S. J. P. N. TRUST'S
HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI
Accredited at 'A' Grade by NAAC
Programmes Accredited by NBA: CSE, ECE, EEE & ME.
Department of Computer Science & Engineering

Course: Programming in Java(18CS653)

Module 3: Introducing Classes

Prof. Prasanna Patil
Asst. Prof. , Dept. of Computer Science & Engg.,
Hirasugar Institute of Technology, Nidasoshi

Class

- Class is a user-defined data type which contains instance variables & methods.
- A class is declared by use of the class keyword. A simplified general form of a **class definition is shown here:**

The General Form of a Class:

```
class classname
{
type     instance-variable1;
.
.
.
type     instance-variableN;
type    methodname1(parameter-list)
{
// body of method
}
type     methodnameN(parameter-list)
{
// body of method
}
}
```

A Simple Class

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

Declaring Objects:

- When you create a class, you are creating a new data type. You can use this type to declare objects of that type.

```
Box mybox = new Box();
```

- The **new** operator dynamically allocates memory for an object.

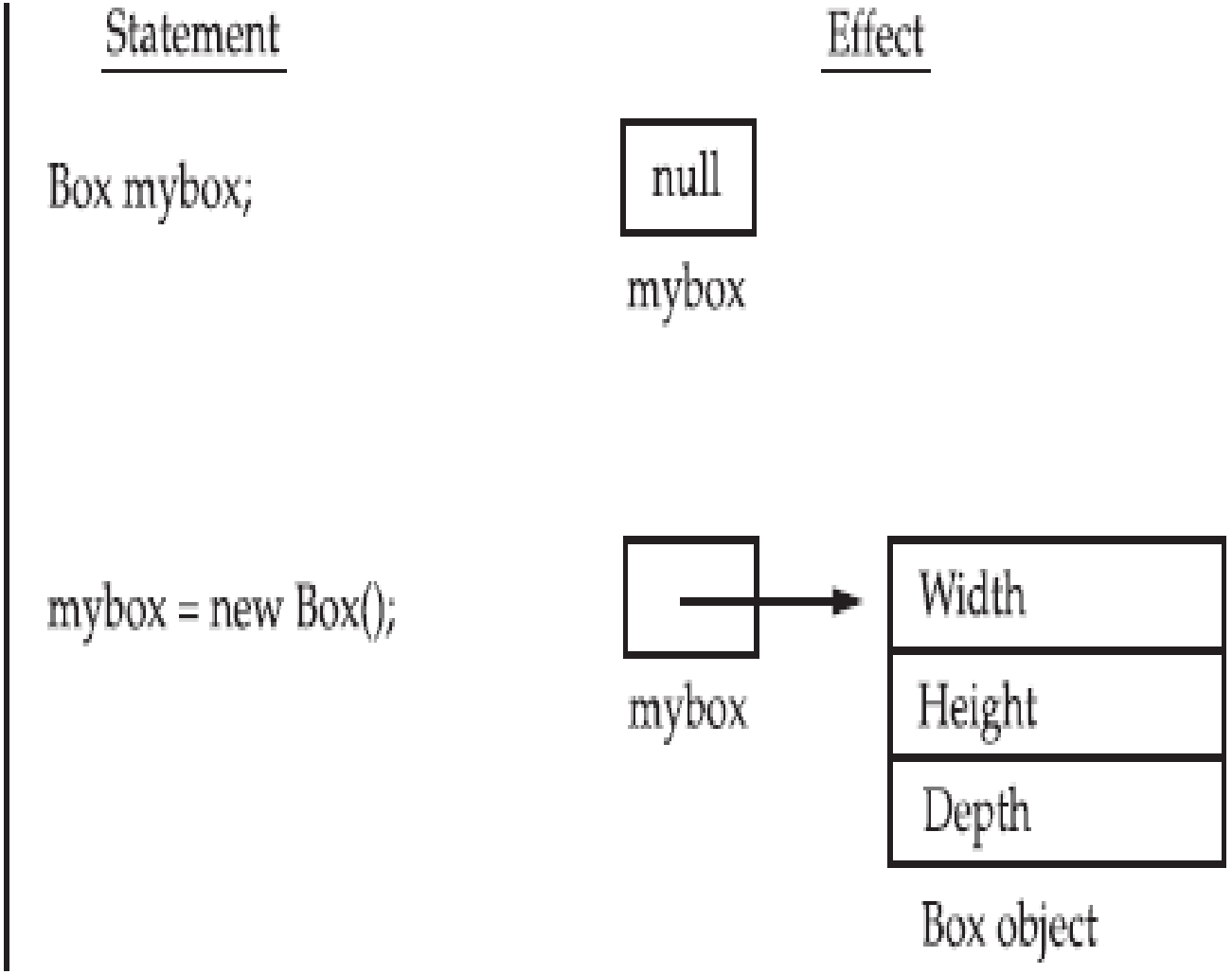
A program that uses the Box class

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

```
// This class declares an object of type Box.
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;

     // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;
    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;
    System.out.println("Volume is " + vol);
  }
}
```

FIGURE 6-1
Declaring an object of type **Box**



This program declares two Box objects.

```
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;
  // assign values to mybox1's instance
     variables
  mybox1.width = 10;
  mybox1.height = 20;
  mybox1.depth = 15;
  /* assign different values to mybox2's instance
