Object-Oriented Programming in Java: Advanced Capabilities

Topics in This Section

- Abstract classes
- Interfaces
- @Override
- Visibility modifiers
- Enums
- JavaDoc options
- The classpath

Handling Mixed-but-Related Types

- **We have**
  - Circle, Rectangle, and Square classes
- **We want to be able to**
  - Call getArea on an instance of any of three, even if we do not know which of the three types it is
  - Make an array of mixed shapes and calculate the sum of the areas
  - Make this array-summing method flexible enough to handle future types of shapes (Triangle, Ellipse, etc.)

Attempt 1 (Failure)

- **Implement each shape independently**
  - Give each of Circle, Rectangle, and Square a getArea method
- **Make Object[] containing mixed instances**
  - Pass this to ShapeUtils.sumAreas
- **In sumAreas, define parameter as Object[]**
  - Then loop down, call getArea on each, add up result

Attempt 1: Shapes

- **Circle**

```
public class Circle {
   ...
   public double getArea() { ... }
}
```

- **Rectangle**

```
public class Rectangle {
   ...
   public double getArea() { ... }
}
```

- **Square**

```
public class Square extends Rectangle {
   ...
   public double getArea() { ... }
}
```

Attempt 1: Desired Testing Code

```
public class ShapeTest {
  public static void main(String[] args) {
    Object[] shapes = { new Circle(10),
                        new Rectangle(5, 10),
                        new Square(10) };
    System.out.println("Sum of areas: " +
                       ShapeUtils.sumAreas(shapes));
  }
}
```

Attempt 1: Utility Class

```
public class ShapeUtils {
  public static double sumAreas(Object[] shapes) {
    double sum = 0;
    for(Object s: shapes) {
      sum = sum + s.getArea();
    }
    return(sum);
  }
  ...
}
```

This will not even compile! Why not?

Attempt 2 (Reasonable but Imperfect)

- **Make a class called Shape**
  - Define a getArea method that always returns -1
    - Since no real shape can have a negative area, you will notice if you call getArea and get back a negative number
- **Have all shapes extend this base class**
  - Circle, Rectangle, and Square directly or indirectly extend Shape
    - Each provide a more specific definition of getArea
- **Make Shape[] containing mixed instances**
  - Pass this to ShapeUtils.sumAreas
- **In sumAreas, define parameter as Shape[]**
  - Then loop down, call getArea on each, add up result

Attempt 2: Base Class

```
public class Shape {
   public double getArea() {
      return(-1);
   }
}
```

Attempt 2: Shapes

- **Circle**

```
public class Circle extends Shape {
   ...
   public double getArea() { ... }
}
```

- **Rectangle**

```
public class Rectangle extends Shape {
   ...
   public double getArea() { ... }
}
```

- **Square**

```
public class Square extends Rectangle {
   ...
   public double getArea() { ... }
}
```

Attempt 2: Utility Class

```
public class ShapeUtils {
  public static double sumAreas(Shape[] shapes) {
    double sum = 0;
    for(Shape s: shapes) {
      sum = sum + s.getArea();
    }
    return(sum);
  }
  ...
}
```

It is somewhat conventional to call this class Shapes instead of ShapeUtils.
However, that name is a bit confusing to beginners.

Attempt 2: Testing Code

```
public class ShapeTest {
   public static void main(String[] args) {
      Shape[] shapes = { new Circle(10),
                         new Rectangle(5, 10),
                         new Square(10) };
      System.out.println("Sum of areas: " +
                         ShapeUtils.sumAreas(shapes));
   }
}
```

A good try, especially for someone new to OOP. But, although it works, it does have two deficiencies. What are they?

Abstract Classes

- **Idea**
  - A class that you cannot directly instantiate (i.e., on which you cannot use "new")
  - But you can subclass it and instantiate the subclasses
  - Methods marked abstract in parent class must be implemented by all child classes (unless they are also abstract)
- **Syntax**

```
public abstract class SomeClass {
  private SomeType instanceVar;
  public abstract SomeType abstractMethod(…);
  public SomeType concreteMethod(…) { … }
}
```

Semicolon only. No curly braces with body.

