Programming in Java

Features of Java

1. Simple, Small and Familiar
2. Compiled and Interpreted
3. Object Oriented
4. Platform Independent and portable
5. Robust and Secure
6. Distributed / Network Oriented
7. Multithreaded and Interactive
8. High Performance
9. Dynamic

Simple, Small and Familiar

Similar to C/C++ in syntax

But eliminates several complexities of

- No operator overloading
- No direct pointer manipulation or pointer arithmetic
- No multiple inheritance
- No malloc() and free() – handles memory automatically

Compiled and Interpreted

Java works in two stages

Java compiler translate the source code into byte code.

Java interpreter converts the byte code into machine level representation.

Byte Code:

-A highly optimized set of instructions to be executed by the java runtime system, known as java virtual machine (JVM).

-Not executable code.

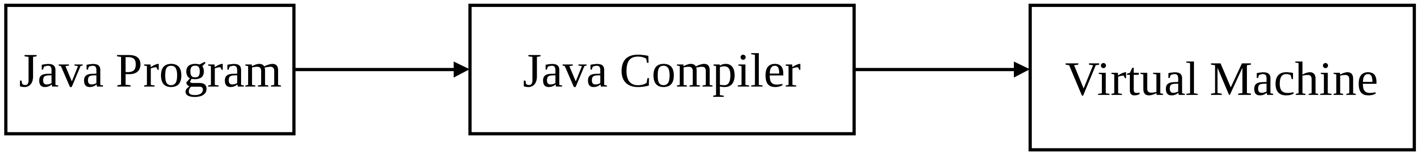
Java Virtual Machine (JVM):

- Need to be implemented for each platform.
- Although the details vary from machine to machine, all JVM understand the same byte code.

Java Virtual Machine

✓ Java compiler produces an intermediate code known as byte code for a machine, known as JVM.

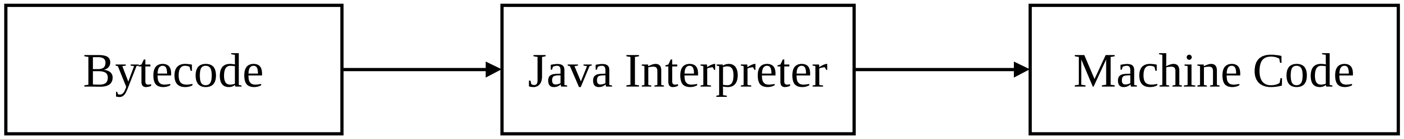
✓ It exists only inside the computer memory.


Java Program

Java Compiler

Virtual Machine

✓ Machine code is generated by the java interpreter by acting as an intermediary between the virtual machine and real machine.


Bytecode

Java Interpreter

Machine Code

Object Oriented

- Fundamentally based on OOP
  - Classes and Objects
  - Efficient re-use of packages such that the programmer only cares about the interface and not the implementation
  - The object model in java is simple and easy to extend.

Platform Independent and Portable

“Write-Once Run-Anywhere”

Changes in system resources will not force any change in the program.

The Java Virtual Machine (JVM) hides the complexity of working on a particular platform

Convert byte code into machine level representation.

Robust and Secure

Designed with the intention of being secure

- No pointer arithmetic or memory management!
- Strict compile time and run time checking of data type.
- Exception handling
- It verifies all memory access
- Ensure that no viruses are communicated with an applet.

Distributed and Network Oriented

- Java grew up in the days of the Internet
  - Inherently network friendly
  - Original release of Java came with Networking libraries
  - Newer releases contain even more for handling distributed applications
    - RMI, Transactions

Multithreaded and Interactive

Handles multiple tasks simultaneously.

Java runtime system contains tools to support multiprocess synchronization and construct smoothly running interactive systems.

High Performance

Java performance is slower than C

Provisions are added to reduce overhead at runtime.

Incorporation of multithreading enhance the overall execution speed.

Just-in-Time (JIT) can compile the byte code into machine code.

- Can sometimes be even faster than compiled C code!

Dynamic

Capable of dynamically linking new class libraries, methods and objects.

Java can use efficient functions available in C/C++.

Installing new version of library automatically updates all programs

Why portable and Secure?

- The output of java compiler is not executable code.

Once JVM exists for a platform, any java program can run on it.

The execution of byte code by the JVM makes java programs portable.

Java program is confined within the java execution environment and cannot access the other part of the computer.

The Primitive Types

- There are exactly eight primitive data types in Java
- Four of them represent whole valued signed numbers:
  - byte, short, int, long
- Two of them represent floating point numbers:
  - float, double
- One of them represents characters:
  - char
- And one of them represents boolean values:
  - boolean

Numeric Primitive Types

- ✓ The difference between the various numeric primitive types is their size, and therefore the values they can store:

Type	**Storage**	**Min Value**	**Max Value**
byte	**8 bits**	**-128**	**127**
short	**16 bits**	**-32,768**	**32,767**
int	**32 bits**	**-2,147,483,648**	**2,147,483,647**
long	**64 bits**	**< -9 x 10^{18}**	**> 9 x 10^{18}**
float	**32 bits**	**+/- 3.4 x 10^{38} with 7 significant digits**	
double	**64 bits**	**+/- 1.7 x 10^{308} with 15 significant digits**	

Character Primitive Type

✓ It uses unicode to represent character.

✓ The char type is unsigned 16 bit values ranging from 0 to 65536.

✓ ASCII still ranges from 0 to 127.

Example:

```
class test
{   public static void main (String args[])
    {
            char ch1, ch2;
            ch1=88;
            ch2=‘Y’;
            System.out.println (“ch1 and ch2: “ + ch1+” “+ch2);
    }
}
```

Output: ch1 and ch2: X Y

Booleans

- ✓ Size is 1 bit – two value: true and false.
- ✓ This type is returned by all relational operators.
- ✓ <u>Example:</u>

```
boolean b;
b= true;
```

1. System.out.println(“b is “+b);
2. System.out.println(“10>9 is “ +(10>9));

<u>Output:</u>

```
b is true
10>9 is true
```

Literals

- ✓ Integer Literals

  1. base 10 – 1,2,43 etc.
  2. base 8 – octal values are denoted in java by a leading 0.
  3. base 16 – hexadecimal values are denoted by leading 0x or 0X.

- Any whole number is by default integer (32 bits).
- To specify a long literal, the number should appended with an upper- or lowercase L.

Literals

- ✓ Floating point Literals

  1. Standard Notation – 3.14159, 0.6667, 2.0 etc.

  2. Scientific Notation – 6.022E23, 2e+100.

- Floating point literals are by default of type double.
- To specify a float literal, we must append an F or f to the constant.
- ✓ Boolean Literals
- Two values – true and false.
- True is not equal 1 and false is not equal to 0.
- They can be assigned to variable declared as boolean.

Variables

- ✓Variable is a name for a location in memory.
- ✓Declaring a variable:

  type identifier [=value][,identifier [=value]….];
- ✓The initialization expression must result in a value of the same or compatible type as that specified for the variable.
- ✓When a variable is not initialized, the value of that variable is undefined.

Type Conversion in Primitive Types

- Widening
  - When one type of data is assigned to another type of variable, an automatic(implicit) type conversion will take place if,the two types are compatible and the destination type is larger than the source type.
  - Example: `int i; byte b; i=b; //valid conversion`
- Narrowing
  - Converting a value of larger range to value of smaller range.
  - Example: `int i; byte b; b=i; //invalid`

Type Casting in Primitive Types

- A cast is an explicit type conversion of incompatible types.
- General form : `(target-type)value`

Target-type is the desired type to convert the value to.

- A different type of conversion called truncation will occur when a floating point value is assigned to an integer type.
- If the size of the whole number is too large to fit into the target int type, then that value will be reduced modulo the target type’s range.

Type Conversion in Expressions

- In addition to assignments, type conversions may occur in expressions.
- Promotion rules that apply to expressions:
  - all byte and short values are promoted to int
  - If one operand is long,the whole expression is promoted to long
  - If one operand is a float ,the whole expression is promoted to float
  - If one operand is double ,the whole expression is promoted to double

Operators in Java

Classified into four groups:

1. Arithmetic Operator
2. Bitwise Operator
3. Relational Operator
4. Logical Operator

Arithmetic Operators

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%
Increment	++
Addition Assignment	+=
Subtraction Assignment	-=
Multiplication Assignment	*=
Division Assignment	/=
Modulus Assignment	%=
Decrement	--

Assignment Operators

- ✓ There are many assignment operators, including the following:

Operator	Example	Equivalent To
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Bitwise Operator

~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Shift Right
>>>	Shift Right zero fill
<<	Shift left
& =	Bitwise AND Assignment
|=	Bitwise OR Assignment
^=	Bitwise XOR Assignment
>>=	Shift Right Assignment
>>>=	Shift Right zero fill Assignment
<<=	Shift Left Assignment

Relational operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
= =	equal to
!=	not equal to

- The outcome of these operations is a boolean value.
- = = , != can be applied to any type in java.
- Only numeric types are compared using ordering operator.

Boolean Logical Operator

&	Logical AND
|	Logical OR
^	Logical XOR
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND Assignment
|=	OR Assignment
^ =	XOR Assignment
= =	Equal to
!=	Not equal to
?:	Ternary if-then-else

Short Circuit Logical Operators

- ✓ || Short circuit logical OR
- ✓ && Short circuit logical AND
- ✓ If the left operand is sufficient to determine the result, the right operand is not evaluated

```
if (demon !=0 && num / demon >10)
```

This type of processing must be used carefully

The Conditional Operator

- ✓ The conditional operator is similar to an if-else statement, except that it forms an expression that returns a value
- ✓ For example:
- ✓
  ```
  larger = ((num1 > num2) ? num1 : num2);
  ```
- ✓ If num1 is greater that num2, then num1 is assigned to larger; otherwise, num2 is assigned to larger
- ✓ The conditional operator is *ternary* because it requires three operands

Decision Making and Branching

✓ When a program breaks the sequential flow and jumps to another part of the code, it is known as branching. When branching is done on a condition it is known as conditional branching.

✓ Three decision making statements:

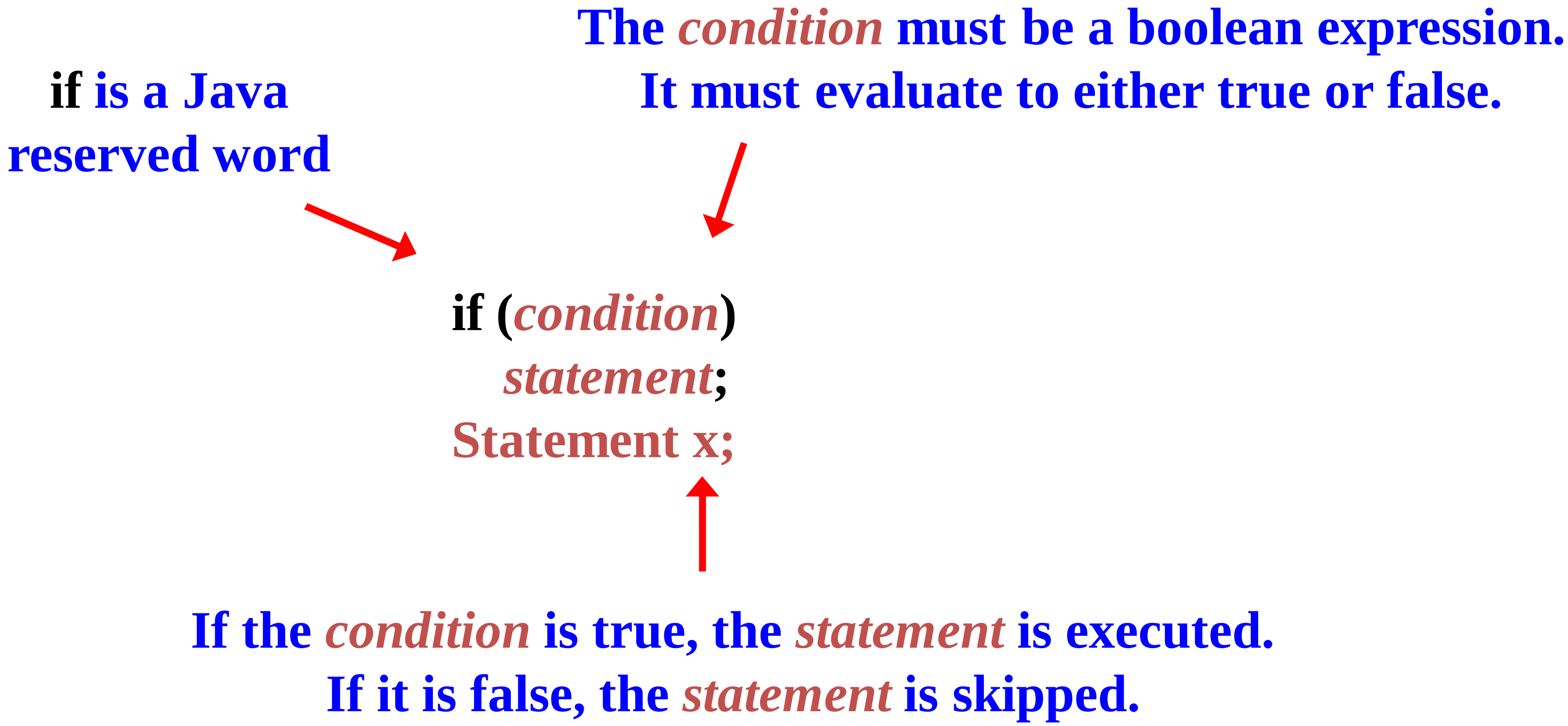
1. if statement

2. switch statement

3. conditional operator statement

The if Statement

- ✓ The *if statement* has the following syntax:



The *condition* must be a boolean expression. It must evaluate to either true or false.

if is a Java reserved word

```
if (condition)
   statement;
Statement x;
```

If the *condition* is true, the *statement* is executed.
If it is false, the *statement* is skipped.

The if-else Statement

✓ An *else clause* can be added to an if statement to make an *if-else statement*

```
if ( condition )
  statement1;
else
  statement2;
Statement x;
```

✓If the `condition` is true, `statement1` is executed; if the condition is false, `statement2` is executed

✓One or the other will be executed, but not both

Nested if….Else Statements

✓ The if..else statement can be contained in another if or else statement.

```
if (test condition1)
{
        if (test condition2)
                statement-1;
        else
                statement-2;
}
else
        statement-3;

statement-x;
```

The switch Statement

- ✓ The *switch statement* provides another means to decide which statement to execute next

- ✓ The switch statement evaluates an expression, then attempts to match the result to one of several possible *cases*

- ✓ The expression of a switch statement must result in an *integral type*, meaning an int or a char

- ✓ Each case contains a value and a list of statements

- ✓ The flow of control transfers to statement associated with the first value that matches

The switch Statement

- ✓ The general syntax of a switch statement is:

switch **and** **case** **are** **reserved** **words**

```
switch (expression) {
   case value1:
      statement-list1
   case value2:
      statement-list2
   case value3:
      statement-list3
}
```

If *expression* matches *value2*, control jumps from here

The switch Statement

- ✓ Often a *break statement* is used as the last statement in each case's statement list
- ✓ A break statement causes control to transfer to the end of the switch statement
- ✓ If a break statement is not used, the flow of control will continue into the next case
- ✓ Sometimes this can be appropriate, but usually we want to execute only the statements associated with one case

The switch Statement

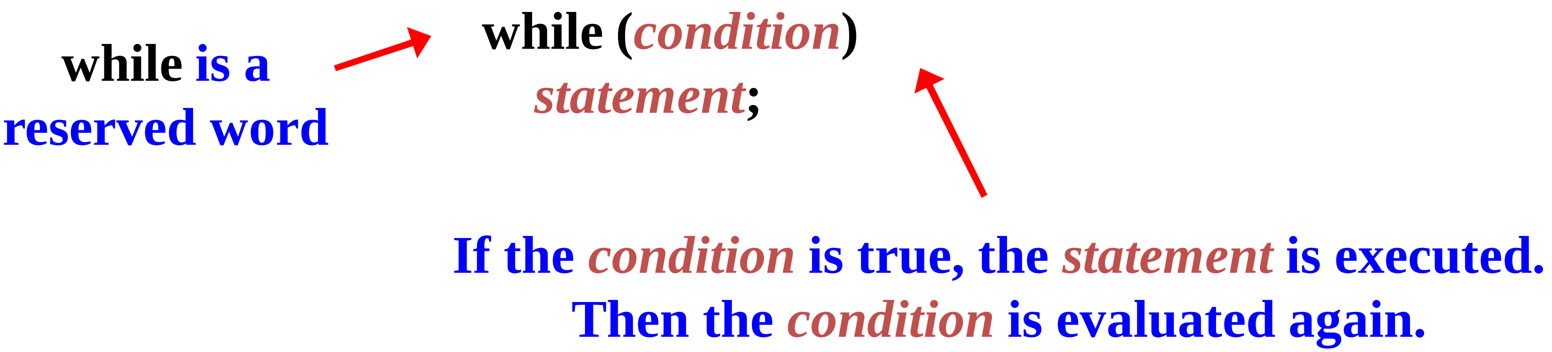
- ✓ A switch statement can have an optional *default case*
- ✓ The default case has no associated value and simply uses the reserved word default
- ✓ If the default case is present, control will transfer to it if no other case value matches
- ✓ If there is no default case, and no other value matches, control falls through to the statement after the switch

Repetition Statements

- ✓ *Repetition statements* allow us to execute a statement multiple times
- ✓ Often they are referred to as *loops*
- ✓ Like conditional statements, they are controlled by boolean expressions
- ✓ Java has three kinds of repetition statements:
  - ✓ the *while loop*
  - ✓ the *do loop*
  - ✓ the *for loop*
- ✓ The programmer should choose the right kind of loop for the situation

The while Statement

- ✓ The *while statement* has the following syntax:



while (*condition*)
***statement*;**

while is a reserved word

If the *condition* is true, the *statement* is executed. Then the *condition* is evaluated again.

The *statement* is executed repeatedly until the *condition* becomes false.

while Loop Example

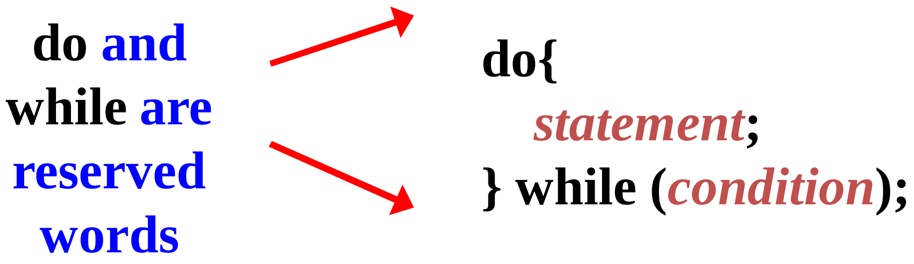
```
final int LIMIT = 5;
int count = 1;

while (count <= LIMIT) {

    System.out.println(count);
    count += 1;
}
```

Output:

```
1
2
3
4
5
```

The do Statement

- ✓ The *do statement* has the following syntax:



do and **while** are reserved words

```
do{
   statement;
} while (condition);
```

The *statement* is executed once initially, and then the *condition* is evaluated

The *statement* is executed repeatedly until the *condition* becomes false

do-while Example

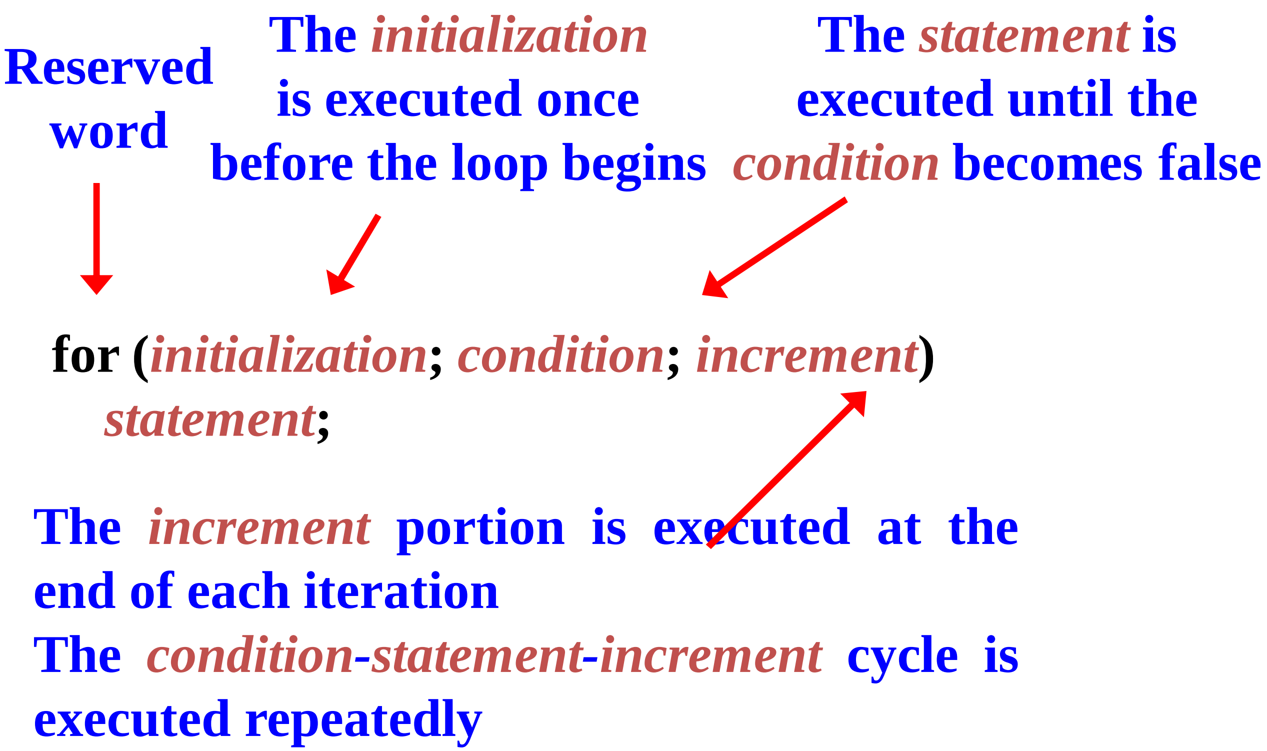
```
final int LIMIT = 5;
int count = 1;


do {
    System.out.println(count);
    count += 1;
} while (count <= LIMIT);
```

Output:

```
1
2
3
4
5
```

The for Statement

✓ The *for statement* has the following syntax:



Reserved word

The *initialization* is executed once before the loop begins

The *statement* is executed until the *condition* becomes false

for (*initialization*; *condition*; *increment*)
**   *statement*;**

The *increment* portion is executed at the end of each iteration

The *condition*-*statement*-*increment* cycle is executed repeatedly

The for Statement

- A for loop is functionally equivalent to the following while loop structure:

```
initialization;
while (condition) {
   statement;
   increment;
}
```

The for Statement

- ✓ Like a while loop, the condition of a for statement is tested prior to executing the loop body
- ✓ Therefore, the body of a for loop will execute zero or more times
- ✓ It is well suited for executing a loop a specific number of times that can be determined in advance

Import Statement

- ✓ A java package is a collection of related classes.
- ✓ In order to access the available classes in the package, the program must specify the complete dot seperated package path.
- ✓ The general format:

  import package-level1.[package-level2.]classname|*
- ✓ Two form of import statement:

  1. import package.class;

  2. import package.*;

Example:

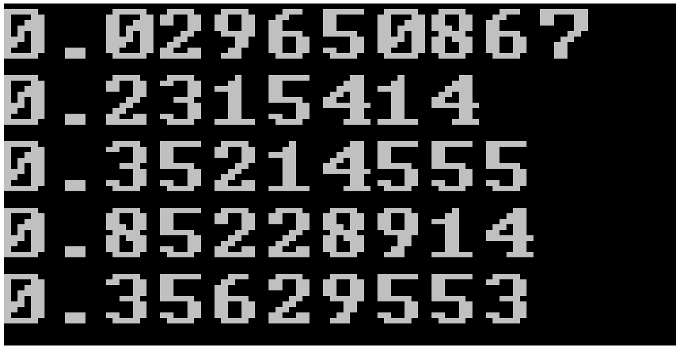
import java.util.Scanner;

import java.util.*;

Example

```
import java.util.Random;

class random
{
        public static void main(String args[])
        {
                Random r = new Random();
                int i;
                float v;

                for(i=0;i<5;i++)
                {
                        v=r.nextFloat();
                        System.out.println(v);
                }
        }
}
```

Output:



```
0.029650867
0.2315414
0.35214555
0.85228914
0.35629553
```

✓java.lang in automatically imported with every java program.

✓System.out.println() belongs to java.lang.

One-Dimensional Array

- ✓ Creating an array is a two steps process:

  1. type var_name[];

  2. var_name = new type [size];

- ✓ Example: int month_days [];

  month_days = new int [12];

  - first line declares *month_days* as an array variable, no array actually exists.

  - Actual, physical array of integers, is allocated using **new** and assign it to *month_days*.

  -The elements of array are automatically initialized to 0.

- ✓ int month_days [] = new int[12];

One-Dimensional Array

- ✓ Once array is created, a specific element in the array can be accessed by specifying it’s index within square brackets.

  Example: month_days[0]=31;

  month_days[1]=28;

- ✓ Arrays can be initialized when they are declared.

  Example:

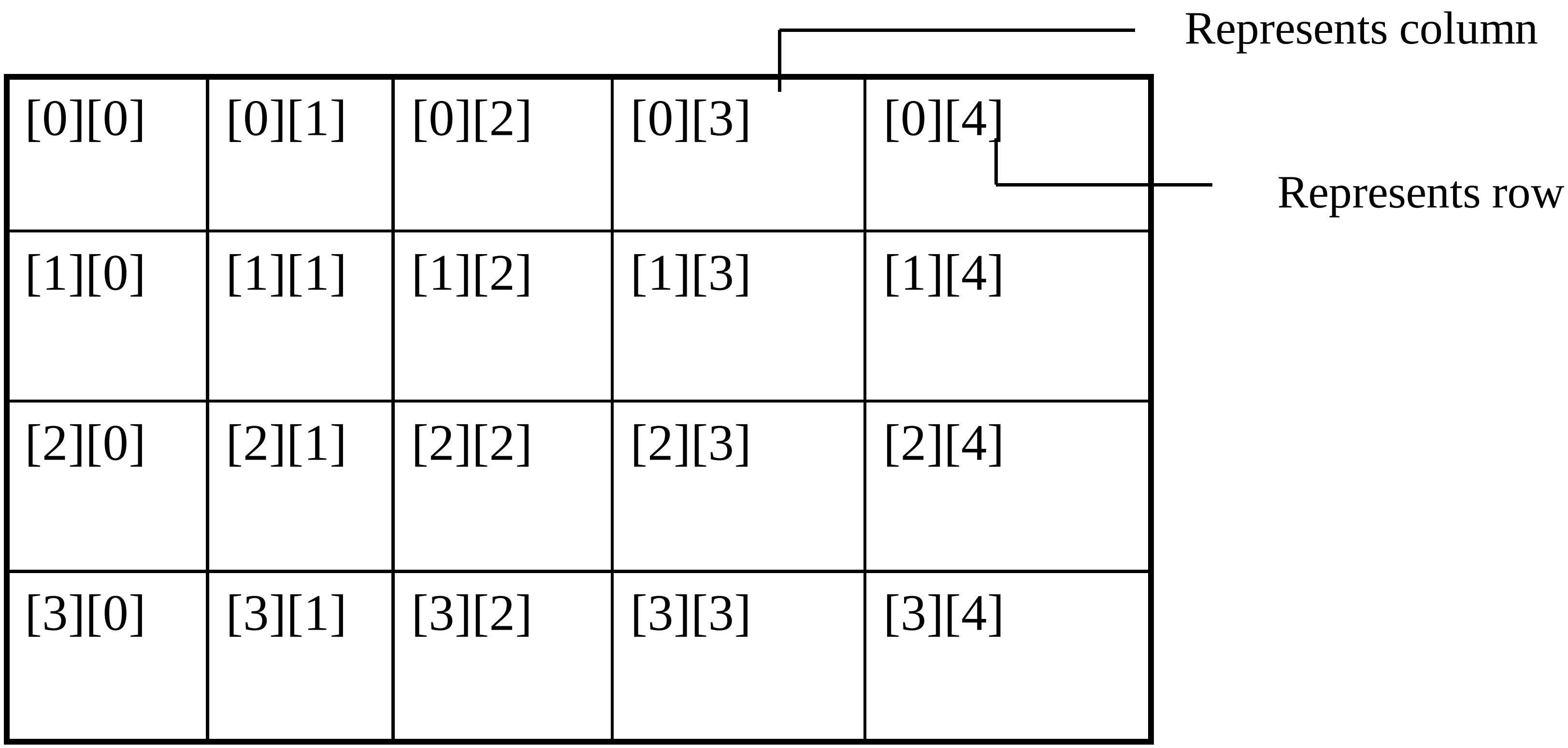
  int month_days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

Example

```
import java.util.Scanner;

class array_avg
{
        public static void main(String args[])
        {
                int i,sum=0;
                float avg;

                int a[] = new int[5];

                Scanner test=new Scanner(System.in);

                System.out.println("Enter the input:");

                for(i=0;i<5;i++)
                {
                a[i]=test.nextInt();
                sum=sum+a[i];
                }
                avg=sum/5;

                System.out.println("The average value is: " + avg);

        }
}
```

Multi-Dimensional Arrays

- ✓ A two-dimensional array can be declared as
  int twoD [] [] = new int[4][5];



[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]

Represents column

Represents row

Example

```
int tmp[][] = new int[3][3];
int i,j;

Scanner test=new Scanner(System.in);

for(i=0;i<3;i++)
        for(j=0;j<3;j++)
        {
                System.out.print("Enter Input:");

                tmp[i][j]=test.nextInt();
        }

for(i=0;i<3;i++)
{
        for(j=0;j<3;j++)
                System.out.print(tmp[i][j] + " ");

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

Alternative Array Declaration Syntax

- ✓ Type [] var_name;
- ✓ Example: int [] a = new int[3];
- ✓ int [] num1, num2, num3;

  same as

  int num1[], num2[], num3[];

Class Declaration

```
class Circle {
  double radius = 1.0;

  double findArea(){
    return radius * radius * 3.14159;
  }
}
```

Declaring Object Reference Variables

```
ClassName objectReference;
```

Example:

```
Circle myCircle;
```

Creating Objects

```
objectReference = new ClassName();
```

Example:

```
myCircle = new Circle();
```

The object reference is assigned to the object reference variable.

Declaring/Creating Objects in a Single Step

```
ClassName objectReference = new ClassName();
```

Example:

```
Circle myCircle = new Circle();
```

Accessing Objects

- Referencing the object’s data:

```
objectReference.data
 myCircle.radius
```

- Invoking the object’s method:

```
objectReference.method
 myCircle.findArea()
```

Constructors

Constructors are used to initialize objects.

Constructors are a special kind of methods that are invoked to construct objects.