     variables */
  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

A Closer Look at new

- The new operator dynamically allocates memory for an object. It has this general form:

  class-var = new classname( );

- Here, class-var is a variable of the class type being created.
- The classname is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the constructor for the class.

- It is important to understand that new allocates memory for an object during run time.
- The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.
- However, since memory is finite, it is possible that new will not be able to allocate memory for an object because insufficient memory exists.
- If this happens, a run-time exception will occur.

Distinction between class and object

- A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members.
- When you declare an object of a class, you are creating an instance of that class.
- Thus, a class is a logical construct.
- An object has physical reality.

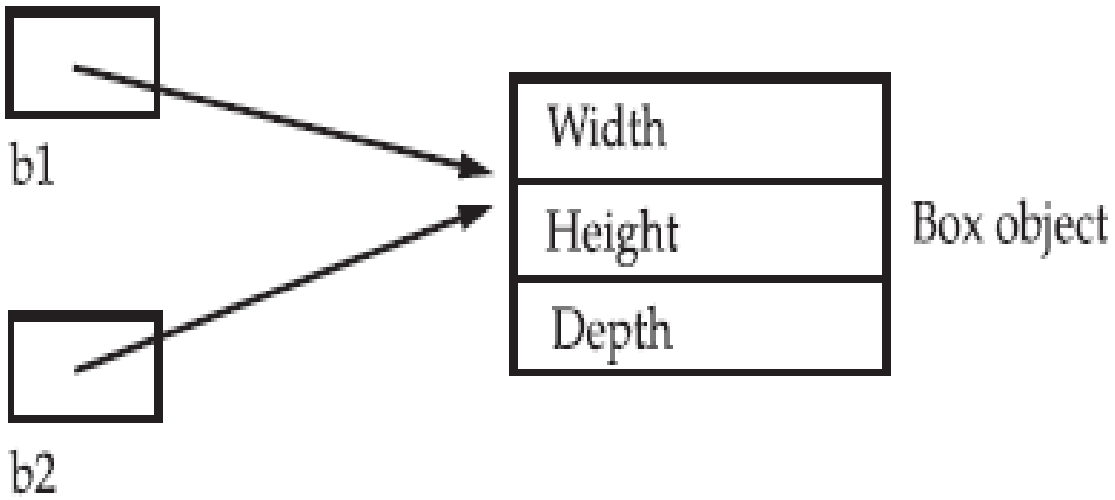
Assigning Object Reference Variables

- Object reference variables act differently than you might expect when an assignment takes place.

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

- you might think that b1 and b2 refer to separate and distinct objects.
- After this fragment executes, 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.



Introducing 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.

Accessing class members:

- Class members can be accessed using dot(.) operator as follows:

  ObjectName.VariableName

  ObjectName.methodName(parameter_list);

Adding a Method to the Box Class:

```
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();
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;
    mybox1.volume();
  }
}
```

Output:
Volume is 3000.0

Returning a Value:

