Chapter 11

Inheritance and Polymorphism

Composition is one example of code reuse. We have seen how classes can be composed with other classes to make more sophisticated classes. In this chapter we will see how classes can be reused in a different way. Using inheritance, a programmer can make a new class out of an existing class, thereby providing a way to create objects with enhanced behavior.

11.1 Inheritance

Recall `TextTrafficLight` (10.2). `TextTrafficLight` objects simulate the cycling behavior of simple, real-world traffic lights. In `Intersection` (10.4) we modeled a four-way intersection synchronizing the operation of four `TextTrafficLight` objects. Suppose the traffic at our intersection has grown, and it has become difficult for vehicles to make left-hand turns because the volume of oncoming traffic is too great. The common solution is to add a left turn arrow that allows opposing traffic to turn left without interference of oncoming traffic. Clearly we need an enhanced traffic light that provides such a left turn arrow. We could copy the existing `TrafficLightModel` source code, rename it to `TurnLightModel`, and then commence modifying it so that it provides the enhance functionality. We then could copy the `TextTrafficLight` code and modify it so that it can display a left turn arrow. This is not unreasonable given the size of our classes, but there are some drawbacks to this approach:

- Whenever code is copied and modified there is the possibility of introducing an error. It is always best, as far as possible, to leave working code untouched.
- If the code is copied and a latent error is discovered and fixed in the original code, the copied code should be repaired as well. The maintainers of the original code may not know who is using copies of their code and, therefore, cannot notify all concerned parties of the change.

Object-oriented languages provide a mechanism that addresses both of these issues. New classes can be built from existing classes using a technique known as *inheritance*, or *subclassing*. This technique is illustrated in `TurnLightModel` (11.1):

```
public class TurnLightModel extends TrafficLightModel {
    //  Add a new state indicating left turn
    public static int LEFT_TURN = 1000;
```



```
    //  Creates a light in the given initial state
    public TurnLightModel(int intialState) {
        super(intialState);
    }

    //  Add LEFT_TURN to the set of valid states
    public boolean isLegalState(int potentialState) {
        return potentialState == LEFT_TURN
                    || super.isLegalState(potentialState);
    }

    //  Changes the state of the light to its next state in its normal cycle.
    //  Properly accounts for the turning state.
    public void change() {
        int currentState = getState();
        if (currentState == LEFT_TURN) {
            setState(GO);
        } else if (currentState == STOP) {
            setState(LEFT_TURN);
        } else {
            super.change();
        }
    }
}
```

Listing 11.1: `TurnLightModel`—extends the `TrafficLightModel` class to make a traffic light with a left turn arrow

In `TurnLightModel` (11.1):

- The reserved word `extends` indicates that `TurnLightModel` is being derived from an existing class—`TrafficLightModel`. We can say this in English various ways:
  - `TurnLightModel` is a *subclass* of `TrafficLightModel`, and `TrafficLightModel` is the *superclass* of `TurnLightModel`.
  - `TurnLightModel` is a *derived class* of `TrafficLightModel`, and `TrafficLightModel` is the *base class* of `TurnLightModel`.
  - `TurnLightModel` is a *child class* of `TrafficLightModel`, and `TrafficLightModel` is the *parent class* of `TurnLightModel`.
- By virtue of being a subclass, `TurnLightModel` inherits all the characteristics of the `TrafficLightModel` class. This has several key consequences:
  - While you do not see a `state` instance variable defined within the `TurnLightModel` class, all `TurnLightModel` objects have such an instance variable. The `state` variable is inherited from the superclass. Just because all `TurnLightModel` objects have a `state` variable does not mean the code within their class can access it directly—`state` is still private to the superclass `TrafficLightModel`. Fortunately, code within `TurnLightModel` can see `state` via the inherited `getState()` method and change `state` via `setState()`.

  - While you see neither `getState()` nor `setState()` methods defined in the `TurnLightModel` class, all `TurnLightModel` objects have these methods at their disposal since they are inherited from `TrafficLightModel`.

- `TurnLightModel` inherits the state constants `OFF`, `STOP`, `CAUTION`, and `GO` and adds a new one, `LEFT_TURN`. `LEFT_TURN`'s value is defined so as to not coincide with the previously defined state constants. We can see the values of `OFF`, `STOP`, `CAUTION`, and `GO` because they are publicly visible, so here we chose 1,000 because it is different from all of the inherited constants.

