SCHOOL OF FUTURE TECH

ITM SKILLS UNIVERSITY

COURSE:- Java Programming
Sem:- III
Module: 2

By:
Dr. Ritu Jain,
Associate Professor,
Computer Science-ISU

Class

- A class is a user-defined blueprint or template from which objects are created. It represents the set of properties or methods that are common to all objects of one type.
- Class defines a new data type. Once defined, this new type can be used to create objects of that type.
- A class is a template for an object and an object is an instance of a class.
- When you define a class, you specify the
  - data it contains and
  - the code that operates on that data.

Object

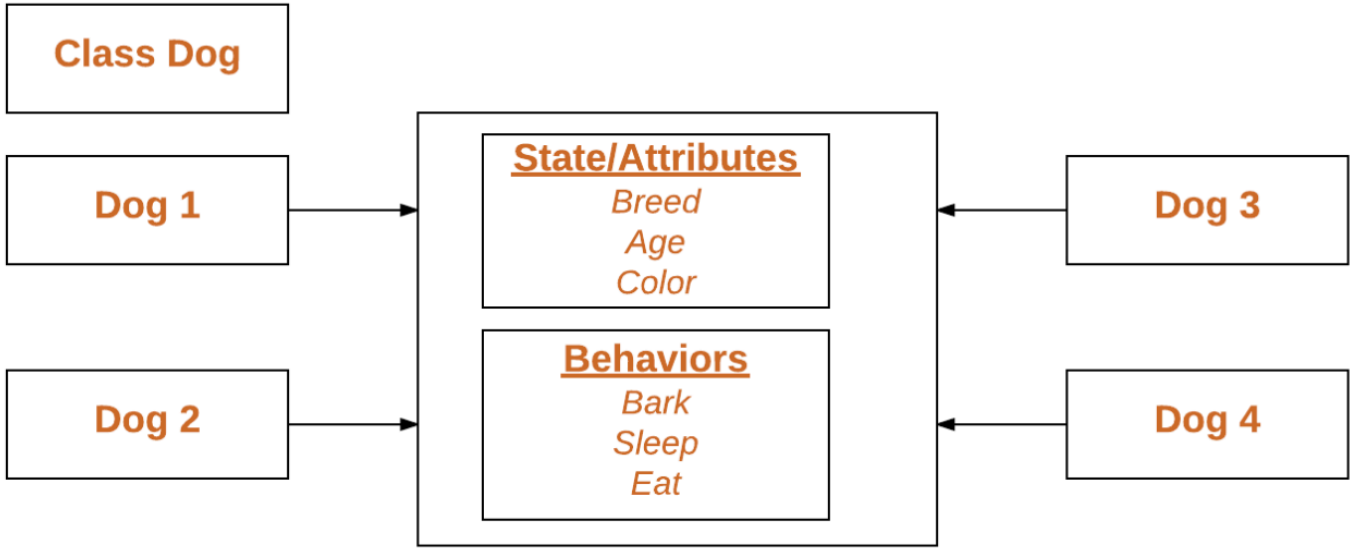
- It represents real-life entity.
- Real-life entities share two characteristics: they all have attributes and behavior.

An object consists of:

- **State:** It is represented by *attributes* of an object. It also shows properties of an object.
- **Behavior:** It is represented by *methods* of an object. It shows response of an object with other objects.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Declaring Objects

Declaring Objects (Also called instantiating a class)
When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.



Basic General form of a class

```
class classname {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
      // body of method
    }
    type methodname2(parameter-list) {
      // body of method
    }
    // ...
    type methodnameN(parameter-list) {
       // body of method
    }
}
```

Class

- The variables defined within a class are called *instance variables*.
- The code is contained within *methods*.
- The variables and the methods within a class are called as *members* of the class.
- Each instance of a class contains its own copy of the instance variables. This implies that the data for one object is separate and unique from the data of another object.
- Remember that Java classes do not need to have a **main()** method. You only specify a **main()** method in a class if that class is the starting point of your program.

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

The output produced by this program is shown here:

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

How to create objects

- A class declaration only creates a template; it does not create an actual object.
- General form to create object:
  - *Class_name class-var* = new *classname* ( );

- Example:
  - Box mybox = new Box();

OR

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

- The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is then stored in the variable.
- The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created.
- If no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor.

