Java

Introduction

History of Java

- Java was originally developed by Sun Microsystems starting in 1991
  - James Gosling
  - Patrick Naughton
  - Chris Warth
  - Ed Frank
  - Mike Sheridan
- This language was initially called ***Oak***
- Renamed ***Java*** in 1995

What is Java

- A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language -- **Sun Microsystems**
- **Object-Oriented**
  - No free functions
  - All code belong to some class
  - Classes are in turn arranged in a hierarchy or package structure

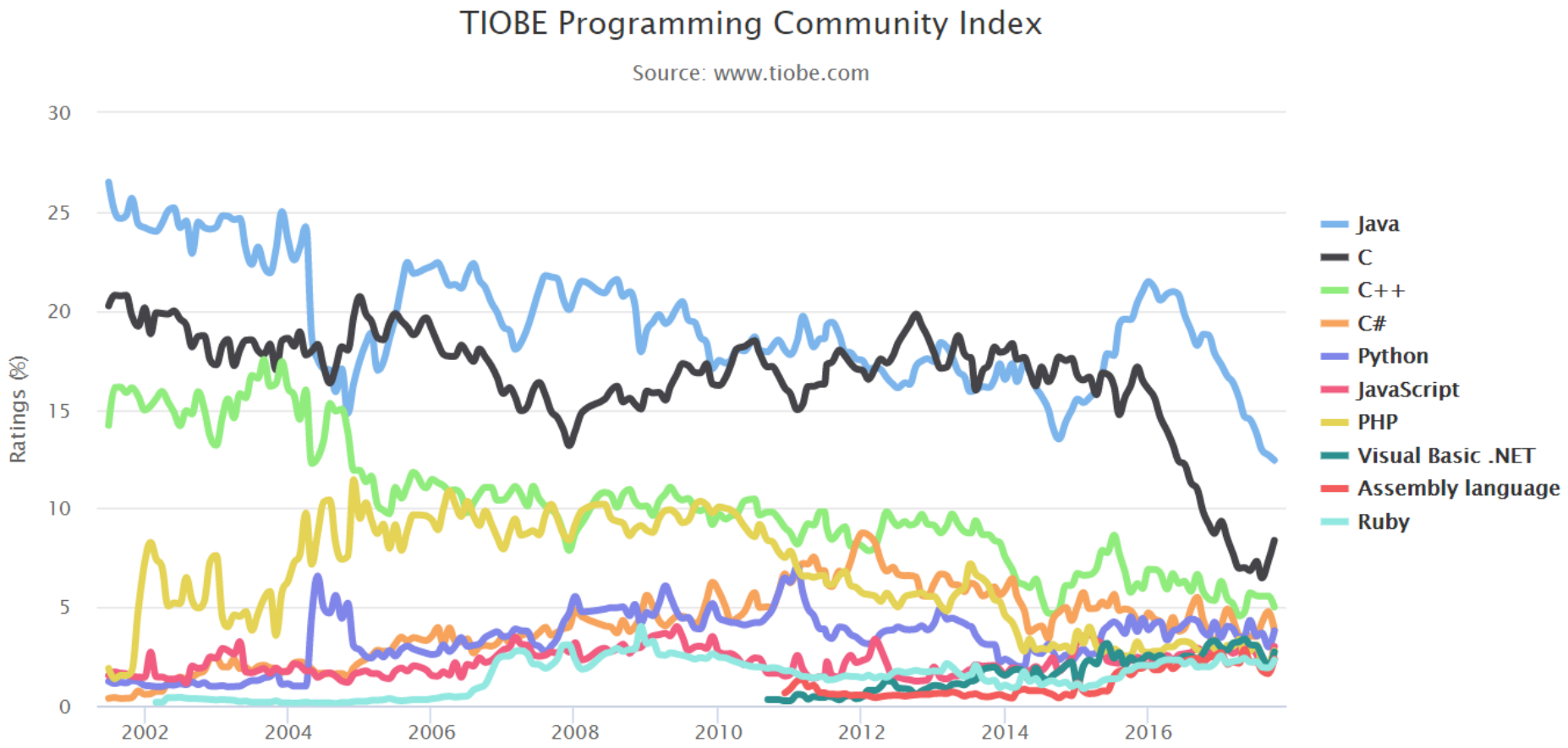
What is Java

- **Distributed**
  - Fully supports IPv4, with structures to support IPv6
  - Includes support for Applets: small programs embedded in HTML documents
- **Interpreted**
  - The program are compiled into Java Virtual Machine (JVM) code called bytecode
  - Each bytecode instruction is translated into machine code at the time of execution

What is Java

- **Robust**
  - Java is simple – no pointers/stack concerns
  - Exception handling – try/catch/finally series allows for simplified error recovery
  - Strongly typed language – many errors caught during compilation

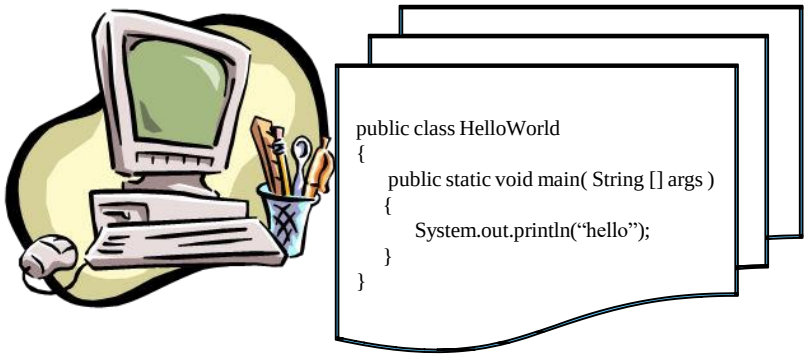
Java – The Most Popular



Java Editions

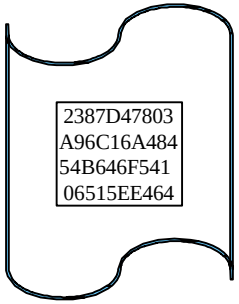
- Java 2 Platform, Standard Edition (J2SE)
  - Used for developing Desktop based application and networking applications
- Java 2 Platform, Enterprise Edition (J2EE)
  - Used for developing large-scale, distributed networking applications and Web-based applications
- Java 2 Platform, Micro Edition (J2ME)
  - Used for developing applications for small memory-constrained devices, such as cell phones, pagers and PDAs

Java platform



HelloWorld.java

Compile
javac

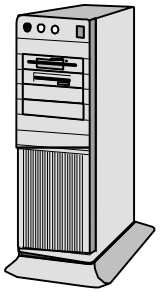


HelloWorld.class

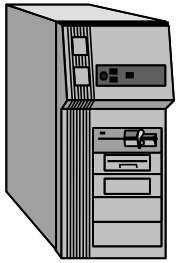
Java
Bytecode

Java
Interpreter

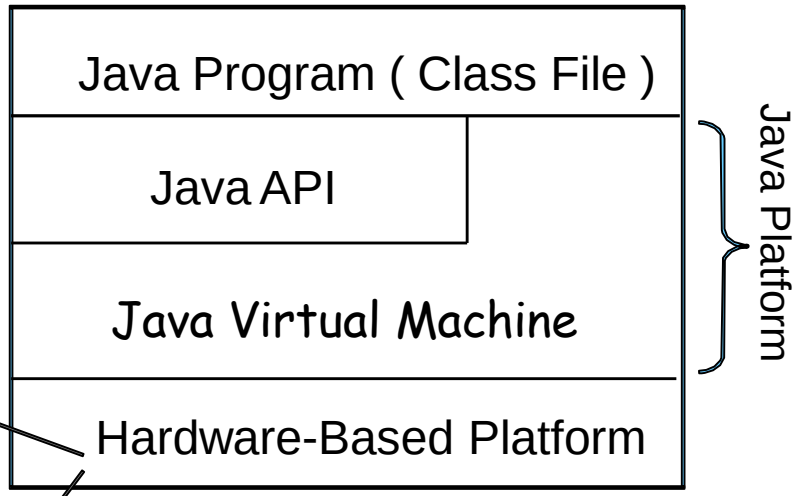
Windows NT



Java
Interpreter



Power Macintosh



Java Development Environment

- Edit
  - Create/edit the source code
- Compile
  - Compile the source code
- Load
  - Load the compiled code
- Verify
  - Check against security restrictions
- Execute
  - Execute the compiled

Phase 1: Creating a Program

- Any text editor or Java IDE (Integrated Development Environment) can be used to develop Java programs
- Java source-code file names must end with the ***.java*** extension
- Some popular Java IDEs are
  - NetBeans
  - Eclipse
  - JCreator
  - **IntelliJ**

Phase 2: Compiling a Java Program

- ***javac Welcome.java***
  - Searches the file in the current directory
  - Compiles the source file
  - Transforms the Java source code into bytecodes
  - Places the bytecodes in a file named **Welcome.class**

Bytecodes *

- They are not machine language binary code
- They are independent of any particular microprocessor or hardware platform
- They are platform-independent instructions
- Another entity (interpreter) is required to convert the bytecodes into machine codes that the underlying microprocessor understands
- This is the job of the **JVM** (Java Virtual Machine)

JVM (Java Virtual Machine) *

- It is a part of the JDK and the foundation of the Java platform
- It can be installed separately or with JDK
- A virtual machine (VM) is a software application that simulates a computer, but hides the underlying operating system and hardware from the programs that interact with the VM
- It is the JVM that makes Java a portable language

JVM (Java Virtual Machine) *

- The same bytecodes can be executed on any platform containing a compatible JVM
- The JVM is invoked by the java command
  - ***java Welcome***
- It searches the class Welcome in the current directory and executes the main method of class Welcome
- It issues an error if it cannot find the class Welcome or if class Welcome does not contain a method called main with proper signature

Phase 3: Loading a Program *

- One of the components of the JVM is the class loader
- The class loader takes the .class files containing the programs bytecodes and transfers them to RAM
- The class loader also loads any of the .class files provided by Java that our program uses

Phase 4: Bytecode Verification *

- Another component of the JVM is the bytecode verifier
- Its job is to ensure that bytecodes are valid and do not violate Java's security restrictions
- This feature helps to prevent Java programs arriving over the network from damaging our system

Phase 5: Execution

- Now the actual execution of the program begins
- Bytecodes are converted to machine language suitable for the underlying OS and hardware
- Java programs actually go through two compilation phases
  - Source code -> Bytecodes
  - Bytecodes -> Machine language

Editing a Java Program

Welcome.java

```
1
2   public class Welcome {
3       public static void main(String[] args) {
4           System.out.println("Hello Java");
5           System.out.printf("I like %s\n", "Java");
6           String strDepartment = "CSE";
7           System.out.print("We study in " + strDepartment + "\n");
8       } // end method main
9   } // end class Welcome - NOTE: no semicolon is required here
10
```

Examining Welcome.java

- A Java source file can contain multiple classes, but only one class can be a public class
- Typically Java classes are grouped into packages (similar to namespaces in C++)
- A public class is accessible across packages
- The source file name must match the name of the public class defined in the file with the .java extension

Examining Welcome.java

- In Java, there is no provision to declare a class, and then define the member functions outside the class
- Body of every member function of a class (called method in Java) must be written when the method is declared
- Java methods can be written in any order in the source file
- A method defined earlier in the source file can call a method defined later

Examining Welcome.java

- ***public static void main(String[] args)***
  - **main** is the starting point of every Java application
  - **public** is used to make the method accessible by all
  - **static** is used to make main a static method of class Welcome. Static methods can be called without using any object; just using the class name. JVM call main using the **ClassName.methodName** (*Welcome.main*) notation
  - **void** means main does not return anything
  - **String args[ ]** represents an array of String objects that holds the command line arguments passed to the application. *Where is the length of args array?*

Examining Welcome.java

- Think of JVM as a outside Java entity who tries to access the main method of class Welcome
  - main must be declared as a public member of class Welcome
- JVM wants to access main without creating an object of class Welcome
  - main must be declared as static
- JVM wants to pass an array of String objects containing the command line arguments
  - main must take an array of String as parameter

Examining Welcome.java

- ***System.out.println()***
  - Used to print a line of text followed by a new line
  - **System** is a class inside the Java API
  - **out** is a public static member of class System
  - **out** is an object of another class of the Java API
  - **out** represents the standard output (similar to stdout or cout)
  - **println** is a public method of the class of which out is an object

Examining Welcome.java

- **System.out.print()** is similar to **System.out.println()**, but does not print a new line automatically
- **System.out.printf()** is used to print formatted output like printf() in C
- In Java, characters enclosed by double quotes ("") represents a String object, where String is a class of the Java API
- We can use the plus operator (+) to concatenate multiple String objects and create a new String object

Compiling a Java Program

- Place the .java file in the bin directory of your Java installation
  - ***C:\Program Files\Java\jdk1.8.0_144\bin***
- Open a command prompt window and go to the bin directory
- Execute the following command
  - ***javac Welcome.java***
- If the source code is ok, then javac (the Java compiler) will produce a file called Welcome.class in the current directory

Compiling a Java Program

- If the source file contains multiple classes then javac will produce separate .class files for each class
- Every compiled class in Java will have their own .class file
- .class files contain the bytecodes of each class
- So, a .class file in Java contains the bytecodes of a single class only

Executing a Java Program

- After successful compilation execute the following command
  - ***java Welcome***
  - *Note that we have omitted the .class extension here*
- The JVM will look for the class file *Welcome.class* and search for a *public static void main(String args[ ])* method inside the class
- If the JVM finds the above two, it will execute the body of the main method, otherwise it will generate an error and will exit immediately

Another Java Program

A.java

```
1  public class A {
2      private int a;
3
4      public A()
5      {
6          this.a = 0;
7      }
8
9      public void setA(int a)
10     {
11         this.a = a;
12     }
13
14     public int getA()
15     {
16         return this.a;
17     }
18
19     public static void main(String args[])
20     {
21         A ob;
22         ob=new A();
23         ob.setA(10);
24         System.out.println(ob.getA());
25     }
26 }
27
```

Examining A.java

- The variable of a class type is called a reference
  - *ob* is a reference to A object
- Declaring a class reference is not enough, we have to use new to create an object
- Every Java object has to be instantiated using keyword **new**
- We access a public member of a class using the dot operator (.)
  - Dot (.) is the only member access operator in Java.
  - Java does not have **::, ->, &** and ***

Primitive (built-in) Data types

- Integers
  - **byte** **8-bit integer (new)**
  - short 16-bit integer
  - int 32-bit signed integer
  - long 64-bit signed integer
- Real Numbers
  - float 32-bit floating-point number
  - double 64-bit floating-point number
- Other types
  - **char** **16-bit, Unicode 2.1 character**
  - boolean true or false, *false is not 0 in Java*

Boolean Type

Boolean.java

```
1  public class Boolean {
2      public static void main(String[] args) {
3          int a = 10;
4          if (a > 0) // if (a) will give compilation error
5          {
6              System.out.println("Inside If");
7          }
8          boolean b = false;
9          if (b)
10         {
11             System.out.println("Inside If");
12         }
13         else
14         {
15             System.out.println("Inside Else");
16         }
17     }
18 }
19
```