Motivation

- **Enforces behavior**
  - Guarantees that all subclasses will have certain methods
  - Allows you handle collections of mixed-but-related types
  - Makes sure that your method on the mixed types will still work in the future when new types are defined

- **Note**
  - Although abstract classes were widely used through Java 7, they are less used in Java 8. This is because new features were added to interfaces (next section) in Java 8, so now interfaces can do almost everything abstract classes can do, except for having mutable (modifiable) instance variables. And, interfaces have advantages that abstract classes lack. Abstract classes are still used, but interfaces more so.
  - Conclusion: look briefly at the upcoming example to see the basic idea, but concentrate more on interfaces in the next section.

Attempt 3 (Good)

- **Make an abstract class called Shape**
  - Define the specification for a getArea method
- **Have all shapes extend this base class**
  - Circle, Rectangle, and Square directly or indirectly extend Shape
    - Each provide a definition of getArea
- **Make Shape[] containing mixed instances**
  - Pass this to ShapeUtils.sumAreas
- **In sumAreas, define parameter as Shape[]**
  - Then loop down, call getArea on each, add up result

Attempt 3: Base Class

```
public abstract class Shape {
   public abstract double getArea();
}
```

Attempt 3: Shapes

- **Circle**

```
public class Circle extends Shape {
   ...
   public double getArea() { ... }
}
```

- **Rectangle**

```
public class Rectangle extends Shape {
   ...
   public double getArea() { ... }
}
```

- **Square**

```
public class Square extends Rectangle {
   ...
   public double getArea() { ... }
}
```

Attempt 3: Utility Class

```
public class ShapeUtils {
  public static double sumAreas(Shape[] shapes) {
    double sum = 0;
    for(Shape s: shapes) {
      sum = sum + s.getArea();
    }
    return(sum);
  }
  ...
}
```

Attempt 3: Testing Code

```
public class ShapeTest {
  public static void main(String[] args) {
    Shape[] shapes = { new Circle(10),
                       new Rectangle(5, 10),
                       new Square(10) };
    System.out.println("Sum of areas: " +
                       ShapeUtils.sumAreas(shapes));
  }
}
```

A very good solution, and this illustrates the general benefit of abstract classes. However, in this specific case, Shape has no instance variables, so an interface is slightly more flexible than an abstract class.

Interfaces

- **Idea**
  - A model for a class. Usually like an abstract class but without any concrete methods or instance variables
    - However, Java 8 interfaces *can* have concrete (default) methods and also static methods. This is covered in later section on Java 8 interfaces.
- **Syntax**

```
public interface Interface1 {
  SomeType method1(…);
}
public interface Interface2 {
  SomeType method2(…);
}
public class SomeClass implements Interface1, Interface2 {
  // Real definitions of method1 and method 2
}
```

Motivation

- **Enforces behavior**
  - Like abstract classes, guarantees classes have certain methods
- **More flexibility that abstract classes**
  - Classes can implement multiple interfaces
    - You cannot directly extend multiple abstract classes
- **New features in Java 8 interfaces**
  - Interfaces can have static methods
    - Example shown on upcoming slides in this section
  - Interfaces can have concrete (default) methods
    - Example and more details in later section on Java 8 interfaces
- **Restriction**
  - Even in Java 8, interfaces cannot have mutable (modifiable) instance variables

Concrete (Default) Methods

- **Java 8 interfaces can have real methods**
  - Not just method specifications
  - Interfaces still cannot have instance variables
  - Label the concrete methods with "default"
- **Example**

```
public interface SomeInterface {
  String method1(); // Method specification

  default String method2() { // Real (concrete) method
    // Normal code, perhaps calling method1
  }
  ...
}
```

Java 8: Interfaces and Abstract Classes

	Java 7 and Earlier	Java 8 and Later
Abstract Classes	• Can have concrete methods and abstract methods • Can have static methods • Can have instance variables • Class can directly extend one	(Same as Java 7)
Interfaces	• Can only have abstract methods – no concrete methods • Cannot have static methods • Cannot have mutable instance variables • Class can implement any number	• Can have concrete (default) methods and abstract methods • Can have static methods • Cannot have mutable instance variables • Class can implement any number

Conclusion: there is little reason to use abstract classes in Java 8. Except for instance variables, Java 8 interfaces can do everything that abstract classes can do, plus are more flexible since classes can implement more than one interface. This means (arguably) that Java 8 has real multiple inheritance. Default and static methods in interfaces are covered in more detail in a later section on Java 8 interfaces.