- The constructor appears to be calling an instance method named `super`:

```
super(initialState);
```

  In fact, `super` is a reserved word, and when it is used in this context it means call the superclass's constructor. Here, `TurnLightModel`'s constructor ensures the same initialization activity performed by `TrafficLightModel`'s constructor will take place for `TurnLightModel` objects. In general, a subclass constructor can include additional statements following the call to `super()`. If a subclass constructor provides any statements of its own besides the call to `super()`, they must follow the call of `super()`

- `TurnLightModel` provides a revised `isLegalState()` method definition. When a subclass redefines a superclass method, we say it *overrides* the method. This version of `isLegalState()` expands the set of integer values that map to a valid state. `isLegalState()` returns true if the supplied integer is equal to `LEFT_TURN` or is approved by the superclass version of `isLegalState()`. The expression:

```
super.isLegalState(potentialState)
```

  looks like we are calling `isLegalState()` with an object reference named `super`. The reserved word `super` in this context means execute the superclass version of `isLegalState()` on behalf of the current object. Thus, `TurnLightModel`'s `isLegalState()` adds some original code (checking for `LEFT_TURN`) and reuses the functionality of the superclass. It in essence does what its superclass does plus a little extra.

  Recall that `setState()` calls `isLegalState()` to ensure that the client does not place a traffic light object into an illegal state. `TurnLightModel` does not override `setState()`—it is inherited as is from the superclass. When `setState()` is called on a pure `TrafficLightModel` object, it calls the `TrafficLightModel` class's version of `isLegalState()`. By contrast, when `setState()` is called on behalf of a `TurnLightModel` object, it calls the `TurnLightModel` class's `isLegalState()` method. This ability to "do the right thing" with an object of a given type is called *polymorphism* and will be addressed in § 11.4.

- The `change()` method inserts the turn arrow state into its proper place in the sequence of signals:

  - red becomes left turn arrow,
  - left turn arrow becomes green, and
  - all other transitions remain the same (the superclass version works fine)

  Like `isLegalState()`, it also reuses the functionality of the superclass via the `super` reference.

Another interesting result of inheritance is that a `TurnLightModel` object will work fine in any context that expects a `TrafficLightModel` object. For example, a method defined as


```
public static void doTheChange(TrafficLightModel tlm) {
    System.out.println("The light changes!");
    tlm.change();
}
```

obviously accepts a `TrafficLightModel` reference as an actual parameter, because its formal parameter is declared to be of type `TrafficLightModel`. What may not be so obvious is that the method will also accept a `TurnLightModel` reference. Why is this possible? A subclass inherits all the capabilities of its superclass and usually adds some more. This means anything that can be done with a superclass object can be done with a subclass object (and the subclass object can probably do more). Since any `TurnLightModel` object can do at least as much as a `TrafficLightModel`, the `tlm` parameter can be assigned to `TurnLightModel` just as easily as to a `TrafficLightModel`. The `doTheChange()` method calls `change()` on the parameter `tlm`. `tlm` can be an instance of `TrafficLightModel` or *any subclass* of `TrafficLightModel`. We say an *is a* relationship exists from the `TurnLightModel` class to the `TrafficLightModel` class. This is because any `TurnLightModel` object *is a* `TrafficLightModel` object.

In order to see how our new turn light model works, we need to visualize it. Again, we will use inheritance and derive a new class from an existing class, `TextTrafficLight`. `TextTurnLight` (11.2) provides a text visualization of our new turn light model:

```
public class TextTurnLight extends TextTrafficLight {
    //  Note: constructor requires a turn light model
    public TextTurnLight(TurnLightModel lt) {
        super(lt);  //  Calls the superclass constructor
    }

    //  Renders each lamp
    public String drawLamps() {
        //  Draw non-turn lamps
        String result = super.drawLamps();
        //  Draw the turn lamp properly
        if (getState() == TurnLightModel.LEFT_TURN) {
            result += " (<)";
        } else {
            result += " ( )";
        }
        return result;
    }
}
```

Listing 11.2: `TextTurnLight`—extends the `TextTrafficLight` class to make a traffic light with a left turn arrow

`TextTurnLight` (11.2) is a fairly simple class. It is derived from `TextTrafficLight` (10.2), so it inherits all of `TextTrafficLight`'s functionality, but the differences are minimal:

- The constructor expects a `TurnLightModel` object and passes it to the constructor of its superclass. The superclass constructor expects a `TrafficLightModel` reference as an actual parameter. A `TurnLightModel` reference is acceptable, though, because a `TurnLightModel` *is a* `TrafficLightModel`.

