Introducing Classes

- Class defines the shape and nature of an object.
- Class forms the basis for object-oriented programming in Java.
- Any concept can be implemented in a Java program must be encapsulated within a class.

Class Fundamentals

- A class defines a new data type. Once defined, this new type can be used to create objects of that type.
- Thus, a class is a *template* for an object, and an object is an *instance* of a class.

The General Form of a Class

- Class specifies the data that it contains and the code that operates on that data.
- While very simple classes may contain only code or only data, most real-world classes contain both.
- A class is declared by use of the **class** keyword.
- A simplified general form of a **class** definition is shown here:

```
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
        }
}
```

- The data, or variables, defined within a **class** are called *instance variables.*
- The code is contained within *methods.*
- Collectively, the methods and variables defined within a class are called *members* of the class.
- Thus the methods that determine how a class' data can be used.
- Each object of the class contains its own copy of these variables.

- Thus, the data for one object is separate and unique from the data for another.

A Simple Class

- Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**.

```
class Box
 {
        double  width;
        double height;
        double depth;
}

class BoxDemo2
{
        public static void main(String args[])
         {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;

                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;

                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;

                // compute volume of first box
                vol = mybox1.width * mybox1.height * mybox1.depth;
                System.out.println("Volume is " + vol);
              // compute volume of second box
              vol = mybox2.width * mybox2.height * mybox2.depth;
              System.out.println("Volume is " + vol);
              }

              }

              output:

              Volume is 3000.0
              Volume is 162.0
```

mybox1's data is completely separate from the data contained in **mybox2**.

Declaring Objects

- When a class is created , we are creating a new data type.
- We can use this type to declare objects of that type.
- However, obtaining objects of a class is a two-step process.
- First, we must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
- Second, we must acquire an actual, physical copy of the object and assign it to that variable by using the **new** operator.
- The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it
- 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
```

Assigning Object Reference Variables

```
Box b1 = new Box();
Box b2 = b1;
```

- **b1** and **b2** will both refer to the *same* object.
- The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**.
- Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.
- Although **b1** and **b2** both refer to the same object, they are not linked in any other way.
- For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**.
- For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

  Here, **b1** has been set to **null**, but **b2** still points to the original object.

Introducing Methods

- Classes usually consist of two things: instance variables and methods.
- This is the general form of a method:

```
type name(parameter-list) {
// body of method
}
```

- Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that we create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name.*
- The *parameter-list* is a sequence of type and identifier pairs separated by commas.

```
class Box
{
        double  width;
        double height;
        double depth;
        //
        void volume()
         {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}

class BoxDemo3
{
        public static void main(String args[])
         {
                Box mybox1 = new Box();
                Box mybox2 = new Box();

                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;

                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;

                // display volume of first box
                mybox1.volume();

                // display volume of second box
                mybox2.volume();
                }
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Returning a Value

```
class Box
 {
        double  width;
        double height;
        double depth;
        // compute and return volume
        double volume()
        {
        return width * height * depth;
        }
}
class BoxDemo4
{
        public static void main(String args[])
         {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;

                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;

                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;

                // 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);
                }
}
```

Adding a Method That Takes Parameters

- Parameters allow a method to be generalized.
- That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.

- Here is a method that returns the square of the number 10:

```
int square()
{
return 10 * 10;
}
```

- While this method does, indeed, return the value of 10 squared, its use is very limited.
- However, if we modify the method so that it takes a parameter, as shown next, then we can make **square( )** much more useful.

```
int square(int i)
{
return i * i;
}
```

- Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

```
// This program uses a parameterized method.

class Box
 {
        double  width;
        double height;
        double depth;

        double volume()
         {
                return width * height * depth;
        }

        void setDim(double w, double h, double d)
        {
        width = w;
        height = h;
        depth = d;
        }
}

class BoxDemo5
{
        public static void main(String args[])
        {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;
```

```
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

        // 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);
    }
}
```

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Thus automatic initialization is performed through the use of a constructor.
- 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.
- the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.

```
class Box
 {
        double  width;
        double height;
        double depth;
        Box()
        {

        }
      System.out.println("Constructing Box"); width = 10;
      height = 10;
      depth = 10;
        double volume()
        {
                return width * height * depth;
        }
}
```

```
class BoxDemo6
{
        public static void main(String args[])
         {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                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);
        }
}
```

Output:
Constructing Box Constructing
Box Volume is 1000.0
Volume is 1000.0

- Both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created.
- Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume.

