

Inheritance and Polymorphism

Polymorphism

Dynamic Binding

Casting Objects and the instanceof Operator

The Object's equals Method

Polymorphism

- The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism.
- Polymorphism means that a variable of a supertype (superclass) can refer to a subtype (subclass) object.
- We can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

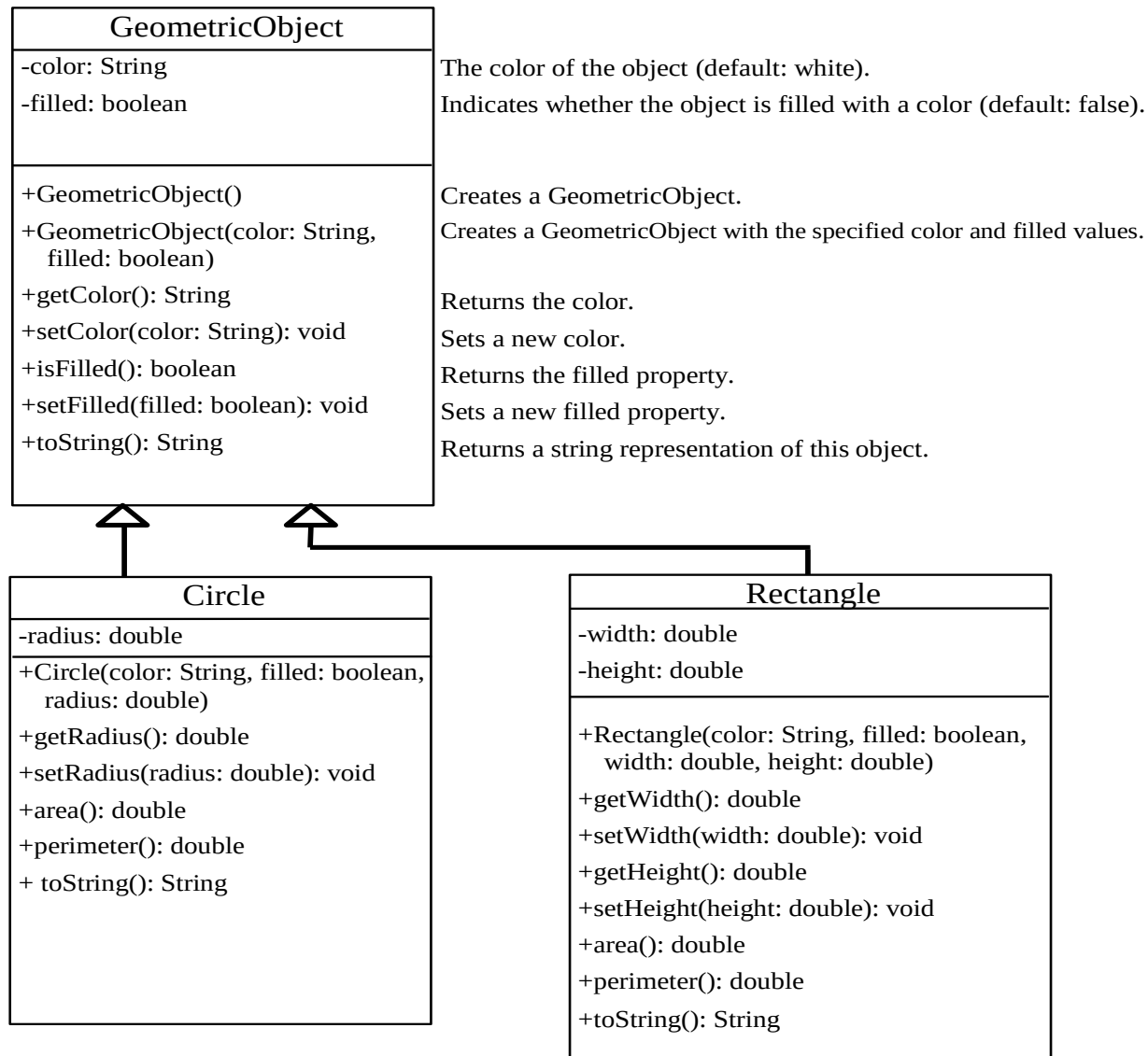
Polymorphism

- 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, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle.
- Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.