- In `drawLamps()`, a `TextTurnLight` object must display four lamps instead of only three. This method renders all four lamps. The method calls the superclass version of `drawLamps()` to render the first three lamps:

  ```
  super.drawLamps();
  ```

  and so the method needs only draw the last (turn) lamp.

  Notice that the `draw()` method, which calls `drawLamps()`, is not overridden. The subclass inherits and uses `draw()` as is, because it does not need to change how the "frame" is drawn.

The constructor requires clients to create `TextTurnLight` objects with only `TurnLightModel` objects. A client may not create a `TextTurnLight` with a simple `TrafficLightModel`:

```
//  This is illegal
TextTurnLight lt
    = new TextTurnLight(new TrafficLightModel
                                (TrafficLightModel.RED));
```

The following interaction sequences demonstrates some of the above concepts. First, we will test the light's cycle:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> TextTurnLight lt = new TextTurnLight
                        (new TurnLightModel
                              (TrafficLightModel.STOP));

> System.out.println(lt.show());
[(R) ( ) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[( ) ( ) ( ) (<)]
> lt.change(); System.out.println(lt.show());
[( ) ( ) (G) ( )]
> lt.change(); System.out.println(lt.show());
[( ) (Y) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[(R) ( ) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[( ) ( ) ( ) (<)]
> lt.change(); System.out.println(lt.show());
[( ) ( ) (G) ( )]
> lt.change(); System.out.println(lt.show());
[( ) (Y) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[(R) ( ) ( ) ( )]
```

All seems to work fine here. Next, let us experiment with this *is a* concept. Reset the Interactions pane and enter:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> TextTrafficLight lt = new TextTurnLight
```



```
                        (new TurnLightModel
                              (TrafficLightModel.STOP));

> System.out.println(lt.show());
[(R) ( ) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[( ) ( ) ( ) (<)]
> lt.change(); System.out.println(lt.show());
[( ) ( ) (G) ( )]
> lt.change(); System.out.println(lt.show());
[( ) (Y) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[(R) ( ) ( ) ( )]
```

Notice that here the variable `lt`'s declared type is `TextTrafficLight`, not `TextTurnLight` as in the earlier interactive session. No error is given because a `TextTurnLight` object (created by the `new` expression) *is a* `TextTrafficLight`, and so it can be assigned legally to `light`. Perhaps Java is less picky about assigning objects? Try:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
>  Intersection light = new TextTurnLight
                             (new TurnLightModel
                                   (TrafficLightModel.STOP));
Error: Bad types in assignment
```

Since no superclass/subclass relationship exists between `Intersection` and `TextTurnLight`, there is no *is a* relationship either, and the types are not assignment compatible. Furthermore, the *is a* relationship works only one direction. Consider:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> TextTurnLight lt2 = new TextTrafficLight
                              (new TrafficLightModel
                                    (TrafficLightModel.STOP));

ClassCastException: lt2
```

All `TurnLightModel`s are `TrafficLightModel`s, but the converse is not true. As an illustration, all apples are fruit, but it is not true that all fruit are apples.

While inheritance may appear to be only a clever programming trick to save a little code, it is actually quite useful and is used extensively for building complex systems. To see how useful it is, we will put our new kind of traffic lights into one of our existing intersection objects and see what happens. First, to simplify the interactive experience, we will define `TestIntersection` (11.3), a convenience class for making either of the two kinds of intersections:

```
public class TestIntersection {
    public static Intersection makeSimple() {
```



```
        return new Intersection(
             new TextTrafficLight
                  (new TrafficLightModel(TrafficLightModel.STOP)),
             new TextTrafficLight
                  (new TrafficLightModel(TrafficLightModel.STOP)),
             new TextTrafficLight
                  (new TrafficLightModel(TrafficLightModel.GO)),
             new TextTrafficLight
                  (new TrafficLightModel(TrafficLightModel.GO)));
    }
    public static Intersection makeTurn() {
        return new Intersection(
             new TextTurnLight
                  (new TurnLightModel(TrafficLightModel.STOP)),
             new TextTurnLight
                  (new TurnLightModel(TrafficLightModel.STOP)),
             new TextTurnLight
                  (new TurnLightModel(TrafficLightModel.GO)),
             new TextTurnLight
                  (new TurnLightModel(TrafficLightModel.GO)));
    }
}
```

Listing 11.3: `TestIntersection`—provides some convenience methods for creating two kinds of intersections

Both methods are class (`static`) methods, so we need not explicitly create a `TestIntersection` object to use the methods. The following interactive session creates two different kinds of intersections:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> simple = TestIntersection.makeSimple();
> simple.show();
                 [(R) ( ) ( )]

