

Lecture 16: Subtyping

October 17, 2002

16.1 Subclassing

Suppose we have a class for representing bicycles. Here's a partial implementation:

```
class Bicycle {
    private int frameSize;
    private int chainringGears;
    private int freewheelGears;
    ...

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

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

    // returns: sales tax owed on this bicycle
    public float salesTax () {
        return cost () * 0.05;
    }

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

A new class representing bicycles with headlamps can accommodate late nights (or early mornings).

```
class LightedBicycle {
    private int frameSize;
    private int chainringGears;
    private int freewheelGears;
    private BatteryType battery;
```

```

...

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

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

// returns: sales tax owed on this bicycle
public float salesTax () {
    return cost () * 0.05;
}

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

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

```

We created `LightedBicycle` from `Bicycle` by copy-and-paste programming, which is not only tiresome but also unmaintainable. If a bug is found in one copy of the code, it is easy to forget to propagate the fix to all the other versions of the code. Other kinds of changes — like an increase in the sales tax rate — must also be propagated by hand. It is also very hard to comprehend the distinction between the two classes, since the differences are lost in a mass of similarities.

Object-oriented programming languages like Java use subclassing to overcome these difficulties. Subclassing permits us to reuse the implementation of `Bicycle`, overriding methods and adding new methods where necessary to create a `LightedBicycle`:

```

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

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

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

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

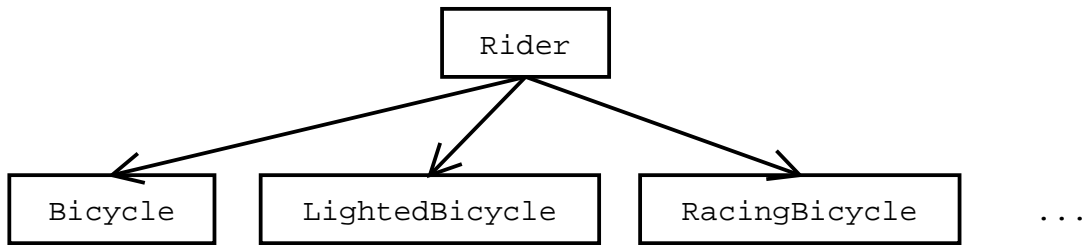
```

}

`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.

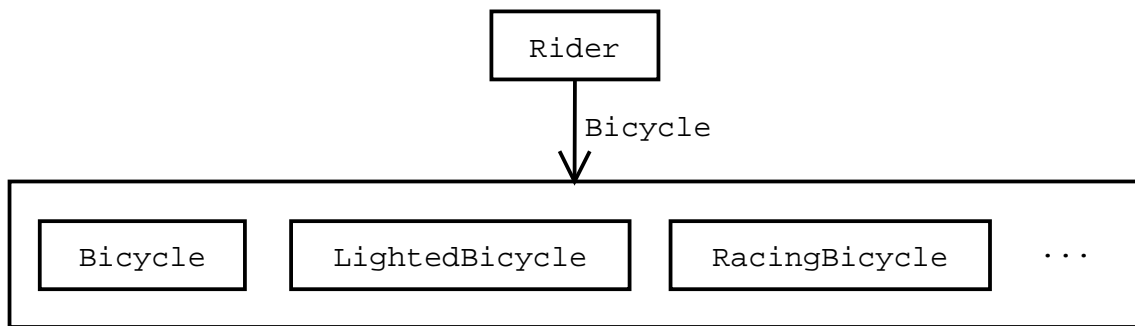
The `cost` method shows another capability of subclassing. Methods can be overridden to provide a new implementation in a subclass, which in turn can change the behavior of inherited methods. Thus, `LightedBicycle` can reuse `Bicycle`'s `salesTax` method, because the call to `cost` inside `salesTax` invokes the version of `cost` appropriate to the runtime type of the object (`LightedBicycle`), not the version that was originally defined in `Bicycle`. Regardless of the declared type of the variable used to refer to an object, a call to a method with multiple implementations (having the same signature) is always selected based on the run-time type of the object.

Subclassing reuses not only the implementation of the superclass, but also its interface. As a result, a client that knows how to call methods on the superclass can use any subclass in the same way, effectively decoupling the client from the individual subclasses. Suppose the `Rider` class models people who ride bicycles. In the absence of subclassing, the module dependence diagram would look like this:



`Rider` would have to know how to ride each kind of bicycle in the system. Worse, the code for `Rider` would have to test which type of bicycle object it had been passed, which would be ugly, verbose, and error-prone.

With subclassing, the MDD looks more like this:



But now a problem arises: the edge labeled `Bicycle` represents a set of assumptions that `Rider` is making about the bicycle. This set of assumptions consists not only of the particular methods and method signatures of the `Bicycle` class (which Java guarantees all its subclasses will support), but also their *specifications*. If a subclass of `Bicycle` does not obey the specification for `Bicycle`, then a `Rider` will be unable to use it correctly. The subclassing relationship between `LightedBicycle` and `Bicycle` is not sufficient; we also need a *subtyping* relationship.

16.2 Subtyping

Type A is a *subtype* of type B when A's specification implies B's specification. That is, any object (or class) that satisfies A's specification also satisfies B's specification. Type theorists often write the subtype relation using a squared-off subset symbol:

$$A \sqsubseteq B$$

Another way to put this is that anywhere in the code, if you expect a B object, an A object would also be acceptable. Code written to work with B objects (and to depend on their properties) is guaranteed to continue to work if it is supplied with A objects instead. Furthermore, the behavior of the code will be the same, if we only consider the aspects of A's behavior that are also included in B's behavior. A may introduce new behaviors that B does not have, like new methods, but it may only change existing B behaviors in ways that are compatible with B's specification.

For example, let's look at the `goHome` method in `Bicycle`. Here's a more complete specification:

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

In `LightedBicycle`, the precondition is weaker, because we don't need daylight anymore:

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

Because `LightedBicycle.goHome` can be called in all the contexts that `Bicycle.goHome` permits, `LightedBicycle` is a *subtype* of `Bicycle`. (Judging only from this method, of course; the other methods would have to be checked as well.)

A `RacingBicycle` has a stronger postcondition; it not only guarantees to get you home, but it will get you home fast:

