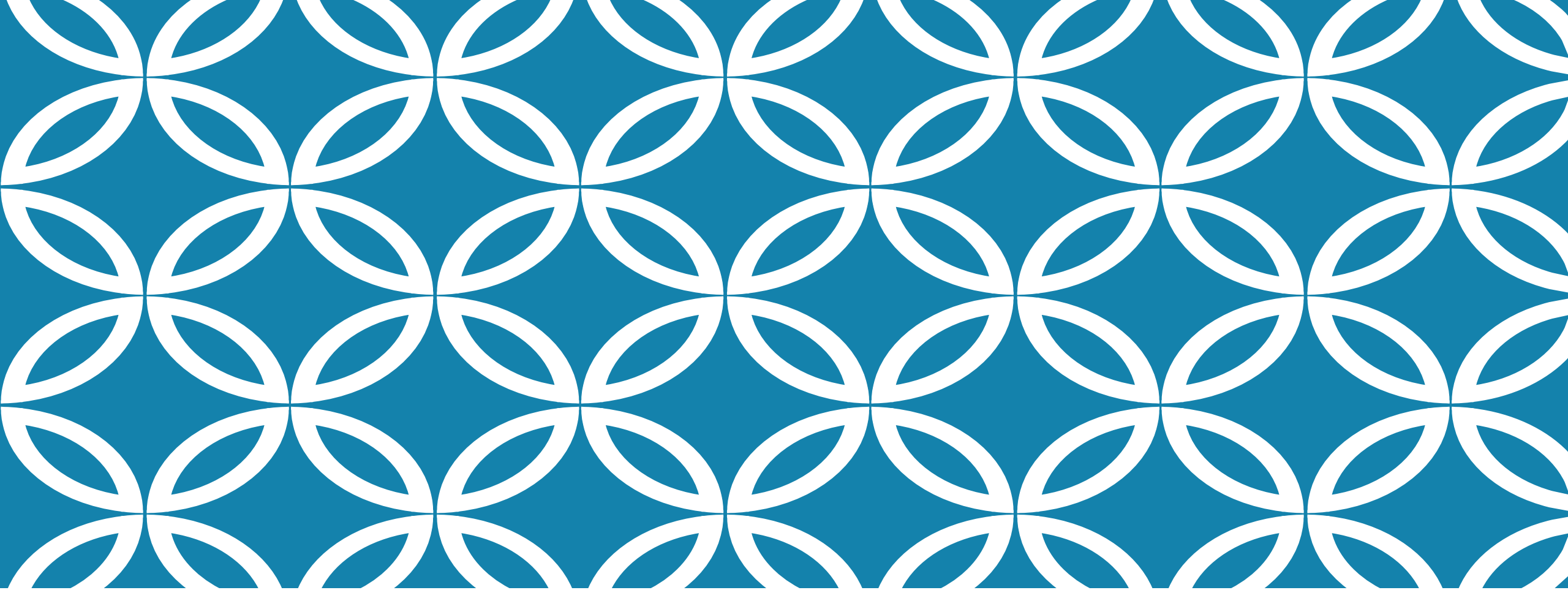


Chapter 8
Inheritance

Inheritance Basics

Inheritance is a mechanism wherein a new class is derived from an existing class. In Java, classes may inherit or acquire the properties and methods of other classes. A class derived from another class is called a subclass, whereas the class from which a subclass is derived is called a superclass. The keyword “extends” is used to derive a subclass from the superclass, as illustrated by the following example

The general form of a **class** declaration that inherits a superclass is shown here:

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

```
// A simple example of inheritance.

// Create a superclass.
class A {
  int i, j;

  void showij() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

```
// Create a subclass by extending class A.
class B extends A {
  int k;

  void showk() {
    System.out.println("k: " + k);
  }
  void sum() {
    System.out.println("i+j+k: " + (i+j+k));
  }
}
class SimpleInheritance {
  public static void main(String args[]) {
    A superOb = new A();
    B subOb = new B();

    // The superclass may be used by itself.
    superOb.i = 10;
    superOb.j = 20;
    System.out.println("Contents of superOb: ");
    superOb.showij();
    System.out.println();

    /* The subclass has access to all public members of
       its superclass. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Contents of subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();

    System.out.println("Sum of i, j and k in subOb:");
    subOb.sum();
  }
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

