Java Programming 1

Polymorphism
Part 1

Contents

- Introduction
- Examples
- The Mechanics of Polymorphism
  - Static and Dynamic Binding
  - Casting Objects
- The `instanceof` Operator

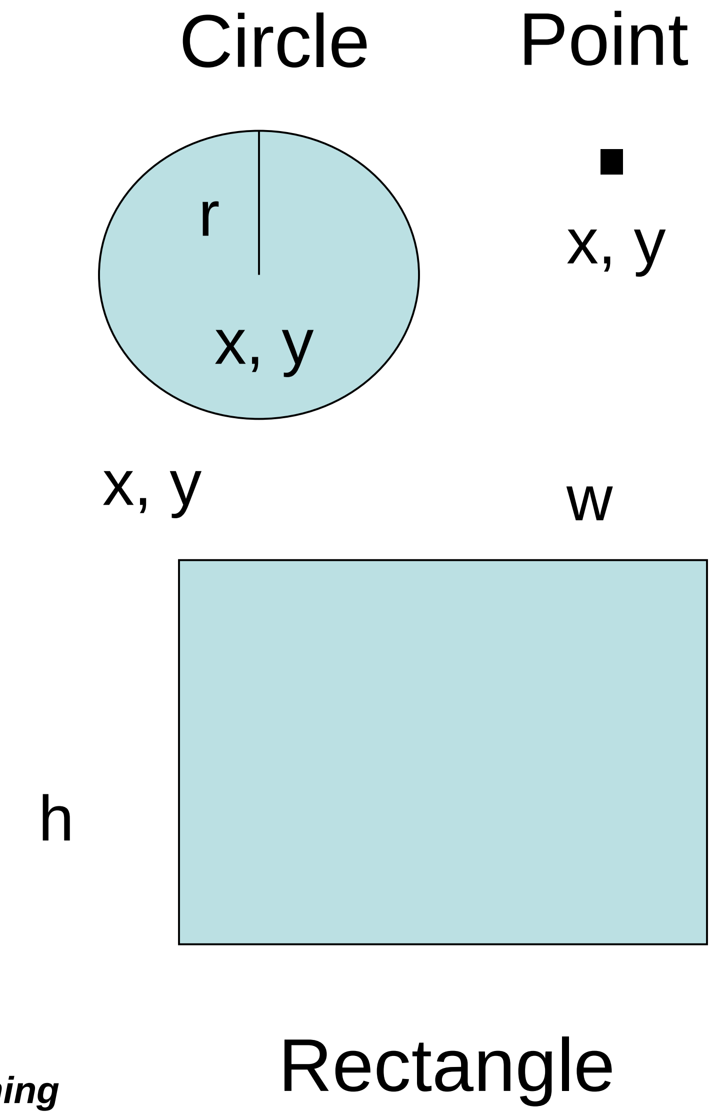
- Reference
  - Bruce Eckel, Thinking in Java, Chapter 7 http://www.faqs.org/docs/think_java/TIJ309.htm

Introduction

- Polymorphism is one of the most important concepts of object-oriented programming.
- In general, it means the occurrence of something in multiple forms.
- In programming, polymorphism is the ability for same code to be used with several different types of objects and behave differently depending on the actual type of object used.

Example: Drawing Shapes

- Write a program to maintain a list of shapes created by the user, and print the shapes when needed.
- The shapes needed in the application are:
  - points
  - lines
  - rectangles
  - circles
  - etc...



In Conventional Programs

- When you use the C language you should:
  - Define the *struct* data type to store parameters of the shape
    - One field is for the type of the shape: point, circle, etc.
  - Write the functions to draw each shape (separate for each shape).
  - Check the type of the shape first to select the right function to draw.

```
typedef struct shape {
    int typeS; // point = 0, circle = 1,
                   // line = 2, rectangle = 3
    int x, y // parameters of the shape
. . .
} ;
shape varShape;
. . .
if (varShape.typeS == 1) then
            DrawCircle(varShape);
else if (varShape.typeS == 3) then
            DrawRectangle(varShape);
else if (varShape.typeS == 0) then
            DrawPoint(varShape);
else if(varShape.typeS == 2) then
            DrawLine(varShape);
```

Using Polymorphism

- You need only to write:
  - varShape.Draw()
- How to do this?