```
myCircle = new Circle(5.0);
 myCircle = new Circle();
myCircle1 = myCircle;
```

- · Constructors must have the same name as the class itself.
- · Constructors do not have a return type—not even void.
- · Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

Constructors, cont.

A constructor with no parameters is referred to as a default constructor.

A constructor with parameters is called as parameterized constructor.

A constructor with the same object as parameter is called as copy constructor.

```
Circle(double r) {
  radius = r;
 }
Circle() {
  radius = 1.0;
 }
Circle (Circle c) {
 radius = c.radius;}
```

Method overloading

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.* Method overloading is one of the ways that Java implements polymorphism.

Example

-
  ```
  class OverloadDemo {
  void test() {
  System.out.println("No parameters");
  }
  void test(int a) {
  System.out.println("a: " + a);
  }
  void test(int a, int b) {
  System.out.println("a and b: " + a + " "
  + b);
  }
  double test(double a) {
  System.out.println("double a: " + a);
  return a*a;
  }
  }
  ```

```
class Overload {
        public static void main(String
        args[]) {
        OverloadDemo ob = new
        OverloadDemo();
        double result;
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.2);
        System.out.println("Result of
        ob.test(123.2): " + result);
        }
        }
```

Class/Static Variables, Constants, and Methods

Class variables are shared by all the instances of the class.

Class methods are not tied to a specific object.

Class constants are final variables shared by all the instances of the class. To declare class variables, constants, and methods, use the **static** modifier.

static Variables

- You want to create a class member that will be used independently of any object of the class
- Also called **class variables**
- One copy of a *static* variable is created per class
- *static* variables are not associated with an object
- *static* constants are often declared as *public*
- To define a *static* variable, include the keyword *static* in its definition:
- Syntax:

```
accessSpecifier static dataType variableName;
```

- Example:

```
public static int countAutos = 0;
```

static Methods

- Also called **class methods**
- Often defined to access and change *static* variables
- *static* methods cannot access instance variables:
  - *static* methods are associated with the class, not with any object.
  - *static* methods can be called before any object is instantiated, so it is possible that there will be no instance variables to access.

Rules for *static* and Non-*static* Methods

	static Method	Non-*static* Method
Access instance variables?	no	yes
Access *static* class variables?	yes	yes
Call *static* class methods?	yes	yes
Call non-*static* instance methods?	no	yes
Use the object reference *this*?	no	yes

main method is always static and public

- They are public because they must be accessible to the JVM to begin execution of the program. Main is the first method that would get executed in any class.

  They are static because they must be available for execution without an object instance. you may know that you need an object instance to invoke any method. So you cannot begin execution of a class without its object if the main method was not static.

The Keyword this

- Use this to refer to the current object.
- Use this to invoke other constructors of the object.

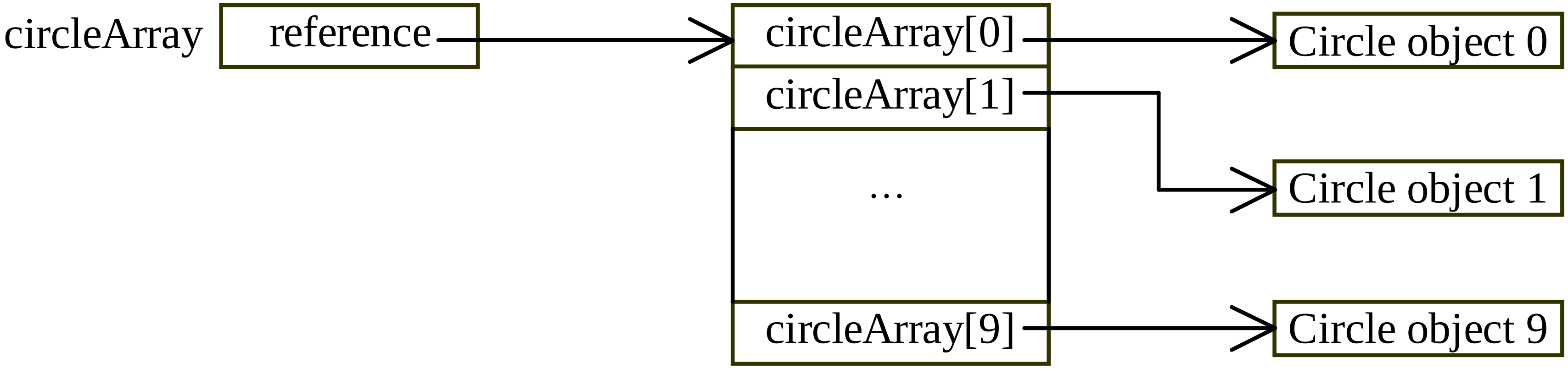
Array of Objects

```
Circle[] circleArray = new Circle[10];
```

An array of objects is actually an *array of reference variables*.

Array of Objects, cont.

```
Circle[] circleArray = new Circle[10];
For (i=0; i<3; i++)
circleArray[i] = new Circle();
```



A Student Class Example

```
import java.util.Scanner;

class Student
     {
     private static int count=0;
     private int no;
     private String name;
     private int age;
     private int height, weight;
     private String course;
```

```
Student ()
        { no=0;
         age=0;
         height=0;
         weight=0;
         name= " ";
         course= " "; };

        Student(int n, String s , int ag, int h, int w,
        String c)
        {
        no = n;
        name = s;
        age = ag;
        height = h;
        weight = w;
        course = c;
        }
```

-

```
Student (Student st)
{
no = st.no;
name = st.name;
age = st.age;
height = st.height;
weight = st.weight;
course = st.course;
}
```

```
static { System.out.println("Count initialized");
    count =1000;}

    static void stcount()
            {
//          System.out.println(no + name);   - Not permitted
            count++;
            }

    static int noofobjects() {return count;}
```

```
void disp(int a )
    { if (age < a) System.out.println(no + " "+  name + " "+ age);}

    void disp()
    { System.out.println(no + " "+  name + " "+ age);}
```

```
void set()
    {
    Scanner s = new Scanner(System.in);
    System.out.println("Enter no name age ht wt and course");
    no = s.nextInt();
    name = s.next();
    age = s.nextInt();
    height = s.nextInt();
    weight = s.nextInt();
    course = s.next();
    }

    protected void finalize(){};
    }
```

```
class stu
    {
    public static void main(String args[])
    {
    Student[] s = new Student[4];
    final int no = 3;
    for (int i = 0; i<no; i++)
    {
    s[i] = new Student();
    Student.stcount();
    s[i].set();
    }
System.out.println("No of student objects"+
    Student.noofobjects());
    System.out.println("\n\nNo Name   Age");
    for (int i = 0; i<no; i++)
    s[i].disp();
    System.out.println("\n\nNo Name   Age less than 18");
    for (int i = 0; i<no; i++)
    s[i].disp(18);
    }
    }
```

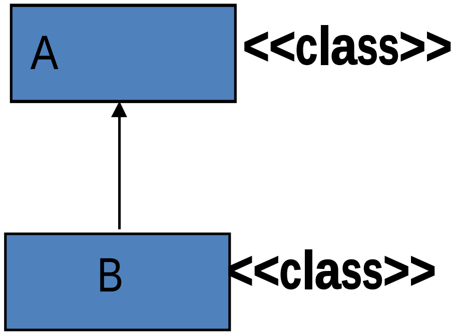
Inheritance

1. Reusability is achieved by INHERITANCE
2. Java classes Can be Reused by extending a class. Extending an existing class is nothing but reusing properties of the existing classes.
3. The class whose properties are extended is known as super or base or parent class.
4. The class which extends the properties of super class is known as sub or derived or child class
5. A class can either extends another class or can implement an interface

class B extends A { ..... }

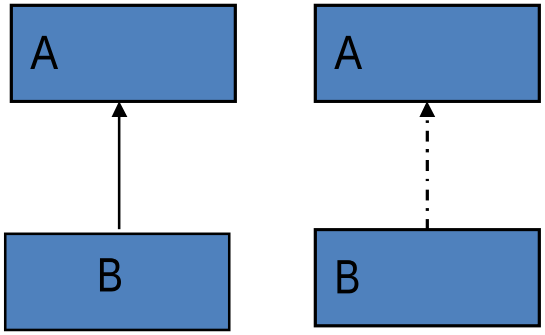
A super class

B sub class

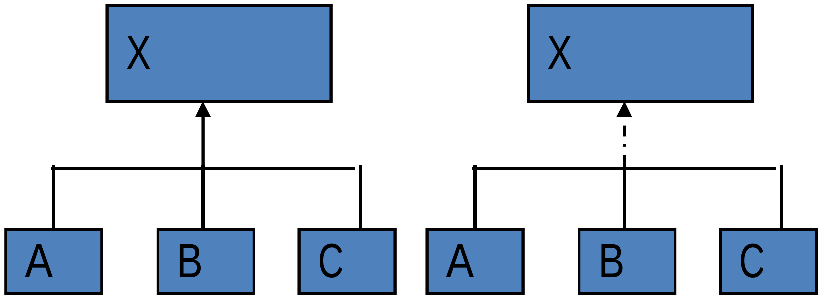


Various Forms of Inheritance

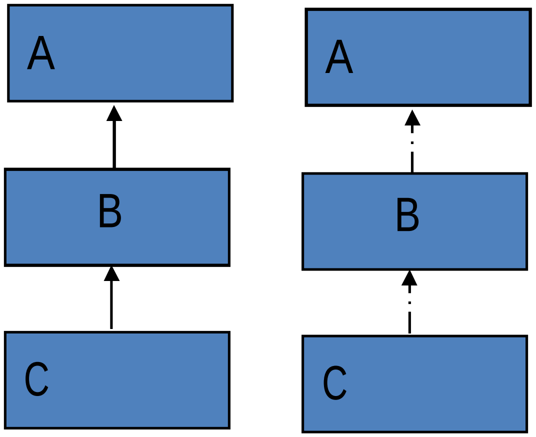
Single Inheritance



Hierarchical Inheritance

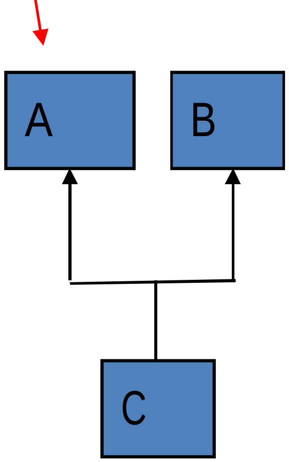


MultiLevel Inheritance



NOT SUPPORTED BY JAVA

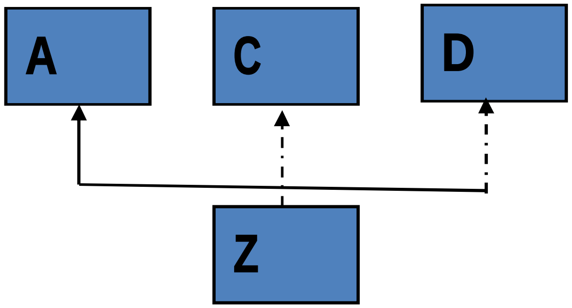
Multiple Inheritance



Forms of Inheritance

- **Mulitiple Inheritance can be implemented by implementing multiple interfaces not by extending multiple classes**

Example :



class Z extends A ,B

{

} ***WRONG***

OR

class Z extends A extends B

{

} ***WRONG***

Defining a Subclass

Syntax :

```
class <subclass name> extends <superclass name>
{
 variable declarations;
 method declarations;
}
```

- Extends keyword signifies that properties of the super class are extended to sub class
- Sub class will not inherit private members of super class

Access Control

Access Modifiers → Access Location ↓	public	protected	**Friendly/Default**	**private**
Same Class	**Yes**	**Yes**	**Yes**	**Yes**
sub classes in same package	**Yes**	**Yes**	**Yes**	**No**
Other Classes in Same package	**Yes**	**Yes**	**Yes**	**No**
Subclasses in other packages	**Yes**	**Yes**	**No**	**No**
Non-subclasses in other packages	**Yes**	**No**	**No**	**No**

Inheritance Basics

1. Whenever a sub class object is created ,super class constructor is called first.

2. If super class constructor does not have any constructor of its own OR has an unparametrized constructor then it is automatically called by Java Run Time by using call super()

3. If a super class has a parameterized constructor then it is the responsibility of the sub class constructor to call the super class constructor by call

   super(<parameters required by super class>)

4. Call to super class constructor must be the first statement in sub class constructor

Inheritance Basics

When super class has a Unparametrized constructor

```
class A
{
A()
{
System.out.println("This is constructor of class A");
}
} // End of class A
class B extends A
{
B()
{
super();
System.out.println("This is constructor of class B");
}
} // End of class B
```

Optional

Cont.....

```
class inhtest
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

OUTPUT

```
This is constructor of class A
This is constructor of class B
```

```
class A
{
private int a;
A( int x)
{
a =x;
System.out.println("This is constructor of
class A");
} }
class B extends A
{
private int b;
private double c;
B(int x,int y,double z)
{
super(x);
b=y;
c=z;
System.out.println("This is constructor of
class B");
} }
```

```
B b1 = new B(8,10,8.6);
```

OUTPUT

```
This is constructor of class A
This is constructor of class B
```

```
class A
{
private int a;
protected String name;
A(int x, String s)
{
a = x;
name = s;
}
void print()
{
System.out.println("a="+a);
}
}
```

```
class B extends A
{
int b;
double c;
B(int a,String n,int b,double c)
{
super(a,n);
this.b=b;
this.c =c;
}
void show()
{
//System.out.println("a="+a);
print();
System.out.println("name="+name);
System.out.println("b="+b);
System.out.println("c="+c);
}
}
```

a is private in class A

Call to print() from super class A

Accessing name field from super class (super.name)

```
class xyz3
{
public static void main(String args[])
{
B b1 = new B(10,"OOP",8,10.56);
b1.show();
}
}
```

```
E:\Java>java xyz3
a=10
name=OOP
b=8
c=10.56
```

USE OF super KEYWORD

- Can be used to call super class constrctor

  super();

  super(<parameter-list>);

- Can refer to super class instance variables/Methods

  super.<super class instance variable/Method>

```
class A
{
private int a;
A( int x)
{
a =x;
System.out.println("This is constructor of
class A");
}
void show()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is Display in
A");
}
}
```

```
class B extends A
{
private int b;
private double c;
B(int x,int y,double z)
{
super(x);
b=y;
c=z;
System.out.println("This is constructor of
class B");
}
void show()
{
super.show();
System.out.println("b="+b);
System.out.println("c="+c);
display();
}
}
```

Use of final keyword in java

1. final keyword in java can be used for following
   (i) class declaration
   (ii) variable declartion
   (iii) method decalaration

2. final keyword for class means class cannot be inherited. Final classes can not have subclasses.

3. final keyword used with variable declaration makes it constant whose value cannot be changed. Final variables should be initialized to some values at the time of declaration.

4. final keyword used with method definition means method can not be overridden by subclasses (Makes sense only when inheritance is used)

final class

```
final class ABC
{
…………………
…………………
}
class a extends ABC
{
……
} //  Wrong / Invalid
```

final instance variable

```
final int x = 40;
final double x = 3.4;
final double PI = 3.14
```

final methods

```
class ABC
{
…………………
final void show()..
}
class a extends ABC
{
void show()……
} //  Wrong / Invalid
```

METHOD OVERRIDING

1. Sub class can override the methods defined by the super class.

2. Overridden Methods in the sub classes should have same name, same signature , same return type and may have either the same or higher scope than super class method.

3. Java implements Run Time Polymorphism/ Dynamic Method Dispatch by Method Overriding. [Late Binding]

4. Call to Overridden Methods is Resolved at Run Time.

5. Call to a overridden method is not decided by the type of reference variable Rather by the type of the object where reference variable is pointing.

6. While Overriding a Method, the sub class should assign either same or higher access level than super class method.

EXAMPLE METHOD OVERRIDING

```
Class A
{
void show()
{
System.out.println("Hello This is show() in A");
}// End of show() Method
} // End of class A
class B extends A
{
void show()
{
System.out.println("Hello This is show() in B");
}// End of show() Method
} // End of class B
```

B class overrides show() method from super class A

```
class override
{
public static void main(String args[])
{
// super class reference variable
// can point to sub class object
A a1 = new A();
a1.show();
a1 = new B();
a1.show();
}
}
```

Call to show() of A class

Call to show() of B class

```
class A
{
void show(int a)
{
System.out.println("Hello This is show()
in A");
}
}
class B extends A
{
void show()
{
System.out.println("Hello This is show()
in B");
}
}
```

Is this Method
Overriding

NO

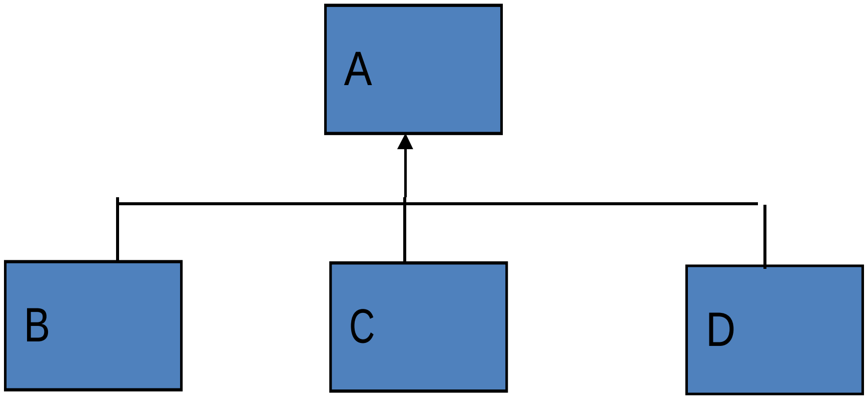
```
class override1
{
public static void main(String args[])
{
/*
A a1 = new B();
a1.show(); */

A a1 = new A();
a1.show(10);

B b1 = new B();
b1.show(10);
b1.show();  }
```

```
OUTPUT
Hello This is show() in A
Hello This is show() in A
Hello This is show() in B
```

Dynamic Method Dispatch

1. Super class reference variable can refer to a sub class object.
2. ***Super class variable if refers to sub class object can call only overridden methods***.
3. Call to an overridden method is decided by the type of object referred to.

```
A a1 = new B();
a1.show(); // call to show() of B
a1 = new C();
a1.show(); // call to show() of C
a1 = new D();
a1.show();  // call to show() of D
```



Assume show() Method is Overridden by sub classes

DYNAMIC METHOD DISPATCH

```
class A
{
void show()
{
System.out.println("Hello This is
show() in A");
}
}
class B extends A
{
void show()
{
System.out.println("Hello This is
show() in B");
}
}
```

```
class C extends A
{
void show()
{
System.out.println("Hello This is
show() in C");
}
}
class D extends A
{
void show()
{
System.out.println("Hello This is
show() in D");
}
}
```

CONTINUED.....

```
class override2
{
public static void main(String args[])
{
A a1 = new A();
a1.show();
a1 = new B();
a1.show();
a1 = new C();
a1.show();
a1 = new D();
a1.show();
}
}
```

```
Hello This is show() in A

Hello This is show() in B

Hello This is show() in C

Hello This is show() in D
```

Examples Overriding

```
class A
{
void show() { .... }
}
class B extends A
{
void show() { .... }
void show(int x) { ... }
void print() { ... }
}
```

```
A a1 = new B();
a1.show() ;        // Valid
// a1.show(10);  //   Invalid
//a1.print();        //   Invalid
```

When a super class variable points to a sub class object, then it can only call overridden methods of the sub class.

Method overloading vs method overriding

- **Method overloading** means having two or more **method**s with the same name but different signatures in the same scope. These two **method**s may exist in the same class or another one in base class and another in derived class.
- **Method overriding** means having a **different implementation of the same method** in the **inherited class**. These two **method**s would have the **same signature, but different implementation**. One of these would exist in the **base class** and another in the **derived class**. These cannot exist in the same class.

Abstract Classes

1. An abstract class is a class that can have *abstract methods* (i.e a method with only heading with no body of executable statements)
2. We can not create an object of abstract classes i.e abstract class objects can not be instantiated
3. An abstract class needs to be extended by sub classes to provide the implementation for the abstract methods.
4. Abstract classes may contain static methods
5. abstract and static keyword combination is wrong
   abstract static void print(); wrong
6. Abstract classes may extend either another abstract class or concrete class
7. Abstract classes may include constructors, nested classes and interfaces
8. Abstract classes has either public, protected, private or package accessibility

Abstract Classes

- **Syntax :**

```
 abstract class <classname>
{

  ……………………….

  abstract <return type> methodname(<parameter List>);
  abstract <return type> methodname(<parameter List>);
}
```

Note:

1. **Abstract class can have one or more abstract methods**
2. **Abstract classes may extend another class , implements another interface , may have concrete methods**

```
abstract class A
{
public int show()
{
System.out.println("Class A");
return 0;
}
}

class sample
{
public static void main(String args[])
{
A a1 = new A();
}
}
```

What's wrong with the code

```
sample.java:14: A is abstract; cannot be instantiated
A a1 = new A();
     ^
```

```
abstract class A
{
public int show()
{
System.out.println("Class A");
return 0;
}
void print();
} // End of class A
```

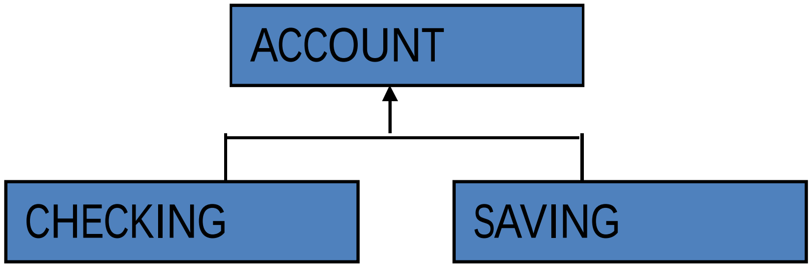
What’s wrong with the code

```
sample.java:8: missing method body, or
declare abstract
void print();
     ^
1 error
```

```
abstract class A
{
public int show()
{
System.out.println("Class A");
return 0;
}
abstract void print();
} // End of class A
```

EXAMPLES ABSTRACT CLASS

```
abstract class Account
{

private String name;
private String actno;
private double balance;
private Address addr;

// Overloaded Constructors
Account(String n,String a) { }
Account(String n,String a,double b) { }

// Accessor Methods
String getName()      { return name;}
String getactno()     { return actno;}
double getbalance() { return balance;}
// provide abstract methods
abstract double withdraw(double amount);
abstract void deposit(double amount);
}
```



```
abstract class Account
{
private String name;
private String actno;
private double balance;
private Address addr;

// Overloaded Constructors
Account(String n,String a)
{
name = n;
actno= a;
balance = 0.0;
}
Account(String n,String a,double b)
{
name = n;
actno= a;
balance = b;
}
```

```
// Accessor Methods
String getName()  { return name;}
String getactno() { return actno;}
double getbalance() { return balance;}

// Mutator Method only for balance
void setbalance(double amount)
{  this.balance = amount;}
void showAccountDetails()
{
System.out.println("Name :"+this.getName());
System.out.println("Account No :"+this.getactno());
System.out.println("Balance :"+this.getbalance());
}
// provide abstract methods
abstract double withdraw(double amount);
abstract void deposit(double amount);
} // END OF Account CLASS
```

```
class Saving extends Account
{
Saving(String n,String a)
{
super(n,a);
System.out.println("Saving Account Created");
System.out.println("Name :"+this.getName());
System.out.println("Account No :"+this.getactno());
System.out.println("Balance :"+this.getbalance());
showAccountDetails();
}
Saving(String n,String a,double b)
{
super(n,a,b);
System.out.println("Saving Account Created");
System.out.println("Name :"+this.getName());
System.out.println("Account No :"+this.getactno());
System.out.println("Balance :"+this.getbalance());
showAccountDetails();
}
double withdraw(double amount)
{
 /*
 if( balance == 0) return 0.0;
 if( balance < amount ) return 0.0;
 balance = balance - amount;
*/

 if(this.getbalance() == 0) return 0.0;
 if(this.getbalance() < amount ) return 0.0;
 setbalance(getbalance() - amount);
 return amount;
}
void deposit(double amount)
{
 setbalance(getbalance() + amount);
 return ;
}
}//end of Saving class
```

```
class Checking extends Account
{
Checking(String n,String a,double b)
{
super(n,a,b);
System.out.println("Checking Account Created");
showAccountDetails();
}
double withdraw(double amount)
{
/*
 if( balance - 100 == 0) return 0.0;
 if( balance -100 < amount ) return 0.0;
 balance = balance - amount - 100;
*/

 if(this.getbalance() - 100 == 0) return 0.0;
 if(this.getbalance() - 100 < amount ) return 0.0;
 setbalance(this.getbalance() - amount);
 return amount;
}
```

```
void deposit(double amount)
{
 setbalance(this.getbalance() + 0.9 *
amount)  ;
 return ;
}
}//end of Checking class
```

```
class AccountTest
{
public static void main(String args[])
{
Checking c1 =  new Checking("Rahul Sharma","C106726",100000);
Checking c2 =  new Checking("Raman Kumar","C106727",100000);

Saving s1 =  new Saving("Kumar Sharma","S106726",100000);
Saving s2 =  new Saving("Mohan Lal","S106727");

c1.withdraw(2000);
c1.showAccountDetails();
c2.deposit(10000);
c2.showAccountDetails();
s1.deposit(900);
s1.showAccountDetails();
s2.withdraw(400);
s2.showAccountDetails();
}
}
```

INTERFACES IN JAVA

1. **Java Does not support Multiple Inheritance directly. Multiple inheritance can be achieved in java by the use of interfaces.**
2. **We need interfaces when we want functionality to be included but does not want to impose implementation.**
3. **Implementation issue is left to the individual classes implementing the interfaces.**
4. **Interfaces can have only abstract methods and final fields.**
5. **You can declare a variable to be of type interface. But you can not create an object belonging to type interface.**
6. **Interface variable can point to objects of any class implementing the interface.**
7. **Another way of implementing Run Time Polymorphism.**

Similarities between Interfaces and classes

- is compiled into byte code file
- can be either public,protected, private or package accessibility
- can not be public unless defined in the file having same name as interface name
- serve as a type for declaring variables and parameters

Differences between Interfaces and classes

- Declares only method headers and public constants
- Has no constructors
- Can be implemented by a class
- Can not extend a class
- Can extend several other interfaces

General Form

- Syntax :

```
<access specifier> interface <interface name>  extends [ <interface1> ,
<interface 2> ......]
{
 [public][final] variablename 1  = value;
 ..................................................
 [public][final] variablename N = value;
 [public][abstract] <return type> methodname 1(<parameter lis>);
 [public][abstract] <return type> methodname 2(<parameter lis>);
 .....................................................................................
 [public][abstract] <return type> methodname N(<parameter lis>);
}
```

Examples

```
public interface  A
{
double PI = 3.14156;
void show();
void display();
}
```

Should be typed in file A.java

By Default public final Should be initialized

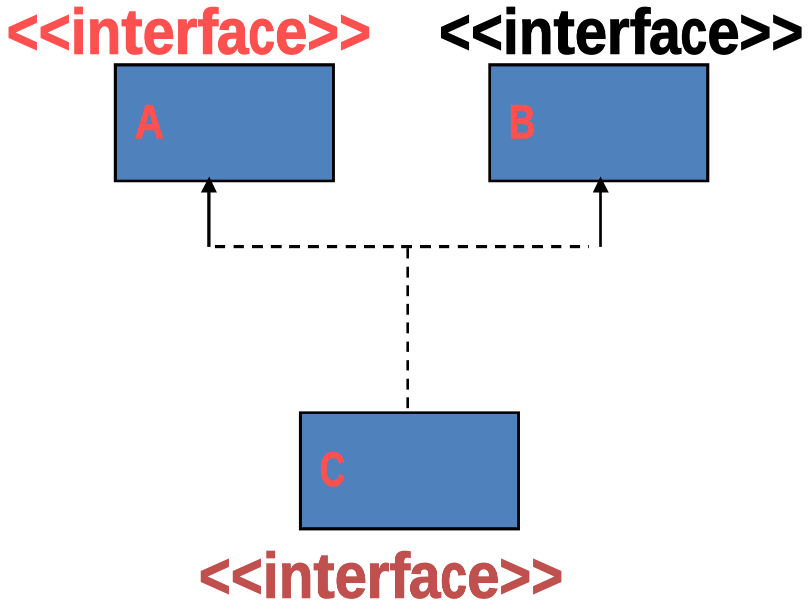
double PI; → Wrong

Can have only abstract methods. Each method is by default public abstract

```
class XYZ implements A
{
public void show()   { ..... }
public void display() { .....  }
}
```