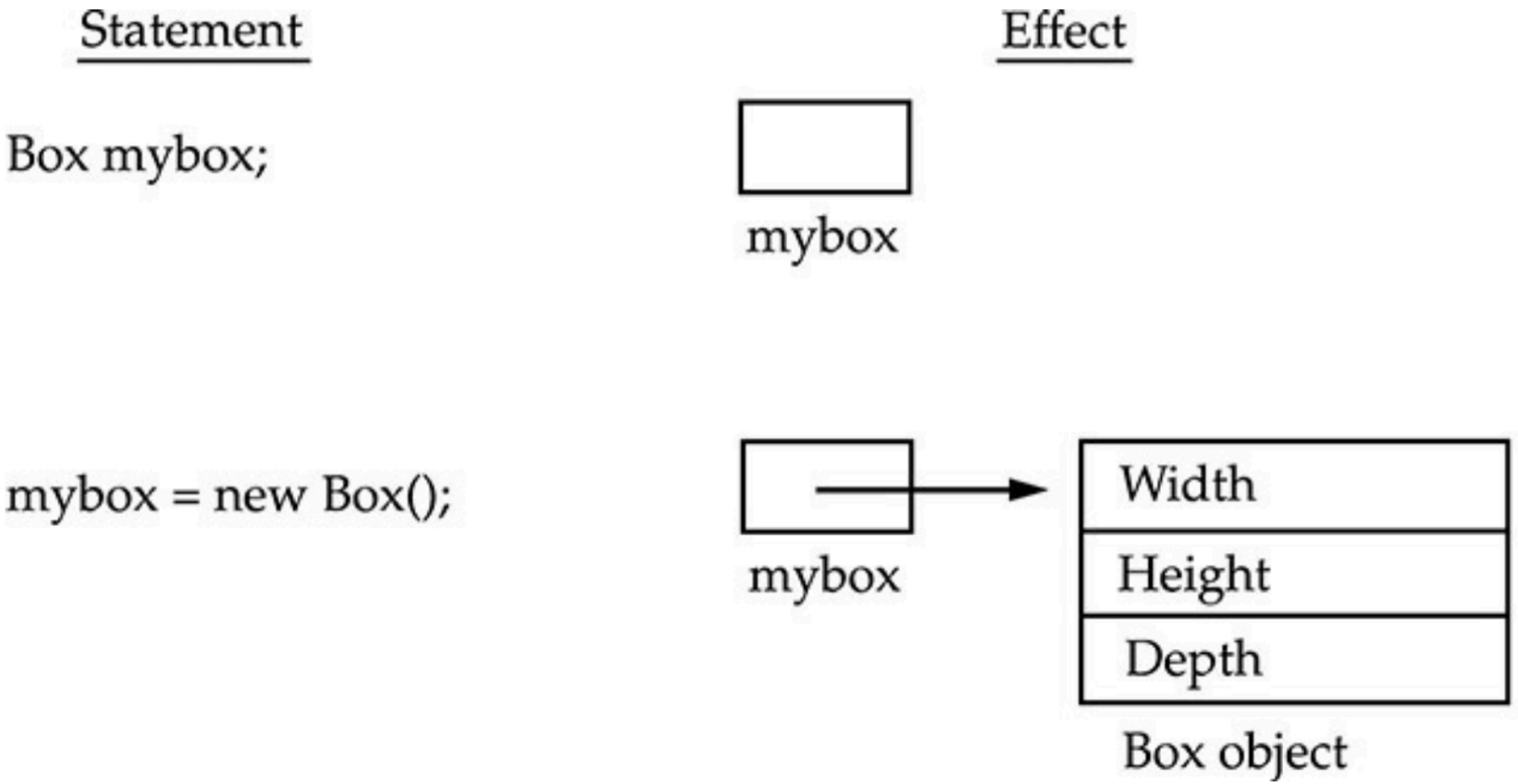
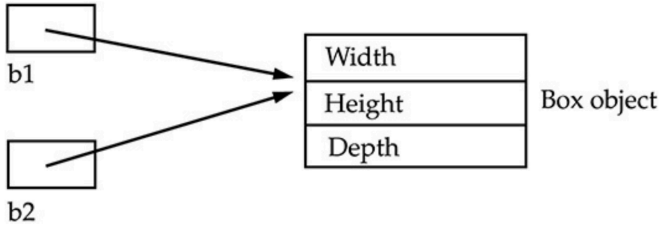