```
class RECT {
  double   length;
  double    breadth;
  void    set(double a, double b) {
    length = a;
    breadth = b;
  }
  double  area( ) {
     return (length * breadth);
  }
}

class RECTDemo {
  public static void main(String args[ ]) {
    RECT b1 = new RECT( );
    b1.set(4.0,5.0);
    System.out.println("Area of Rectangle = " +b1.area( ));
  }
}
```

Constructors :

- It is a special method which is used to initialize the values of instance-variables at the time of creation of objects.
- **Features:**
  - Constructor will have same name as that of class name.
  - It does not specify a return type not even void.
  - It should be declared in public section.

- Types of Constructors:
  1. **default constructor: It is a constructor which do not take any argument.**
  2. **Parameterized constructor: It is a constructor which takes any number of parameters.**
- Default constructor is automatically loaded by the compiler.

Example of default constructor:

```
class B {
  int x;
  B( ) {
    System.out.println("Constructing Box");
    x = 10;
  }
  void show( ) {
    System.out.println("x = " + x);
  }
}

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

Output:

Constructing Box

x = 10

Example of Parameterized Constructor :

```
class B {
   int x, y;
   B(int a, int b) {
      x = a;
      y = b;
   }
   void show( ) {
      System.out.println("x = " + x);
      System.out.println("y = " + y);
   }
}
class MB {
   public static void main(String args[]) {
      B b1 = new B(4, 5);
      b1.show( );
   }
}
```

Output:

x = 4

y = 5

The this Keyword:

- 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.
- Eg:

```
class Box {
  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.
- When a local variable has the same name as an instance variable, the local variable hides the instance variable.
- You can use this keyword to resolve any name space collisions that might occur between instance variables and local variables.

```
Box(double width, double height, double depth)
{
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

program to implement the stack of 10 integer values.

```
class Stack {
  int stck[] = new int[10];
  int top;
  Stack() {
    top = -1;
  }
  void push(int item) {
    if(top == 9)
      System.out.println("Stack is full.");
    else
    {
      stck[++top] = item;
      System.out.println(“Pushed item is = “+stck[top]);
    }
  }
```

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

class TestStack {
  public static void main(String args[])
  {
    Stack k1 = new Stack();
    k1.push(10);
    k1.push(20);
    k1.push(30);
    System.out.println(“popped item is = “+k1.pop());
    System.out.println(“popped item is = “+k1.pop());
  }
}
```

Garbage Collection

- Since objects are dynamically allocated by using the new operator, you might be wondering 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 takes a different approach; it handles deallocation for you automatically.
- The technique that accomplishes this is called garbage collection.
- It works like this:
  - 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.
  - There is no explicit need to destroy objects as in C++.
  - Garbage collection only occurs sporadically (if at all) during the execution of your program.
  - It will not occur simply because one or more objects exist that are no longer used.

- **The finalize( ) Method:**
  - Sometimes an object will need to perform some action when it is destroyed.
  - For example, if an object is holding some non-Java resource such as a file handle or character font, then you 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, you 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, you 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 garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.
- Right before an asset is freed, the Java run time calls the finalize( ) method on the object.
- 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.

Overloading Methods

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

- 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);
  }
  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }
  double test(double a) {
     System.out.println("double a: " + a);
     return a*a;
  }
}
```

```
class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;
    // 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);
  }
}
```

Output

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

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact.
- In some cases, Java's automatic type conversions can play a role in overload resolution.
- For example, consider the following program:

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}

// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}

// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}

void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
```

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

Output
No parameters
a and b: 10 20
a: 88
Inside test(double) a: 123.2

Overloading Constructors

- In addition to overloading normal methods, you can also overload constructor methods.
- The method of defining more than one constructors inside a class is called as Constructor Overloading.
- In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.

```
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;
// get volume of first box
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);
}
}
```

Output :

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Using Objects as Parameters

- It is both correct and common to pass objects to methods.
- For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```

Output :
ob1 == ob2: true
ob1 == ob3: false

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a subroutine.
  - Call by Value
  - Call by Reference

Call by Value

- This approach copies the value of an argument into the formal parameter of the subroutine.
- Therefore, changes made to the parameter of the subroutine have no effect on the argument.

```
// Primitive types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
}
}
```

Output :

```
a and b before call: 15 20
a and b after call: 15 20
```

Call by Reference

- In this approach, a reference to an argument (not the value of the argument) is passed to the parameter.
- Inside the subroutine, this reference is used to access the actual argument specified in the call.
- This means that changes made to the parameter will affect the argument used to call the subroutine.