Non-primitive Data types

- The non-primitive data types in java are
  - Objects
  - Array
- Non-primitive types are also called reference types

```
public class Box {
    int L, W, H;

    Box(int l, int w, int h)
    {
        L = l;
        W = w;
        H = h;
    }

    public static void main(String[] args)
    {
        Box p; // p is a reference pointing to null
        p = new Box( l: 1, w: 2,  h: 3); // now the actual object is created
    }
}
```

Primitive vs. Non-primitive type

- Primitive types are handled by value – the actual primitive values are stored in variable and passed to methods

  int x = 10;

  public MyPrimitive(int x) { }

- Non-primitive data types (objects and arrays) are handled by reference – the reference is stored in variable and passed to methods

  Box b = new Box(1,2,3);

  public MyNonPrimitive(Box x) { }

Primitive vs. Non-primitive type

- Primitive types are handled by value
  - There is no easy way to swap two primitive integers in Java
  - No method like **void swap(int *x, int *y)**
  - Can only be done using object or array
- But do we actually need a method to swap?
  - **x += (y - (y = x))** does the same in a single statement

Java References

- Java references are used to point to Java objects created by new
- Java objects are **always** passed **by reference** to other functions, ***never by value***
- Java references act as pointers but does not allow pointer arithmetic
- We cannot read the value of a reference and hence cannot find the address of a Java object
- We cannot take the address of a Java reference

Java References

- We can make a Java reference point to a new object
  - By copying one reference to another

    ClassName ref2 = ref1; // Here ref1 is declared earlier
  - By creating a new object and assign it to the reference

    ClassName ref1 = new ClassName();
- We cannot place arbitrary values to a reference except the special value **null** which means that the reference is pointing to nothing

  ClassName ref1 = 100; // compiler error

  ClassName ref2 = null; // no problem

Java References

Box.java

```java
public class Box {
    int L, W, H;

    Box(int l, int w, int h)
    {
        L = l;
        W = w;
        H = h;
    }

    public static void main(String[] args)
    {
        Box b1; // b1 refers to null
        Box b2; // b2 refers to null
        b1 = new Box( l: 8,  w: 5,  h: 7); // b1 refers to new object (8, 5, 7)
        b2 = b1; // b2 refers to b1, so both refers (8, 5, 7)
        b1 = new Box( l: 3,  w: 9,  h: 2); // b1 refers to new object (3, 9, 2)
        b1 = b2; // b1 refers to b2, what happens to object (3, 9, 2)
    }
}
```

Textbook

- We will mostly follow Java 8, if time permits will see the new features of Java 9
- Books
  - Java: The Complete Reference, 9th Edition by Herbert Schildt
  - Effective Java, 2nd edition by Joshua Bloch (for future)
- Web
  - http://rifatshahriyar.github.io/CSE107.html

Java

More Details

Array

Arrays

- A group of variables containing values that all have the same type
- Arrays are fixed-length entities
- In Java, arrays are objects, so they are considered reference types
- But the elements of an array can be either primitive types or reference types

Arrays

- We access the element of an array using the following syntax
  - name[index]
  - “index” must be a nonnegative integer
    - “index” can be int/byte/short/char but not long
- In Java, every array knows its own length
- The length information is maintained in a public final int member variable called length

Declaring and Creating Arrays

- int c[ ] = new int [12]
  - Here, “c” is a reference to an integer array
  - “c” is now pointing to an array object holding 12 integers
  - Like other objects arrays are created using “new” and are created in the heap
  - “int c[ ]” represents both the data type and the variable name. Placing number here is a syntax error
  - **int c[12]; // compiler error**

Declaring and Creating Arrays

- int[ ] c = new int [12]
  - Here, the data type is more evident i.e. “int[ ]”
  - But does the same work as
    - int c[ ] = new int [12]
- Is there any difference between the above two approaches?

Declaring and Creating Arrays

- int c[ ], x
  - Here, ‘c’ is a reference to an integer array
  - ‘x’ is just a normal integer variable
- int[ ] c, x;
  - Here, ‘c’ is a reference to an integer array (same as before)
  - But, now ‘x’ is also a reference to an integer array

Arrays

ArrayDemo.java

```java
public class ArrayDemo {
    public static void main(String[] args) {
        int [] a = new int[10];
        for (int i = 0; i < a.length; i++) {
            a[i] = i;
        }
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}

```

Using an Array Initializer

- We can also use an array initializer to create an array
  - int n[ ] = {10, 20, 30, 40, 50}
- The length of the above array is 5
- n[0] is initialized to 10, n[1] is initialized to 20, and so on
- The compiler automatically performs a “new” operation taking the count information from the list and initializes the elements properly

Arrays of Primitive Types

- When created by “new”, all the elements are initialized with default values
  - byte, short, char, int, long, float and double are initialized to zero
  - boolean is initialized to false
- This happens for both member arrays and local arrays

Arrays of Reference Types

- String [] str = new String[3]
  - Only 3 String references are created
  - Those references are initialized to “null” by default
  - Need to explicitly create and assign actual String objects in the above three positions.
    - str[0] = new String(“Hello”);
    - str[1] = “World”;
    - str[2] = “I” + “ Like” + “ Java”;

Passing Arrays to Methods

void modifyArray(double d[ ]) {…}

double [] temperature = new double[24];

modifyArray(temperature);

- Changes made to the elements of ‘d’ inside “modifyArray” is visible and reflected in the “temperature” array
- But inside “modifyArray” if we create a new array and assign it to ‘d’ then ‘d’ will point to the newly created array and changing its elements will have no effect on “temperature”

Passing Arrays to Methods

- Changing the elements is visible, but changing the array reference itself is not visible

```
void modifyArray(double d[ ]) {
    d[0] = 1.1; // visible to the caller
}
void modifyArray(double d[ ]) {
    d = new double [10];
    d[0] = 1.1; // not visible to the caller
}
```

Multidimensional Arrays

- Can be termed as array of arrays.
- int b[ ][ ] = new int[3][4];
  - Length of first dimension = 3
    - b.length equals 3
  - Length of second dimension = 4
    - b[0].length equals 4
- int[ ][ ] b = new int[3][4];
  - Here, the data type is more evident i.e. “int[ ][ ]”

Multidimensional Arrays

- int b[ ][ ] = { { 1, 2, 3 }, { 4, 5, 6 } };
  - b.length equals 2
  - b[0].length and b[1].length equals 3
- All these examples represent rectangular two dimensional arrays where every row has same number of columns
- Java also supports jagged array where rows can have different number of columns

Multidimensional Arrays

Example – 1

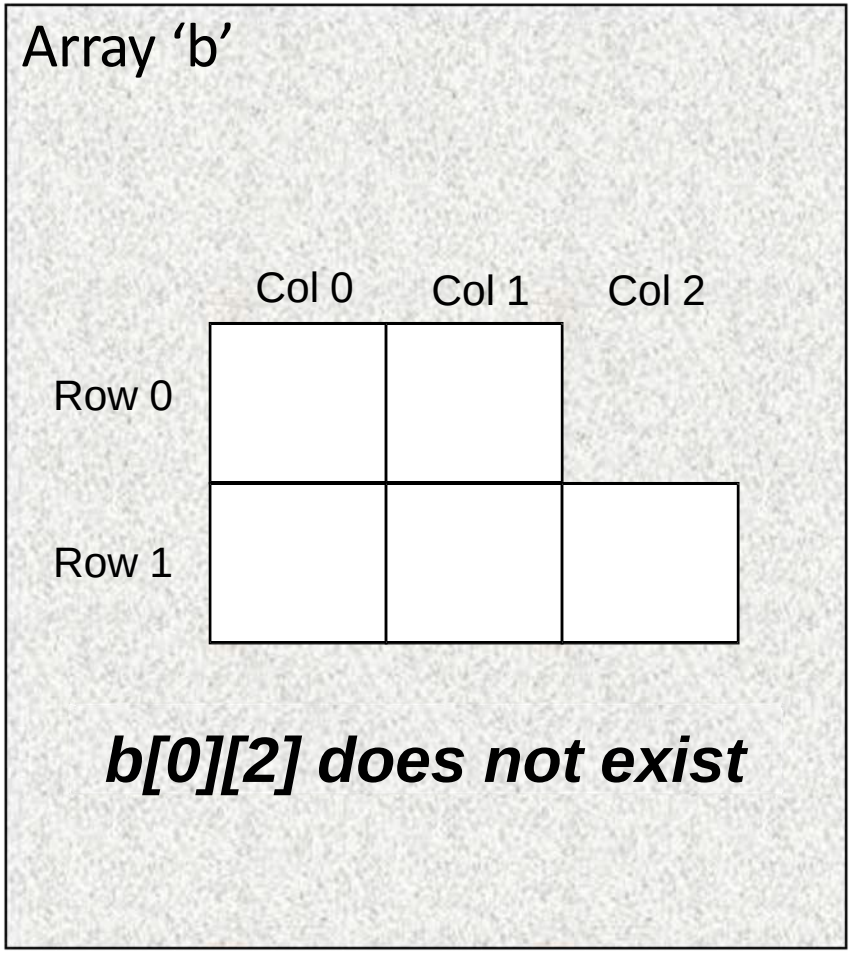
```
int b[ ][ ];
b = new int[2][ ];
b[0] = new int[2];
b[1] = new int[3];
b[0][2] = 7; //will throw an exception
```

Example – 2

```
int b[ ][ ] = { { 1, 2 }, { 3, 4, 5 } };
b[0][2] = 8; //will throw an exception
```

In both cases

b.length equals 2
b[0].length equals 2
b[1].length equals 3



Command Line Arguments

Using Command-Line Arguments

- java MyClass arg1 arg2 … argN
  - words after the class name are treated as command-line arguments by Java
  - Java creates a separate String object containing each command-line argument, places them in a String array and supplies that array to main
  - That's why we have to have a String array parameter (String args[ ]) in main
  - We do not need a "argc" type parameter (for parameter counting) as we can easily use "args.length" to determine the number of parameters supplied.

Using Command-Line Arguments

CommandLineTest.java

```
1  public class CommandLineTest {
2      public static void main(String[] args) {
3          System.out.println( args.length );
4
5          for( int i = 0; i < args.length; i++)
6          {
7              System.out.println( args[i] );
8          }
9      }
10 }
11
```

java CommandLineTest Hello 2 You

3
Hello
2
You

For-Each

For-Each version of the for loop

ForEachTest.java

```
1  public class ForEachTest {
2      public static void main(String[] args) {
3          int numbers [] = {1,2,3,4,5};
4          for(int x : numbers)
5          {
6              System.out.print(x + " ");
7              x = x * 10; // no effect on numbers
8          }
9          System.out.println();
10
11         int numbers2 [][] = { {1,2,3}, {4,5,6}, {7,8,9} };
12         for(int []x:numbers2)
13         {
14             for(int y:x)
15             {
16                 System.out.print(y + " ");
17             }
18             System.out.println("");
19         }
20     }
21 }
```

Scanner

Scanner

- It is one of the utility class located in the java.util package
- Using Scanner class, we can take inputs from the keyboard
- Provides methods for scanning
  - Int
  - float
  - Double
  - Line etc.

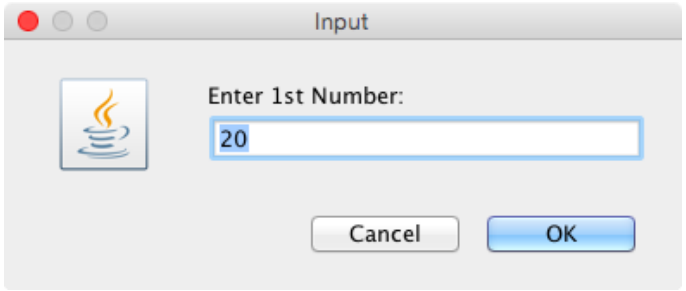
Scanner

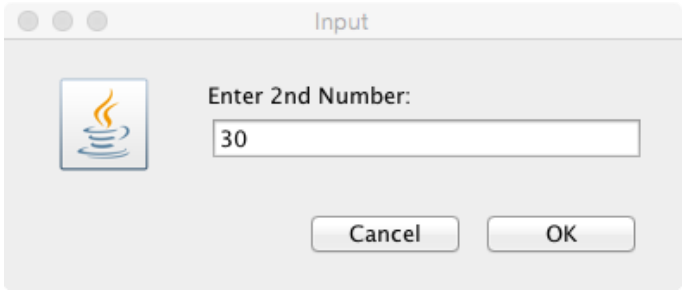
```
3  import java.util.Scanner;
4
5  public class ScannerTest {
6      public static void main(String[] args) {
7          Scanner scn=new Scanner(System.in);
8          while(scn.hasNextLine())
9          {
10             System.out.println(scn.nextLine());
11         }
12     }
13 }
```

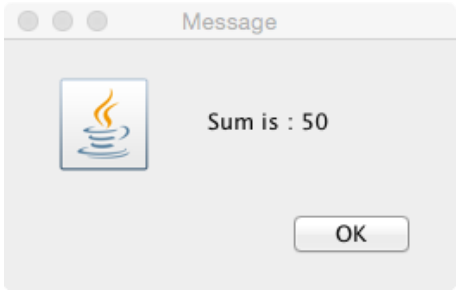
```
3  import java.util.Scanner;
4
5  public class ScannerTest {
6      public static void main(String[] args) {
7          Scanner scn=new Scanner(System.in);
8          while(scn.hasNextInt())
9          {
10             System.out.println(scn.nextInt());
11         }
12     }
13 }
```

JOptionPane

```
3   import javax.swing.JOptionPane;
4
5   public class JOptionPaneTest {
6       public static void main(String[] args) {
7           String s1 = JOptionPane.showInputDialog(null,"Enter 1st Number:");
8           String s2 = JOptionPane.showInputDialog(null,"Enter 2nd Number:");
9           int num1 = Integer.parseInt(s1);
10          int num2 = Integer.parseInt(s2);
11          JOptionPane.showMessageDialog(null,"Sum is : " + (num1+num2));
12      }
13  }
```







Static

Static Variables

- When a member (both methods and variables) is declared static, it can be accessed before any objects of its class are created, and without reference to any object
- Static variable
  - Instance variables declared as static are like global variables
  - When objects of its class are declared, no copy of a static variable is made

Static Methods & Blocks

- Static method
  - They can only call other static methods
  - They must only access static data
  - They cannot refer to ***this*** or ***super*** in any way
- Static block
  - Initialize static variables.
  - Get executed exactly once, when the class is first loaded

Static

```
3   public class StaticTest {
4       static int a = 3, b;
5       int c;
6
7       static void f1(int x) {
8           System.out.println("x = " + x);
9           System.out.println("a = " + a);
10          System.out.println("b = " + b);
11          // System.out.println("c = " + c); // Error
12      }
13      int f2() {
14          return a*b;
15      }
16      static {
17          b = a*4;
18          // c = b; // Error
19      }
20      public static void main(String[] args) {
21          f1(42); // StaticTest.f1(84);
22          System.out.println("b = " + b);
23          //System.out.println("Area = " + f2());   // Error
24      }
25  }
```

