Lecture – 19

Polymorphism and Dynamic Binding

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*
- Polymorphism is derived from 2 Greek words: poly(many) and morph(forms): meaning **many forms**
- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features
- A subclass is a specialization of its superclass
- Every instance of a subclass is also an instance of its superclass
- Example: Dog **ISA** Animal but Animal **ISA** Dog is **incorrect**

Polymorphism

- An object of a subclass can be used wherever its superclass object is used known as *Polymorphism*
- In simple terms, polymorphism means that a variable of a supertype can refer to a subtype object.

```
class A{}
class B extends A{}
A a=new B();
```

```
Animal myAnimal = new Animal();
   Animal myDog = new Dog();
   Animal myCat = new Cat();
```

Example

```
public class Animal {
    public void makeSound() {
        System.out.println("Some animal sounds.");
    }
}

class Dog extends Animal {
    public void makeSound() {
        System.out.println("Woof woof!");
    }
}

class Cat extends Animal {
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

```
class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();
    Animal myDog = new Dog();
    Animal myCat = new Cat();

    myAnimal.makeSound(); // Output: Some animal sounds.
    myDog.makeSound(); // Output: Woof woof!
    myCat.makeSound(); // Output: Meow!
  }
}
```

‘dog’ not only possesses the characteristics of a **Dog** but also those of an **‘Animal’**

Types of Polymorphism

- There are two types of polymorphism in Java:
  - Compile-time polymorphism (Method Overloading)
  - Runtime polymorphism (Method Overriding)

Compile-time polymorphism/Static Polymorphism/Early Binding

- The method to be called is determined at compile-time based on the number, type, and order of the arguments passed to the method.

```
public class MathUtils {
    public static int add(int x, int y) {
        return x + y;
    }
    public static double add(double x, double y) {
        return x + y;
    }
    public static int add(int x, int y, int z) {
        return x + y + z;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        int sum1 = MathUtils.add(2, 3);
        double sum2 = MathUtils.add(2.5, 3.5);
        int sum3 = MathUtils.add(2, 3, 4);
    }
}
```

Runtime polymorphism/Dynamic Binding/Dynamic Method Dispatch

- A method can be implemented in several classes along the inheritance chain.
- The JVM decides which method is invoked at runtime