```
// Objects are passed by reference.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o) {
o.a *= 2;
o.b / = 2;
}
}
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);
}
}
```

Output :

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

Note :

- When a primitive type is passed to a method, it is done by use of call-by-value.
- Objects are implicitly passed by use of call-by-reference.

Returning Objects

- A method can return any type of data, including class types that you create.
- For example, in the following program, the incrByTen( ) method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
} }
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: “ + ob2.a);
}
}
```

Output :

ob1.a: 2

ob2.a: 12

ob2.a after second increase: 22

Recursion

- Java supports recursion.
- Recursion is the process of defining something in terms of itself.
- Recursion is the attribute that allows a method to call itself.
- A method that calls itself is said to be recursive

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
int fact(int n) {
  int result;
  if(n==1) return 1;
  result = n* fact(n-1) ;
  return result;
 }
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

Output :

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

- When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start.
- As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.
- Recursive methods could be said to "telescope" out and back.

- Advantages :
  - The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.
  - Some types of AI-related algorithms are most easily implemented using recursive solutions.
- Disadvantages:
  - Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls.
  - Many recursive calls to a method could cause a stack overrun.

```
// Another example that uses recursion.
class RecTest {
int values[];
RecTest(int i) {
values = new int[i];
}
// display array -- recursively
void printArray(int i) {
if(i==0) return;
else printArray(i-1);
System.out.println("[" + (i-1) + "] " + values[i-1]);
}
}
class Recursion2 {
public static void main(String args[]) {
RecTest ob = new RecTest(10);
int i;
for(i=0; i<5; i++)
 ob.values[i] = i;
ob.printArray(5);
}
}
```

Output :

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

Introducing Access Control

- It may be necessary in some situations to restrict the access to certain variables & methods from outside the class.
- This can be achieved in Java by applying visibility modifiers or access specifiers.
- Java supports four types of access specifiers.
  - Public : It has a widest possible visibility & accessible everywhere.
  - No Specifier : When no access modifier is specified, the member defaults to a limited version of public accessibility known as “friendly” level of access.
  - Protected : The visibility level of protected field lies in between the public & friendly access.
  - Private : They are accessible only within their own class.

TABLE 9-1
Class Member Access

	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

```
/* This program demonstrates the
    difference between
public and private.
*/
class Test {
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
}
}
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be
    accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an
    error
// ob.c = 100; // Error!
// You must access c through its
    methods
ob.setc(100); // OK
System.out.println("a, b, and c: " +
    ob.a + "," +
ob.b + " ," + ob.getc());
}
}
```

- Output :
- a, b, and c: 10, 20, 100

Understanding static:

- It is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword static.
- You can declare both methods and variables to be static.
  - If instance-variable is declared with static, all instances of the class share the same static variable.
  - If method is declared with static, then by using class name that method can be called.
- Methods declared as static have several restrictions:
  - 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);
}
}
```

Output :

```
Static block initialized.
x = 42
a = 3
b = 12
```

- Outside of the class in which they are defined, static methods and variables can be used independently of any object.
- To do so, you need only specify the name of their class followed by the dot operator.
- For example, if you wish to call a static method from outside its class, you can do so using the following general form:

  classname.method( )

- Here, classname is the name of the class in which the static method is declared.

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Output :
a = 42
b = 99

Introducing final

- A variable can be declared as final. Doing so prevents its contents from being modified.

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

- Variables declared as final do not occupy memory on a per-instance basis.
- Thus, a final variable is essentially a constant.

Arrays Revisited

- Arrays are implemented as objects.
- Because of this, there is a special array attribute that you will want to take advantage of.
- Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its length instance variable.
- All arrays have this variable, and it will always hold the size of the array.

```
// This program demonstrates the length array
   member.
class Length {
public static void main(String args[]) {
int a1[] = new int[10];
int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is " + a1.length);
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
}
}
```

Output :

length of a1 is 10
length of a2 is 8
length of a3 is 4