Final

- Declare a final variable, prevents its contents from being modified
- final variable must initialize when it is declared
- It is common coding convention to choose all uppercase identifiers for final variables

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Unsigned right shift operator

- The >> operator automatically fills the high-order bit with its previous contents each time a shift occurs
- This preserves the sign of the value
- But if you want to shift something that doesn't represent a numeric value, you may not want the sign extension
- Java's >>> shifts zeros into the high-order bit

int a= -1; a = a >>>24;

11111111 11111111 11111111 11111111 [-1]
00000000 00000000 00000000 11111111 [255]

Nested and Inner Classes

Nested Classes

- It is possible to define a class within another classes, such classes are known as nested classes
- The scope of nested class is bounded by the scope of its enclosing class. That means if class B is defined within class A, then B doesn’t exists without A
- The nested class has access to the members (including private!) of the class in which it is nested
- The enclosing class doesn’t have access to the members of the nested class

Static Nested Classes

- Two types of nested classes.
  - Static
  - Non-Static
- A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object
- That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used

Static Nested Classes

```
1  class OuterStaticInner {
2      private int outer_x = 100;
3
4      void test() {
5          Inner inner = new Inner();
6          inner.display( outer: this);
7      }
8      // this is a static nested class
9      static class Inner {
10         void display(OuterStaticInner outer) {
11             System.out.println(outer.outer_x);
12         }
13     }
14 }
15
16 public class StaticNestedClassDemo {
17     public static void main(String[] args) {
18         OuterStaticInner outer = new OuterStaticInner();
19         outer.test();
20         OuterStaticInner.Inner x = new OuterStaticInner.Inner();
21         x.display(outer);
22     }
23 }
```

Inner Classes

- The most important type of nested class is the inner class
- An inner class is a non-static nested class
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do
- Thus, an inner class is fully within the scope of its enclosing class

Inner Classes

```
1   class Outer1
2   {
3       private int outer_x = 100;
4
5       void test() {
6           Inner inner = new Inner();
7           inner.display();
8       }
9       // this is an inner class
10      class Inner {
11          void display() {
12              System.out.println(outer_x);
13          }
14      }
15  }
16
17  public class InnerClassDemo1 {
18      public static void main(String[] args) {
19          Outer1 outer = new Outer1();
20          outer.test();
21          Outer1.Inner innerObj = outer.new Inner();
22          innerObj.display();
23      }
24  }
```

Inner Classes

```java
class Outer2
{
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    class Inner {
        int y = 10; // y is local to Inner
        void display() { System.out.println(outer_x); }
    }

    void showy() {
        //System.out.println(y); // error, y not known here!
    }
}

public class InnerClassDemo2 {
    public static void main(String[] args) {
        Outer2 outer = new Outer2();
        outer.test();
    }
}
```

Variable Arguments

```
1  public class VarArgsTest {
2      static void vaTest(int ... v){
3          for(int x: v) {
4              System.out.print(x + " ");
5          }
6          System.out.println();
7      }
8      static void vaTest(boolean ... v){
9          for(boolean x: v) {
10             System.out.print(x + " ");
11         }
12         System.out.println();
13     }
14     static void vaTest(String msg, int ... v){
15         System.out.print(msg + " ");
16         for(int x: v) {
17             System.out.print(x + " ");
18         }
19         System.out.println();
20     }
21     public static void main(String[] args) {
22         vaTest( msg: "Testing",  ...v: 10, 20);
23         vaTest( ...v: true, false, false);
24         vaTest( ...v: 1, 2, 3);
25     }
26 }
```

Variable Arguments Ambiguity

```
1  public class VarArgsTest {
2      static void vaTest(int ... v){
3          for(int x: v) {
4              System.out.print(x + " ");
5          }
6          System.out.println();
7      }
8      static void vaTest(boolean ... v){
9          for(boolean x: v) {
10             System.out.print(x + " ");
11         }
12         System.out.println();
13     }
14     static void vaTest(int n, int ... v){
15         for(int x: v) {
16             System.out.println(x + " ");
17         }
18     }
19     public static void main(String[] args) {
20         vaTest(); // ambiguity type 1 because of int and boolean but works with int and double
21         vaTest(1, 2, 3); // ambiguity type 2 with vaTest(int n, int ... v) and vaTest(int ... v)
22     }
23 }
24
```

Java

Strings

String related classes

- Java provides three String related classes
- java.lang package
  - ***String*** class: Storing and processing Strings but Strings created using the String class cannot be modified (**immutable**)
  - ***StringBuffer*** class: Create flexible Strings that can be modified
- java.util package
  - ***StringTokenizer*** class: Can be used to extract tokens from a String

String

String

- String class provide many constructors and more than 40 methods for examining in individual characters in a sequence
- You can create a String from a String value or from an array of characters.
  - String newString = new String(stringValue);
- The argument stringValue is a sequence of characters enclosed inside double quotes
  - String message = new String (“Welcome”);
  - String message = “Welcome”;

String Constructors

```
3   public class StringConstructorTest {
4       public static void main(String[] args) {
5           char charArray[ ] = { 'b',  'i',  'r',  't',  'h',  ' ',  'd',  'a',  'y'} ;
6           byte byteArray[ ] = { ( byte)  'n',  ( byte)  'e',  ( byte)  'w', ( byte)  ' ',
7                   ( byte)  'y',  ( byte)  'e',  ( byte)  'a',  ( byte)  'r'} ;
8
9           String s = new String("hello"); // hello
10          String s1 = new String() ; //
11          String s2 = new String(s) ; // hello
12          String s3 = new String(charArray); // birth day
13          String s4 = new String(charArray, 6, 3); // day
14          String s5 = new String(byteArray, 4, 4); // year
15          String s6 = new String(byteArray); // new year
16          String s7 = "Wel"  +  "come"; // Welcome
17
18          System.out.println(s);
19          System.out.println(s1);
20          System.out.println(s2);
21          System.out.println(s3);
22          System.out.println(s4);
23          System.out.println(s5);
24          System.out.println(s6);
25          System.out.println(s7);
26      }
27  }
```

String Length

- Returns the length of a String
  - ***length()***
- Example:

  String s1="Hello";

  System.out.println(s1.length());

Extraction

- Get the character at a specific location in a string
  - ***s1.charAt(1)***
- Get the entire set of characters in a string
  - ***s1.getChars(0, 5, charArray, 0)***

Extracting Substrings

- substring method enable a new String object to be created by copying part of an existing String object
  - ***substring (int startIndex)*** - copies the characters form the starting index to the end of the String
  - ***substring(int beginIndex, int endIndex)*** - copies the characters from the starting index to one beyond the endIndex

String Comparisons

- *equals*
  - Compare any two string objects for equality using lexicographical comparison. ***s1.equals("hello")***
- *equalsIgnoreCase*
  - ***s1.equalsIgnoreCase(s2)***
- *compareTo*
  - ***s1.compareTo(s2)***
  - s1 > s2 (positive), s1 < s2 (negative), s1 = s2 (zero)

String Comparisons

```java
public class StringEqualsTest {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = new String( original: "Hello");
        String s3 = "Hello";
        System.out.println("s1 == Hello " + s1.equals("Hello")); // true
        System.out.println("s1 == s2 " + s1.equals(s2)); // true
        System.out.println("s1 == s3 " + s1.equals(s3)); // true
        System.out.println("s2 == s3 " + s2.equals(s3)); // true
        System.out.println(s1 == s2); // false
        System.out.println(s1 == s3); // true
        System.out.println(s2 == s3); // false
    }
}
```

String Comparisons

- *regionMatches* compares portions of two String objects for equality
  - ***s1.regionMatches (0, s2, 0, 5)***
  - ***s1.regionMatches (true, 0, s2, 0, 5)***
- If the first argument is true, the method ignores the case of the characters being compared
- *startsWith* and endsWith check whether a String starts or ends with a specified String
  - ***s1.startsWith (s2)***
  - ***s1.endsWith (s2)***

String Concatenation

- Java provide the *concat* method to concatenate two strings.

  String s1 = new String (“Happy ”);

  String s2 = new String (“Birthday”);

  String s3 = s1.concat(s2);

  s3 will be “Happy Birthday”

String Search

- Find the position of character/String within a String
  - ***int indexOf(char ch)***
  - ***int lastIndexOf(char ch)***

String Split

- split() method splits a String against given regular expression and returns a character array
- ***String test = "abc,def,123";***

  String[] out = test.split(",");

  out[0] - abc, out[1] - def, out[2] - 123

String Conversions

- Generally, the contents of a String cannot be changed once the string is created,
- Java provides conversion methods
- ***toUpperCase()*** and ***toLowerCase()***
  - Converts all the characters in the string to lowercase or uppercase
- ***trim()***
  - Eliminates blank characters from both ends of the string
- ***replace(oldChar, newChar)***
  - Replaces a character in the string with a new character

String to Other Conversions

- The String class provides ***valueOf*** methods for converting a character, an array of characters and numeric values to strings
  - ***valueOf*** method take different argument types

String to Other Conversions

Type	To String	From String
boolean	String.valueOf(boolean)	Boolean.parseBoolean(String)
byte	String.valueOf(int)	Byte.parseByte(String, int base)
short	String.valueOf(int)	Short.parseShort (String, int base)
Int	String.valueOf(int)	Integer.parseInt (String, int base)
long	String.valueOf(long)	Long.parseLong (String, int base)
float	String.valueOf(float)	Float.parseFloat(String)
double	String.valueOf(double)	Double.parseDouble(String)

String Conversion Example

- To convert an int to a String (3 different ways):

  int n = 123;

  String s1 = Integer.toString(n);

  String s2 = String.valueOf(n);

  String s3 = n + "";

- To convert a string to an int:

  String s = "1234";

  int n = Integer.parseInt(s);

StringBuffer

StringBuffer

- Can be used wherever a string is used
  - More flexible than String
  - Can add, insert, or append new contents into a string buffer
- The StringBuffer class has three constructors and more than 30 methods for managing the buffer and for modifying strings in the buffer
- Every StringBuffer is capable of storing a number of characters specified by its capacity

StringBuffer Constructors

- ***public StringBuffer()***
  - No characters in it and an initial capacity of 16 characters
- ***public StringBuffer(int length)***
  - No characters in it and an initial capacity specified by the length argument
- ***public StringBuffer(String string)***
  - Contains String argument and an initial capacity of the buffer is 16 plus the length of the argument

StringBuffer

```
3   public class StringBufferTest {
4       public static void main(String[] args) {
5           StringBuffer sb = new StringBuffer("hello");
6           System.out.println(sb.capacity());
7           sb.setLength(20);
8           System.out.println(sb.length());
9           System.out.println(sb.charAt(0));
10          sb.append('w');
11          sb.append("orld");
12          System.out.println(sb);
13          sb.insert(5, ' ');
14          System.out.println(sb);
15          sb.delete(5, sb.length());
16          System.out.println(sb);
17          sb.reverse();
18          System.out.println(sb);
19      }
20  }
21
```

StringTokenizer

StringTokenizer

- Break a string into pieces (tokens) so that information contained in it can be retrieved and processed
- Specify a set of characters as delimiters when constructing a StringTokenizer object
- **StringTokenizer** class is available since **JDK 1.0** and the **String.split()** is available since **JDK 1.4**
- *StringTokenizer is a legacy class, retained for compatibility reasons, the use is discouraged!*

StringTokenizer

- Constructors
  - ***StringTokenizer(String str, String delim)***
  - ***StringTokenizer(String str)***
- Methods
  - ***hasMoreToken()*** - Returns true if there is a token left in the string
  - ***nextToken()*** - Returns the next token in the string
  - ***nextToken(String delim)*** - Returns the next token in the string after reseting the delimiter to delim
  - ***countToken( )*** - Returns the number of tokens remaining in the string tokenizer

StringTokenizer

```
3   import java.util.StringTokenizer;
4
5   public class StringTokenizerTest {
6       public static void main(String[] args) {
7           String s = new String("what's your news");
8           StringTokenizer tokens = new StringTokenizer(s);
9           System.out.println(tokens.countTokens());
10          while (tokens.hasMoreTokens()) {
11              System.out.println(tokens.nextToken());
12          }
13
14          String t = new String("what's,your,news");
15          tokens = new StringTokenizer(t, ",");
16          System.out.println(tokens.countTokens());
17          while (tokens.hasMoreTokens()) {
18              System.out.println(tokens.nextToken());
19          }
20      }
21  }
```

Java

Inheritance

Inheritance

- Same inheritance concept of C++ in Java with some modifications
- In Java,
  - One class inherits the other using ***extends*** keyword
  - The classes involved in inheritance are known as ***superclass*** and ***subclass***
  - ***Multilevel*** inheritance but no ***multiple*** inheritance
  - There is a special way to call the superclass's ***constructor***
  - There is automatic ***dynamic method dispatch***

Simple Inheritance

```java
class A {
    int i, j;

    void showij() {
        System.out.println(i+" "+j);
    }
}

class B extends A{
    int k;

    void showk() {
        System.out.println(k);
    }

    void sum() {
        System.out.println(i+j+k);
    }
}
```

```java
public class SimpleInheritance {
    public static void main(String[] args) {
        A superOb = new A();
        superOb.i = 10;
        superOb.j = 20;
        superOb.showij();
        B subOb = new B();
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        subOb.showij();
        subOb.showk();
        subOb.sum();
    }
}
```

Practical Example

```
3   class Box {
4       double width, height, depth;
5
6       Box(Box ob) {
7           width = ob.width; height = ob.height; depth = ob.depth;
8       }
9
10      Box(double w, double h, double d) {
11          width = w; height = h; depth = d;
12      }
13
14      Box() { width = height = depth = 1; }
17
18      Box(double len) { width = height = depth = len; }
21
22      double volume() { return width * height * depth; }
25  }
26
27  class BoxWeight extends Box {
28      double weight;
29
30      BoxWeight(double w, double h, double d, double m) {
31          width = w; height = h; depth = d; weight = m;
32      }
33  }
```

Superclass variable reference to Subclass object

```
34
35  public class RealInheritance {
36      public static void main(String[] args) {
37          BoxWeight weightBox = new BoxWeight( w: 3,  h: 5,  d: 7,  m: 8.37);
38          System.out.println(weightBox.weight);
39          Box plainBox = weightBox; // assign BoxWeight reference to Box reference
40          System.out.println(plainBox.volume()); // OK, volume() defined in Box
41          System.out.println(plainBox.weight); // Error, weight not defined in Box
42          Box box = new Box( w: 1,  h: 2,  d: 3); // OK
43          BoxWeight wbox = box; // Error, can't assign Box reference to BoxWeight
44      }
45  }
46
```

