Unit 4: Encapsulation

After reading this unit, students should

- understand composite data type as a even-higher level abstraction over variables
- understand encapsulation as an object-oriented (OO) principle
- understand the meaning of class, object, fields, methods, in the context of OO programming
- be able to define a class and instantiate one as an object in Java
- appreciate OO as a natural way to model the real world in programs
- understand reference types in Java and its difference from the primitive types

Abstraction: Composite Data Type

Just like functions allow programmers to group instructions, give it a name, and refer to it later, a *composite data type* allows programmers to group *primitive types* together, give it a name (a new type), and refer to it later. This is another powerful abstraction in programming languages that help us to think at a higher conceptual level without worrying about the details. Commonly used examples are mathematical objects such as complex numbers, 2D data points, multi-dimensional vectors, circles, etc, or everyday objects such as a person, a product, etc.

Defining composite data type allows programmers to abstract away (and be separated from the concern of) how a complex data type is represented.

For instance, a circle on a 2D plane can be represented by the center (`x`, `y`) and its radius `r`, or it can be represented by the top left corner (`x`,`y`) and the width `w` of the bounding square.

In C, we build a composite data type with `struct`. For example,

```c
typedef struct {
  double x, y; // (x,y) coordinate of the center.
  double r; // radius
} circle;
```

Once we have the `struct` defined, we are not completely shielded from its representation, until we write a set of functions that operates on the `circle` composite type. For instance,

```
double circle_area(circle c) { ... };
bool   circle_contains_point(circle c, double x, double y) { ... };
  :
```

Implementing these functions requires knowledge of how a circle is represented. Once the set of functions that operates on and manipulates circles is available, we can use the *circle* type without worrying about the internal representation.

If we decide to change the representation of a circle, then only the set of functions that operates on a circle type need to be changed, but not the code that uses circles to do other things. In other words, the representation of the circle and the set of functions that operation on and manipulate circles, fall on the same side of the abstraction barrier.

Abstraction: Class and Object (or, Encapsulation)

We can further bundle the composite data type *and its associated functions* on the same side of the abstraction barrier together, into another abstraction called a *class*.

A class is a data type with a group of functions associated with it. We call the functions as *methods* and the data in the class as *fields* (or *members*, or *states*, or *attributes*[1]). A well-designed class maintains the abstraction barrier, properly wraps the barrier around the internal representation and implementation, and exposes just the right *method interface* for others to use.

The concept of keeping all the data and functions operating on the data related to a composite data type together within an abstraction barrier is called *encapsulation*.

Let's see how we can encapsulate the fields and methods associated together, using `Circle` as an example, in Java.

```
// Circle v0.1
class Circle {
  double x;
  double y;
  double r;

  double getArea() {
    return 3.141592653589793 * r * r;
  }
}
```

The code above defines a new class using the keyword `class`, give it a name `Circle`[2], followed by a block listing the member variables (with types) and the function definitions.

Just like we can create variables of a given type, we can create *objects* of a given class. Objects are *instances* of a class, each allowing the same methods to be called, and each containing the same set of variables of the same types, but (possibly) storing different values.

In Java, the keyword `new` creates an object of a given class. For instance, to create a `Circle` object, we can use

```java
Circle c = new Circle();
```

To access the fields and the methods, we use the `.` notation. For example, `object.field` or `object.method(..)`. For instance,

```java
c.r = 10;     // set the radius to 10
c.getArea(); // return 314.1592653589793
```

Object-Oriented Programming

A program written in an *object-oriented language* such as Java consists of classes, with one main class as the entry point. One can view a running object-oriented (or OO) program as something that instantiates objects of different classes and orchestrates their interactions with each other by calling each others' methods.

One could argue that an object-oriented way of writing programs is much more natural, as it mirrors our world more closely. If we look around us, we see objects all around us, and each object has certain properties, exhibits certain behavior, and they allow certain actions. We interact with the objects through their interfaces, and we rarely need to know the internals of the objects we used every day (unless we try to repair them)[3].

To model a problem in an object-oriented manner, we typically model the nouns as classes and objects, the properties or relationships among the classes as fields, and the verbs or actions of the corresponding objects as methods.

Reference Types in Java

We mentioned in Unit 2 that there are two kinds of types in Java. You have been introduced to the primitive types. Everything else in Java is a reference type.

The `Circle` class is an example of a reference type. Unlike primitive variables, which never share the value, a reference variable stores only the reference to the value, and therefore two reference variables can share the same value. For instance,

```java
Circle c1 = new Circle();
Circle c2 = c1;
System.out.println(c2.r); // print 0
c1.r = 10.0;
System.out.println(c2.r); // print 10.0
```

The behavior above is due to the variables `c1` and `c2` referencing to the same `Circle` object in the memory. Therefore, changing the field `r` of `c1` causes the field `r` of `c2` to change as well.

Special Reference Value: `null`

Any reference variable that is not initialized will have the special reference value `null`. A common error for beginners is to declare a reference variable and try to use it without instantiating an object:

```java
Circle c1;
c1.r = 10.0;  // error
```

Line 2 would lead to a run-time error message

```
|  Exception java.lang.NullPointerException
```

Remember to *always instantiate a reference variable* before using it.

Further Readings

- Oracle's Java Tutorial on Classes and Objects

---

1. Computer scientists just could not decide what to call this :(↩
2. As a convention, we use PascalCase for class name and camelCase for variable and method names in Java. ↩
3. This is a standard analogy in an OOP textbook. In practice, however, we often have to write programs that include abstract concepts with no tangible real-world analogy as classes. ↩