PRESENTATION ON CORE JAVA

Syllabus

- **Java Programming Fundamentals**
- **Introducing Data Types and Operators**
- **Program Control Statements**
- **Introducing Classes, Objects and Methods**
- **More Data Types and Operators**
- **String Handling**
- **A Closer Look at Methods and Classes**

- Inheritance
- Interfaces
- Packages
- Exception Handling
- Multithreaded Programming
- Enumerations, Auto boxing and Annotations
- Generics
- Applets
- Swing Fundamentals
- Networking with Java.net
- Exploring Collection Framework

- **The History and Evolution of Java**
- programming language
- **Machine Language**
- **Assembly Language**
- BASIC, COBOL, FORTRAN
- The Birth of Modern Programming: C 1970(Dennis Richie)
- Because of complexity, C++ came in 1979 by Bjane Strostup: The Next Step(OOPS)
- The Stage Is Set for Java

  Java was conceived by **James Gosling** and 5 other in 1991.

  "Oak," but was renamed "Java" in 1995.

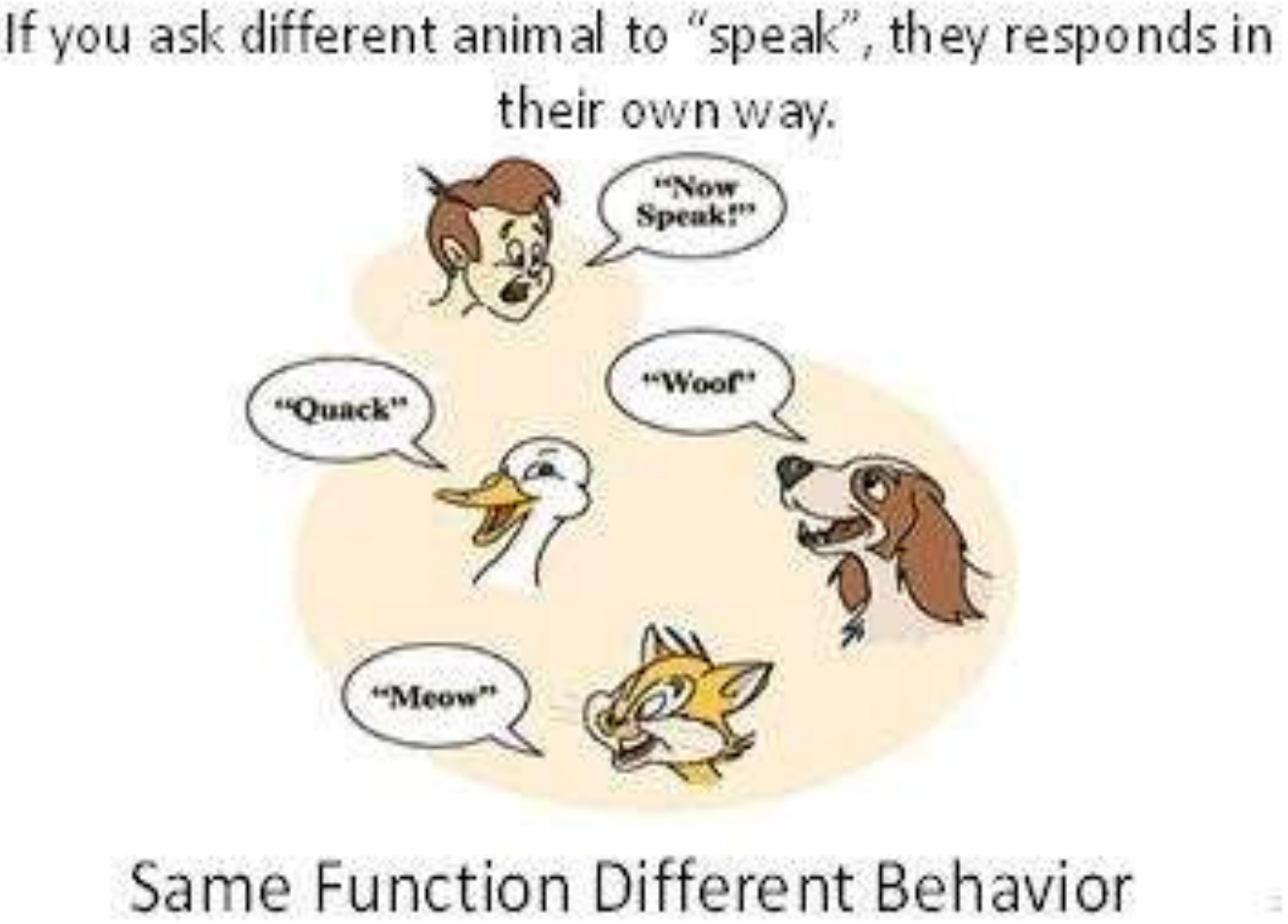
  "Internet version of C++."
- The Creation of Java
- The C# Connection
- How Java Changed the Internet

- Java Applets
- Security
- Portability

- **The key attributes of Object oriented Programming:**

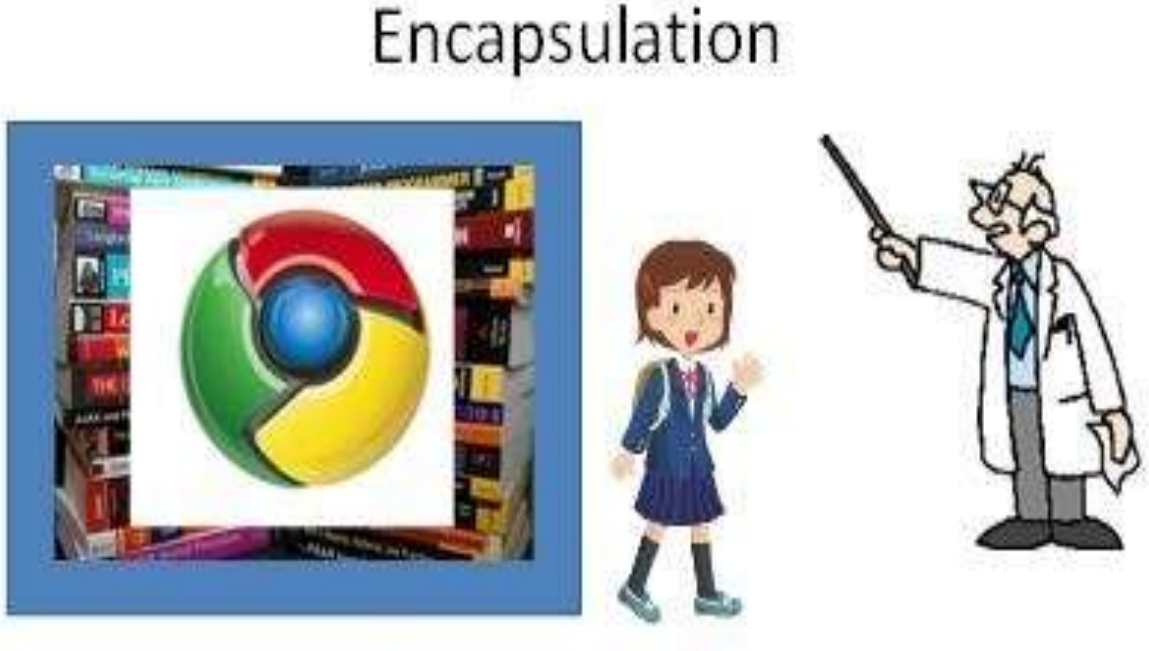
Polymorphism

"Same Function different behavior"



Encapsulation

"Hiding the Unnecessary"

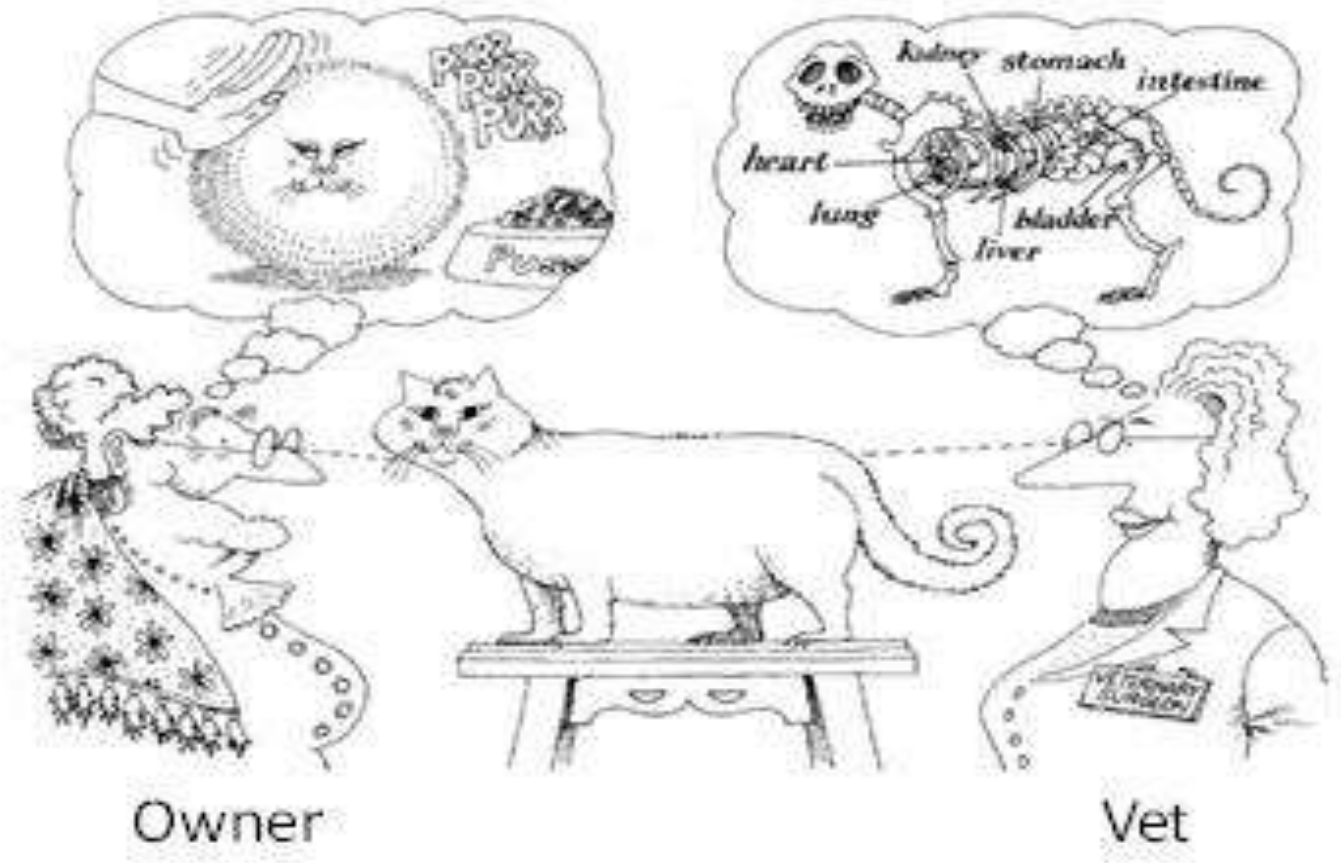


Inheritance

"Modeling the Similarity"

Abstraction

"Eliminate the Irrelevant, Amplify the Essential"



Java Development Kit (JDK)

- The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.
-

```
public class Hello { // Save as "Hello.java"
public static void main(String[] args)
{
System.out.println("Hello, world!");
// print message
}
}
```

- **<u>Common Syntax Errors in Java</u>**
  (*compile time errors*)
- The following list describes some errors that are easy to make in writing Java ... class name does not match file name
  (usually this is due to uppercase vs. lower case mistake or simple typo)
- misspelled variable name
  (use of variable name does not match name in its declaration)
- missing semicolon after assignment or method call statement
- missing semicolon after variable definition
- missing semicolon after import statement

- **Identifiers in Java**
- *Identifiers* are the names of variables, methods, classes, packages and interfaces

II Java's Primitive Types

- Integers This group includes **byte, short, int, and long, which are for whole-valued**
- signed numbers.
- • Floating-point numbers This group includes **float and double, which represent**
- numbers with fractional precision.

