# 6.170 Recitation #5: Subtypes and Inheritance

## What is true subtyping?

True Subtyping is not exactly the same as Inheritance.

As seen in an earlier lecture, class A is a true subtype of class B if and only if, whenever B is expected, A can be used. For example, let's consider the following two classes Square and Rectangle:

```
class Square { public int width; }

class Rectangle { public int width,height; }
```

Square is not a true subtype of Rectangle, and Rectangle is also not a true subtype of Square.
To see this, let's consider the following 2 functions:

```
int calculateArea (Square x)              { return
(x.width)*(x.width); }

int calculateCircumference (Rectangle x)  { return
2*(x.width+x.height); }
```

**`calculateArea()`** expects a **`Square`**. If we give it a **`Rectangle`** with **`width==2`** and **`height==3`**,
its answer will be 4 rather than the correct answer of 6.
So **`Rectangle`** is not a true subtype of **`Square`**.

**`calculateCircumference()`** expects a **`Rectangle`**. We can't give it a **`Square`**, since it won't be able to find the **`height`** field.
So **`Square`** is not a true subtype of **`Rectangle`** either.

In general, how do we decide if a type is a true subtype of another type? One way to be more rigorous about it is to apply the substitution principle:

## Substitution Principle

Basically, it states that if code depends on an object of the supertype, then you can substitute an object of a subtype without breaking the system.

The methods of a subtype must hold certain relationships to the methods of the supertype, and the subtype must guarantee that any properties of the supertype (such as representation invariants or specification constraints) are not violated by the subtype.

There are three necessary properties:

- 1. For each method in the supertype, the subtype must have a corresponding method. (The subtype is allowed to introduce additional, new methods that do not appear in the supertype.)
- 2. Each method in subtype that corresponds to one in the supertype must require equal or less than the supertype method. (that is, the method in the subtype has a equal or weaker precondition).
- 3. Each method in subtype that corresponds to one in the supertype must achieve equal or more than the supertype method. (that is, the method in the subtype has a equal or stronger postcondition).

What does condition 2 mean? What exactly must the method in the subtype satisfy? Here are some necessary conditions:

- It must not have additional "requires" clause.
- Each existing "requires" clause must be no more strict than the one in the supertype method.
- Each argument type must be supertypes of the ones in the supertype method. (This feels strange, but it actually makes sense, because any arguments passed to the supertype method will then surely be legal arguments to the subtype method)

What does condition 3 mean? What exactly must the method in the subtype satisfy? Here are some necessary conditions:

- The subtype method must not throw more exceptions
- The subtype method must not modify more variables
- Each clause describing the result of the supertype method must be matched by an equal or stronger clause in the subtype method's description.
- The subtype method's return type must be a subtype of the supertype method's return type.

For example, consider the following 2 methods:

```
class B
 {
  Bicycle f(Bicycle arg);
 }

class A
 {
  RacingBicycle f(Vehicle arg);
 }
```

Method B.f takes a Bicycle as its argument, but A.f can accept any Vehicle (which includes all Bicycles). Method B.f returns a Bicycle as its result, but A.f returns a RacingBicycle (which is itself a Bicycle). So A.f is a refinement of B.f. (And if everything else in class A are refinement of class B also, then class A is a true subtype of class B)

# A More Elaborate Example: Bicycle

```
class Bicycle {
   private int framesize;
   private int chainringGears;
   private int freewheelGears;

   ...

   // returns the number of gears on this bicycle
   public int gears() { return chainringGears * freewheelGears; }

   // returns the cost of this bicycle
   public float cost() { ... }

   // returns the sales tax owed on this bicycle
   public float salesTax() { return cost() * .0825; }

   // effects: transports the rider from work to home
   public void goHome() { ... }
}
class LightedBicycle {
   private int framesize;
   private int chainringGears;
   private int freewheelGears;
   private BatteryType battery;

   ...

   // returns the number of gears on this bicycle
   public int gears() { return chainringGears * freewheelGears; }

   // returns the cost of this bicycle
   float cost() { ... }

   // returns the sales tax owed on this bicycle
   public float salesTax() { return cost() * .0825; }

   // effects: transports the rider from work to home
   public void goHome() { ... }

   // effects: replaces the existing battery with the argument b
   public void changeBattery(BatteryType b) { ... }
}
```

Java and other programming languages use subclassing to overcome these difficulties. Subclassing permits reuse of implementations and overriding of methods.

