ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of January 21, 2013

Abstract

- Inheritance (part II)
  - Polymorphism

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Circle

Let's complete the implementation of the class **Circle**.

Where would you implement the method **area()**?

In the class **Shape** or int the class **Circle**?

Circle

```
public class Circle extends Shape {

    private double radius;

    public double getRadius() { return radius; }

    public double area() {
        return Math.PI * radius * radius;
    }

    public void scale( double factor ) {
        radius *= factor;
    }
}
```

Rectangle

Similarly, let's complete the implementation of the class **Rectangle**.

Where would you implement the method **area()**?

In the class **Shape** or int the class **Rectangle**?

Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public double area() {
        return width * height;
    }

    public void scale(double factor) {
        width = width * factor;
        height = height * factor;
    }

}
```

Don't get the wrong impression that inheritance is restricted to the classes that you are defining yourself. Inheritance is often used to specialize existing classes of the Java library.

```
import java.awt.TextField;
public class TimeField extends TextField {
    public Time getTime() {
        return Time.parseTime( getText() );
    }
}
// java.lang.Object
//   |
//   +--java.awt.Component
//         |
//         +--java.awt.TextComponent
//               |
//               +--java.awt.TextField
//                     |
//                     +--TimeField
```

Polymorphism

From the Greek words *polus* = many and *morphê* = forms, literally means has many forms.

1. *Ad hoc* polymorphism (overloading): a method name is associcated with different blocs of code

2. Inclusion (subtyping, data) polymorphism: an identifer (a reference variable) is associated with data of different types with the use of a subtype relation

In Java, a variable or a method is polymorphic if it refers to objects of more than one "class/type".

Method overloading

Method overloading means that two methods can have the same name but different signatures (the signature consists of the name and formal parameters of a method but not the return value).

Constructors are often overloaded, this occurs for the class `Shape`:

```
Shape() {
  x = 0.0;
  y = 0.0;
}
Shape( int x, int y ) {
  this.x = x;
  this.y = y;
}
```

$\Rightarrow$ Method overloading is sometimes referred to as *ad hoc* polymorphism (*ad hoc* = for a specific purpose).

Overloading (contd)

In Java, some operators are overloaded, consider the "+" which adds two numbers or concatenates two strings, a user can overload a method but not an operator.

Since the signatures are different, Java has no problem finding the right method:

```
static int sum( int a, int b, int c ) {
  return a + b + c;
}
static int sum( int a, int b ) {
  return a + b;
}
static double sum( double a, double b ) {
  return a + b;
}
```

Overloading (contd)

The class **PrintStream** has a specific **println** method for each primitive type (a good example of overloading):

```
println()
println( boolean )
println( char )
println( char[] )
println( double )
println( float )
println( int )
println( long )
```

Overloading (contd)

Pros: all the methods that implement a similar behaviour have the same name.

Cons: still have to provide one implementation for each behaviour.

“True” polymorphism: motivation 1

Problem: implement the method **isLeftOf** that returns **true** if **this Shape** is to the left of its argument.

isLeftOf

```
Circle c1, c2;
c1 = new Circle( 10, 20, 5 );
c2 = new Circle( 20, 10, 5 );

if ( c1.isLeftOf( c2 ) ) {
    System.out.println( "c1 isLeftOf c2" );
} else {
    System.out.println( "c2 isLeftOf c1" );
}
```

isLeftOf

```
Rectangle r1, r2;
r1 = new Rectangle( 0, 0, 1, 1 );
r2 = new Rectangle( 100, 100, 200, 400 );

if ( r1.isLeftOf( r2 ) ) {
    System.out.println( "r1 isLeftOf r2" );
} else {
    System.out.println( "r2 isLeftOf r1" );
}
```

isLeftOf

```
if ( r1.isLeftOf( c1 ) ) {
    System.out.println( "r1 isLeftOf c1" );
} else {
    System.out.println( "c1 isLeftOf r1" );
}

if ( c2.isLeftOf( r2 ) ) {
    System.out.println( "c2 isLeftOf r2" );
} else {
    System.out.println( "r2 isLeftOf c2" );
}
```

Absurd solution!

```
public boolean isLeftOf( Circle c ) {
    return getX() < c.getX();
}
public boolean isLeftOf( Rectangle r ) {
    return getX() < r.getX();
}
```

Why is that solution absurd?

Absurd solution!

```
public boolean isLeftOf( Circle c ) {
    return getX() < c.getX();
}
public boolean isLeftOf( Rectangle r ) {
    return getX() < r.getX();
}
```

- As many implementations as kinds of shape!
- All the implementations are the same!
- Whenever a new kind of **Shape** is defined (say Triangle) then a method **iLeftOf** must be created!

Solution

What do you propose?

The method **getX()** is common to all the Shapes; all shapes have a **getX()**.

```
public boolean isLeftOf( ``Any Shape'' s ) {
    return getX() < s.getX();
}
```

How does one write "Any Shape"?

Solution

Implement the method **isLeftOf** in the class **Shape** as follows.