Example 1

```
class Person {
 private String name;
public Person(String name) {
  this.name = name;
}
public String introduction() {
   return "My name is " + name + ".";
 }
}
class Student extends Person {
 private String id;
 public Student(String name, String id){
   super(name);
   this.id = id;
 }
 public String getID() {  return id; }
 public String introduction() {
   return "I am a student. " +
   super.introduction() + " My ID is "+ id + ".";
 }
}
```

```
public class PolymorphismDemo1 {
  public static void main(String[] args) {
       Student s =
          new Student("Saito","s115333");
        Person p = s;
        System.out.println(s.introduction());
        System.out.println(p.introduction());
   }
}
```

- Output of this program:
  - I am a student. My name is Saito. My ID is s115333 .
  - I am a student. My name is Saito. My ID is s115333.

Comments on the Previous Slide

- Consider two simple classes:
  - Person
  - Student (this one is a subclass of Person)
- Why do they print the same output?
    - System.out.println(s.introduction());
    - System.out.println(p.introduction());
- Because the same message (*introduction()*) is sent to the same object, in this case Student.
- Why is the object the same (Student)?

Recall: Primitive Assignment

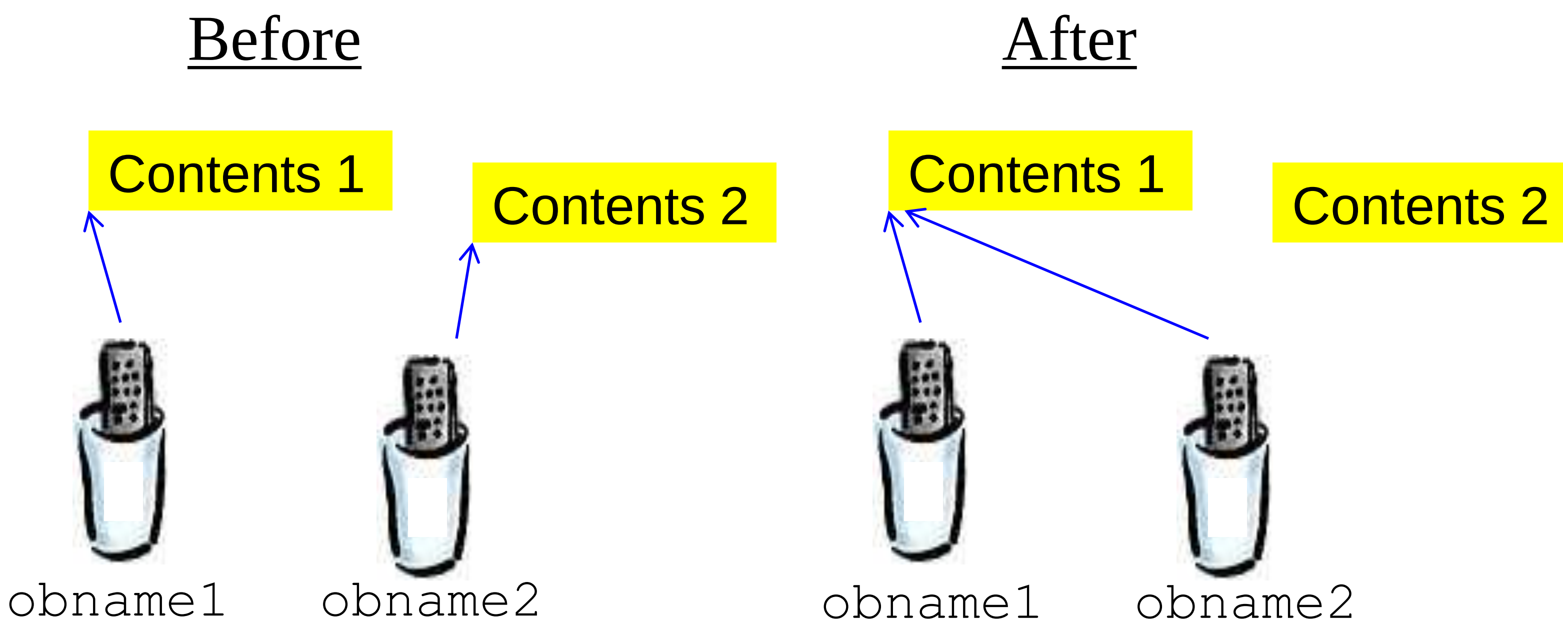
- The act of assignment takes a copy of a value and stores it in a variable.
- For primitive types:

```
num2 = num1;
```

