Why OOP?

• First, you may know that there are many programming paradigms.¹
• OOP is the solid foundation of modern (large-scale) software design.
• In particular, great reuse mechanism and abstraction are realized by:
  • Encapsulation isolates the internals (private members) from the externals, fulfilling the abstraction and providing the sufficient accessibility (public methods).
  • Inheritance provides method overriding w/o changing the method signature.²
  • Polymorphism exploits the superclass as a placeholder to manipulate the implementations (sub-type objects).

¹See https://en.wikipedia.org/wiki/Programming_paradigm.
²This leads to the need of “single interface” as mentioned before.
code reuse

generality

abstraction

generics

type parameters

variables

application programming interface (API)

inheritance
+ method overriding
+ subtype polymorphism

abstract class & interface as user interface; subclass as implementation
• This leads to the production of frameworks\(^3\), which actually do most of the job, leaving the (application) programmer only with the job of customizing with business logic rules and providing hooks into it.

• This greatly reduces programming time and makes feasible the creation of larger and larger systems.

• In analog, 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.

\(^3\)See https://spring.io/.
```java
class Animal {
    /* ignore the previous part */
    void speak() {}
}

class Dog extends Animal {
    /* ignore the previous part */
    @Override
    void speak() { System.out.println("woof"); }
}

class Cat extends Animal {
    /* ignore the previous part */
    @Override
    void speak() { System.out.println("meow"); }
}

class Bird extends Animal {
    /* ignore the previous part */
    @Override
    void speak() { System.out.println("tweet"); }
}
```
public class PolymorphismDemo {

    public static void main(String[] args) {

        Animal[] animals = {new Dog(), new Cat(), new Bird()};
        for (Animal each : animals)
            each.speak();

    }

}
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.).

In other words, the references are clients asking the objects (right-hand side) for services!

---

$^4$See https://en.wikipedia.org/wiki/Liskov_substitution_principle.

$^5$Also see https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).
Casting

• **Upcasting** (widening conversion) is to cast the `U` object/variable to the `T` variable.

```
1 U u1 = new U(); // trivial
2 T t1 = u1; // ok
3 T t2 = new U(); // ok
```

• **Downcasting** (narrow conversion) is to cast the `T` variable to a `U` variable.

```
1 U u2 = (U) t2; // ok, but dangerous. why?
2 U u3 = new T(); // error! why?
```
Solution: instanceof

• Upcasting is always allowed, but **downcasting is not always true even when you use the cast operator.**
  • In fact, type checking at compile time is unsound just because the cast operator violets the functionality of type checking.
• Moreover, **T**-type reference can also point to the siblings of **U**-type.
  • Recall that **T**-type is used as the placeholder.
• **We can use instanceof** to check if the referenced object is of the target type **at runtime.**
The class inheritance can be represented by a digraph (directed graph).

For example, D is the subtype of A and B, which are both reachable from D on this digraph.
class A {}
class B extends A {}
class C extends A {}
class D extends B {}
class E extends B {}
class F extends D {}

public class InstanceofDemo {

    public static void main(String[] args) {

        Object o = new D();

        System.out.println(o instanceof A); // output true
        System.out.println(o instanceof B); // output true
        System.out.println(o instanceof C); // output false
        System.out.println(o instanceof D); // output true
        System.out.println(o instanceof E); // output false
        System.out.println(o instanceof F); // output false
    }
}

final

- 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.
  - For example, again, Math.
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.\(^6\)
- 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.

\(^6\)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.
Another IS-A Relationship: Interface

- In some situations, objects are supposed to work together without a vertical relationship.
  - Consider the class Bird inherited from Animal and Airplane inherited from Transportation.
  - Both Bird and Airplane are able to fly in the sky.
  - Let’s call the method fly(), for example.

- By semantics, the method fly() could not be defined in their superclasses. (Why?)

- Similar to the case study of Student, we wish those flyable objects go flying but in a single interface.
  - Using Object as the placeholder?

- Clearly, we need a horizontal relationship.
interface Flyable {
    void fly(); // implicitly public and abstract
}

class Animal {}

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

class Transportation {}

class Airplane extends Transportation implements Flyable {
    void flyByMagic() {
        System.out.println("flying with magicssssss");
    }
    @Override
    public void fly() { flyByMagic(); }
}
how planes fly

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);
   }

   public static void goFly(Flyable f) {
      f.fly();
   }
}
• An interface is a **contract** between the object and the client.

• As shown, **an interface is a reference type, just like classes**.

• Unlike classes, interfaces are used to define methods w/o implementation so that they **cannot** be instantiated (directly).

• A class could implements **one or multiple** interfaces by providing method bodies for each predefined signature.
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()
...
**Interfaces (2/2)**

- An interface can extend another interfaces.
  - Like a collection of contracts, in some sense.
- For example, **Runnable**\(^7\) and **Serializable**\(^8\) are two of Java interfaces.
- In JDK8, we have new features as follows:
  - we can declare **static** fields\(^9\) and methods in the interfaces;
  - we can also define **default** methods in the interfaces;
  - Java provides so-called **functional interfaces** for **lambdas** which are widely used in **the stream framework**. (Stay tuned in Java 2!)

