main(String[] args) {
                System.out.println(Simple.add(1, 2));
        }
}
```

Last, but not the least, the library should be constantly updated and optimized. Documentation is important. If the library is not documented well, you may totally forget a function you programmed a year ago. Proper package names should be used to indicate the function of classes and methods. For example, you can name first layer of the packages by following the package names of standard Java library: programcreek.util, programcreek.io, programcreek.math, programcreek.text, etc. You domain specific knowledge then can be used in the next level. In addition, always do enough research first and make sure there is no imple-

ments of what you want to do before you start program anything. Libraries from industry utilizes the power of thousands of smart developers.

3

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.

3.1 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.

3.2 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 Some-Class();

- when the bytecodes make a static reference to a class. For example, System.out.

3.3 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.

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.

4.1 CODE EXAMPLE

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

```java
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).

4.2 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
```

5

JAVA DOUBLE EXAMPLE

Have you ever met the situation that you get an integer but you really want a double.

For the following method, devide(2,3) will return 0.0.

```java
public static double devide(int x, int y){
        return x/y;
}
```

The problem is that x/y does int division. If you want it to do double division, you can cast one of the operand. Both (double)x/y and x/(double)y will work.

```java
public static double devide(int x, int y){
        return (double)x/y;
}
```

Very often you want to round a double. There are multiple ways to do it, the following is a commonly used and simple method. If you want to round the result to 2 digits, you can use the following code:

```java
public static double devide(int x, int y){
        double z= (double)x/y;
        double pro = Math.round(z * 100);
        return pro/100;
}
```

DIAGRAM TO SHOW JAVA STRING'S IMMUTABILITY

Here are a set of diagrams to explain Java String's immutability.

6.1 DECLARE A STRING

```
String s = "abcd";
```

s stores the reference of the string object. The arrow below should be interpreted as "store reference of".
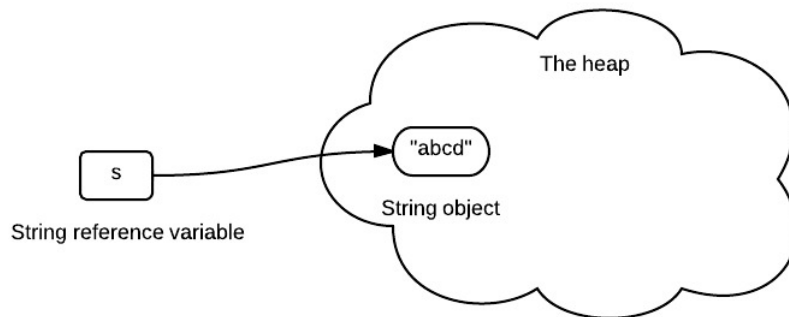
6.2 ASSIGN A STRING VARIABLE TO ANOTHER STRING VARIABLE

```
String s2 = s;
```

s2 stores the same reference value, since it is the same string object.
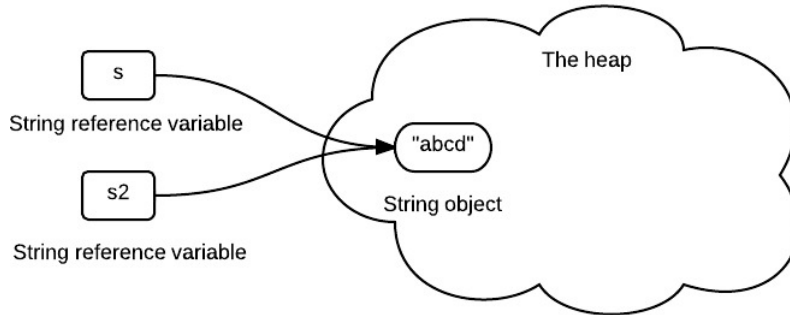
6.3 CONCAT STRING

```
s = s.concat("ef");
```

s now stores the reference of newly created string object.
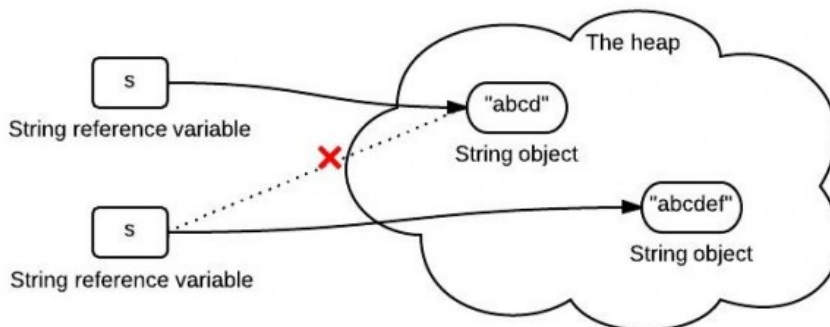
6.4   SUMMARY

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.

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.

7.1 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
```

7.2 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:
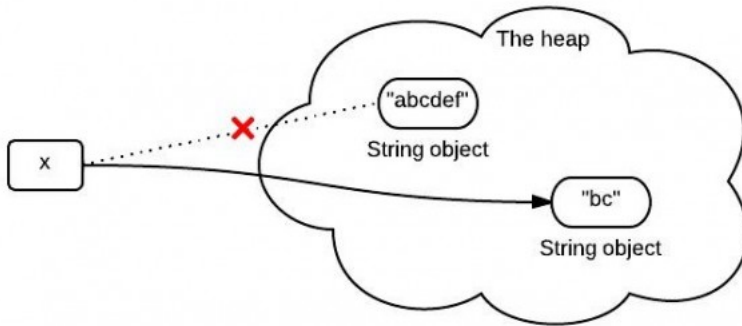
However, this diagram is not exactly right or it represents what really happens in the heap. What really happens when substring() is called is different between JDK 6 and JDK 7.

7.3 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.
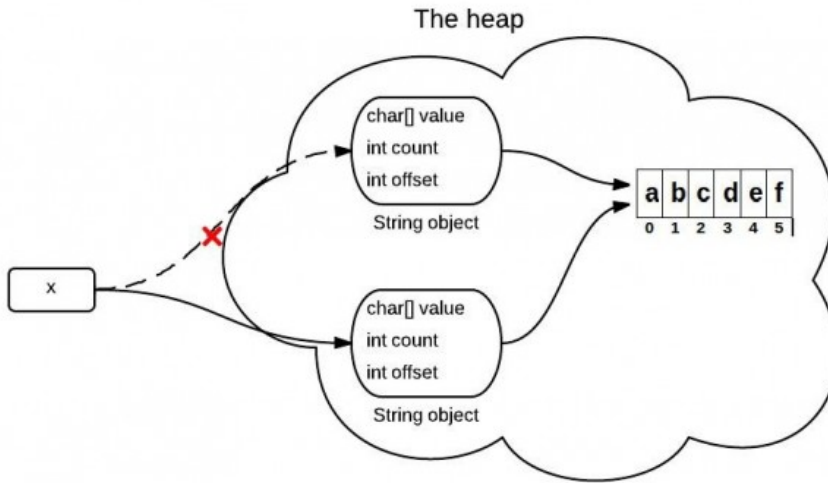
The following code is simplified and only contains the key point for explain this problem.

```
//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);
}
```

7.4 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) + ""
```

7.5 SUBSTRING() IN JDK 7

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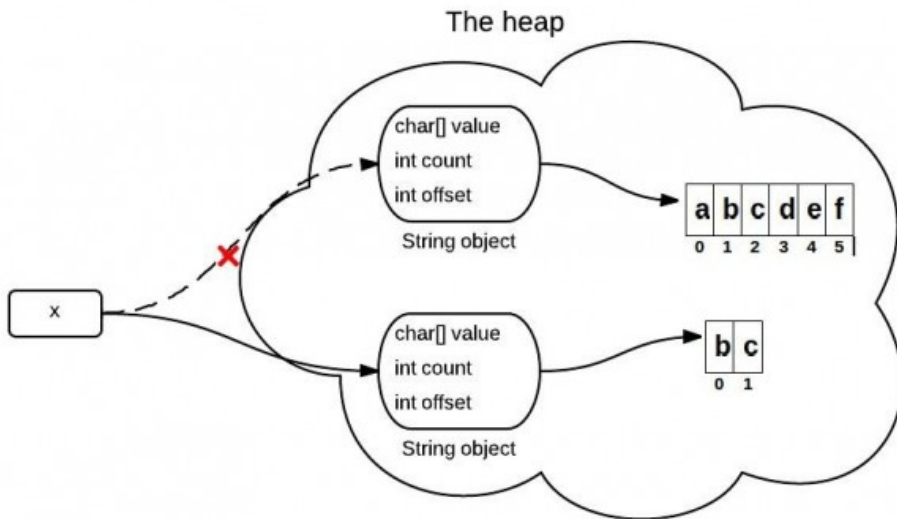

```
//JDK 7
public String(char value[], int offset, int count) {
        //check boundary
        this.value = Arrays.copyOfRange(value, offset, offset +
            count);
}

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.

WHY STRING IS IMMUTABLE IN JAVA ?

This is an old yet still popular question. There are multiple reasons that String is designed to be immutable in Java. A good answer depends on good understanding of memory, synchronization, data structures, etc. In the following, I will summarize some answers.

8.1 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";
```
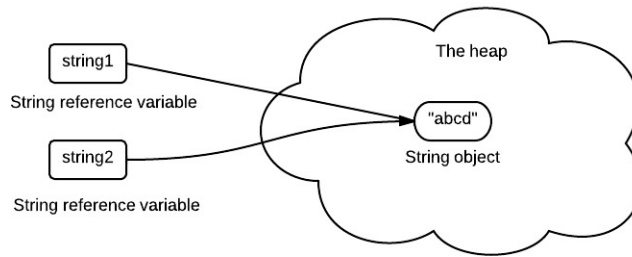
If string is not immutable, changing the string with one reference will lead to the wrong value for the other references.

8.2 ALLOW STRING TO CACHE ITS HASHCODE

The hashcode of string is frequently used in Java. For example, in a HashMap. Being immutable guarantees that hashcode will always the same, so that it can be cashed without worrying the 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.
```

8.3 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);
}
```

In summary, the reasons include design, efficiency, and security. Actually, this is also true for many other "why" questions in a Java interview.

STRING IS PASSED BY "REFERENCE" IN JAVA

This is a classic question of Java. Many similar questions have been asked on stackoverflow, 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.

9.1 A CODE FRAGMENT THAT IS INTERESTING & CONFUSING

```java
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:

```cpp
void change(string &x) {
    x = "cd";
}

int main(){
    string x = "ab";
    change(x);
```

```
    cout << x << endl;
}
```

it prints "cd".

