Classes, Instances and Methods

(Budd chapter 4, 5)

Plan

- Encapsulation and Instantiation
- Classes
- Methods
- Messages
- Object Creation
- Memory Recovery

Same Ideas, Different Terms

All OOP languages have the following concepts, although the terms they use may differ:

- **classes**, object type, factory object
- **instances**, objects
- **message passing**, method lookup, member function invocation, method binding
- **methods**, member function, method function
- **inheritance**, subclassing

Encapsulation and Instantiation

Classes provide a number of very important capabilities:

- **Encapsulation** - The purposeful hiding of information, thereby reducing the amount of details that need to be remembered/communicated among programmers.
- **A Service View** - The ability to characterise an object by the service it provides, without knowing how it performs its task.
- **Instantiation** - The ability to create multiple instances of an abstraction.

Internal and External Views

Encapsulation means there are two views of the same system.

- The **outside**, or service view, describes what an object does.
- The **inside**, or implementation view, describes how it does it.

Behavior and State

A class can also be viewed as a combination of **behavior** and **state**.

- **Behavior**: The actions that an instance can perform in response to a request. Implemented by methods.
- **State**: The data that an object must maintain in order to successfully complete its behavior. Stored in instance variables (also known as data members, or data fields).
Methods
Although syntax will differ depending upon language, all methods have the following:

- A **name** that will be matched to a message to determine when the method should be executed.
- A **signature**, which is the combination of return type and argument types. Methods with the same name can be distinguished by different signatures.
- A **body**, which is the code that will be executed when the method is invoked in response to a message.

Constructors

- A **constructor** is a method that is used to initialize a newly constructed object. In C++, Java, C# and many other languages it has the same name as the class.
- An **accessor** (or getter) is a method that simply returns an internal data value.
- A **setter** (sometimes called a mutator method) is a method that is used to change the state of an object.

Order of Methods

For the most part, languages don't care about the order that methods are declared. Here are some guidelines:

- List important topics first.
- Constructors are generally very important, list them first.
- Put public features before private ones.
- Break long lists into groups.
- List items in alphabetical order to make it easier to search.

Remember that class definitions will often be read by people other than the original programmer.

Messages are not Function Calls

Note the following differences between a **message** and a function call (or procedure call):

- A **message** is always given to some object, called the receiver.
- The action performed in response is determined by the receiver, different receivers can do different actions in response to the same message.

Message Passing Syntax

Although the syntax may differ in different languages, all messages have three identifiable parts:

```
martian.met(a, 42, 27);
```

- The message receiver
- The message selector
- An optional list of arguments
Class Static Data Fields

• Idea is that all instances of a class can share a common data field.
• Simple idea, but how to resolve the following. All instances have the same behavior:
  – Either they all initialize the common area, which seems bad, or
  – Nobody initializes the common area, which is also bad.
• Example: static array that stores the squares of first 100 integers
• In Java the initialization of a static data field can be accomplished by a static block

```java
Static { // static block
    count = 0;
}
```

Statically Types and Dynamically Typed Languages

• A statically typed language requires the programmer to declare a type for each variable. The validity of a message passing expression will be checked at compile time, based on the declared type of the receiver.
• A dynamically typed language associates types with values, not with variables. A variable is just a name. The legality of a message cannot be determined until run-time.
• Java, C++, C#, Object Pascal are statically typed languages
• Smalltalk, CLOS, Python dynamically typed
• Objective-C both (type id can hold any object value)

The Receiver Variable

Inside a method, the receiver can be accessed by means of a pseudo-variable

• Called this in Java, C++, C#
• Called self in Smalltalk, Objective-C, Object Pascal
• Called current in Eiffel

Within a method, a message expression or a data access with no explicit receiver is implicitly assumed to refer to this

Object Creation

• In most programming languages objects must be created dynamically, usually using the new operator

  A declaration simply names a variable, the new operator is needed to create the new object value.

Pointers and Memory Allocation

• OO languages use pointers in their underlying representation
• All objects references are pointers in their internal representation
• Not all languages expose this representation to the programmer
• Java has no pointers that the programmers can see (in contrast to C++)

Memory Recovery

Objects must be recovered at run-time. There are two broad approaches to this:

• Force the programmer to explicitly say when a value is no longer being used
  – Object Pascal ( free aCard; )
  – Objective C ([ aCard free ];)
  – C++ ( delete aCard; )
• Use a garbage collection system that will automatically determine when values are no longer being used, and recover the memory.
  – Java, C#, Smalltalk
Memory Errors

Garbage collection systems impose a run-time overhead, but prevent a number of potential memory errors:

- Running out of memory because the programmer forgot to free values
- Using a memory value after it has been recovered
- Free the same value twice