- Characters This group includes **char, which represents symbols in a character set,**
- like letters and numbers.
- • Boolean This group includes **boolean, which is a special type for representing**
- true/false values.

- Integers
- Name Width Range
- long 64 –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- int 32 –2,147,483,648 to 2,147,483,647
- short 16 –32,768 to 32,767
- byte 8 –128 to 127
- Let’s look at each type of integer.

- **Floating Point Types**
- Name Width in Bits Approximate Range
- double 64 4.9e–324 to 1.8e+308
- float 32 1.4e–045 to 3.4e+038
- **Booleans**
- Java has a primitive type, called **boolean, for logical values. It can have only one of two**
- possible values, **true or false.**

- **Integer Literals**
- **Floating-Point Literals**
- **Boolean Literals**
- **Character Literals**
- **String Literals**
- **Variables**
- *type identifier [= value][, identifier [= value] ...] ;*

- Declaration of variable
- // Demonstrate dynamic initialization.
- class DynInit {
- public static void main(String args[]) {
- double a = 3.0, b = 4.0;
- // c is dynamically initialized
- double c = Math.sqrt(a * a + b * b);
- System.out.println("Hypotenuse is " + c);
- }
- }

```
class Scope {
public static void main(String args[]) {
int x; // known to all code within main
x = 10;
if(x == 10) { // start new scope
int y = 20; // known only to this block
// x and y both known here.
System.out.println("x and y: " + x + " " + y);
x = y * 2;
}
// y = 100; // Error! y not known here
// x is still known here.
System.out.println("x is " + x);
}
}
```

- Operators

The Basic Arithmetic Operators

- **1 Arithmetic Compound Assignment Operators**
- **2. Increment and decrement operators**
- **The Bitwise Operators**
- **Relational Operators**
- **The Assignment Operator**
- **The ? Operator**

- **Type conversion in Assignments**
- 1.Widening conversion or Automatic
- 2. Narrowing Conversion
- **Java's Automatic Conversions**
- When one type of data is assigned to another type of variable, an *automatic type conversion*
- will take place if the following two conditions are met:
- • The two types are compatible.
- • The destination type is larger than the source type.
- Eg. byte a = 40;
- byte b = 50;
- byte c = 100;
- int d = a * b / c;

- Narrowing conversion
- Syntax
- *(target-type) value*
- *Eg.*
- int a;
- byte b;
- // …
- b = (byte) a;

- **The Type Promotion Rules**
- if one operand is a **long, the whole expression is promoted to long. If one operand is a float, the entire**
- expression is promoted to **float. If any of the operands is double, the result is double.**

- **Operator precedence**
- Highest
- ( ) [ ] .
- ++ – – ~ !
- * / %
- + –
- >> >>> <<
- > >= < <=
- == !=
- &
- ^
- |
- &&
- ||
- ?:
- = op=
- Lowest

- **Input characters from the Keyword**
- The Scanner class can be found in the **java.util** library.

- import java.util.Scanner;
- public class StringVariables {
-
  }

- **Reading data from keyboard:**
- There are many ways to read data from the keyboard. For example:
- InputStreamReader
- Console
- Scanner
- DataInputStream etc

- **ADVANCED**
- The usual way to step through all the elements of an array in order is with a "standard" for loop, for example,
- for (int i = 0; i < myArray.length; i++) { System.out.println(myArray[i]); } The so-called **enhanced for loop** is a simpler way to do this same thing. (The colon in the syntax can be read as "in.")
- for (int myValue : myArray) { System.out.println(myValue); }

- Enhanced for loop java: Enhanced for loop is useful when scanning the array instead of using for loop. Syntax of enhanced for loop is:
  for (data_type variable: array_name)
  Here array_name is the name of array.
- **Java enhanced for loop integer array**
- **class** EnhancedForLoop { **public static void** main(String[] args) { **int** primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29}; **for** (**int** t: primes) { System.out.println(t); } } }
- Download Enhanced for loop program.
- Output of program:

- **Java enhanced for loop strings**
- **class** EnhancedForLoop { **public static void** main(String[] args) { String languages[] = { "C", "C++", "Java", "Python", "Ruby"}; **for** (String sample: languages) { System.out.println(sample); } } }

IV unit

- **Class Fundamentals**
- **The General Form of a Class**
- Aclass is declared by use of the **class keyword.** A simplified general form of a **class**
- class *classname {*
- *type instance-variable1;*
- *type instance-variable2;*

- *// ...*
- *type instance-variableN;*
- *type methodname1(parameter-list) {*
- // body of method
- }
- *type methodname2(parameter-list) {*
- // body of method
- }
- *// ...*
- *type methodnameN(parameter-list) {*
- // body of method
- }
- }

- Good example yourself
- **Declaring Objects**
- Box mybox = new Box();
- This statement combines the two steps just described. It can be rewritten like this to show
- each step more clearly:
- Box mybox; // declare reference to object
- mybox = new Box(); // allocate a Box object

- **Introducing Methods**
- This is the general form of a method:
- *type name(parameter-list) {*
- // body of method
- }

- **Adding a Method That Takes Parameters**
- int square()
- {
- return 10 * 10;
- }
- While this method does, indeed, return the value of 10 squared, its use is very limited.
- However, if you modify the method so that it takes a parameter, as shown next, then you
- can make **square( ) much more useful.**
- int square(int i)
- {
- return i * i;
- }

- **Constructors**
- A*constructor initializes an object immediately upon creation. It has the same name as the*
- class in which it resides and is syntactically similar to a method. Once defined, the constructor
- is automatically called immediately after the object is created, before the **new operator completes.**
- Constructors look a little strange because they have no return type, not even **void. This is**
- because the implicit return type of a class' constructor is the class type itself. It is the constructor's
- job to initialize the internal state of an object so that the code creating an instance will have
- a fully initialized, usable object immediately.

- **Parameterized Constructors**
- /* Here, Box uses a parameterized constructor to
- initialize the dimensions of a box.
- */
- class Box {
- double width;
- double height;
- double depth;
- // This is the constructor for Box.
- Box(double w, double h, double d) {
- width = w;
- height = h;
- depth = d;
- }
- // compute and return volume
- double volume() {
- return width * height * depth;
- }
- }

```
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

The output from this program is shown here:

```
Volume is 3000.0
```

- **The this Keyword**
- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines
- the **this keyword. this can be used inside any method to refer to the *current object. That is,***
- **this is always a reference to the object on which the method was invoked. You can use this**
- anywhere a reference to an object of the current class' type is permitted.
- To better understand what **this refers to, consider the following version of Box( ):**
- // A redundant use of this.
- Box(double w, double h, double d) {
- this.width = w;
- this.height = h;
- this.depth = d;
- }

- **Instance Variable Hiding**
- // Use this to resolve name-space collisions.
- Box(double width, double height, double depth) {
- this.width = width;
- this.height = height;
- this.depth = depth;
- }

- **Arrays**
- An *array* is a group of like-typed variables that are referred to by a common name.
- **One-Dimensional Arrays**
- A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first
- must create an array variable of the desired type. The general form of a one-dimensional
- array declaration is
- *type var-name*[ ];
- Here, *type* declares the base type of the array. The base type determines the data type of each
- element that comprises the array. Thus, the base type for the array determines what type of
- data the array will hold. For example, the following declares an array named **month_days**
- with the type “array of int”:
- int month_days[];

```
// Average an array of values.
class Average {
public static void main(String args[]) {
double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
double result = 0;
int i;
for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
}
}
```

- **Multidimensional Arrays**
- In Java, *multidimensional arrays are actually arrays of arrays. These, as you might expect, look* and act like regular multidimensional arrays.
- int twoD[][] = new int[4][5];
- This allocates a 4 by 5 array and assigns it to **twoD. Internally this matrix is implemented as** an *array of arrays of* ***int.***

```
// Demonstrate a two-dimensional array.
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

- **Assigning an array reference to another array in Java**
- up vote 2 down vote favorite
- int a[]={1, 2, 3, 4, 5}; int b[]={4, 3, 2, 1, 0}; a=b; System.out.println("a[0] = "+a[0]); This displays a[0] = 4 as obvious because a is assigned a reference to b.

- **Alternative Array Declaration Syntax**
- There is a second form that may be used to declare an array:
- *type[ ] var-name;*
- Here, the square brackets follow the type specifier, and not the name of the array variable.
- For example, the following two declarations are equivalent:
- int al[] = new int[3];
- int[] a2 = new int[3];
- The following declarations are also equivalent:
- char twod1[][] = new char[3][4];
- char[][] twod2 = new char[3][4];
- This alternative declaration form offers convenience when declaring several arrays at the
- same time. For example,
- int[] nums, nums2, nums3; // create three arrays
- creates three array variables of type **int. It is the same as writing**
- int nums[], nums2[], nums3[]; // create three arrays

- when you create a **String object, you are creating a string that**
- cannot be changed. That is, once a **String object has been created, you cannot change the**
- characters that comprise that string. At first, this may seem to be a serious restriction. However,
- such is not the case. You can still perform all types of string operations. The difference is that
- each time you need an altered version of an existing string, a new **String object is created**
- that contains the modifications. The original string is left unchanged. This approach is used
- because fixed, immutable strings can be implemented more efficiently than changeable ones.
- For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer**
- and **StringBuilder. Both hold strings that can be modified after they are created.**
- The **String, StringBuffer, and StringBuilder classes are defined in java.lang. Thus, they**
- are available to all programs automatically. All are declared **final, which means that none of**
- these classes may be subclassed. This allows certain optimizations that increase performance
- to take place on common string operations. All three implement the **CharSequence interface.**
- One last point: To say that the strings within objects of type **String are unchangeable means**
- that the contents of the **String instance cannot be changed after it has been created. However,**
- a variable declared as a **String reference can be changed to point at some other String object**
- at any time.

- **Character Extraction**
- **charAt( )**
- **getChars( )**
- **getBytes( )**
- **toCharArray( )**

- **String Comparison**

equals( ) and equalsIgnoreCase( )

startsWith( ) and endsWith( )

equals( ) Versus ==

- **compareTo( )**
- which will construct a new copy of the string with your modifications complete.
- **substring( )**
- You can extract a

Overloading of methods

- When an overloaded method is invoked, Java uses the type and/or number of
- arguments as its guide to determine which version of the overloaded method to actually
- call. Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is
- insufficient to distinguish two versions of a method.

