05 Abstract Classes and Interfaces

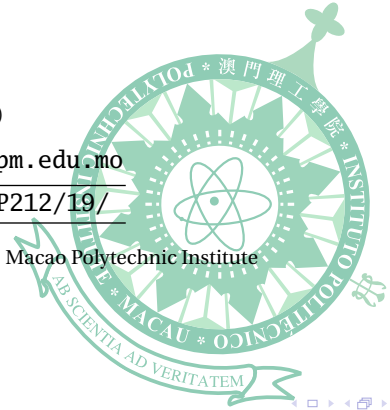
Instructor: Ke Wei（柯韋）

A319 Ext. 6452 wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

September 26, 2019



Outline

Common Methods without Common Implementation

- Both *Circle* and *Rectangle* contain the *getArea* and *getPerimeter* methods.
- The common methods should be put into their superclass *Shape*.
- The implementations of these common methods are completely different in different classes.
- There is no common implementation to be put into the superclass. The implementation *completely* depends on the specific type of shape.
- Such methods are referred to as *abstract methods*. They have signatures in the superclass, but no implementation.
- If a class contains abstract methods, it must be declared as an *abstract class*.

Example: *Shape*

```
1  public abstract class Shape {
2      private String name;
3      protected Shape() { this("UNKNOWN"); }
4      protected Shape(String name) {
5          this.name = name;
6      }
7      public String getName() { return name; }
8      public void setName(String name) { this.name = name; }
9      @Override public String toString() {
10         return "Shape [Name: "+name+"]";
11     }
12     public abstract double getArea();
13     public abstract double getPerimeter();
14 }
```

Abstract Methods

- Abstract methods capture the function, not the implementation.
- Abstract methods are placeholders that are meant to be overridden. Thus, we don't have `private` or `static` abstract methods.
- Abstract methods allow us to write code that makes use of a function without knowing the implementation, this helps to partially specify a *framework*.

```
1 public abstract class Shape { ...
2     public abstract double getArea();
3     public static double getTotalArea (Shape[] ss) {
4         double ta = 0.0;
5         for ( Shape s : ss ) ta += s.getArea();
6         return ta;
7     }
8 }
```

Abstract Classes

- An abstract class is a class that is declared `abstract`.

  `public abstract class` *Shape* `...`
- An abstract class may or may *not* include abstract methods.
- Abstract classes cannot have instances of their own.
- Abstract classes can define constructors, which are invoked in the constructors of their subclasses.
- Abstract classes can be inherited, and they are designed to be used as superclasses.
- An abstract method cannot be contained in a non-abstract class, directly or indirectly by inheritance.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be declared abstract.
- An abstract subclass can declare an abstract method to override a concrete method from its superclass.

Interfaces

- An interface is a class-like construct that contains *only* constants and abstract methods.
- To distinguish an interface from a class, Java uses the `interface` keyword to declare an interface.

```
public interface Stroke {
    int THIN = 1, THICK = 8;
    void setStroke(int stroke);
    int getStroke();
}
```

- In an interface, all data fields are `public final static` and all methods are `public abstract`. These modifiers can be omitted.
- As with an abstract class, an interface cannot have instances of its own.
- Like an abstract class, an interface can be used as a data type for a variable, as the result of casting, and so on.

Interfaces

- An interface is a class-like construct that contains *only* constants and abstract methods.
- To distinguish an interface from a class, Java uses the `interface` keyword to declare an interface.

```
public interface Stroke {
    public static final int THIN = 1, THICK = 8;
    public abstract void setStroke(int stroke);
    public abstract int getStroke();
}
```

- In an interface, all data fields are `public final static` and all methods are `public abstract`. These modifiers can be omitted.
- As with an abstract class, an interface cannot have instances of its own.
- Like an abstract class, an interface can be used as a data type for a variable, as the result of casting, and so on.

Declaring a Class to Implement an Interface

- We cannot set the pen width of a *Circle*, because *Circle* does not implement *Stroke*.
- We can declare a subclass of *Circle* — *OutlinedCircle* to implement the interface while retaining all the features in *Circle*.

```
1 public class OutlinedCircle extends Circle implements Stroke {
2     private int stroke = Stroke.THIN;
3     @Override public void setStroke(int stroke) { this.stroke = stroke; }
4     @Override public int getStroke() { return stroke; }
5 }
```

- We write a method to set the pen width of all object that implements *Stroke*.

```
1 public static void setAllStrokes(Stroke[] a, int stroke) {
2     for ( Stroke x : a ) x.setStroke(stroke);
3 }
```

Implementing Multiple Interfaces

Suppose we have an interface to return the diagonal of a shape.

```
public interface Diagonal { double getDiagonal(); }
```

We define an *OutlinedRectangle* that has a pen width and a diagonal.

```
1 public class OutlinedRectangle extends Rectangle implements Stroke, Diagonal {
2     private int stroke = Stroke.THIN;
3     @Override public void setStroke(int stroke) { this.stroke = stroke; }
4     @Override public int getStroke() { return stroke; }
5     @Override public double getDiagonal() {
6         return Math.hypot(getWidth(), getHeight());
7     }
8 }
```

Using Classes That Implement Interfaces

The method *maxDiagonal* returns the object with the maximum diagonal.

```
1 public static Diagonal maxDiagonal(Diagonal[] a) {
2     Diagonal m = null;
3     for ( Diagonal x : a )
4         if ( m == null || m.getDiagonal() < x.getDiagonal() )
5             m = x;
6     return m;
7 }
```