9.2   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:
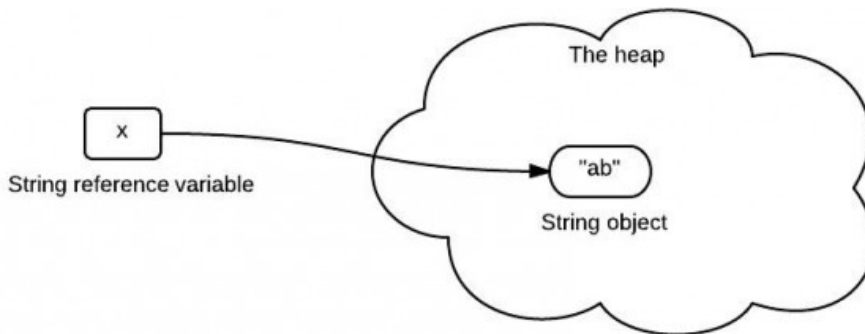
Because java is pass-by-value, the value of x is the reference to "ab". When the method change() gets invoked, it creates a new "cd" object, and x now is pointing to "cd" like the following:
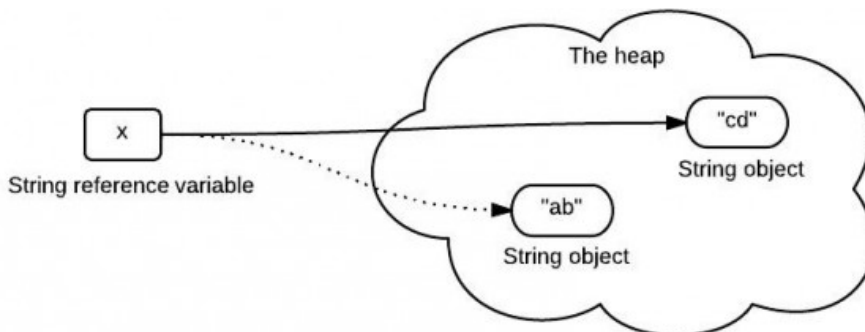
It seems to be a pretty reasonable explanation. They are clear that Java is always pass-by-value. But what is wrong here?

9.3   WHAT THE CODE REALLY DOES?

The explanation above has several mistakes. To understand this easily, it is a good idea to briefly walk though 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.
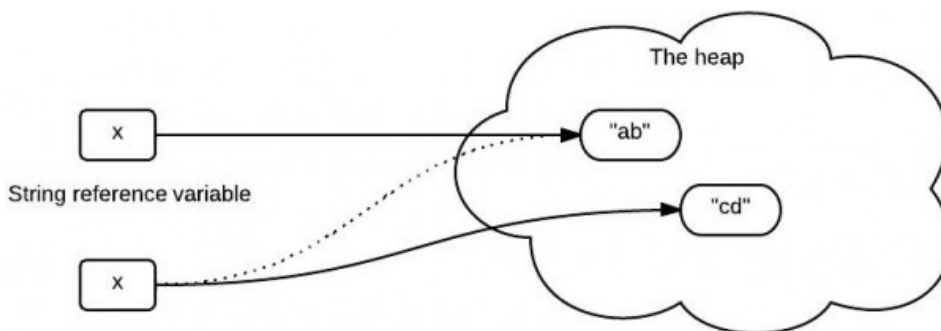
9.4   THE WRONG EXPLANATION

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!

9.5   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.

```java
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");
}
```

10

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:

```java
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?

10.1   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};
```

10.2   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".

10.3   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.

11

WHAT EXACTLY IS NULL IN JAVA?

Let's start from the following statement:

```
String x = null;
```

11.1 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 variables "x". "x" stores String reference. It is null here.

The following visualization gives a better sense about this concept.
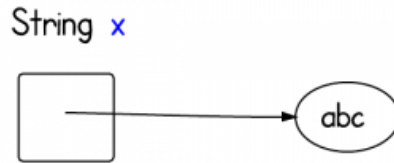
11.2 WHAT EXACTLY IS NULL IN MEMORY?

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 of something similar like itis on other C like languages.

11.3 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 steak, the local variable are located on that frame.

12

COMPARABLE VS COMPARATOR IN JAVA

Comparable and Comparator are two interfaces provided by Java Core API. From their names, you can tell that 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.

12.1 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 to show the usage:

```java
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) {
```

```java
                        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.
```

Comparator is capable of comparing two DIFFERENT types of objects. The method required for implementation is compare(). Now let's use another way to compare those TVs size. The common use of Comparator is sorting. Both Collections and Arrays classes provide a sort method which use a Comparator.

```java
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();
```

```java
                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:

```java
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]
```

JAVA EQUALS() AND HASHCODE() CONTRACT
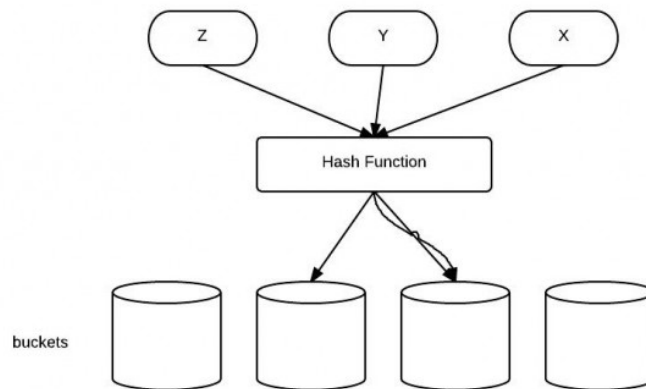
The Java super class java.lang.Object has two very important methods defined:

```
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.

13.1   A COMMON MISTAKE

Common mistake is shown in the example below.

```java
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 == ((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:
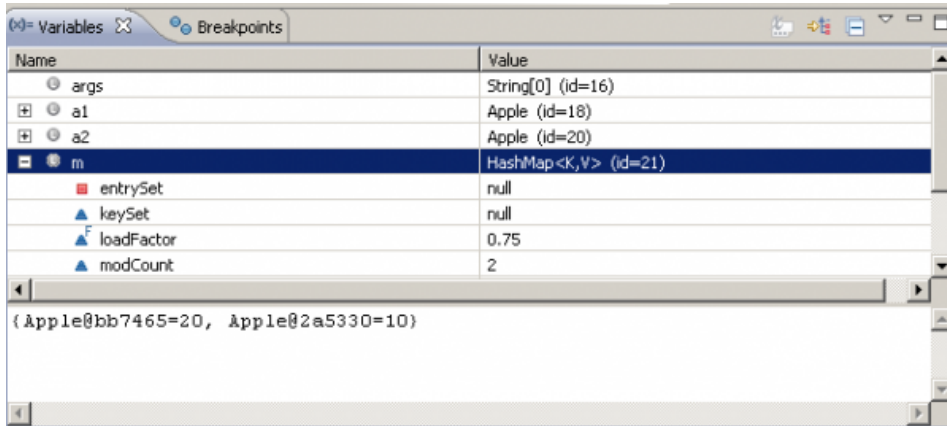
13.2 PROBLEM CAUSED BY HASHCODE()

The problem is caused by the un-overridden method "hashCode()". The contract between equals() and hasCode() 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.

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

```
}
```

14

OVERRIDING AND OVERLOADING IN JAVA WITH EXAMPLES

14.1 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. 2. Reference type determines which overloaded method will be used at compile time. 3. Polymorphism applies to overriding, not to overloading.

14.2 EXAMPLE OF OVERRIDING

Here is an example of overriding. After reading the code, guess the output. Easy!

```java
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 Main
{
    public static void main(String [] args){
        new Main().go();
    }
    void go(){
        new Hound().bark();
        ((Dog) new Hound()).bark();
        //((Dog) new Hound()).sniff();
    }
}
```

Output? Yes, here you go.

```
bowl
bowl
```

A better example:

```
class Animal {
        void stinky() {
                System.out.println("stinky animal !");
        }
}

class Dog extends Animal {
        public void stinky() {
                System.out.println("stinky dog !");
        }

        public void bark() {
                System.out.println("wow wow");
        }
}

class Cow extends Animal {
        public void stinky() {
                System.out.println("stinky cow !");
        }
}

public class TestOverriding {
```

```
        public static void main(String[] args) {
                Animal obj = new Dog();
                obj.stinky();
        }
}
```

When you create object like the above and call a method:

```
Animal obj = new Dog();
obj.stinky();
```

What compiler does is that it checks the class type of object which is Animal here. After that it checks whether the stinky() exist in Animal or not. Always remember that objects are created at run-time. So compiler has no way to know that the Dog class stinky() method is to be called. So at compile time class type of reference variable is checked to check such a method exist or not.

Now at run-time, the JVM knows that though the class type of obj is Animal, at run time it is referring to the object of Dog. So it calls the stinky() of Dog class. This is called Dynamic Polymorphism.

WHAT IS INSTANCE INITIALIZER IN JAVA?

In this post, an example is first given to illustrate what are instance variable initializer, instance initializer and static initializer. Then how instance initializer works is explained.

15.1 EXECUTION ORDER

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

```java
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
```

15.2 HOW DOES JAVA INSTANCE INITIALIZER WORK?

The instance initializer above contains a print statement. To understand how it works, we can think of it as a variable assignment statement(e.g. b = 0), then this would not difficult to understand.

Instead of

```
int b = 0
```

, you could write

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

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

15.3 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?)

16

WHY FIELD CAN'T BE OVERRIDDEN?

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

16.1 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.

```java
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"?

16.2 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 variables/class fields cannot be overridden like methods. When subclass defines a field with same name, it just declares a new field. Therefore, they can not be accessed polymorphically. They can not be overridden, which also means they are hidden and can be access though some ways.

16.3 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);
```

17

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.

17.1 STATIC NESTED CLASSES

```java
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
```

17.2 MEMBER INNER CLASS

Member class is instance-specific. It has access to all methods, fields, and the Outer's this reference.

```java
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
```

17.3 METHOD-LOCAL INNER CLASSES

```java
public class Outer {
        private String x = "outer";

        public void doStuff() {
                class MyInner {
                        public void seeOuter() {
                                System.out.println("x is " + x);
                        }
                }
```

```java
                MyInner i = new MyInner();
                i.seeOuter();
        }

        public static void main(String[] args) {
                Outer o = new Outer();
                o.doStuff();
        }
}
```

```
x is outer
```

```java
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
```

17.4 ANONYMOUS INNER CLASSES

This is frequently used when you add an action listener to a widget in a GUI application.

```java
button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        comp.setText("Button_has_been_clicked");
    }
});
```

18

WHAT IS INNER INTERFACE IN JAVA?

18.1 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();
}
```

18.2 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.

