CS415

INTRODUCTION TO COMPUTER SCIENCE

FALL 2017

9 POLYMORPHISM

CHAPTER 5

LAST TIME

- Interfaces
- When to use interfaces
- Interfaces compared to inheritance

PREVIEW

- Polymorphism in Java
  - Inheritance
  - Interfaces
- Using polymorphism
- Advantages and limits of polymorphism

OBJECT-ORIENTED PROGRAMMING

- The 3 most important characteristics of OO
  - Encapsulation
  - Inheritance
  - Polymorphism

WHAT IS POLYMORPHISM?

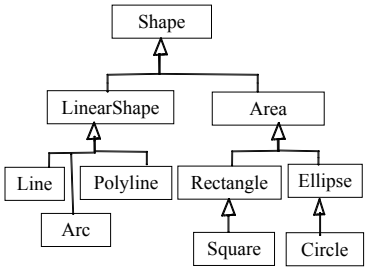
- Literally, many shapes
- Ability of the same object to be many types
- Ability of the same variable to refer to objects of different types
- Ability of different (but related) objects to respond to the same message in different (appropriate) ways

KINDS OF POLYMORPHISM

- Inheritance polymorphism
  - a Rectangle is-a Shape
  - an Ellipse is-a Shape
- Interface polymorphism
  - a Rectangle acts-as Colorable
  - a Car acts-as Colorable

INHERITANCE POLYMORPHISM

- All descendants of Shape **are** Shape objects;
- All respond to Shape messages.
- If *move* is a Shape method, code below uses polymorphism through the variable s.




```
Shape s;
s = new Line ( x1, y1, x2, y2 );
s.move( dx, dy );
s = new Rectangle(  x3, y3 );
s.move( dx, dy );
s = new Circle( x4, y4, r );
s.move( dx, dy );
```


METHOD RESOLUTION REVISITED

```
Shape s;
s = new Line ( x1, y1, x2, y2 );
s.move( dx, dy );
s = new Rectangle(  x3, y3 );
s.move( dx, dy );
s = new Circle( x4, y4, r );
s.move( dx, dy );
```

- Each *s.move* invocation calls the “right” method based on Java’s method resolution algorithm.
- For example, the last *s.move* is resolved as follows:
- **if** *Circle* has overridden *move*, execute *Circle.move*
  else if *Ellipse* has overridden *move*, execute *Ellipse.move*
  else if *Area* has overridden *move*, execute *Area.move*
  else execute *Shape.move*

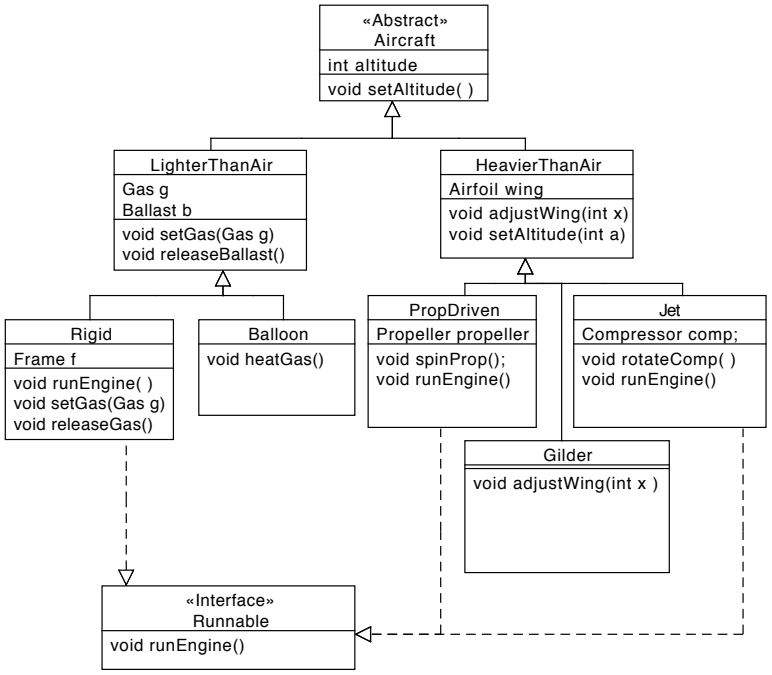
DYNAMIC BINDING

- This method resolution is an example of dynamic binding.
- The decision to invoke a particular method (the binding of the method name to a particular method definition) is determined at run-time (dynamically), rather than at compile time
- Compile-time binding is called static binding