Polymorphism

- Let us revisit the example of the superclass GeometricObject and its two subclasses Circle and Rectangle (UML in next slide).
- Let us assume we want to write a program in which we want to create 40 objects of type GeometricObject, 50 of type Circle, and 30 of type Rectangle. How do we represent them in the program? Do we create 120 ($40 + 50 + 30$) references? Of course not. When we have many objects we use arrays. So now let us assume we create three arrays: one of type GeometricObject, one of type Circle, and one of type Rectangle to store the objects.

UML Diagram revisited



Polymorphism

We can create the three arrays as follows:

```
GeometricObject [] g = new GeometricObject[40];
```

```
Circle [] c = new Circle[50];
```

```
Rectangle [] r = new Rectangle[30];
```

Now assume we wrote the necessary java statements to store the 120 objects in the three arrays based on their type. How do we display the information of these objects using the toString method?

Polymorphism

To display the information of the objects we use the following three loops:

```
for (int i = 0; i < g.length; i++) // can use i < 40
    System.out.print(g[i].toString()); // calling toString of GeometricObject
```

```
for (int i = 0; i < c.length; i++) // can use i < 50
    System.out.print(c[i].toString()); // calling toString of Circle
```

```
for (int i = 0; i < r.length; i++) // can use i < 30
    System.out.print(r[i].toString()); // calling toString of Rectangle
```

Polymorphism

- Now imagine that the class `GeometricObject` has 8 more subclasses other than `Circle` and `Rectangle` such as `Triangle`, `Parallelogram`, `Rhombus`, `Octagon`, etc. Does that mean we create 11 arrays to store the objects from different types and write 11 loops to display the information of all objects?
- Using Polymorphism we can create one array instead of 11 to store all objects and use one loop only to display the information of all objects since as mentioned before polymorphism means that a variable of a supertype (superclass) can refer to a subtype (subclass) object.

Polymorphism

Polymorphism is a Greek word. Poly means many and morphe means form. So polymorphism means "having many forms".

In OOP polymorphism is translated as follows:

A reference from a superclass can refer to objects from different types provided that these types are from subclasses of the superclass. For example, if we have a `GeometricObject` reference `g1` (`GeometricObject g1;`) then we can write any of the following statements:

```
g1 = new GeometricObject("Red", true); // g1 refers to a GeometricObject
```

```
g1 = new Circle("Red", true, 1); // g1 refers to a Circle object
```

```
g1 = new Rectangle("Red", true, 5, 6); // g1 refers to a Rectangle object
```

Hence, `g1` can refer to different types, i.e. it has many forms.

Polymorphism

So now we can create only one array (instead of 11) of type `GeometricObject`:

```
GeometricObject [] g = new GeometricObject[120];
```

And then we can store inside it the 120 objects of type `GeometricObject`, `Circle`, and `Rectangle` as follows:

```
g[0] = new GeometricObject("Red", true); // g[0] refers to a GeometricObject
g[1] = new Circle("Red", true, 1); // g[1] refers to a Circle object
g[2] = new Rectangle("Red", true, 5, 6); // g[2] refers to a Rectangle object
...
g[119] = new _____; // g[119] can refer to an object of any of the 3 types
```

In general `g[i]` can refer to an object of any of the 3 types so `g[i]` is called a **polymorphic reference**. The array `g` itself is called a **polymorphic array**.

Note: Of course we can fill the array using a loop.

Polymorphism

Now to display the information of all the objects in the array we write only one loop as follows:

```
for (int i = 0; i < g.length; i++)  
    System.out.print(g[i].toString());
```

The question now is which toString method will be called above? Is the toString of GeometricObject, that of Circle, or that of Rectangle? The answer to that is found in the next section (11.8 Dynamic Binding).

