KIT101 Notes — 12 — More Object Orientation

Revised: 2017-04-22

References: L&L 4.1–4.5, 9.1–9.4

Software is Complex

Software is possibly the most complex "thing" ever produced by humans, and also the least forgiving of errors. The earliest programs were *monolithic*, equivalent to have all the instructions inside the `main()` method of a Java class. As a monolithic program grows it becomes harder to understand and maintain (and keep free of errors).

One of the next innovations was *procedural programming*, where common and reusable sections of code were packaged up in procedures (in other words, methods). In procedural programming, data and the operations on it are kept quite separate, but data is often quite widely accessible. (Many procedures that don't need to be able to modify certain data still have access to it.) The techniques introduced so far in these notes have primarily been in this style, but with the addition of using objects defined by others.

A further refinement of procedural programming is object-oriented programming, where the different parts of a software solution are modelled as different classes of objects. Each object (hence, each class) knows how to do a few things really well, and takes responsibility for keeping track of the relevant information and modifying it as needed. The overall program arises from the interactions of the objects in it.

Of course, now the problem becomes deciding what kinds of objects are needed and how they

should interact. (Software is still complex.)

Object-oriented problem solving

The steps of object-oriented (OO) problem solving are similar to the problem solving steps covered in Week 2, but we can expand on some of the details.

1. Understand the problem (get the *requirements* right)
2. Dissect the problem into manageable pieces
    - Determine the (different kinds of) **objects** that need to be modelled
    - Ask yourself: can these be modelled with an existing class or primitive type? (If so you could save time and reduce the complexity of the software you have to write.)
3. Design a solution
    - What **properties** will the objects need? (Properties are the variables they contain)
    - How will the objects **interact**? (What methods will they each have and call on each other?)
4. Consider alternatives and refine the solution
5. Implement the solution
6. (In conjunction with 5) Write and test the new classes by creating objects (in purpose-built but simple testing programs) to test the solution to the problem

Later parts of these notes have suggested structures for object-oriented programs. This part considers a single example new class type that is less about storing data (as in the previous part) and more about being a self-contained component of a larger program.

Encapsulation

A key concept in OO design is encapsulation, which refers to the way that an object takes responsibility for its own data and also protects that data from uninvited external modification. In essence, an object is a black box to other parts of the program. This gives rise to two “views” of an object:

- External (‘client’ or consumer) view of an object
    - interaction of the object with other objects in the program
    - what the object does
    - send messages (call methods)
- Internal (supplier/producer) view of an object
    - structure of its data
    - algorithms used by its methods
    - how the object achieves its behaviour

Instance data access

To support encapsulation, the declarations of instance variables (variables declared inside a class but outside any method) can also be preceded by a level of access. Whereas methods are generally `public` (visible in any other part of the program), instance fields are normally `private` (visible only within the code written in that class).

Aside: There's also **`protected`**, which makes a variable visible inside that class and any subclasses that extend it (and also to any classes in the same package).

Modelling a game die: An example of object-oriented problem solving

Imagine we wish to model a game die, or in fact a die with (almost) any number of sides. The class describing such objects would have attributes for:

- the number of faces on the die (which makes it more flexible and reusable)
- the current face value (since it's not a physical die we need to know which value is on the 'top')
- some source of randomness

because it would also have the abilities to:

- randomise its current face (roll itself)
- report its current face value
- and report how many faces it has (because it's a software die we can't just *look* at it to see)

So the high-level model of our Die class could be:

Die

`int numFaces`
`int faceValue`
`Random generator`

`int getFaces()`
`int getFaceValue()`
`int roll()`

from which you can deduce that `numFaces` and `faceValue` are *read only* properties; they have a getter but no setter.

Based on this high-level design we could create this initial implementation with just the instance variables.

```
import java.util.Random;

public class Die {
    private int numFaces; //number of faces
    private int faceValue; //current side up
    private Random generator; //source of next face

    //More code to be written...
}
```

Preparing the Die for use with its constructor

Unlike the data-oriented classes described in Part 11 of the notes, it doesn't make sense for the Die to request values for all of its variables from outside. Only the number of faces should be under the control of whoever (that is, whichever other part of the program) creates the Die object, and even then it should remain fixed once the Die has been created.