Using super

```
3   class BoxWeightNew extends Box {
4       double weight;
5
6       BoxWeightNew(BoxWeightNew ob) {
7           super(ob);
8           weight = ob.weight;
9       }
10
11      BoxWeightNew(double w, double h, double d, double m) {
12          super(w, h, d);
13          weight = m;
14      }
15
16      BoxWeightNew() {
17          super(); // must be the 1st statement in constructor
18          weight = 1;
19      }
20
21      BoxWeightNew(double len, double m) {
22          super(len);
23          weight = m;
24      }
25
26      void print() {
27          System.out.println("Box(" + width + ", " + height +
28                             ", " + depth + ", " + weight + ")");
29      }
30  }
```

Using super

```
31
32  public class SuperTest {
33      public static void main(String[] args) {
34          BoxWeightNew box1 = new BoxWeightNew(10, 20, 15, 34.3);
35          BoxWeightNew box2 = new BoxWeightNew(2, 3, 4, 0.076);
36          BoxWeightNew box3 = new BoxWeightNew();
37          BoxWeightNew cube = new BoxWeightNew(3, 2);
38          BoxWeightNew clone = new BoxWeightNew(box1);
39          box1.print();
40          box2.print();
41          box3.print();
42          cube.print();
43          clone.print();
44      }
45
46  }
47
```

Using super

```
3  class C {
4      int i;
5      void show() {
6      }
7  }
8
9  class D extends C {
10     int i; // this i hides the i in C
11
12     D(int a, int b) {
13         super.i = a; // i in C
14         i = b; // i in D
15     }
16
17     void show() {
18         System.out.println("i in superclass: " + super.i);
19         System.out.println("i in subclass: " + i);
20         super.show();
21     }
22 }
23
24 public class UseSuper {
25     public static void main(String[] args) {
26         D subOb = new D( a: 1,  b: 2);
27         subOb.show();
28     }
29 }
```

Multilevel Inheritance

```
3  class X {
4      int a;
5      X() {
6          System.out.println("Inside X's constructor");
7      }
8  }
9
10 class Y extends X {
11     int b;
12     Y() {
13         System.out.println("Inside Y's constructor");
14     }
15 }
16
17 class Z extends Y {
18     int c;
19     Z() {
20         System.out.println("Inside Z's constructor");
21     }
22 }
23
24 public class MultilevelInheritance {
25     public static void main(String[] args) {
26         Z z = new Z();
27         z.a = 10;
28         z.b = 20;
29         z.c = 30;
30     }
31 }
```

Inside X's constructor
Inside Y's constructor
Inside Z's constructor

Method Overriding

```java
class Base {
    int a;
    Base(int a) {
        this.a = a;
    }
    void show() {
        System.out.println(a);
    }
}

class Child extends Base {
    int b;

    Child(int a, int b) {
        super(a);
        this.b = b;
    }

    // the following method overrides Base class's show()
    @Override // this is an annotation (optional but recommended)
    void show() {
        System.out.println(a + ", " + b);
    }
}
```

```java
public class MethodOverride {
    public static void main(String[] args) {
        Child o = new Child( a: 10,  b: 20);
        o.show();
        Base b = o;
        b.show(); // will call show of Override
    }
}
```

Dynamic Method Dispatch

```
3  class P {
4      void call() {
5          System.out.println("Inside P's call method");
6      }
7  }
8  class Q extends P {
9      void call() {
10         System.out.println("Inside Q's call method");
11     }
12 }
13 class R extends Q {
14     void call() {
15         System.out.println("Inside R's call method");
16     }
17 }
18
19 public class DynamicDispatchTest {
20     public static void main(String[] args) {
21         P p = new P(); // object of type P
22         Q q = new Q(); // object of type Q
23         R r = new R(); // object of type R
24         P  x;          // reference of type P
25         x = p;         // x refers to a P object
26         x.call();      // invoke P's call
27         x = q;         // x refers to a Q object
28         x.call();      // invoke Q's call
29         x = r;         // x refers to a R object
30         x.call();      // invoke R's call
31     }
32 }
```

Abstract Class

- ***abstract class A***
- contains abstract method ***abstract method f()***
- No instance can be created of an abstract class
- The subclass must implement the abstract method
- Otherwise the subclass will be a abstract class too

Abstract Class

```java
abstract class S {
    // abstract method
    abstract void call();
    // concrete methods are still allowed in abstract classes
    void call2() {
        System.out.println("This is a concrete method");
    }
}

class T extends S {
    void call() {
        System.out.println("T's implementation of call");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        //S s = new S(); // S is abstract; cannot be instantiated
        T t = new T();
        t.call();
        t.call2();
    }
}
```

Anonymous Subclass

```
3  abstract class S {
4      // abstract method
5      abstract void call();
6      // concrete methods are still allowed in abstract classes
7      void call2() {
8          System.out.println("This is a concrete method");
9      }
10 }
11
12 class AbstractDemo {
13     public static void main(String args[]) {
14         //S s = new S(); // S is abstract; cannot be instantiated
15         S s = new S() {
16             void call() {
17                 System.out.println("Call method of an abstract class");
18             }
19         };
20         s.call();
21     }
22 }
```

Using final with Inheritance

To prevent overriding

```
class A {
    final void f() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void f() { // Error! Can't override.
        System.out.println("Illegal!");
    }
}
```

To prevent inheritance

```
final class A {
    //...
}

// The following class is illegal.
class B extends A { // Error! Can't subclass A
    //...
}
```

Object Class

- There is one special class, ***Object***, defined by Java
- All other classes are subclasses of Object
- That is, Object is a superclass of all other classes
- This means that a reference variable of type Object can refer to an object of any other class
- Also, since arrays are implemented as classes, a variable of type Object can also refer to any array

Object's toString()

- The **toString**( ) method returns a string that contains a description of the object on which it is called
- Also, this method is automatically called when an object is output using println()
- Many classes override this method
- Doing so allows them to provide a description specifically for the types of objects that they create

Object's toString()

```
3   class Point {
4       int x, y;
5
6       Point(int x, int y) {
7           this.x = x;
8           this.y = y;
9       }
10
11      @Override
12      public String toString() {
13          return "(" + x + ", " + y + ")";
14      }
15  }
16
17  public class ObjectTest {
18      public static void main(String[] args) {
19          Point p1 = new Point( x: 10, y: 20);
20          // without override toString() method the
21          // following will print something like this
22          // Point@3cd1a2f1
23          System.out.println(p1);
24      }
25  }
26
```

Object's equals() and hashCode()

- **==** is a reference comparison, whether both variables refer to the same object
- Object's **equals**() method does the same thing
- String class override **equals**() to check contents
- If you want two different objects of a same class to be equal then you need to override **equals**() and **hashCode**() methods
  - **hashCode**() needs to return same value to work properly as keys in Hash data structures

Object's equals() and hashCode()

```java
import java.util.HashMap;
import java.util.Objects;

class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Point)) {
            return false;
        }
        Point p = (Point) o;
        if (p.x == this.x && p.y == this.y) return true;
        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

```java
public class ObjectTest {
    public static void main(String[] args) {
        Point p1 = new Point( x: 10, y: 20);
        Point p2 = new Point( x: 10,  y: 20);
        System.out.println(p1.equals(p2));
        System.out.println(p1 == p2);
        HashMap m = new HashMap();
        m.put(p1, "Hello");
        System.out.println(m.get(p2));
    }
}
```

Java

Package, Interface & Exception

Package

Package

- Java package provides a mechanism for partitioning the class name space into more manageable chunks
  - Both **naming** and **visibility** control mechanism
- Define classes inside a package that are not accessible by code outside that package
- Define class members that are exposed only to other members of the same package
- This allows classes to have intimate knowledge of each other
  - Not expose that knowledge to the rest of the world

Declaring Package

- ***package pkg***
  - Here, pkg is the name of the package
- ***package MyPackage***
  - creates a package called MyPackage
- The package statement defines a name space in which classes are stored
- If you omit the package statement, the class names are put into the **default package**, which has no name

Declaring Package

- Java uses file system directories to store packages
  - the .class files for any classes that are part of MyPackage must be stored in a directory called MyPackage
- More than one file can include the same package statement
- The package statement simply specifies to which package the classes defined in a file belong
- To create hierarchy of packages, separate each package name from the one above it by use of a (.)

Package Example

```
1  package MyPackage;
2
3  class Balance {
4      String name;
5      double bal;
6
7      Balance(String n, double b) {
8          name = n;
9          bal = b;
10     }
11     void show() {
12         if(bal < 0)
13             System.out.print("--> ");
14         System.out.println(name + ": $" + bal);
15     }
16 }
17
18 public class AccountBalance {
19     public static void main(String[] args) {
20         Balance current[] = new Balance[3];
21         current[0] = new Balance("K. J. Fielding", 123.23);
22         current[1] = new Balance("Will Tell", 157.02);
23         current[2] = new Balance("Tom Jackson", -12.33);
24         for(Balance b : current) {
25             b.show();
26         }
27     }
28 }
```

javac -d . AccountBalance.java

java MyPackage.AccountBalance

Package Syntax

- The general form of a multilevel package statement
  - ***package pkg1[.pkg2[.pkg3]]***
  - ***package java.awt.image***
- In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions
- The general form of the import statement
  - ***import pkg1 [.pkg2].(classname | *)***
  - ***import java.util.Scanner***

Access Protection

- Packages act as containers for classes and other subordinate packages
- Classes act as containers for data and code
- The class is Java's smallest unit of abstraction
- Four categories of visibility for class members
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different package
  - Classes that are neither in the same package nor subclasses

Access Protection

- The three access modifiers provide a variety of ways to produce the many levels of access required
  - private, public, and protected
- The following applies only to members of classes

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access Protection

- Anything declared ***public*** can be accessed from anywhere
- Anything declared ***private*** cannot be seen outside of its class
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package (***default access***)
- If you want to allow an element to be seen outside your current package, but only to classes that subclass the class directly, then declare that element ***protected***

Access Protection

- A non-nested class has only two possible access levels
  - default and public
- When a class is declared as public, it is accessible by any other code
- If a class has default access, then it can only be accessed by other code within its same package
- When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class

Interface

Interface

- We can call it a pure abstract class having no concrete methods
  - All methods declared in an interface are implicitly **public** and **abstract**
  - All variables declared in an interface are implicitly **public**, **static** and **final**
- *An interface can't have instance variables, so can't maintain state information unlike class*
- A class can only extend from a **single class**, but a class can implement **multiple interfaces**

Implementing Interface

- When you implement an interface method, it must be declared as public
- By implementing an interface, a class signs a contract with the compiler that it will definitely provide implementation of all the methods
- If it fails to do so, the class will be considered as abstract
- Then it must be declared as abstract and no object of that class can be created

Simple Interface

```
3  interface Callback
4  {
5      void callback(int param);
6  }
7
8  class Client implements Callback
9  {
10     public void callback(int p)
11     {
12         System.out.println("callback called with " + p);
13     }
14 }
15
16 public class InterfaceTest {
17     public static void main(String[] args) {
18         // Can't instantiate an interface directly
19         //Callback c1 = new Callback();
20         //c1.callback(21);
21         Client c2 = new Client();
22         c2.callback(42);
23         // Accessing implementations through Interface reference
24         Callback c3 = new Client();
25         c3.callback(84);
26     }
27 }
```

Applying Interfaces

```
3  interface MyInterface {
4      void print(String msg);
5  }
6
7  class MyClass1 implements MyInterface {
8      public void print(String msg) {
9          System.out.println(msg + ":" + msg.length());
10     }
11 }
12
13 class MyClass2 implements MyInterface {
14     public void print(String msg) {
15         System.out.println(msg.length() + ":" + msg);
16     }
17 }
18
19 public class InterfaceApplyTest {
20     public static void main(String[] args) {
21         MyClass1 mc1 = new MyClass1();
22         MyClass2 mc2 = new MyClass2();
23         MyInterface mi; // create an interface reference variable
24         mi = mc1;
25         mi.print("Hello World");
26         mi = mc2;
27         mi.print("Hello World");
28     }
29 }
```

Variables in Interfaces

```
3  import java.util.Random;
4
5  interface SharedConstants {
6      int NO = 0;
7      int YES = 1;
8      int LATER = 3;
9      int SOON = 4;
10     int NEVER = 5;
11 }
12
13 class Question implements SharedConstants {
14     Random rand = new Random();
15     int ask() {
16         int prob = (int) (100 * rand.nextDouble());
17         if (prob < 30) return NO;
18         else if (prob < 60) return YES;
19         else if (prob < 75) return LATER;
20         else if (prob < 98) return SOON;
21         else return NEVER;
22     }
23 }
24
25 public class InterfaceVariableTest {
26     public static void main(String[] args) {
27         Question q = new Question();
28         System.out.println(q.ask());
29     }
30 }
```

Extending Interfaces

```
3  interface I1 {
4      void f1();
5      void f2();
6  }
7
8  interface I2 extends I1 {
9      void f3();
10 }
11
12 class MyClass implements I2 {
13     public void f1() { System.out.println("Implement f1"); }
16     public void f2() { System.out.println("Implement f2"); }
19     public void f3() { System.out.println("Implement f3"); }
22 }
23
24 public class InterfaceExtendsTest {
25     public static void main(String[] args) {
26         MyClass m = new MyClass();
27         m.f1();
28         m.f2();
29         m.f3();
30     }
31 }
```

Default Interface Methods

- Prior to JDK 8, an interface could not define any implementation whatsoever
- The release of JDK 8 has changed this by adding a new capability to interface called the *default method*
  - A default method lets you define a default implementation for an interface method
  - Its primary motivation was to provide a means by which interfaces could be expanded without breaking existing code

Default Interface Methods

```
3  interface MyIF {
4      // This is a "normal" interface method declaration.
5      int getNumber();
6      // This is a default method. Notice that it provides
7      // a default implementation.
8      default String getString() {
9          return "Default String";
10     }
11 }
12
13 class MyIFImp implements MyIF {
14     // Only getNumber() defined by MyIF needs to be implemented.
15     // getString() can be allowed to default.
16     public int getNumber() {
17         return 100;
18     }
19 }
20
21 public class InterfaceDefaultMethodTest {
22     public static void main(String[] args) {
23         MyIFImp m = new MyIFImp();
24         System.out.println(m.getNumber());
25         System.out.println(m.getString());
26     }
27 }
```

Multiple Inheritance Issues

```
3  interface Alpha {
4      default void reset() {
5          System.out.println("Alpha's reset");
6      }
7  }
8
9  interface Beta {
10     default void reset() {
11         System.out.println("Beta's reset");
12     }
13 }
14
15 class TestClass implements Alpha, Beta {
16     public void reset() {
17         System.out.println("TestClass's reset");
18     }
19 }
```

```
3  interface Alpha {
4      default void reset() {
5          System.out.println("Alpha's reset");
6      }
7  }
8
9  interface Beta extends Alpha {
10     default void reset() {
11         System.out.println("Beta's reset");
12         // Alpha.super.reset();
13     }
14 }
15
16 class TestClass implements Beta {
17
18 }
```

Static Methods in Interface