Recall: Reference Assignment

- For object references, the reference (address, the location) is copied:

```
obname2 = obname1;
```



Example 2

```
class Person {
  private String name;
 public Person(String name) {
   this.name = name;
 }
 public String introduction() {
    return "My name is " + name + ".";
  }
}
class Student extends Person {
  private String id;
  public Student(String name, String id){
    super(name);
    this.id = id;
  }
 public String getID() {  return id; }
 public String introduction() {
    return "I am a student. " +
    super.introduction() + " My ID is "+ id + ".";
  }
}
```

```
public class PolymorphismDemo2 {
  public static void main(String[] args) {
    m(new Student("Saito", "s115333"));
    m(new Person("Tanaka"));
  }
  public static void m(Person x) {
    System.out.println(x.introduction());
  }
}
```

- Output of this program:
  - I am a student. My name is Saito. My ID is s115333 .
  - My name is Tanaka.

Comments on the Previous Slide

- Method *m* takes a parameter of the *Person* type. An object of a subtype can be used wherever its supertype value is required.
  - This feature is known as *polymorphism*.
- When the method *m(Person x)* is executed, the argument x's *introduction* method is invoked. *x* may be an instance of *Student* or *Person*. Classes *Student* and *Person* have their own implementation of the *introduction* method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.
  - This capability is known as *dynamic binding*.

Comments on Examples 1 and 2

- Example 1: The Java compiler cannot decide at compilation time which method must be called when the program is running:
  - Introduction() of the Person class or of the Student class
    - System.out.println(s.introduction());
    - System.out.println(p.introduction());
- Example 2: The same situation
    - System.out.println(x.introduction());
- A decision is made when the program is running.

Static and Dynamic Binding

- Non-polymorphic methods (static methods) are “bound”
  - at compile time
  - called *early binding* or static binding.
- Polymorphic methods are “bound”
  - at run time
  - called *late binding* or dynamic binding (also called dynamic dispatch).
- Alternate views of polymorphism:
  - One objects sends a message to another object without caring about the type of the receiving object.
  - The receiving object responds to a message appropriately for its type.
- Java methods are polymorphic by default
  - *static* or *final* (*private* methods are implicitly *final*) are bound at compile time.

Note: Polymorphic Methods

- Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in a superclass is redefined in a subclass, the method defined in the superclass is hidden.

Note: Polymorphic Methods

```
class Parent {
    public static void myStaticMethod()  {
        System.out.println("A");
    }
    public void myInstanceMethod()   {
        System.out.println("B");
    }
} // End of the Parent class
public class Child extends Parent {
    public static void myStaticMethod()  {
        System.out.println("C");
    }
    public void myInstanceMethod()   {
        System.out.println("D");
    }
```

```
public static void main(String[] args)  {
        Parent o1 = new Parent();
        Parent o2 = new Child();
        Child  o3 = new Child();

        Parent.myStaticMethod(); // A
        Child.myStaticMethod();   // C
        o1.myStaticMethod();       // A
        o1.myInstanceMethod();  // B
        o2.myStaticMethod();       // A
        o2.myInstanceMethod();  // D
        o3.myStaticMethod();       // C
        o3.myInstanceMethod();  // D
        myStaticMethod();           // C
        myInstanceMethod();// Compiler Error
    }  // End of main method
} // End of the Child class
```

Comments on the Previous Slide

- Notice that *o2.myStaticMethod* invokes *Parent.myStaticMethod()*. If this method were truly overridden, we should have invoked *Child.myStaticMethod*, but we didn't. Rather, when you invoke a static method, even if you invoke it on an instance, you really invoke the method associated with the "compile-time type" of the variable. In this case, the compile-time type of *o2* is *Parent*. Therefore, we invoke *Parent.myStaticMethod()*.
- However, when we execute the line *o2.myInstanceMethod()*, we really invoke the method *Child.myInstanceMethod()*. That's because, unlike static methods, instance methods CAN be overridden. In such a case, we invoke the method associated with the run-time type of the object. Even though the compile-time type of *o2* is *Parent*, the run-time type (the type of the object *o2* references) is *Child*. Therefore, we invoke *Child.myInstanceMethod* rather than *Parent.myInstanceMethod()*.