Figure 6-1 Declaring an object of type **Box**

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. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

Assigning Object Reference Variables

- Box b1 = new Box(); Box b2 = b1;
- 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.
- Box b1 = new Box(); Box b2 = b1;
  b1 = null;
- Here, **b1** has been set to **null**, but **b2** still points to the original object.
- **REMEMBER** When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Methods in class

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

  type name(*parameter-list*) {
  // body of method }
- *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope.
- The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:
    - return *value*;
- Here, *value* is the value returned.

Methods in class

- Methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions.
- In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Accessing class members

- *dot (.) Operator is used to access instance variable as well as methods of a class.*
- Every object have its own copy of instance variable. Changes to the instance variables of one object have no effect on the instance variables of another.
- Call a variable as follows:
  - objectname.variablename;
  - Ex: mybox1.width=10;
- Call a class method as follows
  - objectname.methodname();
  - Ex: mybox1.volume();

Example 1 to show how to define classes, instance variable and methods in the class. Create objects and access instance variables and call instance methods of a class through objects

```
// This program includes a method inside the box class.

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

  // display volume of a box
  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();

    // 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;

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

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

Accessing class members

- When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator.
- However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly.
- The same thing applies to methods.

```
class Employee {
   // declaring variables
   int emp_id;
   String name;
   String dept;
   float salary;
   void add_info (int id, String n, String d, float sal) {
      this.emp_id = id;
      this.name = n;
      this.dept = d;
      this.salary = sal;
   }
    void display() {
      System.out.println("Employee id: " + emp_id );
      System.out.println("Employee name: " + name );
      System.out.println("Employee department: " + dept );
      System.out.println("Employee salary: " + salary );
   }
}
```

```
public class EmployeeClass {
    public static void main(String[] args) {
        // creating objects of class Employee
    Employee e1 = new Employee();
    Employee e2 = new Employee();
    Employee e3 = new Employee();
        e1.add_info (101, "Naman", "Salesforce", 45000);
    e2.add_info (102, "Riya", "Tax", 25000);
    e3.add_info (103, "Anu", "Development", 55000);
      e1.display();
    e2.display();
    e3.display();
    }
}
```

Constructor

- It can be time consuming to initialise all of the variables in a class each time when they are created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created.
- This automatic initialisation is performed through the use of a constructor. A constructor initialises an object as soon as it is created.
- **Rules for constructor:**
  - *It has the same name as the name of the class in which it resides.*
  - *Constructors do not have a return type, not even* ***void****.* This is because the implicit return type of the class' constructor is the class type itself. This is because the implicit return type of a class' constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

Example of Constructor with no parameters

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

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

class BoxDemo6 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    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);
  }
}
```

If Constructor is not defined?

- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor.
- When using the default constructor, all non- initialized instance variables will have their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively.

Parameterized Constructors

- We saw that the above constructor initialized all the objects with the same values.
- But actually what is needed in practice is that we should be able to set different initial values.
- For this purpose, we make use of parameterized constructors and each object gets initialized with the set of values specified in the parameters to its constructor.
- The following modification in the above program will illustrate :

Example of Parameterized Constructor

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume() {

    return width * height * depth;
  }
}

class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

this Keyword

- **this** is a keyword that can be used inside any method to refer to the current object i.e **this** is always a reference to the object on which the method was invoked. This is useful when a method needs to refer to the object that invoked it.

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

//use of **this** is redundant, but perfectly correct.

Instance Variable hiding

- In Java, it is illegal to declare two local variables with the same name within the same or enclosing scopes.
- However, you can have local variables, including formal parameters to methods which overlap with the names of the class' instance variables.
- But when a local variable has the same name as the instance variable, then the local variable hides the instance variable.
- We can use the **this** keyword to refer directly to the object and thus resolve the name space collisions that might occur between the instance variables and the local variables.