```
Object obj = new GeometricObject();
System.out.println(obj.toString());
```

- Which `toString()` method is invoked by `obj`?
- Declared type and Actual type?

Runtime polymorphism/Dynamic Binding/Dynamic Method Dispatch

- **Declared Type:** The type that declares a variable is called the variable’s declared type. E.g. `Object obj`
  - This variable `obj` can hold
    - either `null` value: `obj = null`
    - Reference to an instance of declared type or its subtype:
      `obj = new Object()` OR `obj = new GeometricObject`
- **Actual Type:** The type of the variable is the actual class for the object referenced by the variable
- `obj` actual type is `GeometricObject()`
- Because `obj` references an object created using `new GeometricObject()`
- Which `toString()` method is invoked by `obj` is determined by `obj`’s actual type. This is known as dynamic binding

Dynamic Binding

Dynamic binding works as follows:

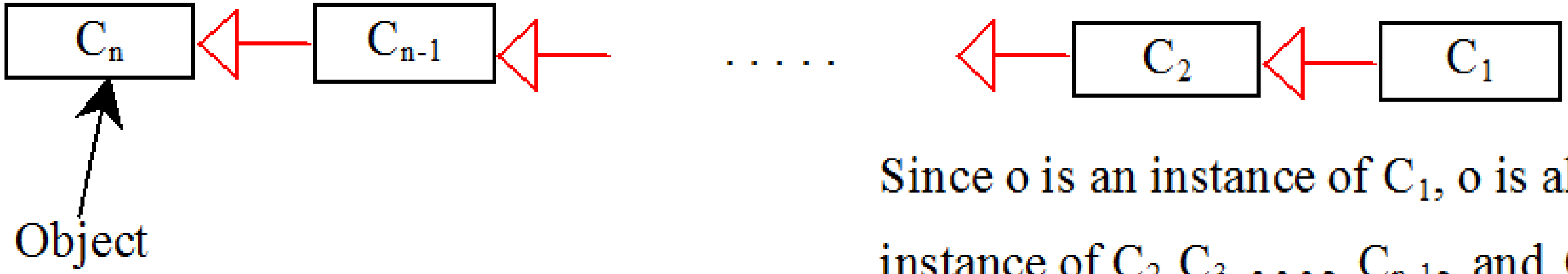
Suppose an object obj is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n,(where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n).

C_n is the most **general class**, and C_1 is the most **specific class**.

In Java, C_n is the **Object** class.

If obj invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found.

Once an implementation is found, the search stops and the first-found implementation is invoked.

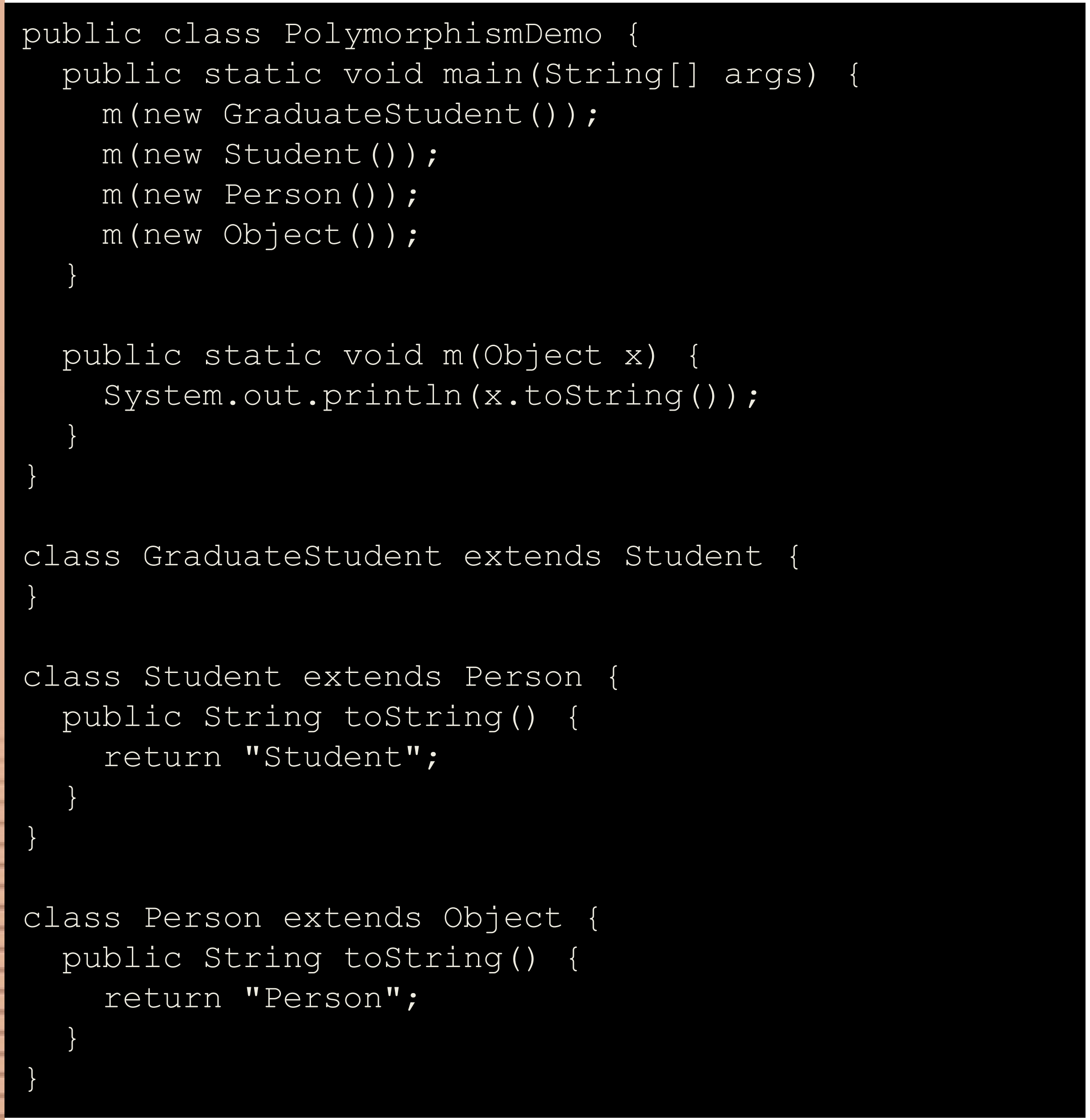


Since o is an instance of C_1, o is also an instance of C_2, C_3, . . . , C_{n-1}, and C_n

Method Matching vs. Binding

- Matching a method signature and binding a method implementation are two issues.
- **Method Matching:** The **declared type** of the reference variable decides which method to match at compile time. The compiler finds a matching method according to the parameter type, number of parameters, and order of the parameters at compile time [**Overloading**]
- **Method Binding:** A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime, decided by the **actual type** of the variable. [**Overriding**]

Generic Programming



