COMP6700/2140 Objects

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

March 2017

What is object?

1. A unit of data
2. Capable of performing operation on those data (and on what is passed as the input)
3. Protected by “walls” from outside interference to ensure the object state is not invalidated (this protection is known as *encapsulation*)

Procedural Paradigm

The procedural (structural) programming paradigm, which is what we follow when writing separate methods and then calling them from the `main` method, divide the virtual world into three distinct parts:

- The *Program Control* (coded `main` method)
- *Data* (our variables of the primitive data type, and data structures like arrays)
- *Operations* acting on *Data* (our methods)

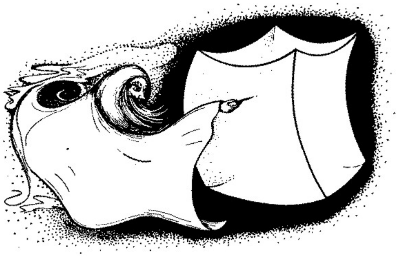
Data and *Operations* are *very* different:

- *Data* are passive: they change only as a result of *Operations* acting on them at the behest of *Program Control*
- *Operations* have no lasting state, their role is only to change *Data* in accordance with *Program Control*

Program Control is responsible for **every** detail of computation. This results in overly complex logic of the main program. Can this global responsibility be relieved?

Easing the Burden in OOP

Poor old man (or is it "Poor us programmers"?)



(from "Emperor's New Mind" by R. Penrose)

Object-Oriented Paradigm

"This division [between data and operations] is grounded in the way computers work, so it's not one that you can easily ignore or push aside. Like the equally pervasive distinctions between matter and energy and between nouns and verbs, it forms the background against which we work. At some point, all programmers — even object-oriented programmers — must lay out the data structures that their programs will use and define the functions that will act on the data." (from *The Objective-C Programming Language* manual, © Apple Computer, Inc.)

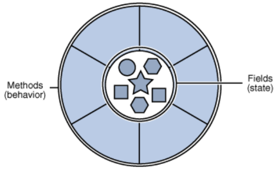
> *"Object-oriented programming doesn't so much dispute this view of the world as restructure it at a higher level. It groups operations and data into modular units called* objects *and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design."*

The first idea of OO programming was due to Ole-Johan Dahl and Kirsten Nygaard (*Simula-67*); it was duly embraced by the industry only in the late 1980s, after a successful implementation in *Smalltalk-80* (a language + run-time environment; Java has repeated the same approach 15 years later).

Object

- The world of Procedural Program is like a factory assembly line in which everything happens in accordance with rules imposed from outside
- The world of Object-Oriented Program is like a biocoenosis — a community of organisms integrated in one system and subject to external forces

Object = data + operations



Object is meant to simulate an entity from the real world (a tangible object like a table, but not only them — events and relationships can be also modelled by objects, *eg* an assignment submission, or marriage, or…).

Objects

OO programming creates objects in an attempt to model real-world entities:

- Tangible things (Students, Cars, Offices)
- Roles (Voter, Electoral Office)
- Incidents (Submission of Assignment, Act of Voting, Parachute Jump)
- Interactions (Traffic Collision, Vote — relate Roles and Incidents)
- Specification and Services (Set of Election Rules, Mathematical Rules *etc*)

Not every time an object can be constructed from the above list, but at least everything which goes as a *noun or noun phrase* can be considered as an object (which may or may not be appropriate). When deciding to model something by an object, one has to establish:

1. Whether the potential object can have *states*?
2. What, if any, *behaviour* the would be object can have?

By answering "yes" to both question, a virtual counterpart to the real one can be defined (whether it will be featured as an object in the final model, depends on design and other considerations.

OO Paradigm vs Procedural Paradigm

Benefits of building your software system as a collection of objects (from *The Java Tutorial*):

- Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
- Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
- Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Yet the dynamics — the run-time properties — of an OO system can be much more complex than the dynamics of a procedural system. The relevant simile would be to compare an assembly line with the life cycle in a habitat populated by numerous species living and eating (incl. each other) side by side. Which way is better to build complex system? No simple answer is possible. And (some) software developers still question the overall benefit which OO has brought into the industry.

How objects are defined?

Java is a strongly typed language: every value (variable) has a type. The type determines what can be done with the value, which operations it can be subjected to.

