CH. 2
OBJECT-ORIENTED PROGRAMMING

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
INHERITANCE
MOTIVATIONS

- Suppose you will want to model objects for shapes. Many of the objects will have common features, maybe colors, or the ability to compute their areas, or computing overlap between them. BUT, is there a way to reduce the amount of repeated code? Improve the robustness (correctness) of the model? Design this type of model hierarchy?

- How about an example of allied characters in a game? Some help you by healing, some help offensively, some help defensively. However, all of these types of allies have commonality. So the same questions exist!

- The answer is to use inheritance – modeling types and subtypes in a way that reduces duplicated components.
Inheritance is a type/sub-type relationship (parent/child) denoted with an arrow pointed to the type in a UML diagram.

- A superclass (base class) is the inherited object type.
- A subclass (derived class) is the inheriting object type.
- All of the state (data fields) and behavior (methods) of the superclass is inherited (“handed-down”) to the subclass.
  - The superclass constructors are NOT inherited.
1. public class A {
   2.   private int a;
   3. }
4. public class B extends A {
   5.   private int b;
   6. }

- In Java, the keyword `extends` denotes an inheritance relationship.
- In this example, by inheritance, B is an object whose state is defined by two ints, the one in A and the one in B.
- In this relationship, the superclass is responsible for constructing (initializing) the superclass’s data fields, while the subtype is responsible for the subclass’s data fields.
CONSTRUCTION IN INHERITANCE

• The superclass constructor is not inherited, so how do we construct it’s part of memory?

• They are invoked explicitly (by the programmer) or implicitly (by the Java compiler)

• We use the `super` keyword to invoke explicitly

• The Java compiler will always attempt to invoke the no-arg constructor implicitly

• Caveats:
  • We must use the keyword super, otherwise error
  • It must be the very first line of the constructor, otherwise error

• Explicitely:
  ```java
  public B() {
      super(); //note this is like any constructor, we are free to pass parameters as well!
  }
  ```

• Implicitely:
  ```java
  public B() {
      //java inserts super() – always calling the no-arg constructor
  }
  ```
SUPER

• A reference to the superclass
  • Synonymous to `this`

• Can be used to
  • Call superclass constructor
  • Call methods/data fields of superclass

```java
1. public class A {
2.   int x;
3.   public A(int a) {x = a;}
4.   public void printA() {System.out.print(x);}
5. }
6. public class B extends A {
7.   int y;
8.   public B(int a, int b) {
9.     super(a); //Example of construction
10.    y = b;
11. }
12. public void printB() {
13.    super.printA(); //Example of method invocation
14.    System.out.print("", "+ y);
15. }
16. }
```
DEFINING A SUBCLASS

• A subclass inherits from a superclass. You can also:
  • Add new properties
  • Add new methods
  • Override the methods of the superclass

• Conceptually a subclass represents a smaller set of things, so we make our subclass *more detailed* to model this
A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as **method overriding**.

Note this is different than **method overloading** – two functions named identically with different signatures.
**OVERRIDING**

1. public class Shape {
2.     private Color c;
3.     
4.         /** other parts omitted for brevity */
5.     
6.         public void draw() {
7.             StdDraw.setPenColor(c);
8.         }
9.     }
10. }

1. public class Circle extends Shape {
2.     private double x, y;
3.     private double radius;
4.     
5.         /** other parts omitted for brevity */
6.     
7.         public void draw() {
8.             super.draw();
9.             StdDraw.filledCircle(x, y, radius);
10.         }
11. }
12. }

Circle overrides the implementation of draw
THE JAVA OBJECT CLASS

• Every class in Java is descended from the `java.lang.Object` class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

• Java Object provides for a few basic functions, like `toString()`.

• We will use others as we go.
POLYMORPHISM
POLYMORPHISM

• **Polymorphism** means that a variable of a superclass (supertype) can refer to a subclass (subtype) object

```java
Shape s = new Circle(5);
```

• Under the context of polymorphism, the supertype here is the **declared type** and the subtype is the **actual type**

• Polymorphism implies that an object of a subtype can be used wherever its supertype value is required
WHY WOULD YOU EVER DO THIS?

