Prepared by Linan Qiu <lq2137@columbia.edu>

# Inheritance and Interface

Recall the first principle. This is our first encounter. **Inheritance** and **Interfaces** have to sole purpose of making your life easy (ie. allowing you to be lazy).

## Inheritance

Let's say you have a class like this:

```
// Person.java

public class Person {
  public String name;

  public Person(String name) {
    this.name = name;
  }

  public void sayName() {
    System.out.println(name);
  }
}
```

Now in your test class, you have:

```
// Test.java

public class Test {
  public static void main(String[] args) {
    Person tom = new Person("Tom");
    tom.sayName(); // should print Tom
  }
}
```

This is all fine. Now let's say you want to havea class called `Student`. You can do this:

```
// Student.java
```

```java
public class Student {
    public String name;
    public int year;

    public Student(String name) {
        this.name = name;
    }

    public void sayName() {
        System.out.println(name);
    }

    public void sayYear() {
        System.out.println(year);
    }
}
```

This is bad. We find that there's a lot of repeat between the code in `Person` and `Student`. Even more importantly, the two classes have **is-a** relationship. A student is a person! In this case, a student should do every a person does. A person may not do everything a student does. Hence, if a code modifies the functionality of `Person` in, say the following way:

```java
// in Person.java

public void sayName() {
  System.out.println(">>> " + name);
}
```

then the same behavior should be seen in `Student`. However, we'd have to edit the code manually if we have `Student` as it is now. So that's bad. That's not being lazy. Instead, we can make `Student` inherit `Person`.

```java
// Student.java

public class Student extends Person {
  public int year;

  public Student(String name) {
    super(name);
  }

  public void sayYear() {
    System.out.println(year);
  }
}
```

Notice that now we are no longer doing duplicate work in `Student`. We say that `Student` is a subclass of `Person`. We can take this even further!

```
// TA.java

public class TA extends Student {
    public boolean isHeadTA;

    public TA(String name) {
        super(name);
    }

    public void sayIsHeadTA() {
        if (isHeadTA) {
            System.out.println("I'm a Head TA!");
        } else {
            System.out.println("I'm not a Head TA!");
        }
    }
}

// Professor.java

public class Professor extends Person {
    public boolean teachesDataStructures;

    public Professor(String name) {
        super(name);
    }

    public void sayTeachesDataStructures() {
        if (teachesDataStructures) {
            System.out.println("I'm awesome.");
        } else {
            System.out.println("I'm slightly less awesome.");
        }
    }
}
```

`Student` is a `Person`, `Professor` is a `Person` and `TA` is a `Student` and hence is also a `Person`. Visualize this relationship:

```
        Person
        /    \
       /      \
```

```
   Student  Professor
     /
    /
  TA
```

Then, let's say you create a method like this in `Test.java`:

```
public static boolean compareName(Person a, Person b) {
    return a.compareTo(b);
}
```

Then, we can do:

```
// Test.java

public class Test {
  public static int compareName(Person a, Person b) {
      return a.name.compareTo(b.name);
  }

  public static void main(String[] args) {
    Person tom = new Person("Tom");
    TA linan = new TA("Linan");
    System.out.println(compareName(tom, linan));
  }
}
```

Because `TA` is a `Person`. We can do the same for `Professor` and `Student`.

In fact, instead of declaring `TA linan = new TA("Linan")`, we can declare `Person linan = new TA("Linan")`. However, we just won't be able to call the `TA` specific methods / variables from `linan`. We'll only be able to use it as if it was a `Person`.

This is not meant to be a comprehensive textbook on inheritance. Instead, it serves to remind you **why** we do inheritance and under what circumstances. Why? Because we're lazy. Under what circumstance? When something exhibits the **is-a** relationship.

