Inheritance

Basics

- Inheritance defines parent-child relationships.
  - Base (or super) class, and derived (or sub-) class
- Child inherits all functionality of its parent
  - Typically add new methods and member variables.
- One of the basic features of all OO languages.

Basic Program

```
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));
} }
public class basics {
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.Reverse is not possible. */
subOb.i = 7;subOb.j = 8;subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}}
```

Output:

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

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

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

Member Access and Inheritance

```
// 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);
}
}
//final and static variables cannot be inherited
```

Note:

Sub object can access private members using member function of super class. (possible)

Member Access and Inheritance

```
// Create a superclass.
class A {
int i; // public by default
protected int j; // private to outside
    world
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;
}}
```

```
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);
}
}
```

Output:

22

Practical Example

```
class Box {
double width;double height;double depth;
Box(Box ob)
{ // pass object to constructor
width = ob.width;height = ob.height;
depth = ob.depth; }
Box(double w, double h, double d) {
width = w;height = h;depth = d;}
Box() {width = -1; height = -1; depth = -1;}
Box(double len)
{width = height = depth = len; }
// compute and return volume
double volume() {
return width * height * depth; } }
class BoxWeight extends Box {
double weight; // weight of box
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
```

- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects
- It can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification.
- For example, the following class inherits **Box** and adds a color attribute.

```
class ColorBox extends Box {
int color; // color of box
ColorBox(double w, double h, double d, int c)
{
width = w;
height = h;
depth = d;
color = c;
}
}
```

Using super

- subclass to directly access or initialize super class variables on its own.

```
class Box {
private double width;
private double height;
private double depth;
Box(Box ob) {
width = ob.width;height = ob.height;
depth = ob.depth;}
Box(double w, double h, double d) {
width = w;height = h;depth = d;}
Box() {
width = -1; height = -1depth = -1;}
Box(double len) {
width = height = depth = len; }
double volume() {
return width * height * depth;} }
```

```
class BoxWeight extends Box {
double weight;
BoxWeight(BoxWeight ob) {super(ob);weight = ob.
weight;}
BoxWeight(double w, double h, double d, double m)
 {super(w, h, d); // call superclass constructor
weight = m; }
BoxWeight() {super();weight = -1;}
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);
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();
vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
} }
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

A Second Use for super

- 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
```

Creating a Multilevel Hierarchy

```
class Box {
private double width;private double height;
private double depth;
 Box(Box ob) {
width = ob.width;height = ob.height;
depth = ob.depth;}
Box(double w, double h, double d) {
width = w;height = h;depth = d;}
Box() {width = -1height = -1; depth = -1; }
Box(double len) {width = height = depth
=len;}
double volume() {return width * height *
depth;
}}
class BoxWeight extends Box {
double weight;
BoxWeight(BoxWeight ob) {super(ob);
weight = ob.weight;}
BoxWeight(double w, double h, double d,
double m) {
super(w, h, d); weight = m;}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);weight = m;}}
// Add shipping costs
class Shipment extends BoxWeight {
double cost;
Shipment(Shipment ob) {super(ob);cost = ob.
cost; }
// constructor when all parameters are
specified
Shipment(double w, double h, double d,
double m, double c) {
super(w, h, d, m); cost = c;}
// default constructor
Shipment() {super();cost = -1;}
// constructor used when cube is created
Shipment(double len, double m, double c) {
super(len, m);cost = c;}}
class DemoShipment {
public static void main(String args[]) {
```

```
Shipment shipment1 =
new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 =
new Shipment(2, 3, 4, 0.76, 1.28);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is "
+ shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println();
vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is "
+ shipment2.weight);
System.out.println("Shipping cost: $" + shipment2.cost);
}}
```

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41
Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

When Constructors Are Called

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

Output:

Inside A's constructor.

Inside B's constructor.

Inside C's constructor.

```
class A {
A(){System.out.println("Inside a's def constructor.");}
A(int a) {
System.out.println("Inside A's constructor."); } }
class B extends A {
    B(){System.out.println("Inside B's def constructor.");}
B(int a) {
System.out.println("Inside B's constructor."); } }
class C extends B {
C(int a) {
System.out.println("Inside C's constructor."); } }
```

```
class cons {
public static void main(String args[]) {
    C c = new C(10);
    }
    }
```

Output:

Inside a's def constructor.

Inside B's def 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.

Example

```
class AA { int i, j;
AA(int a, int b) {i = a;j = b;}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class BB extends AA {
    int k;
    BB(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[]) {
BB subOb = new BB(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

Output:

K:3

Access the superclass version of an overridden function

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

Output:

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

Reason-Why Overridden Methods?

- 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.
- understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization

Methods with differing type signatures are overloaded – not overridden.

```
class x{int i, j;
x(int a, int b) {i = a;j = b;}
// display i and j
void show() {
System.out.println("i and j: " + i + " "
+ j); } }
class y extends x { int k;
y(int a, int b, int c) {
super(a, b);k = c; }
//overload show()
void show(String msg) {
System.out.println(msg + k); } }
```

```
class sample {
public static void main(String arg
 {
y subOb = new y(1, 2, 3);
subOb.show("This is k: "); // this
show() in y
subOb.show(); // this calls show(
}
}
```

Output:

This is k: 3

i and j: 1 2

Dynamic Method Dispatch

- 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.
- This is how Java implements run-time polymorphism.
- 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.
- *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.

Example

```
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 dis
{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(reverse is not
possible)
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}}
```

Output:

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

```
class Box {
double width;double height;double depth;
Box(Box ob) {
width = ob.width;height = ob.height;
depth = ob.depth; }
Box(double w, double h, double d) {
width = w;height = h;depth = d;}
Box() {width = -1; height = -1; depth = -1;}
Box(double len)
{width = height = depth = len; }
double volume() {
return width * height * depth; } }
class BoxWeight extends Box {
double weight; // weight of box
BoxWeight(double w, double h, double d,
double m) {
width = w;height = h;depth = d;weight = m; }}
class in{
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 System.out.println("
Volume of plainbox is " + vol);
/* The following statement is invalid because
plainboxdoes not define a weight member. */
// System.out.println("Weight of plainbox is " +
plainbox.weight);} }
```

Output:

```
Volume of weightbox is 105.0
Weight of weightbox is 8.37
Volume of plainbox is 105.0
```

Note:

Superclass Variable/reference Can Refer a Subclass Object. If method overrirding not done, super class reference/variable cant access

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.");} }
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.")
;
class apply {
public static void main(String args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
figref = f;
System.out.println("Area is " + figref.area());
    }       }
```

Output:

```
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
Area for Figure is undefined.
Area is 0.0
```

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.
- 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.
- Java's solution to this problem is the *abstract method.*
- Syntax:

abstract type name(parameter-list)
;

- 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.
- 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.(no objects)
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.
- And, all subclasses of must override **the function**. Otherwise, we will receive a compile-time error.

Example

```
abstract class AA {
abstract void callme();
// concrete methods are still allowed
in abstract classes
void callmetoo() {
System.out.println("This is a concrete
method.");
}  }
class BB extends AA {
void callme() {
System.out.println("B's
implementation of callme.");
} }
```

```
class abst {
public static void
main(String args[]) {
BB b = new BB();
b.callme();
b.callmetoo();
} }
```

Output:

B's implementation of callme.

This is a concrete method.

Program

```
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 fig {
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())
;
}    }
```

The Object Class

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

Object class

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.

void wait( ) void wait(long milliseconds)

void wait(long milliseconds,int nanoseconds)

Using final to Prevent Overriding

```
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!");}}
```

- If 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

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

Thank You