```
class RacingBicycle extends Bicycle {
    ...
    // requires: windspeed < 20mph && daylight
    // effects: transports rider from work to home
    //           in elapsed time < 10 minutes
    void goHome () { ... }
}
```

Since `RacingBicycle.goHome` guarantees at least as good an outcome as `Bicycle.goHome`, we have a subtyping relationship here too.

Here are two examples of subclasses that are *not* subtypes:

```
class TandemBicycle extends Bicycle {
    ...
    // requires: windspeed < 20mph && daylight
    //           && two riders
    // effects: transports riders from work to home
    void goHome () { ... }
}

class MotorBike extends Bicycle {
    ...
    // requires: windspeed < 20mph && daylight
    // effects: transports riders from work to home
    // throws: OutOfGasException if fuel runs out
    void goHome () { ... }
}
```

`TandemBicycle` has a stronger precondition, so it cannot be used in all the circumstances that a single-rider `Bicycle` can. And `MotorBike`, though it might be appealing to a tired rider, does not guarantee at least as strong a postcondition as `Bicycle`. It may leave the rider stranded by the side of the road, out of gas.

Another example of the difference between subclassing and subtyping was considered in some detail in a previous lecture: the `Object` contract. Every class in Java is guaranteed to be a subclass of `Object`, even if you don't put an `extends` clause on your class declaration. But Java does not guarantee that your class is a *subtype* of `Object`. You may fail to implement the specification of `equals` correctly, or fail to ensure that two equal objects always return the same hash code. If you write a class that is not a subtype of `Object`, then other classes that depend on the `Object` contract (like hash tables and lists) may not handle your objects correctly. Subclassing is a function of Java, which is provided by the Java compiler and enforced by the Java type system. Subtyping is a function of specifications, which Java knows nothing about.

Here's a trickier example that shows that it is sometimes *impossible* to subtype. We know from elementary school that every square is a rectangle. Suppose we wanted to make `Square` a subtype of `Rectangle`, which is a mutable type with a `setSize` method:

```
class Rectangle {
    ...
    // effects: sets this.width=w and this.height=h
    void setSize (int w, int h) { ... }
}

class Square {
    ...
}
```

Which of the following methods is right for `Square`?