18.3   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.

```java
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);
    }
}
```

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

18.4 A SIMPLE EXAMPLE OF INNER INTERFACE?

Map.java

```java
public interface Map {
```

```java
        interface Entry{
                int getKey();
        }

        void clear();
}
```

MapImpl.java

```java
public class MapImpl implements Map {


        class ImplEntry implements Map.Entry{
                public int getKey() {
                        return 0;
                }
        }

        @Override
        public void clear() {
                //clear
        }
}
```

19

CONSTRUCTORS OF SUB AND SUPER CLASSES IN JAVA?

This post summarizes a commonly asked question about Java constructors.

19.1 WHY CREATING AN OBJECT OF THE SUB CLASS INVOKES ALSO THE CONSTRUCTOR OF THE SUPER CLASS?

```java
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:

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

19.2 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 one for the class, which is argument-less. If a constructor is defined, e.g. Super(String s), compiler will not insert the default argument-less one. This is the situation for the Super class above.

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 add the following Super() constructor to the Super class, OR remove the self-defined Super constructor.

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

19.3 EXPLICITLY CALL SUPER CONSTRUCTOR IN SUB CONSTRUCTOR

The following code is OK:

```java
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.

19.4   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.

19.5   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

JAVA ACCESS LEVEL FOR MEMBERS: PUBLIC, PROTECTED, PRIVATE

Java access level contains two parts: class level and member level. For class level, it can be public or no explicit modifier(package-private). For member access level, it can be public, private, protected, or package-private (no explicit modifier).

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

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

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.

```java
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() {
        }
}
```

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.

22.1 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.

```java
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();
    }
}
```

22.2 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:

```java
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.

23

DIAGRAM OF EXCEPTION HIERARCHY

In Java, exception can be checked or unchecked. They both fit into a class hierarchy. The following diagram shows 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 uncheck exceptions. They are exceptions that are not expected to be recovered, such as null pointer, divide by 0, etc.
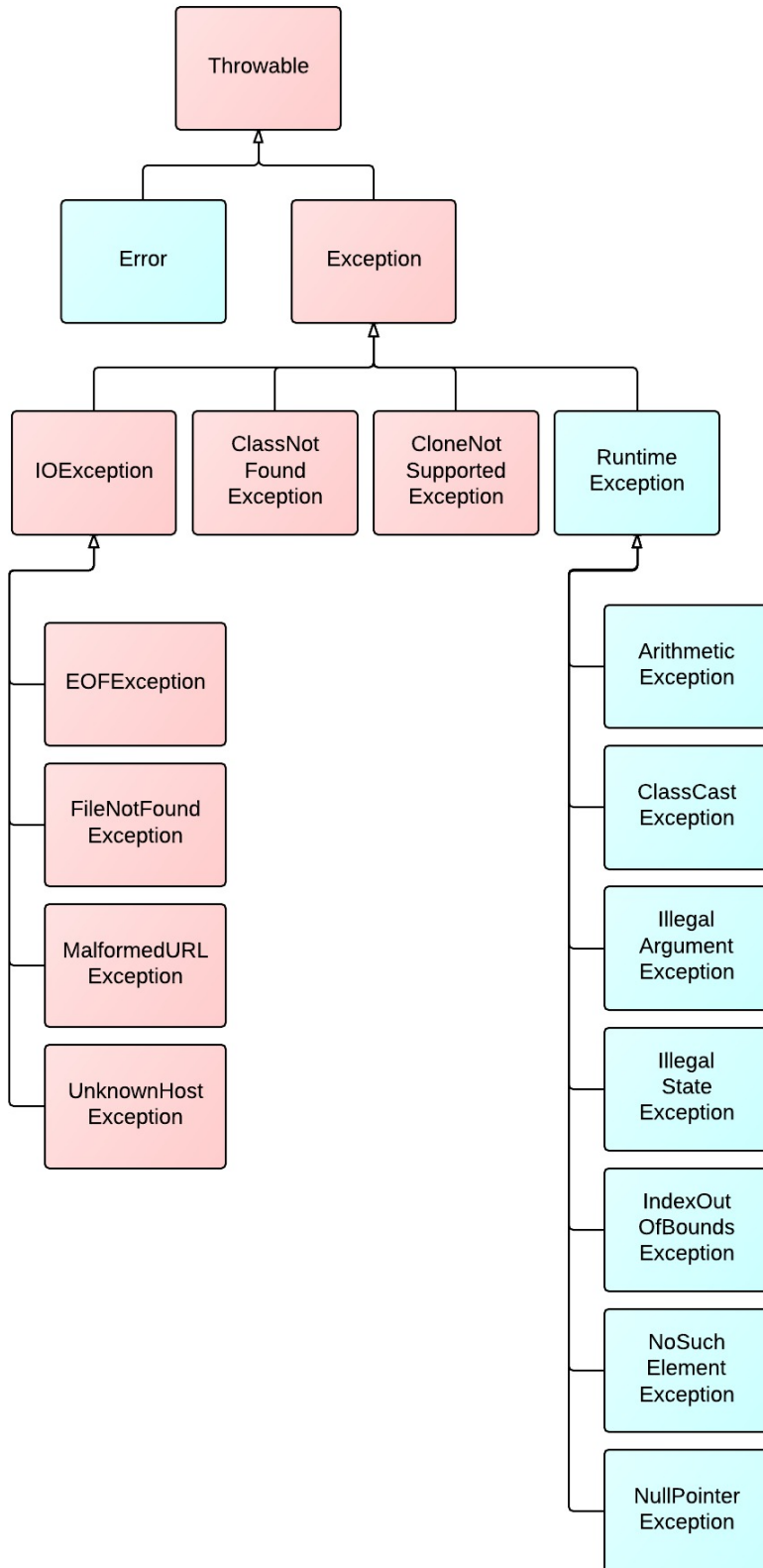
Check out top 10 questions about Java exceptions.

24

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:

```java
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:

```java
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/.

25

JAVA WRITE TO A FILE - CODE EXAMPLE

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.

```java
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:

```java
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 throws 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.

26

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.

27

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?

```java
//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();
}
```

```java
//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();
        }
}
```

```java
//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();
}
```

27.1 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.

28

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.
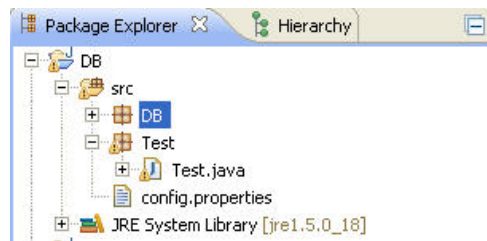
2. The following is the code.

```java
package Test;

import java.io.IOException;
import java.util.Properties;

public class Test {

        public static void main(String[] args) {
                Properties configFile = new Properties();
                try {
```

```java
                    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".

29

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".

29.1 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.
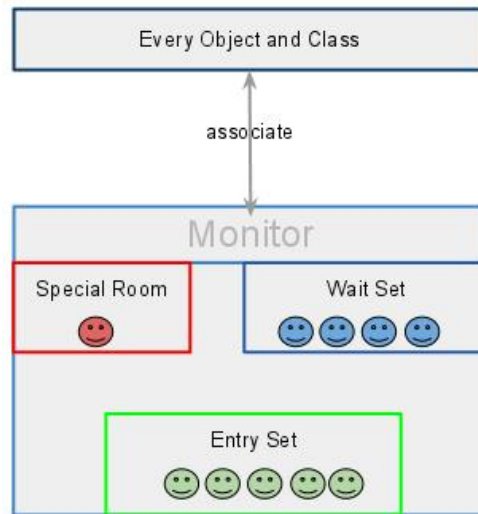
In brief, a monitor is a facility which monitors the threads' access to the special room. It ensures that only one thread can access the protected data or code.

29.2 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.

29.3   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.

THE INTERFACE AND CLASS HIERARCHY DIAGRAM OF JAVA COLLECTIONS

30.1 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.
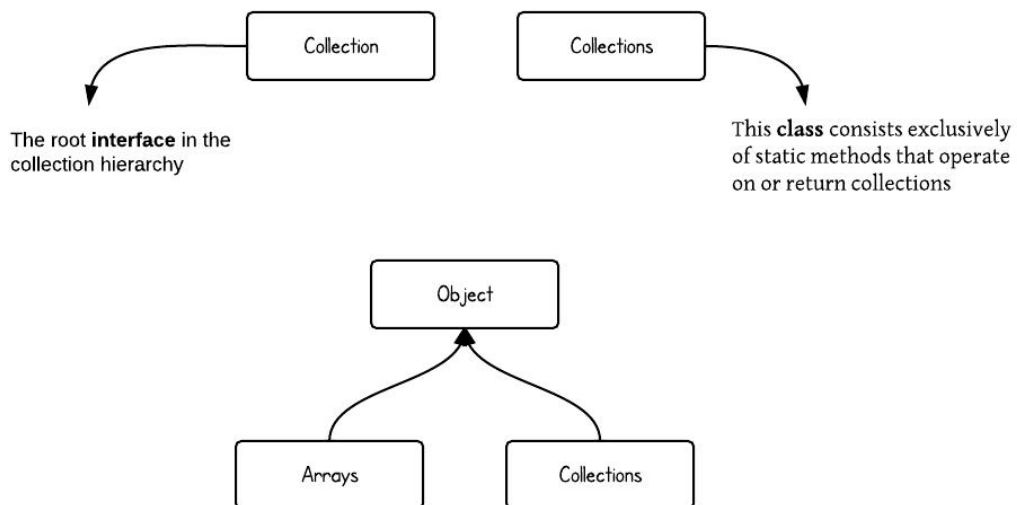
30.2   CLASS HIERARCHY OF COLLECTION

The following diagram demonstrates class hierarchy of Collection.
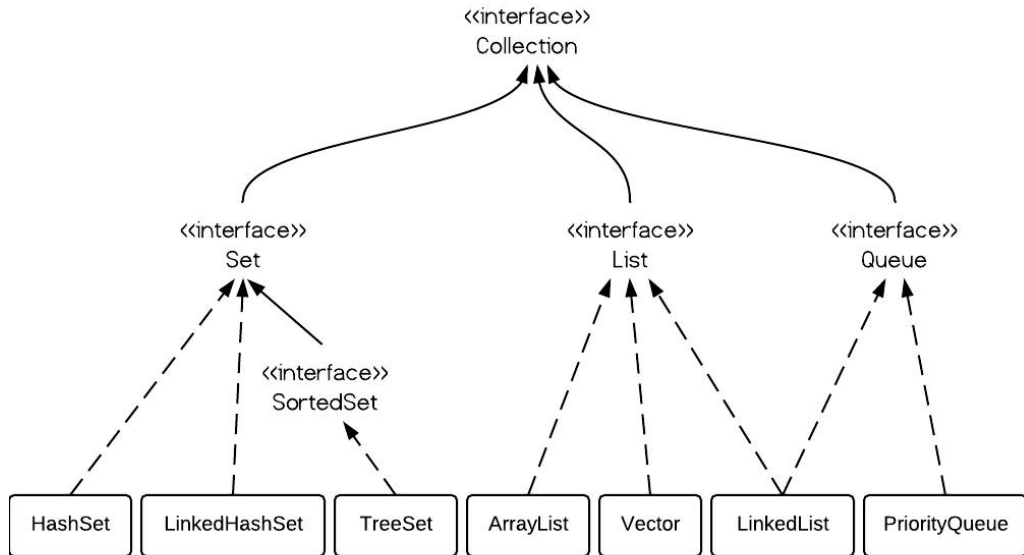
30.3   CLASS HIERARCHY OF MAP