```
3   interface MyIFStatic {
4
5       int getNumber();
6
7       default String getString() {
8           return "Default String";
9       }
10
11      // This is a static interface method.
12  @   static int getDefaultNumber() {
13          return 0;
14      }
15  }
16
17  public class InterfaceStaticMethodTest {
18      public static void main(String[] args) {
19          System.out.println(MyIFStatic.getDefaultNumber());
20      }
21  }
```

Exception

Exception Handling

- Uncaught exceptions
- Caught exceptions
- try
- catch
- finally
- throw
- throws
- Creating custom exceptions

Uncaught Exceptions

```
3   public class TestException1
4   {
5       public static void main(String args[]) {
6           int a = 10, b = 0;
7           int c = a/b; // ArithmeticException: / by zero
8           System.out.println(a);
9           System.out.println(b);
10          System.out.println(c);
11          String s = null;
12          System.out.println(s.length()); // NullPointerException
13      }
14  }
```

Caught Exceptions

```
3   public class TestException2
4   {
5       public static void main(String args[])
6       {
7           int a = 10, b = 0, c = 0;
8           try {
9               c = a/b;
10          } catch(Exception e) {
11              System.out.println (e);
12          } finally {
13              // finally block will always execute
14              System.out.println ("In Finally");
15          }
16          System.out.println(a);
17          System.out.println(b);
18          System.out.println(c);
19      }
20  }
```

Caught Exceptions

```
3   public class TestException5
4   {
5       public static void main(String args[])
6       {
7           int a = 10, b = 0, c = 0;
8           try {
9               c = a / b;
10          } catch(ArithmeticException e1) {
11              System.out.println(e1);
12          } catch(NullPointerException e2) {
13              System.out.println(e2);
14          } finally {
15              // finally block will always execute
16              System.out.println ("In Finally");
17          }
18          System.out.println(a);
19          System.out.println(b);
20          System.out.println(c);
21      }
22  }
```

catch(ArithmeticException | NullPointerException e)

Throws

```
3   public class TestException3
4   {
5       public static void f() throws Exception {
6           int a = 10;
7           int b = 0;
8           int c = a/b;
9       }
10
11      public static void main(String args[])
12      {
13          try {
14              f();
15          } catch(Exception e) {
16              System.out.println (e);
17              e.printStackTrace();
18          }
19          System.out.println("Hello World");
20      }
21  }
```

Creating Custom Exceptions

```
3  class MyException extends Exception {
4      private int detail;
5
6      MyException(int a) {
7          detail = a;
8      }
9
10     public String toString() {
11         return "My Exception :  "  + detail;
12     }
13 }
14
15 public class TestException4 {
16     static void compute(int a) throws MyException {
17         if(a > 10) {
18             throw new MyException(a);
19         }
20         System.out.println(a);
21     }
22
23     public static void main(String args[]) {
24         try {
25             compute(10);
26             compute(20);
27         } catch(MyException e) {
28             System.out.println(e);
29         }
30     }
31 }
```

Java

Enumeration, Type Wrappers and Autoboxing

Enumeration

- An enumeration is a list of named constants
- Java enumerations is similar to enumerations in other languages with some differences
- In Java, an enumeration defines a class type
- In Java, an enumeration can have constructors, methods, and instance variables
- ***Example****: EnumDemo.java*

```java
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
public class EnumDemo {
    public static void main(String args[]) {
        Apple ap;
        ap = Apple.RedDel;
        // Output an enum value.
        System.out.println("Value of ap: " + ap);
        System.out.println();
        ap = Apple.GoldenDel;
        // Compare two enum values.
        if(ap == Apple.GoldenDel)
            System.out.println("ap contains GoldenDel.\n");
        // Use an enum to control a switch statement.
        switch(ap) {
            case Jonathan:
                System.out.println("Jonathan is red.");
                break;
            case GoldenDel:
                System.out.println("Golden Delicious is yellow.");
                break;
            case RedDel:
                System.out.println("Red Delicious is red.");
                break;
            case Winesap:
                System.out.println("Winesap is red.");
                break;
            case Cortland:
                System.out.println("Cortland is red.");
                break;
```

Enumeration

- All enumerations automatically contain two predefined methods:

  public static enum-type [ ] values( )

  – Returns an array that contains a list of the enumeration constants

  public static enum-type valueOf(String s)

  – Returns the enumeration constant whose value corresponds to the string passed in s

- ***Example****: EnumDemo2.java*

```java

public class EnumDemo2 {
    public static void main(String args[])
    {
        Apple ap;
        System.out.println("Here are all Apple constants:");
        // use values()
        Apple allApples[] = Apple.values();
        for(Apple a : allApples)
            System.out.println(a);
        System.out.println();
        // use valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}


```

Enumeration

- Java enumeration is a class type
  - Although you don't instantiate an enum using new
- Enumeration can have constructors, instance variables and methods
  - Each enumeration constant is an object of its enumeration type
  - The constructor is called when each enumeration constant is created
  - Each enumeration constant has its own copy of any instance variables defined by the enumeration
- ***Example****: EnumDemo3.java*

```java
// Use an enum constructor, instance variable, and method.
enum AppleNew {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
    private int price; // price of each apple
    // Constructor
    AppleNew(int p) {
        price = p;
    }

    int getPrice() {
        return price;
    }
}

public class EnumDemo3 {
    public static void main(String args[]) {
        AppleNew ap;
        // Display price of Winesap.
        System.out.println("Winesap costs " +
                AppleNew.Winesap.getPrice() +
                " cents.\n");
        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(AppleNew a : AppleNew.values())
            System.out.println(a + " costs " + a.getPrice() +
                    " cents.");
    } }
```

Type Wrappers

- Despite the performance benefit offered by the primitive types, there are times when you will need an object representation
  - you can't pass a primitive type by reference to a method
  - many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types

Type Wrappers

- Java provides ***type wrappers***
  - classes that encapsulate a primitive type within an object
- The type wrappers are:
  - Character
  - Boolean
  - Double, Float, Long, Integer, Short, Byte

Type Wrappers

```
public class WrapDemo {
    public static void main(String args[]) {
        Integer iOb = new Integer(100);
        int i = iOb.intValue();
        System.out.println(i + " " + iOb);
    }
}
```

- The process of encapsulating a value within an object is called ***boxing***

  Integer iOb = new Integer(100);

- The process of extracting a value from a type wrapper is called ***unboxing***

  int i = iOb.intValue();

Auto (boxing/unboxing)

- ***Autoboxing***
  - the process by which a primitive type is automatically encapsulated into its equivalent type wrapper whenever an object of that type is needed
  - There is no need to explicitly construct an object
- ***Auto-unboxing***
  - the process by which the value of a boxed object is automatically extracted from a type wrapper when its value is needed
  - There is no need to call a method such as intValue() or doubleValue()

Autoboxing

- With autoboxing, it is no longer necessary to manually construct an object in order to wrap a primitive type
- You need only assign that value to a type-wrapper reference
- Java automatically constructs the object for you

  Integer iOb = 100; // autobox an int100

- Notice that the object is not explicitly created through the use of new. Java handles this for you, automatically

Auto-unboxing

- To unbox an object, simply assign that object reference to a primitive-type variable

  int i = iOb; // auto-unbox

- Java handles the details for you
- ***Example****: AutoBoxingUnboxingDemo.java*

```java
/**
 * Created by rifat on 31/10/2015.
 */
public class AutoBoxingUnboxingDemo {

    static int m(Integer v) {
        return v ; // auto-unbox to int
    }

    public static void main(String args[]) {
        // Pass an int to m() and assign the return value
        // to an Integer.  Here, the argument 100 is autoboxed
        // into an Integer.  The return value is also autoboxed
        // into an Integer.
        Integer a  = m(100);
        System.out.println(a);

        Integer iOb, iOb2;
        int i;
        iOb = 100;
        System.out.println("Original value of iOb: " + iOb);
        // The following automatically unboxes iOb,
        // performs the increment, and then reboxes
        // the result back into iOb.
        ++iOb;
        System.out.println("After ++iOb: " + iOb);
        // Here, iOb is unboxed, the expression is
        // evaluated, and the result is reboxed and
        // assigned to iOb2.
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 after expression: " + iOb2);
```

```
    Integer iOb, iOb2;
    int i;
    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);
    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);
    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);
    // The same expression is evaluated, but the
    // result is not reboxed.
    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);

    Integer intOb = 100;
    Double doubleOb = 98.6;
    // Auto-unboxing also allows you to mix different types of
    // numeric objects in an expression. Once the values are unboxed,
    // the standard type promotions and conversions are applied.
    doubleOb = doubleOb + intOb;
    System.out.println("dOb after expression: " + doubleOb);

}
```

Java

Thread

Multitasking

- Multitasking allows several activities to occur concurrently on the computer
- Levels of multitasking:
  - **Process-based multitasking**
    - Allows programs (processes) to run concurrently
  - **Thread-base multitasking (multithreading)**
    - Allows parts of the same process (threads) to run concurrently

Multitasking

- Advantages of multithreading over process-based multitasking
  - Threads share the same address space
  - Context switching between threads is usually inexpensive
  - Communication between thread is usually inexpensive
- Java supports ***thread-based multitasking*** and provides high-level facilities for ***multithreaded programming***

Main Thread

- When a Java program starts up, one thread begins running immediately
- This is called the ***main thread*** of the program
- It is the thread from which the child threads will be spawned
- Often, it must be the last thread to finish execution

Main Thread

```
3   public class MainThread {
4       public static void main(String[] args) {
5           Thread t = Thread.currentThread();
6           System.out.println("Current thread: " + t);
7           // change the name of the thread
8           t.setName("My Thread");
9           System.out.println("After name change: " + t);
10          try
11          {
12              for(int n = 5; n > 0; n--)
13              {
14                  System.out.println(n);
15                  Thread.sleep(1000);
16              }
17          }catch (InterruptedException e)
18          {
19              System.out.println("Main thread interrupted");
20          }
21      }
22  }
```

How to create Thread

1. By implementing ***Runnable*** Interface
2. By extending the ***Thread*** class itself

- *Implementing Runnable*
  - Need to implement the public void run() method
- *Extending Thread*
  - Need to override the public void run() method
- Which one is better ?

Implementing Runnable

```
3   class NewThread1 implements Runnable
4   {
5       Thread t;
6       NewThread1() {
7           t = new Thread(this, "Runnable Thread");
8           t.start();
9       }
10      // This is the entry point for the thread.
11      public void run() {
12          try {
13              for(int i = 5; i > 0; i--) {
14                  System.out.println("Child Thread: " + i);
15                  Thread.sleep(500);
16              }
17          } catch (InterruptedException e) {
18              System.out.println("Child interrupted.");
19          }
20          System.out.println("Exiting child thread.");
21      }
22  }
23
24  public class RunnableThread {
25      public static void main(String[] args) {
26          new NewThread1();
27      }
28  }
```

Extending Thread

```
3   class NewThread2 extends Thread
4   {
5       NewThread2() {
6           super("Extends Thread");
7           start();
8       }
9       // This is the entry point for the thread.
10      public void run() {
11          try {
12              for(int i = 5; i > 0; i--) {
13                  System.out.println("Child Thread: " + i);
14                  Thread.sleep(500);
15              }
16          } catch (InterruptedException e) {
17              System.out.println("Child interrupted.");
18          }
19          System.out.println("Exiting child thread.");
20      }
21  }
22
23  public class ExtendsThread {
24      public static void main(String[] args) {
25          new NewThread2();
26      }
27  }
```

Multiple Threads

- It is possible to create more than one thread inside the main
- In multiple threads, often you will want the main thread to finish last. This is accomplished by
  - using a large delay in the main thread
  - using the **join()** method
- Whether a thread has finished or not can be known using **isAlive()** method
- ***Example****: MultipleThreads.java, JoinAliveThreads.java*

```java
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadName) {
        name = threadName;
        t = new Thread(this, name);
        t.start(); // Start the thread
    }
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i+"\t"+Thread.currentThread());
                System.out.println();
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}
public class MultipleThreads {
    public static void main(String[] args) {
        NewThread newThread1 = new NewThread("One");
```

```java
}
public class MultipleThreads {
    public static void main(String[] args) {
        NewThread newThread1 = new NewThread("One");
        NewThread newThread2 =new NewThread("Two");
        NewThread newThread3 =new NewThread("Three");
        Thread thread=new Thread(newThread2);
        thread.start();

         try {
             // wait for other threads to end
             Thread.sleep(10000);
         } catch (InterruptedException e) {
             System.out.println("Main thread Interrupted");
         }
         System.out.println("Main thread exiting.");
    }
}

```

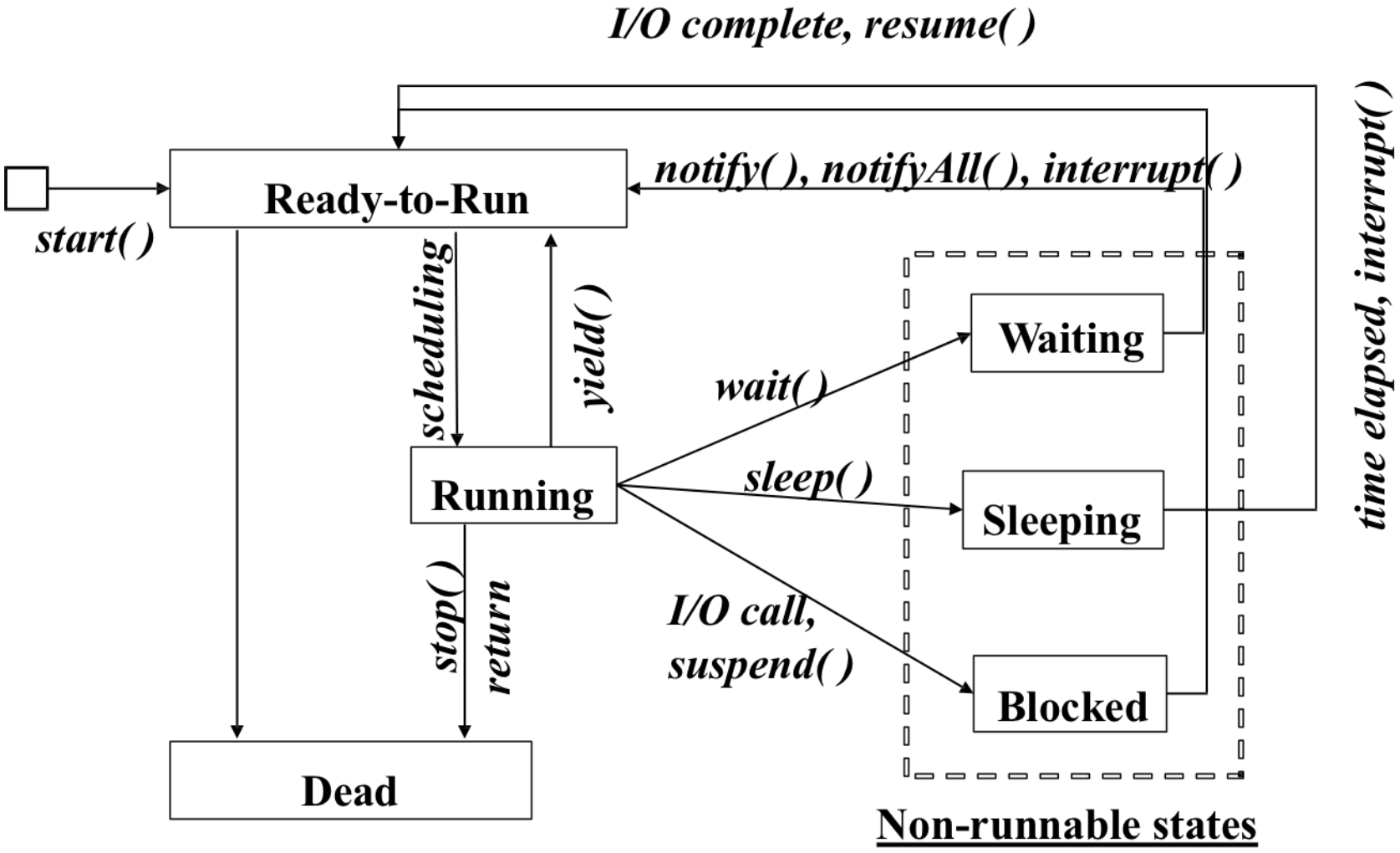
```java
public class JoinAliveThreads {
    public static void main(String[] args) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");