Multiple alternative constructors can be defined for a class, distinguished by their differing parameter lists. For instance, it is common to have a parameterless constructor that uses known default values for the object's data, plus at least one other constructor that accepts alternative values for its data. In the `Die` example, the parameterless constructor could be the following, which creates a die with six sides, an initial value of 1 and also instantiates its random number generator:

```
public Die() {
    numFaces = 6;
    faceValue = 1;
    generator = new Random();
}
```

Since there are dice with different numbers of sides, an alternative constructor could be defined that allows the number of faces to be set. This constructor would use knowledge of the minimum realistic number of faces an actual die can have (four if the edges between faces are sharp) and override the client's requested number if a value that's too low is given (`MIN_FACES` would be defined earlier in `Die`'s code):

```
public Die(int faces) {
    if (faces >= MIN_FACES) {
        numFaces = faces;
    } else { //too small, set to standard die default
        numFaces = 6;
    }
    faceValue = 1;
    generator = new Random();
}
```

The `this` keyword

Inside the methods of an object the keyword **`this`** refers to the object itself. It can be used to distinguish between the variables in *this* object and another of the same class or between an instance variable and a method parameter that have the same name. More importantly, it allows us to save on the amount of code we write...

`this` in a constructor

`this()` can be used as the first line in one constructor to call another constructor in the same class, which is useful if the two constructors have a lot of code in common. Consider the two constructors for `Die`: both set the initial face value to 1 and instantiate the random number generator, but in one the number of die faces is always set to 6 and in the other this value is a parameter. So we can rewrite the two constructors as follows:

```
public class Die {
    ...

    public Die() {
        this(6); //calls Die(int faces) constructor
    }

    public Die(int faces) {
        if (faces >= MIN_FACES) {
            numFaces = faces;
        } else { //too small, set to standard die default
            numFaces = 6;
        }
        faceValue = 1;
        generator = new Random();
    }

    ...
}
```

Implementing the other methods

As `numFaces` and `faceValue` are just read only properties they only have a getter each, and no way for code outside `Die` to directly modify their values (after the `Die` has been created).

Activity: Implement the getters for `numFaces` and `faceValue`

Follow the pattern for defining a getter method for the two properties.

Here are two possible implementations. Note that we've made a choice to use a slightly different name for the numFaces property, which is perfectly valid as long as it still makes sense (and sometimes the internal name will need to be different to the external one).

```
/** Returns the Die's current face value. */
public int getFaceValue() {
    return faceValue;
}

/** Returns the number of faces on the Die. */
public int getFaces() {
    return numFaces;
}
```

The `Die`'s roll() method should randomly assign `faceValue` a new value with uniform probabilty, to simulate rolling the die, and then rather than require the caller to separately call getFaceValue it can return that value as well.

Activity: Roll the die

Implement the `Die`'s roll() method according to the above description.

```
/** Simulates rolling the die; returns the new face. */
public int roll() {
    faceValue = generator.nextInt(numFaces) + 1;
    return faceValue;
}
```

Variable Scope and Lifetime

You've now seen (and written) a variety of programs that declare variables in different places: inside main(), inside a method to be called from main(), as method parameters, and as instance variables. A variable's *scope* is the region of a program where it is visible, and is largely determined by the location of its declaration (i.e., which block of { } it is defined in). (Of course, access modifiers like **`public`** and **`private`** can also be applied to instance variables to affect their visibility.)

The following table summarises the differences between *local* data, *instance* data and *parameters* (which are really another kind of local data).