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
```

```
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
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;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, **test( ) is overloaded four times. The first**

- **Overloading constructors**
- /* Here, Box defines three constructors to initialize
- the dimensions of a box various ways.
- */
- class Box {
- double width;
- double height;
- double depth;
- // constructor used when all dimensions specified
- Box(double w, double h, double d) {
- width = w;
- height = h;
- depth = d;
- }
- // constructor used when no dimensions specified
- Box() {
- width = -1; // use -1 to indicate
- height = -1; // an uninitialized
- depth = -1; // box
- }

```
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
```

- vol = mybox1.volume();
- System.out.println("Volume of mybox1 is " + vol);
- // get volume of second box
- vol = mybox2.volume();
- System.out.println("Volume of mybox2 is " + vol);
- // get volume of cube
- vol = mycube.volume();
- System.out.println("Volume of mycube is " + vol);
- }
- }
- The output produced by this program is shown here:
- Volume of mybox1 is 3000.0
- Volume of mybox2 is -1.0
- Volume of mycube is 343.0

- Static
- • They can only call other **static methods.**
- • They must only access **static data.**
- • They cannot refer to **this or super in any way.**
- // Demonstrate static variables, methods, and blocks.
- class UseStatic {
- static int a = 3;
- static int b;
- static void meth(int x) {

- System.out.println("x = " + x);
- System.out.println("a = " + a);
- System.out.println("b = " + b);
- }
- static {
- System.out.println("Static block initialized.");
- b = a * 4;
- }
- public static void main(String args[]) {
- meth(42);
- }
- }
- As soon as the **UseStatic class is loaded, all of the static statements are run. First, a is set to 3,**
- then the **static block executes, which prints a message and then initializes b to a * 4 or 12. Then**
- **main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the**
- two **static variables a and b, as well as to the local variable x.**
- Here is the output of the program:
- Static block initialized.
- x = 42
- a = 3
- b = 12

- **Introducing Nested and Inner Classes**
- It is possible to define a class within another class; such classes are known as *nested classes.*
- // Demonstrate an inner class.
- class Outer {
- int outer_x = 100;
- void test() {
- Inner inner = new Inner();
- inner.display();
- }
- // this is an inner class

```
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

- **Varargs: Variable-Length Arguments**
- This feature is called *varargs and it is*
- short for *variable-length arguments. A method that takes a variable number of arguments*
- is called a *variable-arity method, or simply a varargs method.*

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
static void vaTest(int v[]) {
System.out.print("Number of args: " + v.length
+
" Contents: ");
for(int x : v)
System.out.print(x + " ");
```

```
}
public static void main(String args[])
{
// Notice how an array must be created to
// hold the arguments.
int n1[] = { 10 };
int n2[] = { 1, 2, 3 };
int n3[] = { };
vaTest(n1); // 1 arg
vaTest(n2); // 3 args
vaTest(n3); // no args
}
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

- Avariable-length argument is specified by three periods (**...). For example, here is how**
- **vaTest( ) is written using a vararg:**
- static void vaTest(int ... v) {
- This syntax tells the compiler that **vaTest( ) can be called with zero or more arguments. As a** result, **v is implicitly declared as an array of type int[ ]. Thus, inside vaTest( ), v is accessed**
- using the normal array syntax. Here is the preceding program rewritten using a vararg:
- // Demonstrate variable-length arguments.
- class VarArgs {
- // vaTest() now uses a vararg.
- static void vaTest(int ... v) {
- System.out.print("Number of args: " + v.length +
- " Contents: ");
- for(int x : v)
- System.out.print(x + " ");
- System.out.println();
- }
- public static void main(String args[])
- {

- // Notice how vaTest() can be called with a
- // variable number of arguments.
- vaTest(10); // 1 arg
- vaTest(1, 2, 3); // 3 args
- vaTest(); // no args
- }
- }

- **Inheritance Basics**
- To inherit a class, you simply incorporate the definition of one class into another by using
- the **extends keyword. The following program**
- creates a superclass called **A and a subclass called B. Notice how the keyword extends is**
- used to create a subclass of **A.**
- // A simple example of inheritance.
- // Create a superclass.
- class A {
- int i, j;
- void showij() {
- System.out.println("i and j: " + i + " " + j);
- }
- }
- // Create a subclass by extending class A.
- class B extends A {
- int k;
- void showk() {
- System.out.println("k: " + k);
- }

```
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
```

- subOb.i = 7;
- subOb.j = 8;
- subOb.k = 9;
- System.out.println("Contents of subOb: ");
- subOb.showij();
- subOb.showk();
- System.out.println();
- System.out.println("Sum of i, j and k in subOb:");
- subOb.sum();
- }
- }
- The output from this program is shown here:
- Contents of superOb:
- i and j: 10 20
- Contents of subOb:
- i and j: 7 8
- k: 9
- Sum of i, j and k in subOb:
- i+j+k: 24

- The general form of a **class declaration that inherits a superclass is shown here:**
- class *subclass-name extends superclass-name {*
- // body of class
- }

- **Member Access and Inheritance**
- Although a subclass includes all of the members of its superclass, it cannot access those
- members of the superclass that have been declared as **private. For example, consider the**
- following simple class hierarchy:
- /* In a class hierarchy, private members remain
- private to their class.
- This program contains an error and will not
- compile.
- */
- // Create a superclass.
- class A {
- int i; // public by default
- private int j; // private to A

```
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

- This program will not compile because the reference to **j inside the sum( ) method of B**
- causes an access violation. Since **j is declared as private, it is only accessible by other members** of its own class. Subclasses have no access to it.

- **Using super**
- **Using super to Call Superclass Constructors**
- Asubclass can call a constructor defined by its superclass by use of the following form of **super:**
- super(*arg-list);*
- Here, *arg-list specifies any arguments needed by the constructor in the super*

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
```

```
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
```

```
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;
vol = mybox1.volume();
```

```
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();
vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();
vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
}
```

- This program generates the following output:
- Volume of mybox1 is 3000.0
- Weight of mybox1 is 34.3
- Volume of mybox2 is 24.0
- Weight of mybox2 is 0.076
- Volume of mybox3 is -1.0
- Weight of mybox3 is -1.0
- Volume of myclone is 3000.0
- Weight of myclone is 34.3
- Volume of mycube is 27.0
- Weight of mycube is 2.0

- **A Second Use for super**
- The second form of **super acts somewhat like this, except that it always refers to the superclass**
- of the subclass in which it is used. This usage has the following general form:
- super.*member*
- Here, *member can be either a method or an instance variable.*
- This second form of **super is most applicable to situations in which member names of**
- a subclass hide members by the same name in the superclass. Consider this simple class
- hierarchy:
- // Using super to overcome name hiding.
- class A {
- int i;
- }// Create a subclass by extending class A.
- class B extends A {
- int i; // this i hides the i in A
- B(int a, int b) {
- super.i = a; // i in A
- i = b; // i in B
- }
- void show() {
- System.out.println("i in superclass: " + super.i);
- System.out.println("i in subclass: " + i);
- }
- }

- class UseSuper {
- public static void main(String args[]) {
- B subOb = new B(1, 2);
- subOb.show();
- }
- }
- This program displays the following:
- i in superclass: 1
- i in subclass: 2
- Although the instance variable **i in B hides the i in A, super allows access to the i defined**
- in the superclass. As you

- This program displays the following:
- i in superclass: 1
- i in subclass: 2
- Although the instance variable **i in B hides the i in A, super allows access to the i defined**
- in the superclass. As you will see, **super can also be used to call methods that are hidden by a**
- subclass.

- **Creating a Multilevel Hierarchy**
- As mentioned, it is perfectly acceptable to use a subclass as a superclass of another.
- For example, given three classes called **A, B, and C, C can be a subclass of B, which is a**
- subclass of **A. When this type of situation occurs, each subclass inherits all of the traits**
- found in all of its superclasses. In this case, **C inherits all aspects of B and A. To see how**
- a multilevel hierarchy can be useful, consider the following program. In it, the subclass
- **BoxWeight is used as a superclass to create the subclass called Shipment. Shipment inherits**
- all of the traits of **BoxWeight and Box, and adds a field called cost, which holds the cost of**
- shipping such a parcel.
- // Extend BoxWeight to include shipping costs.
- // Start with Box.
- class Box {
- private double width;
- private double height;
- private double depth;
- // construct clone of an object
- Box(Box ob) { // pass object to constructor
- width = ob.width;
- height = ob.height;
- depth = ob.depth;
- }
- // constructor used when all dimensions specified
- Box(double w, double h, double d) {
- width = w;
- height = h;
- depth = d;
- }

```
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// Add weight.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
```

```
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}
```

```
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
// Add shipping costs.
class Shipment extends BoxWeight {
double cost;
// construct clone of an object
Shipment(Shipment ob) { // pass object to constructor
super(ob);
cost = ob.cost;
}
```

```
// constructor when all parameters are specified
Shipment(double w, double h, double d,
double m, double c) {
super(w, h, d, m); // call superclass constructor
cost = c;
}
// default constructor
Shipment() {
super();
cost = -1;
}
// constructor used when cube is created
Shipment(double len, double m, double c) {
super(len, m);
cost = c;
}
}
```

```
class DemoShipment {
public static void main(String args[]) {
Shipment shipment1 =
new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 =
new Shipment(2, 3, 4, 0.76, 1.28);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is "
+ shipment1.weight);
System.out.println("Shipping cost: $" +
shipment1.cost);
System.out.println();
```

- vol = shipment2.volume();
- System.out.println("Volume of shipment2 is " + vol);
- System.out.println("Weight of shipment2 is "
- + shipment2.weight);
- System.out.println("Shipping cost: $" + shipment2.cost);
- }
- }
- The output of this program is shown here:
- Volume of shipment1 is 3000.0
- Weight of shipment1 is 10.0
- Shipping cost: $3.41
- Volume of shipment2 is 24.0
- Weight of shipment2 is 0.76
- Shipping cost: $1.28

- **When Constructors Are Called**
- If **super( ) is not used, then the default or parameterless constructor of each superclass**
- will be executed. The following program illustrates when constructors are executed:
- // Demonstrate when constructors are called.
- // Create a super class.
- class A {
- A() {
- System.out.println("Inside A's constructor.");
- }
- }

```
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

The output from this program is shown here:

```
Inside A’s constructor
Inside B’s constructor
Inside C’s constructor
```

- Super class reference and subclass objects
- class A {
- void callme() {
- System.out.println("Inside A's callme method");
- }
- }
- class B extends A {
- // override callme()
- void callme() {
- System.out.println("Inside B's callme method");
- }
- }
- class C extends A {
- // override callme()
- void callme() {
- System.out.println("Inside C's callme method");
- }
- }
- class Dispatch {
- public static void main(String args[]) {
- A a = new A(); // object of type A
- B b = new B(); // object of type B
- C c = new C(); // object of type C
- A r; // obtain a reference of type A

- r = a; // r refers to an A object
- r.callme(); // calls A's version of callme
- r = b; // r refers to a B object
- r.callme(); // calls B's version of callme
- r = c; // r refers to a C object
- r.callme(); // calls C's version of callme
- }
- }
- The output from the program is shown here:
- Inside A’s callme method
- Inside B’s callme method
- Inside C’s callme method

- Method Overriding
- In a class herarchy, when a method in a subclass has the same return type and signature as a method in a superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is caled from within a subclass, it will always refer to the version of that method defined in the subclass.

```
Class A
{
Int I, j;
A(int a , int b)
{i=a;
J=b;
}
}
```

```
Class B extends A
{
Int k;
B(int a, int b, int c)
{
Super(a,b);
K=c;
}
Void show()
{
```

- System.out.println(“k:”+k);
- }
- }
- Class Override
- {
- Public static void main(String [] args)
- {
- B subob=new B(1,2,3);
- Subobj.show();
- }
- }

- Output
- K:3
- **Overridden method support polymorphism**
- Method overriding forms the basis for one of Java's most powerful concepts: dynamic methodnamic mehtod dispatch is the mechanism by which a call to an overridden method is resolved at run time rather than compile time.

```
Class Sup
{
Void who()
{
System.out.println(“who() in Sup”);
}
}
Class Sub1 extends Sup
{
```

- Void who()
- {
- System.out.println(“who() in Sub1”);
- }
- }
- Class Sub2 extends Sup
- {
- Void who()
- {
- System.out.println(“who() in Sub2”);

- }
- }

- Class DynDispDmo
- {
- Public static void main(String[] args)
- {
- Sup superOb=new Sup();
- Sub1 Subob1=new Sub1();
- Sub2 suob2=new Sub2();
- Sup supref;
- Supref=superOb;
- Supref.who();

- Supref=subob1;
- Supref.who();
- Supref=subob2;
- Supref.who();
- }
- }
- Who() in sup
- Who() in sub1
- Who() in sub2

- **Using Abstract Classes**
- There are situations in which you will want to define a superclass that declares the structure
- of a given abstraction without providing a complete implementation of every method. That
- is, sometimes you will want to create a superclass that only defines a generalized form that
- will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a
- class determines the nature of the methods that the subclasses must implement.