Here is class hierarchy of Map.
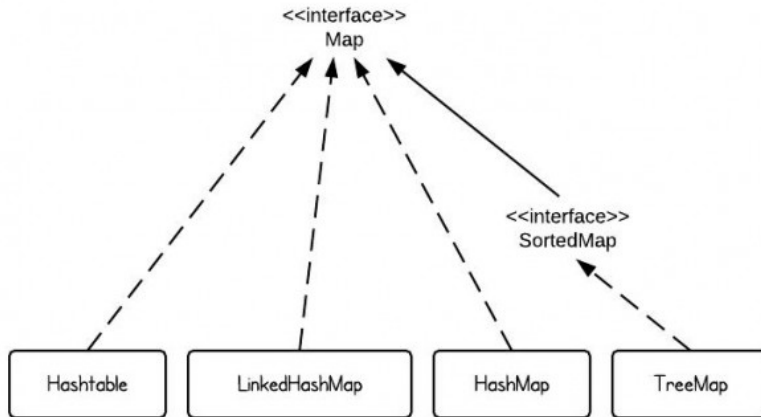
30.4   SUMMARY OF CLASSES

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

30.5   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}
```

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.

```java
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();
```

```java
        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 .
```

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 though the key information for deeper understanding of this method.

32.1 A SIMPLE EXAMPLE SHOWING HOW TO USE ARRAYS.SORT()

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.

```java
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;
        }
}
```

```java
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
```

32.2 STRATEGY PATTERN USED

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.

```java
class Dog{
        int size;
        int weight;
```

```java
        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.

32.3 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.

```java
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>{
```

```java
        @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
```

32.4   SUMMARY

To summarize, the takeaway messages from Arrays.sort():

- generic - super

- strategy pattern

- merge sort - nlog(n) time complexity

- Java.util.Collections#sort(List <T >list, Comparator <? super T >c) has similar idea with Arrays.sort.

ARRAYLIST VS. LINKEDLIST VS. VECTOR

33.1 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. A Set is a set of unique and unordered elements. The following is the class hierarchy diagram of Collection. From that you have a general idea of what I'm talking about.
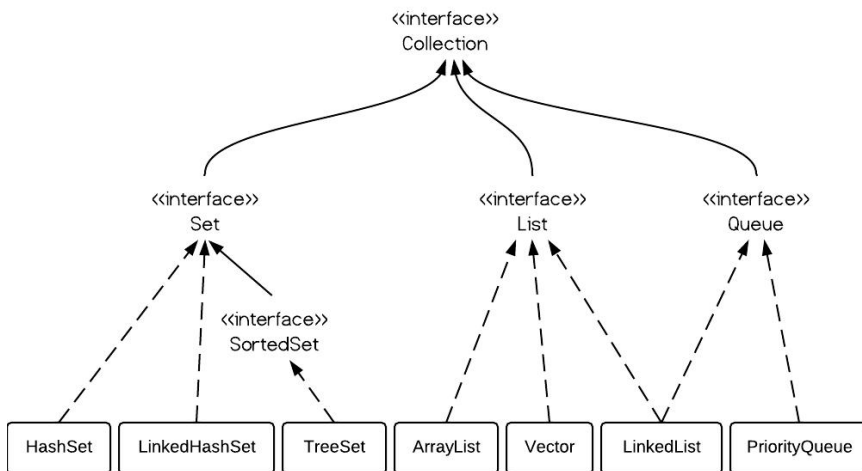
33.2   ARRAYLIST VS. LINKEDLIST VS. VECTOR

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. It's 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 space as more elements are added. Vector each time doubles its array size, while ArrayList grow 50

LinkedList, however, also implements Queue interface which adds more methods than ArrayList and Vector, such as offer(), peek(), poll(), etc.

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.

33.3   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());
}
```

33.4 LINKEDLIST EXAMPLE

```java
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.

33.5 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.

33.6 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:

```java
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

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 Linked-HashSet. 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 be store the insertion order, LinkedHashSet should be used.

34.1 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.
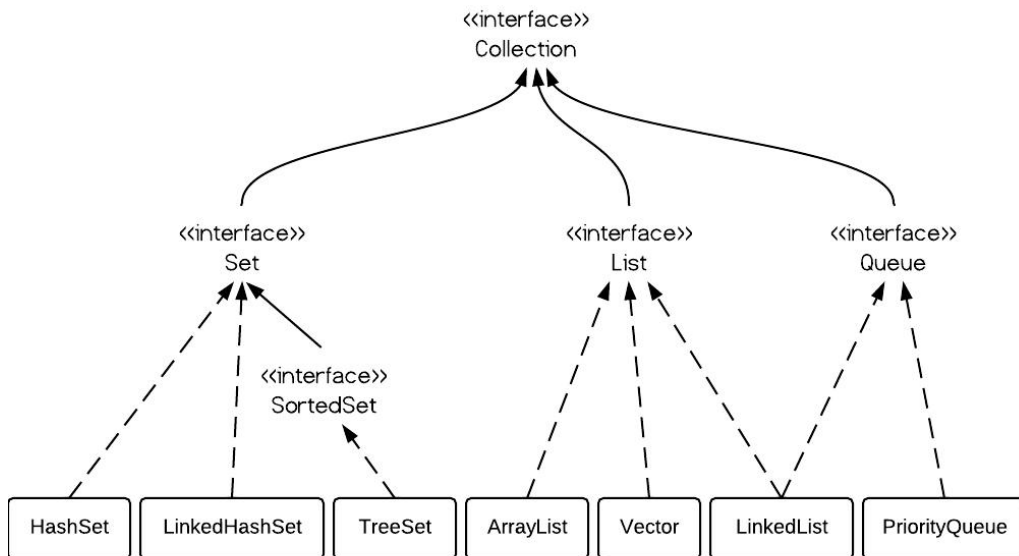
34.2 HASHSET VS. TREESET VS. LINKEDHASHSET

HashSet is Implemented using a hash table. Elements are not ordered. The add, remove, and contains methods has 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).

34.3 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:

```java
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
```

34.4 HASHSET EXAMPLE

```java
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.

34.5 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
```

34.6 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 wee that HashSet is the fastest one.

```
HashSet:  2244768
TreeSet:  3549314
LinkedHashSet:  2263320
```

* The test is not precise, but can reflect the basic idea.

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 Linked-HashMap.

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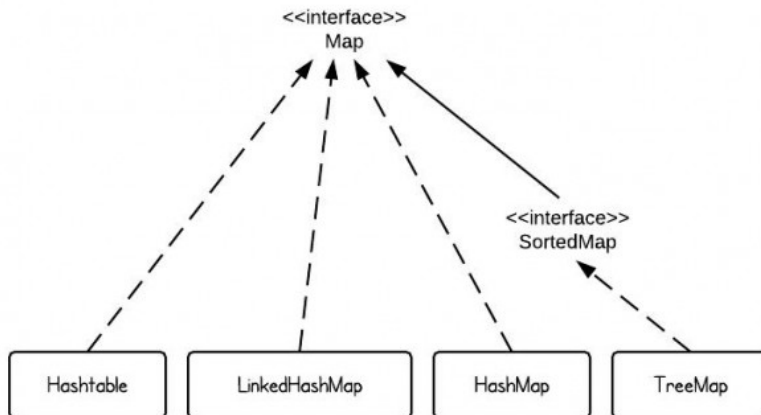
There are 4 commonly used implementations of Map in Java SE - HashMap, TreeMap, Hashtable and LinkedHashMap. If we use one sentence to describe each implementation, it would be the following:

- 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.

35.2 HASHMAP

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

```java
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 == 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.

35.3 TREEMAP

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

```java
class Dog {
        String  color;

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

35.4   HASHTABLE

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

35.5   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.

```java
class Dog {
        String color;

        Dog(String c) {
                color = c;
        }

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

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

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

public class TestHashMap {
```

```java
        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
```

36

EFFICIENT COUNTER IN JAVA

You may often use HashMap as a counter to understand the frequency of something from database or text. This articles compares the 3 different approaches to implement a counter by using HashMap.

36.1 THE NAIVE COUNTER

If you use such a counter, your code may look like the following:

```java
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

36.2   THE BETTER COUNTER

Naturally we want a mutable integer to avoid creating many Integer objects. A mutable integer class is defined as follows:

```java
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 convinently
        public String toString(){
                return Integer.toString(val);
        }
}
```

And the counter is improved and changed to the following:

```java
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.

36.3   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);
        }
}
```

36.4   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 very much difference between Naive and Better, which indicates that creating objects are expensive!

```java
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;
```

```java
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.

36.5 COMMENT FROM KEITH(FROM COMMENT LIST BELOW)

One of the best comments I've ever received!

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.

36.6   CONCLUSION SO FAR

The winner is the last one which uses int arrays.

37

FREQUENTLY USED METHODS OF JAVA HASHMAP

HashMap is very useful when a counter is required.

```java
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);
}
```

37.1 LOOP THROUGH HASHMAP

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

```java
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());
}
```

37.2 PRINT HASHMAP

```java
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
    }
}
```

37.3 SORT HASHMAP BY VALUE

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

```java
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 o would merge keys
    }
}
```

```java
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.

38

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.

38.1 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?

```java
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)
```

38.2 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.

38.3 WILDCARDS AND BOUNDED WILDCARDS

List<? >- List can contain any type

```java
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
    .
```

38.4 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'll get an ArrayStore-Exception during run-time.

<div style="text-align: right; font-size: 3em;">39</div>

WHY DO WE NEED GENERIC TYPES IN JAVA?

Generic types are extensively used in Java collections. The fundamental question is why we need Generic Types in Java? Understanding this question will help you better understand related concepts. I will use a short and simple example to show why Generic is introduced to Java.