        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        System.out.println("Main thread exiting.");
    }
}

```

Thread States



Thread Pool

- Thread Pools are useful when you need to limit the number of threads running in your application
  - Performance overhead starting a new thread
  - Each thread is also allocated some memory for its stack
- Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool
  - As soon as the pool has any idle threads the task is assigned to one of them and executed

Thread Pool

- Thread pools are often used in multi threaded servers
  - Each connection arriving at the server via the network is wrapped as a task and passed on to a thread pool
  - The threads in the thread pool will process the requests on the connections concurrently
- Java provides Thread Pool implementation with ***java.util.concurrent.ExecutorService***

ExecutorService

```
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.TimeUnit;
6
7  public class ExecutorServiceTest {
8      public static void main(String[] args) throws Exception{
9          ExecutorService executorService = Executors.newFixedThreadPool(10);
10
11         for (int i = 0; i < 20; i++) {
12             executorService.execute(new Runnable() { // execute or submit
13                 public void run() {
14                     System.out.println("Running task");
15                     for (int j = 5; j > 0; j--) {
16                         System.out.println(j);
17                     }
18                 }
19             });
20         }
21         executorService.shutdown();
22         executorService.awaitTermination(1, TimeUnit.MINUTES);
23         System.out.println(executorService);
24     }
25 }
```

Synchronization

- When two or more threads need access to a **shared resource**, they need some way to ensure that the resource will be used by only one thread at a time
- The process by which this is achieved is called **synchronization**
- Key to synchronization is the concept of the **monitor**
- A monitor is an object that is used as a mutually exclusive lock
  - Only one thread can own a monitor at a given time

Synchronization

- When a thread acquires a lock, it is said to have entered the monitor
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor
- These other threads are said to be waiting for the monitor

Synchronization

- Two way to achieve synchronization.
- Synchronized method

  synchronized void call(String msg) { }
- Synchronized block

  public void run() {

  synchronized(target) { target.call(msg); }

  }
- ***Example****: SynchronizedBlock.java, SynchronizedMethod.java, SynchronizationLock.java*

```java
class SharedConsole2 {
     void print(String msg) {
        synchronized (this) {
            System.out.print("[" + msg);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Interrupted");
            }
            System.out.println("]");
        }
    }
}

class UserConsoleSync2 implements Runnable {
    String msg;
    SharedConsole2 console;
    Thread t;

    public UserConsoleSync2(SharedConsole2 c, String s) {
        console = c;
        msg = s;
        t = new Thread(this);
        t.start();
    }
```

```
19
20      public UserConsoleSync2(SharedConsole2 c, String s) {
21          console = c;
22          msg = s;
23          t = new Thread(this);
24          t.start();
25      }
26
27      public void run() {
28          console.print(msg);
29      }
30  }
31
32  public class SynchronizedBlock {
33      public static void main(String[] args) {
34          SharedConsole2 target = new SharedConsole2();
35          UserConsoleSync2 ob1 = new UserConsoleSync2(target, "Hello");
36          UserConsoleSync2 ob2 = new UserConsoleSync2(target, "Synchronized")
37          UserConsoleSync2 ob3 = new UserConsoleSync2(target, "World");
38      }
39  }
40
```

```java
class SharedConsole1 {
    synchronized void print(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class UserConsoleSync implements Runnable {
    String msg;
    SharedConsole1 console;
    Thread t;

    public UserConsoleSync(SharedConsole1 c, String s) {
        console = c;
        msg = s;
        t = new Thread(this);
        t.start();
    }
```

```
    public UserConsoleSync(SharedConsole1 c, String s) {
        console = c;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run() {
        console.print(msg);
    }
}

public class SynchronizedMethod {
    public static void main(String[] args) {
        SharedConsole1 target = new SharedConsole1();
        UserConsoleSync ob1 = new UserConsoleSync(target, "Hello");
        UserConsoleSync ob2 = new UserConsoleSync(target, "Synchronized")
        UserConsoleSync ob3 = new UserConsoleSync(target, "World");
    }
}
```

```java
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Lock;

class SharedConsole3 {
    Lock lock = new ReentrantLock();
     void print(String msg) {
         lock.lock();
         System.out.print("[" + msg);
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            System.out.println("Interrupted");
         }
         System.out.println("]");
         lock.unlock();
     }
}

class UserConsoleSync3 implements Runnable {
    String msg;
    SharedConsole3 console;
    Thread t;

```

```
    public UserConsoleSync3(SharedConsole3 c, String s) {
        console = c;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run() {
        console.print(msg);
    }
}

public class SynchronizationLock {
    public static void main(String[] args) {
        SharedConsole3 target = new SharedConsole3();
        UserConsoleSync3 ob1 = new UserConsoleSync3(target, "Hello");
        UserConsoleSync3 ob2 = new UserConsoleSync3(target, "Synchronized");
        UserConsoleSync3 ob3 = new UserConsoleSync3(target, "World");
    }
}
```

Inter Thread Communication

- One way is to use polling
  - a loop that is used to check some condition repeatedly
  - Once the condition is true, appropriate action is taken
- Java includes an elegant inter thread communication mechanism via the **wait()**, **notify()** and **notifyAll()** methods
- These methods are implemented as final methods in Object, so all classes have them
- All three methods can be called only from within a synchronized method

Inter Thread Communication

- ***wait()***
  - tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify()
- ***notify()***
  - wakes up the first thread that called wait() on same object
- ***notifyAll()***
  - wakes up all the threads that called wait() on same object. The highest priority thread will run first
- ***Example****: IncorrectPC.java, CorrectPC.java, PCBlockingQueue.java*

```java
class Q {
    int n;
    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run()  {
        int i = 0;
        while(true) {
            q.put(i++);
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {
```

```
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

public class IncorrectPC {
    public static void main(String[] args) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Press Control-C to stop.");
    }
}
```

```java
class QC {
    int n;
    boolean valueSet = false;
    synchronized int get() {
        while(!valueSet) { // always use while instead of if
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        }
        System.out.println("Got: " + n);
        valueSet = false;
        notifyAll(); // always use notifyAll instead of notify
        return n;
    }
    synchronized void put(int n) {
        while(valueSet) { // always use while instead of if
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notifyAll(); // always use notifyAll instead of notify
```

```
    }
}
class ProducerC implements Runnable {
    QC q;
    ProducerC(QC q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {
                System.out.println("Interrupted");
            }
        }
    }
}
class ConsumerC implements Runnable {
    QC q;
    ConsumerC(QC q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
```

```
                Thread.sleep(1000);
            } catch(InterruptedException e) {
                System.out.println("Interrupted");
            }
        }
    }
}
class ConsumerC implements Runnable {
    QC q;
    ConsumerC(QC q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
public class CorrectPC {
    public static void main(String[] args) {
        QC q = new QC();
        new ProducerC(q);
        new ConsumerC(q);
        new ConsumerC(q);
        System.out.println("Press Control-C to stop.");
    }
}
```

```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

class ProducerB implements Runnable {
    BlockingQueue q;

    ProducerB(BlockingQueue q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;
        try {
            while (true) {
                q.put(i++ + "");
                System.out.println("Put: " + (i-1));
                Thread.sleep(1000);
            }
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

class ConsumerB implements Runnable {
    BlockingQueue q;
```

```
26  class ConsumerB implements Runnable {
27      BlockingQueue q;
28
29      ConsumerB(BlockingQueue q) {
30          this.q = q;
31          new Thread(this, "Consumer").start();
32      }
33
34      public void run() {
35          try {
36              while (true) {
37                  System.out.println("Got: " + q.take());
38              }
39          } catch(InterruptedException e) {
40              System.out.println("Interrupted");
41          }
42      }
43  }
44
45  public class PCBlockingQueue {
46      public static void main(String[] args) {
47          BlockingQueue q = new ArrayBlockingQueue(1);
48          new ProducerB(q);
49          new ConsumerB(q);
50          System.out.println("Press Control-C to stop.");
51      }
52  }
53
```

Suspend, Resume and Stop

- Suspend
  - ***Thread t; t.suspend();***
- Resume
  - ***Thread t; t.resume();***
- Stop
  - ***Thread t; t.stop();***
  - Cannot be resumed later
- suspend and stop can sometimes cause serious system failures
- ***Example****: SuspendResume.java*

SuspendResume.java

```java
// Suspending and resuming a thread the modern way.
class NewThreadSR implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;

    NewThreadSR(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for (int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i+"\n");
                Thread.sleep(200);
                synchronized (this) {
                    while (suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }

    synchronized void mySuspend() {
        suspendFlag = true;
    }

```

```
    synchronized void myResume() {
        suspendFlag = false;
        notify();
    }
}
public class SuspendResume {
    public static void main(String[] args) {
        NewThreadSR ob1 = new NewThreadSR("One");
        NewThreadSR ob2 = new NewThreadSR("Two");
        try {
            Thread.sleep(1000);
            ob1.mySuspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myResume();
            System.out.println("Resuming thread One");
            ob2.mySuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myResume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            Thread.currentThread().join();
            //ob1.t.join();
            //ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Java

Networking

InetAddress

- Java has a class ***java.net.InetAddress*** which abstracts network addresses
- Major methods
  - getLocalHost()
  - getByAddress()
  - getByName()
- ***Example****: HostInfo.java, AddressGenerator.java, Resolver.java*

```java

import java.net.InetAddress;
import java.net.UnknownHostException;

public class HostInfo
{
    public static void main(String args[] )  {
        try {
            InetAddress ipAddress = InetAddress.getLocalHost();
            System.out.println(ipAddress);
        } catch ( UnknownHostException ex ) {
            System.out.println(ex);
        }
    }
}
```

```java
import java.net.InetAddress;
import java.net.UnknownHostException;

public class AddressGenerator
{
    public static void main(String args[] ) {
        byte address[] = new byte[4];
        address[0] = (byte) 192;
        address[1] = (byte) 168;
        address[2] = (byte) 0;
        address[3] = (byte) 63;
        try {
            InetAddress ipAddress = InetAddress.getByAddress(address);
            System.out.println(ipAddress);
        }
        catch (UnknownHostException ex) {
            System.out.println(ex);
        }
    }
}
```

```java
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Resolver
{
    public static void main(String args[] ) {
        try {
            InetAddress ipAddress = InetAddress.getByName("www.google.com");
            System.out.println(ipAddress);
        }
        catch ( UnknownHostException ex ) {
            System.out.println(ex);
        }
    }
}
```

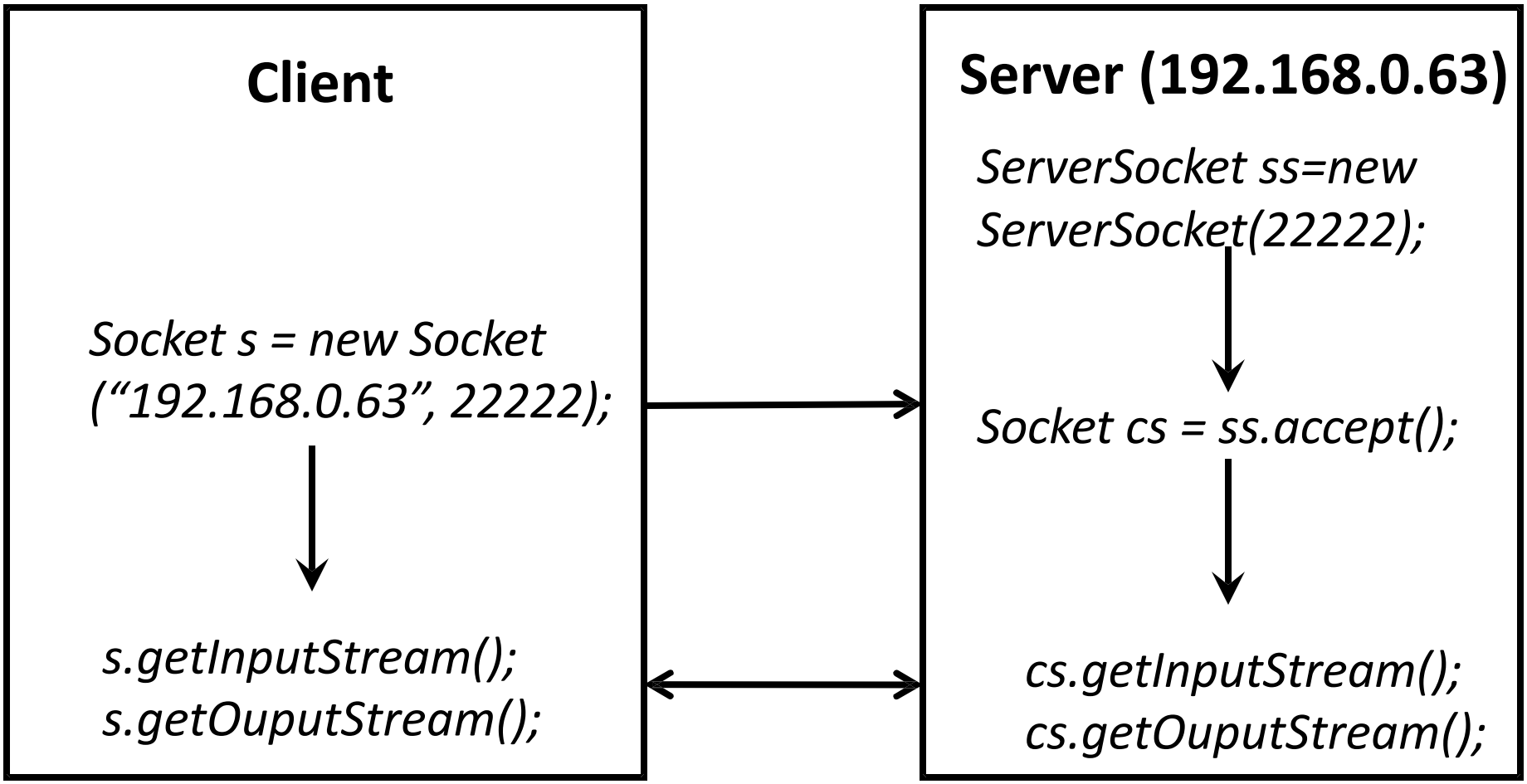
TCP

- TCP stands for Transmission Control Protocol
- TCP is connection-oriented
- It provides reliability
- What is Server and Client?
  - A server is a piece of software which advertises and then provides some service on request.
  - A client is a piece of software (usually on a different machine) which makes use of some service.