	Local Data	Instance Data	Parameters
Declared	inside method	inside class, outside all methods	in method header
Scope	within the method	all methods in the class	within the method
Purpose	Data only needed in method	To share data between methods in the class	To pass data in to a method from elsewhere
Lifetime	Created anew each time method is called	Created when object is instantiated	Created anew each time method is called

Advanced topic: Existing classes can be extended

Rather than start a class definition completely from scratch, it is possible to extend existing

classes and to add additional data and behaviour (methods) to the objects of the new class. For instance, if there was already a `Polygon` class we could extend it to create the `Rectangle` class described in other parts of these notes: `Polygon` may have a fairly basic implementation, such as:

```
public class Rectangle extends Polygon {
    //lots of code not shown...
}
```

The general form of creating a class by extending another is:

```
public class ClassName extends SuperclassName{
    variable (property) declarations
    method declarations
}
```

where the only addition is **`extends`** `SuperclassName`.

Terminology

Extending an existing class creates an 'is-a' relationship between the new 'subclass' and the 'superclass'. For instance, a Rectangle is-a Polygon (but not all Polygons are Rectangles).

Again using the example from above, `Rectangle` can be said to **extend** (or **subclass** [as a verb]) `Polygon`, meaning that `Rectangle` is a subclass of `Polygon`, and `Polygon` is a **superclass** of `Rectangle`

Uses for extending an existing class

A subclass inherits all the data and methods of its superclass. It can then *add* to those (so we achieve code reuse by not having to reimplement common content) and may also *replace* (override) existing method implementations.

> **Note:** A variable of a superclass type can refer to an object of that type or any of its subclasses. So we could declare a variable of type `Polygon` but at runtime make it refer to a `Rectangle` or (if we'd implemented them) a `Triangle` or `Square` or `Dodecagon`, and so on. We can then call any method defined in `Polygon`, because all of those subclasses inherit it, but if they overrode the inherited behaviour then their customised version of the method will be called. This powerful feature of object-oriented languages is referred to as *polymorphism*.

toString, the most common method to override

Implicitly, every time you declare a new class, it actually extends the class confusingly named

`Object` class, which defines a small set of common methods that all objects in Java have. These include the `toString` method, which is why *any* object can be passed to `System.out.println()` and it may safely call toString() to obtain a String representation of it.

So whenever you provide an implementation of toString() you are actually overriding the default implementation (the one that produces the ugly *ClassName*@*hexadecimal-number* output).

Adding features and overriding implementations

Continuing the example from above, we may have a fairly basic implementation of `Polygon` that provides a default version of the area() method (because it has no way of knowing how to calculate its area yet):

```
public class Polygon {

    /** Returns the area of this polygon. */
    public int area() {
        return 0;
    }
}
```

Then when we implement `Rectangle` we could replace this default behaviour with a customised implementation that works for rectangles:

```
public class Rectangle extends Polygon {
    private int width;
    private int height;

    //constructor, getters, setters...

    public int area() {
        return width * height;
    }
}
```

Note: In practice we may actually implement the Polygon class a little differently, using the keyword **`abstract`** on both the class and in the method header for `area` (and there would be no method body). Doing so would force any subclasses to provide an implementation of that method. But abstract classes are outside the scope of this unit, so we won't discuss this further.

Constructors are not inherited

The only parts of a superclass that aren't inherited are the constructors, which gives the author of a subclass more freedom to determine how objects of that class are created. For example, we

could create a custom 20-sided game die based on the Die implemented above by creating a subclass that does not provide a constructor that accepts the number of faces:

```
public class D20 extends Die {
    //no need for data as that's inherited from Die

    /** Creates a 20-sided game die. */
    public D20() {
        super(20); //Call's the Die(int faces) constructor
    }
}
```

Note: To invoke a superclass's constructor directly from a subclass's constructor use the keyword **`super`**, in the same way that you can use the keyword **`this`** in a constructor to refer to another constructor defined in the same class.

A wrinkle: `private` data is not visible inside subclasses

Instance variables marked as **`private`** cannot be directly accessed inside subclasses (the subclasses must use the getters and setters just as if they were another part of the program external to the original class). If you are planning to create a set of related classes then you can allow subclasses direct access to instance data in a superclass by marking the fields **`protected`** instead of **`private`**.

Note: There is no consensus on which approach is better, having subclasses 'wrap' their superclass but effectively be external to its code, or allowing them direct access to their inherited data. What you end up doing will depend on your own preferences (developed with time and practice) and what is required by your employer or client.