Parameterized Constructors

- While the **Box( )** constructor in the preceding example initializes with value 10.all boxes have the same dimensions.
- **Box** objects of various dimensions can be assigned by using parameterized constructor.

```
class Box
{
        double  width;
        double height;
        double depth;

        Box(double w, double h, double d)
         {
                width = w;
                height = h;
```

```
                depth = d;
        }

        double volume()
        {
                return width * height * depth;
        }
}
class BoxDemo7
 {
        public static void main(String args[])
        {
                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);
        }
}
```

Output:
Volume is 3000.0
Volume is 162.0

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.

```
// A redundant use of this.
                Box(double w, double h, double d)
                 {
                        this.width = w;
                        this.height = h;
                        this.depth = d;
                }
```

Instance Variable Hiding

- it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.

- However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.

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

Garbage Collection

- Since objects are dynamically allocated by using the **new** operator, how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.
- Java handles deallocation automatically.
- The technique that accomplishes this is called *garbage collection.*
- when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- Garbage collection only occurs sporadically (if at all) during the execution of program.

The finalize( ) Method

- An object will need to perform some action when it is destroyed.
- if an object is holding some non-Java resource such as a file handle or character font, then we might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called *finalization.*
- By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, simply define the **finalize( )** method.
- The Java run time calls that method whenever it is about to recycle an object of that class.
- Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.
- The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.
- **finalize( )** is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope

A Stack Class

- Stacks are controlled through two operations traditionally called *push* and *pop*.
- To put an item on top of the stack, we will use push.
- To take an item off the stack, we will use pop.
- Here is a class called **Stack** that implements a stack for integers:

```
// This class defines an integer stack that can hold 10 values.
class Stack
{
    int stck[] = new int[10];
    int tos;
    // Initialize top-of-stack
    Stack()
    {
            tos = -1;
    }
    // Push an item onto the stack
    void push(int item)
    {
        if(tos==9)
        System.out.println("Stack is full.");
        else
        stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop()
    {
        if(tos < 0)
        {
        System.out.println("Stack underflow.");
        return 0;
        }
        else
        return stck[tos--];
    }
}
class TestStack
{
    public static void main(String args[])
    {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();
```

```
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
        System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
        System.out.println(mystack2.pop());
        }
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

Inheritance

- One class can acquire the properties of another class.
- A class that is inherited is called a *superclass.*
- The class that does the inheriting is called a *subclass.* Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Inheritance Basics

- To inherit a class, 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**.the keyword **extends** is used to create a subclass of **A**.

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

output:

```
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 subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B**.
- Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself.
- a subclass can be a superclass for another subclass.
- The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name
{
        // body of class
}
```

- Java does not support the inheritance of multiple superclasses into a single subclass.
- But a subclass can become a superclass of another subclass.
- However, no class can be a superclass of itself.

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**.

```
// 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.

A More Practical Example

- the **Box** class developed will be extended to include a fourth component called **weight**.
- Thus, the new class will contain a box's width, height, depth, and weight.

```
// This program uses inheritance to extend Box.

class Box
{
        double  width;
        double height;
        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;
        }
}

// Here, Box is extended to include weight.
class BoxWeight extends Box
 {
        double weight; // weight of box

        // constructor for BoxWeight
        BoxWeight(double w, double h, double d, double m) {
        width = w;
        height = h;
        depth = d;
        weight = m;
        }
}

class DemoBoxWeight
{
        public static void main(String args[])
        {
                BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
                BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
                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);
        }
}
```

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
```

- the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box
{
        int color; // color of box

        ColorBox(double w, double h, double d, int c)
        {
                width = w;
                height = h;
                depth = d;
                color = c;
        }
}
```

