Unit 18: Interface

After taking this unit, students should:

- understand interface as a type for modeling "can do" behavior
- understand the subtype-supertype relationship between a class and its interfaces

Modeling Behavior

We have seen how we can write our program using superclasses (including abstract ones) to make our code more general and flexible. In this unit, we will kick this up one more notch and try to write something even more general, through another abstraction.

Let's reexamine this method again:

```java
// version 0.3
double findLargest(Shape[] array) {
  double maxArea = 0;
  for (Shape curr : array) {
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
    }
  }
  return maxArea;
}
```

Note that all that is required for this method to work, is that the type of objects in `array` supports a `getArea` method. While `Shape` that we defined in the previous unit meets this requirement, it does not have to be. We could pass in an array of countries or an array of HDB flats. It is unnatural to model a `Country` or a `Flat` as a subclass of `Shape` (recall inheritance models the IS-A relationship).

To resolve this, we will look at an abstraction that models what can an entity do, possibly across different class hierarchies.

Interface

The abstraction to do this is called an *interface*. An interface is also a type and is declared with the keyword `interface`.

Since an interface models what an entity can do, the name usually ends with the -able suffix[1] .

Suppose we want to create a type that supports the `getArea()` method, be it a shape, a geographical region, or a real estate property. Let's call it `GetAreable`:

```java
interface GetAreable {
  public abstract double getArea();
}
```

All methods declared in an interface are `public abstract` by default. We could also just write:

```java
interface GetAreable {
  double getArea();
}
```

Now, for every class that we wish to be able to call `getArea()` on, we tell Java that the class `implements` that particular interface.

For instance,

```java
abstract class Shape implements GetAreable {
  private int numOfAxesOfSymmetry ;

  public boolean isSymmetric() {
    return numOfAxesOfSymmetry > 0;
  }
}
```

The `Shape` class will now have a `public abstract double getArea()` thanks to it implementing the `GetAreable` interface.

We can have a concrete class implementing an interface too.

```java
class Flat extends RealEstate implements GetAreable {
    private int numOfRooms;
    private String block;
    private String street;
    private int floor;
    private int unit;

    @Override
    public double getArea() {
        :
    }
}
```

For a class to implement an interface and be concrete, it has to override all abstract methods from the interface and provide an implementation to each, just like the example above. Otherwise, the class becomes abstract.

With the `GetAreable` interface, we can now make our function `findLargest` even more general.

```java
// version 0.3
double findLargest(GetAreable[] array) {
  double maxArea = 0;
  for (GetAreable curr : array) {
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
    }
  }
  return maxArea;
}
```

Note:

- A class can only extend from one superclass, but it can implement multiple interfaces.
- An interface can extend from one or more other interfaces, but an interface cannot extend from another class.

Interface as Supertype

If a class C implements an interface I, $C <: I$. This definition implies that a type can have multiple supertypes.

In the example above, `Flat` <: `GetAreable` and `Flat` <: `RealEstate`.

Casting using an Interface

Like any type in Java, it is also possible cast using an Interface. Lets consider an interface `I` and two classes `A` and `B`. Note that `A` does not implement `I`

```java
interface I {
  :
}

class A {
  :
}

class B implements I {
    :
}
```

Now lets, consider the following code excerpt:

```
I i1 = new B(); // Compiles, widening type conversion
I i2 = (I) new A(); // Also compiles?
```

Note that even though `A` does not implement `I`, the Java compiler allows this code to compile. Constrast this with casting between classes which have no subtype relationship:

```
A a = (A) new B(); // Does not compile
B a = (B) new A(); // Does not compile
```

How do we explain this? Well, the Java compiler will not let us cast, when it is provable that it won't work, i.e. casting between two classes which have no subtype relationship. However, for interfaces, there is the *possibility* that a subclass *could* implement the interface and therefore Java allows it to compile. Consider one such potential subclass `AI`:

```
class AI extends A implements I{
  :
}
```

The lesson here is that when we are using typecasting, we are telling the compiler that *we know best*, and therefore it will not warn us or stop us from making bad decisions. It is important to always be sure when you use an explicit typecast.

Impure Interfaces

As we mentioned at the beginning of this module, it is common for software requirements, and their design, to continuously evolve. But once we define an interface, it is difficult to change.

Suppose that, after we define that `GetAreable` interface, other developers in the team starts to write classes that implement this interface. One fine day, we realize that we need to add more methods into the `getAreable`. Perhaps we need methods `getSqFt()` and `getMeter2()` in the interface. But, one cannot simply change the interface and add these abstract methods now. The other developers will have to change their classes to add the implementation of two methods, or else their code would not compile!

This is what happened to the Java language when they transitted from version 7 to version 8. The language needed to add a bunch of useful methods to standard interfaces provided by the Java library, but doing so would break existing code in the 1990s that rely on these interfaces.

The solution that Java came up with is the allow an interface to provide a default implementation of methods that all implementation subclasses will inherit (unless they override). A method with default implementation is tagged with the `default` keyword. This leads to a less elegant situation where an `interface` has some abstract methods and some non-abstract default methods. In CS2030S, we refer to this as *impure interfaces* and it is a pain to explain since it breaks our clean distinction between a class and an interface. *We prefer not to talk about it* -- but it is there in Java 8 and up.

---

1. Although in recent Java releases, this is less common. ↩