Attempt 4 (Best)

- **Make an interface called Shape**
  - Define the specification for a getArea method
- **Have all shapes implement this interface**
  - Circle, Rectangle, and Square directly or indirectly implement Shape
    - Each provide a definition of getArea
- **Make Shape[] containing mixed instances**
  - Pass this to Shape.sumAreas
- **In sumAreas, define parameter as Shape[]**
  - Then loop down, call getArea on each, add up result
  - Move the sumAreas method to the Shape interface
    - Java enforces that you call it via Shape.sumAreas, never just by sumAreas.

Attempt 4: Main Interface

```
public interface Shape {
   double getArea(); // Method specification
}
```

Attempt 4: Shapes

- **Circle**

```
public class Circle implements Shape {
   ...
   public double getArea() { ... }
}
```

- **Rectangle**

```
public class Rectangle implements Shape {
   ...
   public double getArea() { ... }
}
```

- **Square**

```
public class Square extends Rectangle {
   ...
   public double getArea() { ... }
}
```

Attempt 4: Static Method

```
public static double sumAreas(Shape[] shapes) {
   double sum = 0;
   for(Shape s: shapes) {
      sum = sum + s.getArea();
   }
   return(sum);
}
```

Where should I put this method?

I could put this static method in ShapeUtils as in the previous examples. But, since Java 8 interfaces allow static methods, a more natural place is in the Shape interface itself!

Attempt 4: Final Interface

```
public interface Shape {
   double getArea(); // Method specification

   static double sumAreas(Shape[] shapes) { // Static method
     double sum = 0;
     for(Shape s: shapes) {
       sum = sum + s.getArea();
     }
     return(sum);
   }
}
```

Unlike with classes, the default visibility of methods in interfaces is public, so there is a tiny change to the static method shown on the previous slide: the omission of the “public” declaration.

Attempt 4: Testing Code

```
public class ShapeTest {
  public static void main(String[] args) {
    Shape[] shapes = { new Circle(10),
                       new Rectangle(5, 10),
                       new Square(10) };
    System.out.println("Sum of areas: " +
                       Shape.sumAreas(shapes));
  }
}
```

@Override – Quick Preview

- **Idea**
  - When you override a method from the parent class or interface, you can mark it with @Override
    - Optional but strongly recommended
- **Syntax**

```
public class Parent {
  public void blah() { ... }
}

public class Child extends Parent {
  @Override
  public void blah() { ... }
}
```

Motivation

- **Catches errors at compile time instead of run time**
  - If you make a type in the name or signature of overridden method, it would still compile but would give wrong answer at compile time
    - This point applies only to extending regular classes, not to extending abstract classes or implementing interfaces
- **Expresses design intent**
  - Tells later maintainer "the meaning of this method comes from the parent class, I am not just inventing a new method"
    - This point applies to regular classes, abstract classes, and interfaces

Example

- **Parent class**

```
public class Ellipse implements Shape {
   public double getArea() { … }
}
```

If Ellipse does not properly define getArea, code won't even compile since then the class does not satisfy the requirements of the interface.

- **Child class (mistake!)**

```
public class Circle extends Ellipse {
   public double getarea() { … }
}
```

This code will compile, but when you call getArea at runtime, you will get version from Ellipse, since there was a typo in this name (lowercase a).

- **Catching mistake at compile time**

```
public class Circle extends Ellipse {
   @Override
   public double getarea() { … }
}
```

This tells the compiler "I think that I am overriding a method from the parent class". If there is no such method in the parent class, code won't compile. If there is such a method in the parent class, then @Override has no effect on the code. Recommended but optional. More on @Override in later sections.