[( ) ( ) (G)]                    [( ) ( ) (G)]

                 [(R) ( ) ( )]
> simple.change(); simple.show();
                 [(R) ( ) ( )]

[( ) (Y) ( )]                    [( ) (Y) ( )]

                 [(R) ( ) ( )]
> simple.change(); simple.show();
                 [( ) ( ) (G)]

[(R) ( ) ( )]                    [(R) ( ) ( )]

                 [( ) ( ) (G)]
```



```
> simple.change(); simple.show();
                 [( ) (Y) ( )]

[(R) ( ) ( )]                    [(R) ( ) ( )]

                 [( ) (Y) ( )]
> simple.change(); simple.show();
                 [(R) ( ) ( )]

[( ) ( ) (G)]                    [( ) ( ) (G)]

                 [(R) ( ) ( )]
> simple.change(); simple.show();
                 [(R) ( ) ( )]

[( ) (Y) ( )]                    [( ) (Y) ( )]

                 [(R) ( ) ( )]
> turn = TestIntersection.makeTurn();
> turn.show();
                 [(R) ( ) ( ) ( )]

[( ) ( ) (G) ( )]                    [( ) ( ) (G) ( )]

                 [(R) ( ) ( ) ( )]
> turn.change(); turn.show();
                 [(R) ( ) ( ) ( )]

[( ) (Y) ( ) ( )]                    [( ) (Y) ( ) ( )]

                 [(R) ( ) ( ) ( )]
> turn.change(); turn.show();
                 [( ) ( ) ( ) (<)]

[(R) ( ) ( ) ( )]                    [(R) ( ) ( ) ( )]

                 [( ) ( ) ( ) (<)]
> turn.change(); turn.show();
                 [( ) ( ) (G) ( )]

[(R) ( ) ( ) ( )]                    [(R) ( ) ( ) ( )]

                 [( ) ( ) (G) ( )]
> turn.change(); turn.show();
                 [( ) (Y) ( ) ( )]

[(R) ( ) ( ) ( )]                    [(R) ( ) ( ) ( )]

                 [( ) (Y) ( ) ( )]
> turn.change(); turn.show();
```



```
                [(R) ( ) ( ) ( )]

[( ) ( ) ( ) (<)]                    [( ) ( ) ( ) (<)]

                [(R) ( ) ( ) ( )]
> turn.change(); turn.show();
                [(R) ( ) ( ) ( )]

[( ) ( ) (G) ( )]                    [( ) ( ) (G) ( )]

                [(R) ( ) ( ) ( )]
> turn.change(); turn.show();
                [(R) ( ) ( ) ( )]

[( ) (Y) ( ) ( )]                    [( ) (Y) ( ) ( )]

                [(R) ( ) ( ) ( )]
> turn.change(); turn.show();
                [( ) ( ) ( ) (<)]

[(R) ( ) ( ) ( )]                    [(R) ( ) ( ) ( )]