• Allow types to be defined at runtime, instead of at compile time:

```java
1. Scanner s = new Scanner(System.in);
2. Shape shape = null;
3. String tag = s.next();
4. if (tag.equals("Circle")) {
    //user wants a circle
5.     double r = s.nextDouble();
6.     shape = new Circle(r, Color.red);
7. }
8. else if (tag.equals("Rectangle")) {
    //User wants a rectangle
9.     double w = s.nextDouble(), h = s.nextDouble();
10.    shape = new Rectangle(w, h, Color.red);
11. }
12. System.out.println("Area: " + shape.area()); //works no matter what!
```
WHY WOULD YOU EVER DO THIS?

• Arrays can only store one type

1. Circle[] circles; //all circles
2. Rectangle[] rects; //all rectangles
3. Shape[] shapes; //depends on subtypes! Can have some circles and some rectangles.
WHY WOULD YOU EVER DO THIS?

• Lets say we have an array of Shape shapes then we can do something like:

```
1. double total = 0;
2. for (int i = 0; i < shapes.length; ++i)
3.    total += shapes[i].area(); //Uses specific
4.  //instance’s function
5. return total;
```
POLYMORPHISM DEMO

1. public class PolymorphismDemo {
2.   public static void main(String[] args) {
3.     m(new Student());
4.     m(new Person());
5.     m(new Object());
6.   }
7.   public static void m(Object x) {
8.     System.out.println(x.toString());
9.   }
10. }
11. }
12. class Student extends Person {
13.   public String toString() {
14.     return "Student";
15.   }
16. }
17. class Person {
18.   public String toString() {
19.     return "Person";
20.   }
21. }
22. }

- Method m takes a parameter of the Object type. You can invoke it with any object.
- When the method m(Object x) is executed, the argument x's toString method is invoked. Classes Student, Person, and Object have their own implementation of the toString method.
- The correct implementation is dynamically determined by the Java Virtual Machine. This is called dynamic binding.
- Polymorphism allows superclass methods to be used generically for a wide range of object arguments (any possible subclass). This is known as generic programming.
POLYMORPHISM AND TYPE CONVERSION

• So when assigning a value of a subtype to a variable of a supertype, the conversion is implicit:
  \[
  \text{Shape } s = \text{new Circle}(5); \quad //\text{implicit conversion from Circle to Shape}
  \]
  This is called **upcasting**.

• When going from a supertype value to a subtype variable, the conversion must be explicit:
  \[
  \text{Circle } c = (\text{Circle})s; \quad //\text{explicit conversion from Shape to circle}
  \]
  This is called **downcasting**. This type of casting might not always succeed, why?
THE INSTANCEOF OPERATOR

• Use the `instanceof` operator to test whether an object is an instance of a class:

```java
1. Object myObject = new Circle();
2. /** Perform downcasting only if myObject is an instance of Circle */
3. if (myObject instanceof Circle) {
4.   System.out.println("The circle diameter is " +
5.     ((Circle)myObject).getDiameter());
6. }
```
The `equals()` method compares the contents of two objects. The default implementation of the equals method in the `Object` class is as follows:

```java
public boolean equals(Object obj) {
    return this == obj;
}
```

What is the problem? How do we fix it?
- `==` for objects compares their memory addresses, not their values.

As an example of overriding the method for our `Circle`:

```java
public boolean equals(Object o) {
    if (o instanceof Circle) {
        return radius == ((Circle)o).radius;
    } else {
        return false;
    }
}
```
• Extend your account model to add two subtypes: one for a checking account and one for a savings account.
  • For the method withdraw in your account super class, override its functionality in checkings and savings.
  • Override the toString() method for the accounts

• Exemplify polymorphism in a main program that allows a user to create an account and make deposits/withdrawals from it
ADVANCED CONCEPTS OF INHERITANCE
THE PROTECTED VISIBILITY (SCOPE) MODIFIER

• The **protected** modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or *its subclasses*, even if the subclasses are in a different package.