Dynamic Binding

We look again at the loop below while assuming we have the array shown in slide 10:

```
for (int i = 0; i < g.length; i++)  
    System.out.print(g[i].toString());
```

Although the reference `g[i]` is of type `GeometricObject`, the call `g[i].toString()` does not necessarily mean a call to the `toString()` of `GeometricObject`.

When `i = 0`, `g[i]` refers to a `GeometricObject` therefore the `toString` of class `GeometricObject` will be called.

When `i = 1`, `g[i]` refers to a `Circle` object therefore the `toString` of class `Circle` will be called.

When `i = 2`, `g[i]` refers to a `Rectangle` object therefore the `toString` of class `Rectangle` will be called.

Dynamic Binding

We conclude that, it is not the reference type that determines which `toString` will be called. It is rather the object type that will determine it. This is known as **dynamic binding**. Binding is linking a method call to a specific method. It is dynamic because it happens during runtime. It means that the method to be executed for a specific method call is not known until run-time.

Therefore, the method call `g[i].toString( )` has many forms (can be a call to different methods having the same name and found in a superclass and its subclasses). This call is called a **polymorphic call**.

So we can say that Polymorphism is the use of **polymorphic references** and **polymorphic calls**.

The main advantage of polymorphism is that it makes the code simpler. A single variable name can be used to store variables of multiple data types. Then we can treat these different types as if they are one type.

Dynamic Binding

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

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 and the instanceof Operator

One object reference can be typecast (converted) into another object reference. This is called casting object.

We have implicit casting and explicit casting.

It is a good practice, to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the instanceof operator.

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.

The statement below uses an Object reference to refer to a Student object.

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

This 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 compilation 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. Let us consider the example of the superclass **GeometricObject** and its two subclasses **Circle** and **Rectangle** again:

```
GeometricObject g1, g2;
```

```
g1 = new Circle("Red", true, 1); // g1 refers to a Circle object
```

```
g2 = new Rectangle("Red", true, 5, 6); // g2 refers to a Rectangle object
```

```
Circle c1 = (Circle)g1; // casting from superclass to subclass Circle
```

```
Rectangle r1 = (Rectangle)g2; // casting from superclass to subclass Rectangle
```

The instanceof Operator

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

```
GeometricObject g1 = new Circle();  
// Some lines of code ...  
// Perform casting if g1 is an instance of Circle  
if (g1 instanceof Circle) // instanceof  
    System.out.println("The circle area is " +  
        ((Circle)g1).area()); // casting
```

