Polymorphism

- The word **polymorphism** literally means “many forms.”
- Java allows 4 types of polymorphism:
  - coercion (casting)
  - ad hoc polymorphism (overloading)
  - subtype polymorphism
  - parametric polymorphism (**generics**)
- Modeling polymorphism in a programming language lets you create a **uniform** interface to different kinds of operands, arguments, and objects.

---

Example: Uniform Interface

```java
class Student {
    void doMyWork() { /* Do not know the detail. */}
}

class HighSchoolStudent extends Student {
    void writeHomework() {
        System.out.println("Write homework orz");
    }

    void doMyWork() { writeHomework(); }
}

class CollegeStudent extends Student {
    void writeReports() {
        System.out.println("Write reports qq");
    }

    void doMyWork() { writeReports(); }
}
```
public class PolymorphismDemo {

    public static void main(String[] args) {
        HighSchoolStudent h = new HighSchoolStudent();
        goStudy(h);
        CollegeStudent c = new CollegeStudent();
        goStudy(c);
    }

    // uniform interface, multiple implementations
    // for future extension (scalability)
    static void goStudy(Student s) {
        s.doMyWork();
    }

    /* no need to write these methods
    static void goStudy(HighSchoolStudent s) {
        s.writeHomework();
    }
    
    static void goStudy(CollegeStudent s) {
        s.writeReports();
    }
    */
Subtype Polymorphism

- For convenience, let $U$ be a subtype of $T$.
- **Liskov Substitution Principle** states that $T$-type objects may be replaced with $U$-type objects without altering any of the desirable properties of $T$ (correctness, task performed, etc.).\(^2,^3\)

---

\(^2\)See

\(^3\)Also see
Casting

- **Upcasting** (widening conversion) is to cast the \textbf{U} object to the \textbf{T} variable.

\begin{verbatim}
1 T t = new U();
\end{verbatim}

- **Downcasting** (narrow conversion) is to cast the \textbf{T} variable to a \textbf{U} variable.

\begin{verbatim}
1 U u = (U) t; // t is T variable reference to a U object.
\end{verbatim}

- Upcasting is always allowed, but downcasting is allowed only when a \textbf{U} object is passed to the \textbf{U}-type variable.
  - Java type system makes sure that the referenced object provides services adequate for \textbf{T} type.
• However, type-checking in compilation time is unsound.
• The operator `instanceof` checks if an object reference is an instance of a type, and returns a boolean value.
Example

```java
class T {}
class U extends T {}

public class InstanceofDemo {
    public static void main(String[] args) {
        T t1 = new T();

        System.out.println(t1 instanceof U); // output false
        System.out.println(t1 instanceof T); // output true

        T t2 = new U(); // upcasting

        System.out.println(t2 instanceof U); // output true
        System.out.println(t2 instanceof T); // output true

        U u = (U) t2; // downcasting; this is ok.

        u = (U) new T(); // pass the compilation; fail during execution!
    }
}
```
Abstraction, Method Overriding, and Polymorphism

• JVM invokes the appropriate method for the current object by looking up from the bottom of the class hierarchy to the top.
  • These methods are also called virtual methods.

• This preserves the behaviors of the subtype objects and the super-type variables play the role of placeholder.

• We often manipulate objects in an abstract level; we don’t need to know the details when we use them.
  • For example, computers, cellphones, driving.
Exercise

- Imagine that we have a zoo with some animals.

```java
class Animal {
    void speak() {}
}
class Dog extends Animal {
    void speak() { System.out.println("woof"); }
}
class Cat extends Animal {
    void speak() { System.out.println("meow"); }
}
class Bird extends Animal {
    void speak() { System.out.println("tweet"); }
}

class PolymorphismDemo {
    public static void main(String[] args) {
        Animal[] zoo = {new Dog(), new Cat(), new Bird()};
        for (Animal a: zoo) a.speak();
    }
}
```
A final variable is a variable which can be initialized once and cannot be changed later.

- The compiler makes sure that you can do it only once.
- A final variable is often declared with static keyword and treated as a constant, for example, `Math.PI`.

A final method is a method which cannot be overridden by subclasses.

- You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.

A class that is declared final cannot be inherited.
Abstract Classes

- An abstract class is a class declared **abstract**.
- The classes that sit at the top of an object hierarchy are typically **abstract** classes.\(^4\)
- These **abstract** class may or may not have **abstract** methods, which are methods declared **without implementation**.
  - More explicitly, the methods are declared without braces, and followed by a semicolon.
  - If a class has one or more **abstract** methods, then the class itself must be declared **abstract**.
- All **abstract** classes cannot be instantiated.
- Moreover, **abstract** classes act as placeholders for the subclass objects.

\(^4\)The classes that sit near the bottom of the hierarchy are called **concrete** classes.
Abstract methods and classes are in italic.

In this example, the abstract method `draw()` and `resize()` should be implemented depending on the real shape.
• Not all classes share a vertical relationship.
• Instead, some are supposed to perform the specific methods without a vertical relationship.
  • Consider the class Bird inherited from Animal and Airplane inherited from Transportation.
  • Both Bird and Airplane are able to be in the sky.
  • So they should perform the method canFly(), for example.
• By semantics, the method canFly() could not be defined in their superclasses.
• We need a horizontal relationship.
Example

```java
interface Flyable {
    void fly(); // implicitly public, abstract
}

class Animal {}

class Bird extends Animal implements Flyable {
    void flyByFlappingWings() {
        System.out.println("flapping wings");
    }

    public void fly() { flyByFlappingWings(); }
}

class Transportation {}

class Airplane extends Transportation implements Flyable {
    void flyByMagic() {
        System.out.println("flying with magicssssss");
    }

    public void fly() { flyByMagic(); }
}
```
how planes fly

air

magic

air

some more magic

very important magic

air

magic
public class InterfaceDemo {
    public static void main(String[] args) {
        Bird b = new Bird();
        goFly(b);

        Airplane a = new Airplane();
        goFly(a);
    }

    static void goFly(Flyable f) {
        f.fly();
    }
}
Interfaces

• An interface forms a contract between the object and the outside world.
  • For example, the buttons on the television set are the interface between you and the electrical wiring on the other side of its plastic casing.