```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

- Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming.
- If a method’s parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter’s subclasses (e.g., Student or String).
- When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

Polymorphism – Example

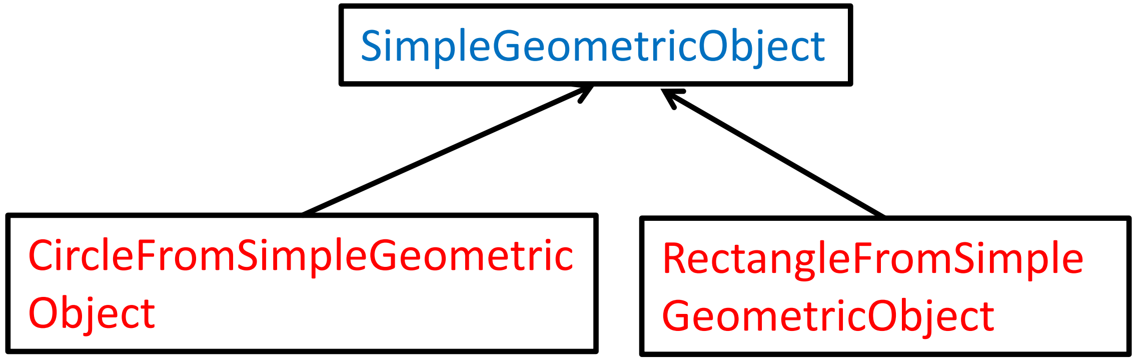
```
public class PolymorphismDemo {
    public static void main(String[] args) {
            displayObject(new CircleFromSimpleGeometricObject (1, "red", false));
           displayObject(new RectangleFromSimpleGeometricObject(1, 1, "black", true));
     }
     public static void displayObject(SimpleGeometricObject object) {
           System.out.println("Created on " + object.getDateCreated() +
                                    ". Color is " + object.getColor());
     }
```

Object of Subclass

Object of Superclass

```
Created on Mon Mar 09 19:25:20 EDT 2011. Color is red
Created on Mon Mar 09 19:25:20 EDT 2011. Color is black
```

You can always pass an instance of a subclass to a parameter of its superclass type


SimpleGeometricObject
CircleFromSimpleGeometric Object
RectangleFromSimple GeometricObject

Show the Output

```
public class Test {
public static void main(String[] args) {
  new Person().printPerson();
  new Student().printPerson();
}
class Student extends Person {
   @Override
   public String getInfo() {
      return "Student";
    }
}
class Person {
   public String getInfo() {
      return "Person";
   }
   public void printPerson() {
      System.out.println(getInfo());
   }
}
```

```
public class Test {
public static void main(String[] args) {
  new Person().printPerson();
  new Student().printPerson();
}
class Student extends Person {
   private String getInfo() {
      return "Student";
    }
}
class Person {
   private String getInfo() {
      return "Person";
   }
   public void printPerson() {
      System.out.println(getInfo());
   }
}
```

Show the Output

```
public class Test {
    public static void main(String[] args) {
          A a = new A(3);
    }
}
class A extends B {
    public A(int t) {
       System.out.println("A's constructor is invoked");
    }
}
class B {
  public B() {
      System.out.println("B's constructor is invoked");
   }
 }
```

Show the output

```
public class Test {
     public static void main(String[] args) {
             new A();
             new B();
     }
}
class A {
   int i = 7;
   public A(){
      setI(20);
      System.out.println("i from A is "+ i);
   }
   public void setI(int i){
      this.i = 2 * i;
   }
}
class B extends A{
   public B(){
      System.out.println("i from B is "+ i);
   }
   public void setI(int i){
      this.i = 3 * i;
   }
}
```