                [( ) ( ) ( ) (<)]
```

Notice that our original `Intersection` class was not modified at all, yet it works equally as well with `TextTurnLight` objects! This is another example of the "magic" of inheritance. A `TextTurnLight` object can be treated exactly like a `TextTrafficLight` object, yet it behaves in a way that is appropriate for a `TextTurnLight`, not a `TextTrafficLight`. A `TextTrafficLight` object draws three lamps when asked to `show()` itself, while a `TextTurnLight` draws four lamps. This is another example of polymorphism (see § 11.4).

11.2 Protected Access

We have seen how client access to the instance and class members of a class are affected by the `public` and `private` specifiers:

- Elements declared `public` within a class are freely available to code in any class to examine and modify.
- Elements declared `private` are inaccessible to code in other classes. Such private elements can only be accessed and/or influenced by `public` methods provided by the class containing the private elements.

Sometimes it is desirable to allow special privileges to methods within subclasses. Java provides a third access specifier—`protected`. A `protected` element cannot be accessed by other classes in general, but it can be accessed by code within a subclass. Said another way, `protected` is like `private` to non-subclasses and like `public` to subclasses.

Class designers should be aware of the consequences of using `protected` members. The `protected` specifier weakens encapsulation (see Section 8.7). Encapsulation ensures that the internal details of a class cannot be disturbed by client code. Clients should be able to change the state of an object only through the public methods provided. If these public methods are correctly written, it will be impossible for client code to put an object into an undefined

or illegal state. When fields are made `protected`, careless subclassers may write methods that misuse instance variables and place an object into an illegal state. Some purists suggest that `protected` access never be used because the potential for misuse is too great.

Another issue with `protected` is that it limits how superclasses can be changed. Anything `public` becomes part of the class's interface to all classes. Changing public members can break client code. Similarly, anything `protected` becomes part of the class's interface to its subclasses, so changing protected members can break code within subclasses.

Despite the potential problems with the `protected` specifier, it has its place in class design. It is often convenient to have some information or functionality shared only within a family (inheritance hierarchy) of classes. For example, in the traffic light code, the `setState()` method in `TrafficLightModel` (10.1) might better be made `protected`. This would allow subclasses like `TurnLightModel` (11.1) to change a light's state, but other code would be limited to making a traffic light with a specific initial color and then alter its color only through the `change()` method. This would prevent a client from changing a green light immediately to red without the caution state in between. The turn light code, however, needs to alter the basic sequence of signals, and so it needs special privileges that should not be available in general.

Since encapsulation is beneficial, a good rule of thumb is to reveal as little as possible to clients and subclasses. Make elements `protected` and/or `public` only when it would be awkward or unworkable to do otherwise.

11.3 Visualizing Inheritance

The Unified Modeling Language is a graphical, programming language-independent way of representing classes and their associated relationships. The UML can quickly communicate the salient aspects of the class relationships in a software system without requiring the reader to wade through the language-specific implementation details (that is, the source code). The UML is a complex modeling language that covers many aspects of system development. We will limit our attention to a very small subset of the UML used to represent class relationships.