Using *this*

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

Getters and Setters

- In Java, Getter and Setter are methods used to protect your data and make your code more secure. Getter and Setter make the programmer convenient in setting and getting the value for a particular data type.
- **Getter in Java:** Getter returns the value (accessors), it returns the value of data type int, String, double, float, etc. For the program's convenience, the getter starts with the word "get" followed by the variable name.
  - Eg: public int getNum(){return num;}
- Getter method begin with "is" if return type is boolean.
  - Eg. public boolean isHappy(){ return happy;}
- **Setter in Java:** While Setter sets or updates the value (mutators). It sets the value for any variable used in a class's programs. and starts with the word "set" followed by the variable name.
  - Eg. public void setNum(int n){ num=n;}

Example program: getters and setters

```
class Box1{
    double width,height,depth;
    double volume(){
        return (width*height*depth);
    }
    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double getDepth() {
        return depth;
    }

    public void setDepth(double depth) {
        this.depth = depth;
    }
}

public class BoxDemoEx2 {
    public static void main(String []args){
        Box1 b1=new Box1();
        b1.width=b1.height=b1.depth=5;
        b1.volume();
    }
}
```

Access Control through Access modifiers

- **public:** Member can be accessed by any other code. The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
- p**rivate:** member cannot be accessed from outside the class.
- **protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Default access level:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

```java
// Example for concept of class:  DogClass
public class DogClass {
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public DogClass(String name, String breed, int age,
                String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }
    public String getName() { return name; }
    public String getBreed() { return breed; }
    public int getAge() { return age; }
    public String getColor() { return color; }

    public String toString()
    {
        return ("Hi my name is "+name
                + ".\nMy breed,age and color are "
                + breed + "," + age
                + "," + color);
    }

    public static void main(String[] args)
    {
        DogClass tuffy
                = new DogClass("tuffy", "papillon", 5, "white");
        System.out.println(tuffy.toString());
    }
}
```

Example 2 to show how to define classes and create objects and call parametrized instance methods, getters, setters

```
class Rectangle
{
        private int length=0, width=0;
        void setData(int x, int y)
        {
                length=x;
                width=y;
        }
      int getLength(){return length;
      }
       int getWidth(){return width;}
        int rectArea()
        {
                int area=length*width;
                return(area);
        }
}
class RectArea
{
        public static void main(String args[])
        {
           int area;
           Rectangle rect=new Rectangle();
           rect.setData(10,20);
          area=rect.rectArea();
System.out.println("Area:"+area);
        }
}
```

Ex of Class Concept:Rectangle

```
public class Circle {

    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return Math.PI * radius * radius;
    }
    public double getCircum) {
        return 2 * Math.PI * radius;
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        int r = 5;
        Circle circle = new Circle(r);
        System.out.println("Radius of the circle is " + r);

        System.out.println("The area of the circle is " + circle.getArea());

        System.out.println("The circumference of the circle is " +
circle.getCircum());

        r = 8;
        circle.setRadius(r);

        System.out.println("\nRadius of the circle is " + r);

        System.out.println("The area of the circle is now " +
circle.getArea());

        System.out.println("Circumference of the circle is now " +
circle.getCircum());
    }
}
```

Compile Time Polymorphism: Method Overloading

Compile Time Polymorphism: Method Overloading

- Method Overloading allows different methods to have the *same name, but different signatures*.
- Signature can differ by either:
  - number of parameters
  - type of parameters
  - mixture of both
- Java supports overloading. It is one of the ways that Java implements the concept of polymorphism.
- Method overloading supports polymorphism because it is one way that Java implements "one interface, multiple methods" paradigm.
- In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data.

Compile Time Polymorphism: Method Overloading

- Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name.
- For instance, in C, the function **abs( )** returns the absolute value of an integer, **labs( )** returns the absolute value of a long integer, and **fabs( )** returns the absolute value of a floating-point value.
- Since C does not support overloading, each function has its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember.
- This situation does not occur in Java, because each absolute value method can use the same name.

Compile Time Polymorphism: Method Overloading

- The value of overloading is that it allows related methods to be accessed by use of a common name.
- Thus, the name **abs** represents the *general action* that is being performed. It is left to the compiler to choose the right *specific* version for a particular circumstance.
- You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one.
- You should only overload closely related operations.