- To declare an
- abstract method, use this general form:
- abstract *type name(parameter-list);*
- // A Simple demonstration of abstract.
- abstract class A {
- abstract void callme();
- // concrete methods are still allowed in abstract classes
- void callmetoo() {
- System.out.println("This is a concrete method.");
- }
- }
- class B extends A {
- void callme() {
- System.out.println("B's implementation of callme.");
- }
- }
- class AbstractDemo {
- public static void main(String args[]) {
- B b = new B();
- b.callme();
- b.callmetoo();
- }
- }

```
// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
```

```
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
```

- figref = t;
- System.out.println("Area is " + figref.area());
- }
- }

- **Using final with Inheritance**
- The keyword **final has three uses**
- **Using final to Prevent Overriding**
- While method overriding is one of Java's most powerful features, there will be times when
- you will want to prevent it from occurring. To disallow a method from being overridden,
- specify **final as a modifier at the start of its declaration. Methods declared as final cannot**
- be overridden. The following fragment illustrates **final:**
- class A {
- final void meth() {
- System.out.println("This is a final method.");
- }
- }
- class B extends A {
- void meth() { // ERROR! Can't override.
- System.out.println("Illegal!");
- }
- }

- **Using final to Prevent Inheritance**
- We can prevent a class rom bein inherited by preceding its declaration with final. **Declaring a class as final implicitly declares all of its methods** as **final, too**
- Here is an example of a **final class:**
- final class A {
- // ...
- }
- // The following class is illegal.
- class B extends A { // ERROR! Can't subclass A
- // ...
- }

- **Using final with Data Members**
- Final can also be appled to member variables to create what amounts to named constants. If we prcede a class variable’s name with final, its value cannot be chaned throughout the lifetime of our program.
- class finall
- {

- public static void main(String args[])
- {
- final int a=10;
- a=12;
- System.out.println("a is"+a);
- }

- }

- **The Object Class**
- One special class, Object defined by Java. All other classes are subclasses of Object. Means Object is the superclass of all other classes. This means that reference variable of type Object can refer to an object of any other class.

- Following are the methods.
- Object clone() Creates a new object that is same as the object being cloned.
- Boolean equals(Object object) Determine whether one object equal to another
- void finalize() Called before an unused object is recycled.
- Class getClass() Obtains the class of an object at run time.
- int hashcode() Returns hashcode associated with the invoking object.

- void notify( ) Resumes execution of a thread waiting on the invoking object.
- void notifyAll( ) Resumes execution of all threads waiting on the invoking object.
- String toString( ) Returns a string that describes the object.
- void wait( ) waits on another thread for execution
- void wait(long milliseconds)
- void wait(long milliseconds,
- int nanoseconds)

- **Interfaces:**
- Using the keyword interface you can fully abstract a class interface from its implementation.
- Defining an Interface
- Access interface name
- {
- Return-type method-name1(parameter-list);
- Return-type method-name2(parameter-list);
- Type final-varname1=value;
- Type final-varname2=value;
- Retun-type mehod-nameN(parameter-list);
- }
- }
- When no access specifier is included then default access results. Name should be valid identifier. And methods are declared have no bodies. They are abstract methods. They are implicitly final and static.

- **Creating an interface and Implementing interface:**
- General form is
- Class classname [extends superclass] [implements interface1, interface2
- {
- //class-body
- }
- If we are implementing two interface we have separate it with comma. The methods that implement an interface must be declared as public.

- eg.
- interface look
- {
- void show(int a);
- }
- class hi implements look
- {
- public void show(int b)
- {
- System.out.println("interface variable"+b);
- }

```
}
class inter
{
public static void main(String args[])
{
hi obj=new hi();
obj.show(10);
}
}
```

Output:

interface variable10

- Partial implementation
- Sometimes if we don’t want to implement the method in interface or we want again partial implementation we can make the subclass as abstract.

- Eg.
- interface look
- {
- void show(int a);
- }
- abstract class hi implements look
- {

- public void display()
- {
- System.out.println("I am in abstract class");
- }
- }
- class inter
- {
- public static void main(String args[])
- {

- }
- }

- **Using interface reference**
- Same like class from interface also we can create references.
- Interface look
- {
- void show(int a);
- }
- class hi implements look
- {
- public void show(int b)
- {
- System.out.println("interface variable"+b);
- }

```
}
class inter
{
public static void main(String args[])
{
look ref;
hi obj=new hi();
ref=obj;
ref.show(10);
}
}
```

- **Implementing Multiple interface:**
- A class can implement more than one interface. To do, simply specify each interface in a comma-separated list. The class must implement all of the methods specified by each interace.
- Eg.
- Interface IA
- {
- Void doSomething();
- }
- Interface IB
- {
- Void doSomethingElse();
- }

- Class Myclass implements IfA, IfB
- {
- Public void doSomething()
- {
- System.out.println(“Doing Something”);
- }
- Public void doSomethingElse()
- {
- System.out.println(“Doing Something else”);
- }
- }
- Class MultiImpDemo
- {
- Public static void main(String args[])

- IfA aRef;
- IfB bRef;
- MyClass obj=nre MyClass();
- aRef=obj;
- aRef.doSomething();
- bRef=obj;
- bRef.doSomethingElse();
- }
- }
- Output
- Do something
- Do somethingelse

- **Constants in interface**
- The primary purpose of interface to declare methods. It can also include variables. Variables are not an instance. They are implicitly public, final and static and must be initialized. Large rograms often make use of several constant values that describe such things as array size, various limits, and special values.
- Eg.
- Interace Iconst
- {
- Int MIN=0;
- Int MAX=10;
- String ERRRORMSG="Boundary Error";
- }

```
Class IConstDemo implements Iconst
{
Public static void main(String[] args)
{
Int[] nums=new int[MAX];
for(int i=MIN;i<(MAX+1);i++)
{
If(i>=MAX)
System.out;println(ERRORMSG);
Else
{
Nums[i]=i
System.out.print(nums[i]+"");
}}}
```

- **Interfaces can be extended**
- One interface can inherit another by use of the keyword extends. The syntax is the same as or inheritin classes. When a class implements an interface that inherits another inrface, it mus provide implementations for all methods defined within the interace inheritance chain.
- Eg.
- interface A
- {
- void meth1();
- void meth2();
- }
- interface B extends A
- {
- Void meth3();
- }

```
class MyClass implements B
{
public void meth1()
{
System.out.println(“Implements meth1().”);
}
public void meth2()
{
System.out.println(“Implements meth2().”);
}
public void meth3()
{
System.out.println(“Implements meth3().”);
} }
```

```
class IfExtend
{
public static void main(String args[])
{
MyClass ob=new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();


}
```

- **Nested Interface:**
-
-
- An interface can be decalred a memberof another interface or of a class. Such an interface called a member interface or a nested interface. An interface nested in a class can use any access modifier. An interface nested in another interace is implicitly public.
- // A nested interface emaple.
- //This interface contains a nested interface.

```
Interface A
{
Public interface NestedIF
{
boolean isNotNegative(int x);
}
Void  doSomething();
}

 class B implements A.NestedIF
{
```

- Public Boolean isNotNegative(int x)
- {
- Return x<0? false:true;
- }
- }
- Class NestedIFDemo
- {
- Public static void main(String args[])
- {
- //use a nested interface reference
- A.NestedIF nif= new B();
-

- If(nif.isNotNeative(10))
- System.out.println(“10 is not negative”);
- If(nif..isNotNegative(-12))
- System.out.println(“this won’t be displayed”);
- }
- }

- **PACKAGES**
- **Package fundamentals**
- Packages are containers for classes that are used to keep the class name space compartmentalized. In other words, to avoid collision of using same class name we can use packages. We can use different class name in different packages. But we can't use same class name in same package.
- Classes defined within a package can be made private to aht packae and not accessible by outside code.
- **Defining a Package:**
- General form is
- package pkg;
- We can define multilevel package like this.
- package pakg1.pkg2.pkg3;
- package names are case sensitive.

- Eg.
- Package bookback;
- Class Book
- {
- Private String title;
- Private String author;
- Private int pubdate;
- Bood(String t, String a , int d)
- {
- title=t;
- author=a;
- pubdate=d;
- }
- void show()
- {
- System.out.println(title);
- System.out.println(author);
- System.out.println(pudate);
- }
- }
-

- Class BookDemo
- {
- Public static void main(String[] args)
- {
- Book[] books= new Book[5];
- Books[0]= new Book(“Java Fundamentals”, “Shildt”,2014);
- for(int i=0;i<books.lenth;i++)
- {
- Books[i].show();
- System.out.println();
- }

- }
- }
-
- Call this file BookDemo.java and put it in a directory called bookpack.
- Next, compile the file.
- javac bookpack/BookDemo.java
- Then run
- java bookpack.BookDemo
-
-
-
-
-
-
-

- **Access Protection**
-

	private	no modifer	protected	public
Same class	**y**	**y**	**y**	**y**
Same package With nonsubclass	**n**	**y**	**y**	**y**
Same package with Subclass	**n**	**y**	**y**	**y**
Different packagewith subclass	**n**	**n**	**y**	**y**
Different package With non subclass	**n**	**n**	**n**	**y**

Give any example

- **Importing Packages**
- In Java source file import statement is made after package statement.

- This is the general form of import statement.
- Import pkg1.pkg2.classname;
- Here pkg1 is the top level package. Classname is the name of the class used

- For eg.
- import java.util.Date;
- Import java.io.*;

- If we did not import java.util. Date; we have to do like this.
- Class Mydate extends java.util.Date
- {
- }

```
package House;
public class Room
{
public void show()
{
System.out.println("Total rooms are 4");
}
}

package Accessories;
import House.*;
class Furniture
{
void calculate()
{
System.out.println("Totally 4 furniture");
}
}
```

```
class Furnidemo
{
public static void main(String args[])
{
Room obj=new Room();
obj.show();
}
}
```

Output

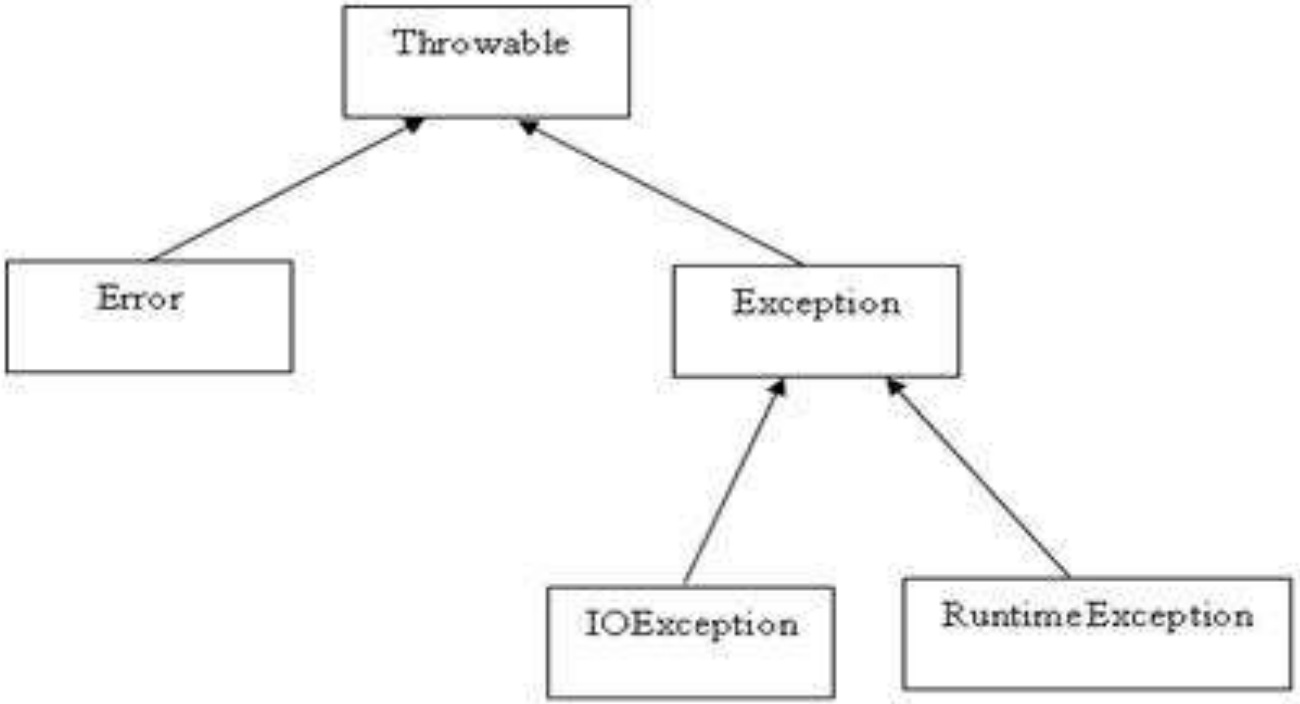
Total rooms are 4

- **Static import**
- JDK 5 added a new feature to Java called static import that expands the capabilities of the
- import keyword. By following import with the keyword static, an import statement can
- be used to import the static members of a class or interface. When using static import, it is
- possible to refer to static members directly by their names, without having to qualify them
- with the name of their class.

