20251024 Friday

- Test 2 format:
  morning section 1:   6 prefer online test,   4 prefer local
  afternoon section 2: 2 prefer online test, 13 prefer local
- So, I will give you freedom to do it either way.

Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype object.

A class defines a type. A type defined by a subclass is called a **subtype**, and a type defined by its superclass is called a **supertype**. Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

PolymorphismDemo

Polymorphism, Dynamic Binding and Generic Programming (1 of 2)

```
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";
  }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

DynamicBindingDemo

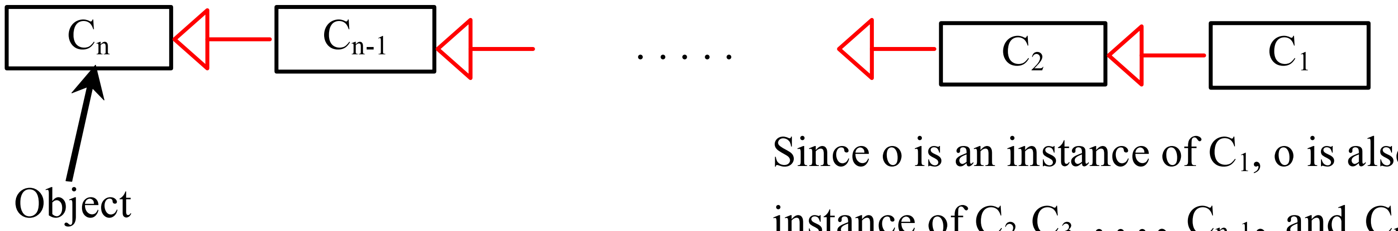
Polymorphism, Dynamic Binding and Generic Programming (2 of 2)

An object of a subtype can be used wherever its supertype value is required. This feature is known as **polymorphism**.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as **dynamic binding**.

Dynamic Binding

Dynamic binding works 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.



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. 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 implementation of the method at runtime.

Generic Programming (1 of 2)

```
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.toSt
ring());
 }
}
```

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

Generic Programming (2 of 2)

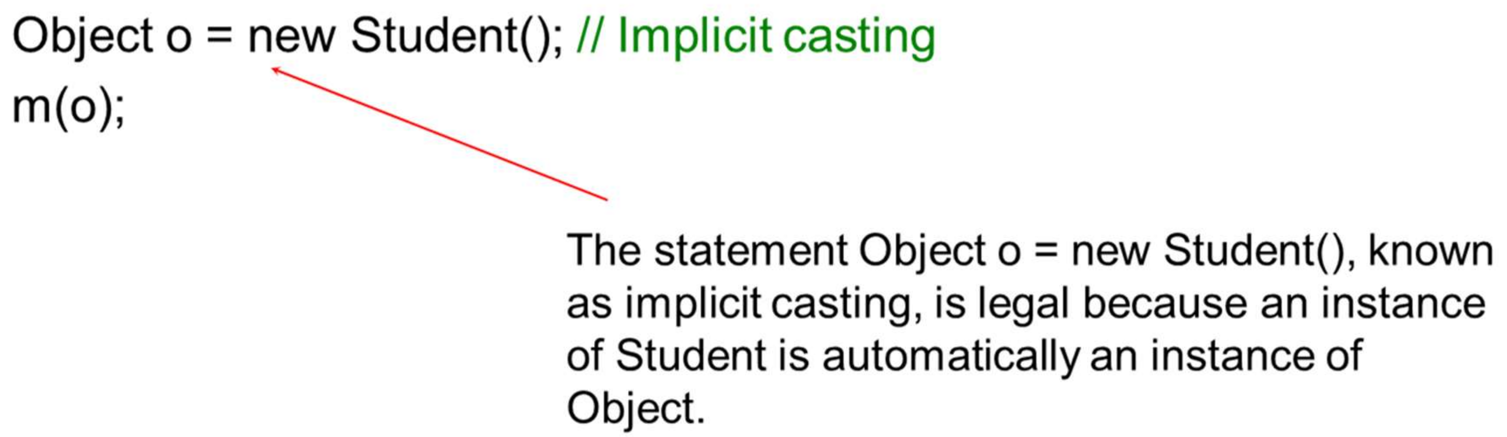
Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method’s parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter’s subclasses (e.g., Student or String). When an object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

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 the preceding section, the statement

```
    m(new Student());
```

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



```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

Student b = o;

A compile error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? This is because a Student object is always an instance of Object, but an Object 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. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

Student b = (Student)o; // Explicit casting

Casting From Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The `instanceof` Operator

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

```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of
  Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

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. 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.

Example: Demonstrating Polymorphism and Casting

This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method to display the objects. The displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

CastingDemo

The `equals` Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj){
  return this == obj;
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
   return radius ==((Circle)o).radius;
   }
   else
    return false;
}
```