Interfaces Can Extend Multiple Interfaces

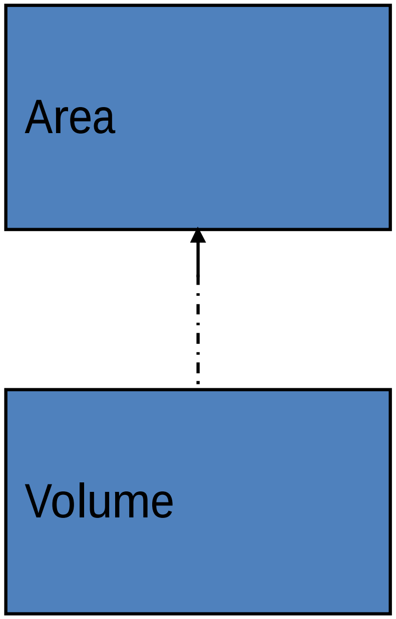
```
interface A
{
int a=10;
void show();
}
interface B
{
int b=10;
void print();
}
interface C extends A,B
{
void display();
}
```



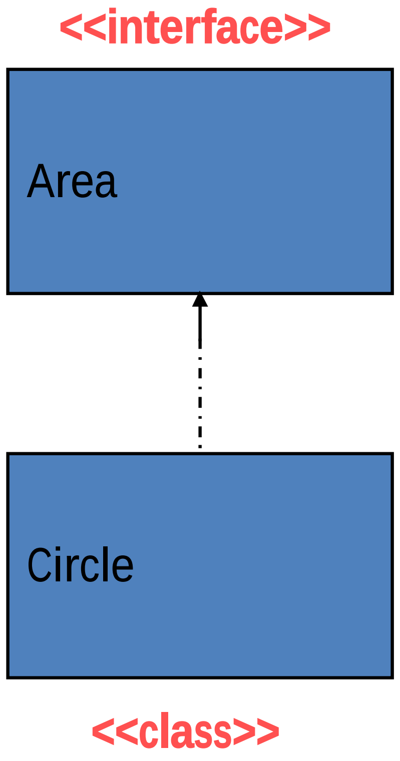
Interface Can not Extend a Class

Interface Example

```
interface Area
{
double PI = 3.1456;
double area();
double perimeter();
}
interface Volume extends Area
{
double volume();
}
```



```
class Circle implements Area
{
private double radius;
Circle(double radius)
{
this.radius = radius;
}
double getRadius() { return radius;}
public double area()
{
return PI * radius * radius;
}
public double perimeter()
{
return 2 * PI * radius;
}
}
```



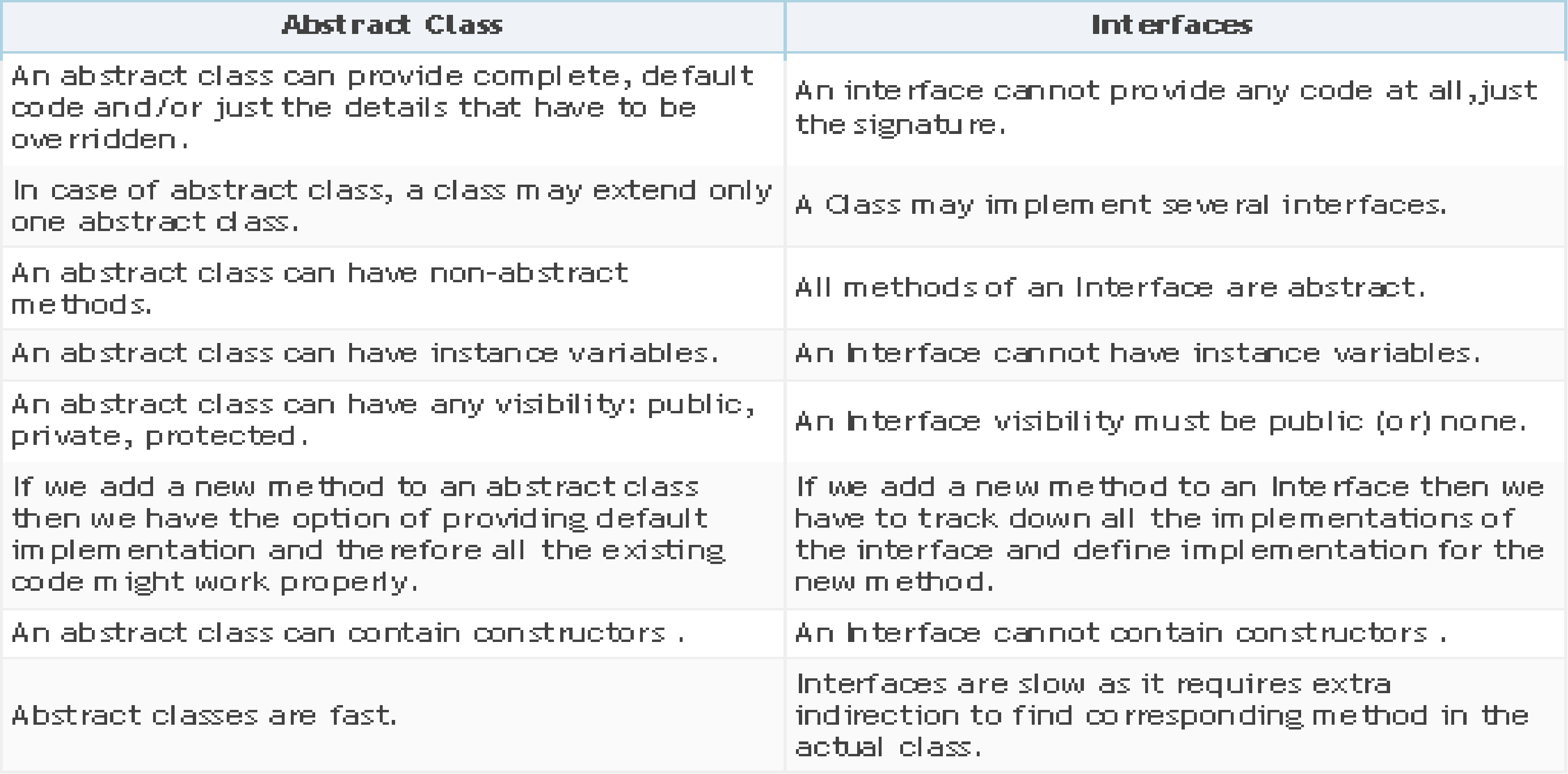
<<interface>>

Area

Circle

<<class>>

Abstract class vs Interface



Abstract Class	Interfaces
An abstract class can provide complete, default code and/or just the details that have to be overridden.	An interface cannot provide any code at all,just the signature.
In case of abstract class, a class may extend only one abstract class.	A Class may implement several interfaces.
An abstract class can have non-abstract methods.	All methods of an Interface are abstract.
An abstract class can have instance variables.	An Interface cannot have instance variables.
An abstract class can have any visibility: public, private, protected.	An Interface visibility must be public (or) none.
If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.	If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method.
An abstract class can contain constructors .	An Interface cannot contain constructors .
Abstract classes are fast.	Interfaces are slow as it requires extra indirection to find corresponding method in the actual class.

Packages

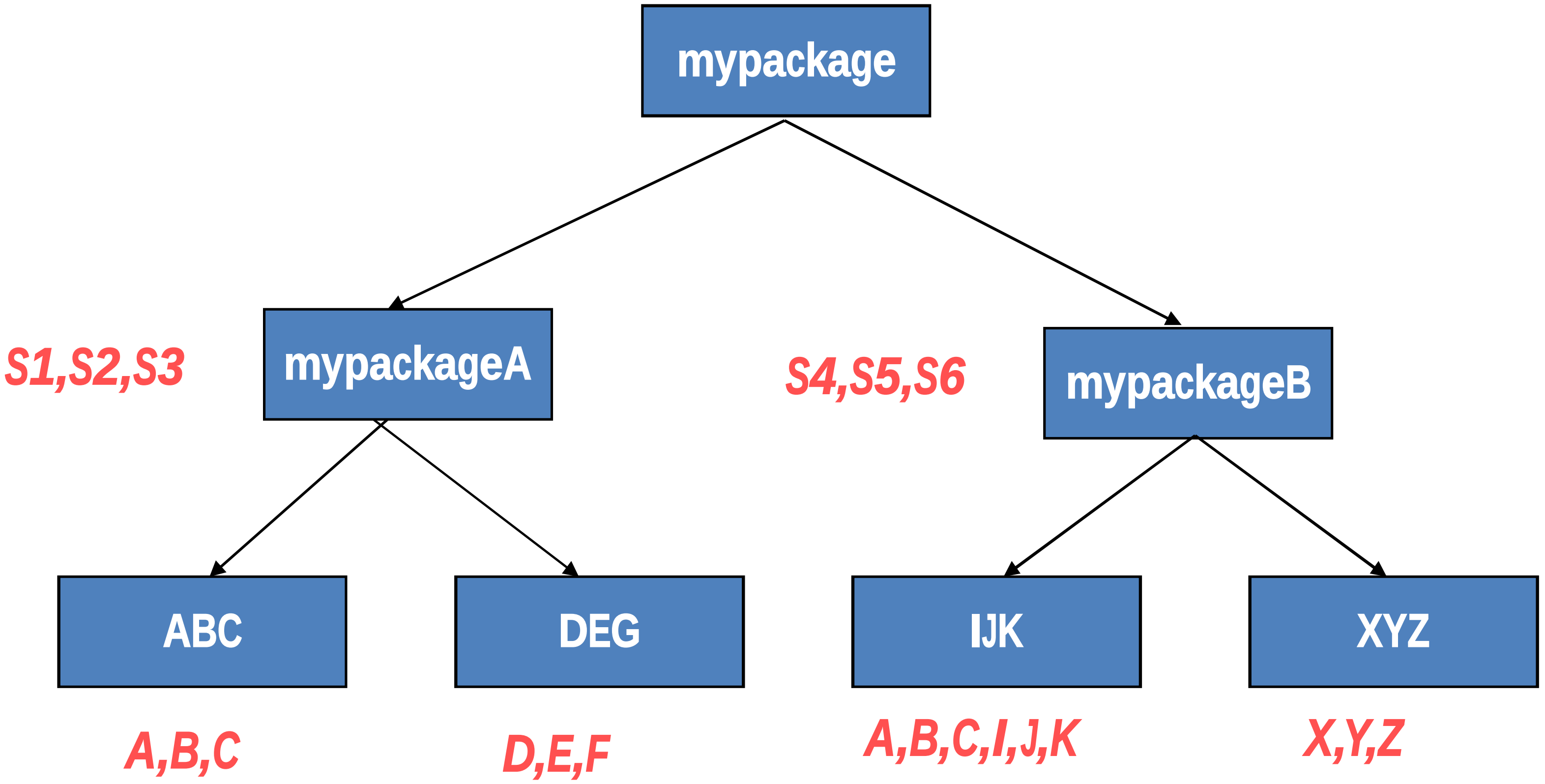
- Packages enable grouping of functionally related classes
- Package names are dot separated, e.g., java.lang.
- Package names have a correspondence with the directory structure
- Packages Avoid name space collision. There can not be two classes with same name in a same Package But two packages can have a class with same name.
- Exact Name of the class is identifed by its package structure. << Fully Qualified Name>>

java.lang.String ; java.util.Arrays; java.io.BufferedReader ; java.util.Date

How To Create a Package

- Packages are mirrored through directory structure.
- To create a package, First we have to create a directory /directory structure that matches the package hierarchy.
- Package structure should match the directory structure also.
- To make a class belongs to a particular package include the package statement as the first statement of source file.

Exercise - Creating Packages



Package ABC and IJK have classes with same name.

A class in ABC has name ***mypackage.mypackageA.ABC.A***

A class in IJK has name ***mypackage.mypackageB.IJK.A***

How to make a class Belong to a Package

- *Include a proper package statement as first line in source file*

Make class S1 belongs to mypackageA

```
package mypackage.mypackageA;
public class S1
{
public S1( )
{
System.out.println("This is Class S1");
}
}
```

Name the source file as S1.java and compile it and store the S1.class file in mypackageA directory. To execute, goto the parent dir and issue the command

```
cd ..

cd..

java mypackage.mypackageA.S1
```

Make class S2 belongs to mypackageA

```
package mypackage.mypackageA;
public class S2
{
public S2( )
{
System.out.println("This is Class S2");
}
}
```

Name the source file as S2.java and compile it and store the S2.class file in mypackageA directory

Importing the Package

- import statement allows the importing of package
- Library packages are automatically imported irrespective of the location of compiling and executing program
- JRE looks at two places for user created packages

  (i) Under the current working directory

  (ii) At the location specified by CLASSPATH environment variable
- Most ideal location for compiling/executing a program is immediately above the package structure.

Example importing
import mypackage.mypackageA.ABC;

```
import mypackage.mypackageA.ABC.*;
class packagetest
{
public static void main(String args[])
{
B b1 = new B();
C c1 = new C();
}
}
```

<< packagetest.java>>

<< Store it in location above the package structure. Compile and Execute it from there>>

```
This is Class B
This is Class C
```

```
import mypackage.mypackageA.ABC.*;
Import mypackage.mypackageB.IJK.*;
class packagetest
{
public static void main(String args[])
{
A a1 = new A();
}
}
```

<< What's Wrong Here>>

```
mypackage.mypackageA.ABC.A a1 = new mypackage.mypackageA.ABC.A();
```

OR

```
mypackage.mypackageB.IJK.A a1 = new mypackage.mypackageB.IJK.A();
```

<< class A is present in both the imported packages ABC and IJK. So A has to be fully qualified in this case>>

CLASSPATH Environmental Variables

- CLASSPATH Environmental Variable lets you define path for the location of the root of the package hierarchy
- Consider the following statement :

  package mypack;

  What should be true in order for the program to find mypack.

  (i) Program should be executed from the location immediately above mypack

  OR

  (ii) mypack should be listed in the set of directories for CLASSPATH

What is an exception?

- An exception is a problem that arises during the execution of a program.
- An exception can occur for many different reasons, including the following:
  - A user has entered invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications or the JVM has run out of memory.
- Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Java Exception Handling-Fundamentals

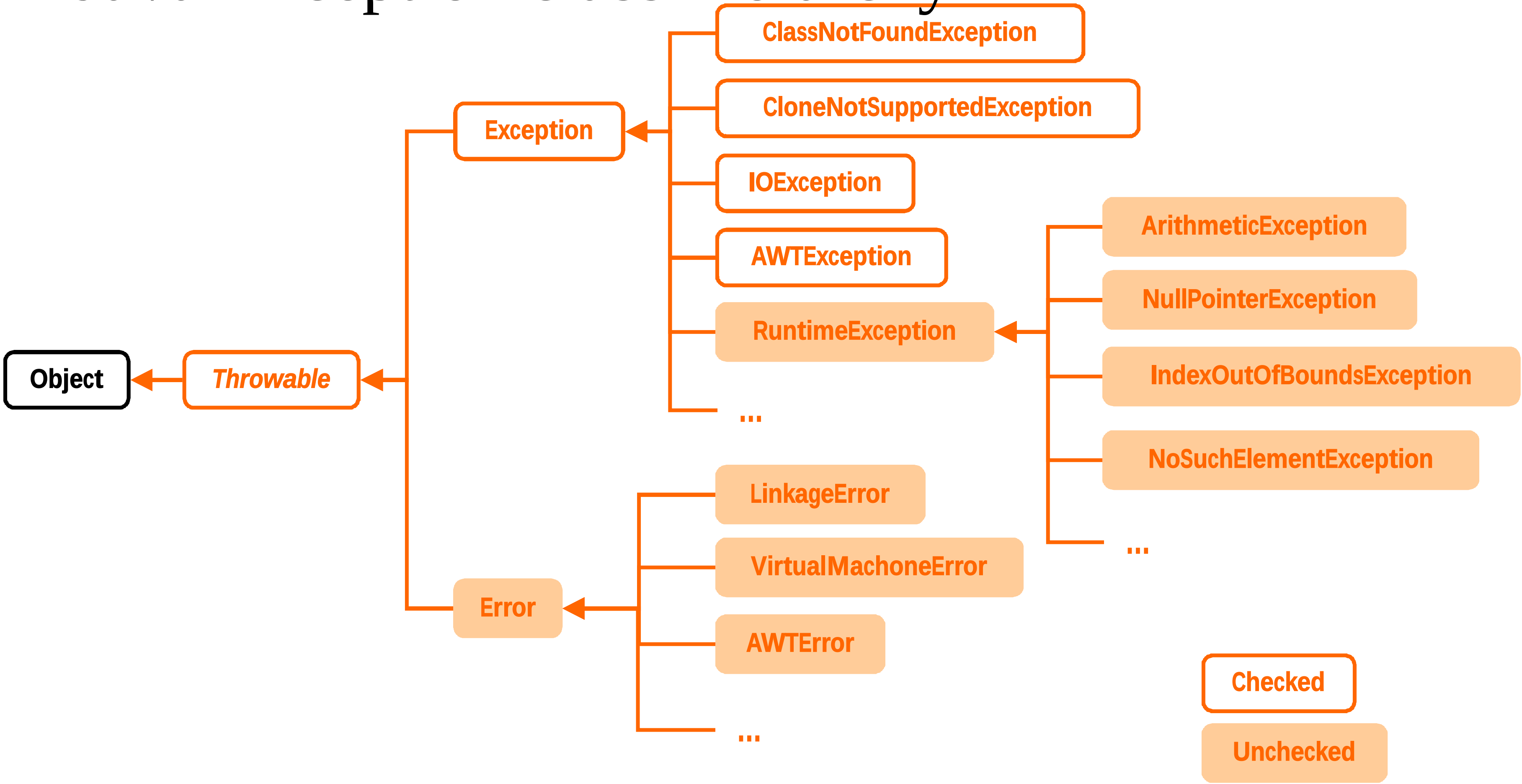
- An exception is an abnormal condition that arises in a code sequence at run time
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error
- An exception can be caught to handle it or pass it on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code

Types of Exceptions

- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that should be handled by the program. They are handled at compile time.
  - Class / File not Found
- **Unchecked Exceptions - Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer.
  - Divide by zero
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.
  - stack overflow

Representing Exceptions

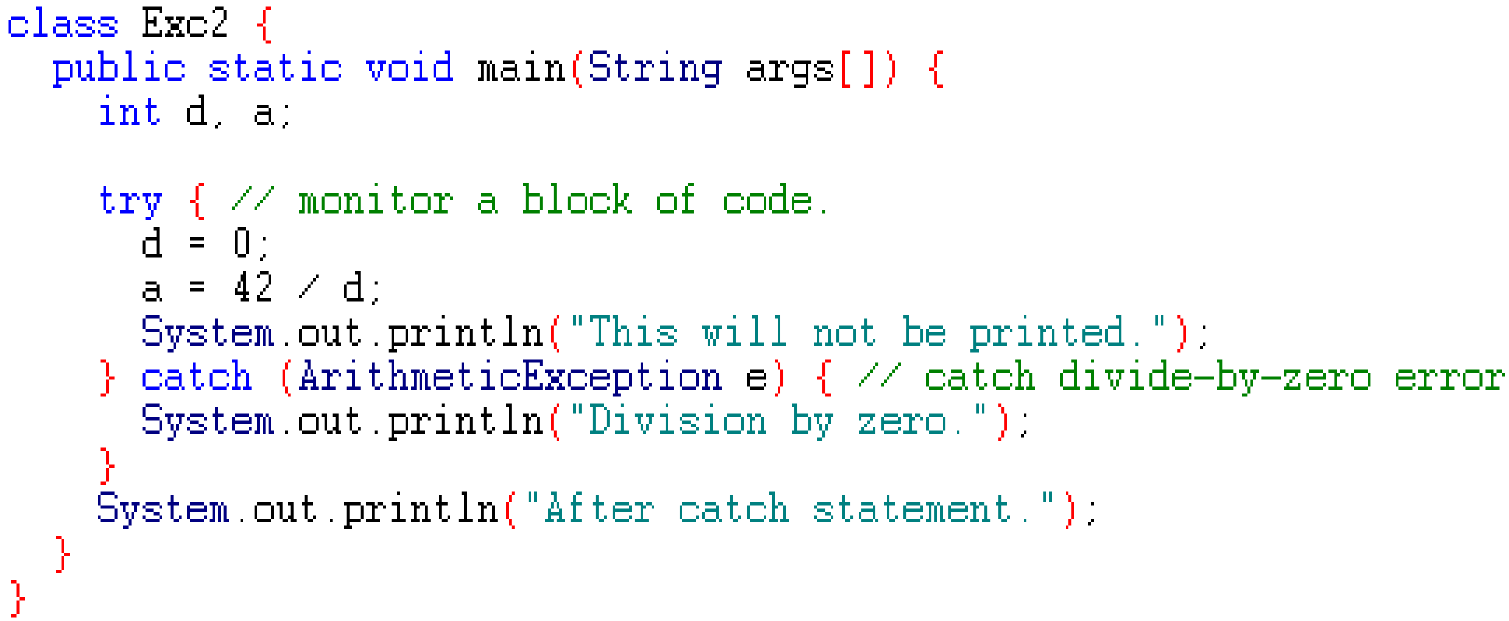
Java Exception class hierarchy



Exception Handling in Java

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**
- Program statements to monitor are contained within a **try** block
- If an exception occurs within the **try** block, it is thrown
- Code within **catch** block catch the exception and handles it

Example



```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

Output:

Division by zero.

After catch statement.

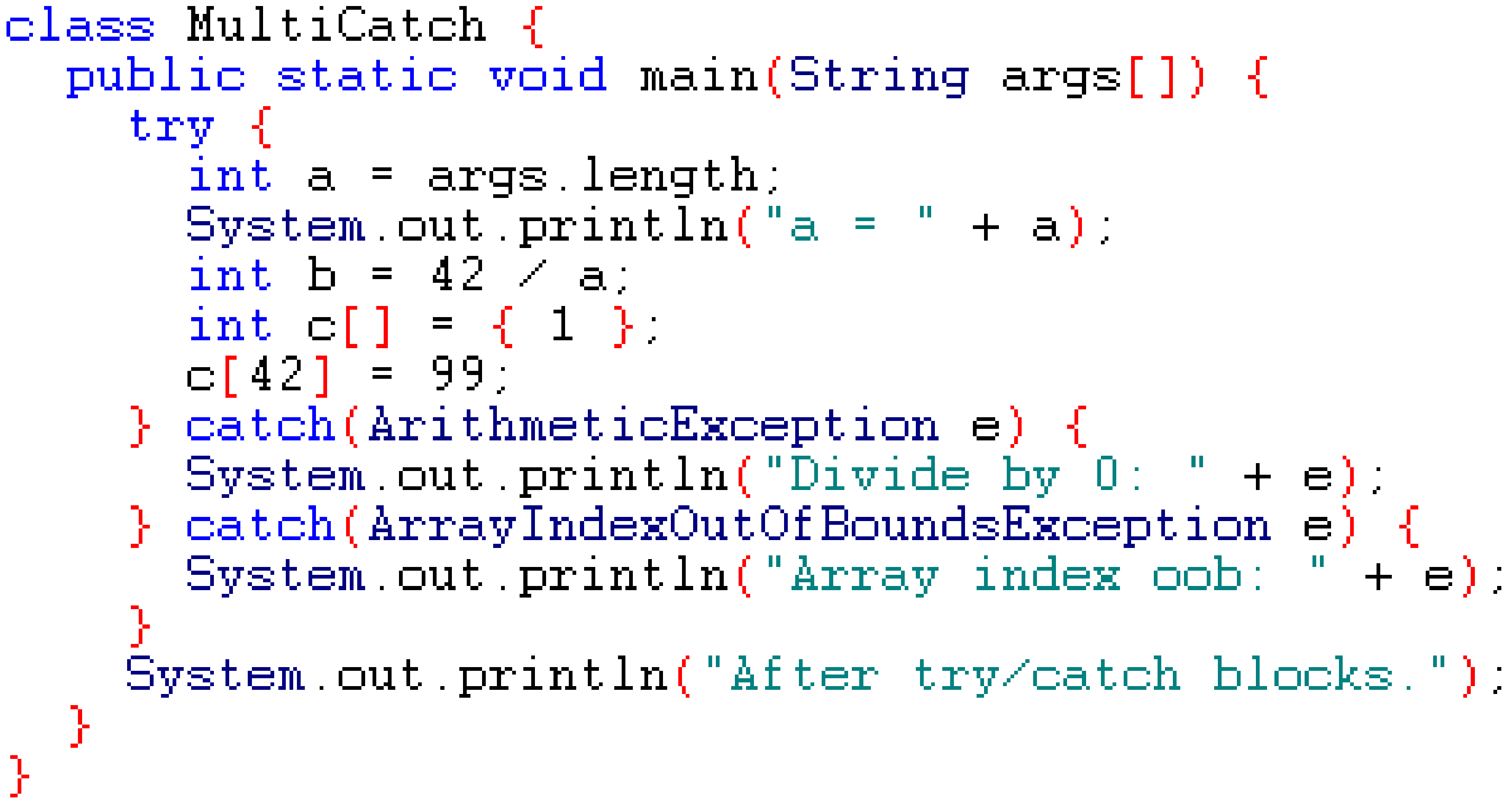
try and catch statement

- The scope of a **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement.
- The statements that are protected by the **try** must be surrounded by curly braces.

Multiple Catch Clauses

- If more than one can occur, then we use multiple catch clauses
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed
- After one **catch** statement executes, the others are bypassed

Example

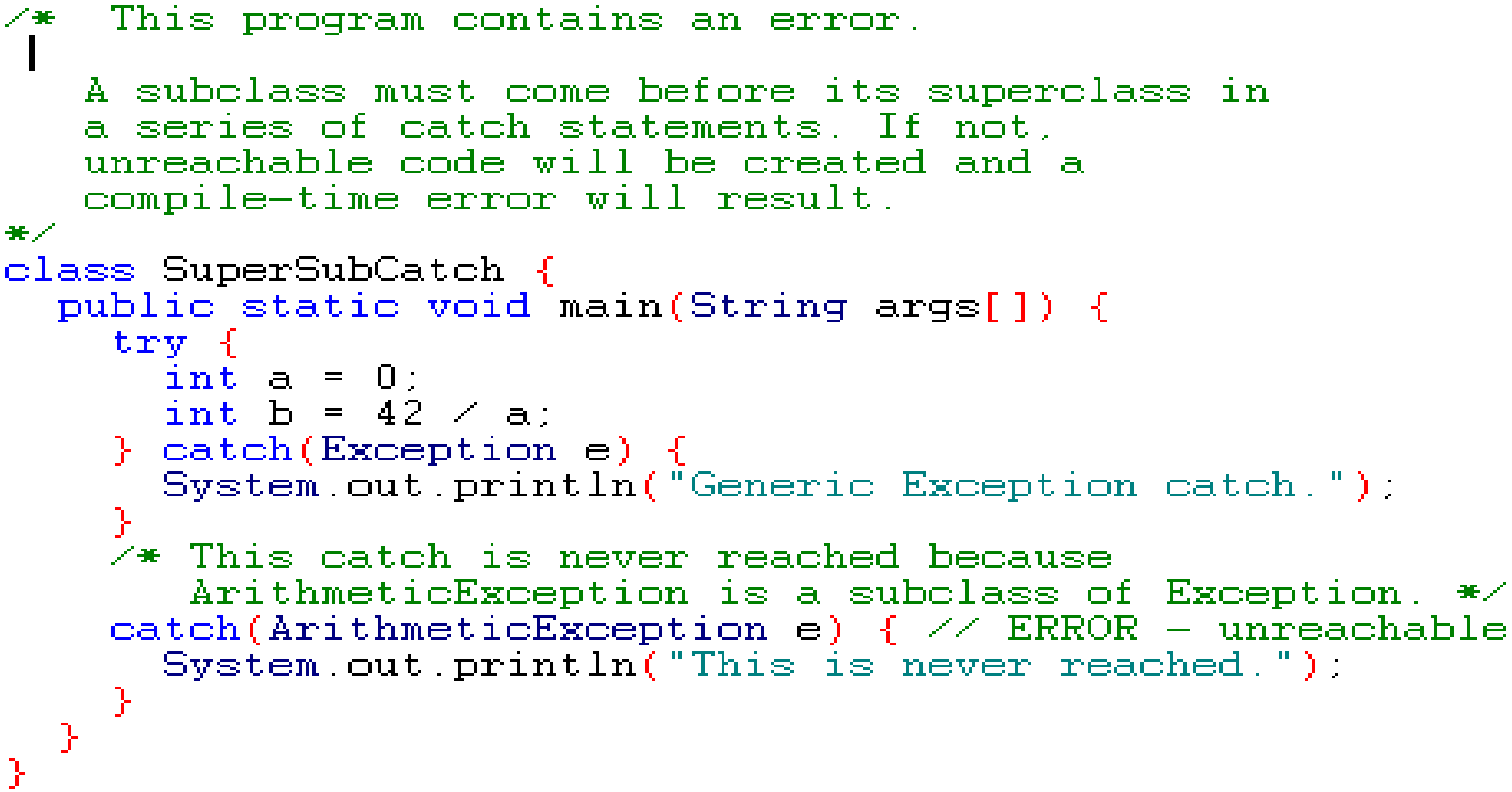


```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

Caution

- Exception subclass must come before any of of their superclasses
- A **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it come after its superclass
- For example, **ArithmeticException** is a subclass of **Exception**
- Moreover, unreachable code in Java generates error

Example



```
/*  This program contains an error.

   A subclass must come before its superclass in
   a series of catch statements. If not,
   unreachable code will be created and a
   compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
      System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

Nested try Statements

- A **try** statement can be inside the block of another try
- Each time a **try** statement is entered, the context of that exception is pushed on the stack
- If an inner **try** statement does not have a catch, then the next **try** statement's catch handlers are inspected for a match
- If a method call within a **try** block has **try** block within it, then it is still nested **try**

Example

```java
// An example nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command line arg is used,
           then an divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command line args are used
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

throw

- It is possible for your program to throw an exception explicitly
- throw *ThrowableInstance*
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass **Throwable**
- There are two ways to obtain a **Throwable** objects:
  - Using a parameter into a catch clause
  - Creating one with the **new** operator

Example

if (n > 5) throw new TooManyException();

```
class TooManyException extends Exception
    {
    TooManyException() {System.out.println("Too many numbers - I can not handle");}
    }
```

if (n < 2) throw new TooFewException("Too few numbers.");

```
class TooFewException extends Exception
    {
    TooFewException(String m){super(m);}
    }
```

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception
- *type method-name parameter-list)* throws *exception-list*

```
 {
      // body of method
 }
```

- It is not applicable for **Error** or **RuntimeException,** or any of their subclasses

```
class buf
   {
   public static void main(String args[]) throws IOException
   {
   BufferedReader b = new BufferedReader(new
   InputStreamReader(System.in));
   System.out.println("Enter first no");
   int n1 = Integer.parseInt(b.readLine());
 }
```

Finally Statement

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- **finally** block will be executed whether or not an exception is thrown.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- Each try clause requires at least one catch or finally clause.