Overloading Methods

- When an overloaded method is invoked, *Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.*
- *Thus, overloaded methods must differ in the type and/or number of their parameters.*
- *While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.*
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
- Remember, return types do not play a role in overload resolution.

```
// 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);
  }

  // Overload test for a double parameter
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}

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

    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " + result);
  }
}
```

Example1: Method Overloading

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
```

Result of ob.test(123.25): 15190.5625

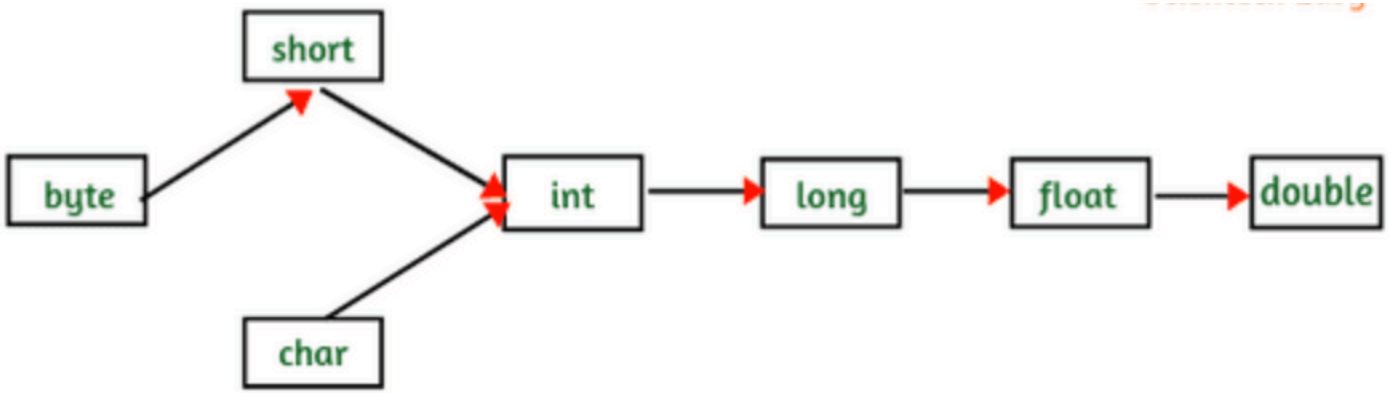
// Example 2: Java program to demonstrate working of method overloading in Java

```
public class Sum {
    public int sum(int x, int y) { return (x + y); }

    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    public double sum(double x, double y)
    {
        return (x + y);
    }

    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output:

```
30
60
31.0
```

Method Overloading: What if the exact prototype does not match with arguments?

- ***Java's automatic type conversions can play a role in overload resolution.***
- 

- **Note: Java tries to use the most specific parameter list it can find.**

//Example: ***Java's automatic type conversions can play a role in overload resolution.***

```
class Demo {
    public void show(int x)
    {  System.out.println("In int" + x);
    }
    public void show(String s)
    { System.out.println("In String" + s);
    }
    public void show(byte b)
    { System.out.println("In byte" + b);
    }
}
class MethodOverloadAutoConv{
    public static void main(String[] args)
    {
        byte a = 25;
        Demo obj = new Demo();

        // byte argument
        obj.show(a);

        // String
        obj.show("hello");

        // int
        obj.show(250);

        // Since char is not available, so the datatype higher than char in terms of range is int.
        obj.show('A');

        // String
        obj.show("A");

        // since float datatype is not available and so it's higher datatype, so at this step there will be an error.
        obj.show(7.5);
    }
}
```

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

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

  // Overload test for a double parameter
  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)
  }
}
```

Example2: Automatic type conversion in Method Overloading

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

Overloading Methods: *Automatic Type Conversion*

- **OverloadDemo** does not define **test(int)**. Therefore, when **test( )** is called with an integer argument inside **Overload**, no matching method is found.
- However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**.
- Of course, if **test(int)** had been defined, it would have been called instead. *Java will employ its automatic type conversions only if no exact match is found*.

Practice question

- Create a Geometry class and overload calculateArea method for square, circle, and rectangle.

Overloading Constructors

- In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
class Box {
  double width;
  double height;
  double depth;

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

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

class OverloadCons {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    double vol;

    // 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);
  }
}
```

Example: Constructor Overloading

The output produced by this program is shown here:

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

Advantages of Method Overloading

- Method overloading improves the Readability and reusability of the program.
- Method overloading reduces the complexity of the program.
- Using method overloading, programmers can perform a task efficiently and effectively.
- Using method overloading, it is possible to access methods performing related functions with slightly different arguments and types.