Demonstrating Polymorphism, instanceof, and Casting

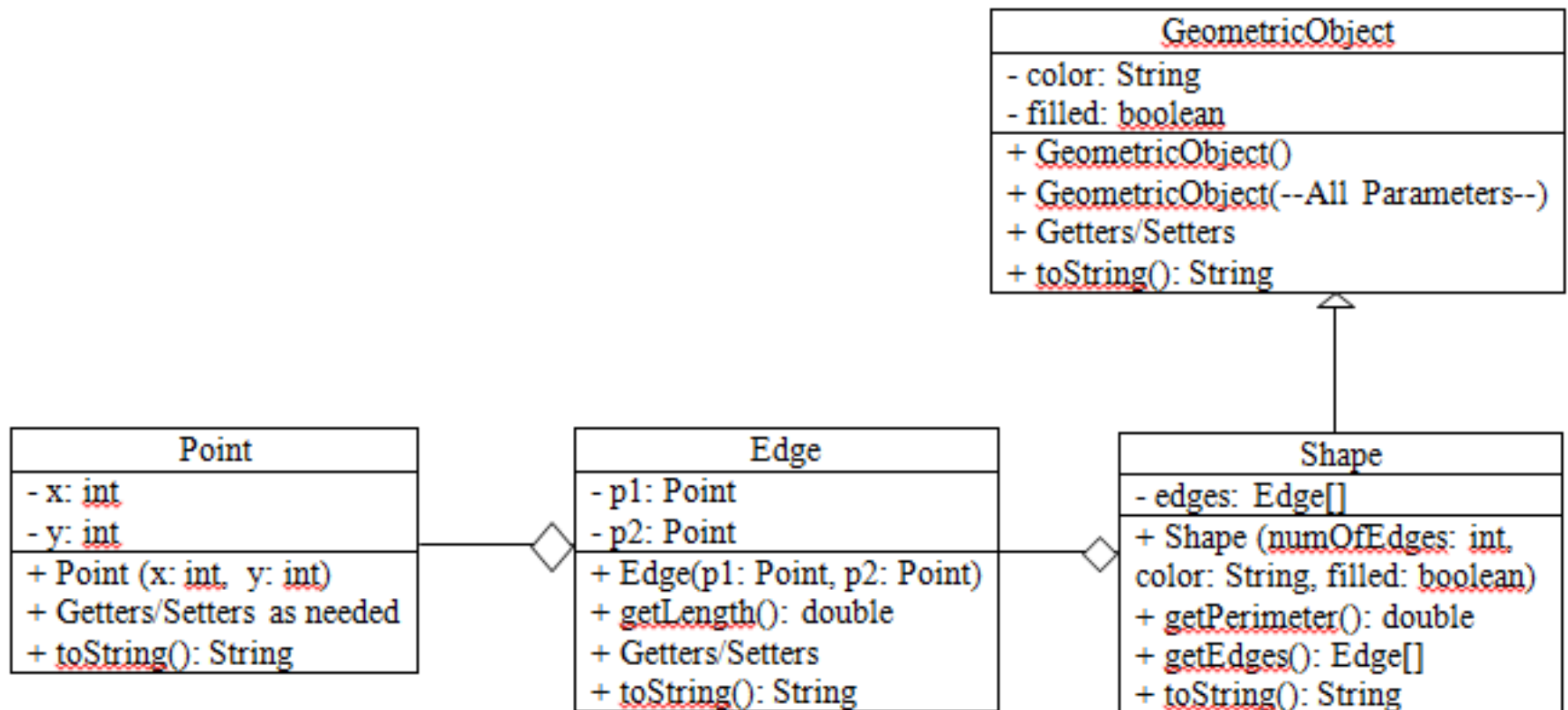
The example in the next slide creates two geometric objects one referring to a circle, and the other to a rectangle. Then it invokes the `displayObject` method to display the objects. This method displays the area if the object is a circle, and displays the perimeter if the object is a rectangle.

Demonstrating Polymorphism, instanceof, and Casting

```
public class CastingDemo {  
    public static void main(String[] args) {  
        GeometricObject g1, g2;  
        g1 = new Circle("Red", true, 1);  
        g2 = new Rectangle("Red", true, 5, 6);  
        displayObject(g1);  
        displayObject(g2);  
    }  
    public static void displayObject(GeometricObject g) {  
        if(g instanceof Circle)  
            System.out.println("Circle area is " + ((Circle)g).area());  
        else if(g instanceof Rectangle)  
            System.out.println("Rectangle perimeter is " +  
                               ((Rectangle)g).perimeter());  
    }  
}
```

Example Illustrating Polymorphism, Casting, and the instanceof Operator

This example involves both the “has a” relationship (composition) and the “is a” relationship (inheritance).



Example Illustrating Polymorphism, Casting, and the instanceof Operator

Download the code of the four classes below from Google classroom.

- GeometricObject:
 - toString() describes the object such as:
`Color: Red NotFilled`
- Point class:
 - toString() returns the point as follows: `(3, 6)`
- Edge class:
 - The getLength method in the Edge class returns the distance between p1 and p2:
$$\sqrt{(x_{p1} - x_{p2})^2 + (y_{p1} - y_{p2})^2}$$
 - toString() returns the edge as follows: `Edge{ (5, 6) , (1, 7) }`
- Shape class:
 - The constructor creates the edges array given the numOfEdges (the size of the array) as well as the color and filled attributes. The constructor creates the shape from random points which coordinates are generated randomly between 0 and 20.
 - getPerimeter() returns the perimeter of the shape which is the sum of the lengths of all edges.
 - toString() returns a complete description of the shape as per the following example:
`Shape with 5 edges.
Color: Red NotFilled
Edges:
Edge{(3, 15) , (13, 4) }
Edge{(13, 4) , (5, 5) }
Edge{(5, 5) , (15, 11) }
Edge{(15, 11) , (14, 10) }
Edge{(14, 10) , (3, 15) }`

Application:

- 1) Create a shape object with number of edges read from the user.
- 2) Display the description of the shape using the toString method.
- 3) Write a method that takes as parameter a shape and returns its longest edge.