Try – finally blocks

```
try
    {
     c=a/b;
     System.out.println("This may be printed or may not");
    }

   finally
   {
   System.out.println("I will always be there");
   }
```

Try – catch – finally blocks

```
try
    {
     d= A[100]+1;
    }
   catch(ArrayIndexOutOfBoundsException e)
    {
    System.out.println("Array - unreachable element "+e);
    }
  finally
  {
  System.out.println("Finally block inside - Always executed");
  }
```

Uncaught Exceptions

```
class exc0{
public static void main(String args[])
{
     int d=0;
     int a=42/d;
}
}
```

Output:

```
java.lang.ArithmeticException: / by zero
        at exc0.main(exc0.java:4)
```

- ✓ A new exception object is constructed and then thrown.
- ✓ This exception is caught by the default handler provided by the java runtime system.
- ✓ The default handler displays a string describing the exception, prints the stack trace from the point at which the exception occurred and terminates the program.

User Defined Exception

Define a subclass of the Exception class.

The new subclass inherits all the methods of Exception and can override them.

```
class MyException extends Exception{
MyException() {System.out.println(Age very low – Please try next year”);}
}
```

Continuation of the Example

```
class test{
    static void compute (int a) throws Myexception{
            if(a>10) throw new MyException();
            System.out.println(“Normal Exit”);
    }
public static void main(String args[]){
    try{
            compute(1);
            compute(20);
            }catch(MyException e){ System.out.println(“Caught “ +e);
}
}
```

Example-2

```
class InvalidRadiusException extends Exception {
        private double r;
        public InvalidRadiusException(double radius){
              r = radius;
         }
         public void printError(){
              System.out.println("Radius [" +  r + "] is not valid");
         }
}
```

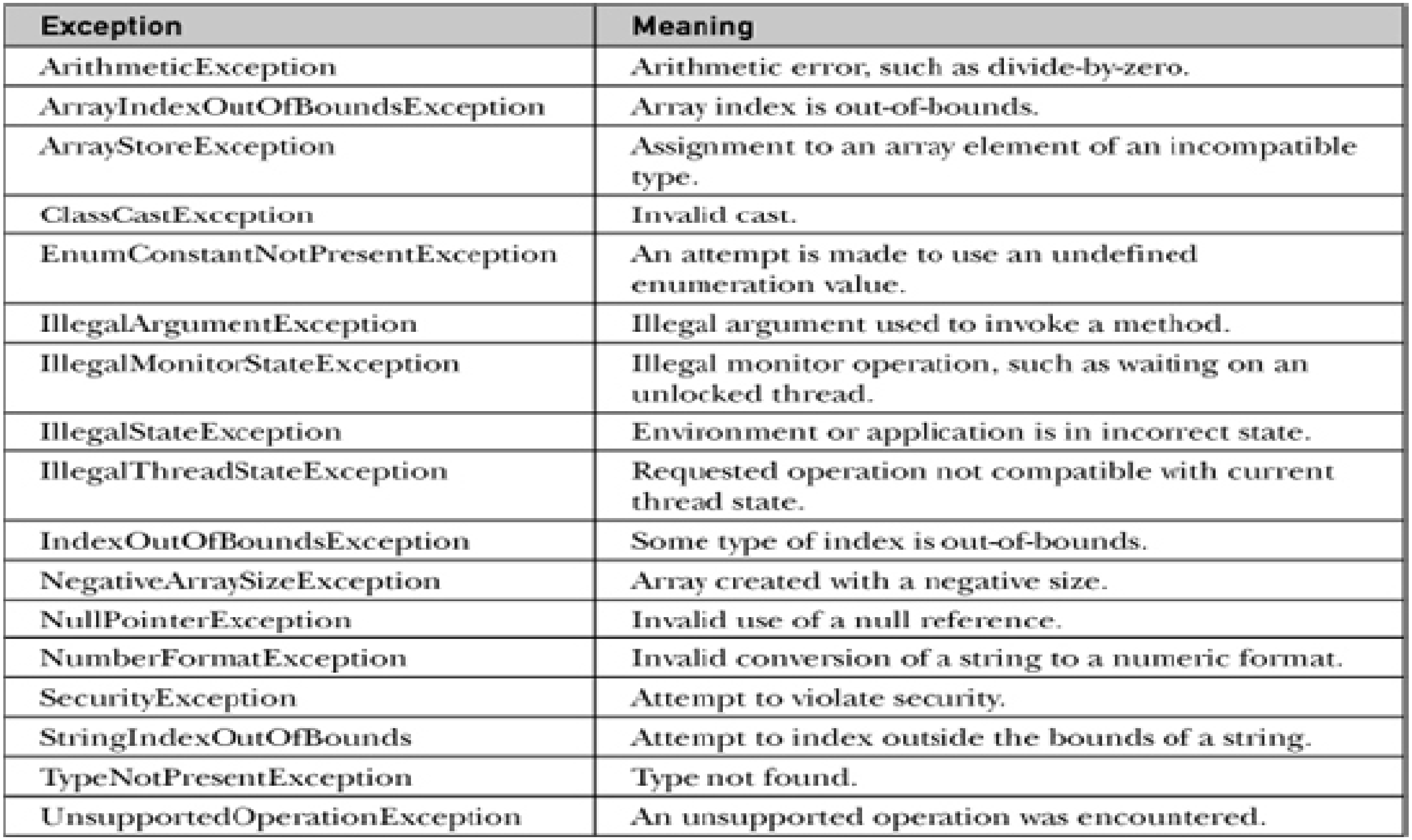
Continuation of Example-2

```
class Circle    {
        double x, y, r;

       public Circle (double centreX, double centreY, double radius ) throws
      InvalidRadiusException {
        if (r <= 0 ) {
                throw new InvalidRadiusException(radius);
        }
        else {
                x = centreX ; y = centreY;  r = radius;
        }
      }
}
```

Continuation of Example-2

```
class CircleTest {
    public static void main(String[] args){
     try{
          Circle c1 = new Circle(10, 10, -1);
          System.out.println("Circle created");
     }
     catch(InvalidRadiusException e)
     {
          e.printError();
     }
    }
}
```

Predefined Exceptions



Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Multi Threading

- Performing operations concurrently (in parallel)
  - – download a file, print a file, receive email, run the clock, more or less in parallel….
- How are these tasks typically accomplished?
    - Operating systems support processes
- What's the difference between a process and a thread?
    - **Processes have their own memory space, threads share memory**
    - **Hence processes are "heavyweight" while threads are "lightweight"**

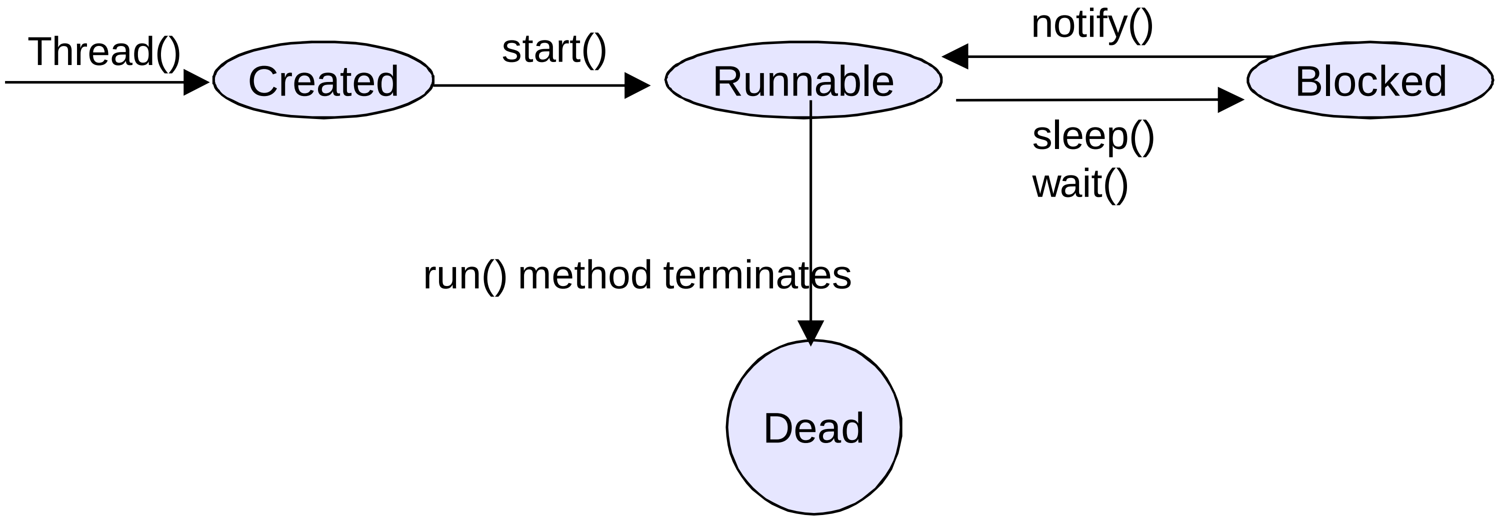
Thread States

Threads can be in one of four states

- Created, Running, Blocked, and Dead

A thread's state changes based on:

- Control methods such as start, sleep, wait, notify
- Termination of the run method



How does a Thread run?

The thread class has a run() method

run() is executed when the thread's start() method is invoked

- The thread terminates if the run method terminates
  - To prevent a thread from terminating, the run method must not end
  - run methods often have an endless loop to prevent thread termination

Creating threads

- In java threads can be created by **extending the Thread class**

  or

- **implementing the Runnable Interface**
- It is more preferred to implement the Runnable Interface so that we can extend properties from other classes
- Implement the **run()** method which is the starting point for thread execution

Thread Methods

- Thread-related methods
  - Constructors
    - **`Thread() -`** Creates a thread
    - **`Thread( threadName ) -`** Creates a thread with name
  - **`run`**
    - Does “work” of a thread
    - Can be overridden in subclass of **`Thread`** or in **`Runnable`** object
  - **`start`**
    - Launches thread, then returns to caller
    - Calls **`run`**

More Thread Methods

- **`static`** void **`sleep( long milliseconds )`**
  - Thread sleeps for number of milliseconds
- **`boolean isAlive()`**
  - Returns **`true`** if **`start`** called and thread not dead
- **`Join() – Waits for a thread to die`**
- **`getPriority()`** - returns this thread's priority
- **`setPriority()`** – sets this threads priority

Using Thread Class

```
class mythread extends Thread{
        mythread()
                {start();}
        public void run(){
                System.out.println("Thread Started");
        }
   }

class thread2
    {
        public static void main(String args[])
                {
                new mythread();
                }
    }
```

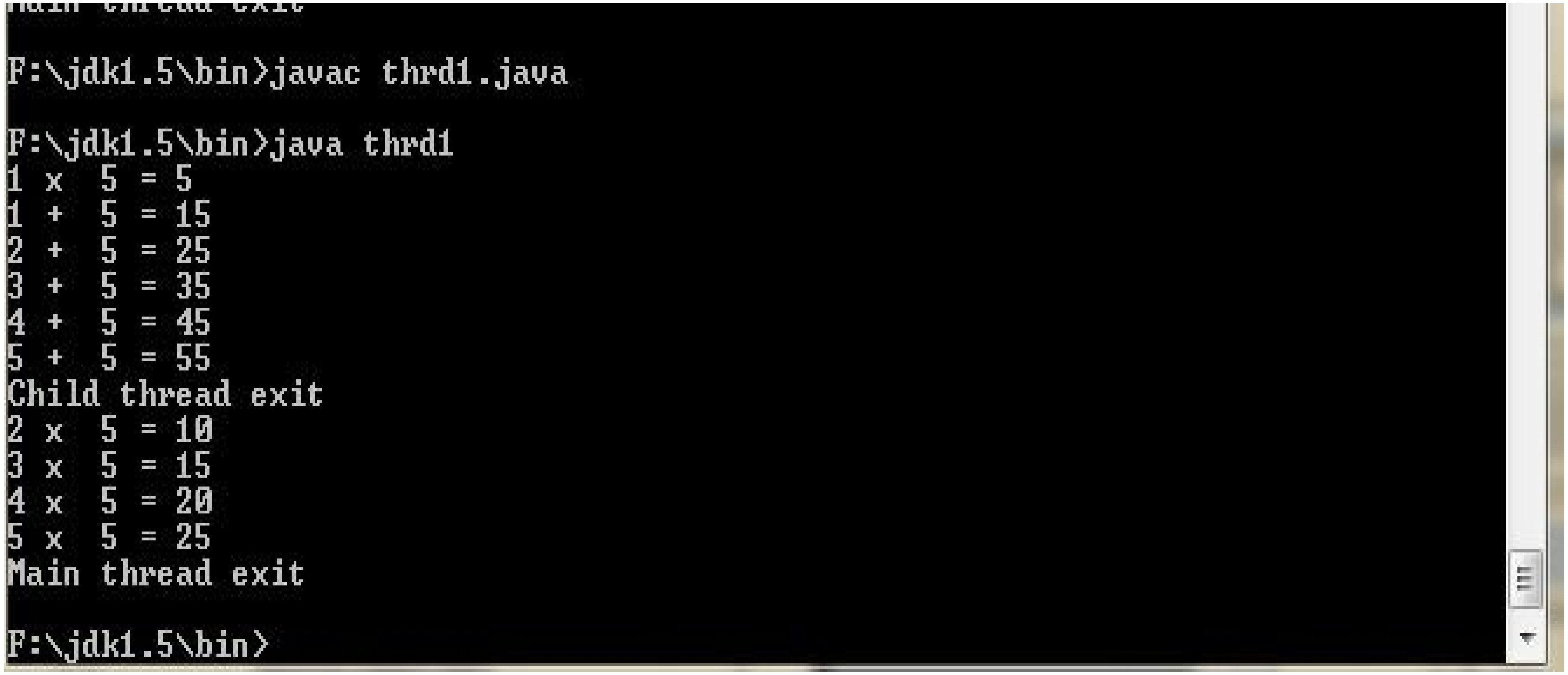
Using Runnable interface

```
class mythread implements Runnable{
        mythread()
                {
                Thread t = new Thread(this);
                t.start();
                }
        public void run(){
                System.out.println("Thread Started");
        }
   }

   class thread1 {
        public static void main(String args[]){
                new mythread();
                }
   }
```

- Calling t.run() does not start a thread, it is just a simple method call.
- Creating an object does not create a thread, calling start() method creates the thread.

```
class thrd1
	{
	public static void main(String args[])
	{
	new thtest();
	try
	{
	for (int i = 1; i<= 5; i++)
		{
		System.out.println(i + " x  5 = " + i
	*5);
		Thread.sleep(1000);
		}
	}
	catch (Exception e){}
	System.out.println("Main thread exit");
	}
	}
```

```
class thtest extends Thread

{

	thtest()
		{
		start();
		}

	public void run()
		{
		try
		{
		for (int i = 1; i<= 5; i++)
		{
		System.out.println(i + " +  5 = " + i +5);
		Thread.sleep(100);
		}
		}
		catch(InterruptedException e){
	System.out.println(e);}

		System.out.println("Child thread exit");
		}
	}
```



```
Main thread exit

F:\jdk1.5\bin>javac thrd1.java

F:\jdk1.5\bin>java thrd1
1 x  5 = 5
1 +  5 = 15
2 +  5 = 25
3 +  5 = 35
4 +  5 = 45
5 +  5 = 55
Child thread exit
2 x  5 = 10
3 x  5 = 15
4 x  5 = 20
5 x  5 = 25
Main thread exit

F:\jdk1.5\bin>
```

A clock Applet

```
import java.awt.*;
import java.applet.*;
import java.util.*;
import java.text.SimpleDateFormat;

/* <applet code=clock1.class height=300 width=300>
</applet> */


public class clock1 extends Applet implements Runnable
        {
        Thread myThread = new Thread(this, "clock");
        Label clock= new Label("");;
        Label dat = new Label(" ");
        Date d = new Date();

        public void init()
        {

        dat.setFont (new Font ("Verdana", Font.BOLD,28));
        dat.setBackground(Color.black);
        dat.setForeground(Color.white);
        add(dat);

        SimpleDateFormat df = new SimpleDateFormat("
        dd:MM:yy ");
        String dt = df.format(d);
        System.out.println(dt);
        dat.setText(dt);
```

```
clock.setFont (new Font ("Verdana", Font.BOLD,28));
        clock.setBackground(Color.blue);
        clock.setForeground(Color.white);
        add(clock);
        myThread.start();
        }


public void run()
        {
        while (true)
        {
        Date d = new Date();
        SimpleDateFormat df1 = new
        SimpleDateFormat(" hh:mm:ss ");
        String t = df1.format(d);
        clock.setText(t);
        }
        }


        }
```

Thread Priorities

- All Java applets / applications are multithreaded
  - Threads have priority from 1 to 10
    - **`Thread.MIN_PRIORITY - 1`**
    - **`Thread.NORM_PRIORITY - 5`** (default)
    - **`Thread.MAX_PRIORITY - 10`**
    - New threads inherit priority of thread that created it
- Java scheduler
  - Keeps highest-priority thread running at all times
  - New high priority threads could postpone execution of lower priority threads
- Priority methods
  - **`setPriority( int priorityNumber )`**
  - **`getPriority()`**

Thread Synchronization

- Monitors
  - Object with **`synchronized`** methods
    - Any object can be a monitor
  - Methods declared **`synchronized`**
    - **`public synchronized int myMethod( int x )`**
    - Only one thread can execute a **`synchronized method`** at a time
      - *Obtaining the lock* and *locking* an object
    - If multiple **`synchronized`** methods, only one may be active

Synchonized blocks

- Synchronized blocks of code

```
synchronized( monitorObject ){
    ...
}
```

  - **`monitorObject`**- Object to be locked while thread executes block of code

```
class sthrd
        {
        public static void main(String args[])
        {
        syn s = new syn();
        thtest t1 = new thtest(s, "Somebody");
        thtest t2 = new thtest(s, "Nobody");
        thtest t3 = new thtest(s, "Everybody");
        }
        }

class thtest implements Runnable
        {
        Thread t;
        String msg;
        syn sy;
        thtest(syn s1,  String s)
                {
                sy = s1;
                msg = s;
                t = new Thread(this,"Child");

                t.start();
                }
```

```
public void run()
                {
                sy.callme(msg);
                }
        }

class syn
        {
        synchronized void callme(String m)
        {
        System.out.println("[ " );
        System.out.println(m);
        System.out.println(" ] " );
        } }
```

Inter Thread Communication

- Thread will **wait** if it cannot proceed
  - May voluntarily call **`wait`** while accessing a **`synchronized`** method
    - Removes thread from contention for monitor object and processor
    - Thread in waiting state
  - Other threads try to enter monitor object
    - Suppose condition first thread needs has now been met
    - Can call **`notify`** to tell a single waiting thread to enter ready state
    - **`notifyAll`** - tells all waiting threads to enter ready state

Producer Consumer Problem

This is a classic example of a multi-process synchronization problem.

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

The producer's job is to generate a piece of data, put it into the buffer and start again.

At the same time, the consumer will be consuming the data one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Producer Consumer Problem

```
class produce
    {
    public static void main(String args[])
    {
    queue q = new queue();
    producer p = new producer(q);
    consumer c = new consumer(q);
    }
    }
```

Producer Consumer Problem

```
class consumer implements Runnable
        {
        queue pq;
        Thread t;
        consumer(queue q)
                {
                pq = q;
                t = new Thread(this);
                t.start();
                }
        public void run()
                {
                while (true)
                {
                try
                {
                pq.get();
                Thread.sleep(800);
                }
                catch(InterruptedException e) {
   System.out.println(e);}
        }} }
```

```
class producer implements Runnable
        {
        queue pq;
        Thread t;
        producer(queue q)
                {
                pq = q;
                t = new Thread(this);
                t.start();
                }
        public void run()
                {
                int i = 0;
                 while (true)
                {
                try
                {

                pq.put(i++); 

                Thread.sleep(1000);

                }
                catch(InterruptedException e) {
   System.out.println(e);}
        } } }
```

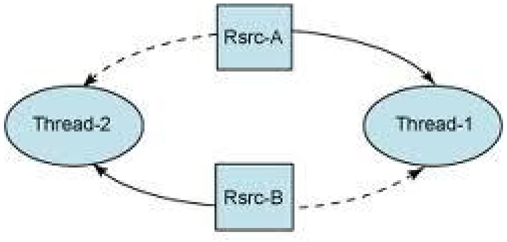
Producer Consumer Problem

```
class queue
    {
    int no;
    boolean produced=false;

    synchronized int get()
        {
        if (produced == false)
        {
        try
        {
        wait();
        }
        catch(Exception e) {
    System.out.println(e);}
        }
        System.out.println("Got " + no);
        produced = false;
        notify();
        return(no);
        }
```

```
    synchronized void put(int n)
            {
            if (produced == true)
            {
            try
            {
            wait();
            }
            catch(Exception e) {
        System.out.println(e);}
            }
            no = n;
            produced = true;
            System.out.println("put " + no);

            notify();
            }
        }
```

Deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.
- In general, it occurs only rarely, when the two threads time-slice in just the right way.



Thread-1 has Rsrc-A and is requesting B
Thread-2 has Rsrc-B and is requesting A

How to avoid deadlock?

- The simplest and most efficient way to avoid deadlock is to ensure that **resources are always acquired in some well-defined order**.
- Use timeouts to release the locks

A Situation

```
class Account { double balance;
int id;
void withdraw(double amount){ balance -= amount; } void deposit(double amount){
      balance += amount; }
void transfer(Account from, Account to, double amount)
{ sync(from);
sync(to);
 from.withdraw(amount);
to.deposit(amount);
release(to); release(from); } }
```

- **Obviously, should there be two threads which attempt to run transfer(a,b,amt) and transfer(b,a,amt) at the same time, then a deadlock is going to occur.**

Applets

- An applet is a Panel that allows interaction with a Java program
- A applet is typically embedded in a Web page and can be run from a browser
- You need special HTML in the Web page to tell the browser about the applet
- For security reasons, applets run in a sandbox: they have no access to the client's file system

Applets vs. Applications

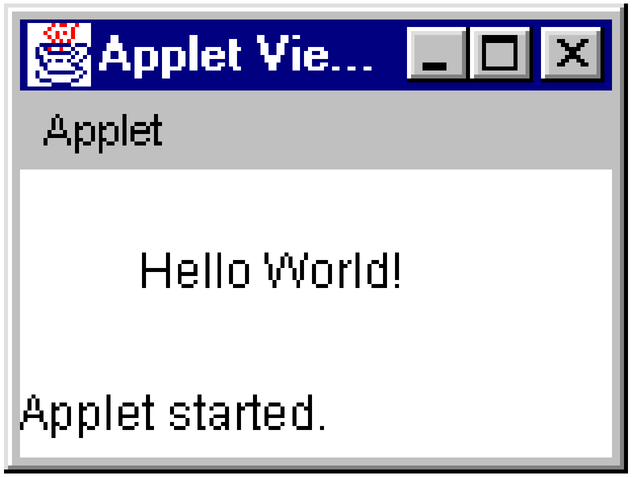
- Extend Applet instead of Frame
- Use of init( ) vs. constructor
- No use of main( )
- Default layout managers
  - application: BorderLayout
  - applet: FlowLayout
- Security – Applets run in a sand box

The genealogy of Applet

```
java.lang.Object
   |
   +----java.awt.Component
          |
          +----java.awt.Container
                 |
                 +----java.awt.Panel
                        |
                        +----java.applet.Applet
```

The simplest reasonable applet

```
import java.awt.*;
import java.applet.Applet;

public class HelloWorld extends Applet {
   public void paint( Graphics g ) {
      g.drawString( "Hello World!", 30, 30 );
   }
}
```



Hello World!

Applet started.