```
// Improved Stack class that uses the length
      array member.
class Stack {
private int stck[];
private int top;
// allocate and initialize stack
Stack(int size) {
stck = new int[size];
top = -1;
}
// Push an item onto the stack
void push(int item) {
if(top==stck.length-1) // use length member
System.out.println("Stack is full.");
else
stck[++top] = item;
}
// Pop an item from the stack
int pop() {
if(top < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[top--];
}
}
class TestStack2 {
public static void main(String args[]) {
Stack mystack1 = new Stack(5);
Stack mystack2 = new Stack(8);
// push some numbers onto the stack
for(int i=0; i<5; i++) mystack1.push(i);
for(int i=0; i<8; i++) mystack2.push(i);
// pop those numbers off the stack
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());
}
}
```

Inheritance

Introduction

- **Inheritance is the process by which one object acquires the properties of another object.**
- Using inheritance, you can create a general class that defines traits common to a set of related items.
- This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- In the terminology of Java, a class that is inherited is called a **superclass .**
- The class that does the inheriting is called a **subclass .**

Syntax

- class Superclassname

{

……

}

- class subclassname extends Superclassname

{

……

}

Types of Inheritance:

1. Single Inheritance.
2. Hierarchical Inheritance
3. Multilevel Inheritance
4. Multiple Inheritance

Using super Keyword :

- super has two general forms.
  - To Call Superclass Constructors.
  - To access a member of the superclass that has been hidden by a **member of a subclass.**

1. Using Super to call superclass constructor:

- A subclass 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.

Example

```
class test
{
private int a, b;
public test(int x, int y)
{
a=x;
b=y;
}
void show()
{
System.out.println(“ a : “ +a );
System.out.println(“ b : “ +b );
}
}
class subtest extends test
{
private int c;
public subtest(int x,int y, int z)
{
super(x,y);
c=z;
}
void display()
{
show();
System.out.println(“ c : “ +c );
}
}

public class Test1
{
public static void main(String args[])
{
subtest S  =  new subtest(4,8,14);
S.display();
}
}
```

Output :
a: 4
b : 8
c: 14

2. To access a member of the superclass that has been hidden by a member of a subclass:

```
class SSIP
{
int i;
}
class SCSIP extends SSIP
{
int i;
SCSIP(int a, int b)
{
super.i = a;
i = b;
}
void display( )
{
System.out.println(“ Super class I =
     ”+super.i);
System.out.println(“ Sub class I = ”+i);
}
}
public class Test1
{
public static void main(String args[])
{
SCSIP S  =  new SCSIP(10,20);
S.display();
}
}
```

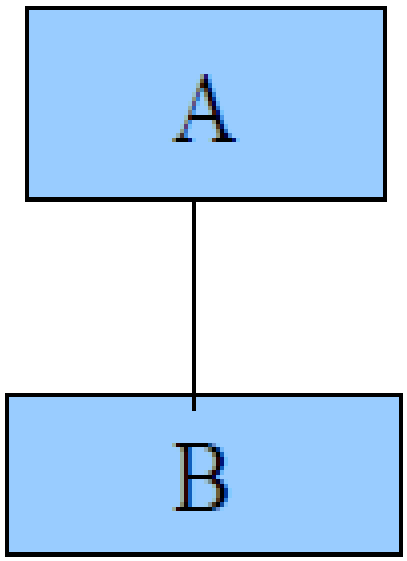
Output:

Super class I = 10

Sub class I = 20

1. Single Inheritance:

- **The process of deriving one subclass from one superclass is called as Single Inheritance.**



Example

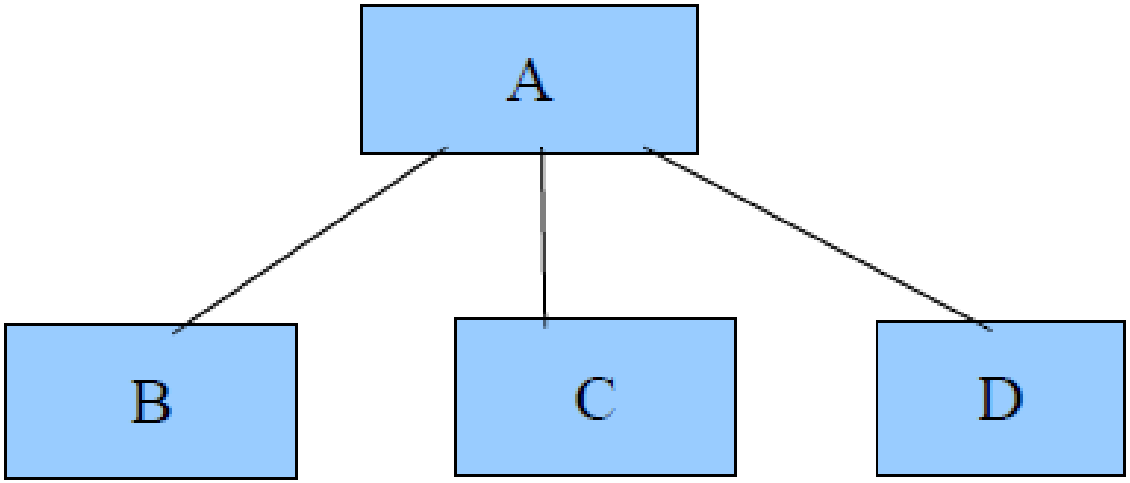
```
class SSIP
{
int i;
SSIP(int a)
{
i=a;
}
void show( )
{
System.out.println(“i= “+i);
}
}
```