TCP Sockets

- Two types of TCP Sockets
- ***ServerSocket***
  - ServerSocket is used by servers so that they can accept incoming connections from client
- ***Socket***
  - Socket is used by clients who wish to establish a connection to a (remote) server

Scenario


Client
Socket s = new Socket
("192.168.0.63", 22222);
s.getInputStream();
s.getOuputStream();
Server (192.168.0.63)
ServerSocket ss=new
ServerSocket(22222);
Socket cs = ss.accept();
cs.getInputStream();
cs.getOuputStream();

TCP Sockets

- ***Example****:*
  - *Server.java*
  - *Client.java*
  - *ReadThread.java*
  - *WriteThread.java*
  - *NetworkUtil.java*
  - *Data.java*

```java
package util;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class NetworkUtil {
    private Socket socket;
    private ObjectOutputStream oos;
    private ObjectInputStream ois;

    public NetworkUtil(String s, int port) {
        try {
            this.socket = new Socket(s, port);
            oos = new ObjectOutputStream(socket.getOutputStream());
            ois = new ObjectInputStream(socket.getInputStream());
        } catch (Exception e) {
            System.out.println("In NetworkUtil : " + e.toString());
        }
    }

    public NetworkUtil(Socket s) {
        try {
            this.socket = s;
            oos = new ObjectOutputStream(socket.getOutputStream());
            ois = new ObjectInputStream(socket.getInputStream());
        } catch (Exception e) {
            System.out.println("In NetworkUtil : " + e.toString());
        }
    }
```

```
    }

    public Object read() {
        Object o = null;
        try {
            o = ois.readObject();
        } catch (Exception e) {
            //System.out.println("Reading Error in network : " + e.toString());
        }
        return o;
    }

    public void write(Object o) {
        try {
            oos.writeObject(o);
        } catch (IOException e) {
            System.out.println("Writing  Error in network : " + e.toString());
        }
    }

    public void closeConnection() {
        try {
            ois.close();
            oos.close();
        } catch (Exception e) {
            System.out.println("Closing Error in network : " + e.toString());
        }
    }
}
```

```java
public class Server {

    private ServerSocket serverSocket;
    public int i = 1;
    public HashMap<String, NetworkUtil> clientMap;

    Server() {
        clientMap = new HashMap<>();
        try {
            serverSocket = new ServerSocket(33333);
            new WriteThreadServer(clientMap, "Server");
            while (true) {
                Socket clientSocket = serverSocket.accept();
                serve(clientSocket);
            }
        } catch (Exception e) {
            System.out.println("Server starts:" + e);
        }
    }

    public void serve(Socket clientSocket) {
        NetworkUtil nc = new NetworkUtil(clientSocket);
        String clientName = (String) nc.read();
        clientMap.put(clientName, nc);
        new ReadThread(nc);
    }

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

```java
import java.util.Scanner;
import util.NetworkUtil;
public class Client {

    public Client(String serverAddress, int serverPort) {
        try {
            System.out.print("Enter name of the client: ");
            Scanner scanner = new Scanner(System.in);
            String clientName = scanner.nextLine();
            NetworkUtil nc = new NetworkUtil(serverAddress, serverPort);
            nc.write(clientName);
            new ReadThread(nc);
            new WriteThreadClient(nc, clientName);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    public static void main(String args[]) {
        String serverAddress = "127.0.0.1";
        int serverPort = 33333;
        Client client = new Client(serverAddress, serverPort);
    }
}
```

```java
import util.NetworkUtil;

public class ReadThread implements Runnable {
    private Thread thr;
    private NetworkUtil nc;

    public ReadThread(NetworkUtil nc) {
        this.nc = nc;
        this.thr = new Thread(this);
        thr.start();
    }

    public void run() {
        try {
            while (true) {
                String s = (String) nc.read();
                if (s != null) {
                    System.out.println(s);
                }
            }
        } catch (Exception e) {
            System.out.println(e);
        } finally {
            nc.closeConnection();
        }
    }
}
```

```java
public class WriteThreadServer implements Runnable {

    private Thread thr;
    String name;
    public HashMap<String, NetworkUtil> clientMap;

    public WriteThreadServer(HashMap<String, NetworkUtil> map, String name) {
        this.clientMap = map;
        this.name = name;
        this.thr = new Thread(this);
        thr.start();
    }

    public void run() {
        try {
            Scanner input = new Scanner(System.in);
            while (true) {
                String s = input.nextLine();
                StringTokenizer st = new StringTokenizer(s);
                String cName = st.nextToken();
                NetworkUtil nc = clientMap.get(cName);
                if (nc != null) {
                    nc.write(name + ":" + s);
                }
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```

```java
public class WriteThreadClient implements Runnable {

    private Thread thr;
    private NetworkUtil nc;
    String name;

    public WriteThreadClient(NetworkUtil nc, String name) {
        this.nc = nc;
        this.name = name;
        this.thr = new Thread(this);
        thr.start();
    }

    public void run() {
        try {
            Scanner input = new Scanner(System.in);
            while (true) {
                String s = input.nextLine();
                nc.write(name + ":" + s);
            }
        } catch (Exception e) {
            System.out.println(e);
        } finally {
            nc.closeConnection();
        }
    }
}

```

```java
import java.io.Serializable;

public class Data implements Serializable {
    private String element;

    public Data(String element) {
        this.element = element;
    }

    public String getElement() {
        return this.element;
    }
}
```

UDP

- UDP stands for User Datagram Protocol
- UDP is not connection-oriented
- It does not provide reliability
- It sends and receives packets known as Datagram

Datagram Packet & Socket

- One type of Packet and one type of Socket.
- ***DatagramPacket***
  - Used to encapsulate Datagram
- ***DatagramSocket***
  - DatagramSocket is used by both server and client to receive DatagramPacket
- ***Example****: DatagramServer.java, DatagramClient.java*

```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
public class DatagramServer implements Runnable {
    DatagramPacket pack;
    DatagramSocket sock;
    DatagramServer() {
        new Thread(this).start();
    }
    public void run() {
        while (true) {
            try {
                send();
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
    public void send() throws Exception {
        byte data[] = "This is a datagram packet".getBytes();
        pack = new DatagramPacket(data, data.length);
        InetAddress add = InetAddress.getByName("127.0.0.1");
        pack.setAddress(add);
        pack.setPort(8000);
        sock = new DatagramSocket();
        sock.send(pack);
        sock.close();
    }
    public static void main(String[] args) {
        new DatagramServer();
    }
}
```

```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;
public class DatagramClient implements Runnable {
    DatagramPacket pack;
    DatagramSocket sock;

    DatagramClient() {
        new Thread(this).start();
    }
    public void run() {
        while (true) {
            try {
                receive();
                Thread.sleep(5000);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
    public void receive() throws Exception {
        byte data[] = new byte[1000];
        pack = new DatagramPacket(data, data.length);
        sock = new DatagramSocket(8000);
        sock.receive(pack);
        System.out.println("Data::" + new String(pack.getData()));
        sock.close();
    }
    public static void main(String[] args) {
        new DatagramClient();
    }
}
```

Java

Collections

Collections

- The java.util package contains one of the Java's most powerful subsystems - The Collections Framework
- Some Interfaces
  - Collection, List, Set, Queue, Deque etc.
- Some Classes
  - ArrayList, LinkedList, Vector, Stack etc.
- Very useful while working with a huge number of objects

Collection Interface

- It is the foundation upon which the Collection framework is built (***interface Collection<E>***)
- It must be implemented by any class that defines a collection
- Some functions

boolean add(E obj)
boolean addAll(Collection c)
void clear()
boolean contains(Object obj)
boolean isEmpty()
int size()
boolean remove(Object obj)
boolean removeAll(Collection c)

List Interface

- ***interface List<E>***
- Some functions

void add(int index, E obj)

boolean addAll(int index, Collection c)

E get(int index)

int indexOf(Object obj)

int lastIndexOf(Object obj)

E remove(int index)

Deque Interface

- ***interface Deque<E>***
- Some functions

void addFirst(E obj)	*void addLast(E obj)*
E getFirst()	*E getLast()*
E peekFirst()	*E peekLast()*
E pollFirst()	*E pollLast()*
E pop()	*void push(E obj)*
E removeFirst()	*E removeLast()*

ArrayList

- It extends the ***AbstractList*** class and implements the ***List*** Interface.
- It is a variable length array of object references and can dynamically increase or decrease in size
- Constructors
  - ArrayList()
  - ArrayList(Collection c)
  - ArrayList(int capacity)
- ***Example****: ArrayListDemo1.java*

```java
class ArrayListDemo1 {
  public static void main(String args[]) {
    // create an array list
    ArrayList al = new ArrayList();
    System.out.println("Initial size of al: " +  al.size());
    // add elements to the array list
    al.add("C");
    al.add("B");
    al.add("D");
    al.add("F");
    al.add(1, "A2");

    System.out.println("Size of al after additions: " +  al.size());

    // display the array list
    System.out.println("Contents of al: " + al);

    // iterate
    for(int i=0;i<al.size();i++) {
      System.out.print(al.get(i)+"  ");
    }
    System.out.println ("");

    System.out.println("Start");
    // print all elements using lambda expression
    al.forEach(e -> System.out.println(e));
    System.out.println("End");
    // Remove elements from the array list
    al.remove("F");
    al.remove(2);

    System.out.println("Size of al after deletions: " + al.size());
    System.out.println("Contents of al: " + al);
  }
}
```

LinkedList

- It extends the ***AbstractSequentialList*** class and implements the ***List***, ***Deque*** and ***Queue*** Interface
- It provides a linked-list data structure
- Constructors
  - LinkedList()
  - LinkedList(Collection c)
- ***Example****: LinkedListDemo.java*

```java
class LinkedListDemo {
  public static void main(String args[]) {
    // create a linked list
    LinkedList ll = new LinkedList();
    // add elements to the linked list
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");

    ll.add(1, "A2");

    System.out.println("Original contents of ll: " + ll);

    // remove elements from the linked list
    ll.remove("F");
    ll.remove(2);

    System.out.println("Contents of ll after deletion: "   + ll);

    // remove first and last elements
    ll.removeFirst();
    ll.removeLast();

    System.out.println("ll after deleting first and last: "  + ll);

    // get and set a value
    Object val = ll.get(2);
    ll.set(2, (String) val + " Changed");

    System.out.println("ll after change: " + ll);
  }
}
```

Java Ln 1, col 21 Sel 0 (1) 838 chars, 37 lines

Arrays

- The ***Arrays*** class provides various methods that are useful when working with arrays
- Some methods such as binarySearch, copyOf, copyOfRange, fill, sort are there
- ***Example****: ArraysDemo.java*

```java
class ArraysDemo {
  public static void main(String args[]) {

    // allocate and initialize array
    int array[] = new int[10];
    for(int i = 0; i < 10; i++)
      array[i] = -3 * i;

    // display, sort, display
    System.out.print("Original contents: ");
    display(array);
    Arrays.sort(array);
    System.out.print("Sorted: ");
    display(array);

    // fill and display
    Arrays.fill(array, 2, 6, -1);
    System.out.print("After fill(): ");
    display(array);

    // sort and display
    Arrays.sort(array);
    System.out.print("After sorting again: ");
    display(array);
    // binary search for -9
    System.out.print("The value -9 is at location ");
    int index =  Arrays.binarySearch(array, -9);
    System.out.println(index);
  }

  static void display(int array[]) {
    for(int i = 0; i < array.length; i++)
      System.out.print(array[i] + " ");
    System.out.println("");
  }
}
```

Vector

- It extends the ***AbstractList*** class and implements the ***List*** Interface
- It implements a dynamic array
- Constructors
  - Vector()
  - Vector(int size)
  - Vector(int size, int incr)
  - Vector(Collection c)
- ***Example****: VectorDemo.java*

```java
class VectorDemo {

  public static void main(String args[]) {
    Vector<Integer> v = new Vector<>();
    System.out.println("Initial size: " + v.size());
    System.out.println("Initial capacity: " +  v.capacity());
    v.addElement(1);
    v.addElement(2);
    v.addElement(3);
    v.addElement(4);
    v.addElement(5);
    System.out.println("First element: " + v.firstElement());
    System.out.println("Last element: " + v.lastElement());

    if(v.contains(3)) System.out.println("Vector contains 3.");
    v.remove(4);

    // iterate
    for(int i=0;i<v.size();i++) {
      Integer a= v.elementAt(i);
      System.out.println(a);
    }

    // enumerate the elements in the vector.
    Enumeration vEnum = v.elements();

    System.out.println("\nElements in vector:");
    while(vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}

```

HashTable

- It stores key-value pairs
- Neither keys nor values can be null
- When using HashTable, you specify an object that is used as a key and the value you want linked to that key
- The key is then hashed and the resulting hash code is used as the index at which the value is stored within the table
- ***Example****: HashTableDemo.java*

```java
class HashTableDemo {
  public static void main(String args[]) {
    Hashtable<String, Double> balance = new Hashtable<>();
    String str;
    double bal;

    balance.put("John Doe", 3434.34);
    balance.put("Tom Smith", 123.22);
    //balance.put("Jane Baker", null); // error
    //balance.put(null, new Double(0)); // error
    balance.put("Jane Baker", 1378.00);
    balance.put("Tod Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Show all balances in hashtable
    Enumeration<String> names= balance.keys();
    while(names.hasMoreElements()) {
        str = names.nextElement();
          System.out.println(str + ": " +  balance.get(str));
    }
    System.out.println();

    String key = "John Doe";
    // Deposit 1,000 into John Doe's account
    bal =  balance.get(key);
    balance.put(key, bal+1000);
    System.out.println(key + "'s new balance: " +  balance.get(key));

    Set set = balance.keySet(); // get set view of keys
    Iterator<String> itr = set.iterator();     // get iterator
    while(itr.hasNext()) {
      str = itr.next();
      System.out.println(str + ": " + balance.get(str));
    }
  }
}
```

HashMap

- It also stores key-value pairs like ***HashTable***
- Differences:

	HashMap	HashTable
Synchronized	No	Yes
Thread-Safe	No	Yes
Keys and values	One null key, any null values	Not permit null keys and values
Performance	Fast	Slow in comparison
Superclass	AbstractMap	Dictionary