- eg.
- import static java.lang.Math.sqrt;
- class Staticimport
- {
- public static void main(String args[])
- {
- System.out.println(sqrt(10));
- }
- }
-
- Ttwo general forms of the import static statement. The first is used, brings into view a single name. Its general form is shown here:
- import static pkg.type-name.static-member-name;
-
-
-
-
-

- The second form of static import imports all static members of a given class or interface.
- Its general form is shown here:
- import static pkg.type-name.*;

- Exception Hierarchy



Uncaught Exceptions

It is necessery that how to handle exceptions in your program, it is useful to see what happens when we don't handle them. It causes a divide-by-zero error:

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 tostop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Any exception that is not caught by your program will ultimately be processed by

the default handler. The default handler displays a string describing the exception, prints a

stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

java.lang.ArithmeticException: / by zero

at Exc0.main(Exc0.java:4)

- **Multiple Catch Clause:**
- In some cases, more than one exception can be raised with a single piece of code. To handle this type of situation , we can specify two or more catch clause, each catching different type of exception.

- Eg.
- import java.lang.*;
- class Multicatch
- {
- public static void main(String args[])
- {
- try
- {
- int a=args.length;
- int b=42/a;
- int c[]={1};
- System.out.println(c[32]);
- }

- catch(ArithmeticException e)
- {
- System.out.println("Division error"+e);
- }
- catch(ArrayIndexOutOfBoundsException e)
- {
- System.out.println("Arrayindex error"+e);
- }
- System.out.println("After try/catch blocks");
- }
- }
- Output is:
- Division error Java.lang.ArithmeticException: / by zero After try/catch blocks

- Here is the output generated by running it both ways:
- C:\>java MultiCatch
- a = 0
- Divide by 0: java.lang.ArithmeticException: / by zero
- After try/catch blocks.
- C:\>java MultiCatch TestArg
- a = 1
- Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
- After try/catch blocks.

- **Catching subclass Exception**
- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. For example, consider the following program:

- /* 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 and catch**
- Here the try statement can be nested. Means, try statement can be inside the block of another try.
- Eg.

```
class Nestr
{
public static void main(String args[])
{
try
{
int a=args.length;
int b=42/a;
try
{
if(a==1)
{
int c=a/(a-a);

}
if(a==2)
```

```
{
int d[]={1};
d[32]=40;
}
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("in nested try block array excn"+e);
}
}
catch(ArithmeticException a)
{
System.out.println("Outside try block array excn"+a);
}
}
}
```

Output:

Here accordint to the error the output is changing.

- **throw**
- we have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:
- throw ThrowableInstance;
- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

- Example
- Class LessBalanceException extends Exception
- {
- Void newAccount(double amount) throws LessBalanceException
- {
- If(amount<500)
- throw new LessBalanceException(“The amount should not be less than 500”);
- }}

- **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. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is the general form of a method declaration that includes a throws clause:
- type method-name(parameter-list) throws exception-list
- {
- // body of method
- }
- Here, exception-list is a comma-separated list of the exceptions that a method can throw. Following is an example of an incorrect program that tries to throw an exception that it does not catch
- // This program contains an error and will not compile.
- class ThrowsDemo {
- static void throwOne() {
- System.out.println("Inside throwOne.");
- throw new IllegalAccessException("demo");
- }
- public static void main(String args[]) {
- throwOne(); } }}

- To make this example compile, you need to make two changes. First, you need to declare that throwOne( ) throws IllegalAccessException. Second, main( ) must define a try/catch statement that catches this exception.
- The corrected example is shown here:
- // This is now correct.
- class ThrowsDemo {
- static void throwOne() throws IllegalAccessException {
- System.out.println("Inside throwOne.");
- throw new IllegalAccessException("demo");
- }
- public static void main(String args[]) {
- try {
- throwOne();
- } catch (IllegalAccessException e) {
- System.out.println("Caught " + e);
- }
- }
- }
- Here is the output generated by running this example program:
- inside throwOne caught java.lang.IllegalAccessException: demo

- **finally**
-
- 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. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:
- // Demonstrate finally.
- class FinallyDemo {
- // Through an exception out of the method.
- static void procA() {
- try {
- System.out.println("inside procA");
- throw new RuntimeException("demo");
- } finally {
- System.out.println("procA's finally");
- }
- }
- // Return from within a try block.

```
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally { System.out.println("procC's finally");
}
}
```

```
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

- **Java's Built-in Exceptions**
- ArithmeticException Arithmetic error, such as divide-by-zero.
- ArrayIndexOutOfBoundsException Array index is out-of-bounds.
- ClassCastException Invalid cast.
- IllegalArgumentException Illegal argument used to invoke a method.
- IllegalMonitorStateException Illegal monitor operation, such as waiting on an unlocked thread.
- IndexOutOfBoundsException Some type of index is out-of-bounds.

- NullPointerException Invalid use of a null reference.
- NumberFormatException Invalid conversion of a string to a numeric format.
- Java's Unchecked RuntimeException Subclasses Defined in java.lang
- Exception Meaning

-
- ClassNotFoundException Class not found.
- IllegalAccessException Access to a class is denied.

- InterruptedException One thread has been interrupted by another thread.
- NoSuchFieldException A requested field does not exist.
- NoSuchMethodException A requested method does not exist.
- Java's Checked Exceptions Defined in java.lang

Creating a Thread

In most general sense, we create thread by instantiating an object of type Thread. Java defines two ways.

1. We can implement the Runnable interface.
2. We can extend the Thread class, itself.

Implementing Runnable

It is the easiest way. To implement Runnable a class need only a single method called run,

syntax is

public void run()

Inside run() , we will define the code that constitutes the new thread. It is important to understand that run() can call other methods. After we create a class that implements Runnable, we will instantiate a object of type Thread within that class. Thread defines several constructors.

Thread(Runnable threadOb, String threadName)

In this constructor threadob is an instance of a class that implements the Runnable interface. The name of new thread is specified by threadName.

Eg.

```
class NewThread implement Runnable
{
Thread t;
NewThread()
{
t=new Thread(this, "demo");
System.out.println("Child thread is"+t);
t.start();
}
public void run()
{
try
{
for(int i=5;i>0;i--)
{
System.out.println("Child thread"+i);
Thread.sleep(500);
}

}
```

```
catch(InterruptedException e)
{
}
System.out.println("Child thread exiting");
}
}
class Newt
{
public static void main(String args[])
{
new NewThread();
try
{
for(int i=5;i>0;i--)
{
System.out.println("Main thread"+i);
Thread.sleep(1000);
}
}
catch(InterruptedException e)
{
}
System.out.println("Main thread exiting");
}
}
```

Output:

```
Child thread is Thread[demo,5,main]
Main thread5
Child thread5
Child thread4
Main thread4
Child thread3
Child thread2
Main thread3
Child thread1
Child thread exiting
Main thread2
Main thread1
Main thread exiting
```

- **Extending Thread:**
- The second way to create thread is to creat a new class that extends Thread, and then to creat an instance of that class. The extending class must override run() method, which is then entry point for the new thread. It must also call start() to begin execution of the new thread.
- EG.
- class Newt2 extends Thread
- {
- Newt2()
- {
- super("New thread");
- System.out.println("Child thread"+this);
- start();
- }
- public void run()
- {
- try
- {
- for(int i=5;i>0;i--)
- {
- System.out.println("Child thread executing"+i);
- Thread.sleep(500);
- }

```
}
catch(InterruptedException e)
{

}
System.out.println("Child thread exiting");
}
}
class Newtext
{
public static void main(String args[])
{
new Newt2();
try
{
for(int i=5;i>0;i--)
{
System.out.println("Main thread executing"+i);
Thread.sleep(1000);
}
}
```

- catch(InterruptedException e)
- {
- }
- System.out.println("Main thread exiting");
- }
- }
- **Creating Multiple Threads**
- This program creates three threads.
- Eg.
- class Newthre implements Runnable
- {
- String str;
- Thread t;
- Newthre(String s)
- {
- //super("demo");
- str=s;
- t=new Thread(this,str);
- System.out.println("Thread name is"+t);
- t.start();
- }

```
public void run()
{
try
{
for(int i=5;i>0;i--)
{
//System.out.println("Child thread is"+t);
System.out.println(str+"thread executing"+i);
Thread.sleep(1000);
}
}
catch(InterruptedException e)
{
}
```

```
System.out.println("Child thread exiting");
}
}
class Newt2
{
public static void main(String args[])
{
//new Newthre();
Newthre obj=new Newthre("demo1");
Newthre obj1=new Newthre("demo2");
Newthre obj2=new Newthre("demo3");
try
{
Thread.sleep(10000);
}
```

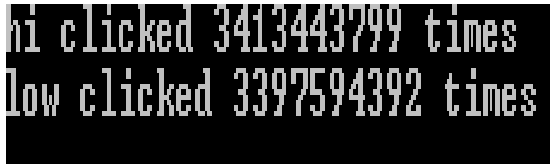
```
catch(InterruptedException e)
{
}
System.out.println("Main thread exiting");
}
}
```

- **Thread Priorites:**
- Higher priority thread get more CPU time than lower priority threads. To set a priority us setPriority() method , member of Thread.
- General form is
- final void setPriority(int level)
- Level means the new priority setting for the calling thread. The value of level must be within the Range MIN_PRIORITY and MAX_PRIORITY. To return default priority, specify NORM_PRIORITY. Is 5.

```
class Newpriority implements Runnable
{
Thread t;
 long click=0;
 boolean running=true;
public Newpriority(int prio)
{
t=new Thread(this);
t.setPriority(prio);
//System.out.prin*+tln("Child thread is "+t);
}
public void run()
{
while(running)
{
click++;
}
}
public void stop()
{
running=false;
}
```

```
public void start()
{
t.start();
}
}
class Newpr
{
public static void main(String args[])
{
//Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
Newpriority hi=new Newpriority(Thread.NORM_PRIORITY+2);
Newpriority lo=new Newpriority(Thread.NORM_PRIORITY-2);
lo.start();
hi.start();
try
{
Thread.sleep(10000);
}
catch(InterruptedException e)
{
```

```
}
lo.stop();
hi.stop();
try
{

hi.t.join();
lo.t.join();

}
catch(InterruptedException e)
{

}
System.out.println("hi clicked "+hi.click+" times");
System.out.println("low clicked "+lo.click+" times");

}



}
```



Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

```
// This program is not synchronized.
class Callme {
```

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

```
class Caller implements Runnable {
String msg;
Callme target;
Thread t;public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run() {
target.call(msg);
}
}
class Synch {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
```

```
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

- Here is the output produced by this program:
- Hello[Synchronized[World]
-]
-]
- If we want to restrict its access to only one thread at a time we simply need to precede call( )'s definition with the keyword synchronized, as shown here:
- class Callme {
- synchronized void call(String msg) {

- This prevents other threads from entering call( ) while another thread is using it. After
- synchronized has been added to call( ), the output of the program is as follows:
- [Hello]
- [Synchronized]
- [World]
- **The synchronized Statement**
- While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.