Example Illustrating Polymorphism, Casting, and the instanceof Operator

In the application below part 4 is used to illustrate the concepts of polymorphism, casting, and the instanceof operator.

- GeometricObject:
 - toString() describes the object such as:
`Color: Red NotFilled`
- Point class:
 - toString() returns the point as follows: `(3, 6)`
- Edge class:
 - The getLength method in the Edge class returns the distance between p1 and p2:
 $\sqrt{(x_{p1} - x_{p2})^2 + (y_{p1} - y_{p2})^2}$
 - toString() returns the edge as follows: `Edge { (5, 6), (1, 7) }`
- Shape class:
 - The constructor creates the edges array given the numOfEdges (the size of the array) as well as the color and filled attributes. The constructor creates the shape from random points which coordinates are generated randomly between 0 and 20.
 - getPerimeter() returns the perimeter of the shape which is the sum of the lengths of all edges.
 - toString() returns a complete description of the shape as per the following example:
`Shape with 5 edges.
Color: Red NotFilled
Edges:
Edge{(3,15), (13,4)}
Edge{(13,4), (5,5)}
Edge{(5,5), (15,11)}
Edge{(15,11), (14,10)}
Edge{(14,10), (3,15)}`

Application:

- 1) Create a shape object with number of edges read from the user.
- 2) Display the description of the shape using the toString method.
- 3) Write a method that takes as parameter a shape and returns its longest edge.
- 4) Write a method that takes as parameter a GeometricObject array and returns an array of shapes that includes all and only the shapes in the array. The size of the returned array must be exactly equal to the number of shapes in the array. Invoke it in the main by passing to it the appropriate argument.

Example Illustrating Polymorphism, Casting, and the instanceof Operator

```
public class Application {  
    public static void main(String[] args) {  
        GeometricObject[] g = new GeometricObject[10];  
        for(int i = 0 ; i < g.length ; i += 2){  
            g[i] = new GeometricObject();  
            g[i+1] = new Shape(3 + i, "Red", false);  
        }  
        Shape[] s;  
        s = getShapes(g);  
        for (int i = 0 ; i < s.length ; i++)  
            System.out.println(s[i]);  
    } // end main
```

// Continued in next slide

Example Illustrating Polymorphism, Casting, and the instanceof Operator

```
public static Shape[] getShapes(GeometricObject[] g) {  
    int numberOfShapeObjects=0;  
    for(int i = 0 ; i < g.length ; i++)  
        if(g[i] instanceof Shape) // instanceof  
            numberOfShapeObjects++;  
    Shape[] shapes = new Shape[numberOfShapeObjects];  
    for(int i = 0 , j = 0; i < g.length ; i++)  
        if(g[i] instanceof Shape){ // instanceof  
            shapes[j] = (Shape)g[i]; // casting  
            j++;  
        }  
    return shapes;  
} // end getShapes  
} // end Application
```

The Object's equals Method

Like the `toString()` method, the `equals()` method is another method defined in the **Object** class. It 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);  
}
```

We can override the **equals** method in the **Circle** class to compare whether two circles are equal based on their radius as follows:

```
public boolean equals(Object o) {  
    if (o instanceof Circle) // instanceof  
        return radius == ((Circle)o).radius; // casting  
    return false; // if o is not a Circle  
}
```

NOTE

The equals method is intended to test whether two objects have the same contents, provided that the method is modified (overridden) in the defining class of the objects.

When used with objects the comparison operator `==` checks whether the two reference variables refer to the same object.

You know from before that when used with primitive data types (such as `int`) the comparison operator `==` is used for comparing two primitive data type values.