```
class SCSIP extends SSIP
{
int j;
SCSIP(int a, int b)
{
super(a);
j=b;
}
void display( )
{
System.out.println(“j= “+j);
}
}
```

```
class MSSIP
{
public static void main(String args[ ])
{
SCSIP s1 = new SCSIP(10,20);
s1.show( );
s1.display( );
}
}
```

2. Hierarchical Inheritance :

- The process of deriving more than one subclass from the single superclass is called as hierarchical inheritance.



```
class PERSON
{
String name, address;
int id;
PERSON( ) { }
PERSON(String s, int a, String t)
{
name = s;
id = a;
address = t;
}
void show( )
{
System.out.println(“Name of the Person = ”+name);
System.out.println(“ID of Person = ”+id);
System.out.println(“Address of Person = ”+address);
}
}
```

```
class EMP extends PERSON
{
double sal, inc;
EMP( ) { }
EMP(String s, int a, String t, double v, double k)
{
super(s, a, t);
sal = v;
inc = k;
}
void cal( )
{
sal = sal + inc;
}
void print( )
{
show( );
System.out.println(“increment = ”+inc);
System.out.println(“New salary = ”+sal);
}}
```

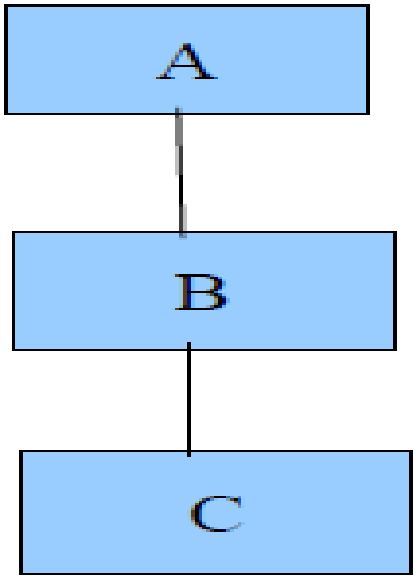
```
class STUDENT extends PERSON
{
int m1, m2, m3;
double avg;
STUDENT( ) { }
STUDENT(String s, int a, String t, int i, int j, int k)
{
super(s, a, t);
m1 = i;
m2 = j;
m3 = k;
}
void display( )
{
show( );
System.out.println(“M1 = ”+m1);
System.out.println(“M2 = ”+m2);
System.out.println(“M3 = ”+m3);
System.out.println(“avg = ”+avg);
}
```

```
void average( )
{
avg = (double) m1+m2+m3/3;
}
}
```

```
class PES
{
public static void main(String args[ ])
{
EMP E1 = new EMP(“Sagar”, 10, “Delhi”, 10000, 2000);
E1.cal( );
E1.print( );
STUDENT S1=new STUDENT(“laxmi”,1, “ Belgaum”, 23,25, 24);
S1.average( );
S1.display( );
}
}
```

3. Multilevel Inheritance:

- The process of deriving new class from the derived class is called as Multilevel inheritance.