Visibility Modifiers

- **public**
  - A public variable or method can be accessed anywhere an instance of the class is accessible
    - From methods within the class
    - From methods outside that have a variable referring to an instance of that class
- **private**
  - A private variable or method is only accessible from methods within the same class
  - Declare *all* instance variables private
    - Except for constants, which are public static final
      `public static final PI = 3.14…;`
  - Declare methods private if they are not part of class contract and are just internal implementation helpers

Visibility Modifiers (Continued)

- **protected**
  - Protected variables or methods can only be accessed by methods within the class, within classes in the same package, and within subclasses
- **[default]**
  - Default visibility indicates that the variable or method can be accessed by methods within the class, and within classes in the same package
  - A variable or method has default visibility if a modifier is omitted. Rarely used!
    - Most modern developers think that the parent class is an intrinsic part of the definition of a class, but that the package is incidental. So, most developers do not use the default visibility and instead always explicitly say private, public, or protected.

When to Use Which

- **private**
  - Very common; use for all instance variables and internally-used methods
    - Use this as first choice
- **public**
  - Common for methods and constructors, but not for instance variables
    - Second choice
- **protected**
  - Used when two classes are tightly coupled and the child needs access to internals of parent
    - Moderately rare
- **[default]** No modifier
  - Very rare. Don’t omit modifier without good reason.

Other Modifiers

- **final**
  - For a variable: cannot be changed after instantiation
    - Widely used to make immutable classes
  - For a class: cannot be subclassed
  - For a method: cannot be overridden in subclasses
- **synchronized**
  - Sets a lock on a section of code or method
    - Only one thread can access the code at any given time
- **volatile**
  - Guarantees that other threads see changes to variable
- **transient**
  - Indicates that values are not stored in serialized objects
- **native**
  - Indicates that method is implemented using C or C++

Enums

- **Idea**
  - Enums are classes with a fixed number of named instances
    - They do *not* correspond to ints as in C++
- **Syntax**

```
public enum Month { JANUARY, ..., DECEMBER }

public class SomeClass {
  public void someMethod() {
    Month jan = Month.JANUARY;
    doSomethingWith(jan);
  }
}
```

Motivation

- **You want only a fixed number of instances**
  - There can be only 12 months, 7 days, etc.
  - You can also implement singletons (classes for which there is only one instance) this way
- **You want to easily compare instances**

```
Month m = findSomeMonth();
if (m == Month.DECEMBER) { ... }
```

- **You want an automatic toString definition**

```
Month m = Month.DECEMBER;
System.out.println(m);  // Prints DECEMBER
```

Capabilities

- **Enums can have methods**
  - Fixed number of instances, but otherwise they are classes. So, they can have public or private methods, just like other classes.
- **Enums can have instance variables**
  - Same point, but as we discussed, mutable instance variables should always be private
- **Enums can have constructors**
  - Constructors must be private. You call them by putting constructor args in parens after the instance name
    - `public enum Blah { FOO(...), BAR(...) ... }`
- **More info and examples**
  - http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Basics: Enum

```
public enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
   THURSDAY, FRIDAY, SATURDAY;
}
```

Basics: Enum Tester

```
public class DayTest {
  public static boolean isWeekend(Day d) {
    return(d == Day.SATURDAY || d == Day.SUNDAY);
  }

  public static void main(String[] args) {
    System.out.println("Monday is weekend? " +
                       isWeekend(Day.MONDAY));
    System.out.println("Saturday is weekend? " +
                       isWeekend(Day.SATURDAY));
  }
}
```

Output:
Monday is weekend? false
Saturday is weekend? true

Methods: Enum

```
public enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
   THURSDAY, FRIDAY, SATURDAY;

   public boolean isWeekend() {
      return(this == SATURDAY || this == SUNDAY);
   }

   public boolean isWeekday() {
      return(!isWeekend());
   }
}
```

Methods: Enum Tester

```
public class DayTest {
  public static void main(String[] args) {
    System.out.println("Monday is weekend? " +
                       Day.MONDAY.isWeekend());
    System.out.println("Saturday is weekend? " +
                       Day.SATURDAY.isWeekend());
  }
}
```