We can pass an array of outlined rectangles to both *maxDiagonal* and *setAllStrokes*.

```
1 OutlinedRectangle[] a = ...
2 setAllStrokes(a, Stroke.THICK);
3 OutlinedRectangle maxDia = (OutlinedRectangle) maxDiagonal(a);
```

Anonymous Classes

- An anonymous class is a *local* class without a name.
- An anonymous class must be used in conjunction with an interface or a superclass.

```
public interface UnaryOp { int op(int x); }
```

- Now, we can define an object that carries a function by using an anonymous class.

```
1 public static UnaryOp incBy(int delta) {
2     return new UnaryOp() {
3         @Override public int op(int x) { return x + delta; }
4     };
5 }
```

- An anonymous class is *defined* and *instantiated* in a single expression using the `new` operator. An anonymous class is useful to *capture* the local context, say `int delta`, which is not available when defining a named class.

Practice: Implementing a Peano Number System

Tasks:

1. Study the Peano number system.
2. Establish the class framework.
3. Implement the *compareTo* method for Peano number objects.
4. Study the algorithms for the *toInt* and *fromInt* methods to convert/create a Peano number object to/from a Java integer.
5. Implement the *toInt* method and the *fromInt* static factory method.
6. Design a test case to test your implementations.

Submission (in a week):

1. Zip your source files, including `N.java`, `Z.java`, `S.java` and `Test.java` into `Peano.zip`.
2. Upload the `Peano.zip`.

The Peano Number System

Objects		$\emptyset$ is a number.
	Given a number a	a^+ is a number.
Relations	Given a number v	$\emptyset \leq v \triangleq$ `true`.
	Given a number a_u^+	$a_u^+ \leq \emptyset \triangleq$ `false`.
	Given numbers a_u^+, a_v^+	$a_u^+ \leq a_v^+ \triangleq a_u \leq a_v$.
	Given numbers u, v	$u = v \triangleq u \leq v \wedge v \leq u$.
Arithmetics	Given a number v	$\emptyset + v \triangleq v$.
	Given numbers a_u^+, v	$a_u^+ + v \triangleq (a_u + v)^+$.
	Given a number v	$\emptyset \times v \triangleq \emptyset$.
	Given numbers a_u^+, v	$a_u^+ \times v \triangleq (a_u \times v) + v$.
	Given a number u	$u^2 \triangleq u \times u$.

Peano Number System — Abstract N

We define three classes, N for abstract numbers, Z for 0 and $S(a)$ for a^+.

```
public abstract class N {
    public abstract  boolean le(N v);
    public           boolean eq(N v) { return this.le(v)
                                          && v.le(this); }
    public abstract  N add(N v);
    public abstract  N mul(N v);
    public           N sq() { return this.mul(this); }
}
```

We leave those operations on specific objects to the subclasses, and implement the operations on general objects.

Peano Number System — Concrete Z and S

```
public class Z extends N { // class for ∅
    @Override public boolean le(N v) { return true; }
    @Override public N add(N v) { return v; }
    @Override public N mul(N v) { return this; }
}
public class S extends N { // class for a+
    private N a;
    public S(N a) { this.a = a; }
    @Override public boolean le(N v) {
        if ( v instanceof Z ) return false;
        else return a.le(((S)v).a);   // v must be an instance of S
    }
    @Override public N add(N v) { return new S(a.add(v)); }
    @Override public N mul(N v) { return a.mul(v).add(v); }
}
```

Peano Number System — Test Case

```
public class Test {
    public static void main(String[] args) {
        N n0   = new Z(),      n1   = new S(n0),
          n2   = n1.add(n1),   n3   = n2.add(n1),
          n4   = n2.sq(),      n5   = n2.add(n3),
          n6   = n3.mul(n2),   n12  = n3.mul(n4),
          n16  = n12.add(n4),  n16p = n4.sq();

        System.out.println(n5.le(n6));
        System.out.println(n12.le(n3));
        System.out.println(n16.eq(n16p));
    }
}
```

This should output: `true`, `false` and `true`.

The *compareTo* Method

- We have defined the total order *le* on our Peano numbers.
- It is easy to implement the *compareTo* method to compare one Peano number with another,
- based on *le*.
- For any two Peano numbers u and v, we have $u \leq v$ or $v \leq u$ or both, but never neither.
- The order on natural numbers has this *trichotomy* property.
- We extend the N class to implement the *compareTo* method declared in the $Comparable\langle N\rangle$ interface.

```
public class N implements Comparable⟨N⟩ {
    ...
    @Override public int compareTo(N v) { ... }
}
```

The *toInt* and *fromInt* Methods

- The objects that store our Peano numbers are actually linked lists, with
- S being a node and a being the link field to the next node.
- Each linked list terminates at a Z object.
- The number that such a linked list represents is the number of S nodes in the list.
- We can write a recursive method to convert a Peano number object to a Java integer, by computing the length of the list.

$$\begin{aligned} toInt(\emptyset) &\triangleq 0, \\ toInt(a^+) &\triangleq 1 + toInt(a). \end{aligned}$$

- Similarly, we can create a Peano number object from a Java integer.

$$\begin{aligned} fromInt(0) &\triangleq \emptyset, \\ fromInt(n) &\triangleq fromInt(n-1)^+, \quad \text{if } n \geq 1. \end{aligned}$$