- This is the general form of the synchronized statement:
- synchronized(object) {
- // statements to be synchronized
- }
-

- // This program uses a synchronized block.
- class Callme {
- void call(String msg) {
- System.out.print("[" + msg);
- try {
- Thread.sleep(1000);
- } catch (InterruptedException e) {
- System.out.println("Interrupted");
- }
- System.out.println("]");
- }
- }

```
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}
}
```

```
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( ) method. This causes the same correct output as the preceding example

- **Interthread Communication**

Multithreading replaces event loop programming by dividing our tasks into discrete, logical units. Threads also provide a secondary benefit, they do away with polling. Polling wastes CPU time.

To avoid polling Java includes an elegant interprocess communication mechnism through the wait(), notify() and notifyAll() methods. **These methods are final methods in Object**. All three methods are called only from within a synchronized context.

wait():tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify()

notify(): wakes up thread that called wait() on the same object.

notifyAll(): wakes up all the threads that called wait() on the same object.

Example same as producer and consumer concept in lab

Suspending, Resuming and Stopping Threads:

If the user doesn't want a thread we can suspend the particular thread for some time and we can resume after some time. The method we are using below:

final void suspend()

final void resume()

```
class susthread implements Runnable
{
Thread t;
String st;
susthread(String str)
{
st=str;
t=new Thread(this,st);
System.out.println("Thread name is"+t);
t.start();
}
public void run()
{
try
{
for(int i=5;i>0;i--)
{
System.out.println(st+" : "+i);
Thread.sleep(200);
}
}
```

```
catch(InterruptedException e)
{
}
}
}
class susthreaddemo
{
public static void main(String args[])
{
susthread football=new susthread("football");
susthread cricket=new susthread("cricket");
try
{
cricket.t.suspend();
Thread.sleep(1000);
cricket.t.resume();
football.t.suspend();
Thread.sleep(1000);
football.t.resume();
}
catch(InterruptedException e)
{
}
```

- }
- }
- Output:
- Thread name isThread[football,5,main]
- Thread name isThread[cricket,5,main]
- football : 5
- football : 4
- football : 3
- football : 2
- football : 1
- cricket : 5
- cricket : 4
- cricket : 3
- cricket : 2
- cricket : 1
- Thread can also stop by using method stop() method.
- final void stop().
- Once we stopped the thread we can’t resume again.
- **Another example producer consumer concept**

- **Enumeration:**
- enum is a datatype with fixed set of constants.
- Eg. Enumeration of the months in the year. An enuermation of the days of the week
- Eg.
- enum name{Manu, kilu, kalu};
- class enn
- {
- public static void main(String args[])
- {

- name obj=name.Manu;
- switch(obj)
- {
- case Manu:
- System.out.println(“handsome man");
- break;
- case kilu:
- System.out.println("kilu ok");

- break;
- case kalu:
- System.out.println(“Lazy man");
- break;
- }
- }
- }
- Output:
- handsome man
- In enum we are using two methods. values() and valueOf().
- Values() are using to display all data in enum and valueOf() is used to take the data by using enum.
- Eg.
- enum apple{red,blue,green}
- class App
- {

- public static void main(String args[])
- {
- apple b;
- b =apple.valueOf("red");
- System.out.println(b.equals(apple.red));

- for(apple a:apple.values())
- {
- System.out.println(a);
- }
- }
- }
- Output

- **Constructors, Methods, Instance variables and enumerations**
- **Enumeration constant is an obect of its enumeration type.Thus anenumeration candefine constructors, add methods and have instance variables.**
-
- **Enum Transport**
- **{ CAR(65), TRUCK(55), ARIPLANE(600), TRAIN(70);**
- **Private int speed;**
- **Transport(int s)**
- **{**
- **Speed=s;**
- **}**
- **Int getSpeed()**
- **{ return speed;}**
- **}**
- **}**

- Class EnumDemo3
- {
- Public static void main(String args[])
- {
- Transport tp;
- System. Out.println(“Speed of an airplane is “ +Transport.AIRPLANE.getspeed()+ “ miles per hour.\n”);
- System.out.println(“Display all transport speeds”);
- For(Transport t: Transport.values())
- System.out.println(t+” typical speed is “+t.getSpeed()+” miles per hour.”);
- }
- }
- **Restrictions**
- Enumeration can’t inherit another class. Second an enum cannot be a superclass.

- **Type Wrappers:**
- Java uses primitive types such as int, double to hold the basic data type supported by the language. Instead of primitive types we can use object to hold these type of values. But it add an unacceptable overhead for simple calculations.

- The Type wrappers are Double, Float, Long, Integer, Short, Byte, Character and Boolean.
- Character:
- Character is a wrapper around a char. The constructor for Character is
- Character(char ch)
- To obtain char value contained in a Character object call charValue().
- char charValue()
- Boolean:
- It is a wrapper around boolean values.

- It defines a constructor
- Boolean(boolean boolvalue)
- Boolean(String boolstring)
- The method used is boolean booleanValue()
- **Numeric Type Wrappers:**
- It is to represent numeric values. These are Bytes, Short, Integer, Long, Float and Double.
- Eg.
- Class wrap
- {
- Public static void main(String args[])
- {
- Integer iob=new Integer(100);
- Int i=iob.intValue();
- System.out.println(i+” ”+iob);//displays 100 100
- }
- }

- The process of encapsulating a value within an object is called boxing.
- Eg. Integer iob=new Integer(100);
- The process of extracting a value from a type wrapper is called unboxing. Eg.
- int i=iob.intValue();
- **Autoboxing:**

Beginning with JDK 5, Java added two important features. Autoboxing and auto-unboxing. Autoboxing is the process by which a primitive type is automatically encapsulated in its equivalent type wrapper without creating object. **auto-unboxing** is the process by which the value of a boxed object is automatically extracted from a type wrapper. No need to call the method intValue() or doubleValue().

Eg. Integer ob=100;//autobox an int

int i=ob;//auto-unbox

- **Annotations(MetaData)**
-
- An important feature added to Java by JDK 5 is the annotation. It enables us to embed supplemental information in to a source file.
- For eg. An annotation might be processed by a source-code generator, by the compiler , or by a deployment tool.
- **Creating and using Annotation**
- An annotation is created thorough a mechanism based on the interface. Here is a simple example.
- @interface MyAnno
- {
- String str();
- int val();
- }
- This declares ann annotation called MyAnno. Notice the @ that precedes the keyword interace. This tlls the compiler that an annotation type is being declared. Here methods act like fields.
-

- All annotation types automatically extend the Annotation interace. Annotation is a superinterace o all annotations. It is declared within the java.lang.annotation package.
-
- We can use this annotation.
- @Myanno(str=”Annotation Example”, val=100)
- Public static void myMeth()(//…
- Now annotation is linked with method myMeth()
-

- **Built-in Annotations**
- Java defined many built-in annotations. Most are specialized. Eight are for general purpose. Four are imported rom java.lan.annotation: @Retentaion, @Documented, @target and @Inherited. Four, @Override, @Deprecated, @SafeVarargs, and @SuppresssWarnings are included in java.lang.
-
- Here an example of @Deprecated to mark the MyClass class and getMs() method as deprecated.
- @Deprecated
- Class MyClass
- {
- Private String msg.
- MyClass(String m)
- {
- Msg=m;
- }
- @Deprecated
- String getMsg()
- {
- return msg;
- }
- }

- Class AnnoDemo{
- Public static void main(String[] args)
- {
- MyClass myobj=new MyClass(“test”);
- System.out.println(myObj.getMs());
- }
- }
-
- **@Retention** specifies the retention policy that will be associated with annotation. It determines how long an annotation is present during the cokmpilation and deployment process.
- **@Documented**: A marker annotation that tells a tool that an annotation is to be documented.
- **@Target** Speciies the types of declarations to which an annotation can be applied. @Target takes one argument, which my be a constructor, ield and method.
- **@Inherited** A marker annotation that causes the annotation for a superclass to be inherited by a subclass.
- **@Override** A method annotated with @Override must override a method from a superclass. If it doesn’t, a compile-time error will result.

- **@Deprecated** A marker annotatioin that indicates that fearure is obsolete and has been replaced by a newer form.
- **@SafeVarargs** A marker annotation that indicates that no unsafe actions related to a varargs parameter in a method or constructo occur.
- **@SuppressWarnings**
- Suppress that one or more warnings that might be issued by the compiler are to be suppressed or secret.
-
-

- **Generics:**
- The term generics means parameterized types. A class, interface o method that uses a type parameter is called generic, as in generic class or generic method.
- Using generics it is possible to create a single class, for eg. That automatically works with diffferent types of data.
- class Gen<T>
- {
- T ob;
- Gen(T ob1)
- {
- ob=ob1;
- }
- T getOb()
- {
- return ob;
- }
-

```
public String toString()
{
System.out.println("class name is "+ob.getClass().getName());
}
}
class GenDemo
{
public static void main(String args[])
{
Gen <Integer>obj=new Gen<Integer>(88);
System.out.println("Integer value is"+obj.getOb());
Gen<String>obj1=new Gen<String>("hello");
```

- System.out.println("String value is"+obj.getOb());
- }
- }
- First object java.lang.String
- Second object java.lang.Integer
- hello
- 999

- **Generic class with two type parameters:**
- We can declare more than one type parameter in a generic type.
- class Gen1<T,V>
- {
- T ob1;
- V ob2;
- Gen1(T ob3, V ob4)
- {
- ob1=ob3;
- ob2=ob4;
- }
- T getOb1()
- {
- return ob1;
- }
- V getOb2()
- {
- return ob2;
- }

```
public String toString()
{
System.out.println("The type of T is"+ob1.getClass().getName());
System.out.println("The type of V is"+ob2.getClass().getName());
return("finding class name");

}
}
class GenDemo1
{
public static void main(String args[])
{
Gen1<Integer, String> obj=new Gen1<Integer, String>(88, "Hello");
System.out.println(obj);
System.out.println("Integer value is"+obj.getOb1());
System.out.println("String  value is"+obj.getOb2());
}
}
```

- Output

- The type of T is java.lang.Integer
- The type of V is java.lang.String
- Finding class name
- Integer value is 88
- String value is Hello

- **Bounded Types**
- Normally the type parameters could be replaced by any class type. Ut sometimes it is useful to limit the types that can be passed to a type parameter.
-

- Eg.
- Class NumericFns<T>
- {
- T num;
- NumercFns (T n)
- {
- num=n;
- }
- double reciprocal()
- {
- return 1/num.doubleValue();//Error
- }
- Double raction()
- {
- return num.doubleValue()-num.intValue()//error
- }
- }
- To handle such situations, Java provides bounded types.
- <T extends superclass>
-
-

- Eg.
- Class NumericFns < T extends Number>
- {
- T num;
- NumericFns(T n)
- {
- Num=n;
- }
- double reciprocal()
- {
- return 1/num.doubleValue();
- }
- double raction()
- {
- return num.doubleValue()-num.intValue();
- }
- }

- Class BoundsDemo
- {
- Public static void main(String args[])
- {
- NumericFns<Inteer> iob=new NumericFns<Integer>(5);
- System.out.println(“Reciprocal of iob is”iob.reciprocal());
- System.out.println(“Fractional componenent of iob is” +iob.fraction());
- }
- }
- Output
- Reciprocal of iob is 0.2
- Fractional cokponent of iob is 0.0

```
1  //  Fig. 18.1: OverloadedMethods.java
2  // Using overloaded methods to print array of different types.
3
4  public class OverloadedMethods
5  {
6     // method printArray to print Integer array
7     public static void printArray( Integer[] inputArray )
8     {
9        // display array elements
10       for ( Integer element : inputArray )
11          System.out.printf( "%s ", element );
12
13       System.out.println();
14    } // end method printArray
15
16    // method printArray to print Double array
17    public static void printArray( Double[] inputArray )
18    {
19       // display array elements
20       for ( Double element : inputArray )
21          System.out.printf( "%s ", element );
22
23       System.out.println();
24    } // end method printArray
25
26    // method printArray to print Character array
```

```
// Fig. 18.3: GenericMethodTest.java
// Using generic methods to print array of different types.

public class GenericMethodTest
{
   // generic method printArray
   public static < E > void printArray( E[] inputArray )
   {
      // display array elements
      for ( E element : inputArray )
         System.out.printf( "%s ", element );

      System.out.println();
   } // end method printArray

   public static void main( String args[] )
   {
      // create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println( "Array integerArray contains:" );
```

