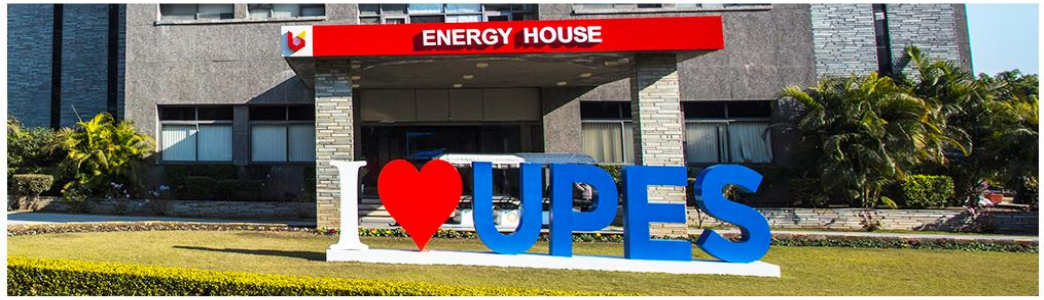

UPES

UNIVERSITY WITH A PURPOSE

Object Oriented Programming

UPES
UNIVERSITY WITH A PURPOSE

Creating a Multilevel Hierarchy

- We 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**.
- 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**.

UPES
UNIVERSITY WITH A PURPOSE

Creating a Multilevel Hierarchy

```
// Extend BoxWeight to include shipping costs.
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 given
Box(double w, double h, double d) {
width = w; height = h; depth = d; }
// constructor used when no dimensions given
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;
}
```

UPES
UNIVERSITY WITH A PURPOSE

Creating a Multilevel Hierarchy

```
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
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);  }  }
```

UPES
UNIVERSITY WITH A PURPOSE

When Constructors Are Executed

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed?

For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor executed before **B**'s, or vice versa?

- The answer is that 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, this order is the same whether or not **super( )** is used.
- If **super( )** is not used, then the default or parameter less constructor of each superclass will be executed.

When Constructors Are Executed

```
// Demonstrate when constructors are executed.
// 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 from this program is shown here:

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

UPES
UNIVERSITY WITH A PURPOSE

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.

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
}
}
```

The output produced by this program is shown here:
k: 3

Note: 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**.

UPES
UNIVERSITY WITH A PURPOSE

Method Overriding

- If we wish to access the superclass version of an overridden method, you can do so by using **super**.
- For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
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 we 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( )**.

UPES
UNIVERSITY WITH A PURPOSE

Method Overloading

- 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
}
}
```

The output produced by this program is shown here:

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

Note: The version of **show( )** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of **show( )** in **B** simply overloads the version of **show( )** in **A**.

UPES
UNIVERSITY WITH A PURPOSE

References

Schildt, H. (2014). *Java: the complete reference*. McGraw-Hill Education Group.

UPES
UNIVERSITY WITH A PURPOSE