---

\(^7\)See Java Multithread.

\(^8\)Used for an object which can be represented as a sequence of bytes. This is called object serialization.

\(^9\)But they should be **final**.
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
## Special Issue: Wrapper Classes

<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

Integer m = new Integer(i);
System.out.println(m == i); // output false?
System.out.println(m.equals(i)); // output true!?
...
```
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.
  - Another example is String objects.
- Good practice when it comes to concurrent programming.\(^\text{10}\)

String str1 = "NTU";
String str2 = "ntu";

System.out.println("str1 = " + str1.toLowerCase());
System.out.println("str1 = " + str1);
str1 = str1.toLowerCase();
System.out.println("str1 = " + str1);
System.out.println(str1 == str2); // output false?!
System.out.println(str1.intern() == str2); // true!

...
Special Issue: 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.

\[^{11}\text{The keyword enum is a shorthand for enumeration.}\]
```java
class Color {
    RED, GREEN, BLUE; // three options

double[] 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;
    }
}
*/
## Package and Access Control

<table>
<thead>
<tr>
<th>Scope</th>
<th>Modifier</th>
<th>private</th>
<th>(package)</th>
<th>protected</th>
<th>public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within the class</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Within the package</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Inherited classes</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Out of package</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Special Issue: Nested Classes

- A nested class is a member of its enclosing class.
- **Non-static** nested classes have access to other members of the enclosing class, even if they are declared `private`.
- Instead, **static** nested classes do not have access to other instance members of the enclosing class.
- We use nested classes when it needs to
  - logically group classes that are only used in one place
  - increase encapsulation
  - lead to more readable and maintainable code
Family of Nested Classes

- Nested classes
  - Inner classes
    - Inner classes
    - Method local Inner classes
    - Anonymous Inner classes
  - Static Nested classes
Example: Inner Class

```java
class OuterClass {

    private int x = 1;
    private InnerClass innerObject = new InnerClass();

    class InnerClass {
        public void print() {
            System.out.println(x); // ok!
        }
    }

    void doSomeAction() { innerObject.print(); }
}

public class InnerClassDemo {

    public static void main(String[] args) {
        new OuterClass().doSomeAction(); // output 1

        new InnerClass(); // you cannot do this
    }
}
```
Example: Method-Local Inner Class

class OuterClass {

    void doSomething() {
        class LocalClass { // should be in the beginning
            private int x = 2;
            void print() { System.out.println(x); }
        }

        new LocalClass().print(); // output 1 and 2
    }
}

Anonymous Class

- Anonymous (inner) classes are an extension of the syntax of the `new` operation, enabling you to declare and instantiate a class at the same time.
- Use them when you need to use these types only once.
abstract class Button {
    abstract void onClicked();
}

public class AnonymousClassDemoOne {

    public static void main(String[] args) {

        Button ok_button = new Button() {
            @Override
            public void onClicked() {
                System.out.println("OK");
            }
        };

        ok_button.onClicked();
    }
}
An interface can be used to instantiate an object indirectly by anonymous classes with implementing the abstract methods.
Another Example: Iterators

- An important use of inner classes is to define an adapter class as a helper object.
- Using adapter classes, we can write classes more naturally, without having to anticipate every conceivable user’s needs in advance.
- Instead, you provide adapter classes that marry your class to a particular interface.
- For example, an iterator is a simple and standard interface to enumerate elements in data structures.
  - The class which implements the interface \texttt{Iterable} has the responsibility to provide an iterator.
  - An iterator is defined in the interface \texttt{Iterator} with two uninplemented methods: \texttt{hasNext()} and \texttt{next()}. 
Example

```java
import java.util.Iterator;

class Box implements Iterable<Integer> { // <...>: generics

    int[] items = {10, 20, 30};

    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            private int ptr = 0;

            public boolean hasNext() {
                return ptr < items.length;
            }

            public Integer next() {
                return items[ptr++];
            }
        };
    }
}
```
public class IteratorDemo {
    public static void main(String[] args) {
        Box myBox = new Box();

        // for-each loop
        for (Integer item: myBox) {
            System.out.println(item);
        }

        // equivalence
        Iterator iterOfMyBox = myBox.iterator();
        while (iterOfMyBox.hasNext())
            System.out.println(iterOfMyBox.next());
    }
}
Static Nested Class

• A static inner class is a nested class declared static.
  • Similar to the static members, they can access to other static
    members without instantiating the outer class.
  • Also, a static nested class does not have access to the instance
    members of the outer class.
• In particular, the static nested class can be instantiated
  directly, without instantiating the outer class object first.
  • Static nested classes act something like a minipackage.
```java
class OuterClass {
    static int x = 1;
    private int y = 2;

    static class StaticClass {
        private int z = 3;
        void doSomething() {
            System.out.println(x);
            System.out.println(y); // you cannot do this
            System.out.println(z);
        }
    }
}

public class StaticNestedClassDemo {
    public static void main(String[] args) {
        new OuterClass.StaticClass().doSomething();
    }
}
```
Fin.