Inheritance

Inheritance

- In the Java language, classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.
- The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.
- A class that is inherited is called a *superclass* (or base class or a parent class). The class that does the inheriting (or derived from base class) is called a *subclass* (also a derived class, extended class, or child class).
- Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

Inheritance

- To inherit a class, use the **extends** keyword in subclass definition.
- **Syntax:**

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

- You can only inherit **one superclass** for any subclass.
- Java **does not support the concept of multiple inheritance**
- **Multilevel Inheritance:** You can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass .
- No class can be a superclass of itself.
- **Private members** of a class are not inherited.
- Every class in java implicitly extends **java.lang.Object** class. So Object class is at the top level of inheritance hierarchy in java.

Example of Inheritance

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

SKILLS UNIVERSITY

Example: private members are not inherited

```
class A {
  int i; // default access
  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);
  }
}
```

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);
  }
}
```

A Superclass Variable Can Reference a Subclass Object

- It is important to understand that it is the *type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.*
- That is, when a reference to a subclass object is assigned to a superclass reference variable*, you will have access only to those members of the object defined by the superclass*. This is why **plainbox** can't access **weight**

Using super keyword

- In the previous examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box**.
  - Duplicate code
  - Instance variable of superclass is accessed directly by subclass
- There will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private)
- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of **super**.
- **super** has two uses:
  - to call superclass' constructor.
  - used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- 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.
- super( ) must always be the first statement executed inside a subclass' constructor.
- When a subclass calls **super()**, it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy.

```
class Bicycle {
    private int gear;
    private int speed;
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }
    public void applyBrake(int decrement){
        speed -= decrement;
    }

    public void speedUp(int increment)
    {       speed += increment;
    }
    public String toString(){
        return ("No of gears are " + gear + "\n"
                + "speed of bicycle is " + speed);
    }
}
```

```
class MountainBike extends Bicycle {
    public int seatHeight;
    public MountainBike(int gear, int speed, int startHeight)
    {
        super(gear, speed);
        seatHeight = startHeight;
    }
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    public String toString()
    {
        return (super.toString() + "\nseat height is "
                + seatHeight);
    }
}
```

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

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

Example of Inheritance: Demo of super()

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

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

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

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

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

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

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }

  // constructor used when cube is created
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}

class DemoSuper {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    BoxWeight mybox3 = new BoxWeight(); // default
    BoxWeight mycube = new BoxWeight(3, 2);
    BoxWeight myclone = new BoxWeight(mybox1);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
    System.out.println();

    vol = mybox3.volume();
    System.out.println("Volume of mybox3 is " + vol);
    System.out.println("Weight of mybox3 is " + mybox3.weight);
    System.out.println();

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

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

Example of Inheritance: Demo of super()

- 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
```

- Notice that **super( )** is passed an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**.
- As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.
-

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

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

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

Slide 1: Previous Example of Inheritance: Demo of super()

```
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

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

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }

  // constructor used when cube is created
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}
```

ITM SKILLS UNIVERSITY

Slide 2: Previous Example of Inheritance: Demo of super()

```
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();
  }
}
```

Slide 3: Previous Example of Inheritance: Demo of super()

Practice question: Inheritance

- Create a class Animal with instance variables:
    - boolean vegetarian
    - String food
    - int numOfLegs
  - Create a no-argument constructor and parameterized constructor. (Use “this”)
  - Create getters and setters
  - Create a toString() method for animal class
  - Create a subclass Cat with instance variable:
    - String color
    - Create a no-argument constructor and parameterized constructor which has all four parameters (Use this and super)
    - Create a toString() method for Cat class
  - Create a subclass Cow with instance variable:
    - String breed
    - Create a no-argument constructor and parameterized constructor which has all four parameters. (Use this and super)
    - Create a toString() method for Cow class

Using super to overcome name hiding

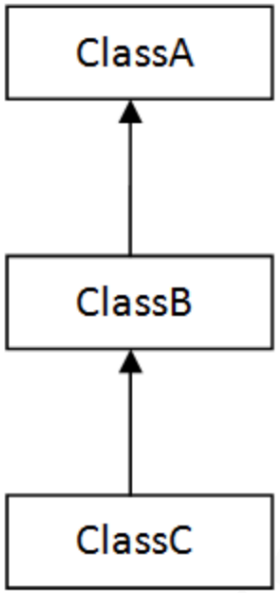
- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.
- Syntax: **super.*member;***
- Here, *member* can be either a method or an instance variable.
- **This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.**