```

// requires: w = l
void setSize (int w, int l) { ... }

void setSize (int l) { ... }

// throws: BadSizeException if w != h
void setSize (int w, int l) throws BadSizeException { ... }

```

The answer is: none! The first one isn't right because the subclass method requires more than the superclass method. The second one is useless, because we still must specify a behavior for `setSize(int, int)` which might be called by a client. And the third one isn't right because it throws an exception that the superclass doesn't mention and clients won't expect. (Changing `BadSizeException` to an unchecked exception will make the third method compile successfully, but it won't change the fact that calls that were legal on a `Rectangle` will fail on a `Square`, which means `Square` isn't a subtype.)

There's no way out of this quandary without changing the superclass. Sometimes subtypes do not accord with our intuition! Or else our intuition about what makes a good supertype is wrong.

Recall that we saw a similar problem in the lecture about equality. The example there concerned a `Point` class with a subclass `ColorPoint` that added a new attribute (`color`). We found that there was no way to implement `ColorPoint.equals` that would guarantee both symmetry and transitivity, so `ColorPoint` could not be a subtype of `Point`.

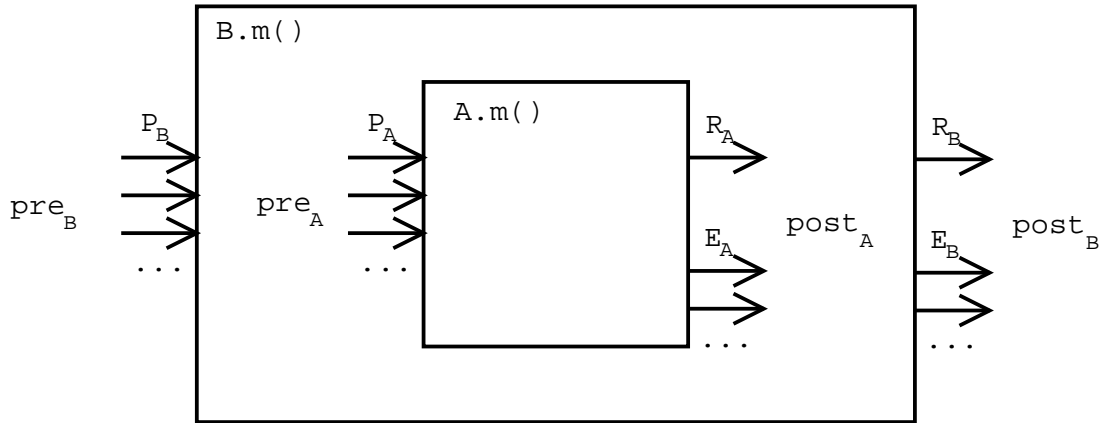
16.3 Substitution Principle

The *substitution principle* is the theoretical underpinning of subtypes. It provides a precise definition of when one type is a subtype of another. Informally, the substitution principle states that a subtype must be substitutable for its supertype. This guarantees that if code is written for the supertype, relying on any aspect of the supertype guaranteed by its specification, but an object of the subtype is substituted, then the code will still work correctly.

The substitution principle has two parts. The first part concerns methods: any methods that the subtype has in common with the supertype must have a certain relationship. The second part concerns overall properties of the supertype. The subtype must guarantee that any properties of the supertype (such as immutability) are not violated by the subtype.

16.3.1 Methods

For each method `m()` in the supertype B, there must be a corresponding method `m()` in the subtype A. The specifications for the methods must be related in the following ways. A useful mnemonic for the required relationships is the picture below, which shows the method interfaces as boxes:



In the figure, parameters are represented as incoming arrows labeled with their declared type (e.g., P_A). The return value is shown as an outgoing arrow labeled with the return type (R_A), and exceptional returns are labeled with the exception type (E_A). Preconditions and postconditions are shown as constraints on the parameters and results, respectively (pre_A and $post_A$)

By the substitution principle, the subtype method $A.m()$ must be callable in all the situations where the supertype method $B.m()$ could be called, having all the effects that $B.m()$ guarantees. We draw the supertype method $B.m()$ on the outside, because outside clients are depending on B's interface. The subtype method $A.m()$ is drawn on the inside because it is attempting to masquerade as $B.m()$, without the client knowing it. Note that the boxes don't represent actual code! We're not implying that $B.m()$ calls $A.m()$, or even that $A.m()$ necessarily overrides $B.m()$. The boxes only represent method interfaces and specifications, which might be implemented by A or B in a variety of ways.

The diagram makes it easier to remember how the relationships go. In each case, implication flows from left to right. Consider specifications first:

- $pre_B \Rightarrow pre_A$. The subtype must have the same or weaker precondition as the supertype, so that the subtype can be called in all states where the supertype could be correctly called.
- $post_A \Rightarrow post_B$. The subtype's effects must imply the supertype's, so that the caller can rely on all the effects that the supertype would have provided.

In true subtyping, the return type and parameter types may also vary:

- $R_A \sqsubseteq R_B$. The subtype method's declared return type may be a subtype of the supertype method's return type. This is called *covariance*, because the direction of subtyping between R_A and R_B is the same as between A and B . Current versions of Java do not permit varying the return type; you may only have $R_A = R_B$. However, the next version (1.5) is expected to allow covariance when you subtype. This will be particularly useful for methods like `clone`:

```
class Object {
    Object clone ();
    ...
}
class Point extends Object {
```

```

        Point clone (); // will be legal in Java 1.5
        ...
    }

```

Changing the return type of `clone` will make it possible to use `clone` without having to cast its return value.

- $P_B \sqsubseteq P_A$. For any given parameter, the subtype's declared parameter type may be a supertype of the corresponding parameter in the supertype. This is called, naturally enough, *contravariance*, because it reverses the direction of the subtyping relationship. Java doesn't support contravariance — you must have $P_B = P_A$ for all parameters. Java is unlikely ever to support contravariance, because of the risk of ambiguity with method overloading.
- $E_A \sqsubseteq E_B$ (or E_B not thrown by A at all). Like the normal return type, exception types also follow covariance, and Java actually supports this now. For example, you can override a superclass method declared to throw `IOException` with a subclass method that throws only the more specific `FileNotFoundException`. An exception may also be omitted entirely if the subtype method never throws it. New exceptions may not be thrown, however.

16.3.2 Properties

Any properties guaranteed by a supertype, such as constraints over the values that may appear in specification fields, must be guaranteed by the subtype as well. The book has a simple example, `FatSet`, which is constrained to contain at least one element. If you subclass `FatSet` and add a new `remove` method that does not guarantee to preserve this property, then your subclass is not a true subtype of `FatSet`. Any clients that depend on this property of `FatSet` would be unable to use your subclass.

Another example of a property is immutability. Java's designers opted not to subclass the immutable type `String` to create the mutable type `StringBuffer`, because `StringBuffer` would not be a proper subtype. Any clients that depend on the immutability of a `String`, e.g. storing a reference without making a defensive copy, or using the object as a key in a hash table, would break when given a `StringBuffer` instead.

One lesson here is that it isn't enough to look only at the methods a subclass has in common with its superclass when determining whether the subclass is a subtype. Any *new methods* introduced in the subclass may also affect its status, if they destroy constraints like `FatSet`'s, or invariants like immutability.

16.4 Final Words

Subtyping is subtle, but subclassing without subtyping is dangerous. The designers of the Java library made an early mistake of this sort that they probably wish they could take back. Java 1.0 had a `Hashtable` class for storing arbitrary associations between keys and values:

```

class Hashtable {
    void put (Object key, Object value);
    Object get (Object key);
    ...
}

```

Java 1.0 also had a class `Properties` for storing a property list, a set of name-value pairs where both are strings. It makes sense to reuse the implementation of `Hashtable` in the implementation of `Properties`, since the two classes are very similar, and in fact share some common methods (like `isEmpty`). Unfortunately, the implementation was reused by making `Properties` a *subclass* of `Hashtable`:

```
class Properties extends Hashtable {
    void setProperty (String key, String value);
    String getProperty (String key);
    ...
}
```

The specification for the `Properties` class states that it should only be used for storing *string* names and values. In fact, its methods for saving and loading the property list from a file would break if any non-string objects were stored in the table. But that means `Properties` is not a subtype of `Hashtable`, because it can't be used in all the contexts where `Hashtable` can be used. Unfortunately, by making `Properties` a subclass of `Hashtable`, the designers implicitly permitted any `Properties` object to be used as if it were a `Hashtable`. For backwards compatibility reasons, this design mistake is still found in the Java library years later.

There is another way to reuse implementation which would have been far better than subclassing in this case: composition (also called aggregation). In other words, make the hash table part of the rep of `Properties`, and delegate calls to it instead of inheriting methods from it:

```
class Properties {
    Hashtable map;
    ...
    void setProperty (String key, String value) {
        map.put (key, value);
    }
    String getProperty (String key) {
        return (String) map.get (key);
    }
    ...
}
```

As a general rule:

If it isn't a subtype, don't make it a subclass.