Output:
Monday is weekend? false
Saturday is weekend? true

Constructors and Instance Vars: Enum

```
public enum Day {
  SUNDAY("Sun"), MONDAY("Mon"), TUESDAY("Tues"),
  WEDNESDAY("Wed"), THURSDAY("Thurs"),
  FRIDAY("Fri"), SATURDAY("Sat");

  private String abbreviation;

  private Day(String abbreviation) {
    this.abbreviation = abbreviation;
  }

  public String getAbbreviation() {
    return(abbreviation);
  }

  // isWeekend and isWeekday methods
}
```

Constructors and Instance Vars: Enum Tester

```
public class DayTest {
  public static void main(String[] args) {
    Day day1 = Day.MONDAY;
    System.out.println(day1.getAbbreviation() +
                       " is weekend? " +
                       day1.isWeekend());
    Day day2 = Day.SATURDAY;
    System.out.println(day2.getAbbreviation() +
                       " is weekend? " +
                       day2.isWeekend());
  }
}
```

Output:
Mon is weekend? false
Sat is weekend? true

JavaDoc Options

Review: Comments

- **Java supports 3 types of comments**
  - // Comment to end of line.
  - /* Block comment containing multiple lines.
    Nesting of comments in not permitted. */
  - /** A JavaDoc comment placed before class
    definition and nonprivate methods.
    Text may contain (most) HTML tags,
    hyperlinks, and JavaDoc tags. */

Review: JavaDoc

- **JavaDoc motivation**
  - Used to generate on-line documentation
- **Building JavaDoc files**
  - From Eclipse
    - Project → Generate Javadoc...
  - From command line

```
> javadoc Foo.java Bar.java
> javadoc *.java
```

- **More details**
  - JavaDoc home page
    - https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

Useful JavaDoc Tags

- **@author**
  - Specifies the author of the document
  - Must use javadoc -author ... to generate in output

```
/** Description of some class ...
 *
 * @author <A HREF="mailto:hall@coreservlets.com">
 *         Marty Hall</A>
 */
```

- **@version**
  - Version number of the document
  - Must use javadoc -version ... to generate in output
- **@param**
  - Documents a method argument
- **@return**
  - Documents the return type of a method

Useful JavaDoc Command-line Arguments

- **-author**
  - Includes author information (omitted by default)
- **-version**
  - Includes version number (omitted by default)
- **-noindex**
  - Tells `javadoc` not to generate a complete index
- **-notree**
  - Tells `javadoc` not to generate the tree.html class hierarchy
- **-link, -linkoffline**
  - Tells `javadoc` where to look to resolve links to other packages

```
-link http://docs.oracle.com/javase/8/docs/api/
-linkoffline c:\jdk1.8\docs\api
```

CLASSPATH

- **Idea**
  - The `CLASSPATH` environment variable defines a list of directories in which to look for classes
    - Default = current directory and system libraries
    - Best practice is to not set this when first learning Java!
- **Setting the CLASSPATH globally**

```
set CLASSPATH = .;C:\java;D:\cwp\echoserver.jar
setenv CLASSPATH .:~/java:/home/cwp/classes/
```

  - The "." indicates the current working directory
- **Supplying a CLASSPATH on the command line**

```
javac -classpath .;D:\cwp  WebClient.java
java -classpath .;D:\cwp WebClient
```

Summary: Most Important Point of Section

- **Interfaces**
  - Let you guarantee that classes will have certain methods
- **Example interface**

```
public interface Shape {
  double getArea();
}
```

  - All classes that claim to be Shapes must define a getArea method
- **Class that uses interface**

```
public class Circle implements Shape {
  public double getArea() {...}
}
```

  - If you forget getArea, code will not compile
- **Other interface features**
  - Interfaces can have static and concrete (default) methods
  - Classes can implement multiple interfaces

Summary: Other Points

- **Abstract classes**
  - In Java 7 and earlier, abstract classes could have concrete methods and interfaces could not. So, abstract classes were common in older code.
    - Now, abstract classes only used if you need instance vars
- **protected and (default) visibility possible**
  - private used the most (always for mutable instance vars)
  - public used second most
  - protected: accessible to subclasses but not outside code
- **@Override**
  - *Always* use when redefining inherited methods
- **Enums**
  - Java classes with fixed number of instances