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
class RefDemo
{
        public static void main(String args[])
         {
                BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
                Box plainbox = new Box();
                double vol;

                vol = weightbox.volume();
                System.out.println("Volume of weightbox is " + vol);
                System.out.println("Weight of weightbox is " + weightbox.weight);
                System.out.println();

                // assign BoxWeight reference to Box reference
                plainbox = weightbox;
                vol = plainbox.volume();                // OK, volume() defined in Box
                System.out.println("Volume of plainbox is " + vol);

                /* The following statement is invalid because plainbox does not define a weight
                member. */
                // System.out.println("Weight of plainbox is " + plainbox.weight);
                }
}
```

- Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects.

- Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

Using super

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
- **super** has two general forms.
  - The first calls the superclass' constructor.
  - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

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 superclass.
- **super( )** must always be the first statement executed inside a subclass' constructor.

```
// BoxWeight now uses super to initialize its Box attributes.

class BoxWeight extends Box
{
double weight;

        BoxWeight(double w, double h, double d, double m)
         {
                super(w, h, d); // call superclass constructor
                weight = m;
        }
}
```

- Here, **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box( )** constructor to be called, which initializes **width**, **height**, and **depth** using these values.

```
class Box
 {
        private double width;
        private double height;
        private double depth;

        // construct clone of an object

        Box(Box ob)
        {
```

```
        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;

BoxWeight(BoxWeight ob)
 {
        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();
        }
}
```

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

- **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
// 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

Creating a Multilevel Hierarchy

- 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**.
- 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.

```
class Box
{
        private double width;
        private double height;
        private double depth;

        // construct clone of an object
        Box(Box ob)
        {
                width = ob.width;
                height = ob.height;
                depth = ob.depth;
        }
```

```
        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
        }


        Box(double len)
         {
                width = height = depth = len;
        }

        double volume()
        {
                return width * height * depth;
        }
}

// Add weight.

class BoxWeight extends Box
{
        double weight;

        BoxWeight(BoxWeight ob)
         {
                super(ob);
                weight = ob.weight;
        }

        BoxWeight(double w, double h, double d, double m)
         {
                super(w, h, d);
                weight = m;
        }

        BoxWeight()
```

```
        {
                super();
                weight = -1;
        }
        BoxWeight(double len, double m)
        {
                super(len);
                weight = m;
        }
}

// Add shipping costs.
class Shipment extends BoxWeight
{
        double cost;

        Shipment(Shipment ob)
        {
                super(ob);
                cost = ob.cost;
        }

        Shipment(double w, double h, double d,double m, double c)
        {
                super(w, h, d, m); // call superclass constructor
                cost = c;
        }

        Shipment()
        {
                super();
                cost = -1;
        }

        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);
    }
}
```

output:

```
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

- given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used.

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

Output:

Inside A's constructor Inside B's constructor Inside C's constructor

Method Overriding

- when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.

```
class
A
{       int i, j;
        A(int a, int b)
         {
```

```
                i = a;
                j = b;
        }

// display i and j
        void show()
         {
                System.out.println("i and j: " + i + " " + j);
        }
}

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);
        subOb.show(); // this calls show() in B
        }
}
 output:
                k: 3
```

- the version of **show( )** inside **B** overrides the version declared in **A**.
- to access the superclass version of an overridden method can be called using **super**.

```
class B extends A
{
        int k;

        B(int a, int b, int c)
         {
                super(a, b);
```

```
                k = c;
        }

        void show()
         {
                super.show(); // this calls A's show()
                System.out.println("k: " + k);
        }
}
```

output:

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**.

- Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

```
class A
{

int i, j;

A(int a, int b)
 {
        i = a;
        j = b;
}

          // display i and j void show()
           {
                System.out.println("i and j: " + i + " " + j);
          }
}

// Create a subclass by extending class A.

class B extends A
{
        int k;

        B(int a, int b, int c)
         {
                super(a, b);
                k = c;
        }

        // overload show()

        void show(String msg)
         {
                System.out.println(msg + k);
        }
}
```

```
class Override
{
        public static void main(String args[])
         {
                B subOb = new B(1, 2, 3);
                subOb.show("This is k: "); // this calls show() in B
                subOb.show(); // this calls show() in A
                }
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

Packages and Interfaces

- *Packages* are containers for classes that are used to keep the class name space compartmentalized.
- Through the use of the **interface** keyword, Java allows to fully abstract the interface from its implementation.
- Using **interface**, we can specify a set of methods that can be implemented by one or more classes.
- The **interface**, itself, does not actually define any implementation.
- A class can implement more than one interface.

Packages

- Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.
- The package is both a naming and a visibility control mechanism.
- It is possible to define classes inside a package that are not accessible by code outside that package.
- We can define class members that are only exposed to other members of the same package.

Defining a Package

- To create a package ,simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored.
- If we skip the **package** statement, the class names are put into the default package, which has no name.
- The general form of the **package** statement:

package *pkg*;

- Here, *pkg* is the name of the package.
- For example, the following statement creates a package called **MyPackage**.

  package MyPackage;
- The general form of a multileveled package statement is shown here:

  package *pkg1*[.*pkg2*[.*pkg3*]];

Finding Packages and CLASSPATH

- How does the Java run-time system know where to look for packages that we create?
- The answer has three parts.
- First, by default, the Java run-time system uses the current working directory as its starting point.
- Second, we can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
- Third, we can use the **-classpath** option with **java** and **javac** to specify the path to our classes.

A Short Package Example

```
package MyPack;

class Balance
 {
        String name;
        double bal;

        Balance(String n, double b)
         {
                name = n;
                bal = b;
        }
        void show()
         {
                if(bal<0)
                System.out.print("--> ");
                System.out.println(name + ": $" + bal);
        }
}
class AccountBalance
{
        public static void main(String args[])
         {
                Balance current[] = new Balance[3];

                current[0] = new Balance("K. J. Fielding", 123.23);
```

```
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);

        for(int i=0; i<3; i++)
         current[i].show();
    }
}
```

- Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Access Protection

- Packages add another dimension to access control.
- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.
- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.
- Anything declared **public** can be accessed from anywhere.
- Anything declared **private** cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the **default access.**
- If we want to allow an element to be seen outside our current package, but only to classes that subclass our class directly, then declare that element **protected**.

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

Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

An Access Example

- This has two packages and five classes.
- Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.

This is file **Protection.java**:

```
package p1;

public class Protection
{
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;

        public Protection()
         {
                System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

This is file **Derived.java**:

```
package p1;

class Derived extends Protection
{
        Derived()
        {
        System.out.println("derived constructor");
        System.out.println("n = " + n);
```

```
        // System.out.println("n_pri = "4 + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}
```

This is file **SamePackage.java**:

```
package p1;

class SamePackage
 {
        SamePackage()
        {
                Protection p = new Protection();
                System.out.println("same package constructor");
                System.out.println("n = " + p.n);

                // System.out.println("n_pri = " + p.n_pri);

                System.out.println("n_pro = " + p.n_pro);
                System.out.println("n_pub = " + p.n_pub);
        }
}
```

- Following is the source code for the other package, **p2**.
- The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection.
- the default only allows access from within the class or the package, not extra-package subclasses.
- the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```
package p2;
class Protection2 extends p1.Protection
{
        Protection2()
        {
        System.out.println("derived other package constructor");

        // System.out.println("n = " + n);

        // System.out.println("n_pri = " + n_pri);
```

```
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}
```

This is file **OtherPackage.java**:

```
package p2;

class OtherPackage
 {
        OtherPackage()
        {

                p1.Protection p = new p1.Protection();
                System.out.println("other package constructor");


                // System.out.println("n = " + p.n);

                // System.out.println("n_pri = " + p.n_pri);

                // System.out.println("n_pro = " + p.n_pro);

                System.out.println("n_pub = " + p.n_pub);
        }
}
.
package p1;

// Instantiate the various classes in p1.

public class Demo
 {
        public static void main(String args[])
         {
                Protection ob1 = new Protection();
                Derived ob2 = new Derived();
                SamePackage ob3 = new SamePackage();
        }
}
```

```
// Demo package p2.

package p2;

public class Demo
 {
        public static void main(String args[])
         {
                Protection2 ob1 = new Protection2();
                OtherPackage ob2 = new OtherPackage();
        }
}
```

Importing Packages

- the **import** statement is used to bring certain classes, or entire packages, into visibility.
- **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.
- This is the general form of the **import** statement:
  import *pkg1*[.*pkg2*].(*classname*|*);
- Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**).

This code fragment shows both forms in use:
import java.util.Date;
import java.io.*;.

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**.

