KIT101 Notes — 14 — Object-oriented Program Design

James Montgomery

Revised: 2018-02-16

References: L&L 4.4, 7.1, 7.2, 7.4

Organising tasks with classes: What belongs in `main()`?

The early tasks you worked on in this unit had all their code in main(), and were often small enough that this was a manageable approach to solving those problems. Later tasks had you splitting up program functionality across different (`static`) methods to be called from main(). This second approach can substantially reduce the effort in implementing, testing and modifying your programs, but it still has limitations. Complex programs residing in a single file (perhaps supplemented by a few data-oriented classes) can be difficult to understand and maintain.

Whether you use a procedural language like C or Pascal, or an object-oriented language like Java or C#, you will eventually need to split your program's functionality across larger 'modules' (hence, different source files). In OO languages each of these becomes its own class of object, which has its own data (a subset of what the program needs to model the problem) and methods (operations to act on that data).

Using an 'organiser' class

One useful heuristic for assigning roles to the class-level components of your software is to have a main or driver class that is quite small, and which contains main(), with another class that implements the key functionality of the program (what you have up to now been putting in the same source file as main), an 'organiser'.

The 'organiser' class is the model, made up of interacting objects/variables. It creates objects of other classes (resources) and has methods to control the interactions between them. It will commonly have a 'top level' method that controls the task at a high level, which calls other methods that perform each of the subtasks.

A 'driver' class containing `main()` is required to start things off:

- Instantiate the 'organiser' class
- Set it running (call the top level method), if necessary

For example, in `Driver.java`:

```
public class Driver {
    public static void main(String[] args) {
        Organiser o = new Organiser();
        o.topLevelAlg();
    }
}
```

while in `Organiser.java`:

```
public class Organiser {
    public void topLevelAlg() {
        /* Top-level algorithm here */
    }
}
```

Tip: Don't use names like Driver and Organiser in your programs (unless they are actually relevant to the domain in which you are working). For example, if you were writing a small application to manage a list of contacts you might call the driver class Contacts (the application's name) and the organiser ContactsManager (since it manages a collection of contacts).

Advantages	Disadvantages
Code in the class is *only* the code needed to do the task (so can be used elsewhere)	Needs an extra file: organiser class as well as class for `main()` and any additional classes created to solve the problem
Straightforward to have methods to do different parts of the task	Seems like a lot of trouble for simple problems (like many of the examples you'll encounter in an introductory programming unit)
Organiser class can be changed without changing the driver, such as when changing from a console interface to GUI	

Tip: Distinction-level task 8.3 has an extended example of how to refactor an existing single-source program with multiple methods into a two-source program consisting of a driver and an organiser.

An intermediate option

Add a `main()` method to the organiser class, as in:

```
public class Organiser {
    public void topLevelAlg() {
        /* Top-level algorithm here */
    }

    public static void main(String[] args) {
        Organiser o = new Organiser();
        o.topLevelAlg();
    }

}
```

This can be convenient but is potentially less flexible. However, `Organiser` can still be instantiated by other classes (with their own `main()` method) if they want to use it in a different way.

Suitable methods for ‘library’ classes

As you produce more software you will be faced with recurring problems that have similar solutions. ‘Library’ classes are classes (of objects) that are general purpose or whose purpose is common enough that they can be reused in many programs to solve different problems. Each library class will typically have:

- Constructor
- One method for each ‘action’ the object can perform
    - Any necessary methods to do the separate parts of the action
- Setters for instance variables (if needed/appropriate)
- Getters for instance variables (if needed)
- A `trace()` method, for use during development (see the note about debugging methods in the appendix on testing)

Suitable methods for ‘organiser’ classes

‘Organiser’ classes define the top-level algorithm to solve a problem. They are problem-specific, and may make use of other problem-specific classes you write as well as library classes. Each organiser class will typically have:

- Constructor (if necessary to do any set up)
- One method for each of the sub-problems. Often these relate to:
    - Input
    - Processing
    - Output
- Any necessary methods to do the separate parts of each sub-problem
- A `trace()` method, for use during development (see the note about debugging methods in the appendix on testing)