A better implementation of LightedBicycle is

```
class LightedBicycle extends Bicycle {
   private BatteryType battery;

   // returns the cost of this bicycle
   float cost() { return super.cost() + battery.cost(); }

   // effects: replaces the existing battery with the argument b
   public void changeBattery(BatteryType b) { ... }
}
```

LightedBicycle need not implement methods and fields that appear in its superclass Bicycle; the Bicycle versions are automatically used by Java when they are not overridden in the subclass.

Consider the following implementations of the goHome method. If these are the only changes, are LightedBicycle and RacingBicycle true subtypes of Bicycle?

```
class Bicycle {
  ...
  // requires: windspeed < 20mph && there is daylight
  // effects: transports the rider from work to home
  void goHome() { ... }
}

class LightedBicycle {
   ...
   // requires: windspeed < 20mph
   // effects: transports the rider from work to home
   void goHome() { ... }
}

class RacingBicycle {
   ...
   // requires: windspeed < 20mph && there is daylight
   // effects: transports the rider from work to home
   //          in less than 10 minutes && gets the rider sweaty
   void goHome() { ... }
}
```

To answer that question, recall the definition of subtyping: can an object of the subtype be substituted anywhere that code expects an object of the supertype? If so, the subtyping relationship is valid.

In this case, both LightedBicycle and RacingBicycle are subtypes of Bicycle. In the first case, the requirement is relaxed; in the second case, the effects are strengthened in a way that still satisfies the superclass's effects.

The cost method of LightedBicycle shows another capability of subclassing in Java. Methods can be overridden to provide a new implementation in a subclass. This enables more code reuse; in particular, LightedBicycle can reuse Bicycle's salesTax method. When salesTax is invoked on a LightedBicycle, the Bicycle version is used instead. Then, the call to cost inside salesTax invokes the version based on the runtime type of the object (LightedBicycle), so the LightedBicycle version is used. Regardless of the declared type of an object, an implementation of a method with multiple implementations (of the same signature) is always selected based on the run-time type.

In fact, there is no way for an external client to invoke the version of a method specified by the declared type or any other type that is not the run-time type. This is an important and very desirable property of Java (and other object-oriented languages). Suppose that the subclass maintains some extra fields which are kept in sync with fields of the superclass. If superclass methods could be invoked directly, possibly modifying superclass fields without also updating subclass fields, then the representation invariant of the subclass would be broken.

# Team Building Exercise

Consider a "movie pay-per-view over the internet" application. There are 4 software components involved:

- `Producer`: the producer has a number of movie files, and can give the movie file out (assuming it has been paid for).
- `Consumer`: the consumer accesses some sort of movie listing, pays for a movie, then begins downloading a movie file. (As will be noted below: during times of high traffic volume, consumers may be told to download movies from each other, rather than from the producer. So the consumer module must allow both upload and download)
- `Dispatcher`: the central dispatcher maintains a list of producers and a list of paying consumers. It refers paying consumer to to corresponding producer (to download a movie). However, if there are too many consumers downloading the same movie, the dispatcher may instruct additional consumer to download the movie from an existing consumer (rather than from the producer).
- `Bank`: the bank authorizes payment, and should generate some sort of trusted receipt, tying a particular customer to a particular merchandise. The other components would likely need to query to bank to confirm the payment is received, and to confirm which product it is paid for, for example.

After the room is separated into 4 groups, each group should be assigned one component. Each group should produce 2 things: first of all, the interface that their component will have. Eg. The "producer" will probably have methods like "getMovie()" and

"listAvailableMovies()". That is, they should list out a list of methods that they think other components will need to call on them.

The second thing each group needs to produce is a list of calls that they think they need to make to the other 3 components. Eg. The "producer" may need to call the "dispatcher" to make sure a consumer has paid already, so it may need to call something like dispatch.checkConsumerStatus().

Afterwards, each team should describe their interface, and the rest of the class should give feedback on the design (was it needlessly complicated? was it sufficient?) Since the other 3 components had expected to be able to call some method on this component to retrieve some data, the other 3 groups should check and see if this component's interface is sufficient for the other 3 components' needs.