In the UML, classes are represented by rectangles. Various kinds of lines can connect one class to another, and these lines represent relationships among the classes. Three relationships that we have seen are composition, inheritance, and dependence.

- **Composition**. An `Intersection` object is composed of four `TextTrafficLight` objects. Each `TextTrafficLight` object manages its own `TrafficLightModel` object, and a `TextTurnLight` contains a `TurnLightModel`. The UML diagram shown in Figure 11.1 visualizes these relationships. A solid line from one class to another with a diamond at one end indicates composition. The end with the diamond is connected to the container class, and the end without the diamond is connected to the contained class. In this case we see that an `Intersection` object contains `TextTrafficLight` objects. A number at the end of the line indicates how many objects are contained. (If no number is provided, the number 1 is implied.) We see that each intersection is composed of four traffic lights, while each light has an associated model.

  The composition relationship is sometimes referred to as the *has a* relationship; for example, a text traffic light *has a* traffic light model.

- **Inheritance**. `TurnLightModel` is a subclass of `TrafficLightModel`, and `TextTrafficLight` is a subclass of `TextTurnLight`. The inheritance relationship is represented in the UML by a solid line with a triangular arrowhead as shown in Figure 11.2. The arrow points from the subclass to the superclass.

  We have already mentioned that the inheritance relationship represents the *is a* relationship. The arrow points in the direction of the *is a* relationship.

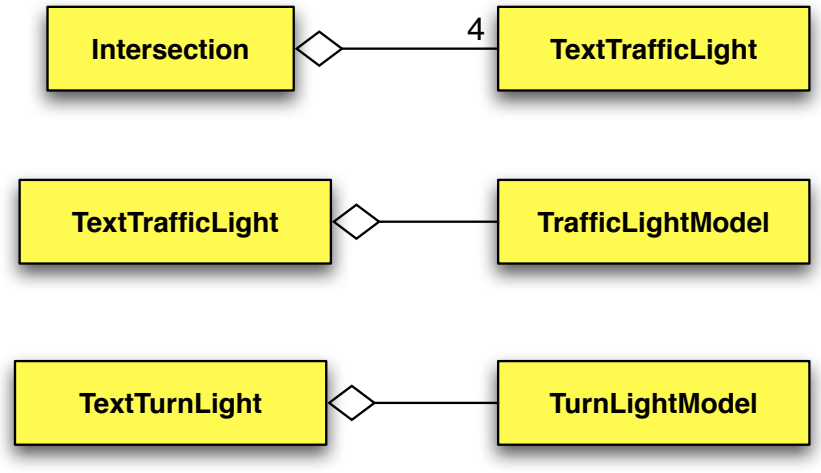

Figure 11.1: UML diagram for the composition relationships involved in the `Intersection` class

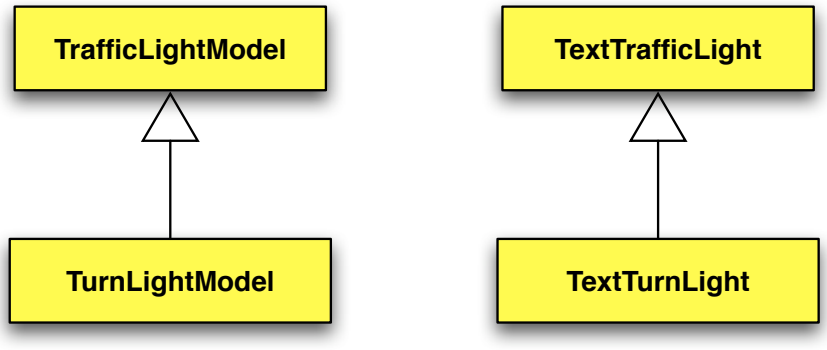

Figure 11.2: UML diagram for the traffic light inheritance relationship

- **Dependence**. We have used dependence without mentioning it explicitly. Objects of one class may use objects of another class without extending them (inheritance) or declaring them as fields (composition). Local variables and parameters represent temporary dependencies. For example, the `TestIntersection` class uses the `Intersection`, `TrafficLightModel`, `TextTrafficLight`, `TurnLightModel`, and `TextTurnLight` classes within its methods (local objects), but neither inheritance nor composition are involved. We say that `TestIntersection` depends on these classes, because if their interfaces change, those changes may affect `TestIntersection`. For example, if the maintainers of `TextTurnLight` decide that its constructor should accept an integer state instead of a `TurnLightModel`, the change would break `TestIntersection`. Currently `TestIntersection` creates a `TextTurnLight` with a `TurnLightModel`, not an integer state.

  A dashed arrow in a UML diagram illustrates dependency. The label ≪uses≫ indicates that a `TestIntersection` object uses the other object is a transient way. Other kinds of dependencies are possible. Figure 11.3 shows the dependency of `TestIntersection` upon `Intersection`.

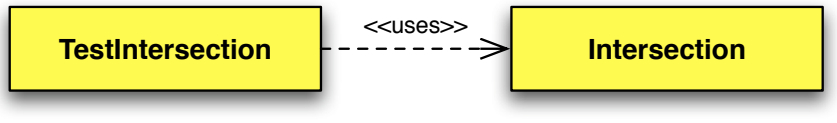

Figure 11.3: UML diagram for the test intersection dependencies

Figure 11.4 shows the complete diagram of participating classes.

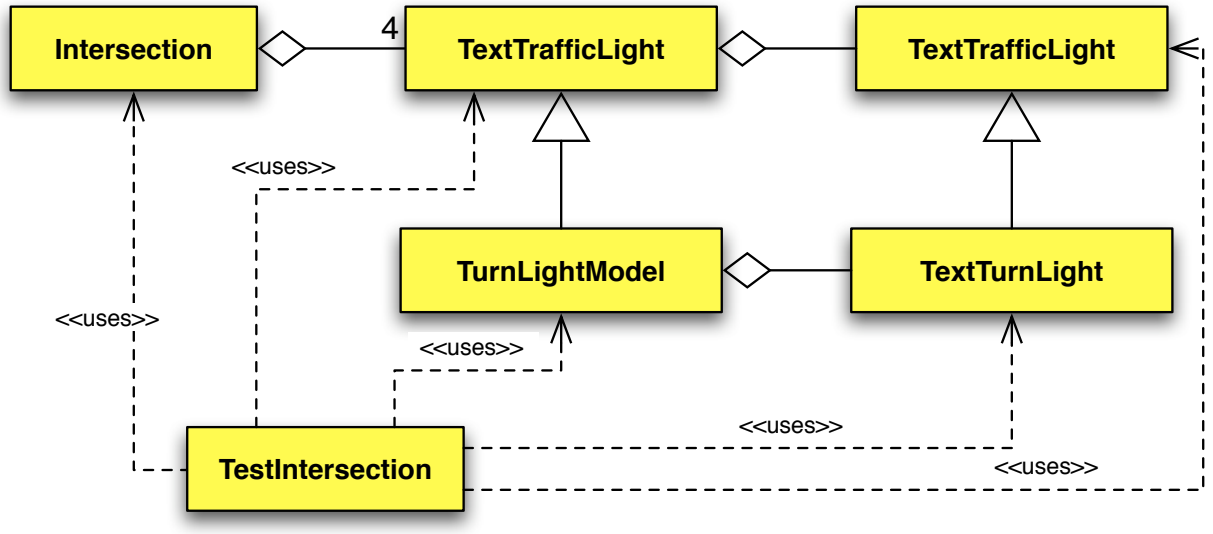

Figure 11.4: the complete UML diagram for the classes used in `TestIntersection`

11.4 Polymorphism

How does the code within `Intersection`'s `show()` method decide which of the following ways to draw a red light?