Applet methods

public void init ()

public void start ()

public void stop ()

public void destroy ()

public void paint (Graphics)

Also:

public void repaint()

public void update (Graphics)

public void showStatus(String)

public String getParameter(String)

Why an applet works

- write an applet by extending the class Applet
- Applet defines methods init( ), start( ), stop( ), paint(Graphics), destroy( )
- These methods do nothing--they are stubs
- Make the applet do something by overriding these methods

public void init ( )

- This is the first method to be executed
- It is an ideal place to initialize variables
- It is the best place to define the GUI Components (buttons, text fields, scrollbars, etc.), lay them out, and add listeners to them

public void start ( )

- Not always needed
- Called after init( )
- Called each time the page is loaded and restarted
- Used mostly in conjunction with stop( )
- start() and stop( ) are used when the Applet is doing time-consuming calculations that you don’t want to continue when the page is not in front

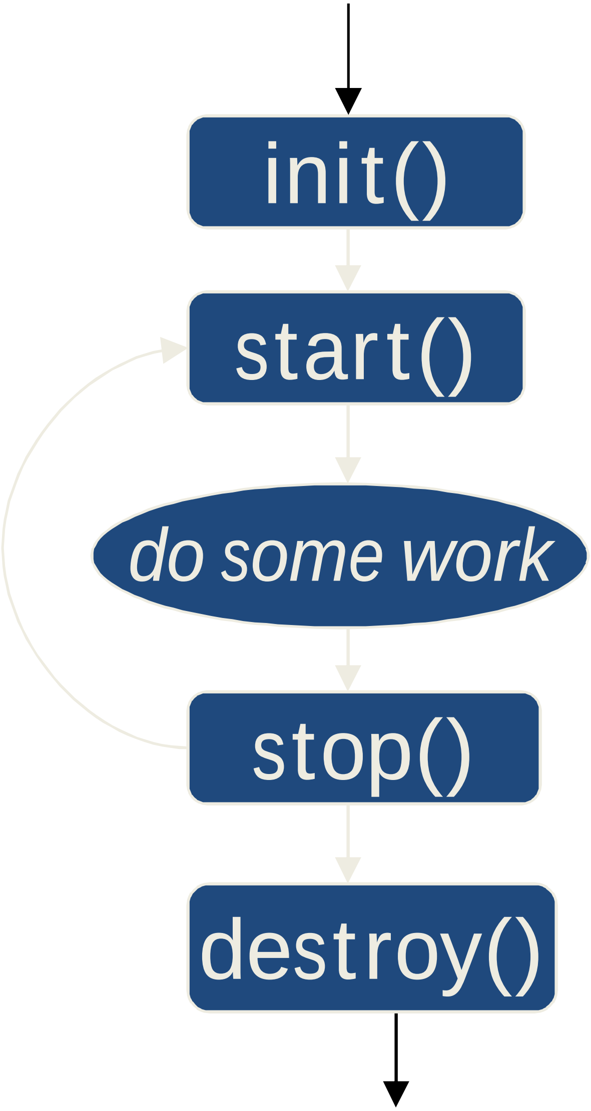
public void stop( )

- Not always needed
- Called when the browser leaves the page
- Called just before destroy( )
- Use stop( ) if the applet is doing heavy computation that you don’t want to continue when the browser is on some other page
- Used mostly in conjunction with start()

public void destroy( )

- Seldom needed
- Called after stop( )
- Use to explicitly release system resources (like threads)
- System resources are usually released automatically

Applet Life Cycle



- init and destroy are only called once each
- start and stop are called whenever the browser enters and leaves the page
- do some work is code called by your listeners
- paint is called when the applet needs to be repainted

public void paint(Graphics g)

- Needed if you do any drawing or painting other than just using standard GUI Components
- Any painting you want to do should be done here, or in a method you call from here
- Painting that you do in other methods may or may not happen
- Never call paint(Graphics), call repaint( )

repaint( )

- Call repaint( ) when you have changed something and want your changes to show up on the screen
- repaint( ) is a request--it might not happen
- When you call repaint( ), Java schedules a call to update(Graphics g)

update( )

- When you call repaint( ), Java schedules a call to update(Graphics g)
- Here's what update does:

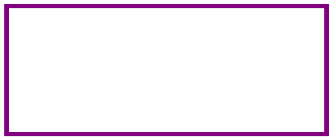
```
public void update(Graphics g) {
    // Fills applet with background color, then
    paint(g);
}
```

Using Colors

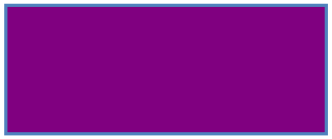
```
    public void paint(Graphics g) {
        g.setColor(Color.BLUE);
        ...draw a rectangle...
        g.setColor(Color.RED);
        ...draw another rectangle...
    }
}
```

Drawing rectangles

- There are two ways to draw rectangles:
- g.drawRect(***left*** , ***top*** , ***width*** , ***height***);



- g.fillRect(***left*** , ***top*** , ***width*** , ***height***);



The complete applet

```
import java.applet.Applet;
import java.awt.*;


public class Drawing extends Applet {

    public void paint(Graphics g) {

        g.setColor(Color.BLUE);
        g.fillRect(20, 20, 50, 30);
        g.setColor(Color.RED);
        g.fillRect(50, 30, 50, 30);
    }
}
```

Some more methods

- g.drawLine(***x1*** , ***y1*** , ***x2*** , ***y2***);
- g.drawOval(***left*** , ***top*** , ***width*** , ***height***);
- g.fillOval(***left*** , ***top*** , ***width*** , ***height***);
- g.drawArc(***left*** , ***top*** , ***width*** , ***height*** ,
  startAngle , ***arcAngle***);
- g.drawString(***string*** , ***x*** , ***y***);

Viewing the applet

- Applet viewer - Write the comment block as given below.

```
     import java.applet.Applet;
     import java.awt.*;

public class drawing extends Applet {
  public void paint(Graphics g) {
    g.setColor(Color.BLUE);
    g.fillRect(20, 20, 50, 30);
    g.setColor(Color.RED);
    g.fillRect(50, 30, 50, 30);
    }
    }
/*
<applet code=drawing width=300 height=300>
</applet>
*/
```

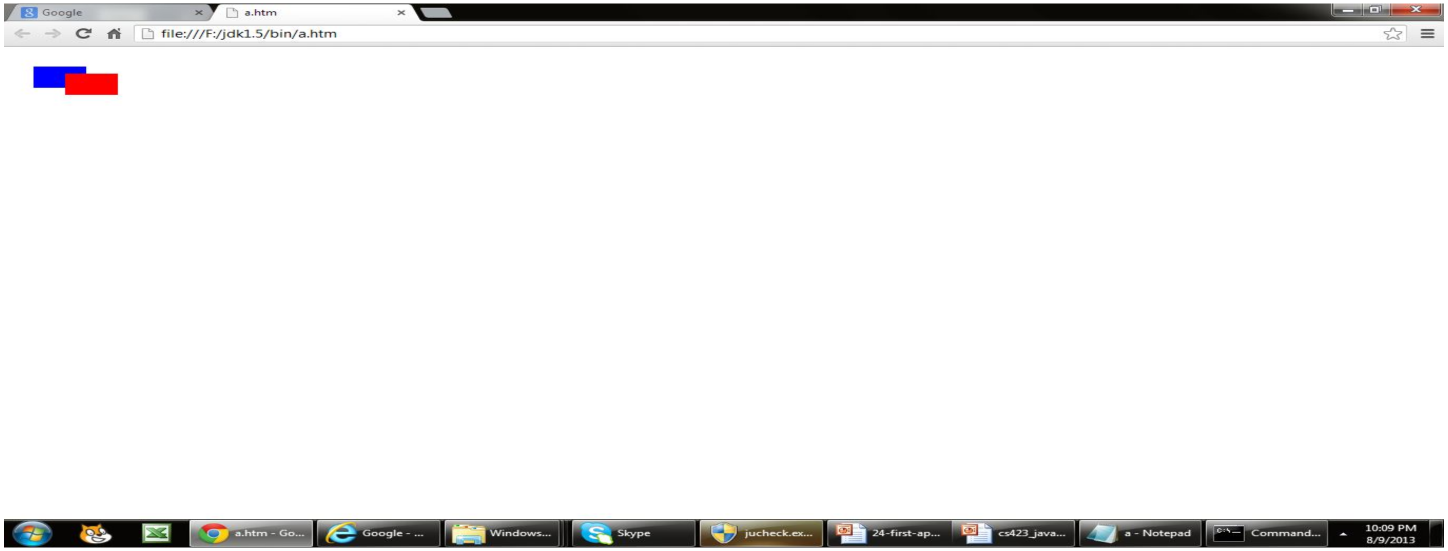
- HTML Page [Refer next slide]

The HTML page

- You can run an applet in an HTML page
- The HTML looks something like this:

  -
    ```
    <html>
        <body>
            <h1>DrawingApplet Applet</h1>
            <applet code="Drawing.class"
                                width="250"  height="200">
            </applet>
        </body>
    </html>
    ```

Output


Google
a.htm
file:///F:/jdk1.5/bin/a.htm
a.htm - Go...
Google - ...
Windows...
Skype
jucheck.ex...
24-first-ap...
cs423_java...
a - Notepad
Command...
10:09 PM
8/9/2013

AWT (Abstract Windowing Toolkit)

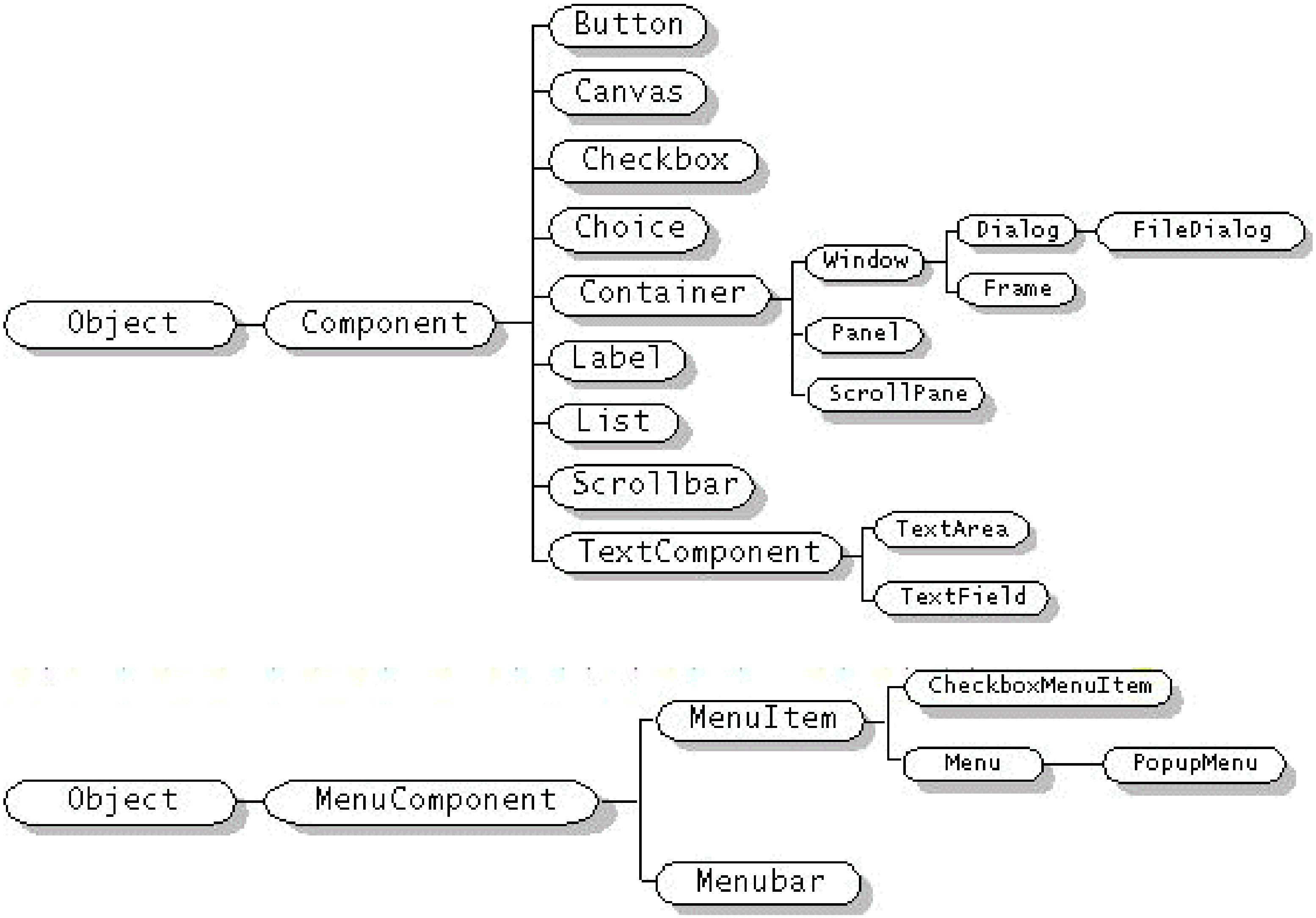
The AWT is roughly broken into three categories

- Components
- Layout Managers
- Graphics

Many AWT components have been replaced by Swing components

It is generally not considered a good idea to mix Swing components and AWT components. Choose to use one or the other.

AWT Class Hierarchy



Component

Component is the superclass of most of the displayable classes defined within the AWT. Note: it is abstract.

MenuComponent is another class which is similar to Component except it is the superclass for all GUI items which can be displayed within a drop-down menu.

The Component class defines data and methods which are relevant to all Components

setBounds
setSize
setLocation
setFont
setEnabled
setVisible
setForeground -- colour
setBackground -- colour

Container

Container is a subclass of Component. (ie. All containers are themselves, Components)

Containers contain components

For a component to be placed on the screen, it must be placed within a Container

The Container class defined all the data and methods necessary for managing groups of Components

- add
- getComponent
- getMaximumSize
- getMinimumSize
- getPreferredSize
- remove
- removeAll

Windows and Frames

The Window class defines a top-level Window with no Borders or Menu bar.

Usually used for application screens

- Frame defines a top-level Window with Borders and a Menu Bar
  - Frames are more commonly used than Windows

Once defined, a Frame is a Container which can contain Components

```
Frame aFrame = new Frame(“Hello World”);
aFrame.setSize(100,100);
aFrame.setLocation(10,10);
aFrame.setVisible(true);
```

Frame Example

```
import java.awt.*;

public class win1
	{
	public static void main(String args[])
	{
	Frame f = new Frame();



	f.setSize(300,300);
	f.setVisible(true);
	}
	}
```

Panels

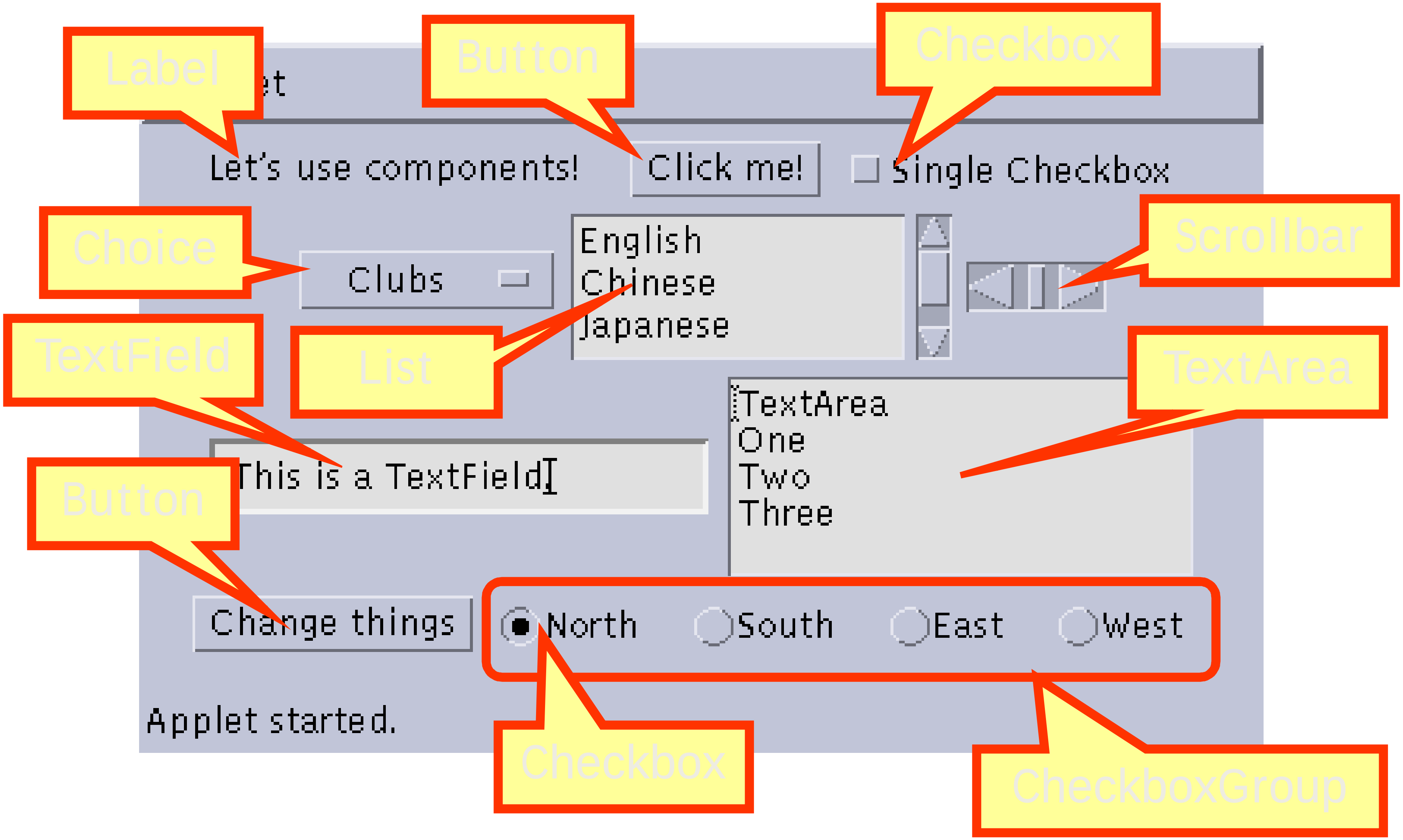
When writing a GUI application, the GUI portion can become quite complex.

To manage the complexity, GUIs are broken down into groups of components. Each group generally provides a unit of functionality.

A Panel is a rectangular Container whose sole purpose is to hold and manage components within a GUI.

```
Panel aPanel = new Panel();
aPanel.add(new Button("Ok"));
aPanel.add(new Button("Cancel"));

Frame aFrame = new Frame("Button Test");
aFrame.setSize(100,100);
aFrame.setLocation(10,10);

aFrame.add(aPanel);
```

Some types of components



Applet started.

Buttons

This class represents a push-button which displays some specified text.

When a button is pressed, it notifies its Listeners. (More about Listeners in the next chapter).

To be a Listener for a button, an object must implement the ActionListener Interface.

```
Panel aPanel = new Panel();
Button okButton = new Button("Ok");
Button cancelButton = new Button("Cancel");

aPanel.add(okButton));
aPanel.add(cancelButton));
```

Labels

This class is a Component which displays a single line of text.

Labels are read-only. That is, the user cannot click on a label to edit the text it displays.

Text can be aligned within the label

```
Label aLabel = new Label("Enter password:");
aLabel.setAlignment(Label.RIGHT);

aPanel.add(aLabel);
```

List

This class is a Component which displays a list of Strings.

The list is scrollable, if necessary.

Sometimes called Listbox in other languages.

Lists can be set up to allow single or multiple selections.

The list will return an array indicating which Strings are selected

```
List aList = new List();
aList.add("Calgary");
aList.add("Edmonton");
aList.add("Regina");
aList.add("Vancouver");

aList.setMultipleMode(true);
```

Checkbox

This class represents a GUI checkbox with a textual label.

The Checkbox maintains a boolean state indicating whether it is checked or not.

If a Checkbox is added to a CheckBoxGroup, it will behave like a radio button.

```
Checkbox creamCheckbox = new Checkbox("Cream");
Checkbox sugarCheckbox = new Checkbox("Sugar");
```

Radio Buttons

```
CheckboxGroup cb = new CheckboxGroup();
  Checkbox bf1 = new Checkbox("Idly", cb,true);
  Checkbox bf2= new Checkbox("Dosa",cb,false);
  Checkbox bf3= new Checkbox("Pongal",cb,false);
  Checkbox bf4= new Checkbox("Poori",cb,false);
```

Choice

This class represents a dropdown list of Strings.

Similar to a list in terms of functionality, but displayed differently.

Only one item from the list can be selected at one time and the currently selected element is displayed.

```
Choice aChoice = new Choice();
aChoice.add("Calgary");
aChoice.add("Edmonton");
aChoice.add("Alert Bay");
```

TextField

This class displays a single line of optionally editable text.

This class inherits several methods from TextComponent.

This is one of the most commonly used Components in the AWT

```
TextField emailTextField = new TextField();
TextField passwordTextField = new TextField();
passwordTextField.setEchoChar("*");
[…]

String userEmail = emailTextField.getText();
String userpassword = passwordTextField.getText();
```

TextArea

This class displays multiple lines of optionally editable text.

This class inherits several methods from TextComponent.

TextArea also provides the methods: appendText(), insertText() and replaceText()

```
// 5 rows, 80 columns
TextArea fullAddressTextArea = new TextArea(5, 80);
```

Using Menu

```
Frame f = new Frame();
MenuBar m = new MenuBar();
Menu colr = new Menu("Colour");
MenuItem c1 = new MenuItem("Red");
MenuItem c2 = new MenuItem("Orange");
MenuItem c3 = new MenuItem("Blue");
MenuItem c4 = new MenuItem("Cyan");
f.setTitle("Menu Test Window");
f.setSize(500,500);
f.setVisible(true);
colr.add(c1);   colr.add(c2);       colr.add(c3);       colr.add(c4);
m.add(colr);
f.setMenuBar(m);
```

Layout Managers

Since the Component class defines the setSize() and setLocation() methods, all Components can be sized and positioned with those methods.

Problem: the parameters provided to those methods are defined in terms of pixels. Pixel sizes may be different (depending on the platform) so the use of those methods tends to produce GUIs which will not display properly on all platforms.

Solution: Layout Managers. Layout managers are assigned to Containers. When a Component is added to a Container, its Layout Manager is consulted in order to determine the size and placement of the Component.

Layout Managers (cont)

There are several different LayoutManagers, each of which sizes and positions its Components based on an algorithm:

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout

For Windows and Frames, the default LayoutManager is BorderLayout. For Panels, the default LayoutManager is FlowLayout.

Flow Layout

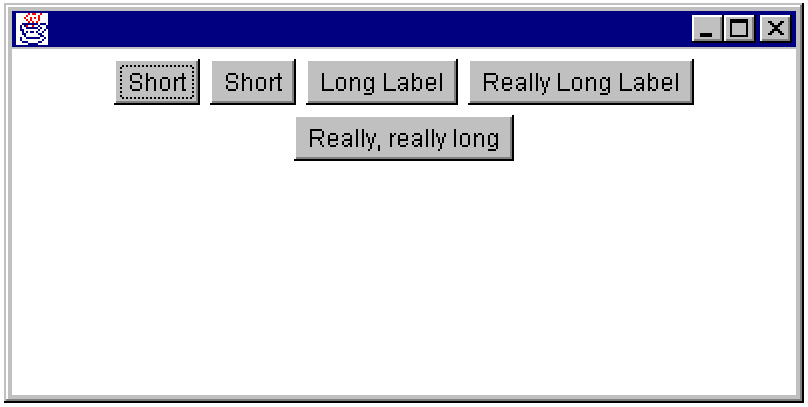
The algorithm used by the FlowLayout is to lay out Components like words on a page: Left to right, top to bottom.

It fits as many Components into a given row before moving to the next row.

```
Panel aPanel = new Panel(new Layout(FlowLayout());
aPanel.add(new Button("Ok"));
aPanel.add(new Button("Add"));
aPanel.add(new Button("Delete"));
aPanel.add(new Button("Cancel"));
```

Example

```
String Labels[] = {"Short", "Short", "Long Label",
                    "Really Long Label", "Really,
really long"};

setLayout(new FlowLayout());
for (int i = 0; i < Labels.length; i++){
{Button temp = new Button (Labels[i]);
add (temp);
}
```



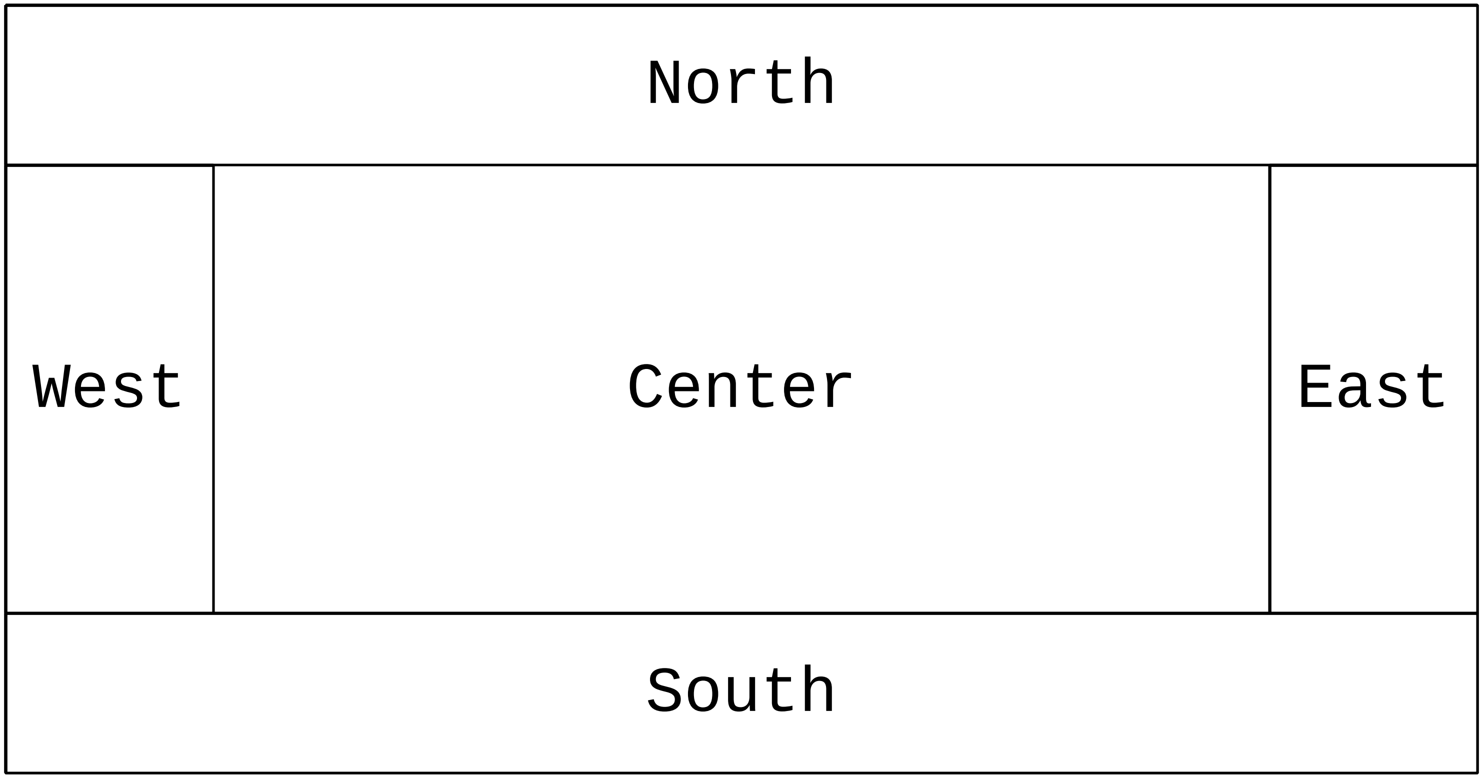
Border Layout

The BorderLayout Manager breaks the Container up into 5 regions (North, South, East, West, and Center).

When Components are added, their region is also specified:

Border Layout (cont)

The regions of the BorderLayout are defined as follows:



North

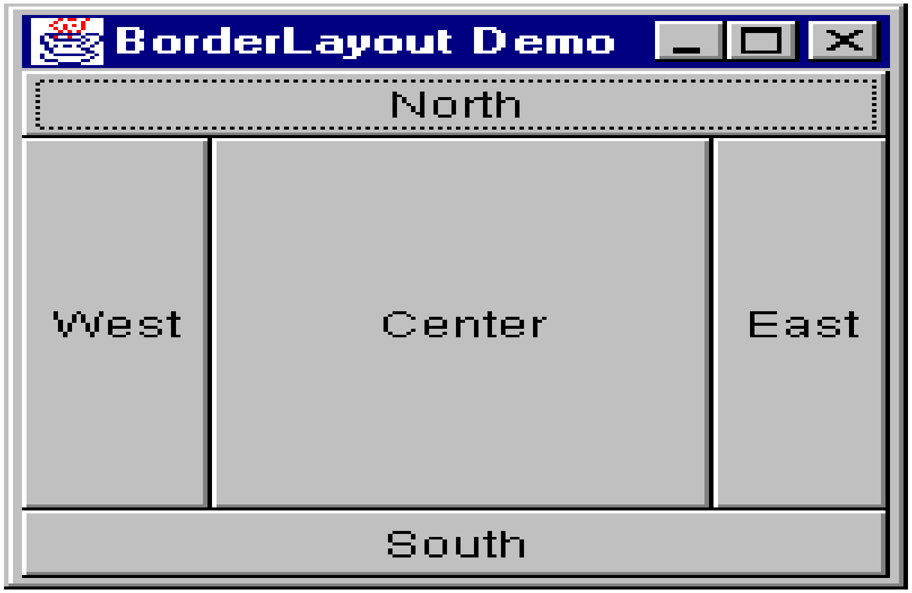
West

Center

East

South

```
setLayout(new BorderLauout());
add (new Button ("North"), BorderLayout.NORTH);
add (new Button ("South"), BorderLayout.SOUTH);
add (new Button ("East"), BorderLayout.EAST);
add (new Button ("West"), BorderLayout.WEST);
add (new Button ("Center"), BorderLayout.CENTER);
```



Grid Layout

The GridLayout class divides the region into a grid of equally sized rows and columns.

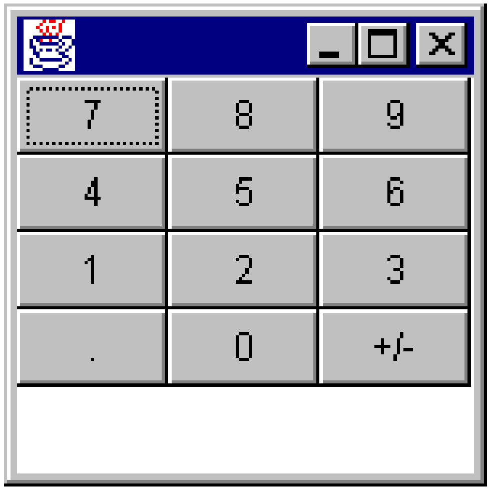
Components are added left-to-right, top-to-bottom.

The number of rows and columns is specified in the constructor for the LayoutManager.