```
27    public static void printArray( Character[] inputArray )
  28    {
  29       // display array elements
  30       for ( Character element : inputArray )
  31          System.out.printf( "%s ", element );
  32
  33       System.out.println();
  34    } // end method printArray
  35
  36    public static void main( String args[] )
  37    {
  38       // create arrays of Integer, Double and Character
  39       Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
  40       Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5,
6.6, 7.7 };
  41       Character[] characterArray = { 'H', 'E', 'L', 'L',
'O' };
  42
  43       System.out.println( "Array integerArray contains:" );
  44       printArray( integerArray ); // pass an Integer array
  45       System.out.println( "\nArray doubleArray contains:"
);
  46       printArray( doubleArray ); // pass a Double array
  47       System.out.println( "\nArray characterArray
contains:" );
  48       printArray( characterArray ); // pass a Character
array
  49    } // end main
  50 } // end class OverloadedMethods
```

- **Some Generic Restrictions:**
- **Some Generic Restrictions:**
-
- There are a few restriction that you need to keep inmind when using generics.
-
- Type parameters cannot be Instantiated.
- It is not possible to create an instance of a type parameter.
-
- Class Gen<T>
- {
- T ob;
- Gen()
- {
- Ob=new T();//Illegal
- }
- }
- **Restrictions on static members**
- No static member can use type a parameter declared by the enclosing class. For emaple, both of the static members of this class are illegal.
- Class Wrong<T>
- Static T ob;//wrong

- Static T getob()//wrong
- {
- return ob;
- }
- }
- **Generic Array Restriction**
- Class Gen <T Extends Number>
- {
- T ob;
- T[] vals;
-
- Gen(T o, T[] nums)
- {
- Ob=o;
- Vals=new T[10]; //can’t create an array of T
- }
- }
-
- **Generic Exception Restriction**
- A generic class cannot extend Throwable.
-
-
-
-
-
-
-

- **Applet Basics**
- Applets are small programs that are desined for transmission over the internet and run within a browser. There are two varieties o applets; Abstract Window Toolkit and thos based on Swing.
- It involves two import statements. Import java.awt.*; contains support or a window-based, graphical user interface. The next import statement imports the java.applet package. This package contains the class Applet.

- Applet Basics:
- All applets are subclasses of Applet. **Applets are not stand-alone programs.** Instead, they run within either a web browser or an applet viewer.
- Execution of an applet does not begin at main(). Actually, few applets even have main() methods. Ouput to our applet's window is not performed by System.out.println(). Output is handled with various AWT methods, such as drawString(). Which outputs a string to a specified X,Y location.

- To use an applet, it is specified in an HTML file. One way to do this is by using the APPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file. To view and test an applet more conveniently, simply include a comment at the head of our Java source code file that contains the APPLET tag.
- /*<applet code="MyApplet" width=200 height=300></applet>*/
- This comment contains an APPLET tag that will run an applet called Myapplet in a window that is 200 pixels wide and 300 pixels high.

- 24 printArray(integerArray); // pass an Integer array
  25 System.out.println("\nArray doubleArray contains:");
  26 printArray(doubleArray); // pass a Double array
  27 System.out.println("\nArray characterArray contains:");
  28 printArray(characterArray); // pass a Character array
  29 } // end main
  30 } // end class GenericMethodTest
-
  Array integerArray contains:
  1 2 3 4 5 6
- Array doubleArray contains:
  1.1 2.2 3.3 4.4 5.5 6.6 7.7

  Array characterArray contains:
  H E L L O

- **Generic Constructors:**
- A constructor can be gneric, even I its class is not.

```
class Summation
{
private int sum;
<T extends Number> Summation (T arg)
{
sum=0;

for(int i=0;i<=arg.intValue();i++)
sum+=i;
}
int getSum()
{
return sum;
}
}
class GenConsDemo
{
public static void main(String[] args)
{
Summation ob=new Summation(4.0);
System.out.println("Summation of 4.0 is" +ob.getSum());
}
}
```

Output

Summation of 4.0 is 10

- **<u>An Applet skelton:</u>**
- The four important methods are **init(), start(), stop() and destroy().**
- AWT based applets will also override the **paint()** method which is defined by the **AWT component class.** This method is called when an applet's output must be redisplayed.
- Eg.
- import java.awt.*;
- import java.applet.*;
- /*<applet code="app" width=100

- height=100></applet>*/

- public class app extends Applet
- {
- public void init()
- {
- }
- public void start()
- {
- }
- public void stop()
- {
- }

- public void destroy()
- {
- }
- public void paint()
- {
- }
- }

Applet Initialization and Termination.

When an applet begins the following methods are called.

Init(), start(), paint()

When an applet is terminated the following method will call.

- Stop()
- Destroy()
- init()
- This method is used to initialise the applet. Only once a time will call.
- start()
- This is called after init(). It is also called to restart an applet after it has been stopped. When init() is called **once** the first time an applet is loaded. Start() is called each time an applet's HTML document is displayed on screen.
- paint()
- The paint() method is called each time our applet's output must be redrawn.

- stop()
- Stop() method is called when a web browser leaves the HTML document containing the applet.
- destroy()
- Destroy() method is called when the environment determines that our applet needs **to be removed** completely from memory.

- **Applet Architecture:**
- An applet is **window based program**. It is not like console-based program.
- Applets are event-driven. The process is like this. **An applet waits until even occurs**. The run-time system notifies the applet about an event **by calling an event handler** that has been provides by the applet.
- Another way user initiates interaction with an applet.
- For eg. When the user clicks the mouse inside the applet's window has input focus, a keypressed event is generated.

- Requesting Repainting:
- Whenever our applet want to update the information displayed in its window, it simply calls repaint(). If we want to display the message or information , after storing the ouput call repaint() method. Then call paint() method automatically and display the stored information.
- void repaint()

- Using the status window:
- In addition to displaying the information in the window, tha applet can also output massage **to status window** of the browser or applet viewer on which it is running. For that call showStatus() with the string that we want to display. The status window is good place to give the user feedback about what is occurring int th eapplet, suggest options or to report some errors.
- Eg.
- import java.awt.*;
- import java.applet.*;
- /*<applet code="Statuswindow" width=200

- height=200></applet>*/
- public class Statuswindow extends Applet
- {
- int x,y;
- String msg;
- public void init()
- {
- setBackground(Color.red);
- setForeground(Color.blue);
- }

- public void paint(Graphics g)
- {g.drawString("I am in applet window",10,10);
- showStatus("hi hi hi I am in status window");
- }
- }
- Output:

- Passing parameters to applets

- **Swing Fundamentals**
- Swing is a set of classes that provides more powerful and flexible GUI components than does the AWT.
- The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. Because the AWT components use native code resources, they are referred to as *heavyweight*.
- The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently **on different platforms**.

- This potential variability threatened the overarching philosophy of Java: **write once, run anywhere. Second, the look and feel** of each Not long after Java's original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed. The solution was Swing. **Introduced in 1997**, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing was fully integrated into Java.
-

- **Components and Containers**
- A Swing GUI consists of two key items: components and containers.
- **Components**
- In general, Swing components are derived from the **JComponent class**. JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component.All of Swing's components are represented by classes defined within the package javax.swing. The following table shows the class names for Swing components
-

- JApplet JButton JCheckBox JCheckBoxMenuItem
- JColorChooser JComboBox JComponent JDesktopPane
- JDialog JEditorPane JFileChooser JFormattedTextField
- JFrame JInternalFrame JLabel JLayeredPane
- JList JMenu JMenuBar JMenuItem
- JOptionPane JPanel JPasswordField JPopupMenu
- JProgressBar JRadioButton JRadioButtonMenuItem JRootPane
- JScrollBar JScrollPane JSeparator JSlider
- JSpinner JSplitPane JTabbedPane JTable
- JTextArea JTextField JTextPane JTogglebutton
- JToolBar JToolT
-
-
-

- Containers
- Swing defines two types of containers. The first are top-level containers**: JFrame, JApplet, JWindow, and JDialog. These containers do not inherit JComponent. They do, however, inherit the AWT classes Component and Container.** Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library. As the name implies, a top-level container must be at the top of a containment hierarchy.
-

- **Understanding Layout Managers**
- All of the components that we have shown so far have been positioned by the default layout manager.Each Container object has a layout manager associated with it. **A layout manager is an instance of any class that implements the LayoutManager interface**. The layout manager is set by the setLayout( ) method. If no call to setLayout( ) is made, then the default layout manager is used. Whenever a container is resized , the layout manager is used to position each of the components within it. The setLayout( ) method has the following general form:
- void setLayout(LayoutManager layoutObj)
- Here, layoutObj is a reference to the desired layout manager. I
-
-
- // **A simple Swing application**.
- import javax.swing.*;
- class SwingDemo {
- SwingDemo() {
- // Create a new JFrame container.
- JFrame jfrm = new JFrame("A Simple Swing Application");
- // Give the frame an initial size.
- jfrm.setSize(275, 100);
- // Terminate the program when the user closes the application.

- jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
- // Create a text-based label.JLabel jlab = new JLabel(" Swing means powerful GUIs.");
- // Add the label to the content pane.
-]) {
- // Create the frame on the event dispatching thread.
- SwingUtilities.invokeLater(new Runnable() {
- public void run() {
- new SwingDemo();
- }
- });
- }

- **JLabel and ImageIcon**
- JLabel is Swing's easiest-to-use component
- . JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. JLabel defines several constructors. Here are three of them:
- JLabel(Icon icon)
- JLabel(String str)
- JLabel(String str, Icon icon, int align)
-

- The easiest way to obtain an icon is to use the ImageIcon class. ImageIcon implements Icon and encapsulates an image. Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's constructor.
- ImageIcon constructor used by the example in this section:
- ImageIcon(String filename)

```
// Demonstrate JLabel and ImageIcon.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet {
public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
```

```
public void run() {
makeGUI();
}}


} catch (Exception exc) {
System.out.println("Can't create because of "
+ exc);
}
}
```

```
private void makeGUI() {
// Create an icon.
ImageIcon ii = new ImageIcon("france.gif");
// Create a label.
JLabel jl = new JLabel("France", ii,
JLabel.CENTER);
// Add the label to the content pane.
add(jl);
}
}
```

- **The Swing Buttons**
- Swing defines four types of buttons: JButton, JToggleButton, JCheckBox, and JRadioButton.
- All are subclasses of the AbstractButton class, which extends JComponent.
- **JButton**
- The JButton class provides the functionality of a push button
-
- Three of its constructors are shown here:
- JButton(Icon icon)
- JButton(String str)
- JButton(String str, Icon icon)

```
// Demonstrate an icon-based JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=450>
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener {
JLabel jlab;
public void init() {
try {
```

```
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();
}
}
);
} catch (Exception exc) {
System.out.println("Can't create because of " +
exc);
}
}
```

```
private void makeGUI() {
// Change to flow layout.
setLayout(new FlowLayout());
// Add buttons to content pane.
ImageIcon france = new ImageIcon("france.gif");
JButton jb = new JButton(france);
jb.setActionCommand("France");
jb.addActionListener(this);
add(jb);
// Create and add the label to content pane.
jlab = new JLabel("Choose a Flag");
add(jlab);
}
// Handle button events.
public void actionPerformed(ActionEvent ae) {
jlab.setText("You selected " + ae.getActionCommand());
}
}
```

- **JToggleButton**
- A useful variation on the push button is called a toggle button. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. JToggleButton implements AbstractButton. Constructor is
- JToggleButton(String str)

- Same as above
- private void makeGUI() {
- // Change to flow layout.
- setLayout(new FlowLayout());
- // Create a label.
- jlab = new JLabel("Button is off.");
- // Make a toggle button.
- jtbn = new JToggleButton("On/Off");
- // Add an item listener for the toggle button.
- jtbn.addItemListener(new ItemListener() {
- public void itemStateChanged(ItemEvent ie) {
- if(jtbn.isSelected())
- jlab.setText("Button is on.");
- else
- jlab.setText("Button is off.");
- }
- });
- // Add the toggle button and label to the content pane.
- add(jtbn);
- add(jlab);
- }

-

- **Check Boxes**
- The JCheckBox class provides the functionality of a check box. Its immediate superclass is JToggleButton
- eg.
- private void makeGUI() {
- // Change to flow layout.
- setLayout(new FlowLayout());
- // Add check boxes to the content pane.
- JCheckBox cb = new JCheckBox("JAVA");
- cb.addItemListener(this);
- add(cb);
- jlab = new JLabel("Select languages");
- add(jlab);
- }
- // Handle item events for the check boxes.