Class A	Variables	i
		J
Class B	Variables	K

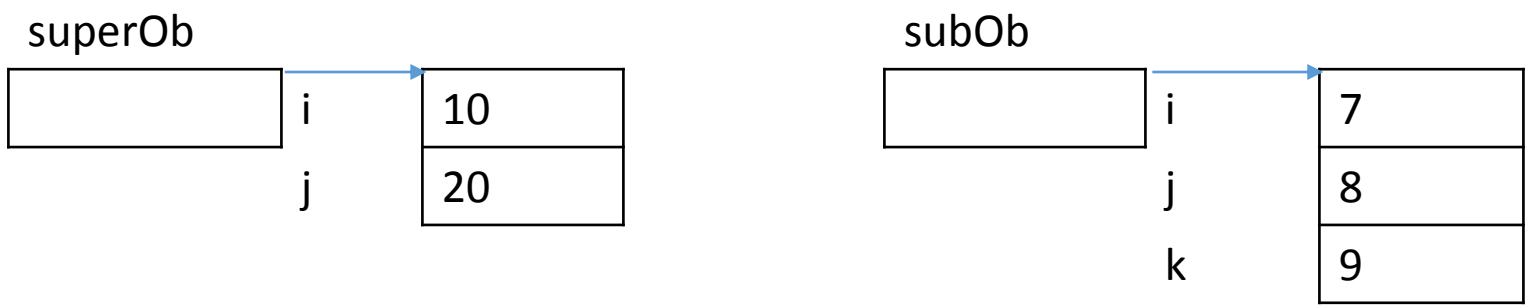
A superOb = new A();

B subOb = new B();

Class A	Variables	i
		J
Class B	Variables	K

A superOb = new A();

B subOb = new B();



Member Access and Inheritance:

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

```
/* In a class hierarchy, private members remain
   private to their class.

   This program contains an error and will not
   compile.
*/

// Create a superclass.
class A {
  int i; // public by default
  private int j; // private to A

  void setij(int x, int y) {
    i = x;
    j = y;
  }
}

// A's j is not accessible here.
class B extends A {
  int total;
  void sum() {
    total = i + j; // ERROR, j is not accessible here
  }
}

class Access {
  public static void main(String args[]) {
    B subOb = new B();

    subOb.setij(10, 12);

    subOb.sum();
    System.out.println("Total is " + subOb.total);
  }
}
```

Another Example

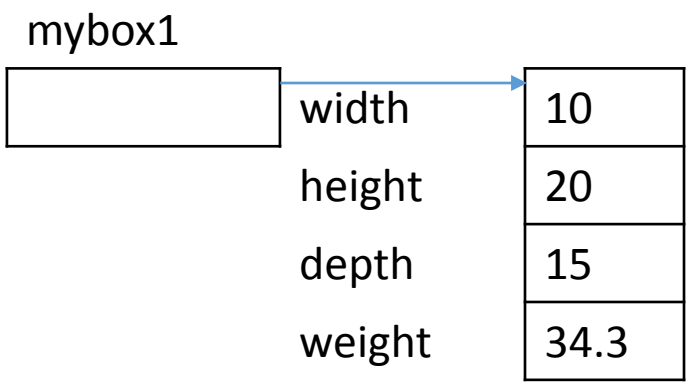
```
// This program uses inheritance to extend Box.
class Box {
  double width;
  double height;
  double depth;

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

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

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

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

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

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

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

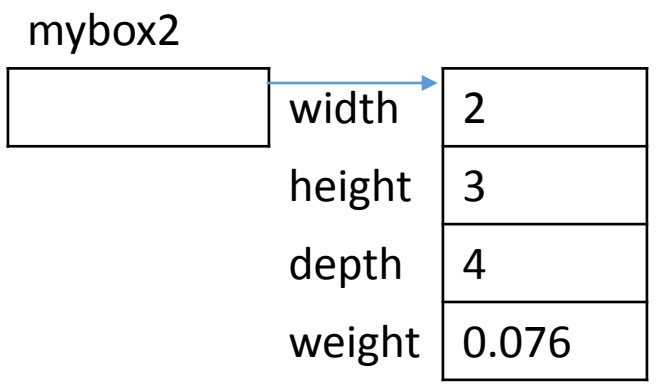
```
class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;