Using super to overcome name hiding

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

Multilevel Inheritance

- 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**



Example 1: Multilevel Inheritance

```
class Animal{
    void eat(){   System.out.println("eating…");   }
}

class Dog extends Animal{
    void bark(){   System.out.println("barking…");    }
}

class BabyDog extends Dog{
    void weep(){    System.out.println("weeping…");     }
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:
weeping…
barking...
eating...

```
class Box {
  private double width;
  private double height;
  private double depth;
  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

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

// Add weight.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

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

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }

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

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

  // construct clone of an object
  Shipment(Shipment ob) { // pass object to constructor
    super(ob);
    cost = ob.cost;
  }

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

  // default constructor
  Shipment() {
    super();
    cost = -1;
  }

  // constructor used when cube is created
  Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
  }
}

class DemoShipment {
  public static void main(String args[]) {
    Shipment shipment1 =
              new Shipment(10, 20, 15, 10, 3.41);
    Shipment shipment2 =
              new Shipment(2, 3, 4, 0.76, 1.28);

    double vol;

    vol = shipment1.volume();
    System.out.println("Volume of shipment1 is " + vol);
    System.out.println("Weight of shipment1 is "
                       + shipment1.weight);
    System.out.println("Shipping cost: $" + shipment1.cost);
    System.out.println();

    vol = shipment2.volume();
    System.out.println("Volume of shipment2 is " + vol);
    System.out.println("Weight of shipment2 is "
                       + shipment2.weight);
    System.out.println("Shipping cost: $" + shipment2.cost);
  }
}
```

Example of Multilevel Inheritance

Multilevel Inheritance

- **super( )** always refers to the constructor in the closest superclass.
  - Example: The **super( )** in **Shipment** calls the constructor in **BoxWeight**. The **super( )** in **BoxWeight** calls the constructor in **Box**.
- In a class hierarchy, if a superclass constructor requires arguments, then all subclasses must pass those arguments "up the line." This is true whether or not a subclass needs arguments of its own.
- In the preceding program, the entire class hierarchy, including **Box**, **BoxWeight**, and **Shipment**, is shown all in one file. In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.

Order of execution of Constructors in Multilevel inheritance

- In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.
- Further, since **super( )** must be the first statement executed in a subclass' constructor. If **super( )** is not used, then compiler inserts super() to call parameterless constructor of superclass.

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

Order of execution of Constructors in Multilevel inheritance

The output from this program is shown here:

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

Inheritance

- Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.
- A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

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 its 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.

Example of Method Overriding

```
// 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
   }
 }
```

Method Overriding and use of super

- When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**.
- If you wish to access the superclass version of an overridden method, you can do so by 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);
  }
}
```

Method Overriding

- 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.

```
// Methods with differing type signatures are overloaded - not
// overridden.
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
  }
}
```

ITM SKILLS UNIVERSITY

Method overriding verses method overloading

Method Overriding: Dynamic Method Dispatch (Run time Polymorphism)

- Method overriding forms the basis for one of Java's most powerful concepts: *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. It 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.

Dynamic Method Dispatch

- When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
// 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
  }
}
```

Example of Dynamic method dispatch

Example: Applying Method Overriding

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

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

Run time Polymorphism: Method Overriding

- 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.*
- Key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization.
- Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.
- 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.

Abstraction: Abstract Classes and Interface

Abstract methods and 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.

Abstract methods and Abstract Classes

- **I**n this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. 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. A subclass must override abstract methods of super class.
- To declare an abstract method, use this general form:

  abstract *type name*(*parameter-list*);

Abstract methods and Abstract Classes

- Any class that contains one or more abstract methods must also be declared abstract.
- To declare an abstract class, 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 declared abstract itself.

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

Example of abstract class and abstract method

Abstract methods and Abstract Classes

- *Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.*
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

```
// Using abstract methods and classes.
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // area is now an abstract method
  abstract double area();
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}

class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // illegal now
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref; // this is OK, no object is created

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

Example of abstract class and abstract method