- Primitive types — "passive bricks of bits"
- Reference types — data and operations packed together as one unit

Data of reference type exist as objects: strings, arrays, I/O streams… Every object

- *may* have (and usually has) one or many data associated with it (these data are "carried" by the object); these data can be either primitive values or references; they are called fields, and each has a defined name and type.
- may have (often has) one or many operations associated with it; these operations, when *invoked*, performed prescribed computations using data fields and/or data passed as parameters; they are called methods, and each has a defined name, a type which value it returns after completing its operations, and type of parameters passed to it at the invocation.
- an object of a name `o`, which has a field `f`, has its value evaluated like this: `o.f`; an object with a method `m(...)`, can have it *invoked* like this: `o.m(...)`; if the method returns a value, the statement `o.m(...)` becomes an expression which evaluates to this value.

Fields and methods are collectively known as members of the class which defines the object.

What's inside a class?

In Java, objects are created in accordance with a template which determines their members. This template is defined via **class declaration**.

The class structure consists of:

1. zero, one or many field declarations (an instance of a class without fields is a stateless object, eg. `java.lang.Object`)
2. zero, one or many method declarations (an instance of a class without methods is possible, but very interesting; when they are required, better to use `enam` types — we'll learn this later)
3. zero, one or many so-called *constructors*, which are like methods but strictly speaking are not — instead of returning a value, they build and return a *reference to an object* of this class; **constructors have the same name as their class**

If you have a reference variable `o` (or a literal *object*, *eg* a string), the operator `.` (*dot*) applied to `o` (or a literal object) gives an access to the object internal resources (object's members), if such access is allowed: `o.f` evaluates the field `f`, `o.m()` invokes the method `m()`.

Constructors are needed to control the creation of an object; when we need to declare and create an object (also called an *instance* of the class), we have to use the operator `new` "applied" to the class constructor:

```
MyType myObject = new MyType();
```

Rules of class definition

1. Think what [reference] type do you need for your program, define them as a new class
    - choose its name carefully — it must convey the essence well, always use **noun**
2. Think what data objects of a new class will contain [it is possible to define a class without data, it's rare but can be useful], decide how these data will be represented — the **type**, **name** and **structural characteristics** (modifiers):
    - declare them as class **fields**; which are exposed (should be rare), which are hidden (normal choice), which are allowed to change
3. Think what actions objects of the class will perform:
    - define necessary **methods** with appropriate *signatures*
4. Think whether you need to control how instances of the class (objects) should be created:
    - define appropriate **constructors**; their parameters are usually used to initialise the class fields. A class may not have a defined constructor (the default, parameterless one is used then), it can also have several (they must differ by the number and type of arguments). Constructors also can be hidden (`private`) — we will learn later why it's useful.

Syntax of class declaration

Methods which can be invoked on an object and its fields define the object type. Object-oriented language provides the programmer with possibility to define their own type.

The type definition is achieved by *class declaration*

Formal ("dry") syntax rules:

```
[<modifier>]* class ClassName [extends ParentClass] [implements AnInterface]* {
   [<access-modifier>] [<scope-modifier>] returnType fieldName [ = initValue];
    ...  repeated as many times as there are fields ...

   [<access-modifier>] [<scope-modifier>] returType methodName([arguments]) {
       body of a method;
   }

   [<access-modifier>] [<scope-modifier>] ClassName(...) {
     ... body of a constructor ...
   }
  ...  more methods, constructors and inner classes ...
}
```

can be made more vivid in an example: let's declare the class *Planet* and create a few planet objects (and make them move ☺).

Class *Planet*

The class *Planet* describes a celestial body:

```
class Planet {
  public String name;
  public Planet orbits; // not all fields are nouns
  public long id; // the catalog number of the new planet
  private static long nextID = 0; //the total planet counter

  public Planet(String s) { //one constructor
    this.name = s;
    id = nextID++; // here the post-increment matter!
  }
  public static long getNumberOfPlanets() { return nextID; }

  public Planet(String s, Planet p) {  //another constructor
    this.name = s;
    this.orbits = p;
    id = nextID++;
  }
}
Planet moon; // variable declaration
Planet sun = new Planet("Sun"); // declaration and instantiation
Planet venus = new Planet("Venus", sun); // another instantiation
```

Object-Orientation vs Procedural Programming (again)

There is no *versus*, one thing does not exclude another:

- Object-Oriented programming is about data and operation representation
- Procedural programming is about structuring a program
- The two paradigms are complimentary, not mutually exclusive.

Less so about Functional Programming (more on it later).

Divine Billiard

Planets are ready, let's play!

(call the method `main`)

Where to look for this topic in the textbook?

Hortsmann's *Core Java for the Impatient*, Ch. 2.1, 2.2