# Polymorphism and Interfaces

Lecture 14
CGS 3416 Spring 2020

March 9, 2020

# Polymorphism and Dynamic Binding

- If a piece of code is designed to work with an object of type X, it will also work with an object of a class type that is derived from X (any subclass of X).
- This is a feature known as **polymorphism** and is implemented by the Java interpreter through a mechanism called **dynamic binding**.
- Suppose there is a base class called `Shape`. Suppose that `Rectangle`, `Triangle`, and `Circle` are all subclasses of `Shape`.
- Then it is legal to attach derived objects to the base reference variables:

```
Shape s1 = new Circle();
Shape s2 = new Rectangle();
Shape s3 = new Triangle();
```

# Polymorphism and Dynamic Binding

- Suppose a method findArea() is called through one of these variables (s1, s2, s3) in the above example.
- The method must exist in the Shape class, but there can be override versions in the subclasses.
- If so, then through dynamic binding, the method that runs will be based on the attached object's type, as a priority over the reference variable type (Shape):

```
s1.findArea(); // from the Circle class
s2.findArea(); // from the Rectangle class
s3.findArea(); // from the Triangle class
```

- If any of these subclasses did not override the finArea() method, then the Shape class' findArea() method will run.

# Polymorphism and Dynamic Binding

- If a method expects a parameter of type X, it is legal to pass in an object of a type derived from X in that slot:

```
// Sample method
public int draw(Shape s)
{ // definition code }

// sample calls
Shape s1 = new Shape();
Shape s2 = new Circle();
Shape s3 = new Rectangle();

draw(s1); // normal usage
draw(s2); // passing in a Circle object
draw(s3); // passing in a Rectangle object
```

# Another Example

- Notice that a useful application of polymorphism is to store many related items, but with slightly different types (i.e. subclasses of the same superclass), in one storage container – for example, an array – and then do common operations on them through overridden functions.
- Assume the setup in the previous example, base class Shape and derived classes `Rectangle, Circle, Triangle`. Suppose the base class has a `findArea()` method (probably abstract, since we don't know how to compute the area for a generic shape), and each derived class has its own `findArea()` method.
- Note that in the for-loop, the appropriate area methods are called for each shape attached to the array, without the need for separate storage for different shape types (i.e. no need for an array of circles, and a separate array of rectangles, etc).

# Another Example

```
Shape[] list = new Shape[size];
// create an array of Shape reference variables

list[0] = new Circle(); // attach a Circle to first
array slot
list[1] = new Rectangle(); // attach a Rectangle to
second slot
list[2] = new Triangle(); // attach a Triangle to
third slot

for (int i = 0; i < list.length; i++)
     System.out.println("The area of shape  " + i +
           " = " + list[i].findArea())
```

# Casting

- Since a derived object can always be attached to a corresponding base class reference variable, this is a type of casting that is implicitly allowed.
- Similarly, direct assignment between variables (derived type assigned into base type) in this order is also allowed, as are explicit cast operations.

```
Shape s1, s2; // Shape is the base class
Circle c; // Circle is a derived class

s1 = new Circle(); // automatically legal

s2 = c; // automatically legal
s1 = (Shape)c; // explicit cast used, but
equivalent to above
```

# Casting

To convert an instance of a superclass (base) to an instance of a subclass (derived), the explicit cast operation must be used:

```
c = s1; // would be illegal -- cast needed
c = (Circle)s1; // legal (though not always so
useful)
```

# The instanceof operator

The instanceof operator checks to see if the first operand (a variable) is an instance of the second operand (a class), and returns a response of type *boolean.*

```
Shape s1;
Circle c1;
// other code.....

if (s1 instanceof Circle)
     c1 = (Circle)s1; // cast to a Circle variable
```

# Interfaces

- Java does not allow multiple inheritance
  - A subclass can only be derived from one base class with the keyword extends
  - In Java, the **interface** can obtain a similar effect to multiple inheritance
- **Interface** - A construct that contains only constants and abstract methods
  - Similar to abstract class
  - Different, since an abstract class can also contain regular variables and methods
  - Can use as a base type name (just like regular base classes)
  - Cannot instantiate (like an abstract class)

# Format for declaring an interface:

```
modifier interface Name
{
constant declarations
abstract method signatures - keyword "abstract"
not needed.  ALL methods in an interface are abstract
}
```

- Use the keyword implements to state that a class will use a certain interface.
- In this example, Comparable is the name of an interface.
- The class ComparableCircle inherits the data from the Comparable interface, and would then need to implement the methods (to be able to use them).

```
class CompCircle extends Circle implements Comparable
{
     // ....
}
```

# Other rules:

- Only single inheritance for classes, with `extends`
- Interfaces can inherit other interfaces (even multiple), with `extends`

```
public interface NewInterface extends interface1,
      ..., interfaceN
```

- classes can implement more than one interface with `implements`

```
public class NewClass extends BaseClass
     implements interface1, ..., interfaceN
```

# The Cloneable interface

- A special interface in the Java.lang package which happens to be empty:

```
public interface Cloneable
{
}
```

- This is a *marker interface* – no data or methods, but special meaning in Java.
- A class can use the clone() method (inherited from class Object) only if it implements `Cloneable`.