```
import java.lang.*;

import java.util.*;
class MyDate extends Date
{
}

class MyDate extends java.util.Date
{
}
```

- if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**,

```
public class Balance
 {
```

```
        String name;
        double bal;

        public Balance(String n, double b)
        {
                name = n;
                bal = b;
        }
        public void show()
         {
                if(bal<0)
                System.out.print("--> ");
                System.out.println(name + ": $" + bal);
        }
}
```

- the **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package.
- **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;

class TestBalance
{
        public static void main(String args[])
         {
                class and call its constructor. */
                Balance test = new Balance("J. J. Jaspers", 99.88);
                test.show(); // you may also call show()
        }
}
```

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- Once interface is defined, any number of classes can implement an **interface**.
- Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface.

Defining an Interface

An interface is defined much like a class. This is the general form of 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;
// ...
return-type method-nameN(*parameter-list*);
type final-varnameN = value;
}

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it isdeclared.

- When it is declared as **public**, the interface can be used by any other code.
- the methods that are declared have no bodies. They end with a semicolon after the parameter list.
- They are abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized.
- All methods and variables are implicitly **public**.
- Here is an example of a simple interface that contains one method called **callback( )** that takes a single integer parameter.

```
interface Callback
{
        void callback(int param);
}
```

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the **implements** clause:

class *classname* [extends *superclass*] [implements *interface* [,*interface*...]]
{
// class-body
}

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**.
- the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

- Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback
 {
// Implement Callback's interface
        public void callback(int p)
         {
                System.out.println("callback called with " + p);
        }
}
```

- Notice that **callback( )** is declared using the **public** access specifier.
- It is both permissible and common for classes that implement interfaces to define additional members of their own.
- For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
class Client implements Callback
{
// Implement Callback's interface

        public void callback(int p)
        {
                System.out.println("callback called with " + p);
        }
        void nonIfaceMeth()
         {
                System.out.println("Classes that implement interfaces " + "may also define other
                members, too.");
        }
}
```

Accessing Implementations Through Interface References

- we can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When we call a method through one of these references, the correct version will be

called based on the actual instance of the interface being referred to

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface
{
        public static void main(String args[])
         {
                Callback c = new Client();
                c.callback(42);
        }
}
```

output :
callback called with 42

- variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**.
- Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class.
- **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**.

the second implementation of **Callback**, shown here to show the polymorphic behavior:

```
// Another implementation of Callback.

class AnotherClient implements Callback
 {
        public void callback(int p)
        {
                System.out.println("Another version of callback");
                System.out.println("p squared is " + (p*p));
        }
}


class TestIface2
 {
        public static void main(String args[])
        {
                Callback c = new Client();
                AnotherClient ob = new AnotherClient();
                c.callback(42);
                c = ob; // c now refers to AnotherClient object
                c.callback(42);
        }
```

```
}
```

output:

```
callback called with 42
Another version of callback
p squared is 1764
```

the version of **callback( )** that is called is determined by the type of object that **c** refers to at run time.

Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**.
- For example:

```
abstract class Incomplete implements Callback
 {
int a, b;
void show()
{
        System.out.println(a + " " + b);
}
// ...
}
```

- the class **Incomplete** does not implement **callback( )** and must be declared as abstract.
- Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

Applying Interfaces

- a class called **Stack** that implemented a simple fixed-size stack.
- the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation.
- First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**.

This interface will be used by both stack implementations.