```
Panel aPanel = new Panel();
GridLayout theLayout = new GridLayout(2,2);
aPanel.setLayout(theLayout);

aPanel.add(new Button("Ok"));
aPanel.add(new Button("Add"));
aPanel.add(new Button("Delete"));
aPanel.add(new Button("Cancel"));
```

Example

```
setLayout(new GridLayout(5,3));
       add (new Button ("7"));
       add (new Button ("8"));
       add (new Button ("9"));
       add (new Button ("4"));
       add (new Button ("5"));
       add (new Button ("6"));
       add (new Button ("1"));
       add (new Button ("2"));
       add (new Button ("3"));
       add (new Button ("."));
       add (new Button ("0"));
       add (new Button ("+/-"));
```



GridBag Layout

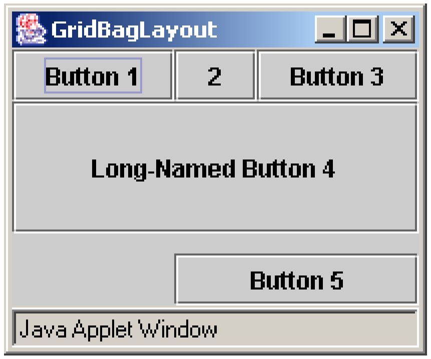
```
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();

JPanel pane = new JPanel();
pane.setLayout(gridbag);

//For each component to be added to this container:
//...Create the component...
//...Set instance variables in the GridBagConstraints instance...

        gridbag.setConstraints(theComponent, c);
        pane.add(theComponent);
```

GridBag Constraints Instance Variables

- gridx, gridy (location in base grid)
- gridwidth, gridheight (no. of cols, or rows)
- fill (extra space)
- ipadx, ipady (internal spacing)
- insets (external spacing via insets object)
- weightx, weighty (relative size among cols or rows when resized)

GridBag Layout Example

```
Button button;

GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
setLayout(gridbag);

button = new Button("Button 1");
c.weightx = 0.5;
c.gridx = 0;
c.gridy = 0;
gridbag.setConstraints(button, c);
add(button);

button = new Button("2");
c.gridx = 1;
c.gridy = 0;
gridbag.setConstraints(button, c);
add(button);
```



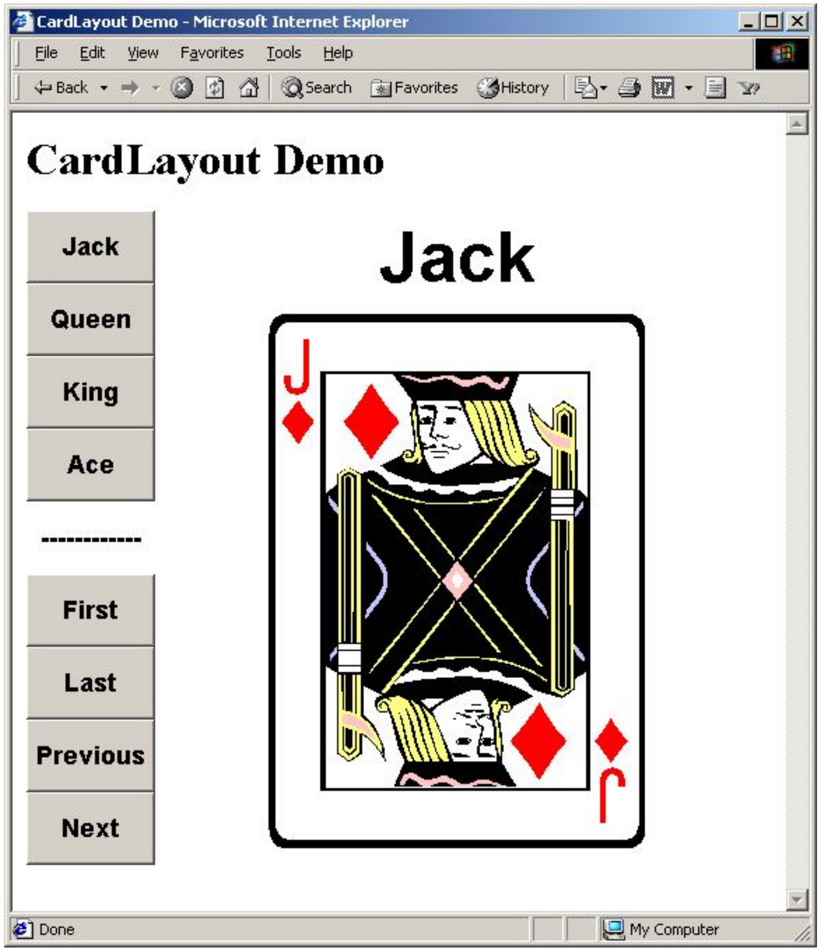
```
.
.
.
button = new  Button("Long-Named Button 4");
c.ipady = 40; //make this component tall
c.weightx = 0.0;
c.gridwidth = 3;
c.gridx = 0;
c.gridy = 1;
gridbag.setConstraints(button, c);
add(button);
.
.
.
```

Card Layout

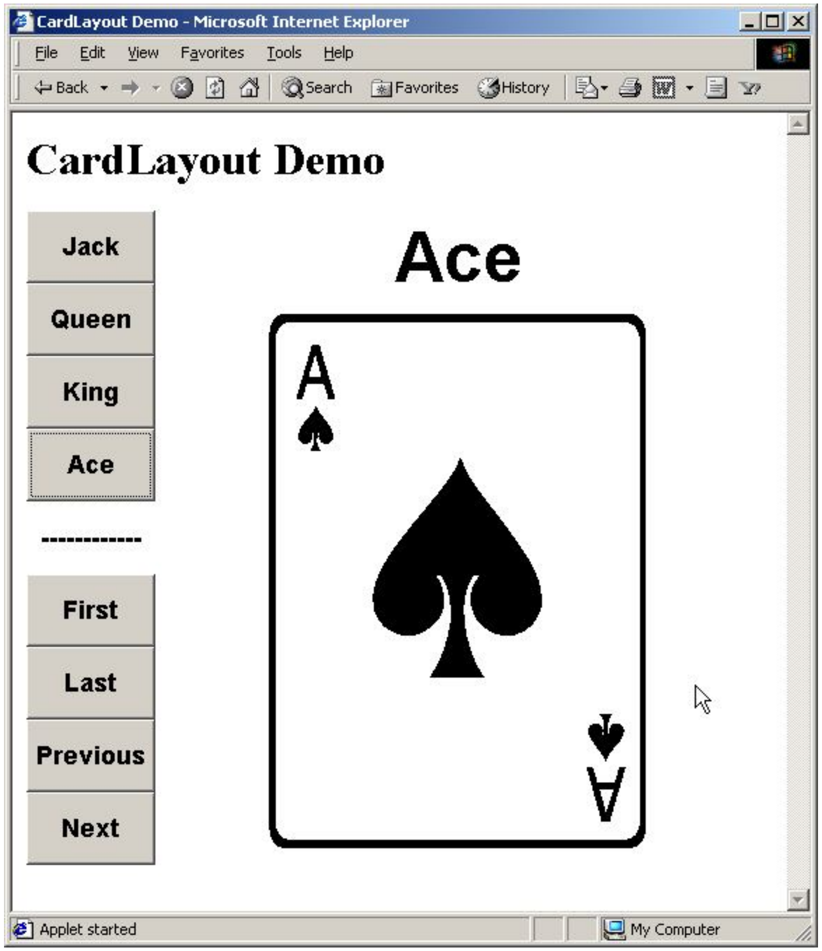
- Stacks components on top of each other, displaying the top one
- Associates a name with each component in window

```
Panel cardPanel;
CardLayout layout new CardLayout();
cardPanel.setLayout(layout);
...
cardPanel.add("Card 1", component1);
cardPanel.add("Card 2", component2);
...
layout.show(cardPanel, "Card 1");
layout.first(cardPanel);
layout.next(cardPanel);
```

Example


CardLayout Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
CardLayout Demo
Jack
Queen
King
Ace

First
Last
Previous
Next
Jack
Done
My Computer


CardLayout Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
CardLayout Demo
Jack
Queen
King
Ace

First
Last
Previous
Next
Ace
Applet started
My Computer

What if I don’t want a LayoutManager?

LayoutManagers have proved to be difficult and frustrating to deal with.

The LayoutManager can be removed from a Container by invoking its setLayout method with a null parameter.

```
Panel aPanel = new Panel();
aPanel.setLayout(null);
```

Example

```
import java.awt.*;
import java.applet.Applet;
public class gridapplet2 extends Applet
        {
        public void init()
        {
        Panel p= new Panel();
        Panel p2= new Panel();
        Panel p3= new Panel();
        p.setLayout(new GridLayout(4,2));
        p2.setLayout(new GridLayout(3,4));
        Label l = new Label("Name");
        TextField t = new TextField(30);
        Label l1 = new Label("Password");
        TextField t1 = new TextField(30);
        t1.setEchoChar('*');
        Label l2 = new Label("Likes");
        Checkbox c1 = new Checkbox("Sugar");
        Checkbox c2= new Checkbox("Ice Cream");
        Checkbox c3= new Checkbox("Coffee");
        Label l3 = new Label("Breakfast");
        CheckboxGroup cb = new CheckboxGroup();
        Checkbox bf1 = new Checkbox("Idly", cb,true);
        Checkbox bf2= new Checkbox("Dosa",cb,false);
        Checkbox bf3= new Checkbox("Pongal",cb,false);
        Button b1 = new Button("Button1");
        Button b2 = new Button("Click ME");
        Label l4 = new Label("Mobiles");
        List lt = new List(2,true);
        lt.add("Nokia");
        lt.add("Samsung");
        lt.add("Blackberry");
        lt.add("apple");
                    Label l5 = new Label("Jewels");
        Choice c = new Choice();
        c.add("Diamond");
        c.add("Platinum");
        c.add("Gold");
        c.add("Silver");
        Label l6 = new Label("Comments");
        TextArea tx = new TextArea(5,10);
        p.add(l);
        p.add(t);
        p.add(l1);
        p.add(t1);
        p.add(l4);
        p.add(lt);
        p.add(l5);
        p.add(c);
        p2.add(l2);
        p2.add(c1);
        p2.add(c2);
        p2.add(c3);
        p2.add(l3);
        p2.add(bf1);
        p2.add(bf2);
        p2.add(bf3);
        p2.add(l6);
        p2.add(tx);
        p2.add(b1);
        p2.add(b2);            add(p);          add(p2);
        }}

/*<applet code=gridapplet2 height=400 width=400>
</applet> */
```

What is an Event?

- GUI components communicate with the rest of the applications through events.
- The source of an event is the component that causes that event to occur.
- The listener of an event is an object that receives the event and processes it appropriately.

Handling Events

- Every time the user types a character or clicks the mouse, an event occurs.
- Any object can be notified of any particular event.
- To be notified for an event,
  - The object has to be registered as an event listener on the appropriate event source.
  - The object has to implement the appropriate interface.

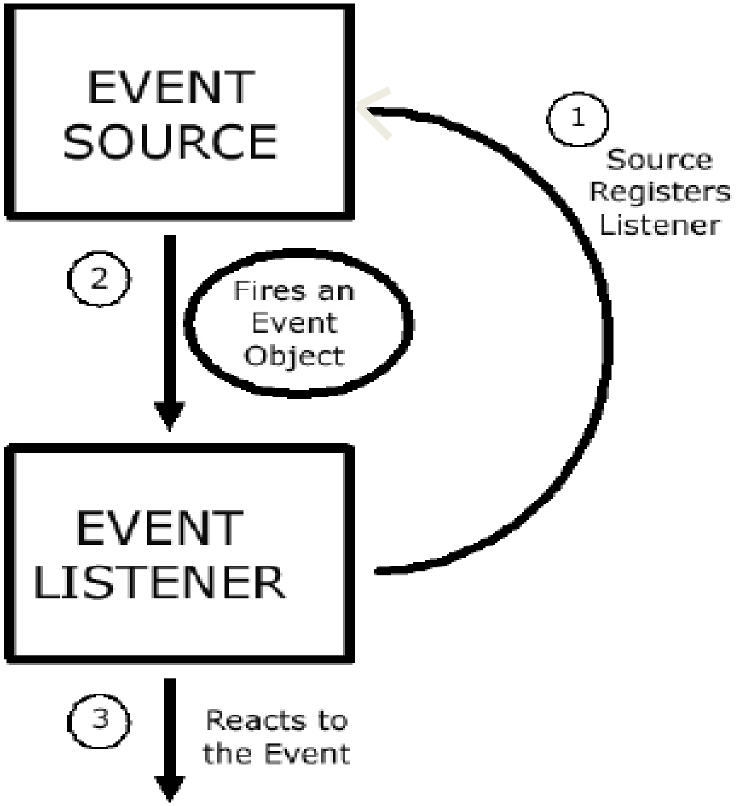
An example of Event Handling

```
import java.awt.event.*;

public class eve implements ActionListener {
  ...
  Button button = new Button("I'm a button!");
  button.addActionListener(this);
  ....
  public void actionPerformed(ActionEvent e) {
  numClicks++;
  label.setText(Integer.toString(numClicks));
  }
}
```

The Event Handling process

- When an event is triggered, the JAVA runtime first determines its source and type.
- If a listener for this type of event is registered with the source, an event object is created.
- For each listener to this type of an event, the JAVA runtime invokes the appropriate event handling method to the listener and passes the event object as the parameter.

The Event Handling Process (contd..)



What does an Event Handler require?

- It just looks for 3 pieces of code!
- First, in the declaration of the event handler class, one line of code must specify that the class implements either a listener interface or extends a class that implements a listener interface.

```
public class DemoClass implements
ActionListener {
```

What does an Event Handler require? (contd..)

- Second, it looks for a line of code which registers an instance of the event handler class as a listener of one or more components because, as mentioned earlier, the object must be registered as an event listener.

```
anyComponent.addActionListener(instanceOf
DemoClass);
```

What does an Event Handler require? (contd..)

- Third, the event handler must have a piece of code that implements the methods in the listener interface.

```
public void actionPerformed(ActionEvent e)
 {
  ...//code that reacts to the action...
 }
```

Types of Events

- Below, are some of the many kinds of events, awt components generate.

Act causing Event	Listener Type
User clicks a button, presses Enter, typing in text field	ActionListener
User closes a frame	WindowListener
User selects an item in checkbox	ItemListener
Clicking a mouse button, while the cursor is over a component	MouseListener

Types of Events (contd..)

Act causing Event	**Listener Type**
User moving the mouse over a component	MouseMotionListener
Component becomes visible	ComponentListener
Table or list selection changes	ListSelectionListener

AWT Events

The following is a list of events in the java.awt.event package:

ActionEvent	- Action has occurred (eg. button pressed)
AdjustmentEvent	- "Adjustable" Component changed
ComponentEvent	- Component Changed
ContainerEvent	- Container changed (add or remove)
FocusEvent	- Focus Changed
HierarchyEvent	- Change in Component Hierarchy
InputEvent	- Superclass of KeyEvent and MouseEvent
InputMethodEvent	- Text Input Events
ItemEvent	- Item Selected or Deselected
KeyEvent	- Keyboard event
MouseEvent	- Mouse event
PaintEvent	- Low level; do not use.
TextEvent	- Text Changed events
WindowEvent	- Window related Events

Event Listeners

- Event listeners are the classes that implement the

  <type>Listener interfaces.

  Example:

  1. ActionListener receives action events

  2. MouseListener receives mouse events.

  The following slides give you a brief overview on some of the listener types.

Listener Interfaces

The Following events have Listeners associated with them:

ActionEvent	- ActionListener
AdjustmentEvent	- AdjustmentListener
ComponentEvent	- ComponentListener
ContainerEvent	- ContainerListener
FocusEvent	- FocusListener
HierarchyEvent	- HierarchyListener
InputMethodEvent	- InputMethodListener
ItemEvent	- ItemListener
KeyEvent	- KeyListener
MouseEvent	- MouseListener
	- MouseMotionListener
TextEvent	- TextListener
WindowEvent	- WindowListener

The ActionListener Method

- It contains exactly one method.

  Example:

  ```
  public void actionPerformed(ActionEvent e)
  ```

  The above code contains the handler for the ActionEvent e that occurred.

The MouseListener Methods

- Event handling when the mouse is clicked.

  ```
  public void mouseClicked(MouseEvent e)
  ```

- Event handling when the mouse enters a component.

  ```
  public void mouseEntered(MouseEvent e)
  ```

- Event handling when the mouse exits a component.

  ```
  public void mouseExited(MouseEvent e)
  ```

The MouseListener Methods (contd..)

- Event handling when the mouse button is pressed on a component.

```
public void mousePressed(MouseEvent e)
```

- Event handling when the mouse button is released on a component.

```
public void mouseReleased(MouseEvent e)
```

The MouseMotionListener Methods

- Invoked when the mouse button is pressed over a component and dragged. Called several times as the mouse is dragged

  ```
  public void mouseDragged(MouseEvent e)
  ```

- Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

  ```
  public void mouseMoved(MouseEvent e)
  ```

The WindowListener Methods

- Invoked when the window object is opened.

  ```
  public void windowOpened(WindowEvent e)
  ```

- Invoked when the user attempts to close the window object from the object's system menu.

  ```
  public void windowClosing(WindowEvent e)
  ```

The WindowListener Methods (contd..)

- Invoked when the window object is closed as a result of calling dispose (release of resources used by the source).

```
public void windowClosed(WindowEvent e)
```

- Invoked when the window is set to be the active window.

```
public void windowActivated(WindowEvent e)
```

The WindowListener Methods (contd..)

- Invoked when the window object is no longer the active window

  ```
  public void windowDeactivated(WindowEvent e)
  ```

- Invoked when the window is minimized.

  ```
  public void windowIconified(WindowEvent e)
  ```

- Invoked when the window is changed from the minimized state to the normal state.

  ```
  public void windowDeconified(WindowEvent e)
  ```

The ItemListener Methods

- Invoked when a checkbox/checkboxmenu item object is selected

```
public void ItemStateChanged(ItemEvent e)
```

The KeyListener Methods

- Invoked when a key is pressed/released/typed

```
public void KeyPressed(KeyEvent e)
public void KeyReleased(KeyEvent e)
public void KeyTyped(KeyEvent e)
```

Adapter classes for Event Handling.

- Why do you need adapter classes?
  - Implementing all the methods of an interface involves a lot of work.
  - If you are interested in only using some methods of the interface.
- Adapter classes
  - Built-in in JAVA
  - Implement all the methods of each listener interface with more than one method.
  - Implementation of all empty methods

Adapter classes - an Illustration.

- Consider, you create a class that implements a MouseListener interface, where you require only a couple of methods to be implemented. If your class directly implements the MouseListener, you *must* implement all five methods of this interface.
- Methods for those events you don't care about can have empty bodies

Illustration (contd..)

```
public class MyClass implements MouseListener {
  ... someObject.addMouseListener(this);
  /* Empty method definition. */
  public void mousePressed(MouseEvent e) { }
  /* Empty method definition. */
  public void mouseReleased(MouseEvent e) { }
  /* Empty method definition. */
  public void mouseEntered(MouseEvent e) { }
  /* Empty method definition. */
  public void mouseExited(MouseEvent e) { }
  public void mouseClicked(MouseEvent e) {
  //Event listener implementation goes here...
  }
}
```

Illustration (contd..)

- What is the result?
  - The resulting collection of empty bodies can make the code harder to read and maintain.
- To help you avoid implementing empty bodies, the API generally includes an *adapter* class for each listener interface with more than one method. For example, the MouseAdapter class implements the MouseListener interface.

Adapter Classes

ComponentAdapter

ContainerAdapter

FocusAdapter

KeyAdapter

MouseAdapter

MouseMotionAdapter

WindowAdapter

How to use an Adapter class?

```
/* Using an adapter class
 */
 public class MyClass extends MouseAdapter {
 ....
   someObject.addMouseListener(this);
 ....
   public void mouseClicked(MouseEvent e) {
     ...//Event listener implementation goes
        // here
   }
 }
```

Using Inner classes for Event Handling

- Consider that you want to use an adapter class but you don’t want your public class to inherit from the adapter class.
- For example, you write an applet with some code to handle mouse events. As you know, JAVA does not permit multiple inheritance and hence your class cannot extend both the Applet and MouseAdapter classes.

Using Inner classes

- Use a class inside your Applet subclass that extends the MouseAdapter class.

```
public class MyClass extends Applet {
 ...
 someObject.addMouseListener(new
                                    MyAdapter());
 ...
  class MyAdapter extends MouseAdapter {
 public void mouseClicked(MouseEvent e) {
//Event listener implementation here...
 }
  }
}
```

Anonymous Inner classes

Adapter class can be anonymous (without a name) as illustrated below.

```
public class extends Applet
{
……..
f.addWindowListener(new WindowAdapter()
{
public void windowClosing (WindowEvent e)
          { System.exit(0); }
    });
}
```

Example

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class eve extends Applet implements ActionListener
        {
        String msg;
        TextField t,t1;
        Checkbox c1,c2;
        public void init()
        {
        setBackground(Color.cyan);
        setForeground(Color.magenta);
        Label l = new Label("Name");
        t = new TextField(30);
        Label l1 = new Label("Password");
        t1 = new TextField(30);
        t1.setEchoChar('*');
        c1 = new Checkbox("Sugar");
        c2= new Checkbox("Ice Cream");
        Button b2 = new Button("Click ME");
        add(l);
        add(t);
        add(l1);
        add(t1);
        add(c1);
        add(c2);
        add(b2);
        b2.addActionListener(this);
        }
```

```
        public void paint(Graphics g)
        {
        g.drawString(msg, 100,100);
        showStatus("My first applet.. Wish me
        good luck");
        }

        public void actionPerformed(ActionEvent
        e)
        {
        msg = t.getText() + " "+ t1.getText();
        if (c1.getState() == true ) msg+= "Sugar";
        if (c2.getState() == true ) msg+=
        "IceCream";
        repaint();
        }
        }
/*<applet code=eve height=400 width=400>
</applet> */
```

IO

- The simplest way to do IO: the scanner class
- Very flexible and easy to use.
- Can read from the console or from a file

The scanner class – Read from a file

- Example

```
import java.util.*;
import java.io.*;

class fsc
        {
        public static void main(String args[]) throws FileNotFoundException
        {
        File f = new File("Ls.txt");
        Scanner s = new  Scanner(f);

        while (s.hasNext())
        {
        System.out.println(s.next());
        }
        }
        }
```

Streams

- A stream is an ordered sequence of bytes that can be used as
  - A source for input (input stream)
  - A destination for output (output stream)
- A program can have multiple streams
  - Examples: console, files, sockets, memory, strings
- The java classes in the package java.io provide utilities for dealing with streams

Streams

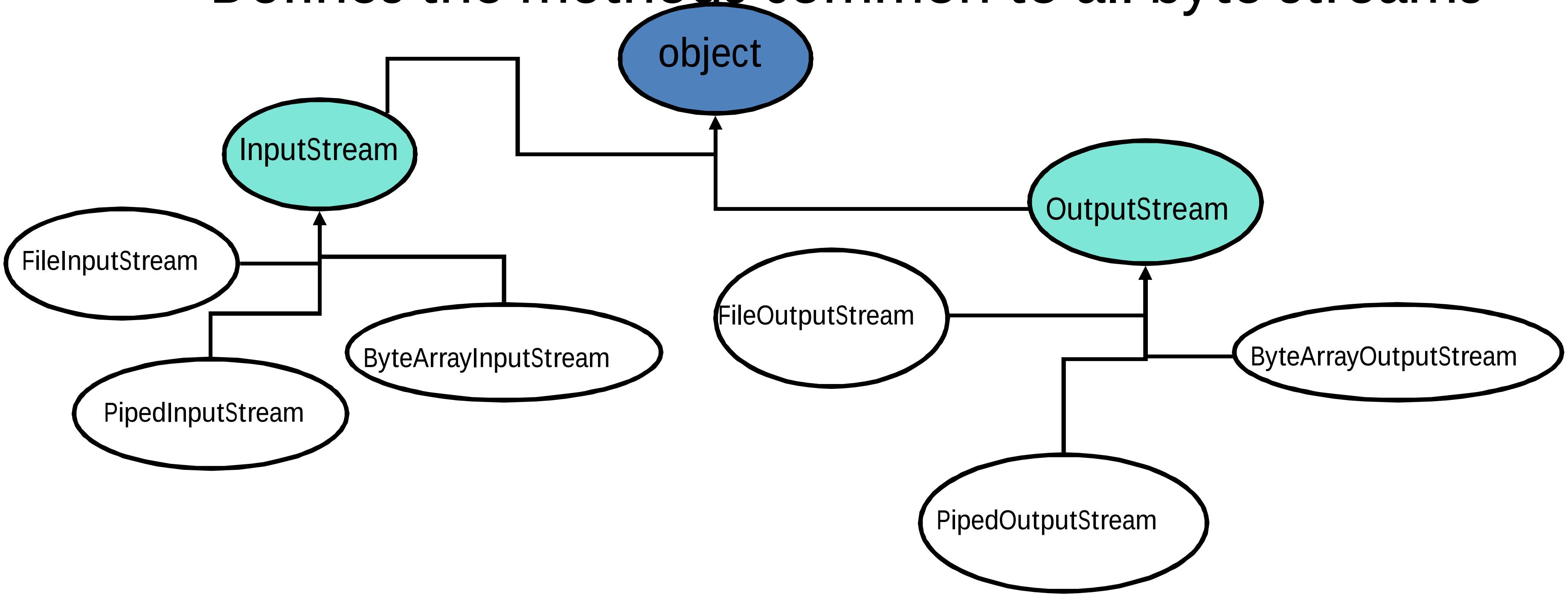
- Java categorizes streams in multiple ways:
  - input or output
  - character (16-bit unicode characters) or byte (8 bits)

Streams

- To read data, a JAVA program opens a stream on a source (such as a file, memory, or a socket) It then reads the information sequentially
- To send information to an external destination, the program opens a stream to the destination and writes the data sequentially to the stream
- Java has predefined byte streams:
  - System.in
  - System.out
  - System.err

Byte Streams

- `InputStream` and `OutputStream`
  - Abstract classes
  - The parent class of all byte streams
  - Defines the methods common to all byte streams



Usage – Byte Stream

- The following methods are used for input, output

- FileInputStream fi = new FileInputStream(“file”);
- FileOutputStream fo = new FileOutputStream(“outfile”);

- ch = fi.read()
- fi.write(ch)
- fi. close()

ByteStreams: example

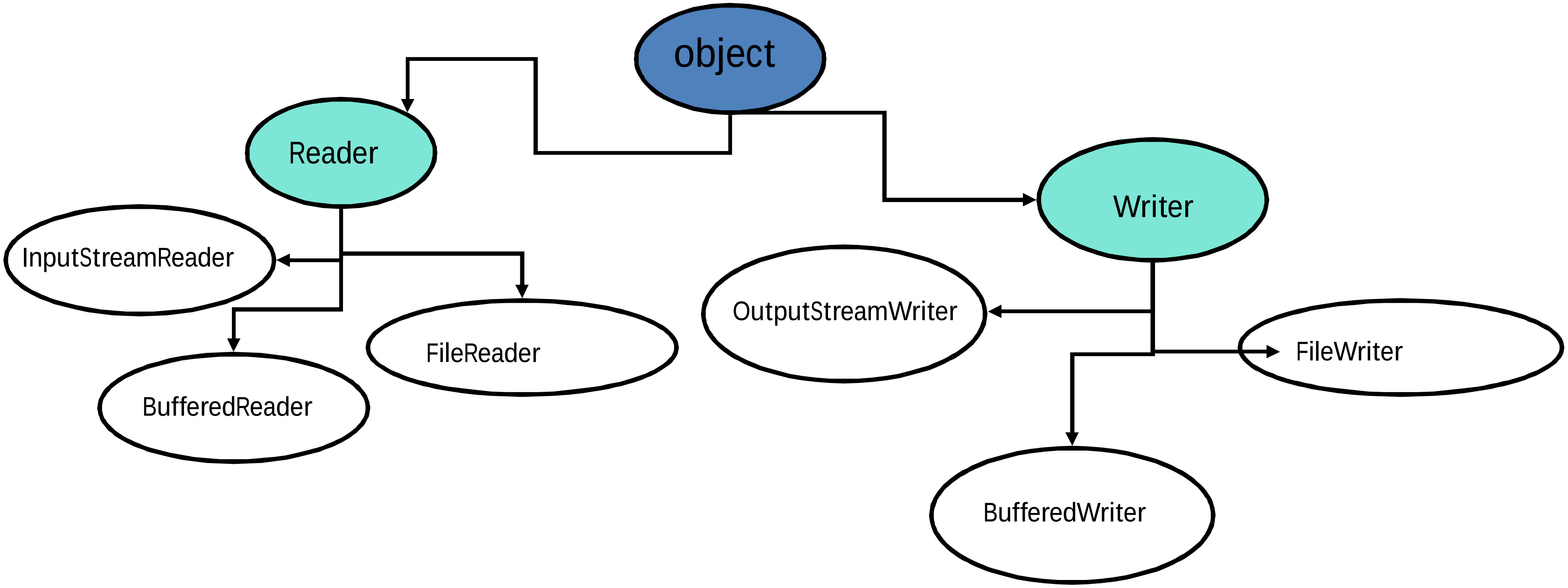
```
// This program counts the number of bytes in a file supplied in the command line

import java.io.*;

class CountBytes {
  public static void main(String[] args)
      throws IOException
 {
      FileInputStream in = new FileInputStream(args[0]);

      int total = 0;
      while (in.read() != -1)
                total++;

      System.out.println(total + “ bytes”);
      }
}
```

Character Streams

- `Reader` and `Writer` streams
  - Abstract classes
  - The parent class of all character streams
  - Defines the methods common to all character streams
    - These methods correspond to InputStream/OutputStream methods
    - Example:
      - Read method of `InputStream` returns lowest 8 bits of an int as a byte
      - Read method of `Reader` returns lowest 16 bits of an int as a char

Character Streams

- `Reader` and `Writer`



Usage – Character Stream

- The following methods are used for input, output

- FileReader fi = new FileReader(“file”);
- FileWriter fo = new FileWriter(“outfile”);

- ch = fi.read()
- fi.write(ch)
- fi. close()

Character Streams: Example

```
// This program finds the total chars and spaces in a file suppled in the
   //command line
import java.io.*;

class CountSpace {
   public static void main(String[] args)
        throws IOException
   {
        FileReader in = new FileReader(args[0]);

        int ch;
        int total;
        int spaces = 0;
        for (total = 0; (ch = in.read()) != -1; total++) {
                if (Character.isWhitespace((char) ch))
                        spaces++;
        }
        System.out.println(total + “ chars, “ + spaces + “ spaces”);
   }
}
```

Example: Console I/O

- To read from the console (System.in) you need to be able to read characters.

  You can change the byte stream to a character stream

  - `InputStreamReader ch = new InputStreamReader(System.in);`

- This will read each *individual* character. To get *tokens*, turn the input stream into a BufferedReader object:

  - `BufferedReader br = new BufferedReader(ch);`

* a *token* is a set of characters surrounded by white space

Example: Console I/O

- Now use the readLine() method of `BufferedReader` to read a line from the console (a string):

```
 String input = br.readLine();
```

- In this case, the token is the entire line – the linefeed character is used to determine when to stop putting characters into the buffer.

- To change a String into a number:

```
  int count = Integer.parseInt(input);
```

- Note: System.out is predefined to print numbers and strings.