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

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

Output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

Box	Variables	width
		height
		depth
BoxWeight	Variables	weight





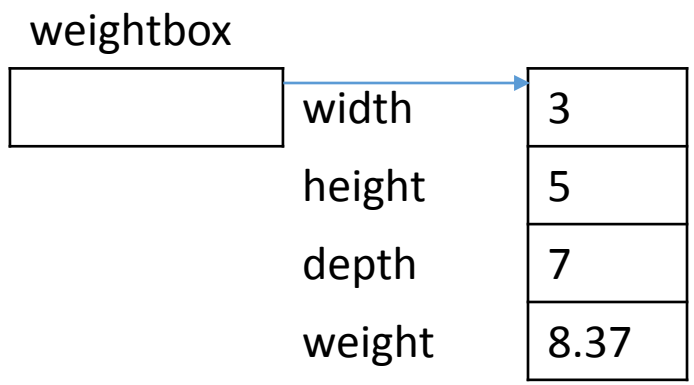
A Superclass Variable Can Reference a Subclass Object

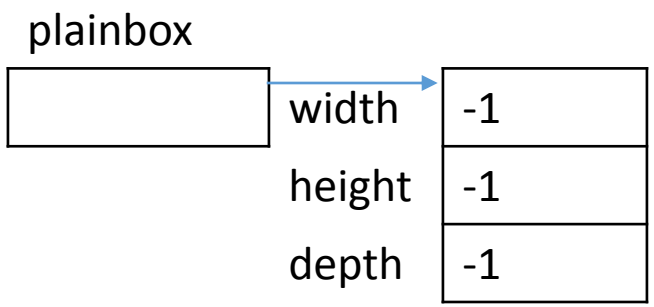
A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
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);
  }
}
```

Box	Variables	width
		height
		depth
BoxWeight	Variables	weight





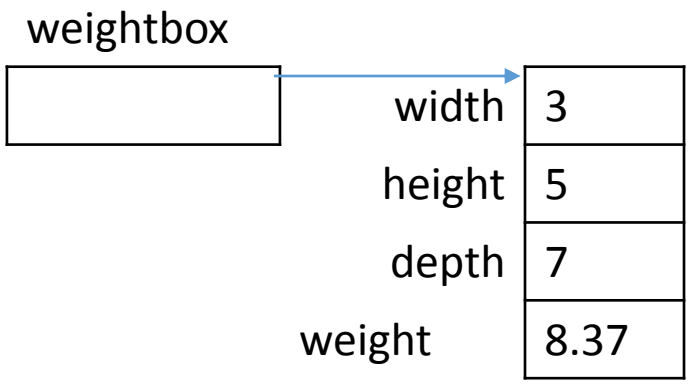
Box	Variables	width
		height
		depth
BoxWeight	Variables	weight

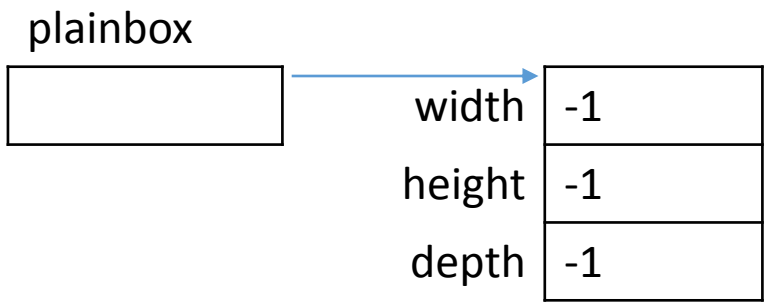
Now

plainbox=weightbox;

vol=plainbox.volume();

What happens?

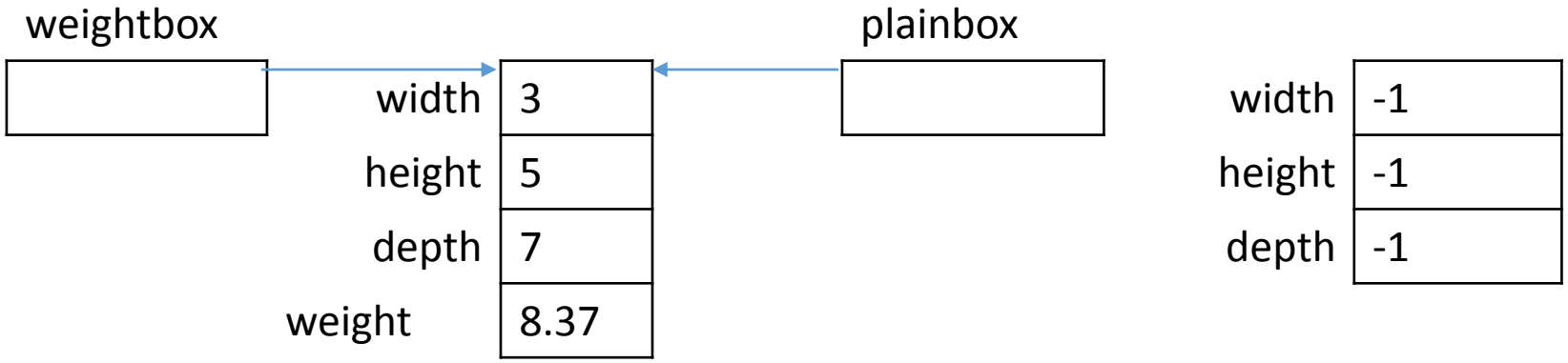




Box	Variables	width
		height
		depth
BoxWeight	Variables	weight

Now

plainbox=weightbox;
vol=plainbox.volume();



Example 1

Box.java