```
interface IntStack
 {
void push(int item);
int pop();
}
```

- The following program creates a class called **FixedStack** that implements a fixed-length

version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.

class FixedStack implements IntStack
 {
        private int stck[];
        private int tos;

        FixedStack(int size)
         {
                stck = new int[size];
                tos = -1;
        }

        public void push(int item)
         {
                if(tos==stck.length-1) // use length member
                System.out.println("Stack is full.");
                else
                stck[++tos] = item;
        }

        public int pop()
        {
                if(tos < 0)
                {
                System.out.println("Stack underflow.");
                return 0;
                }
                else
                return stck[tos--];
        }
}

class IFTest
{
        public static void main(String args[])
         {
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);

        for(int i=0; i<5; i++)
         mystack1.push(i);
```

```
        for(int i=0; i<8; i++)
        mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
        System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
        System.out.println(mystack2.pop());
        }
}
```

- another implementation of **DynaStack** that creates a dynamic stack by use of the same **interface** definition.

```
// Implement a "growable" stack.
class DynStack implements IntStack
{
    private int stck[];
    private int tos;
    // allocate and initialize stack
    DynStack(int size)
    {
        stck = new int[size];
        tos = -1;
    }
// Push an item onto the stack
public void push(int item)
 {
    // if stack is full, allocate a larger stack
    if(tos==stck.length-1)
    {
        int temp[] = new int[stck.length * 2]; // double size
        for(int i=0; i<stck.length; i++) temp[i] = stck[i];
        stck = temp;
        stck[++tos] = item;
    }
    else
        stck[++tos] = item;
}
// Pop an item from the stack
public int pop()
{
    if(tos < 0)
    {
        System.out.println("Stack underflow.");
```

```
                return 0;
            }
            else
                return stck[tos--];
            }
        }
        class IFTest2
        {
            public static void main(String args[])
            {
                DynStack mystack1 = new DynStack(5);
                DynStack mystack2 = new DynStack(8);
                // these loops cause each stack to grow
                for(int i=0; i<12; i++) mystack1.push(i);
                for(int i=0; i<20; i++) mystack2.push(i);
                System.out.println("Stack in mystack1:");
                for(int i=0; i<12; i++)
                System.out.println(mystack1.pop());
                System.out.println("Stack in mystack2:");
                for(int i=0; i<20; i++)
                System.out.println(mystack2.pop());
        }
    }
```

- The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push( )** and **pop( )** are resolved at run time rather than at compile time.

```
class IFTest3
 {
        public static void main(String args[])
         {
        IntStack mystack; // create an interface reference variable
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds; // load dynamic stack
        // push some numbers onto the stack
        for(int i=0; i<12; i++) mystack.push(i);
        mystack = fs; // load fixed stack

        for(int i=0; i<8; i++) mystack.push(i);
        mystack = ds;
        System.out.println("Values in dynamic stack:");
```

```
        for(int i=0; i<12; i++)
        System.out.println(mystack.pop());

        mystack = fs;
        System.out.println("Values in fixed stack:");

        for(int i=0; i<8; i++)
        System.out.println(mystack.pop());
        }
}
```

- **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push( )** and **pop( )** defined by the **DynStack** implementation.
- When it refers to **fs**, it uses the versions of **push( )** and **pop( )** defined by **FixedStack**.
- Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends**.
- The syntax is the same as for inheriting classes

```
interface A
 {
        void meth1();
        void meth2();
}

interface B extends A
{
        void meth3();
}

class MyClass implements B
{
        public void meth1()
        {
                System.out.println("Implement meth1().");
        }

        public void meth2()
        {
                System.out.println("Implement meth2().");
        }
        public void meth3()
        {
```

```
            System.out.println("Implement meth3().");
        }
}
class IFExtend
{
        public static void main(String arg[])
        {
            MyClass ob = new MyClass();
            ob.meth1();
            ob.meth2();
            ob.meth3();
        }
}
```

- any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Exception Handling

- an exception is a run-time error.
- languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.
- Java's exception handling avoids handling problems manually and, in the process, brings run-time error management into the object toriented world.