Console I/O Example

```
BufferedReader b = new BufferedReader(new
InputStreamReader(System.in));
System.out.println("Enter first no");
int n1 = Integer.parseInt(b.readLine());
System.out.println("Enter second no");
int n2 = Integer.parseInt(b.readLine());
System.out.println(n1 + "*" + n2 + "=" + n1*n2);
```

File I/O – Copy one file to another- Method 1 – Read Byte

```
import java.io.*;

class read3
    {
    public static void main(String args[]) throws IOException
    {
    FileInputStream i = new FileInputStream("thread2.java");
    FileOutputStream b = new FileOutputStream("thread2cpy.java");
    int c;
   while (( c = i.read()) != -1)
    b.write(c);
    i.close();
    b.close();
    }
    }
```

File I/O – Copy one file to another- Method 2 (Read char)

```
import java.io.*;

class read2
    {
    public static void main(String args[]) throws IOException
    {
    FileReader i = new FileReader("thread1.java");
    FileWriter b = new FileWriter("thread1cpy.java");
    int c;
   while (( c = i.read()) != -1)
    b.write(c);
    i.close();
    b.close();
    }
    }
```

File I/O – Copy one file to another-Method 3 (Read line)

```
import java.io.*;

class read5
    {
    public static void main(String args[]) throws IOException
    {
    FileReader i = new FileReader("thread3.java");
    FileWriter b = new FileWriter("thread3cpy.java");
    BufferedReader br = new BufferedReader(i);
    PrintWriter pr = new PrintWriter(b);

    String s;
    while (( s = br.readLine()) != null)
    pr.println(s);

    i.close();
    b.close();
    }
    }
```

Random Access File

- Sequential files are *read-only* or *write-only* streams.
- *Sequential* files that cannot be updated without creating a new file.
- It is often necessary to modify files or to insert new records into files. Java provides the RandomAccessFile class to allow a file to be read from and write to at random locations.

File Pointer

- A random access file consists of a sequence of bytes.
- There is a special marker called *file pointer* that is positioned at one of these bytes.
- A read or write operation takes place at the location of the file pointer.
- When a file is opened, the file pointer sets at the beginning of the file.
- When you read or write data to the file, the file pointer moves forward to the next data.

Usage

- RandomAccessFile rf = new RandomAccessFile(“eg.txt”,”rw”);
- RandomAccessFile rf = new RandomAccessFile(“eg.txt”,”r”);

- rf.readInt(); rf.writeInt(i);
- rf.readChar(); rf.writeChar(c);
- rf.readDouble(); rf.writeDouble(d);

- rf.seek(loc) – Positions the file pointer in the location
- rf.getFilePointer() – returns the current position
- rf.length() – Total length of the file
- rf.skipBytes(no) – Skip number of bytes

Example

```
import java.io.*;
class rnd
      {
      public static void main(String args[]) throws IOException
      {
      RandomAccessFile rf = new RandomAccessFile("rd.txt","rw");
      for (int i = 0; i <10; i++)
      rf.writeInt(i*10);
      rf.seek(20); // prints the 6th number - 4 bytes int
      System.out.println(rf.readInt());
      rf.close();
      }
      }
```

The output will be

```
50
```

Append data to a random access file

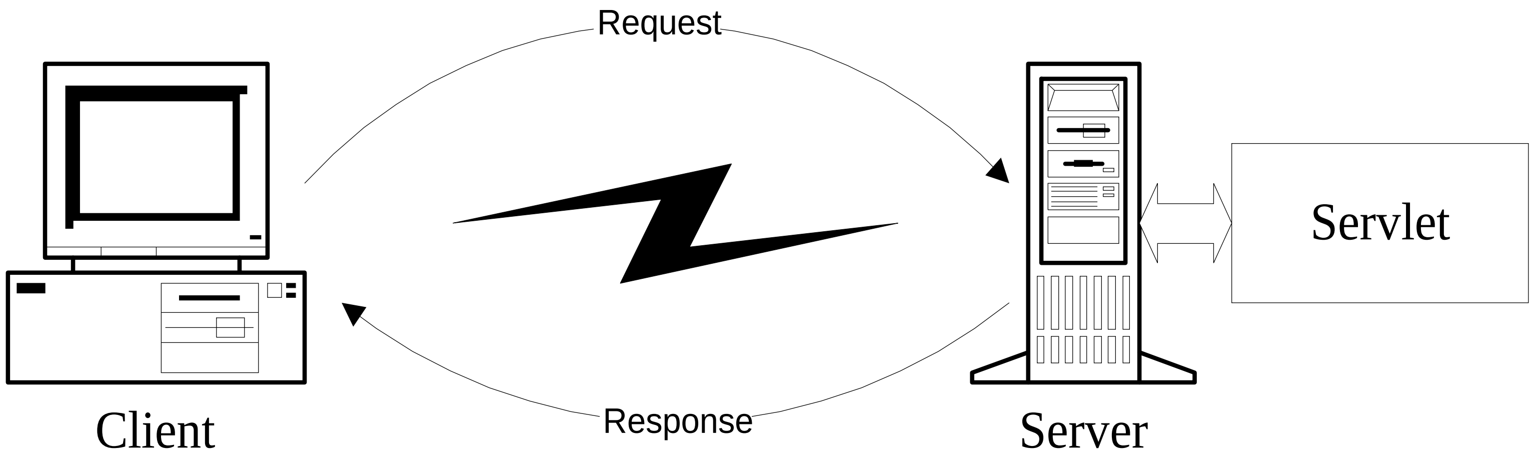
- Move to the end of the file. Write the data

```
import java.io.*;
class rnd
      {
      public static void main(String args[]) throws IOException
      {
      RandomAccessFile rf = new RandomAccessFile("rd.txt","rw");
      for (int i = 0; i <10; i++)
      rf.writeInt(i*10);

      rf.seek(40);
      rf.writeInt(1000);

      rf.seek(rf.length());
      rf.writeInt(999);
      rf.seek(rf.length()-4);
      System.out.println("Last" + rf.readInt());
      rf.close();
      }
      }
```

What is java servlet ?

A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients.

Servlet can be considered as a server-side applet.



Java Servlet Architecture

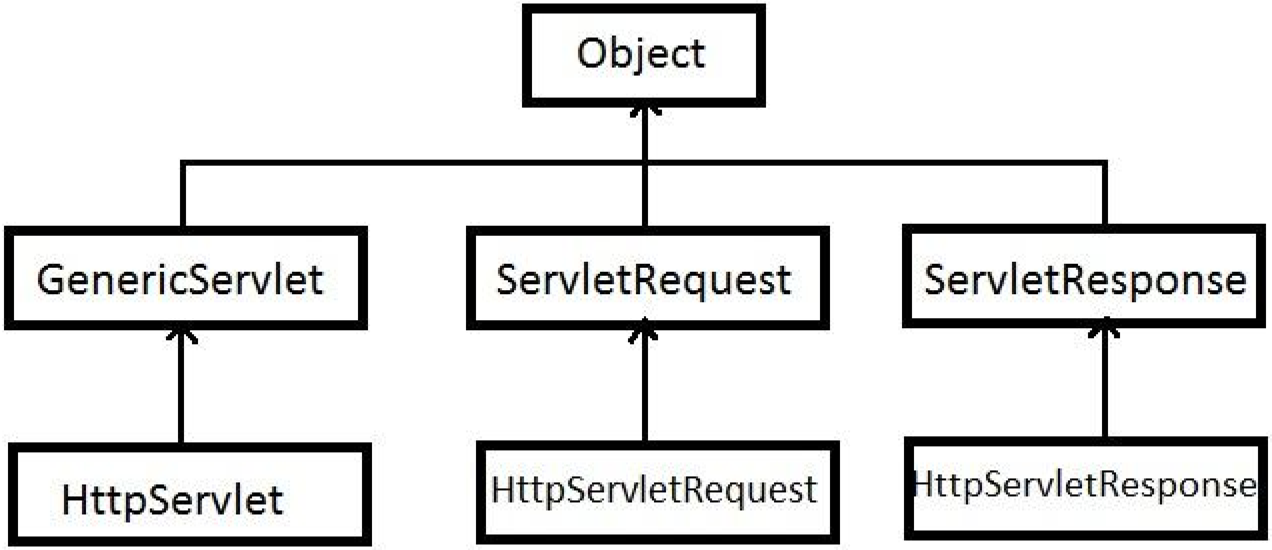
- Two packages make up the servlet architecture
  - ***javax.servlet***
    - Contains generic interfaces and classes that are implemented and extended by all servlets
  - ***javax.servlet.http***
    - Contains classes that are extended when creating HTTP-specific servlets

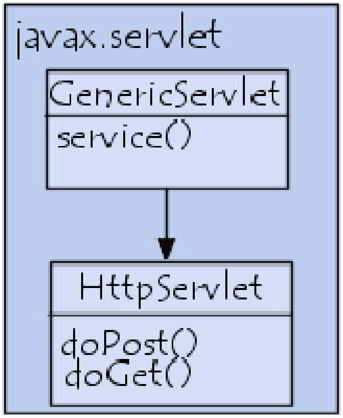
- The heart of servlet architecture is the interface class ***javax.servlet.Servlet***
- It provides the framework for all servlets
- Defines five basic methods – init, service, destroy, getServletConfig and getServletInfo

HttpServlet

- *HttpServlet* class is extended from *GenericServlet* class
- When *HttpServlet.service( )* is invoked, it calls *doGet( )* or *doPost( ),* depending upon how data is sent from the client
- *HttpServletRequest* and *HttpServletResponse* classes are just extensions of *ServletRequest* and *ServletResponse* with HTTP-specific information stored in them

Hierarchy


Object
GenericServlet
ServletRequest
ServletResponse
HttpServlet
HttpServletRequest
HttpServletResponse


javax.servlet
GenericServlet
service()
HttpServlet
doPost()
doGet()

Life Cycle of a Servlet

- Servlets operate in the context of a request and response model managed by a servlet engine

- The engine does the following
  - Loads the servlet when it is first requested
  - Calls the servlet's ***init( )*** method
  - Handles any number of requests by calling the servlet's ***service( )*** method
  - When shutting down, calls each servlet's ***destroy( )*** method

Servlet Life Cycle Summary

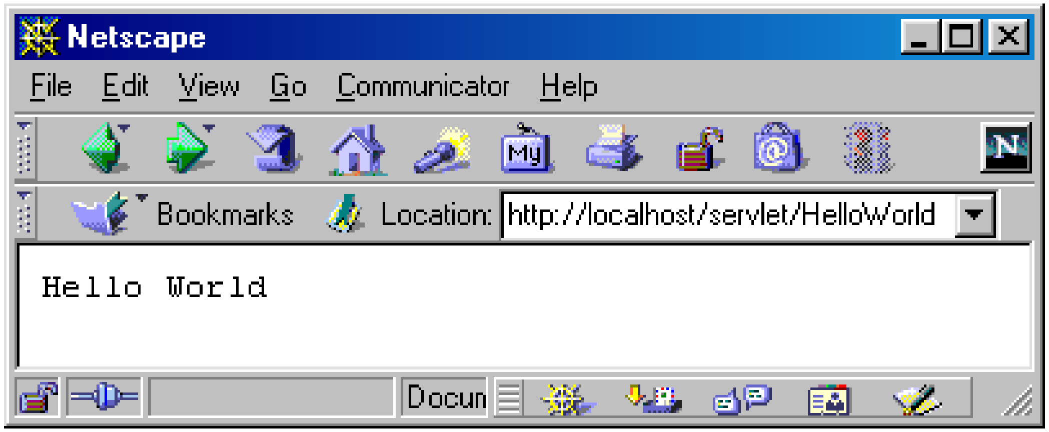
- init
  - Executed once when the servlet is first loaded.
    Not called for each request.
- service
  - Called in a new thread by server for each request.
    Dispatches to doGet, doPost, etc.
    Do not override this method!
- doGet, doPost
  - Handles GET, POST, etc. requests.
  - Override these to provide desired behavior.
- destroy
  - Called when server deletes servlet instance.
    Not called after each request.

Simple Servlet Template

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletTemplate extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {


    PrintWriter out = response.getWriter();
    // Use "out" to send content to browser
  }
}
```

A Simple Servlet That Generates Plain Text

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
      throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("Hello World");
  }
}
```



Location: http://localhost/servlet/HelloWorld

```
Hello World
```

A Servlet That Generates HTML

```
import java.io.*;
import javax.servlet.*;
 import javax.servlet.http.*;
public class HelloWorld extends HttpServlet
     {
      public void doGet(HttpServletRequest request,
     HttpServletResponse response) throws IOException,
     ServletException
     {
     response.setContentType("text/html");
     PrintWriter out = response.getWriter(); out.println("<html>");
     out.println("<body>"); out.println("<h1>Hello World!</h1>");
     out.println("</body>"); out.println("</html>");
} }
```

Form Processing

- **Reading Form Data using Servlet:**
- Servlets handles form data parsing automatically using the following methods depending on the situation:
- **getParameter():** You call request.getParameter() method to get the value of a form parameter.
- **getParameterValues():** Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
- **getParameterNames():** Call this method if you want a complete list of all parameters in the current request.

Get Vs Post method

GET Method:

- All the name value pairs are submitted as a query string in URL.
- It's not secured as it is visible in plain text format in the Location bar of the web browser.
- Length of the string is restricted.

POST Method:

- All the name value pairs are submitted in the Message Body of the request.
- Length of the string (amount of data submitted) is not restricted.
- Post Method is secured because Name-Value pairs cannot be seen in location bar of the web browser.

Get method

```
<html>
<body>
<form action="HelloForm" method="GET">
First Name: <input type="text" name="firstname“>
Last Name: <input type="text" name="lastname" >
<input type="submit" value="Submit" >
</form>
</body>
</html>
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloForm extends HttpServlet
{
public void doGet(HttpServletRequest request,    HttpServletResponse response)
      throws ServletException, IOException
        {
       response.setContentType("text/html");
     PrintWriter out = response.getWriter();

    out.println("<html>\n" +
                    "<body>\n" +
                    “First Name “+

           request.getParameter("firstname") + "\n" +

            “ Last Name"    +

                  request.getParameter("lastname") + "\n" +

            "</body> </html>");
                   }
                  }
```

Using Post method

```
<html>
<body>
<form action="HelloForm" method=“POST">
First Name: <input type="text" name="firstname“>
Last Name: <input type="text" name="lastname" >
<input type="submit" value="Submit" >
</form>
</body>
</html>
```

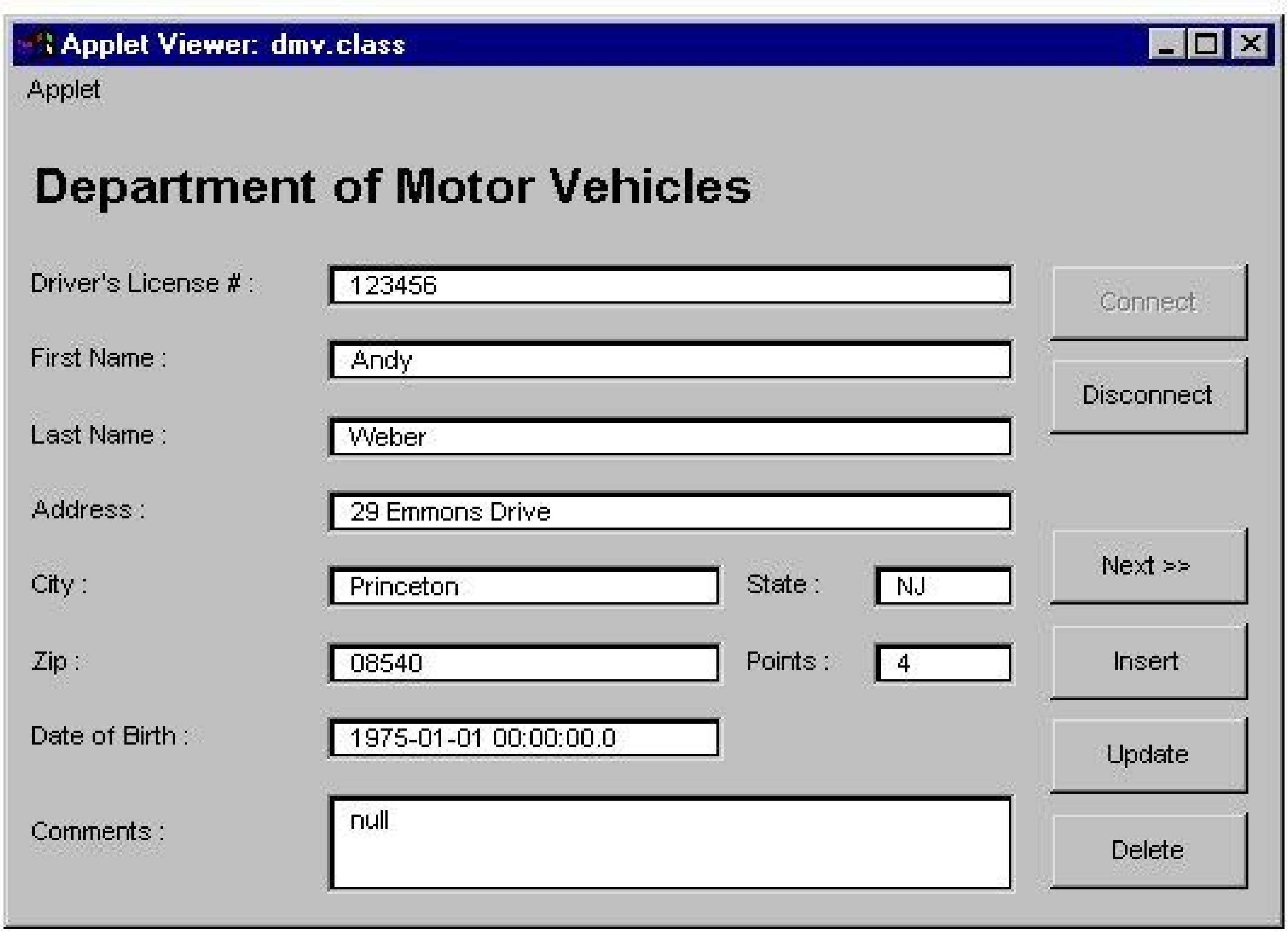
```
// Method to handle POST method request.
 public void doPost(HttpServletRequest request,
  HttpServletResponse response)      throws
  ServletException, IOException
  {    doGet(request, response);  }
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloForm extends HttpServlet
{
public void doGet(HttpServletRequest request,    HttpServletResponse response)
      throws ServletException, IOException
        {
       response.setContentType("text/html");
     PrintWriter out = response.getWriter();

 out.println("<html>\n" +
                      "<body>\n" +
                      "First Name "+

           request.getParameter("firstname") + "\n" +

            " Last Name"    +

                  request.getParameter("lastname") + "\n" +

            "</body> </html>");                              }

public void doPost(HttpServletRequest request, HttpServletResponse response)      throws
       ServletException, IOException
       {     doGet(request, response);  }
}
```

JDBC

- JDBC is a standard interface for connecting to relational databases from Java.
- The JDBC classes and interfaces are in the ***java.sql*** package.

Example



Department of Motor Vehicles

Driver's License # : 123456

First Name : Andy

Last Name : Weber

Address : 29 Emmons Drive

City : Princeton

State : NJ

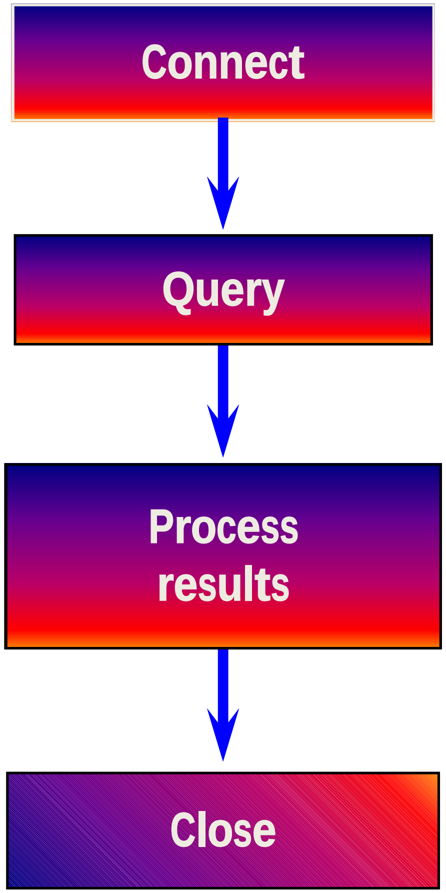
Zip : 08540

Points : 4

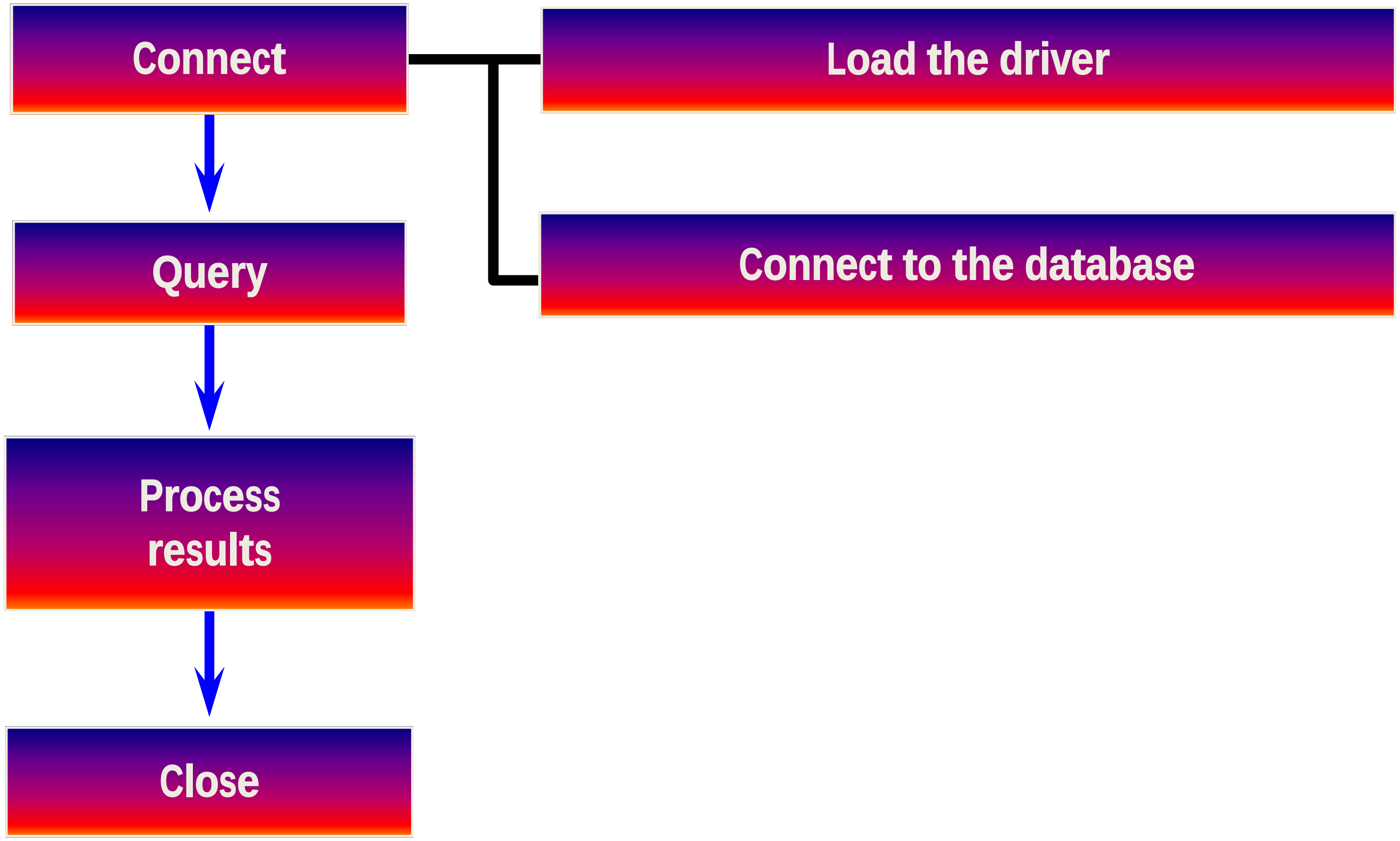
Date of Birth : 1975-01-01 00:00:00.0

Comments : null

Overview of Querying a Database With JDBC



Stage 1: Connect



Connect

Query

Process results

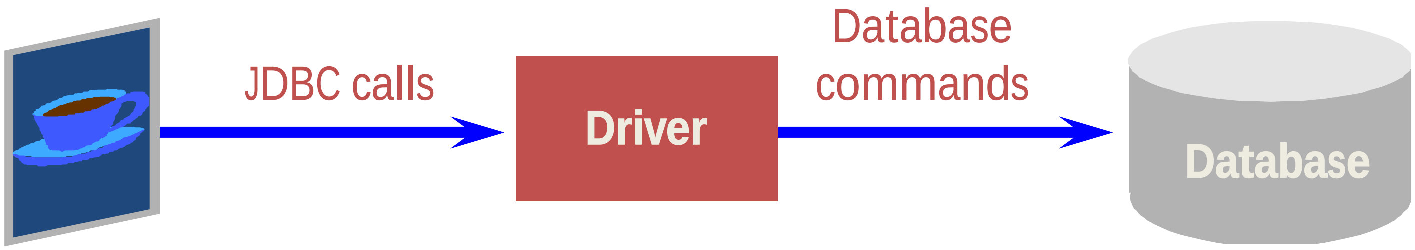
Close

Load the driver

Connect to the database

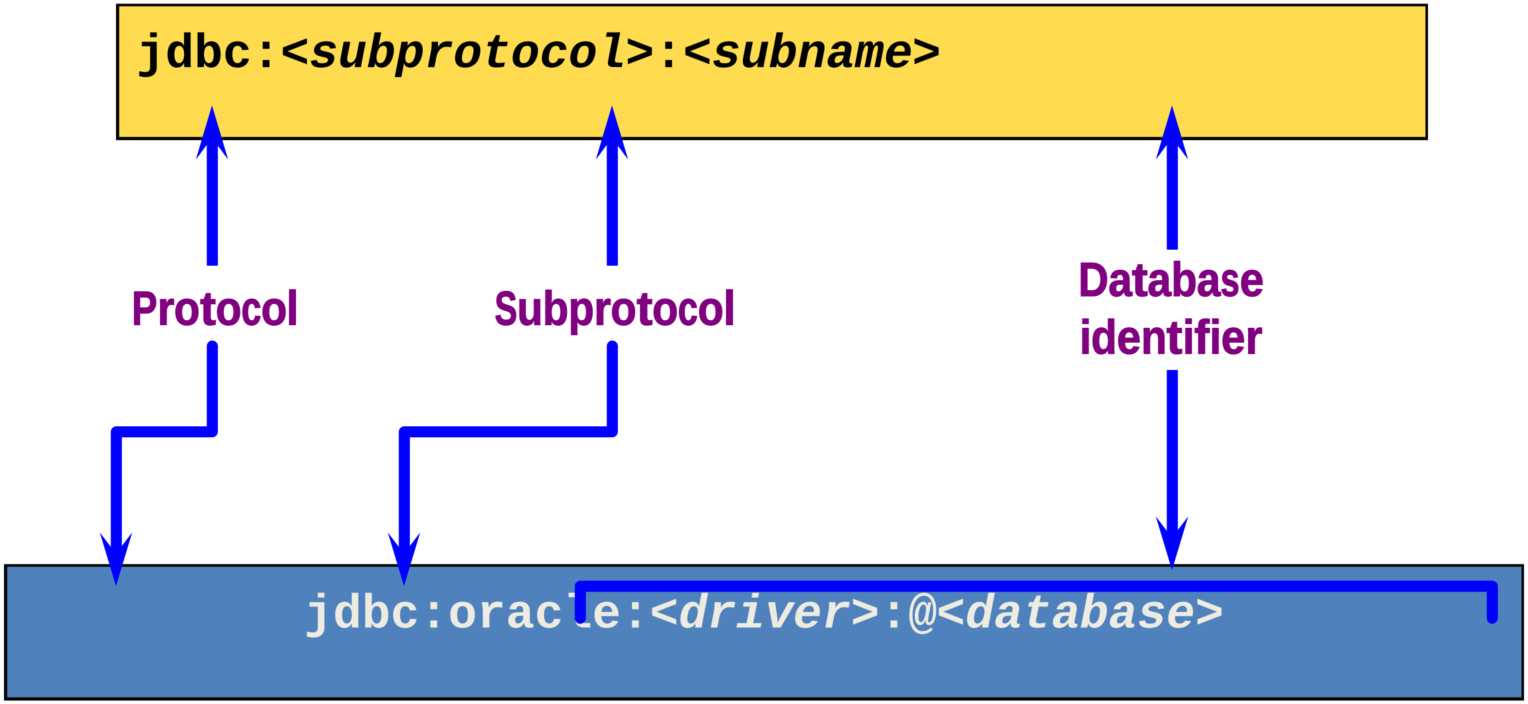
A JDBC Driver

- Is an interpreter that translates JDBC method calls to vendor-specific database commands



About JDBC URLs

- JDBC uses a URL to identify the database connection.



```
jdbc:<subprotocol>:<subname>
```

Protocol

Subprotocol

Database identifier

```
jdbc:oracle:<driver>:@<database>
```

How to Make the Connection

1. Load the driver.

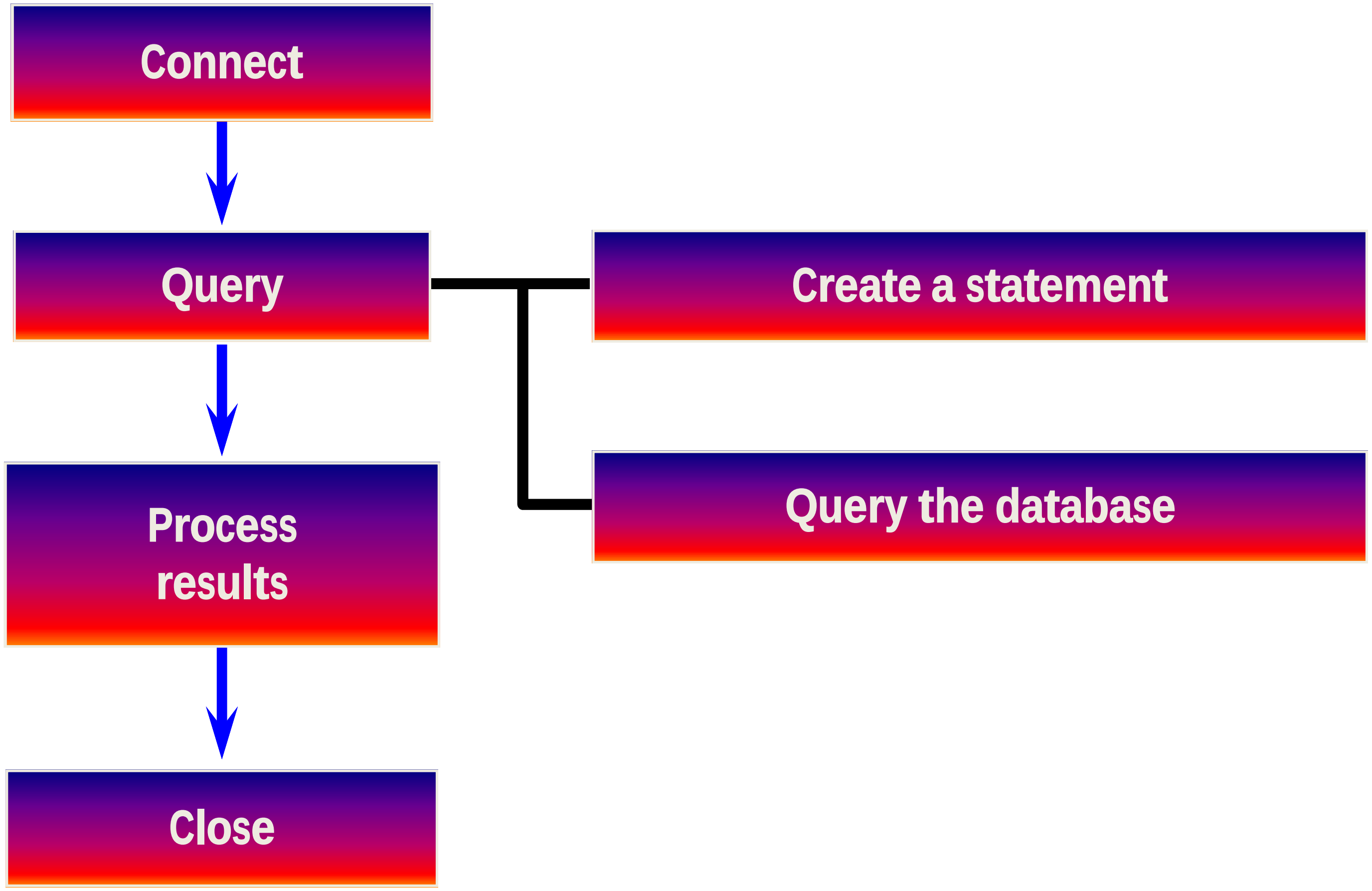
```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

2. **Connect to the database.**

```
Connection conn = DriverManager.getConnection
     (URL, userid, password);
```

```
Connection conn = DriverManager.getConnection
      ("jdbc:oracle:thin:@myhost:1521:orcl",
      "scott", "tiger");
```

Stage 2: Query



Connect

Query

Create a statement

Query the database

Process results

Close

The Statement Object

- A Statement object sends your SQL statement to the database.
- You need an active connection to create a JDBC statement.
- Statement has three methods to execute a SQL statement:
  - **executeQuery()** for QUERY statements
  - **executeUpdate()** for INSERT, UPDATE, DELETE, or DDL statements
  - **execute()** for either type of statement

How to Query the Database

1. Create an empty statement object.