Comments on the Previous Example

- Why do the following lines produce the results as shown in comments:

```
public class Child extends Parent {
 . . .
          public static void main(String[] args)  {
 . . .
                    myStaticMethod();          // C
                    myInstanceMethod();      // Compiler Error
          } // end of the mail method
}             // end of the Child class
```

- We invoke *mystaticMethod()* from the *static main* method, which is a member of the *Child* class. We invoke a static method from a static context. This is correct.
- We tried to invoke *myInstanceMethod()* without a specification of the object. This invocation is done from the *static main* method, which is a member the *Child* class. This generates a compile-time error.

Method Matching vs. Binding

- Matching a method signature and binding a method implementation are two issues.
  - The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.
  - A method may be implemented in several subclasses.
- The Java Virtual Machine dynamically binds the definition of the method at runtime.

Dynamic Binding in Java

- We can conceptually think of the dynamic binding mechanism as follows: Suppose an object *o* 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.
- That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the *Object* class.
- If *o* 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.

Casting Objects

- You have already used the casting operator to convert variables of one primitive type to another.
- *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In Example 2, the statement

  m(new Student (“Saito”, “s115333”));

  assigns the object new Student() to a parameter of the Person type. This statement is equivalent to:

```
Person o =  new Student(“Saito”, “s115333”); // Implicit casting
                                             // It is called up-casting
m(o);
```

- The above statement is known as implicit casting. It is legal because an instance of Student is automatically an instance of Person.

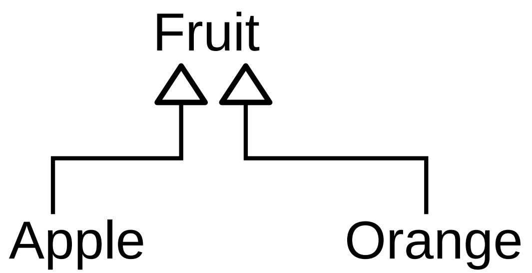
Why is Casting Necessary?

- Consider the following (see Example 1, Slide 7):
  - Person o = new Student(….);
  - Student b = o;
- A compilation error would occur. Why is the statement *Person   o = new Student()* valid and the statement *Student b = o* not?
- This is because a *Student* object is always an instance of *Person*, but a *Person* is not necessarily an instance of *Student*. Even though you can see that *o* is really a Student object, the compiler is not so clever to know it.
- To tell the compiler that *o* is a *Student* object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types:

  Student b = (Student)o; // Explicit casting. It is called down-casting

TIP

- To help understand casting, you may also consider the analogy of fruit, apple, and orange with the *Fruit* class as the superclass for *Apple* and *Orange*.
- An apple is a fruit, so you can always safely assign an instance of *Apple* to a variable for *Fruit* (implicit casting):
  - Fruit f = new Apple();
- However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of *Fruit* to a variable of *Apple*.




```
Fruit f;
Apple a = new Apple();
Orange o = new Orange();
f = a; // implicit casting, up-casting
f = o; // implicit casting, up-casting
if (f instanceof Apple) {
   a = (Apple)f;    // explicit casting
                    // down-casting
}
```

The `instanceof` Operator

- Often you will get into a situation in which you need to rediscover the exact type of the object so you can access the extended methods of that type (see Example 2, slide 11):

```
Person p =  new Student("Saito","s115333");
System.out.println(p.getID()); // Compile-time error:
        // There is no the getID method in the Person class.
```

- Use the `instanceof` operator to test whether an object is an instance of a class:

```
Person p =  new Student("Saito","s115333");
if (p instanceof Student) {
  System.out.println("Student ID: " + ((Student)p).getID());
}
```

Casting-Dot Operator Precedence

- The casting operator has lower precedence than the "." (dot) operator:
  - ((Student)p).getID()
- Without the parentheses the cast is associated with the method (*getID* in our example) and attempts to change its return type:
  - (Student)p.getID()

Summary of Polymorphism, Part 1

- Polymorphism means “multiple forms.”
  - In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different definitions of the dynamically bound methods.
- Polymorphism is a feature that cannot be viewed in isolation (like a *switch* statement can, for example), but instead works only in concert, as part of a “big picture” of class relationships.