Visibility Increases

*private*, none (if no modifier is used), **protected**, **public**
### ACCESSIBILITY SUMMARY

<table>
<thead>
<tr>
<th>Modifier on members in a class</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>default</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
VISIBILITY MODIFIERS FULL EXAMPLE

```java
package p1;

public class C1 {
    public int x;
    protected int y;
    int z;
    private int u;

    protected void m() {
    }
}

public class C2 {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    can access o.z;
    cannot access o.u;
    can invoke o.m();
}

public class C3 extends C1 {
    can access x;
    can access y;
    can access z;
    cannot access u;
    can invoke m();
}

package p2;

public class C4 extends C1 {
    can access x;
    can access y;
    cannot access z;
    cannot access u;
    can invoke m();
}

public class C5 {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;
    cannot access o.u;
    cannot invoke o.m();
}
```
A SUBCLASS CANNOT WEAKEN THE ACCESSIBILITY

• A subclass may override a protected method in its superclass and change its visibility to public.

• However, a subclass cannot "weaken" the accessibility of a method defined in the superclass.
  • For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.
The final modifier, introduced with variables to define constants, e.g., PI, has extended meaning in the context of inheritance:

- A final class cannot be extended:
  ```java
class final Math {
  ...
}
```

- The final method cannot be overridden by its subclasses:
  ```java
public final double getArea() {
  return Math.PI*radius*radius;
}
```
ABSTRACT DATA TYPES

• Abstraction is to distill a system to its most fundamental parts.

• An abstract data type (ADT) is a model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations.
  • This would essentially be the “public interface” of a class

• An ADT specifies what each operation does, but not how it does it
  • Lets repeat, an ADT is the operations not the implementation!
  • We will see that we can implement ADTs in many, many ways
The main structural element in Java that enforces an application programming interface (API) is an interface. An interface is a collection of method declarations with no data and no bodies. They contain only constants and abstract methods (almost like a purely abstract class).

Interfaces do not have constructors and they cannot be directly instantiated.

When a class implements an interface, it must implement all of the methods declared in the interface.

An abstract class also cannot be instantiated, but it can define one or more common methods that all implementations of the abstraction will have.
 INTERFACE EXAMPLE

1. public interface Robot {
2.   void sense(World w);
3.   void plan();
4.   void act(World w);
5.}

USE IMPLEMENTS TO ENFORCE THE INTERFACE

1. public class Roomba implements Robot {
2.   /* code specific to Roomba */
3.   public void sense(World w) { /* Roomba's don't sense */}
4.   public void plan() { /* code for Roomba's actions */}
5.   public void act(World w) { /* code to power motors */}
6.   /* code specific to Roomba */
7. }
ENFORCES?

• In this context, enforce means required by the compiler. A good example is sorting algorithms. In order to sort two things must be comparable. So Java offers a comparable interface so that your own objects can fit into this framework.

• Interfaces are also an example of inheritance. They are "weakly" inherited by implementing classes. So the rules of polymorphism also apply, i.e., an object can be converted to a variable of the interface:

```java
Robot r = new Roomba();
```
SOME INTERESTING POINTS ON ABSTRACT

• An abstract method cannot be contained in a non abstract class
• If a subclass of an abstract superclass does not implement all of the abstract methods, then it must also be declared as abstract
• Cannot use new on an abstract type, but constructors can be defined (for use with super). Also can still use the abstract type for polymorphism!
• An abstract class does not require abstract methods
• A subclass can be abstract even if the superclass is concrete (non abstract)
INTERFACES
THE DETAILS

• Cannot have constructors

• All variables must be `public static final`

• All methods must be `public abstract`

• The last two points imply you don't need to specify any modifiers at all

• Interfaces commonly used as a weaker is-a relationship, specifically is-kind-of referring to possessing certain properties only. Java allows multiple inheritance because of this

• Interfaces can `extend` other interfaces