```
Statement stmt = conn.createStatement();
```

2. Execute the statement.

```
ResultSet rset = stmt.executeQuery(statement);
int count = stmt.executeUpdate(statement);
boolean isquery = stmt.execute(statement);
```

Querying the Database: Examples

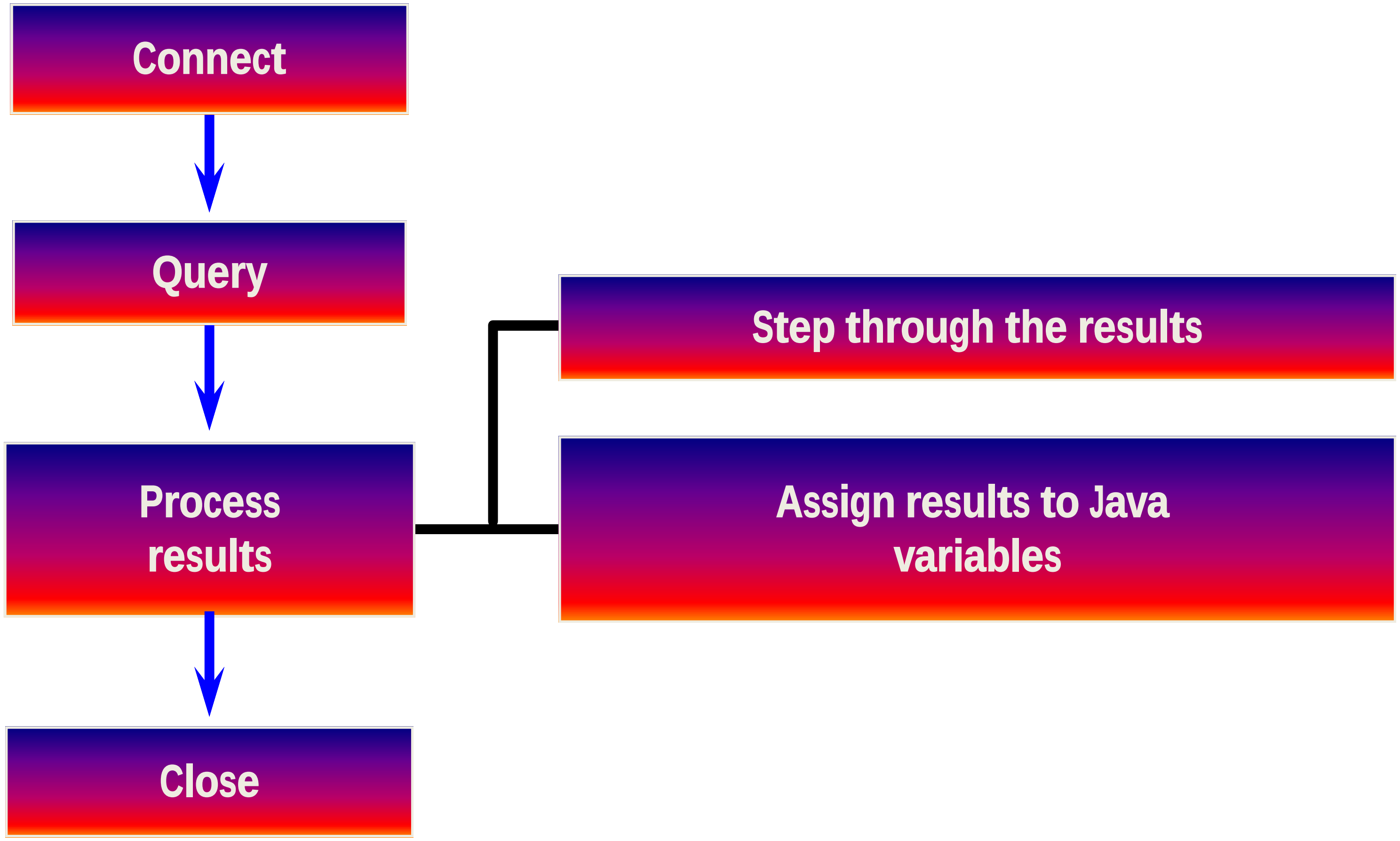
- Execute a select statement.

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery
  ("select REGNO ,NAME from STUDENT");
```

- Execute a delete statement.

```
Statement stmt = conn.createStatement();
int rowcount = stmt.executeUpdate
  ("delete from STUDENT
      where REGNO = 1011");
```

Stage 3: Process the Results



Connect

Query

Process results

Close

Step through the results

Assign results to Java variables

The ResultSet Object

- JDBC returns the results of a query in a ResultSet object.
- A ResultSet maintains a cursor pointing to its current row of data.
- Use next() to step through the result set row by row.
- getString(), getInt(), and so on assign each value to a Java variable.

How to Process the Results

- 1. Step through the result set.

```
while (rset.next()) { … }
```

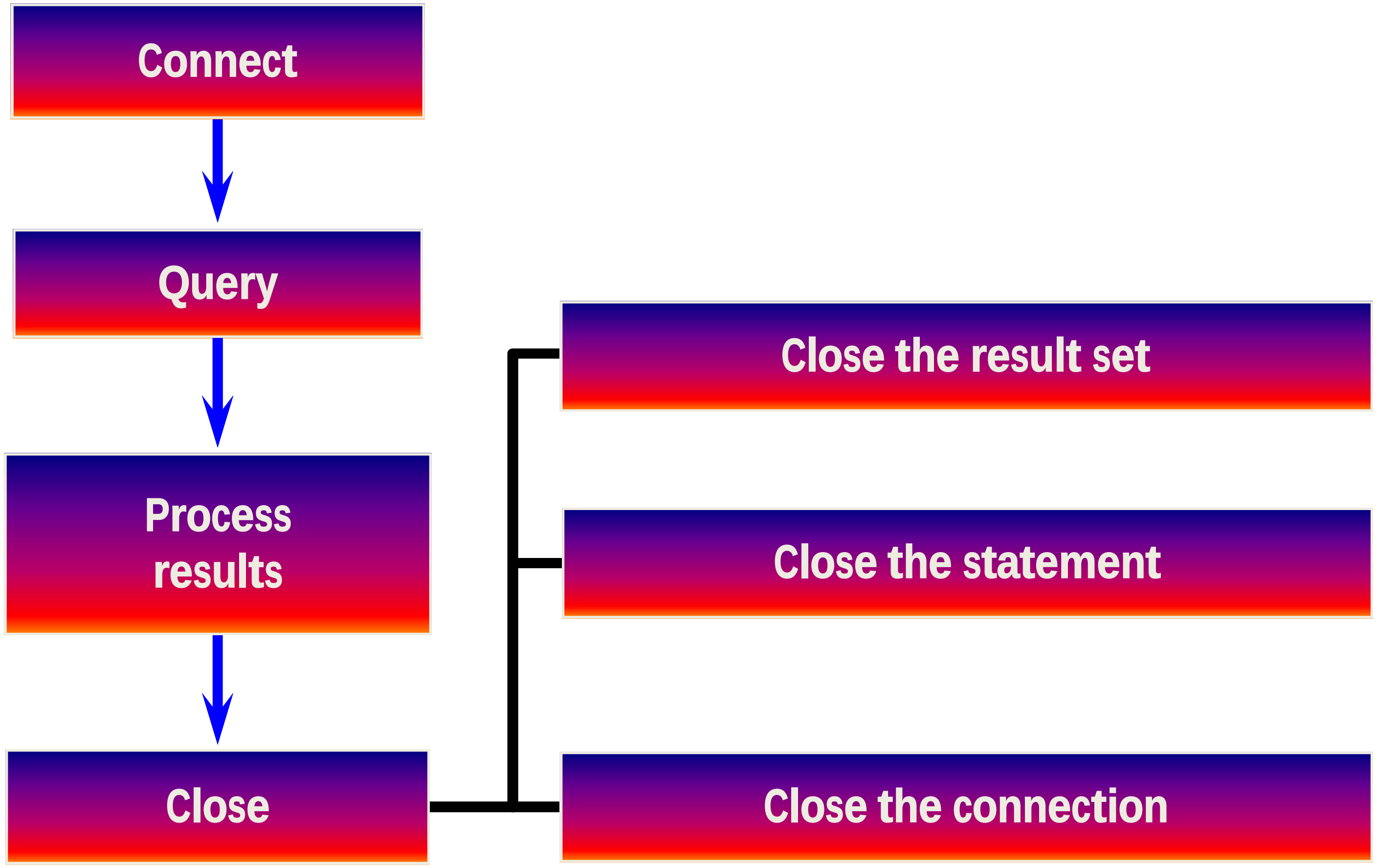
- 2. Use getXXX() to get each column value.

```
String val =
rset.getString(colname);
```

```
String val =
rset.getString(colIndex);
```

```
while (rset.next()) {
  String title = rset.getString("TITLE");
  String year = rset.getString(2);
  … // Process or display the data
}
```

Stage 4: Close



How to Close the Connection

1. Close the ResultSet object.

```
rset.close();
```

2. Close the Statement object.

```
stmt.close();
```

3. Close the connection.

```
conn.close();
```

The PreparedStatement Object

- A PreparedStatement object holds precompiled SQL statements.
- Use this object for statements you want to execute more than once.
- A prepared statement can contain variables that you supply each time you execute the statement.

How to Create a Prepared Statement

1. Register the driver and create the database connection.
2. Create the prepared statement, identifying variables with a question mark (?).

```
PreparedStatement pstmt =
  conn.prepareStatement("update STUDENT
  set STATUS = ? where REGNO = ?");
```

```
PreparedStatement pstmt =
  conn.prepareStatement("select STATUS from
  STUDENT where REGNO = ?");
```

How to Execute a Prepared Statement

1. Supply values for the variables.

```
pstmt.setXXX(index, value);
```

2. Execute the statement.

```
pstmt.executeQuery();
pstmt.executeUpdate();
```

```
PreparedStatement pstmt =
  conn.prepareStatement("update Student
  set STATUS = ? where Regno = ?");
pstmt.setString(1, “PASS);
pstmt.setInt(2, 1111);
pstmt.executeUpdate();
```

The CallableStatement Object

- A CallableStatement object holds parameters for calling stored procedures.
- A callable statement can contain variables that you supply each time you execute the call.
- When the stored procedure returns, computed values (if any) are retrieved through the CallabableStatement object.

How to Create a Callable Statement

- Register the driver and create the database connection.
- Create the callable statement, identifying variables with a question mark (?).

```
String SQL = "{call getEmpName (?, ?)}";
CallableStatement cstmt = conn.prepareCall (SQL);
```

How to Execute a Callable Statement

1. Set the input parameters.

```
cstmt.setXXX(index, value);
```

2. Execute the statement.

```
cstmt.execute(statement);
```

3. Get the output parameters.

```
var = cstmt.getXXX(index);
```

ResultSet enhancements

- **Scrollability**
  - The ability to move backward as well as forward through a result set.
  - The ability to move to any particular position in the result set
- **Sensitivity** must be specified. Sensitivity can detect whether data is changed or not.
  - Sensitive or Insensitive Mode
- **Updatability**
  - Can insert, modify, delete using while navigating a resultset

6 Types of ResultSet

- forward-only/read-only
- forward-only/updatable
- scroll-sensitive/read-only
- scroll-sensitive/updatable
- scroll-insensitive/read-only
- scroll-insensitive/updatable

APIs

java.sql.Connection

Statement createStatement (int resultSetType, int resultSetConcurrency)

resultSetType

ResultSet.TYPE_FORWARD_ONLY

ResultSet.TYPE_SCROLL_INSENSITIVE

ResultSet.TYPE_SCROLL_SENSITIVE

resultSetConcurrency

ResultSet.CONCUR_READ_ONLY

ResultSet.CONCUR_UPDATABLE

Example : Backward

```
Statement stmt = conn.createStatement
(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");
rs.afterLast();
while ( rs.previous() )
{
System.out.println(rs.getString("empno") + " " + rs.getFloat("sal"));
}
...
```

Example : delete row

```
...
rs.absolute(5);
rs.deleteRow();
...
```

Example : update row

```
Statement stmt = conn.createStatement

(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");

if (rs.absolute(10)) // (returns false if row does not exist)

{

rs.updateString(1, "28959");

rs.updateFloat("sal", 100000.0f);

rs.updateRow();

}

// Changes will be made permanent with the next COMMIT operation.

...
```

Example

```
import java.sql.*;
import java.util.Scanner;

public class jdbctest
{ Connection conn=null;

 public static void main(String[] args)
 {
  Scanner sre = new Scanner(System.in);
        try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        Connection conn=DriverManager.getConnection("jdbc:odbc:Student","","");

                    while (true)
                    {
                    System.out.println("1.Select");
                    System.out.println("2.Insert");
                    System.out.println("3.Update");
                    System.out.println("4.Delete");
                    System.out.println("5.Exit");
                    System.out.println("Enter ur choice");
                    int choice = sre.nextInt(6);
```

```
switch (choice)
                 {
                 case 1:
                    System.out.println("[OUTPUT FROM SELECT]");
                    Statement st = conn.createStatement();
                st.execute("select * from stu");
                ResultSet rs = st.getResultSet();
                    System.out.println("REgno    Name    Total");
            while (rs.next())
            {
                 String s = rs.getString(1);
             String s1 = rs.getString(2);
                 int m1 = rs.getInt(3);
                 int m2 = rs.getInt(4);
                 int m3 = rs.getInt(5);
             System.out.println(s + "   " + s1 + "   "  + (m1+m2+m3));
                 }

                 break;
case 2:

     System.out.println("\n[Performing INSERT] ... ");
     st = conn.createStatement();
    st.executeUpdate("INSERT INTO stu VALUES (999, 'AAA', 55,66,77)");
     break;
 case 3:
  System.out.println("\n[Performing UPDATE] ... ");
  st = conn.createStatement();
  st.executeUpdate("UPDATE stu SET regno='111' WHERE name='AAA'");
  break;

case 4:
  System.out.println("\n[Performing DELETE] ... ");
    st = conn.createStatement();
   st.executeUpdate("DELETE FROM stu WHERE name='AAA'");
  break;
case 5: System.exit(0);
}
}
}
    catch (Exception err) {
    System.out.println("ERROR: " + err);
     }

}
}
```

Networking Basics

- **Internet protocol (IP)** addresses
  - Every host on Internet has a unique IP address

    `143.89.40.46, 203.184.197.198`

    `203.184.197.196, 203.184.197.197, 127.0.0.`

  - More convenient to refer to using hostname string
  - `vit.ac.in, gmail.com,`
  - One hostname can correspond to multiple internet addresses:

    www.yahoo.com:

    66.218.70.49; 66.218.70.50; 66.218.71.80; 66.218.71.84; …
  - Domain Naming Service (DNS) maps names to numbers

Networking Basics

- Ports

  Many different services can be running on the host

  A **port** identifies a service within a host

  Many standard port numbers are pre-assigned

  time of day 13, ftp 21, telnet 23, smtp 25, http 80

  IP address + port number = "phone number“ for service

Networking Basics

protocols : rules that facilitate communications between machines

Examples:

- HTTP: HyperText Transfer Protocol
- FTP: File Transfer Protocol
- SMTP: Simple Message Transfer Protocol
- TCP: Transmission Control Protocol
- UDP: User Datagram Protocol

Protocols are standardized and documented

- So machines can reliably work with one another

Networking Basics

Client-Server interaction

- Communication between hosts is two-way, but usually the two hosts take different roles

- **Server waits for client to make request**
  - Server registered on a known port with the host ("public phone number")
  - Usually running in endless loop
  - Listens for incoming client connections

Networking Basics

Client "calls" server to start a conversation

- Client making calls uses hostname/IP address and port number
- Sends request and waits for response

Server offers shared resource (information,database, files, printer, compute power) to clients

Java Socket Programming

- The package java.net provides support for sockets programming (and more).

- Typically you import everything defined in this package with:

```
import java.net.*;
```

Classes

- **`InetAddress`**
- **`Socket`**
- **`ServerSocket`**
- **`DatagramSocket`**
- **`DatagramPacket`**

The InetAddress Class

- Occasionally, you would like to know who is connecting to the server. You can use the InetAddress class to find the client's host name and IP address. The InetAddress class models an IP address. You can use the statement shown below to create an instance of InetAddress for the client on a socket.
-

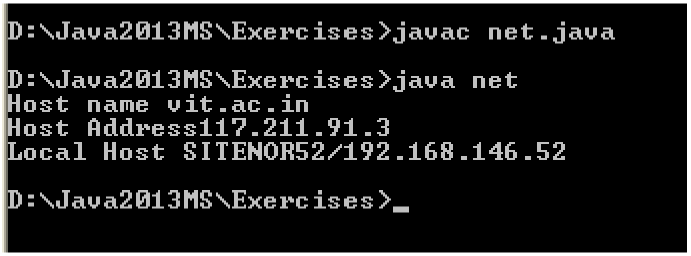
```
InetAddress inetAddress = socket.getInetAddress();
```

-
- Next, you can display the client's host name and IP address, as follows:

```
System.out.println("Client's host name is " +
  inetAddress.getHostName());
System.out.println("Client's IP Address is " +
  inetAddress.getHostAddress());
```

Example

```
•   import java.net.*;
•   import java.io.*;
•   class net
•           {
•           public static void main(String args[]) throws UnknownHostException
•           {
•           InetAddress i = InetAddress.getByName("vit.ac.in");
•
•           System.out.println("Host name " + i.getHostName());
•           System.out.println("Host Address" + i.getHostAddress());
•           System.out.println("Local Host " + i.getLocalHost());
•           }
•           }
```



```
D:\Java2013MS\Exercises>javac net.java

D:\Java2013MS\Exercises>java net
Host name vit.ac.in
Host Address117.211.91.3
Local Host SITENOR52/192.168.146.52

D:\Java2013MS\Exercises>_
```

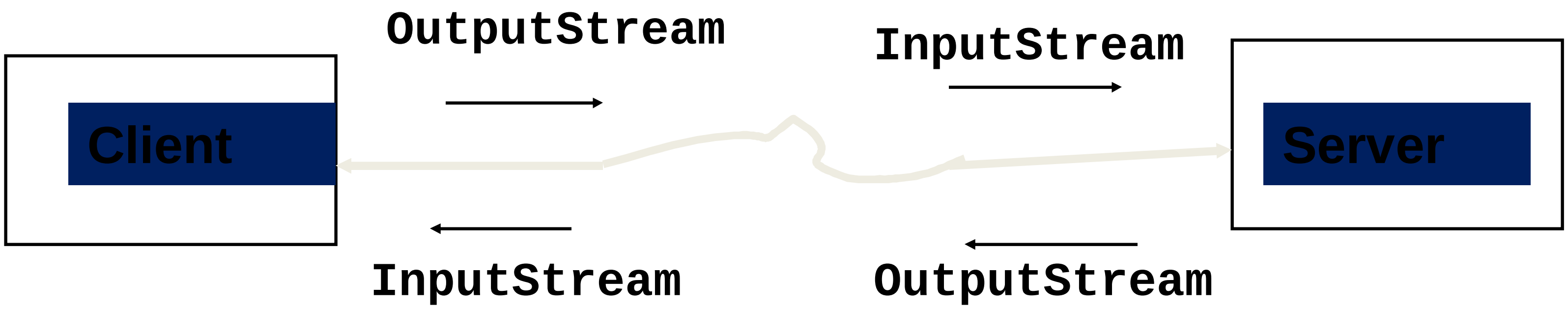
Socket Programming

- **TCP – Transmission Control Protocol**
  - – Connection oriented
    - Handshaking procedure
  - – Reliable byte-stream
- **UDP – User Datagram Protocol**
  - – - Connection less
  - – Unreliable

TCP Socket-Level Programming

java.net.Socket is an abstraction of a bi-directional communication channel between hosts

Send and receive data using streams



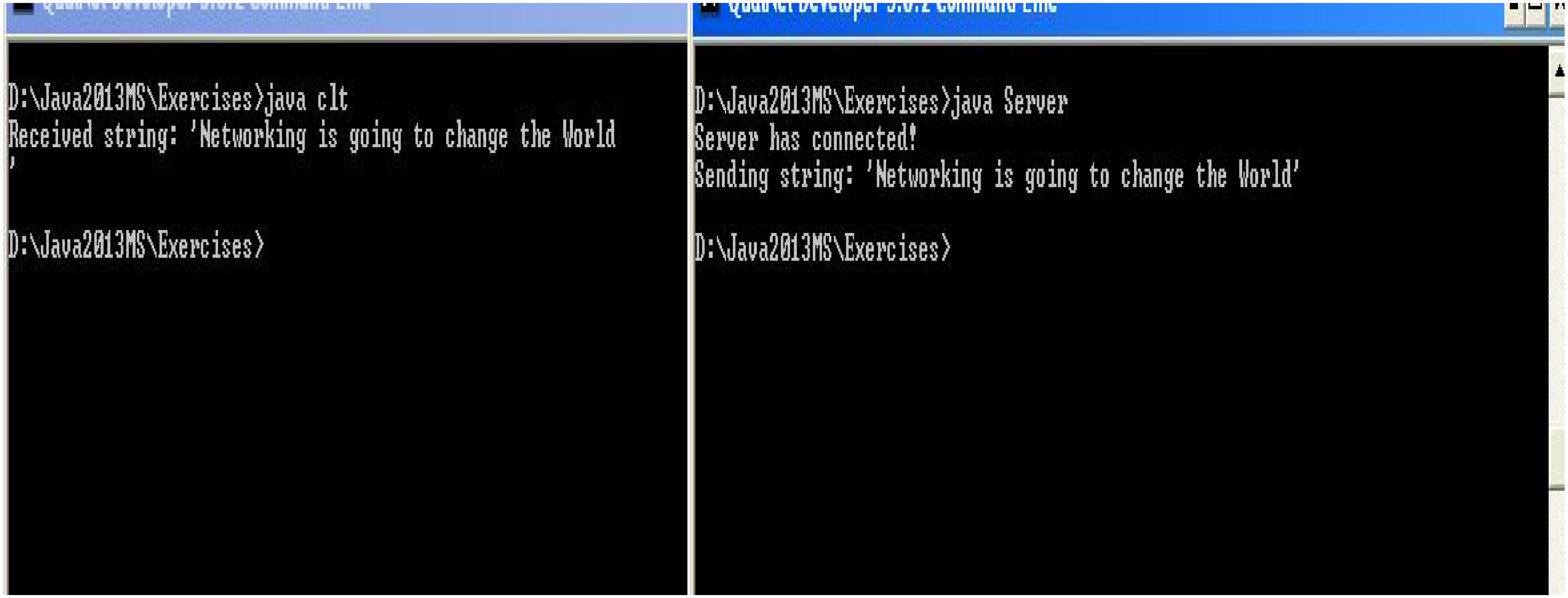
Example – Server Program

```
import java.io.*;
import java.net.*;
class Server {
  public static void main(String args[]) {
    String data = "Networking is going to change the World";
    try {
      ServerSocket srvr = new ServerSocket(1234);
      Socket skt = srvr.accept();
      System.out.print("Server has connected!\n");
      PrintWriter out = new PrintWriter(skt.getOutputStream());
      System.out.print("Sending string: '" + data + "'\n");
      out.print(data);
      out.close();
      skt.close();
      srvr.close();
    }
    catch(Exception e) {
      System.out.print("Whoops! It didn't work!\n");
    }   } }
```

Example – Client Program

```
•   import java.io.*;
•   import java.net.*;

•   class clt {
•      public static void main(String args[]) {
•         try {
•            Socket skt = new Socket("localhost", 1234);
•            BufferedReader in = new BufferedReader(new
    InputStreamReader(skt.getInputStream()));
•            System.out.print("Received string: '");
•              System.out.println(in.readLine()); // Read one line and output it
•              in.close();
•         }
•         catch(Exception e) {
•            System.out.print("Whoops! It didn't work!\n");
•         }   } }
```

Output



```
D:\Java2013MS\Exercises>java clt
Received string: 'Networking is going to change the World
'

D:\Java2013MS\Exercises>
```

```
D:\Java2013MS\Exercises>java Server
Server has connected!
Sending string: 'Networking is going to change the World'

D:\Java2013MS\Exercises>
```

UDP Socket Programming

- java.net.DatagramSocket
- A socket for sending and receiving datagram packets.
- Constructor and Methods

- DatagramSocket(int port): Constructs a datagram socket and binds it to the specified port on the local host machine

- void receive(DatagramPacket p)
- void send(DatagramPacket p)
- void close()
- For receiving:
- **`DatagramPacket( byte[] buf, int len);`**
- For sending:
- **`DatagramPacket(byte[] buf, int len,InetAddress a,int port);`**

Example : Server Program

```
•   import java.io.*;
•   import java.net.*;
•   class UDPServer
•     {
•       public static void main(String args[]) throws Exception
•         {
•           DatagramSocket serverSocket = new DatagramSocket(9876);
•           byte[] rd = new byte[1024];
•           DatagramPacket rp= new DatagramPacket(rd, rd.length);
•           serverSocket.receive(rp);
•           String sentence = new String( rp.getData());
•           System.out.println("RECEIVED: " + sentence);
•         }  }
```

Client Program

```
import java.io.*;
import java.net.*;
class UDPClient
{
  public static void main(String args[]) throws Exception
  {
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress = InetAddress.getByName("localhost");
    byte[] sd = new byte[1024];
    String msg = "This is UDP message";
    sd = msg.getBytes();
    DatagramPacket sp = new DatagramPacket(sd, sd.length, IPAddress, 9876);
    clientSocket.send(sp);
    clientSocket.close();
  }
}
```