```
[(R) ( ) ( )]
```

or

```
[(R) ( ) ( ) ( )]
```

Nothing within `Intersection`'s show method reveals any distinction. It does not use any `if/else` statements to select between one form or another. The `show()` method does the right thing based on the exact kind of traffic light that it is asked to render. Can the compiler determine what to do when it compiles the source code for `Intersection`? The compiler is powerless to do so, since the `Intersection` class was developed and tested *before* the `TurnLightModel` class was ever conceived!

The compiler generates code that at runtime decides which `show()` method to call based on the actual type of the light. The burden is, in fact, on the object itself. The expression

```
northLight.show()
```

is a request to the `northLight` object to draw itself. The `northLight` object draws itself based on the `show()` method in its class. If it is really a `TextTrafficLight` object, it executes the `show()` method of the `TextTrafficLight` class; if it is really a `TextTurnLight` object, it executes the `show()` method of the `TextTurnLight` class.

This process of executing the proper code based on the exact type of the object when the *is a* relationship is involved is called *polymorphism*. Polymorphism is what makes the `setState()` methods work as well. Try to set a plain traffic light to the `LEFT_TURN` state, and then try to set a turn traffic light to `LEFT_TURN`:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> plain = new TextTrafficLight(new TrafficLightModel(TrafficLightModel.STOP));
> turn = new TextTurnLight(new TurnLightModel(TrafficLightModel.STOP));
> plain.setState(TurnLightModel.LEFT_TURN);
> System.out.println(plain.show());
[(R) ( ) ( )]
> turn.setState(TurnLightModel.LEFT_TURN);
> System.out.println(turn.show());
[( ) ( ) ( ) (<)]
```

Remember, `setState()` was not overridden. The same `setState()` code is executed for `TurnLightModel` objects as for `TrafficLightModel` objects. The difference is what happens when `setState()` calls the `isLegalState()` method. For `TrafficLightModel` objects, `TrafficLightModel`'s `isLegalState()` method is called; however, for `TurnLightModel` objects, `TurnLightModel`'s `isLegalState()` method in invoked. We say that `setState()` calls `isLegalState()` polymorphically. `isLegalState()` polymorphically "decides" whether `LEFT_TURN` is a valid state. The "decision" is easy though; call it on behalf of a pure `TrafficLightModel` object, and says "no," but call it oh behalf of a `TurnLightModel` object, and it says "yes."

Polymorphism means that given the following code:

```
TextTrafficLight light;

//  Initialize the light somehow . . .

light.show();
```

we cannot predict whether three lamps or four lamps will be displayed. The code between the two statements may assign `light` to a `TextTrafficLight` object or a `TextTurnLight` depending on user input (§ 8.3), a random number (§ 13.6, time (§ 13.2), or any of hundreds of other criteria.

11.5 Extended Rational Number Class

`Rational` (9.1) is a solid, but simple, class for rational number objects. Addition and multiplication is provided, but what if we wish to subtract or divide fractions? Without subclassing clients could do all the dirty work themselves:

- Subtracting is just adding the opposite:

$$\frac{a}{b} - \frac{c}{d} = \frac{a}{b} + \left(-1 \times \frac{c}{d}\right)$$

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> Rational f1 = new Rational(1, 2),
           f2 = new Rational(1, 4);
> //  f3 = f1 - f2 = f1 + (-1) * f2
> Rational f3 = f1.add(new Rational(-1, 1).multiply(f2));
> f3.show()
"1/4"
> f2 = new Rational(1, 3);
```



```
> f3 = f1.add(new Rational(-1, 1).multiply(f2));
> f3.show()
"1/6"
```

Since `f1` = $\frac{1}{2}$ and `f2` = $\frac{1}{4}$, the statement

```
Rational f3 = f1.add(new Rational(-1, 1).multiply(f2));
```

results in

$$\texttt{f3} = \frac{1}{2} + \left(\frac{-1}{1} \times \frac{1}{4}\right) = \frac{2}{4} - \frac{1}{4} = \frac{1}{4}$$

and for `f2` = $\frac{1}{3}$, the statement

```
f3 = f1.add(new Rational(-1, 1).multiply(f2));
```

results in

$$\texttt{f3} = \frac{1}{2} + \left(\frac{-1}{1} \times \frac{1}{3}\right) = \frac{3}{6} - \frac{2}{6} = \frac{1}{6}$$

- Dividing is multiplying by the inverse:

$$\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \times \frac{d}{c}$$

Interactions

```
> //  1/2 divided by 2/3 = 3/4
> f1.show()
"1/2"
> f2 = new Rational(2, 3);
> f2.show()
"2/3"
> f3 = f1.multiply(new Rational(f2.getDenominator(),
                                f2.getNumerator()));
> f3.show()
"3/4"
```

The problem with this approach is that it is messy and prone to error. It would be much nicer to simply say:

```
f3 = f1.subtract(f2);
```

and

```
f3 = f1.divide(f2);
```

but these statements are not valid if `f1` is a `Rational` object. What we need is an extension of `Rational` that supports the desired functionality. `EnhancedRational` (11.4) is such a class.