```
class STUDENT
{
String name;
int rollno;
STUDENT( ) { }
STUDENT(String s, int r)
{
name = s;
rollno = r;
}
void show( )
{
System.out.println(“Name = ”+name);
System.out.println(“Roll No = ”+rollno);
}}
```

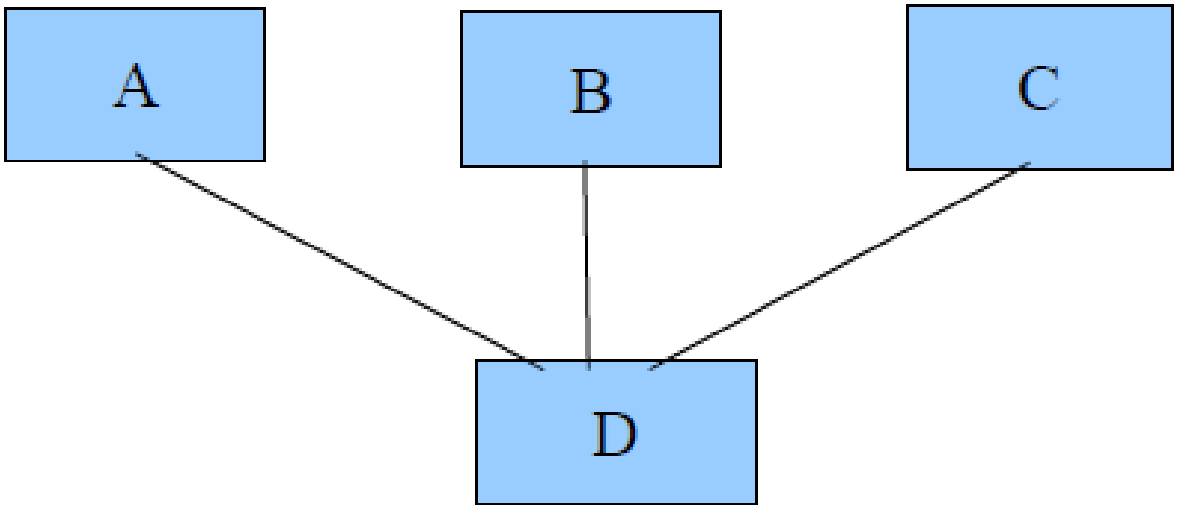
```
class TEST extends STUDENT
{
int m1,m2;
TEST( ) { }
TEST(String s, int r, int x, int y)
{
super(s, r);
m1=x;
m2=y;
}
void display( )
{
show( );
System.out.println(“M1 = ”+m1);
System.out.println(“M2 = ”+m2);
}}
```

```
class RESULT extends TEST
{
double avg;
RESULT( ) { }
RESULT(String s, int r, int x int y)
{
super(s,r,x,y);
}
void cal( )
{
avg = (double) m1+m2 / 2;
}
void Print( )
{
display( );
System.out.println(“avg = “+avg);
}}
```

```
class MSTUDENT
{
public static void main(String args[ ])
{
RESULT R1=new RESULT(“abc”,1,23,25);
R1.cal( );
R1.Print( );
}}
```

4. Multiple Inheritance :

- The process of deriving a single subclass from more than one superclasses is called as Multiple inheritance.



- This type of inheritance is not directly implemented by Java because of
  - **Complexity**
  - **Large Memory Requirement**
  - **Security**
- Same property is achieved in Java by a concept called as “Interface”.

Method Overriding

- In a class hierarchy, 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.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden

```
// Method overriding.
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;
}
// display k – this overrides show() in
    A
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
```

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

- A superclass reference variable can refer to a subclass object.
- Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

```
// Dynamic Method Dispatch
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
}
}
```

Output :

Inside A’s callme method
Inside B’s callme method
Inside C’s callme method

Why Overridden Methods?

- Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness.
- The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

```
class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
double area() {
System.out.println("Area for Figure is undefined.");
return 0;
}
}
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 FindAreas {
public static void main(String args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
figref = f;
System.out.println("Area is " + figref.area());
}
}
```

Output:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

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.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- This is the case with the class Figure used in the preceding example.
- The definition of area( ) is simply a placeholder. It will not compute and display the area of any type of object.

- Java's solution to this problem is the abstract method.
- You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.
- These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass.
- Thus, a subclass must override them—it cannot simply use the version defined in the superclass.
- To declare an abstract method, use this general form:

  abstract type name(parameter-list);
- As you can see, no method body is present.

- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.
- Such objects would be useless, because an abstract class is not fully defined.
- Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

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

```
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.
  - First, it can be used to create the equivalent of a named constant.
  - Using final to Prevent Overriding
  - Using final to Prevent Inheritance

To create the equivalent of a named constant.

- A variable can be declared as final. Doing so prevents its contents from being modified.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
```

- Variables declared as final do not occupy memory on a per-instance basis.
- Thus, a final variable is essentially a constant.

Using final to Prevent Overriding

- 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

- Sometimes you will want to prevent a class from being inherited.
- To do this, precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too.

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

The Object Class

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

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is 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 the hash code 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( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.