```
// 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 10*width * height * depth;
  }
}
```

Box.java

```
// Here, Box is extened 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;
  }
  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

RefDemo.java

```
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);
  }
}
```

Example 2 SubclassReference.java

```
class Parent {
  public void parentPrint() {
    System.out.println("parent print called");
  }
  public static void staticMethod() {
    System.out.println("parent static method called");
  }
}
public class SubclassReference extends Parent {
  public void parentPrint() {
    System.out.println("child print called");
  }
  public static void staticMethod() {
    System.out.println("child static method called");
  }
  public static void main(String[] args) {
    //SubclassReference invalid = new Parent();//Type
mismatch: cannot convert from Parent to
SubclassReference
    Parent obj = new SubclassReference();
    obj.parentPrint();    //method of subclass would
execute as subclass object at runtime.
    obj.staticMethod();    //method of superclass would
execute as reference of superclass.
    Parent obj1 = new Parent();
    obj1.parentPrint();    //method of superclass would
execute as superclass object at runtime.
    obj1.staticMethod();   //method of superclass would
execute as reference of superclass.
    SubclassReference obj3 = new SubclassReference();
    obj3.parentPrint();    //method of subclass would
execute as subclass object at runtime.
    obj3.staticMethod();   //method of subclass would
execute as reference of subclass.
  }
}
```

```
D:\OOPE\temp>java SubclassReference
child print called
parent static method called
parent print called
parent static method called
child print called
child static method called
```

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*);

- *arg-list* specifies any arguments needed by the constructor in the superclass.
- **super( )** must always be the first statement executed inside a subclass' constructor.

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
  double weight; // weight of box