• An interface is also a reference type, just like classes, in which only method signatures are defined.

• So they can be the types of reference variables!
• Note that interfaces cannot be instantiated (directly).
• A class implements one or multiple interfaces by providing method bodies for each predefined signature.
• This requires an object providing a different set of services.
• For example, combatants in RPG can also buy and sell stuffs in the market.
Example

interface Driveable
   startEngine()
   stopEngine()
   accelerate()
   turn()

class Automobile implements Driveable
   startEngine()
   stopEngine()
   accelerate()
   turn()
   honkHorn()
   ...

class Lawnmower implements Driveable
   startEngine()
   stopEngine()
   accelerate()
   turn()

   cutGrass()
   ...

Properties of Interfaces

- The methods of an interface are implicitly public.
- In most cases, the class which implements the interface should implement all the methods defined in the interface.
  - Otherwise, the class should be abstract.
- An interface can declare only fields which are static and final.
- You can also define static methods in the interface.
- An interface can extend another interface, just like a class which can extend another class.
  - In contrast with classes, an interface can extend many interfaces.
• Common interfaces are **Runnable**\(^5\) and **Serializable**\(^6\).

• A new feature since Java SE 8 allows to define the methods with implementation in the interface.
  - A method with implementation in the interface is declared **default**.

\(^5\)See Java Multithread.

\(^6\)Used for an object which can be represented as a sequence of bytes. This is called object serialization.
Timing for Interfaces and Abstract Classes

- Consider using abstract classes if you want to:
  - share code among several closely related classes
  - declare non-static or non-final fields
- Consider using interfaces for any of situations as follows:
  - unrelated classes would implement your interface
  - specify the behavior of a particular data type, but not concerned about who implements its behavior
  - take advantage of multiple inheritance
- Program to interface, not to implementation.\(^7\)

\(^7\)See software engineering or object-oriented analysis and design.
Wrapper Classes

• To treat values as objects, Java supplies standard wrapper classes for each primitive type.

• For example, you can construct a wrapper object from a primitive value or from a string representation of the value.

```java
... Double pi = new Double("3.14");
...
```
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>java.lang.Void</td>
</tr>
<tr>
<td>boolean</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>char</td>
<td>java.lang.Character</td>
</tr>
<tr>
<td>byte</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>short</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>int</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>long</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>float</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>double</td>
<td>java.lang.Double</td>
</tr>
</tbody>
</table>
Autoboxing and Unboxing of Primitives

- The Java compiler automatically wraps the primitives in their wrapper types, and unwraps them where appropriate.

```java
... Integer i = 1; // autoboxing
Integer j = 2;
Integer k = i + 1; // autounboxing and then autoboxing
System.out.println(k); // output 2

System.out.println(k == j); // output true
System.out.println(k.equals(j)); // output true
...
```

- The method `equals()` inherited from `Object` is used to compare two objects.
  - You may override this method if necessary.
Immutable Objects

- An object is considered immutable if its state cannot change after it is constructed.
- Often used for value objects.
- Imagine that there is a pool for immutable objects.
- After the value object is first created, this value object is reused if needed.
- This implies that another object is created when we operate on the immutable object.
For example,

```java
...  
k = new Integer(1);
System.out.println(i == k); // output false (why?)
System.out.println(k.equals(i)); // output true
...  
```

- Good practice when it comes to concurrent programming.\(^8\)
- Another example is String objects.

enum Types

- An `enum` type is an reference type limited to an explicit set of values.
- An order among these values is defined by their order of declaration.
- There exists a correspondence with string names identical to the name declared.

9 The keyword `enum` is a shorthand for enumeration.
Example: Colors

```java
enum Color {
    RED, GREEN, BLUE; // three options

    static Color random() {
        Color[] colors = values();
        return colors[(int) (Math.random() * colors.length)];
    }
}
```

- Note that **Color** is indeed a subclass of **enum** type with 3 **static** and **final** references to 3 Color objects corresponding to the enumerated values.
- This mechanism enhances type safety and makes the source code more readable!
Class Pen {
    Color color;
    Pen(Color color) { this.color = color; }
}

Class Clothes {
    Color color;
    T_Shirt(Color color) { this.color = color; }
    void setColor(Color new_color) { this.color = new_color; }
}

public class EnumDemo {
    public static void main(String[] args) {
        Pen crayon = new Pen(Color.RED);
        Clothes T_shirt = new Clothes(Color.random());
        System.out.println(crayon.color == T_shirt.color);
    }
}
enum Direction {UP, DOWN, LEFT, RIGHT}

/** equivalence */
class Direction {
    final static Direction UP = new Direction("UP");
    final static Direction DOWN = new Direction("DOWN");
    final static Direction LEFT = new Direction("LEFT");
    final static Direction RIGHT = new Direction("RIGHT");

    private final String name;

    static Direction[] values() {
        return new Direction[] {UP, DOWN, LEFT, RIGHT};
    }

    private Direction(String str) {
        this.name = str;
    }
}
*/