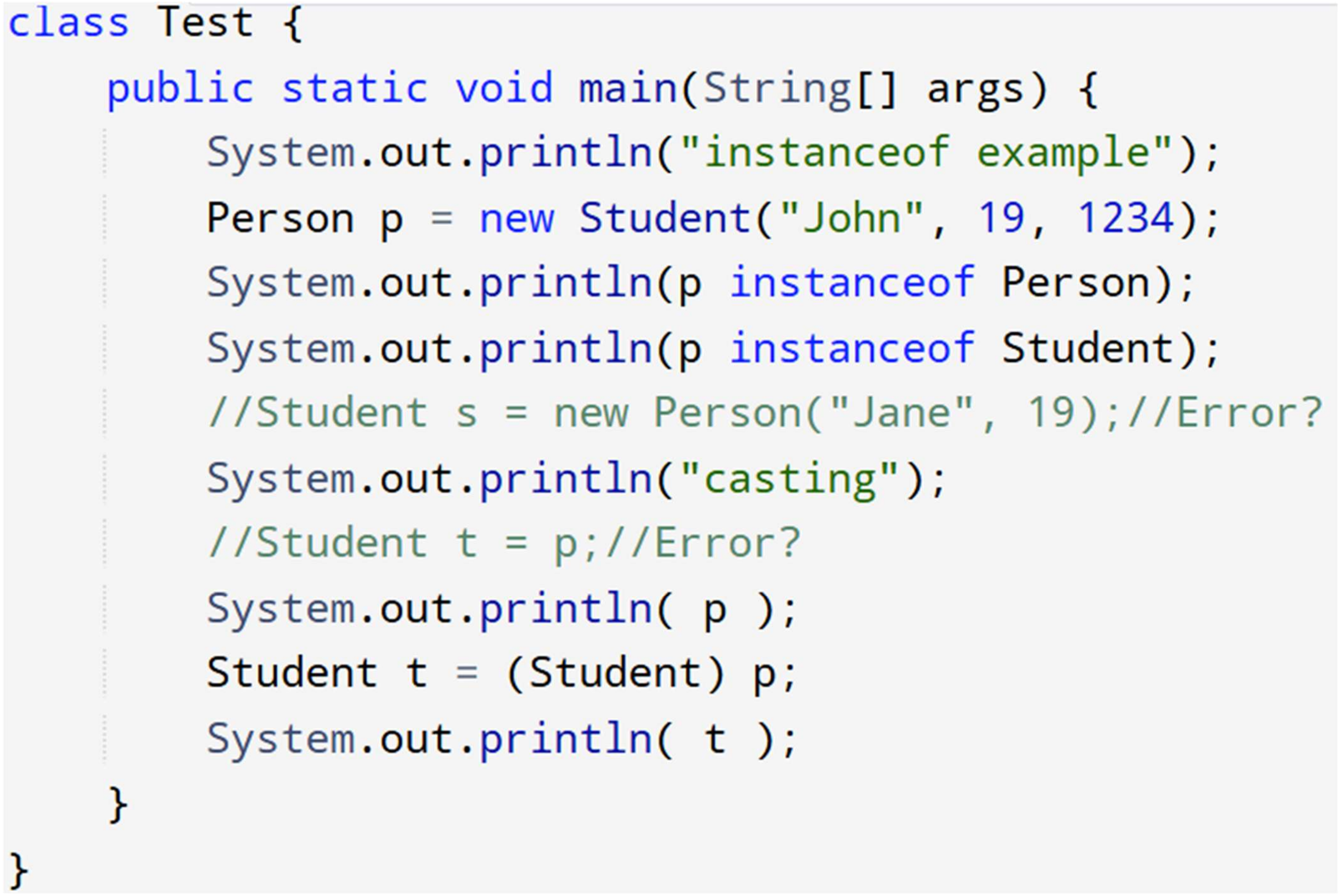
Note (3 of 4)

The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The equals method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

Practice

- “Casting” and “instanceof”

```
class Person{
    String name;
    int age;
    Person(String n, int a){ name=n; age=a; }
}
class Student extends Person{
    int sID;
    Student(String n, int a, int id){
        super(n,a);
        sID=id;
    }
}
```



```java
class Test {
    public static void main(String[] args) {
        System.out.println("instanceof example");
        Person p = new Student("John", 19, 1234);
        System.out.println(p instanceof Person);
        System.out.println(p instanceof Student);
        //Student s = new Person("Jane", 19);//Error?
        System.out.println("casting");
        //Student t = p;//Error?
        System.out.println( p );
        Student t = (Student) p;
        System.out.println( t );
    }
}
```

- Still have time, please practice the “equals” and “Generic Programming”.
- Read textbook, and practice.