```
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
if(cb.isSelected())
jlab.setText(cb.getText() + " is selected");
else
jlab.setText(cb.getText() + " is cleared");
}
}
```

- **Radio Buttons**
- Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time
- private void makeGUI() {
- // Change to flow layout.
- setLayout(new FlowLayout());
- // Create radio buttons and add them to content pane.
- JRadioButton b1 = new JRadioButton("A");
- b1.addActionListener(this);
- add(b1);
- JRadioButton b2 = new JRadioButton("B");
- b2.addActionListener(this);
- add(b2);
- ButtonGroup bg = new ButtonGroup();
- bg.add(b1);
- bg.add(b2);
- b2.addActionListener(this);
- add(b2);

- jlab = new JLabel("Select One");
- add(jlab);
- }
- // Handle button selection.
- public void actionPerformed(ActionEvent ae) {
- jlab.setText("You selected " + ae.getActionCommand());
- }

- **Trees**
- A*tree* is a component that presents a hierarchical view of data. The user has the ability to
- expand or collapse individual subtrees in this display. Trees are implemented in Swing by
- the **JTree** class. A sampling of its constructors is shown here:
- JTree(Object *obj*[ ])
- JTree(Vector<?> *v*)
- JTree(TreeNode *tn*)
- Here are the steps to follow to use a tree:
- 1. Create an instance of **JTree**.
- 2. Create a **JScrollPane** and specify the tree as the object to be scrolled.
- 3. Add the tree to the scroll pane.
- 4. Add the scroll pane to the content pane

```
Eg.
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeDemo" width=400 height=200>
</applet>
*/
public class JTreeDemo extends JApplet {
JTree tree;
JLabel jlab;
public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();
}
}
);
```

```
} catch (Exception exc) {
System.out.println("Can't create because of " + exc);
}
}
private void makeGUI() {

DefaultMutableTreeNode top =new DefaultMutableTreeNode("Course");
DefaultMutableTreeNode a =new DefaultMutableTreeNode("UG");
DefaultMutableTreeNode a1 =new DefaultMutableTreeNode("BA");
top.add(a);
a.add(a1);
a.add(a2);
JTree jt=new JTree(top);
JScrollPane jsp=new JScrollPane(jt);
jsp.setPreferredSize(new Dimension(150,200));
jt.addTreeSelectionListener(this);
add(jsp);
```

- tlab=new JLabel("Select One");
- add(tlab);
- }
- public void valueChanged(TreeSelectionEvent tse)
- {
- tlab.setText("Current Selection "+tse.getPath());
- }
- }
-

- **NETWORKING**
- **Networking Basics**
- **Socket** plays an important role in Networking. Sockets are the foundation of modern networking because a socket allows a single computer to serve many different client at once and to serve many different types of information. This is accomplished through the use of port, which is a numbered socket on a particular machine. A server is allowed to accept multiple clients connected to the **same port number**, although each session is unique. To manage multiple client

- Connections, server process must be **multithreaded**.
- Socket communication takes places through protocol. Different protocols are **Internet Protocol, HTTP(HyperText Transfer Protocol), TCP(Transmission Control Protocol), UDP(User Datagram Protocol).**
- IP: it is **a low level protocol** that breaks data into small packets and sends them to an address across a network.
- TCP: is a higher-level protocol that manages to robustly **string or sequence together these packets for sorting and retransmitting**. It is point to point like telephone call.
- UDP: It can be used directly to support fast, **connectionless, unreliable transport of packets**. It sends independend packets and it is not connection based. Like posting letter. There will not be no order for reaching the letter in the other end.

- Once connection is established , it will check which **port** we are using. It involves different port number. Port number 21 is for FTP. 23 is for Telnet, 25 for e-mail, 80 for HTTP, 119 for net news.
- For eg. HTTP is a protocol that web browsers and servers use to transfer hypertext pages and images.
- It is working like this. When a client requests a file from an HTTP server, an action known as **hit**, it simply sends the name of the file in special format to predefined port and reads back the contents of the file. The server also responds with a status code to tell the client whether or not request can be fulfilled and why.
- In the case of internet key component is address.

- Every computer has uniquely identifies address. It consist of 32-bit values, organized as four 8-bit values.
- **Networking Classes and Interfaces:**
- Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, point-to-point datagram-oriented model. The class contained in Java.net package are-
- Authenticator, CacheRequest, CacheResponse, ContentHandler, CookieHandler, DatagramPacket, Inet6address, InetAddress, ServerSocket, Socket, SocketAdddress, URI, URL, SocketPremission, DatagramSocket, HttpCookie, HttpURLConnection, Inet4Address, URLClassLoader, URLConnection, ResponseCache

- Interface are listed here-
- ContentHandlerFactory, CookiePolicy, DatagramSocketImplFactory, SocketImplFactory, SocketOptions
- **InetAddress:-**
- InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address.
- Factory Methods:
- InetAddress class **has not visible constructors**. To create an InetAddress object, we have to use one of the available factory methods. Three commonly used InetAddress factory methods are-
- Static InetAddress getLocalHost() throws UnknownHostException

- Static InetAddress getByName(String hostName) throws UnknownHostException
- Static InetAddress[] getAllByName(String hostName) throws UnknownHostException
- The **getLocalHost()** method simply returns the InetAddress object that **represents the local host**. The getByName() method returns an InetAddress for a **host name passed to it**. If unable to resolve the host name, **they throw an UnknownHostException**. On the internet, it is common for a single name to be used to represent several machines.
- The **getAllByName()** factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to.

```
import java.net.*;
class Inet
{
public static void main(String args[]) throws UnknownHostException
{
InetAddress address1=InetAddress.getLocalHost();
System.out.println(address1);
InetAddress address2=InetAddress.getByName("yahoo.com");
System.out.println(address2);
InetAddress sw[]=InetAddress.getAllByName("gmail.com");
for(int i=0;i<sw.length;i++)
{
System.out.println(sw[i]);
}
}
}
```

- InetAddresss class has several other methods. The are
- **Boolean equals(Object other)** Returns true if this object has the same internet adddress as other.
- Byte[] getAddress() returns a byte array that represents the object's IP address in network byte order.
- String getHostAddress() Returns a string that represents the host address associated with the InetAddress object.
- String gethostName() Returns a string that represents the name associated with the InetAddress object.
- Boolean isMulticastAddress() Retursn true if this address is a multicast address. Otherwise, it returns false.
- String toString() Returns a string that lists the host name and the IP address for convenience.
-

- **<u>TCP/IP Client sockets:</u>**
- TCP/Ipsockets are used to implement reliable, bidirectional, persistend, **point-to-point**, stream-based connections between hosts on the internet. There are two kinds of TCP sockets in java. One is for servers , and the other for clients. The ServerSocket class is

- Waits for clients to connect before doing anything. The **Socket class is for clients**. It is is designed to connect to server sockets and initiate protocol exchanges.
- The two constructors to create clients sockets are
- Socket(String hostName, int port) throws UnknownException, IOException : Creates a socket connected to the named host and port.
- Socket(InetAddress ipAddress, int port) throws IOException Creates a Socket using a preexisting InetAddress object and a port.
- **Socket defines several methods**.
- InetAddress getInetAddress(): Returns the InetAddress associated with Socket object. It returns null if the socket is not connected.

- Int getPort() Returns the remote port to which the invoking Socket object is connected. It returns 0 if the socket is not connected.
- Int getLocalPort() Returns the local port to which the invoking Socket object is bound. It returns -1 if the socket is not bound.
- InputStream getInputStream() throws IOException
- Returns the InputStream associated with the invoking object.
- OutputStream getOutputStream() throws IOException Returns the OutputStream associated with the invoking socket.
-

- Eg.
- import java.io.*;
- import java.net.*;
- class sock
- {
- public static void main(String args[]) throws

- IOException
- {
- int c;
- Socket soc=new Socket("Intrinsic.net",43);
- InputStream inp=soc.getInputStream();

- OutputStream out=soc.getOutputStream();
- String str=(args.length==0?"Obsorne.com":args[0]);
- byte buf[]=str.getBytes();
- out.write(buf);

- while((c=inp.read())!=-1)
- {
- System.out.println((char)c);
- }
- soc.close();
- }
- }

- Output:
- Here we will get the information about Intrinsic net.

- **URL:-**
- The URL provides a reasonably intelligible form to uniquely identify or address information on the internet.
- All URL share the same basic format. The examples are
- http://www.osborne.com/ and http//www.osborne.com:80/index.htm.
- A URL specification is based on four components. The first is protocol to use, separated from the rest of the locator by a colon(:). Comon protocols are HTTP,FTP, gopher, and file. The second component is the host name or IP addressof the host to use that is followed by //. The third component is port number is an optional parameter. 80 is HTTP port. Most HTTP

- Most HTTP servers will append a file named index.html or index.htm to URLs that refer directly to directory resource.
- Java's URL class has several constructors, each can throw a MalformedURLException.
- URL(String urlSpecifier) throws MalformedURLException
- URL(String protocolName, String hostName, int port, String path) throws MalformedURLException
- URL(String protocolName, string hostname, String path) throws MalformedURLException

- EG.
- import java.io.*;
- import java.net.*;
- class Ur
- {
- public static void main(String args[]) throws

- IOException
- {
- URL input=new URL("http://yahoo.com/hello");
- System.out.println(input.getPort());
- System.out.println(input.getHost());
- System.out.println(input.getProtocol());

- System.out.println(input.getFile());
- }
- }
- **URLConnection**
- It is a general-purpose class for accessing the attributes of remote resource.
- URLConnection has several methods.
- **Int getContentLength()** Returns the size in bytes of the content associated with resource.
- **String getContentType()** Returns the type of content found in the resource. This is the value of the content-type header field. Retuns null if the content type is not available.
- **Long getDate()** Retuns the time and date of response

- Represented in terms of milliseconds since January 1, 1970 GMT.
- **Long getLastModified()** Returns the time And date, represented in terms of milliseconds since January 1, 1970 GMT. Returns zero if the last-modified date is unavailable.
- **InputStream getInputStream()** throws IOException
- Returns an InputStream that is linked to the resource. This stream can be used to obtain the content of the resource.
- **Long getExpiration()** Retuns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970. Returns 0 if not available.

- Eg.
- Import java.net.*;
- Import java.io.*;
- Import java.util.Date;
- Class UrlDemo
- {
- Public static void main(String args[])
- {
- Int c;
- URL hp=new URL(“http://facebook.com”);
- URLConnection hpcon=hp.openConnection();
- Long d=hpcon.getDate();

- S.o.p(new Date(d));
- d=hpcon.getexpiration();
- S.o.p(d);
- d=hpcon.getLastModified();
- S.o.p(d);
- }}
- **HttpURLConnection**
- It is a subclass of URLConnection that provides support for HTTP connections. Methods are
- String getRequestMethod() Retuns a string representing how URL requests are made. The default is GET.

- String getResponseMessage() throws IOException
- Returns the response message associate with response code.
- Void setRequestMethod(String how) throws ProtocolException: Sets the method by which HTTP request are make to that specified by how. Means setting as GET or POST EG.
- Import java.net.*;
- Import java.io.*;
- Import java.util.*;
- Class HttpURLDemo
- {
- Public static void main(String args[])
- {

```
URL hp=new URL("http://www.google.com");
HttpURLConnection
hpcon=(HttpURLConnection)
hp.openConnection();
s.o.p(hpcon.getRequestMethod());
S.o.p("Response Message
is"+hpcon.getResponseMessage());
}}
```