OBJECT ORIENTED PROGRAMMING

Affefah Qureshi

Department of Computer Science

Iqra University, Islamabad Campus.

WHEN CONSTRUCTORS ARE CALLED

- In what order are the constructors for the classes that make up an inheritance hierarchy called?
- In a class hierarchy, constructors are called 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 explicitly used.
  - **If super() is not explicitly used , then the default constructor of each superclass will be executed.**

WHEN CONSTRUCTORS ARE CALLED EXAMPLE

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

Output:

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.
- If we wish to access the superclass version of an overridden function inside subclass, we can do so by using super.
- Method overriding (name hiding) 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.**

OVERRIDING VS. OVERLOADING EXAMPLE

Output:

This is k: 3
i and j: 1 2

```
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
  }
}
```

DYNAMIC METHOD DISPATCH

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

DYNAMIC METHOD DISPATCH EXAMPLE

```
// Dynamic Method Dispatch
class A {
   void callme() {
    System.out.println("Inside A's
   callme method");
  }
}
class B extends A {  // override
   callme()
  void callme() {
   System.out.println("Inside B's
   callme method");
  }
}
class C extends A { // override
   callme()
  void callme() {
   System.out.println("Inside C's
   callme method");
  }
}
```

```
class Dispatch {
  public static void main(String args[])
   {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r;      // obtain a reference of type
    A
    r = a; // r refers to an A object
    r.callme(); // calls A's version of
    callme
    r = b; // r refers to a B object
    r.callme(); // calls B's version of
    callme
    r = c; // r refers to a C object
    r.callme(); // calls C's version of
    callme
  }
}
```

Output:

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

DYNAMIC METHOD DISPATCH EXAMPLE

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

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
```