```
public boolean isLeftOf( Shape s ) {
    return getX() < s.getX();
}
```

isLeftOf

```
Circle c;
c = new Circle( 10, 20, 5 );

Rectangle r;
r = new Rectangle( 0, 0, 1, 1 );

if ( c.isLeftOf( r ) ) {
    System.out.println( "c isLeftOf r" );
} else {
    System.out.println( "r isLeftOf c" );
}
```

isLeftOf

```
if ( c.isLeftOf( r ) ) {
    // ...
```

The method **isLeftOf** of the object designated by **c** is called.

Okay, **c** designates an object of the class **Circle**, which inherits the method **isLeftOf**.

isLeftOf

```
if ( c.isLeftOf( r ) ) {
    // ...
```

Hum, when the method **isLeftOf** is called, the value of the actual parameter, **r**, is copied into the formal parameter **s**.

Does it mean that the following statements are valid!?

```
Shape s;
Rectangle r;
r = new Rectangle( 0, 0, 1, 1 );
s = r;
```

Types

“A variable is a storage location and has an associated type, sometimes called its compile-time type, that is either a primitive type (§4.2) or a reference type (§4.3). A variable always contains a value that is assignment compatible (§5.2) with its type.”

“Assignment of a value of compile-time reference type S (source) to a variable of compile-time reference type T (target) is checked as follows:

- If S is a class type:
  - If T is a class type, then S must either be the same class as T, or S must be a subclass of T, or a compile-time error occurs.”

⇒ Gosling et al. (2000) *The Java Language Specification.*

isLeftOf

Based on that definition, the following statements are valid.

```
Shape s;
Rectangle r;
r = new Rectangle( 0, 0, 1, 1 );
s = r;
```

but "**r = s**" is not!

Polymorphic variable

The variable **s** designates any object that is from a subclass of **Shape**.

```
Shape s;
```

Usage:

```
s = new Circle( 0, 0, 1 );
s = new Rectangle( 10, 100, 10, 100 );
```

Polymorphic method: “true” polymorphism

```
public boolean isLeftOf( Shape other ) {
    boolean result;
    if ( getX() < other.getX() ) {
       result = true;
    } else {
       result = false;
    }
    return result;
}
```

Usage:

```
Circle c = new Circle( 10, 10, 5 );
Rectangle d = new Rectangle( 0, 10, 12, 24 );
if ( c.isLeftOf( d ) ) { ... }
```

Polymorphic variable (contd)

```
Shape s;
Circle c;
c = new Circle( 0, 0, 1 );
s = c;

if ( c.getX() ) { ... } // valid?
if ( s.getX() ) { ... } // valid?

if ( c.getRadius() ) { ... } // valid?
if ( s.getRadius() ) { ... } // valid?
```

Polymorphic variable (contd)

```
Shape s;
Circle c;
c = new Circle( 0, 0, 1 );
s = c;
```

The object designated by **s** is still a **Circle**. The class of object does not change during the execution of the program.

Polymorphic variable (contd)

```
Shape s;
Circle c;
c = new Circle( 0, 0, 1 );
s = c;

if ( s.getX() ) { ... }
```

When **s** is used to designate a **Circle**, the **Circle** is "seen as" a **Shape**, meaning that only the characteristics (methods and variables) of the class **Shape** can be used.

Polymorphic variable (contd)

```
Shape s;
Circle c;
c = new Circle( 0, 0, 1 );
s = c;

if ( s.getX() ) { ... }
```

Here, **s** of type **Shape**, **getX()** is defined in the class **Shape**.

Polymorphic variable (contd)

```
Shape s;
Circle c;
c = new Circle( 0, 0, 1 );
s = c;

if ( s.getX() ) { ... }
```

This makes sense, **s** can be used to designate objects of the class **Shape** or a subclass of **Shape**. This object has all the characteristics of a **Shape**.

Polymorphic variable (contd)

```
Shape s;
Circle c;
c = new Circle( 0, 0, 1 );
s = c;

if ( s.getRadius() ) { ... }
```

The above statement **is not valid**. Why? The method **getRadius()** is not defined in the class **Shape** (or its parents).

Polymorphic variable (contd)

1) The type of a reference variable defines the set of classes whose objects could be designated by the reference.

2) The type of a reference variable defines the set of operations (method calls, access to instance variables, etc.) that are valid.

Polymorphism

Polymorphism is a powerful concept. The method **isLeftOf** can be used to compare not only **Circles** and **Rectangles** but also any future subclass of **Shape**.

```
public class Triangle extends Shape {
   // ...
}
```

“True” polymorphism: motivation 2

Problem: write a method that compares the **area** of any two Shapes.

Absurd solution!

Write methods with the same name and all four possible signatures (method overloading):

(`Circle`, `Circle`), (`Circle`, `Rectangle`), (`Rectangle`, `Circle`) and (`Rectangle`, `Rectangle`).

- As many implementations as pairs of shapes!
- All the implementations are the same!
- Whenever a new kind of **Shape** is defined (say **Triangle**) then new methods **compareTo** must be created!

Solution

What do you propose? How about this?

```
public class Shape {

  // ...