39.1 OVERVIEW OF GENERICS

The goal of implementing Generics is finding bugs in compilation stage, other than waiting for run-time. This can reduce a lot of time for debugging java program, because compile-time bugs are much easier to find and fix. From the beginning, we need to keep in mind that generic types only exist in compile-time. A lot of misunderstanding and confusion come from this.

39.2 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.

39.3   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);
        }
}
```
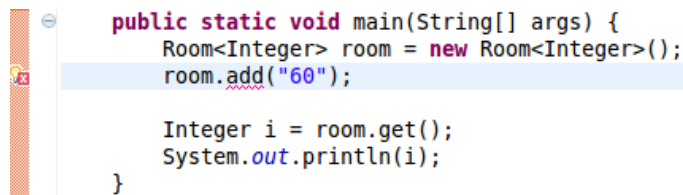
Now if someone adds room.add("60"), a compile-time error will be shown like the following:

```
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.

39.4 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.

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. Then what is the difference between them?

40.1 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:

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

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<?>();
```

40.2   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<?>.

40.3 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 example:

```java
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;
}
```

41

HOW TO CONVERT ARRAY TO ARRAYLIST IN JAVA?

This is a question that is worth to take a look for myself, because it is one of the top viewed and voted questions in stackoverflow. The one who accidentally asks such a question could gain a lot of reputation which would enable him to do a lot of stuff on stackoverflow. This does not make sense so much for me, but let's take a look at the question first.

The question asks how to convert the following array to an ArrayList.

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

41.1 MOST POPULAR AND ACCEPTED ANSWER

The most popular and the accepted answer is the following:

```java
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 - ArrayList(Collection 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"

If the add() method is invoked NOW, the size of the elementData array is not large enough to home one more element. So it will be copied to a new larger array. As the code below indicates, the size grows 1.5 times of old array.

```java
public void ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        // minCapacity is usually close to size, so this is a
            win:
        elementData = Arrays.copyOf(elementData, newCapacity)
            ;
    }
}
```

41.2 NEXT POPULAR ANSWER

The next popular answer is:

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

It is not the best, because the size of the list returned from asList() is fixed. 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.

```java
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)
```

41.3 INDICATIONS OF THE QUESTION

The problem is not hard, and kind of interesting. Every Java programmer knows ArrayList, it is simple but 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 basically indicate the same solution. This is true for a lot of questions, I guess people just don't care, they like answering!

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.

42.1 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.

42.2 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;
}
```

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?
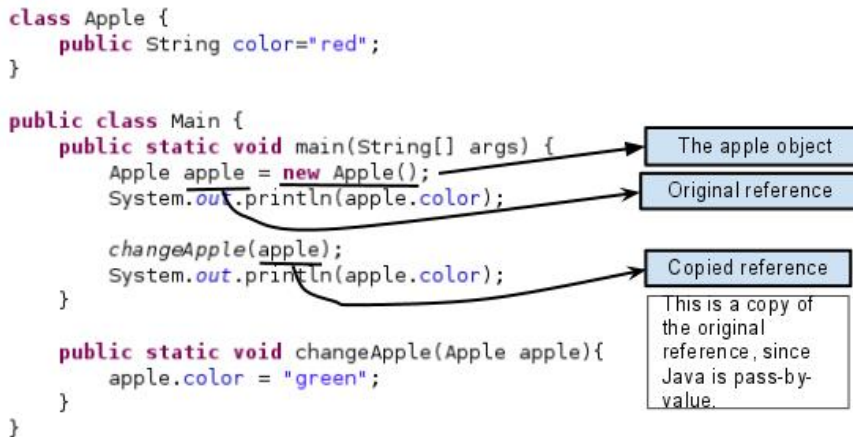
Code:

```java
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";
        }
}
```
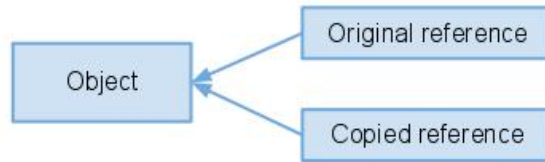
Since the orignal and copied reference refer the same object, the member value gets changed.

Output:

```
red
green
```

43

JAVA REFLECTION TUTORIAL

What is reflection, why is it useful, and how to use it?

43.1 WHAT IS REFLECTION?

"Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine." This concept is often mixed with introspection. The following are their definitions from Wiki:

- Introspection is the ability of a program to examine the type or properties of an object at runtime.

- Reflection is the ability of a program to examine and modify the structure and behavior of an object at runtime.

From their definitions, introspection is a subset of reflection. Some languages support introspection, but do not support reflection, e.g., C++.
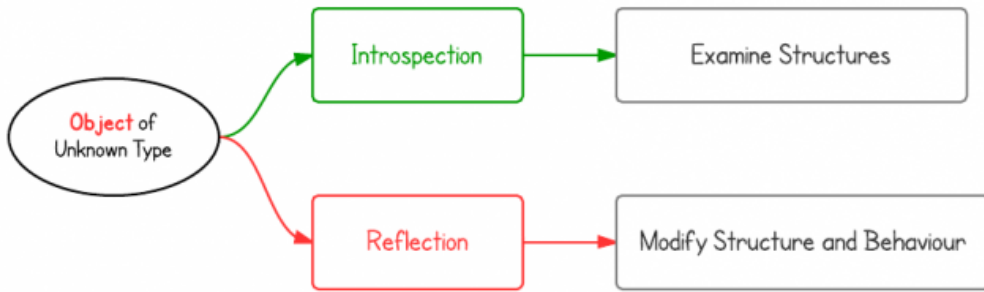
Introspection Example: The instanceof operator determines whether an object belongs to a particular class.

```java
if(obj instanceof Dog){
    Dog d = (Dog)obj;
    d.bark();
}
```

Reflection Example: The Class.forName() method returns the Class object associated with the class/interface with the given name(a string and full qualified name). The forName method causes the class with the name to be initialized.

```java
// with reflection
Class<?> c = Class.forName("classpath.and.classname");
Object dog = c.newInstance();
Method m = c.getDeclaredMethod("bark", new Class<?>[0]);
m.invoke(dog);
```

In Java, reflection is more about introspection, because you can not change structure of an object. There are some APIs to change accessibilities of methods and fields, but not structures.

43.2 WHY DO WE NEED REFLECTION?

Reflection enables us to:

- Examine an object's class at runtime

- Construct an object for a class at runtime

- Examine a class's field and method at runtime

- Invoke any method of an object at runtime

- Change accessibility flag of Constructor, Method and Field

- etc.

Reflection is the common approach of famework.

For example, JUnit use reflection to look through methods tagged with the @Test annotation, and then call those methods when running the unit test. (Here is a set of examples of how to use JUnit.)

For web frameworks, product developers define their own implementation of interfaces and classes and put is in the configuration files. Using reflection, it can quickly dynamically initialize the classes required.

For example, Spring uses bean configuration such as:

```
<bean id="someID" class="com.programcreek.Foo">
    <property name="someField" value="someValue" />
</bean>
```

When the Spring context processes this <bean >element, it will use Class.forName(String) with the argument "com.programcreek.Foo" to instantiate that Class. It will then again use reflection to get the appropriate setter for the <property >element and set its value to the specified value.

The same mechanism is also used for Servlet web applications:

```
<servlet>
    <servlet-name>someServlet</servlet-name>
    <servlet-class>com.programcreek.WhyReflectionServlet</servlet-
        class>
<servlet>
```

43.3   HOW TO USE REFLECTION?

How to use reflection API can be shown by using a small set of typical code examples.

Example 1: Get class name from object

```java
package myreflection;
import java.lang.reflect.Method;

public class ReflectionHelloWorld {
        public static void main(String[] args){
                Foo f = new Foo();
                System.out.println(f.getClass().getName());
        }
}

class Foo {
        public void print() {
                System.out.println("abc");
        }
}
```

Output:

```
myreflection.Foo
```

Example 2: Invoke method on unknown object

For the code example below, image the types of an object is unknown. By using reflection, the code can use the object and find out if the object has a method called "print" and then call it.

```java
package myreflection;
import java.lang.reflect.Method;

public class ReflectionHelloWorld {
        public static void main(String[] args){
                Foo f = new Foo();

                Method method;
                try {
                        method = f.getClass().getMethod("print",
                            new Class<?>[0]);
                        method.invoke(f);
                } catch (Exception e) {
                        e.printStackTrace();
                }

        }
}
```

```java
class Foo {
        public void print() {
                System.out.println("abc");
        }
}
```

```
abc
```

Example 3: Create object from Class instance

```java
package myreflection;

public class ReflectionHelloWorld {
        public static void main(String[] args){
                //create instance of "Class"
                Class<?> c = null;
                try{
                        c=Class.forName("myreflection.Foo");
                }catch(Exception e){
                        e.printStackTrace();
                }

                //create instance of "Foo"
                Foo f = null;

                try {
                        f = (Foo) c.newInstance();
                } catch (Exception e) {
                        e.printStackTrace();
                }

                f.print();
        }
}

class Foo {
        public void print() {
                System.out.println("abc");
        }
}
```

Example 4: Get constructor and create instance

```java
package myreflection;
```

```java
import java.lang.reflect.Constructor;

public class ReflectionHelloWorld {
        public static void main(String[] args){
                //create instance of "Class"
                Class<?> c = null;
                try{
                        c=Class.forName("myreflection.Foo");
                }catch(Exception e){
                        e.printStackTrace();
                }

                //create instance of "Foo"
                Foo f1 = null;
                Foo f2 = null;

                //get all constructors
                Constructor<?> cons[] = c.getConstructors();

                try {
                        f1 = (Foo) cons[0].newInstance();
                        f2 = (Foo) cons[1].newInstance("abc");
                } catch (Exception e) {
                        e.printStackTrace();
                }

                f1.print();
                f2.print();
        }
}

class Foo {
        String s;

        public Foo(){}

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

        public void print() {
                System.out.println(s);
```

```
        }
}
```

Output:

```
null
abc
```

In addition, you can use Class instance to get implemented interfaces, super class, declared field, etc.

Example 5: Change array size though reflection

```java
package myreflection;

import java.lang.reflect.Array;

public class ReflectionHelloWorld {
        public static void main(String[] args) {
                int[] intArray = { 1, 2, 3, 4, 5 };
                int[] newIntArray = (int[]) changeArraySize(
                    intArray, 10);
                print(newIntArray);

                String[] atr = { "a", "b", "c", "d", "e" };
                String[] str1 = (String[]) changeArraySize(atr,
                    10);
                print(str1);
        }

        // change array size
        public static Object changeArraySize(Object obj, int len)
          {
                Class<?> arr = obj.getClass().getComponentType();
                Object newArray = Array.newInstance(arr, len);

                //do array copy
                int co = Array.getLength(obj);
                System.arraycopy(obj, 0, newArray, 0, co);
                return newArray;
        }

        // print
        public static void print(Object obj) {
```

```
            Class <?> c = obj.getClass();
            if (!c.isArray()) {
                    return;
            }

            System.out.println("\nArray_length:_" + Array.
                getLength(obj));

            for (int i = 0; i < Array.getLength(obj); i++) {
                    System.out.print(Array.get(obj, i) + "_");
            }
        }
}
```

Output:

```
Array length: 10
1 2 3 4 5 0 0 0 0 0
Array length: 10
a b c d e null null null null null
```

43.4 SUMMARY

The above code examples shows a very small set of functions provided by Java reflection. Reading those examples may only give you a taste of Java reflection, you may want to Read more information on Oracle website.

HOW TO DESIGN A JAVA FRAMEWORK? - A SIMPLE EXAMPLE

You may be curious about how framework works? A simple framework example will be made here to demonstrate the idea of frameworks.

44.1 GOAL OF A FRAMEWORK

First of all, why do we need a framework other than just a normal library? The goal of framework is defining a process which let developers implement certain functions based on individual requirements. In other words, framework defines the skeleton and developers fill in the flash when using it.

44.2 THE SIMPLEST FRAMEWORK

In the following example, the first 3 classes are defined as a part of framework and the 4th class is the client code of the framework.

Main.java is the entry point of the framework. This can not be changed.