  // initialize width, height, and depth using super()
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }
}
```

A Second Use for super:

The second form of **super** refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable.

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

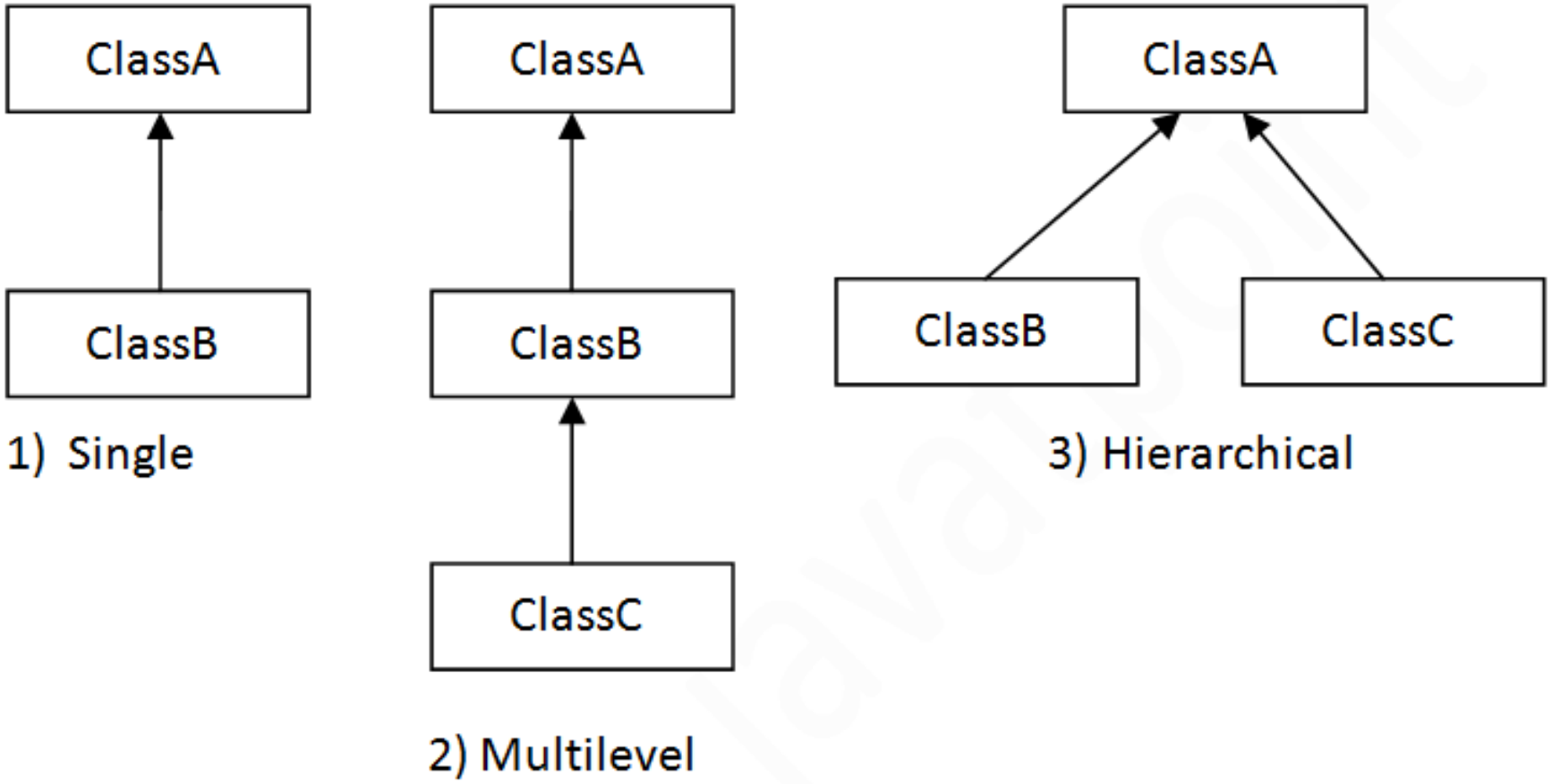
```
// Using super to overcome name hiding.
class A {
  int i;
}

// 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();
  }
}
```

Output:

```
i in superclass: 1
i in subclass: 2
```

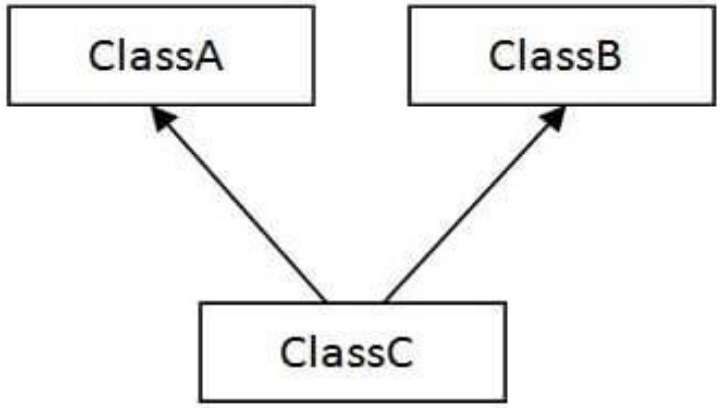
Types of Inheritance



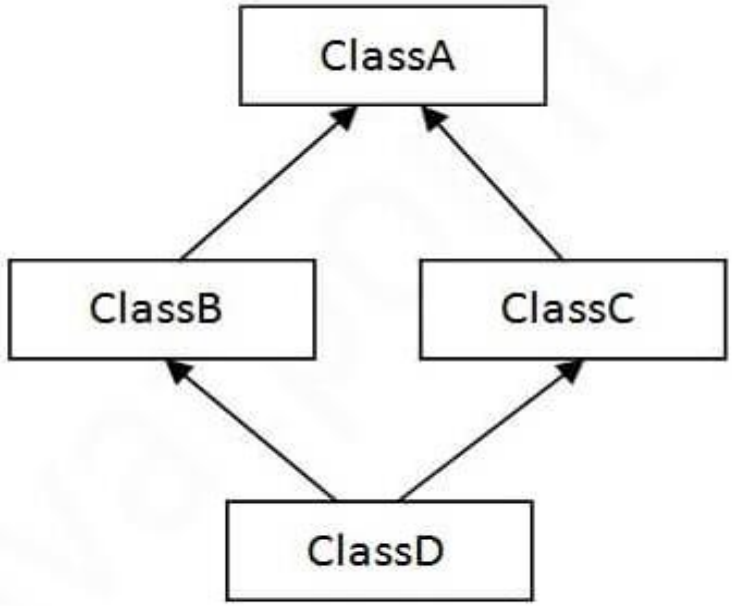
Source: https://www.javatpoint.com/inheritance-in-java

Types of Inheritance

In java programming, multiple and hybrid inheritance is supported through interface only.



4) Multiple



5) Hybrid

Source: https://www.javatpoint.com/inheritance-in-java

Creating a Multilevel Hierarchy:

you can build hierarchies that contain as many layers of inheritance. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**.

```
// Extend BoxWeight to include shipping costs.

// Start with Box.
class Box {
  private double width;
  private double height;
  private double depth;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

Hierarchical Inheritance Example

```
class Animal{
  void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
  void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
 void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
      Cat c=new Cat();
      c.meow();
      c.eat();
      //c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Multiple inheritance: not supported in java

```
class A{
        void msg(){System.out.println("Hello");}
}
class B{
        void msg(){System.out.println("Welcome");}
}

class C extends A,B{//suppose if it were

 public static void main(String args[]){
       C obj=new C();
       obj.msg();//Now which msg() method would be invoked?
}
}
```

When Constructors Are Called

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- Since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used.
- If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed.

```
// Demonstrate when constructors are
called.
// Create a super class.
class A {
  A() {
    System.out.println(“
          Inside A's constructor.");
  }
}

// Create a subclass by extending class A.
class B extends A {
  B() {
    System.out.println(“
        Inside B's constructor.");
  }
}
```

```
// Create another subclass by extending
B.
class C extends B {
  C() {
    System.out.println(“
        Inside C's constructor.");
  }
}

class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

The output of this program is shown here:

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

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
  }
}
```

The output produced by this program is shown here:

```
k: 3
```

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);
  }
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

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

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

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

```
class A {
  int i, j;

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

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

// Create a subclass by extending class A.
class B extends A {
  int k;

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

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

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show("This is k: "); // this calls show() in B
    subOb.show(); // this calls show() in A
  }
}
```

Output:

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

Dynamic Method Dispatch:

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.

```
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 {
void callme() {
// override callme()
    System.out.println("Inside C's callme
                               method");
  }
}
```

```
class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A

    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

The output produced by this program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

Applying Method Overriding:

```
// Using run-time polymorphism.
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());
  }
}
```

The output produced by this program is shown here:

```
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:

- 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();
  }
}
```

```
// 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 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:

Using final to Prevent Overriding:

Methods declared as **final** cannot be overridden.

```
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 with Inheritance:

Using final to Prevent Overriding:

- Methods declared as **final** can sometimes provide a performance enhancement:
  - The compiler is free to *inline* calls to them because it "knows" they will not be overridden by a subclass.
  - When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.
  - Inlining is only an option with **final** methods.
  - Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*.
  - However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

Using final to Prevent Inheritance:

- Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- It is illegal to declare a class as both **abstract** and **final.** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {
  // ...
}

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

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.
- Reference variable of type **Object** can refer to an object of any other class.
- Since arrays are implemented as classes, a variable of type **Object** can also refer to any array.
- The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**. You may override the others.
- The **equals( )** method compares the two objects. It returns **true** if the objects are equivalent, and **false** otherwise.

Object defines the following methods, which means that they are available in every object.

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.

References

- https://www.javatpoint.com/inheritance-in-java

Thank You