  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }
```

Solution

```
public class Shape {
  // ...

  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }
```

The above declaration would not compile! Why? Because, the superclass **Shape** does not have method **area()**.

Solution

Proposal? Let's create a dummy implementation of the method **area()**.

```
public class Shape {
  // ...
  // Must be redefined by the subclasses or else ...

  public double area() {
      return -1.0;
  }
  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }
```

Solution

Too dangerous! The implementer of the subclass is not forced to redefined the method **area()**.

```
public class Shape {
  // ...
  // Must be redefined by the subclasses or else ...

  public double area() {
      return -1.0;
  }
  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }
```

Solution: abstract

The solution is to declare the method **area()** abstract in the superclass **Shape**. An **abstract** method is declared using the keyword **abstract**, it has a signature but no body.

```
public class Shape {
  // ...

  public abstract double area();  // <----

  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }
```

The above definition, alas, does not compile! Why?

Solution: abstract

```
public class Shape {
  // ...

  public abstract double area();  // <----

  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }
```

Imagine creating an object of the class **Shape**, that object would have a method **area()** that has no statements attached to it!

Solution: abstract class

```
public abstract class Shape { // <---
  // ...

  public abstract double area();  // <----

  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }
```

A **class** that has an **abstract method** must be **abstract**. One cannot create an object of an abstract class! The statement “new Shape()” would cause a compile-time error.

Abstract classes

- A class that contains an **abstract method** (declared in that class or inherited) **must** be declared abstract;
- An abstract class cannot be used to create objects;
- A class that contains no abstract methods **can** also be declared abstract to prevent the creation of objects of this class. E.g. Employee, SalariedEmployee, HourlyEmployee.

Solution: abstract class

What have we achieved?

```
public class Triangle extends Shape {

}
```

```
Triangle.java:1: Triangle is not abstract and
does not override abstract method area() in Shape
public class Triangle extends Shape {
       ^
1 error
```

It is now **impossible** to create a concrete subclass of **Shape** that has no method **area()**!

Solution: abstract methods and classes

The declaration of an **abstract method** forces all the (concrete) subclasses to implement that method!

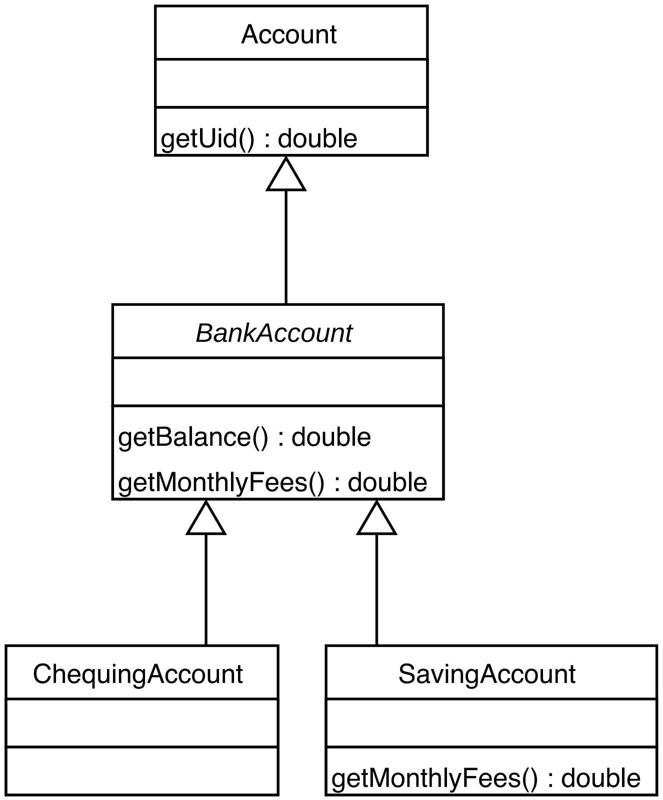
```
public abstract class Shape {
  // ...

  public abstract double area();

  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }

}
```

Late binding (a.k.a. dynamic binding, virtual binding)



Both classes BankAccount and SavingAccount are declaring a method getMontlyFees();

Let's say that the method **getMonthlyFees** of the class **BankAccount** always returns 25.

```
public double getMonthlyFees() {
    return 25.0
}
```

The class **SavingAccount** overwrites this definition with the following.

```
public double getMonthlyFees() {
    double result;
    if ( getBalance() > 5000 ) {
       result = 0.0;
    } else {
       result = super.getMontlyFees();
    }
    return result;
}
```

```
Account a;
BankAccount b;
SavingAccount s;

s = new SavingAccount();
s.getMontlyFees();

b = s;
b.getMontlyFees();

a = b;
a.getMontlyFees();
```

Dynamic Binding

Let **S** (source) be the type of the object currently designated by a reference variable of type **T** (target).

Unless the method is static or final, the lookup i) occurs at runtime and ii) starts at the class **S**: if the method is found, this is the method that will be executed, otherwise the immediate superclass is considered, this process continues until the first occurrence of the method is found.

$\Rightarrow$ Sometimes called late or virtual binding.