```java
//imagine this is the entry point for a framework, it can not be
    changed
public class Main {
        public static void main(String[] args) {
                Human h = new Human(new Walk());
                h.doMove();
        }
}
```

Move.java is the Hook. A hook is where developers can define / extend functions based on their own requirements.

```java
public abstract class Move {
        public abstract void action();
}
```

Human.java is the Template, which reflects the idea of how the framework works.

```java
public class Human {
        private Move move;

        public Human(Move m){
                this.move = m;
        }

        public void doMove(){
                this.move.action();
        }
}
```

This simple framework allows and requires developers to extend "Move" class. Actually, in this simple framework, action() method is the only thing developers are able to change.

Inside of the implementation, different "action" can be programmed to different purpose. E.g. the example below print "5 miles per hour", of course, you can redefine it as "50 miles per hour".

```java
public class Walk extends Move {

        @Override
        public void action() {
                // TODO Auto-generated method stub
                System.out.println("5 miles per hour - it is slow!
                    ");
        }
}
```

44.3 CONCLUSION

The example here just shows how a simple Template and Hook works. A real framework is more complicated than this. Not only does it contain other relations like template-temple relation, but also very complex process about how to efficiently improve performance and programming usability.

WHY DO WE NEED JAVA WEB FRAMEWORKS LIKE STRUTS 2?

There are various kinds of Java web frameworks, such as Spring MVC, JavaServer Faces, Struts 2, etc. For a newbie programmer, there is an exponential learning curve.

Why do I need Java web frameworks like Struts 2? This question can be answered by starting from answering how the Servlet API works.

Here is a post which contains code about how to simply program with Servlet API. You would never use this to really program a large project, but it's good to take a look how it looks like originally.

Here is a simple Servlet which process request from client and generate response html.

```java
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class WelcomeServlet extends HttpServlet {

        @Override
        public void init(ServletConfig config) throws
           ServletException {
                super.init(config);
        }
```

```java
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
        //Get the value of form parameter
        String name = request.getParameter("name");
        String welcomeMessage = "Welcome "+name;

        // Set the content type(MIME Type) of the response
           .
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Write the HTML to the response
        out.println("<html>");
        out.println("<head>");
        out.println("<title> A very simple servlet example
            </title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>"+welcomeMessage+"</h1>");
        out.println("<a href="/servletexample/pages/form.
            html">"+"Click here to go back to input page "
            +"</a>");
        out.println("</body>");
        out.println("</html>");
        out.close();

    }


    public void destroy() {

    }
}
```

This is very simple, real usage won't be easy like this. A real servlet has more work to do as summarized below:

- Binding request parameters to Java types. String name = request.getParameter("name");

- Validating data. E.g. There should not be numbers in people's name.

- Making calls to business logic. E.g. Process the name for some purposes.

- Communicate with the data layer. E.g. Store user data.

- Rendering presentation layer (HTML, and so on). E.g. Return results for client browser.

Of course, we can do all of those by ourselves, which is totally possible. However, that would take a lot of time. And very often, those functions are common features which can be implemented in some certain approach. Struts 2 is such an approach. It provides a standard way to implement those common functions following MVC design patterns.

Here is my previous post about a simple Struts2 application.

46

JVM RUN-TIME DATA AREAS

This is my note of reading JVM specification. I draw a diagram which helps me understand.
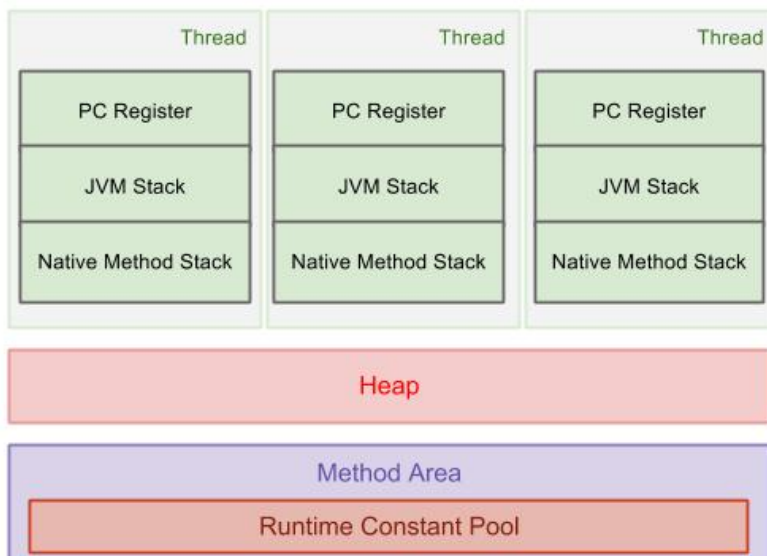
46.1 DATA AREAS FOR EACH INDIVIDUAL THREAD (NOT SHARED)

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: it is used to control each execution of each thread. ãĂĂ
JVM Stack: It contains frames which is demonstrated in the diagram below.

Native Method Stack: it is used to support native methods, i.e., non-Java language
methods. ãĂĂ

46.2   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 meth-
ods and constructors codeãĂĆãĂĂ

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, rang-
ing from numeric literals known at compile-time to method and field references
that must be resolved at run-time.
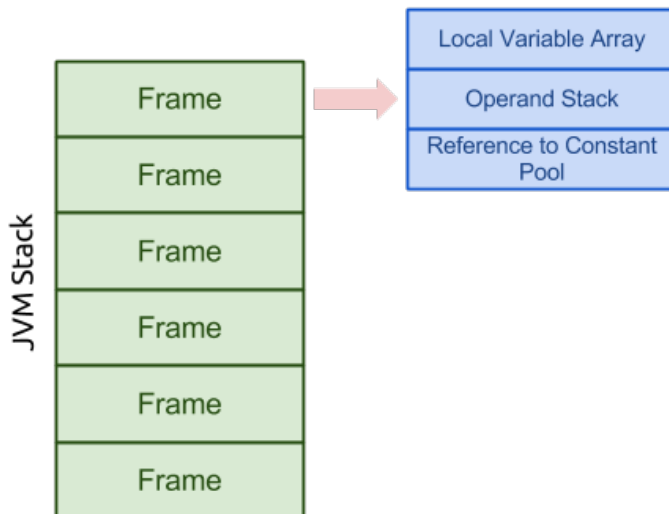
Stack contains Frames, and a frame is pushed to the stack when a method is invoked. A frame contains local variable array, Operand Stack, Reference to Constant Pool.

For more information, please go to the offical JVM specification site.

47

HOW DOES JAVA HANDLE ALIASING?

47.1 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.

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

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

In memory, they both refer to the same location.
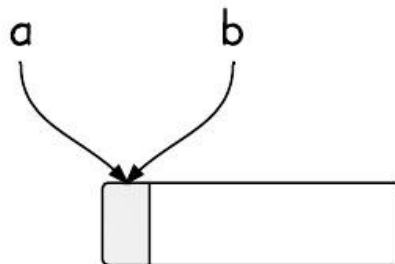
The pointed memory location are pointed by both a and b. During run-time, the actual object stored determines which method to call.

47.2   HOW DOES JAVA HANDLE ALIASING PROBLEM?

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

```java
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:

```java
B[] b = new B[10];
```

```
A[] a = b;

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

and the output is:

```
override method in Child
```

48

## WHAT DOES A JAVA ARRAY LOOK LIKE IN MEMORY?

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

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

48.1 SINGLE-DIMENSION ARRAY

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

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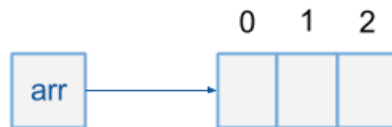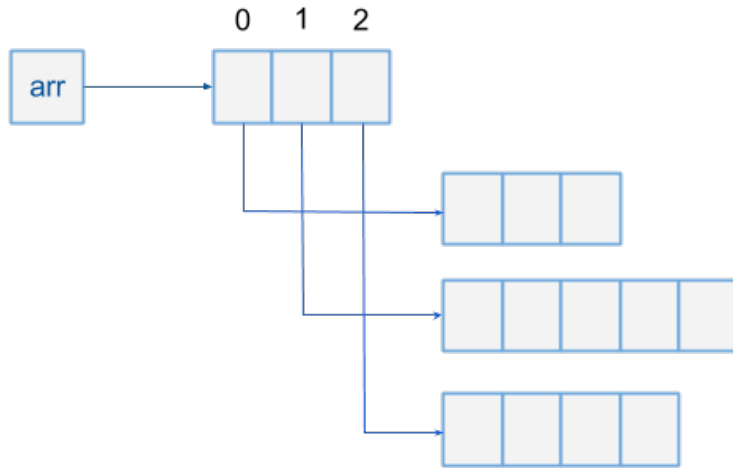

48.2 TWO-DIMENSIONAL ARRAY

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.

48.3   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:
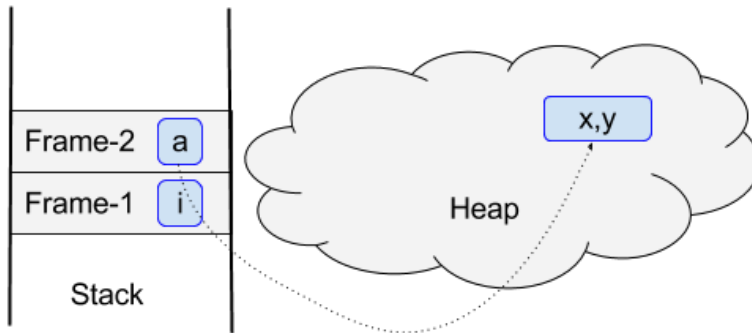
Arrays are treated the same way like objects, so how array locates in memory is straight-forward.

THE INTRODUCTION OF 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.

49.1 WHAT ARE MEMORY LEAKS?

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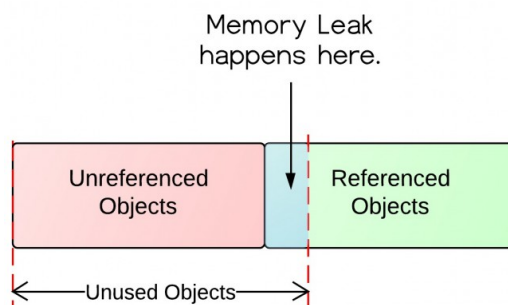
From the diagram, there are referenced objects and unreferenced objects. Unreferenced objects will be garbage collected, while referenced objects will not be garbage collected. Unreferenced objects are surely unused, because no other objects refer to it. However, unused objects are not all unreferenced.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

49.2   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 (t1 - t4) is much longer than B's (t2 - t3). 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.
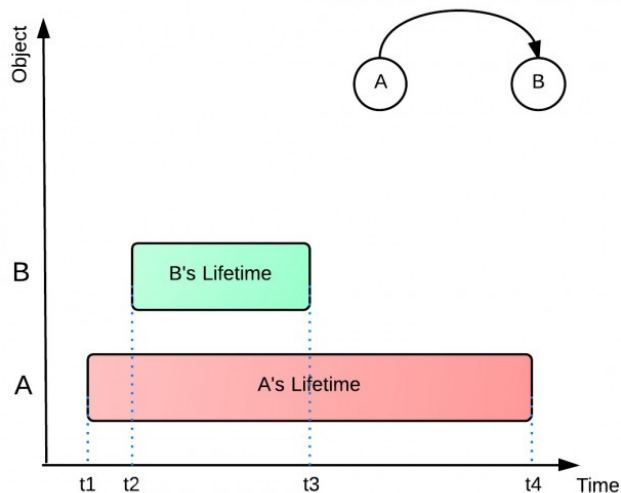
49.3 HOW TO PREVENT MEMORY LEAKS?

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 programer should be alert for memory leaks."[1] Often times member variables of an object that point to other objects need to be null out.

## 49.4 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.

50

WHAT IS SERVLET CONTAINER?

In this post, I write a little bit about the basic ideas of web server, Servlet container and its relation with JVM. I want to show that Servlet container is nothing more than a Java program.

50.1 WHAT IS A WEB SERVER?

To know what is a Servlet container, we need to know what is a Web Server first.
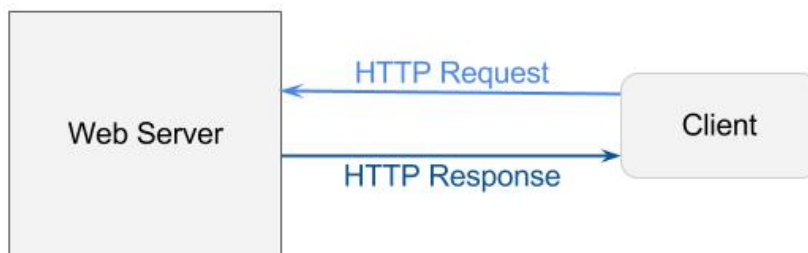
A web server uses HTTP protocol to transfer data. In a simple situation, a user type in a URL (e.g. www.programcreek.com/static.html) in browser (a client), and get a web page to read. So what the server does is sending a web page to the client. The transformation is in HTTP protocol which specifies the format of request and response message.

50.2 WHAT IS A SERVLET CONTAINER?

As we see here, the user/client can only request static webpage from the server. This is not good enough, if the user wants to read the web page based on his input. The basic idea of Servlet container is using Java to dynamically generate the web page on the server side. So servlet container is essentially a part of a web server that interacts with the servlets.
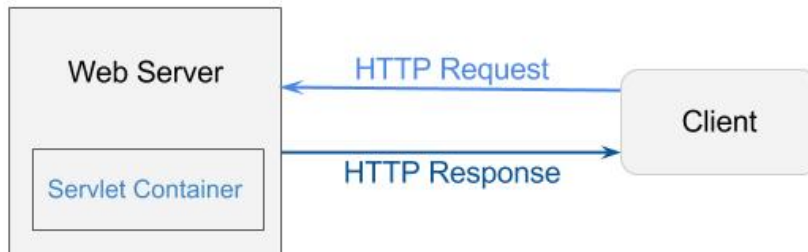
Servlet container is the container for Servlets.

50.3 WHAT IS A SERVLET?

Servlet is an interface defined in javax.servlet package. It declares three essential methods for the life cycle of a servlet - init(), service(), and destroy(). They are implemented by every servlet(defined in SDK or self-defined) and are invoked at specific times by the server.

- The init() method is invoked during initialization stage of the servlet life cycle. It is passed an object implementing the javax.servlet.ServletConfig interface, which allows the servlet to access initialization parameters from the web application.

- The service() method is invoked upon each request after its initialization. Each request is serviced in its own separate thread. The web container calls the service() method of the servlet for every request. The service() method determines the kind of request being made and dispatches it to an appropriate method to handle the request.

- The destroy() method is invoked when the servlet object should be destroyed. It releases the resources being held.

From the life cycle of a servlet object, we can see that servlet classes are loaded to container by class loader dynamically. Each request is in its own thread, and a servlet object can serve multiple threads at the same time(thread not safe). When it is no longer being used, it should be garbage collected by JVM.

Like any Java program, the servlet runs within a JVM. To handle the complexity of HTTP requests, the servlet container comes in. The servlet container is responsible for servlets' creation, execution and destruction.

50.4   how servlet container and web server process a request?

- Web server receives HTTP request

- Web server forwards the request to servlet container

- The servlet is dynamically retrieved and loaded into the address space of the container, if it is not in the container.

- The container invokes the init() method of the servlet for initialization(invoked once when the servlet is loaded first time)

- The container invokes the service() method of the servlet to process the HTTP request, i.e., read data in the request and formulate a response. The servlet remains in the container's address space and can process other HTTP requests.

- Web server return the dynamically generated results to the correct location

The six steps are marked on the following diagram:
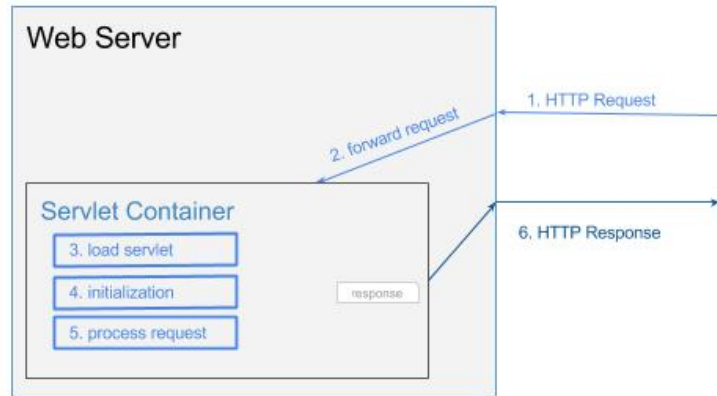
50.5 THE ROLE OF JVM

Using servlets allows the JVM to handle each request within a separate Java thread, and this is one of the key advantage of Servlet container. Each servlet is a Java class with special elements responding to HTTP requests. The main function of Servlet contain is to forward requests to correct servlet for processing, and return the dynamically generated results to the correct location after the JVM has processed them. In most cases servlet container runs in a single JVM, but there are solutions when container need multiple JVMs.

WHAT IS ASPECT-ORIENTED PROGRAMMING?

What is Aspect-Oriented Programming(AOP)? By using the diagram below, the concept can be understood in a few seconds.

51.1 THE CROSS-CUTTING CONCERNS PROBLEM

First take a took at the diagram below, and think about what could be the problem.
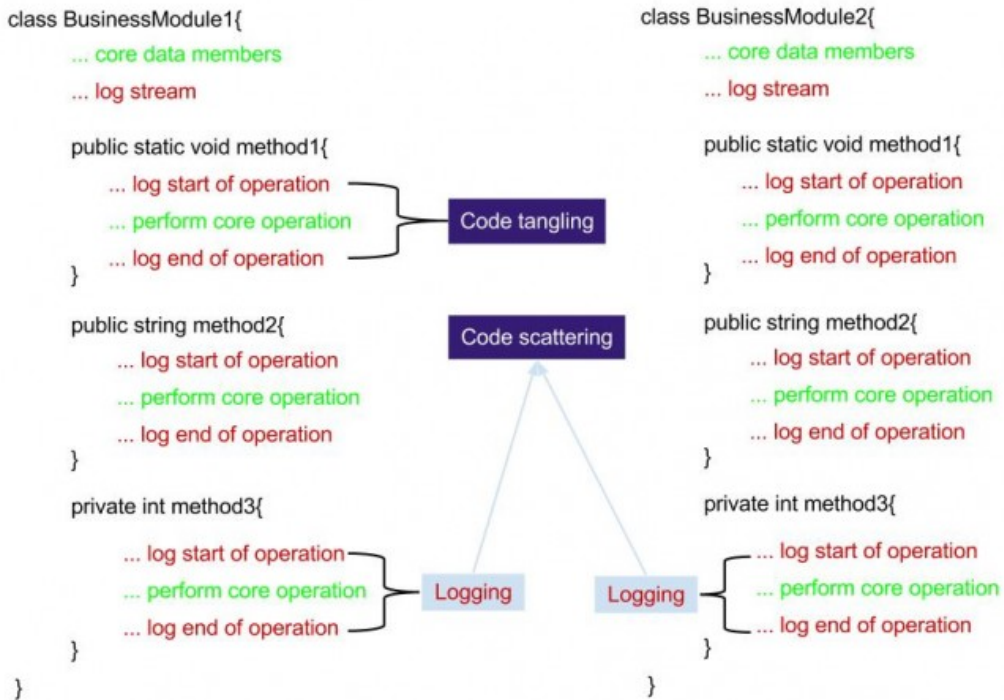
1. Code tangling: the logging code is mixed with business logic. 2. Code scattering: caused by identical code put in every module.

The logging function is a called "cross-cutting concern". That is, a function that is used in many other modules, such as authentication, logging, performance, error checking, data persistence, storage management, to name just a few.

By using Object-Oriented Programming (OOP), we can define low coupling and high cohesion system. However, when it comes to cross-cutting concerns, it does not handle it well for the reason that it does not relation between handle core concerns and cross-cutting concerns.

51.2   SOLUTION FROM AOP