Now there's a special thing called an **abstract class**. Let's say that you decide that `Person` is too abstract a concept. After all, everyone in this class is either a student, a TA, or a professor. So you don't really want anyone to go around using `Person`. You can declare `Person` an abstract class. Then, you won't be able to instantiate `Person` directly. That is:

```java
// in some method far far away

Person linan1 = new Person("Linan"); // compiler screams

TA linan2 = new TA("Linan"); // compiler is cool
Person linan3 = new TA("Linan"); // compiler is still cool
Student linan4 = new TA("Linan"); // compiler is also cool
Person linan5 = new Student("Linan"); // compile is still cool
```

There's something else you can do with **abstract classes**. That is to be even lazier: leave the implementation of methods to the next guy. Let's say you have the following classes:

```java
// Phone.java

public abstract class Phone {
  public abstract void accessAppStore();
}

// IPhone.java

public class IPhone extends Phone {
  public void accessAppStore() {
    System.out.println("Going to Apple Store");
  }
}

// Galaxy.java

public class Galaxy extends Phone {
  public void accessAppStore() {
    System.out.println("Going to Google Play");
  }
}
```

Essentially, you are leaving a method empty and inviting the subclasses to implement them. The subclasses have to implement them or they will get a compiler error. However, they can also leave the methods as abstract. **If a class has one or more methods abstract, it itself is an abstract class**. This makes sense, since you can't instantiate a class with an abstract method; it doesn't make sense to call an undefined method right? say `Phone aPhone = new Phone();` and we do `aPhone.accessAppStore()`. Does it go to the Apple Store or Google Play? Or do something else? Hence, you will get a compiler error.

## Interfaces

Interfaces represent a **can-do** relationship. For example, what can a Jedi do? A Jedi can, among many things, use a lightsaber, use the force, and wear bathrobes and still look cool. Then, in the Java world, we'd make a `Jedi` class implement `LightsaberUser`, `ForceUser`, `BathrobeUser` interfaces. Or take another example: cars.

SUVs can be driven. SUVs can also be refueled. So to represent this relationship, we create 2 interfaces: `Drivable`, `Refuelable`. These interfaces are **contracts**: it specifies what any class implementing it should be able to do.

```
// Driveable.java

public interface Driveable {
  public void accelerate();
  public void turnLeft();
  public void turnRight();
  public int getSpeed();
}

// Refuelable.java

public interface Refuelable {
  public void addPetrol();
  public int getPetrolLevel();
}
```

The methods are empty! This is because the interfaces only tell you what the classes implementing them **should** do, **not how** they should do it. This makes sense, because an SUV can be driveable and refuelable just as a coupe is driveable and refuelable, but they operate entirely differently internally. However, to a driver, all that matters is that it is driveable and refuelable. They can be used in pretty much the same ways. That's essentially the idea behind interfaces.

```
// SUV.java

public class SUV implements Driveable, Refuelable {
  public void accelerate() {
    // do some accelerate thing...
    // ...
  }

  public void turnLeft() {
    // do some turning thing...
```

```
    // ...
  }

  public void turnRight() {
    // turns right...
    // ...
  }

  public int getSpeed() {
    // gets the speed of the car
  }

  public void addPetrol() {
    // adds some petrol...
    // ...
  }

  public int getPetrolLevel() {
    // gets petrol level
  }
}

// Coupe.java

public class Coupe implements Driveable, Refuelable {
  public void accelerate() {
    // do some other accelerate thing...
    // ...
  }

  public void turnLeft() {
    // do some other turning thing...
    // ...
  }

  public void turnRight() {
    // turns right in some other way...
    // ...
  }

  public int getSpeed() {
    // gets the speed of the car
  }

  public void addPetrol() {
    // adds some petrol...
```

```
    // ...
  }

  public int getPetrolLevel() {
    // gets petrol level
  }
}
```

Similar to inheritance, I can create methods like this:

```
// in some class far far away

public void autoPilot(Driveable car) {
  for(int i = 0; i < 100000; i++) {
    // yeah not a good idea at all
    car.accelerate();
  }
}

public void autoRefuel(Refuelable car) {
  while(car.getPetrolLevel() < 100) {
    car.addPetrol();
  }
}
```

Then, I can call these methods on instances of both SUV and Coupe.

```
// in some method far far away

SUV aSuv = new SUV();
autoPilot(aSuv);
autoRefuel(aSuv);
```

## Inheritance vs Interface

When do we use inheritance and interfaces? Whenever we want to be lazy. But when to use which one?

- Inheritance: **is-a** relationship. For example, a student is a person. A toyota is a car. Then, `Student extends Person` and `Toyota extends Car`
- Interface: **can-do** relationship. For example, a TA can code. A toyota can turn. Then, `TA implements Codeable` and `Toyota implements Turnable`