Exception-Handling Fundamentals

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system,
- or they can be manually generated by your code.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Briefly, here is how they work. Program statements that create exceptions are contained within a **try** block.
- If an exception occurs within the **try** block, it is thrown.we can catch this exception (using **catch**) and handle it .
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.

- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

- Here, *ExceptionType* is the type of exception that has occurred.

Exception Types

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch.
- There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error

Uncaught Exceptions

This program includes an expression that intentionally 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** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- Here we don't have 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 our 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)

Using try and catch

- Although the default exception handler provided by the Java run-time system is useful for debugging,we should handle an exception ourself.
- Doing so provides two benefits.
- First, it allows you to fix the error.
- Second, it prevents the program from automatically terminating.
- To handle a run-time error, simply enclose the code inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type to catch

```
class Exc2
{
        public static void main(String args[])
        {
                int d, a;
                try
                {
                     d = 0;
                     a = 42 / d;

                }
                System.out.println("This will not be printed.");
                 catch (ArithmeticException e)
                {
                        System.out.println("Division by zero.");
                }
                System.out.println("After catch statement.");
        }
}
```

This program generates the following output:
Division by zero.
After catch statement.

- A **try** and its **catch** statement form a unit.
- The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement.

```
class HandleError
{
        public static void main(String args[])
        {
                int a=0, b=0, c=0;
                Random r = new Random();

                        for(int i=0; i<32000; i++)
                        {
                                try
                                {
                                        b = r.nextInt();
                                        c = r.nextInt();
                                        a = 12345 / (b/c);
                                }
                                 catch (ArithmeticException e)
                                {
                                        System.out.println("Division by zero.");
                                        a = 0; // set a to zero and continue
                                }
                        System.out.println("a: " + a);
                        }
        }
}
```

Multiple catch Clauses

- more than one exception could be raised by a single piece of code.
- To handle this type of situation, we can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.

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

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.
```

Nested try Statements

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.

```
// An example of nested try statements.
class NestTry
{
        public static void main(String args[])
         {
                try
                {
```

```
                int a = args.length;
                /* If no command-line args are present,
                the following statement will generate
                a divide-by-zero exception. */
                int b = 42 / a;
                System.out.println("a = " + a);
                try
                {
                        // nested try block
                        /* If one command-line arg is used,
                        then a divide-by-zero exception
                        will be generated by the following code. */
                        if(a==1) a = a/(a-a); // division by zero
                        /* If two command-line args are used,
                        then generate an out-of-bounds exception. */
                        if(a==2)
                        {
                                int c[] = { 1 };
                                c[42] = 99; // generate an out-of-bounds exception
                        }
                }
                 catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Array index out-of-bounds: " + e);
                }
        }
        catch(ArithmeticException e)
         {
                System.out.println("Divide by 0: " + e);
         }
    }
}
```

- When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block.
- Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block.
- Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled.
- If we execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block.

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

throw

- 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**.
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

```
class ThrowDemo
{
	static void demoproc()
	 {
		try
		{
			throw new NullPointerException("demo");
		}
		catch(NullPointerException e)
		{
			System.out.println("Caught inside demoproc.");
			throw e; // rethrow the exception
		}
	}
	public static void main(String args[])
	{
		try
		{
			demoproc();
		}
```

```
            catch(NullPointerException e)
            {
                System.out.println("Recaught: " + e);
            }
        }
}
```

- First, **main( )** sets up an exception context and then calls **demoproc( )**.
- The **demoproc( )** method then sets up another exceptionhandling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line.
- The exception is then rethrown.
- Here is the resulting output:

Caught inside demoproc.

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.
- We can 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
}

```
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

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

- Here is the output generated by the preceding program:

inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

Java's Built-in Exceptions

- Inside the standard package **java.lang**, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type **RuntimeException**
- if the method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.*

Java's Unchecked RuntimeException Subclasses Defined in java.lang

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

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

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

Creating Your Own Exception Subclasses

- It is possible to create to create our own exception types to handle situations specific to your applications.
- just define a subclass of **Exception**
- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**.
- Thus, all exceptions, including those that we create, have the methods defined by **Throwable** available to them.

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}

class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Normal exit");
  }

  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```