```
class BusinessModule1{
       ... core data members
       public static void method1{
              ... perform core operation
       }
       public string method2{
              ... perform core operation
       }
       private int method3{
              ... perform core operation
       }
}
```

Log Module

Weaving rules go into an aspect:
Rule 1: Create a logger object
Rule 2: Log the beginning of each
private operation

Compiler

```
class BusinessModule1{
       ... core data members
       ... log stream
       private int method3{
              ... log start of operation
              ... perform core operation
              ... log end of operation
       }
}
```

LIBRARY VS. FRAMEWORK?

What is the difference between a Java Library and a framework? The two concepts are important but sometimes confusing for Java developers.

52.1 KEY DIFFERENCE AND DEFINITION OF LIBRARY AND FRAMEWORK

The key difference between a library and a framework is "Inversion of Control". When you call a method from a library, you are in control. But with a framework, the control is inverted: the framework calls you.
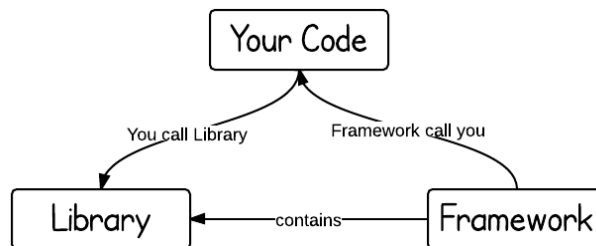
A library is just a collection of class definitions. The reason behind is simply code reuse, i.e. get the code that has already been written by other developers. The classes and methods normally define specific operations in a domain specific area. For example, there are some libraries of mathematics which can let developer just call the function without redo the implementation of how an algorithm works.

In framework, all the control flow is already there, and there's a bunch of pre-defined white spots that you should fill out with your code. A framework is normally more complex. It defines a skeleton where the application defines its own features to fill out the skeleton. In this way, your code will be called by the framework when appropriately. The benefit is that developers do not need to worry about if a design is good or not, but just about implementing domain specific functions.

52.2 THEIR RELATION

Both of them defined API, which is used for programmers to use. To put those together, we can think of a library as a certain function of an application, a framework as the skeleton of the application, and an API is connector to put those together. A typical development process normally starts with a framework, and fill out functions defined in libraries through API.
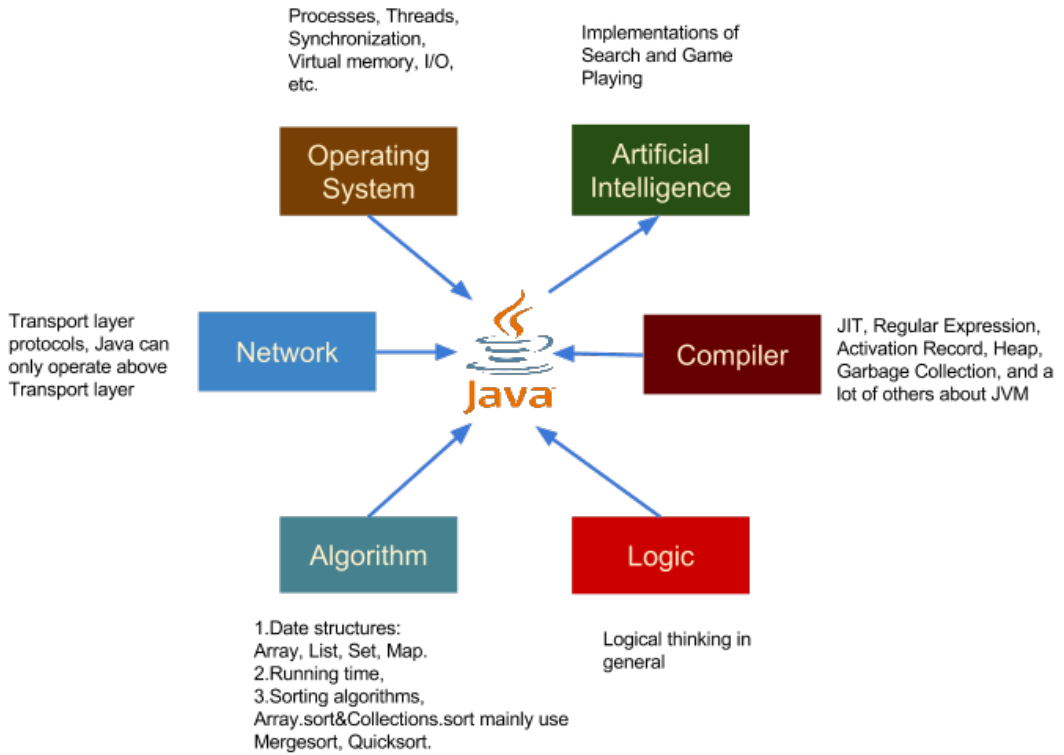
52.3 EXAMPLES

1. How to make a Java library?? 2. How to design a framework??

JAVA AND COMPUTER SCIENCE COURSES

A good programmer does not only know how to program a task, but also knows why it is done that way and how to do it efficiently. Indeed, we can find almost any code by using Google, knowing why it is done that way is much more difficult than knowing how to do it, especially when something goes wrong.

To understand Java design principles behind, Computer Science(CS) courses are helpful. Here is the diagram showing the relation between Java and Operating System, Networks, Artificial Intelligence, Compiler, Algorithm, and Logic.

Processes, Threads,
Synchronization,
Virtual memory, I/O,
etc.

Implementations of
Search and Game
Playing

Operating
System

Artificial
Intelligence

Transport layer
protocols, Java can
only operate above
Transport layer

Network

Java

Compiler

JIT, Regular Expression,
Activation Record, Heap,
Garbage Collection, and a
lot of others about JVM

Algorithm

Logic

1.Date structures:
Array, List, Set, Map.
2.Running time,
3.Sorting algorithms,
Array.sort&Collections.sort mainly use
Mergesort, Quicksort.

Logical thinking in
general

54

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.

```java
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.

55

TOP 10 METHODS FOR JAVA ARRAYS

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

55.1 DECLARE AN ARRAY

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

55.2 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]
```

55.3 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]
```

55.4 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
```

55.5 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);
```

55.6 DECLARE AN ARRAY INLINE

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

55.7 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
```

55.8 COVNERT 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);
```

55.9   CONVERT AN ARRAY TO A SET

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

55.10   REVERSE AN ARRAY

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

55.11   . REMOVE ELEMENT OF AN ARRAY

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

55.12   ONE MORE - CONVERT INT TO BYTE ARRAY

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

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

56

TOP 10 QUESTIONS OF JAVA STRINGS

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

56.1 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.

56.2 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.

56.3 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;
}
```

56.4 HOW TO CONVERT STRING TO INT?

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

Simple, but so frequently used and sometimes ignored.

56.5 HOW TO SPLIT A STRING WITH WHITE SPACE CHARACTERS?

We can simple do split using regular expression. "" stands for white space characters such as " ", "", "", "".

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

56.6 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.

56.7   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.

56.8   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
```

56.9   HOW TO CONVERT STRING TO DATE?

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

56.10   . 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);
```

56.11   ONE MORE DO YOU KNOW HOW TO DETECT IF A STRING CONTAINS ONLY UPPERCASE LETTER?

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, 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 . 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.

```java
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 a 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 an Finite state machine (see Compilers: Principles, Techniques, and Tools (2nd Edition)). But all of the states involved in performing a match resides 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
$5 or [$]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 literature, 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 show to be context-free grammar S → aSb | 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 añ bñ 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 matches the very inner parenthesed 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 nth iteration, the expression stops at the nth 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␣bbbbb␣␣␣ccc␣␣␣␣␣d␣␣";
// " aa bbbbb 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) {
```

```java
    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;
}
```

57.1 . HOW TO USE BACKREFERENCES IN JAVA REGULAR EXPRESSIONS

Backreferences is another useful feature in Java regular expression.

58

TOP 10 QUESTIONS ABOUT JAVA EXCEPTIONS

This article summarizes the top 10 frequently asked questions and answers about Java exceptions. For example, what's the best practice for exception management?

58.1 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 Runtime-Exception is an unchecked exception. You can check out the exception hierarchy diagram before reading the rest.

58.2 BEST PRACTICE FOR EXCEPTION MANAGEMENT

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

58.3 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.

```java
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.

58.4 WHY DO DOUBLE.PARSEDOUBLE(NULL) AND INTEGER.PARSEINT(NULL) THROW DIFFERENT EXCEPTIONS?

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

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

Double.parseDouble(null);
// throws java.lang.NullPointerException
```

58.5 COMMONLY USED RUNTIME EXCEPTIONS IN JAVA

Here are just some of them. IllegalArgumentException ArrayIndexOutOfBoundsException

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");
```

58.6 CAN WE CATCH MULTIPLE EXCEPTIONS IN THE SAME CATCH CLAUSE?

The answer is YES. As long as those exceptions can trace back to the same node in the hierarchy, you can use that one only.

58.7 CAN CONSTRUCTOR THROW EXCEPTIONS IN JAVA?

The answer is YES. Constructor is a special kind of method. Here is a code example.

58.8 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.

```java
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();
        }
}
```

58.9   can return be used in finally block

Yes, it can.

58.10   . 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?

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

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

59

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.

59.1 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 quicklier, because it only changes the list locally.

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

```
|  Arraylist  |  LinkedList
```

	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.

59.2 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,

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

One most frequent incorrect code is

```java
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."

59.3 HOW TO CONVERT LIST TO INT[]?

The easiest way might be using ArrayUtils in Apache Commons Lang library.

```java
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);
}
```

59.4 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);
}
```

59.5 WHAT IS THE BEST WAY TO FILTER A COLLECTION?

Again, you can use third-party package, like Guava or Apache Commons Lang to
fullfil 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.

```java
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:

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

59.6 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.

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

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

59.7 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, there is no short-cut way. Two loops are needed at least.

59.8 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.

59.9 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.

59.10 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.

60

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.

60.1 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());
```

60.2 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();
}
```

60.3   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);
```

60.4 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.

60.5 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");.

```java
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 intilizaing 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.

60.6 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. 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
null key-value   | yes-yes  | yes-yes | no-yes
synchronized     | 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.

60.7   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 unfortunetely is not supported by JDK.

60.8   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. âĂę It's a shame that Cloneable is broken, but it happens. âĂę Cloneable is a weak spot, and I think people should be aware of its limitations.

60.9 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