Object Oriented Programming

Summary
Class Materials

1. Introduction to Object Oriented Programming
2. Abstraction
3. Object Oriented Design
4. Classes and Methods
5. Messages, Instances and Initialization
6. Inheritance and Substitution
7. Subclasses and Subtypes, Static and Dynamic Behavior
8. Implications of Substitution
9. Multiple Inheritance
10. Overloading and Overriding
11. Generics, Examples
12. Object Interconnections (Coupling and Cohesion)
Introduction to Object Oriented Programming

Why is this term used to describe the object-oriented approach to problem solving?

How does language influence thought

Imperative Programming

Characteristics of the object-oriented way of thinking
Abstraction
Abstraction

Abstraction is the purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure.
Information Hiding

- **Purposeful omission of details** in the development of an abstract representation

- Information hiding is what allows abstraction **to control complexity**
Levels of Abstraction in Object Oriented Programs

At the highest level of abstraction we view a program as a community of interacting objects (and interacting programmers).

Important characteristics here are the lines of communication between the various agents.
Levels of Abstraction in Object Oriented Programs

• Levels of Abstraction Clients and Servers
• Levels of Abstraction Interfaces
• Levels of Abstraction – An Implementation
• Levels of Abstraction -- A Method in Isolation
Summary:
Levels of Abstraction

Every level is important, and often we move quickly back and forth between levels
Forms of Abstraction

- Abstraction
- Division into parts
- Specialization
- Multiple views
Is-a and Has-A abstraction

Two of the most important types of abstraction are the following:

- Division into parts -- Has-a abstraction
- Division into specialization -- Is-a abstraction

Both is-a and has-a abstractions will reappear in later chapters and be tied to specific programming language features.
A Short History of Abstraction Mechanisms

Another way to better understand OOP is to put it in context with the history of abstraction in computer science:

i. Assembly languages
ii. Procedures
iii. Modules
iv. ADT
v. The Service View
vi. Objects
vii. The future....
Classes and Methods
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
Objects as Examples of Abstract Data Types

• We noted that object-oriented programming, and objects in particular, can be viewed from many perspectives

• We described the many levels of abstraction from which one could examine a program

• In this chapter, we wish to view objects as examples of abstract data types
Encapsulation

- Programming that makes use of data abstractions is a methodological approach to problem solving where information is consciously hidden in small parts of a program.

- In particular, the programmer develops a series of abstract data types, each of which can be viewed as having two faces:

  push
  pop
  top

  const limit=300;
  var currentTop : 0 .. limit;
  values : array [ 1 .. limit ] of integer;

The outside, or service view, describes what an object does.
The inside, or implementation view, describes how it does it.
Behavior and State

An object 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)
We will use as a running example the class definition for a playing card abstraction, and show how this appears in several languages. Delphi Pascal, Apple Pascal, Ruby, Python, Eiffel, Objective-C and Smalltalk.
Class Definition

C++

class PlayingCard {
    public:
        enum Suits {Spade, Diamond, Club, Heart};
        Suits suit () { return suitValue; }
        int rank () { return rankValue; }
        private:
            Suits suitValue;
            int rankValue;
};

Java

public class PlayingCard {
    public enum Suit { Spade, Diamond, Club, Heart };
    public int suit () { return suitValue; }
    public int rank () { return rankValue; }
    private int suitValue;
    private int rankValue;
}

c#

enum Suits {Spade, Diamond, Club, Heart};

class PlayingCard {
    public Suits suit () { return suitValue;}
    public int rank () { return rankValue; }
    private Suits suitValue;
    private int rankValue;
}
Methods

In the next revision of our playing card abstraction we make the following changes:

- We add a **method** that will return the face color of the card, either red or black

- We add a **data field** to maintain whether the card is face up or face down, and **methods** both to test the state of this value and to flip the card
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 the name and **argument types**. Methods with the same name can be distinguished by different signatures
- A **return type**
- A **body**, which is the code that will be executed when the method is invoked in response to a message
Chapter Summary

In this chapter we have examined the static, or compile time features of classes:

- The syntax used for class definition
- The meaning of visibility modifiers (public and private)
- The syntax used for method definition
- Accessor or getter methods, and mutator or setter methods

Variations on class themes

- Interfaces
- Properties
- Nested classes
- Class data fields
Messages, Instances and Initialization
Dynamic Aspects of Classes

In the last chapter we described the static, or compile time aspects of classes.

In this chapter we examine their run-time features:

• How values are instantiated (or created) ?

• How they are initialized ?

• How they communicate with each other by means of message passing ?
Roadmap

In this chapter we examine:

• Message passing syntax

• Object creation and initialization (constructors)

• Accessing the receiver from within a method

• Memory management or garbage collection
Metaclasses

In Smalltalk (and Objective-C) classes are just objects, instances of class `Class`.