```
public class EnhancedRational extends Rational {
    //  num is the numerator of the new fraction
    //  den is the denominator of the new fraction
    //  Work deferred to the superclass constructor
    public EnhancedRational(int num, int den) {
        super(num, den);
    }
    //  Returns this - other reduced to lowest terms
    public Rational subtract(Rational other) {
        return add(new Rational(-other.getNumerator(),
                                other.getDenominator()));
    }
    //  Returns this / other reduced to lowest terms
    //  (a/b) / (c/d) = (a/b) * (d/c)
    public Rational divide(Rational other) {
        return multiply(new Rational(other.getDenominator(),
                                     other.getNumerator()));
    }
}
```

Listing 11.4: `EnhancedRational`—extended version of the `Rational` class

With `EnhancedRational` (11.4) subtraction and division are now more convenient:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> EnhancedRational f1 = new EnhancedRational(1, 2),
                   f2 = new EnhancedRational(1, 4);
> Rational f3 = f1.subtract(f2);
> f3.show()
"1/4"
> f3 = f1.divide(f2);
> f3.show()
"2/1"
```

11.6 Multiple Superclasses

In Java it is not possible for a class to have more than one superclass. Some languages like C++ and Smalltalk do support multiple superclasses, a concept called *multiple inheritance*.

Even though a class may not have more than one superclass, it may have any number of subclasses, including none.

11.7 Summary

- Inheritance allows us to derive a new class from an existing class.

- The original class is called the superclass, and the newly derived class is called the subclass.
- Other terminology often used instead of superclass/subclass base class/derived class, and parent class/child class.
- The subclass inherits everything from its superclass and usually adds more capabilities.
- The reserved word `extends` is used to subclass an existing class.
- Subclasses can redefine inherited methods; the process is called overriding the inherited method.
- Subclass methods can call superclass methods directly via the `super` reference.
- Constructors can invoke the superclass constructor using `super` in its method call form.
- A subclass inherits everything from its superclass, including private members (variables and methods), but it has no extra privileges accessing those members than any other classes.
- Subclass objects have an *is a* relationship with their superclass; that is, if `Y` is a subclass of `X`, an instance of `Y` *is a* `Y`, and an instance of `Y` *is a* `X` also.
- A subclass object may be assigned to a superclass reference; for example, if `Y` is a subclass of `X`, an instance of `Y` may be assigned to a variable of type `X`.
- A superclass object may **not** be assigned to a subclass reference; for example, if `Y` is a subclass of `X`, an instance whose exact type is `X` may not be assigned to a variable of type `Y`.
- The Unified Modeling Language (UML) uses special graphical notation to represent classes, composition, inheritance, and dependence.
- Polymorphism executes a method based on an object's exact type, not simply its declared type.
- A class may not have more than one superclass, but it may have zero or more subclasses.

11.8 Exercises

1. Suppose the variable `lt` is of type `TextTrafficLight` and that it has been assigned properly to an object. What will be printed by the statement:

```
System.out.println(lt.show());
```

2. What does it mean to override a method?
3. What is polymorphsim? How is it useful?
4. If `TurnLightModel` (11.1) inherits the `setState()` method of `TrafficLightModel` (10.1) and uses it as is, why and how does the method behave differently for the two types of objects?
5. May a class have multiple superclasses?
6. May a class have multiple subclasses?

7. Devise a new type of traffic light that has `TrafficLightModel` (10.1) as its superclass but does something different from `TurnLightModel` (11.1). One possibility is a flashing red light. Test your new class in isolation, and then create a view for your new model. Your view should be oriented horizontally so you can test your view with the existing `Intersection` (10.4) code.

8. Derive `VerticalTurnLight` from `VerticalTextLight` (10.3) that works like `TextTurnLight` (11.2) but that displays its lamps vertically instead of horizontally. Will you also have to derive a new model from `TurnLightModel` (11.1) to make your `VerticalTurnLight` work?