VARIABLE CONCEPTS

```
Shape s;
s = new Line ( x1, y1, x2, y2 );
s.move( dx, dy );
s = new Rectangle(  x3, y3 );
s.move( dx, dy );
s = new Circle( x4, y4, r );
s.move( dx, dy );
```

- The variable, *s*, has a declared type of *Shape*.
- Its actual type changes from *Line* to *Rectangle* to *Circle* based on the object that is assigned to it.
- The actual type of a variable must be the same as its declared type or a subtype of the declared type.



INTERFACE POLYMORPHISM

- An interface defines a type
- If a class implements the interface, all of its objects have that type.

```
// Shape, Vehicle, Table all implement Colorable
Rectangle r = new Rectangle( x, y, color );
Vehicle   v = new Car( x, y, color );
Table     t = new Table( x, y, color );

ColorChooser choose = new ColorChooser();
// ColorChooser has a changeColor( Colorable ) method
choose.changeColor( r ); // user selects
choose.changeColor( v ); //   new colors for 3
choose.changeColor( t ); //   diverse objects
```

CHANGECOLOR METHOD

- ColorChooser.changeColor(Colorable c)
  - Present user with an *interactive* color utility
  - The utility might start by showing the current color of the *Colorable*, which it gets with

```
c.getColor();
```

  - Once the user defines a color and picks OK, the utility changes the *Colorable*'s color with

```
c.setColor( newColor );
```

  - *changeColor* doesn't need to know anything else about the object except the *Colorable* interface

VARIABLE CONCEPTS AGAIN

- Parameters are also variables. The formal parameter (declared at the method definition) is the declared type .
- The actual type is the type of the actual parameter.
- *changeColor* has a formal parameter of type *Colorable*.
- The actual parameter is a *Rectangle*, then *Car*, then *Table*.

```
// ColorChooser has a changeColor( Colorable ) method
choose.changeColor( r ); // user selects
choose.changeColor( v ); //   new colors for 3
choose.changeColor( t ); //   diverse objects
```

- The actual type of a variable must be the same as its declared type or any subtype of the declared type.

METHOD RESOLUTION FOR INTERFACES

- The method resolution algorithm for interfaces is the same as for inheritance and it searches through the inheritance hierarchy
- Java looks for the requested method in the class definition for the actual type;
  - if there is none, it looks at the superclass definition for the method
    - if there is none, it looks in its superclass,
    - etc.

USING POLYMORPHISM

- Polymorphism is used in your program through the appropriate use of variables (and, especially, parameters)
- Use a variable whose declared type is a super type of the actual type.
- Pass an actual parameter to a formal parameter whose declared type is a super type of the actual type.

INCORRECT VARIABLE USE

The actual type of a variable must be the same as its declared type or any subtype of the declared type.

```
Shape s;
Line l;  // Line extends shape

l = new Line( x1, y1, x2, y2 );
s = l;    // Legal: all Lines are Shapes

s.move( dx, dy ); // this is ok

l = s;     // *** Error *****
           // This causes compiler error;
           // not all Shapes are Lines.
s.setP1( p ); // *** Error ****
              // not all Shapes have
              //   setP1 method
```


POWER OF POLYMORPHISM

- Polymorphism increases our ability to write general purpose methods
- `ColorChooser.changeColor( Colorable )` works for any class that implements the `Colorable` interface

LIMITS OF POLYMORPHISM

- Whenever we utilize a variable or parameter polymorphically (i.e., let it represent many classes, not just one specific class), we lose access to any of the features of the actual objects that are not represented in the polymorphic view.

POLYMORPHISM SUMMARY

- Actual type of a parameter or variable can be:
  - if declared type is a class
    - the declared type, or
    - any subtype of the declared type
  - if declared type is an interface
    - any class that implements the declared type, or
    - any subclass of a class that implements the declared type

REVIEW

- Polymorphism in Java
  - Inheritance
  - Interfaces
- Using polymorphism
- The power and limitations of polymorphism

NEXT TIME

- Expressions
- Other primitive data types
  - float
  - double
- Conditionals
- if statements