`new` is just a message given to a class object.

If we want to create constructors, where do we put them?

• They can't be part of the collection of messages of instances of the class, since we don't yet have an instance.

• They can't be part of the messages understood by class `Class`, since not all classes have the same constructor message.

Where do we put the behavior for individual class instances?
Inheritance and Substitution
Roadmap

In this chapter we will start to investigate the concepts of inheritance and substitution:

• The intuitive and practical meanings of inheritance
• The syntax used to describe inheritance and substitution
• Subclass vs. Subtype
• Some of the various forms of inheritance
• The benefits and costs of inheritance
Forms of Inheritance

The choices between inheritance and overriding, subclass and subtypes, mean that inheritance can be used in a variety of different ways and for different purposes.

Many of these types of inheritance are given their own special names. We will describe some of these specialized forms of inheritance:

- Generalization or Extension
- Specialization
- Specification
- Construction
- Limitation
- Variance
Inheritance and Substitution
Subclasses and Subtypes
Static and Dynamic Behavior
Static and Dynamic Behavior
In this chapter we will examine how differences in static and dynamic features effect object-oriented programming languages:

• Static versus Dynamic Typing

• Static and Dynamic Classes in Statically Typed Languages

• Static and Dynamic Method Binding in Statically Typed Languages
What do the terms Static and Dynamic Mean?

In programming languages:

- **Static** almost always means fixed or bound at compile time, and cannot thereafter be changed.

- **Dynamic** almost always means not fixed or bound until run time, and therefore can change during the course of execution.
Static and Dynamic Typing

In a statically typed programming language (Java or Pascal), for example, variables have declared typed -- **fixed at compile time**

In a dynamically typed programming language (Smalltalk or LISP), a variable is just a name. Types are associated with values, not variables.

*A variable can hold different types during the course of execution*
Arguments for and Against

• **Static typing** allows better error detection, more work at compile time and hence faster execution time

• **Dynamic typing** allows greater flexibility, easier to write (for example, no declaration statements)
Static Class and Dynamic Class

• In a statically typed OO language we say the class of the declaration is the *static class* for the variable

• While the class of the value it currently holds is the *dynamic class*

• Most statically typed OO languages constrain the dynamic class to be a child class of the static class

```plaintext
var
    pet : Mammal;
    fido : Dog
begin
    pet := fido; // static class is Mammal, dynamic class is Dog
end;
```
Static and Dynamic Method Binding

Should the binding for information be associated with the static class of a variable or the dynamic class?

Alice holds a small Mammal - asks Bill “does this animal give birth to live young”:

• **Static answer**: All mammals give birth to live young - therefore yes

• What if the Mammal is a platypus?

• **Dynamic answer**: Platypus lay eggs, therefore no

Even statically typed OOP languages can use dynamic binding. But may use static type to determine legality of operation.
Method Binding in C++

C++ is the most complex language. Not only must the programmer use the virtual keyword, but true polymorphism only occurs with pointer or reference variables.

We will see an explanation for the curious C++ semantics when we discuss memory management in the next chapter.

```cpp
class Animal {
public:
    virtual void speak () { cout << "Animal Speak !\n"; }
    void reply () { cout << "Animal Reply !\n"; }
};

class Dog : public Animal {
public:
    virtual void speak () { cout << "woof !\n"; }
    void reply () { cout << "woof again!\n"; }
};

class Bird : public Animal {
public:
    virtual void speak () { cout << "tweet !\n"; }
};

Animal * a;
Dog * b = new Dog();
b->speak();
woof !
a = b;
a->speak();
woof !
Bird c = new Bird();
c->speak();
tweet !
a = c;
a->speak();
tweet !
```
Implications of Substitution
Roadmap

In this chapter we will investigate some of the implications of the principle of substitution in statically typed object-oriented programming languages.

In particular, we will consider:

• The impact on memory management
• The meaning of assignment
• The distinction between testing for identity and testing for equality
Memory Allocation – Stack and Heap Based

Generally, programming languages use two different techniques for allocation of memory:

• **Stack-based allocation**
  • Amount of space required is determined at compile time, based on static types of variables
  • Memory allocation and release is **tied to procedure entry/exit**
  • Can be **performed very efficiently**

• **Heap-based allocation**
  • Amount of space used can be determined at **run-time**, based upon dynamic considerations
  • Memory allocation and release is **not tied to procedure entry/exit**, and either must be handled by user or by a run-time library (**garbage collection**)
  • Generally considered to be **somewhat less efficient**
Assigning a Larger Value to a Smaller Box

x
---
height
width
...

---
y
height
Width
...
Content
cursorLocation
...

44