- Use ***ConcurrentHashMap*** for multi-threading
- ***Example****: HashMapDemo.java*

```java
class HashMapDemo {
    public static void main(String args[]) {
        HashMap<String, Double> balance = new HashMap<>();
        // ConcurrentHashMap balance<String, Double> = new ConcurrentHashMap<>(); // for multi-threading
        String str;
        double bal;
        balance.put("John Doe", 3434.34);
        balance.put("Tom Smith", 123.22);
        balance.put("Jane Baker", null);
        balance.put("Tod Hall", 99.22);
        balance.put("Ralph Smith", -19.08);
        balance.put(null, 0.0);

        // show all balances in hashtable
        Set set = balance.keySet(); // get set view of keys
        // get iterator
        Iterator<String> itr = set.iterator();
        while(itr.hasNext()) {
            str = itr.next();
            System.out.println(str + ": " + balance.get(str));
        }
        System.out.println();

        String key = "John Doe";
        // Deposit 1,000 into John Doe's account
        bal = balance.get(key);
        balance.put(key, bal+1000);
        System.out.println(key + "'s new balance: " +  balance.get(key));

        itr = set.iterator();
        while(itr.hasNext()) {
            str = itr.next();
            System.out.println(str + ": " + balance.get(str));
        }
    }
}
```

Custom Comparator

- Required to sort a collection/array of custom objects
- Must implement the ***Comparable*** interface
- Must implement the following method

  public int compareTo(Object o) {

  }
- ***Example****: ComparatorDemo.java*

```java
class TestClass implements Comparable {
    String  name;
    TestClass(String name) {
        this.name=name;
    }
    @Override
    public int compareTo(Object o) {
        TestClass m = (TestClass) o;
        return this.name.compareTo(m.name);
    }
}
class ComparatorDemo {
    public static void printArrayList(ArrayList<TestClass> al) {
        for(int i=0;i<al.size();i++) {
            System.out.print(al.get(i).name+"  ");
        }
        System.out.println ();
    }
    public static void printArray(TestClass [] ia) {
        for(int i=0;i<ia.length;i++) {
            System.out.print(ia[i].name+"  ");
        }
        System.out.println ();
    }
```

```
    public static void main(String args[]) {
        ArrayList<TestClass> al = new ArrayList<>();

        al.add(new TestClass("C"));
        al.add(new TestClass("A"));
        al.add(new TestClass("E"));
        al.add(new TestClass("B"));
        al.add(new TestClass("D"));
        al.add(new TestClass("F"));

        TestClass ia[]=new TestClass[al.size()];
        al.toArray(ia);

        System.out.println("Collection:");
        printArrayList(al);
        Collections.sort(al);
        printArrayList(al);

        System.out.println("Array:");
        printArray(ia);
        Arrays.sort(ia);
        System.out.println("After Sorting:");
        printArray(ia);
    }
}
```

Java

I/O

File

- Long-term storage of large amounts of data
- Persistent data exists after termination of program
- Files stored on secondary storage devices
  - Magnetic disks
  - Optical disks
  - Magnetic tapes
- Sequential and random access files

File Class

- Provides useful information about a file or directory
- Does not open files or process files
- To obtain or manipulate path, time, date, permissions etc
- Constructor
  - File(String directoryPath)
  - File(String directoryPath, String fileName)
  - File(File dirObj, String fileName)
- ***Example****: FileDemo.java*

```java
import java.io.File;

class FileDemo
{

  static void p(String s)
  {
    System.out.println(s);
  }

  public static void main(String args[])
  {
    File f1 = new File("dir/sample.txt");
    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "not a normal file");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

Directory Class

- Directories are also files
- Contains list of files and directories
- For Directory isDirectory() returns true

  String[] list()

  – returns an array of strings that gives the files and directories contained

  File[] listFiles()

  – Returns array of File objects
- ***Example**: DirectoryDemo.java*

```java
import java.io.File;

class DirectoryDemo
{
  public static void main(String args[])
  {
    String dirname = "dir";
    File f1 = new File(dirname);

    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}

```

Stream Classes

- Java views a File as a stream of bytes.
  - File ends with end-of-file marker or a specific byte number
  - File as a stream of bytes associated with an object.
  - Java also associates streams with devices
    - System.in, System.out, and System.err
  - Streams can be redirected
- Stream is an abstraction that either produces or consumes information

Stream Classes

- Java's stream-based I/O is built upon four abstract classes.
  - InputStream, OutputStream (for byte streams)
  - Reader, Writer (for character streams)
- They form separate hierarchies
- Use the character stream classes when working with characters or strings
- Use the byte stream classes when working with bytes or other binary objects

Byte Stream Classes

- Byte-Stream classes are topped by ***InputStream*** and ***OutputStream*** classes
- ***InputStream*** is an abstract class that defines Java's model of streaming byte input.

  int available() *void close()* *int read()*

  int read(byte buff[]) *int read(byte buff[], int off, int num)*
- ***OutputStream*** is an abstract class that defines Java's model of streaming byte output.

  void flush() *void close()* *void write(int b)*

  void write(byte buff[]) *void write(byte buff[], int off, int num)*

FileInputStream

- ***FileInputStream*** class creates an ***InputStream*** that you can use to read bytes from a file
- Constructors
  - FileInputStream(String filePath)
  - FileInputStream(File fileObj)
- ***Example****: FileInputStreamDemo.java*

```java

class FileInputStreamDemo {
    public static void main(String args[]) throws Exception {
        int size;
        InputStream f = new FileInputStream("TestFile.java");
        System.out.println("Total Available Bytes: " + (size = f.available()));
        int n = size / 40;
        System.out.println("First " + n + " bytes of the file one read() at a time");
        for (int i = 0; i < n; i++) {
            System.out.print((char) f.read());
        }
        System.out.println("\nStill Available: " + f.available());
        System.out.println("Reading the next " + n + " with one read(b[])");
        byte b[] = new byte[n];
        if (f.read(b) != n) {
            System.err.println("couldn't read " + n + " bytes.");
        }
        System.out.println(new String(b, 0, n));
        System.out.println("\nStill Available: " + (size = f.available()));
        System.out.println("Skipping half of remaining bytes with skip()");
        f.skip(size / 2);
        System.out.println("Still Available: " + f.available());
        System.out.println("Reading " + n / 2 + " into the end of array");
        if (f.read(b, n / 2, n / 2) != n / 2) {
            System.err.println("couldn't read " + n / 2 + " bytes.");
        }
        System.out.println(new String(b, 0, b.length));
        System.out.println("\nStill Available: " + f.available());
    }
}

```

FileOutputStream

- ***FileOutputStream*** class creates an ***OutputStream*** that you can use to write bytes to a file
- Constructors
  - FileOutputStream(String filePath)
  - FileOutputStream(File fileObj)
  - FileOutputStream(String path, boolean append)
  - FileOutputStream(File obj, boolean append)
- ***Example****: FileOutputStreamDemo.java, FileCopyDemo.java*

```java

class FileOutputStreamDemo
{
  public static void main(String args[]) throws Exception
    {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();
    OutputStream f0 = new FileOutputStream("file1.txt");

    for (int i=0; i < buf.length; i+=2)
    {
      f0.write(buf[i]);
    }
    f0.close();

    OutputStream f1 = new FileOutputStream("file2.txt");
    f1.write(buf);
    f1.close();

    OutputStream f2 = new FileOutputStream("file3.txt");
    f2.write(buf,buf.length-buf.length/4,buf.length/4);
    f2.close();
  }
}
```

```java

class FileCopyDemo
{
  public static void main(String args[]) throws Exception
  {
    String source="src.flv";
    String destination="copy.flv";
    InputStream in =new FileInputStream(source);
    OutputStream out=new FileOutputStream(destination);
    while(true)
    {
      int c=in.read();
      if(c==-1) break;
      out.write(c);
    }
    in.close();
    out.close();
  }
}

```

Character Streams

- Character Stream classes are topped by ***Reader*** and ***Writer*** class
- ***Reader*** is an abstract class that defines Java's model of streaming character input

  void close() *int read()* *int read(char buff[])*

  int read(char buff[], int off, int num)
- ***Writer*** is an abstract class that defines Java's model of streaming character output

  void flush() void close() void write(int ch)

  void write(char buff[]) *void write(char buff[], int off, int num)*

  void write(String s) *void write(String s, int off, int num)*

FileReader

- ***FileReader*** class creates a ***Reader*** that you can use to read the contents of a file
- Constructors
  - FileReader(String filePath)
  - FileReader(File fileObj)
- ***Example****: FileReaderDemo.java*

```java
import java.io.*;

class FileReaderDemo {
  public static void main(String args[]) throws Exception
  {
    File f=new File("TestFile.java");
    FileReader fr = new FileReader(f);
    char data[]=new char[(int)f.length()];
    fr.read(data);
    System.out.println(new String(data));
    fr.close();
  }
}
```

FileWriter

- ***FileWriter*** class creates a ***Writer*** that you can use to write to a file
- Constructors
  - FileWriter(String filePath)
  - FileWriter(File fileObj)
  - FileWriter(String path, boolean append)
  - FileWriter(File obj, boolean append)
- ***Example****: FileWriterDemo.java*

```java

class FileWriterDemo {
  public static void main(String args[]) throws Exception {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    char buffer[] = new char[source.length()];
    source.getChars(0, source.length(), buffer, 0);

    FileWriter f0 = new FileWriter("file1.txt");
    for (int i=0; i < buffer.length; i += 2) {
      f0.write(buffer[i]);
    }
    f0.close();

    FileWriter f1 = new FileWriter("file2.txt");
    f1.write(buffer);
    f1.close();

    FileWriter f2 = new FileWriter("file3.txt");

    f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
    f2.close();
  }
}

```

BufferedReader

- ***BufferedReader*** is a ***Reader*** that buffers input
- It improves performance by reducing the number of times data us actually physically read from the input stream
- Constructors
  - BufferedReader(Reader reader)
  - BufferedReader(Reader reader, int buffSize)
- ***Example****: BufferedReaderDemo.java*

```java

class BufferedReaderDemo {
  public static void main(String args[]) throws Exception
  {
    FileReader fr = new FileReader("TestFile.java");
    BufferedReader br=new BufferedReader(fr);
    while(true)
    {
      String s=br.readLine();
      if(s==null) break;
      System.out.println (s);
    }
    br.close();
    fr.close();

  }
}
```

BufferedWriter

- ***BufferedWriter*** is a ***Writer*** that buffers output
- It improves performance by reducing the number of times data actually physically written to the output stream
- Constructors
  - BufferedWriter(Writer writer)
  - BufferedWriter(Writer writer, int buffSize)
- ***Example****: BufferedWriterDemo.java*

```java

class BufferedWriterDemo {
  public static void main(String args[]) throws Exception
  {
    FileReader fr = new FileReader("TestFile.java");
    BufferedReader br=new BufferedReader(fr);
    FileWriter fw = new FileWriter("TestFileCopy.java");
    BufferedWriter bw=new BufferedWriter(fw);
    while(true)
    {
      String s=br.readLine();
      if(s==null) break;
      s=s+"\n";
      bw.write(s);
    }
    bw.close();
    fw.close();
    br.close();
    fr.close();

  }
}
```

Serialization

- Serialization is the process of writing the state of an object to a byte stream
  - This is useful when you want to save the state of your program to a persistent storage such as file
  - Later these objects can be restored by using the process of deserialization
- Serialization can be achieved by implementing ***Serializable*** interface

Object(Input/Output)Stream

- ***ObjectInputStream*** class extends the ***InputStream*** class
- It is responsible for reading objects from a stream
- ***ObjectOutputStream*** class extends the ***OutputStream*** class
- It is responsible for writing objects to a stream
- ***Example****: ObjectSerializationDemo.java*

```java
class MyClass implements Serializable {
  String s;
  int i;
  double d;

  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }
  public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
  }
}



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

    // Object serialization
    try {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);
      FileOutputStream fos = new FileOutputStream("serial");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```java
public class ObjectSerializationDemo {
  public static void main(String args[]) {
    // Object serialization
    try {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);
      FileOutputStream fos = new FileOutputStream("serial");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(object1);
      oos.flush();
      oos.close();
    }
    catch(Exception e) {
      System.out.println("Exception during serialization: " + e);
      System.exit(0);
    }

    // Object deserialization
    try {
      MyClass object2;
      FileInputStream fis = new FileInputStream("serial");
      ObjectInputStream ois = new ObjectInputStream(fis);
      object2 = (MyClass)ois.readObject();
      ois.close();
      System.out.println("object2: " + object2);
    }
    catch(Exception e) {
      System.out.println("Exception during deserialization: " + e);
      System.exit(0);
    }
  }
}
```

Data(Input/Output)Stream

- ***DataInputStream*** & ***DataOutputStream*** enable to write or read primitive data to or from a stream
- They implement the ***DataOutput*** & ***DataInput*** interfaces respectively
- Constructors
  - DataOutputStream(OutputStream os)
  - DataInputStream(InputStream is)
- ***Example****: DataIODemo.java*

```java

class DataIODemo
{
  public static void main(String args[]) throws Exception
  {
    FileOutputStream fos =new FileOutputStream("Test.dat");
    DataOutputStream dos=new DataOutputStream(fos);
    dos.writeDouble(98.6);
    dos.writeInt(1000);
    dos.writeBoolean(true);
    dos.close();
    fos.close();

    FileInputStream fis =new FileInputStream("Test.dat");
    DataInputStream dis=new DataInputStream(fis);
    double d=dis.readDouble();
    int i=dis.readInt();
    boolean b=dis.readBoolean();

    System.out.println (d);
     System.out.println (i);
     System.out.println (b);

     dis.close();
     fis.close();


  }
}

```

Console

- It is used to read and write to the console
- It supplies no constructor. A Console object is obtained by calling ***System.console()***
- Important Methods
  - printf
  - readLine
  - readPassword
- ***Example****: ConsoleDemo.java*

```java
class ConsoleDemo
{
  public static void main(String args[]) throws Exception
  {
    String userName,password;
    Console con;
    con=System.console();
    System.out.println (con);
    if(con==null) return;
    userName=con.readLine("Enter UserName:");
    char p[]=con.readPassword("Enter Password:");
    password=new String(p);
    con.printf("UserName:%s\n",userName);
    con.printf("Password:%s\n",password);
  }
}
```

RandomAccessFile

- This class support both reading and writing to a random access file
- A random access file behaves like a large array of bytes stored in the file system
- The file pointer can be read by the **getFilePointer** method and set by the **seek** method
- ***Example****: RandomAccessFileDemo.java*

```java
import java.io.RandomAccessFile;

public class RandomAccessFileDemo {
    public static void main(String[] args) throws Exception {
        RandomAccessFile file = new RandomAccessFile("random.txt", "rw");
        // write something in the file
        file.writeUTF("Hello World\n");
        // set the file pointer at 0 position
        file.seek(0);
        // print the line
        System.out.println(file.readLine());
        // set the file pointer at 0 position
        file.seek(0);
        // write something in the file
        file.writeUTF("This is an example \n Hello World");
        // set the file pointer at 0 position
        file.seek(0);
        // print the line
        System.out.println(file.readLine());
        file.close();
    }
}

```