

Object- Oriented Programming & Design

Part I: OO Fundamentals

Instructor : David Leberknight

These notes may be found as PDF files on-line :

www.softwarefederation.com/cs4448.html

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

Example: David walks his dog, Leroy

- *Find the objects...*

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(1) Object- Oriented Fundamentals - 2

Why 'Object-Oriented'?

- OO methods make it easier to build systems that:
 - solve the right problem
 - work properly
 - are maintainable
 - are extensible
 - are reusable
- Many people find OO *easier to comprehend*.
- The implementation can be less complex.
- Union of data and function is natural.
- Smaller conceptual gap between analysis and implementation.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 3

Why 'Object-Oriented'? (cont.)

- A well-designed set of objects is resilient to reuse and change.
- The modeling process helps create a common vocabulary and shared understanding between developers and users / clients.
- You don't need to be a computer programmer to understand an object model.
- You can make more \$\$\$:^)

Objects allow all of these benefits, but do not provide them.

This course will NOT turn you into a good OO designer; only experience (and a properly configured brain) can do that.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 4

What is this course all about?

- Learn the object-oriented paradigm.
- Learn a subset of the Unified Modeling Language (UML) for visual object modeling.
- Learn why some designs are better than others.
- Learn how to implement these Object-Oriented designs in Java.
- Learn some object-oriented Design Patterns.
- Learn about some of the latest and greatest commercial OO technology.
- Learn about a process to make best use of this technology.
- Be prepared for further study, and to work on an OO project.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 5

The Big OO Concepts

- **Models**
- **Objects**
- **Classes**
- **Encapsulation**
- **Abstraction**
- **Inheritance**
- **Polymorphism**
- **Generalization**

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 6

The Procedural Approach

- System is organized around procedures.
- Procedures send data to each other.
- Procedures and data are clearly separated.
- Focus on data structures, algorithms and sequencing of steps.
- Procedures are often hard to reuse.
- Lack of expressive and powerful visual modeling techniques.
- Transformation of concepts between analysis & implementation.
- This programming paradigm is essentially an abstraction of machine / assembly language.
- Design models are a long step from implementation.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 7

The Object-Oriented Approach

- System is organized around objects.
- Objects send messages (procedure calls) to each other.
- Related data and behavior are tied together in objects.
- Modeling of the domain as objects so that the implementation naturally reflects the problem at hand.
- Visual models are expressive and relatively easy to comprehend.
- Focus on responsibilities & interfaces before implementation.
- Powerful concepts: interfaces, abstraction, encapsulation, inheritance, delegation & polymorphism.
- Visual models of the problem evolve into models of the solution.
- Design models are only a small step from implementation.
- Software is very complex - strive to reduce complexity!

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 8

Example: Temperature Conversion

- The Procedural / Functional approach:

```
float c = getTemperature(); // assume Celcius
float f = toFahrenheitFromCelcius( c );
float k = toKelvinFromCelcius( c );
float x = toKelvinFromFahrenheit( f );
float y = toFahrenheitFromKelvin( k );
```

- The OO approach:

```
Temp temp = getTemperature();
float c = temp.toCelcius();
float f = temp.toFahrenheit();
float k = temp.toKelvin();
```

- What is Temp's internal unit for holding its value?

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 9

Modeling

Successful programs solve problems in the real world.

- They correspond closely to the real world problems they solve.
- They model the problem domain and users' activities.

Modeling enables better communication and visualization *for all stakeholders*. Successful OO designs almost always begin with a visual “object model” of the problem domain, involving both the domain experts and software designers alike.

Would you let a contractor build your new house without blueprints?

The essence of modeling is to show all *pertinent* detail.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 10

Objects

- Represent real or abstract *things*, with a name.
- Have well-defined *responsibilities*.
- Exhibit well-defined *behavior*.
- Have a well-defined *interface*, which is as simple as possible.
- Are self-consistent, coherent, and complete.
- Are (usually) not very complex or large.
- Have knowledge of themselves and the *interfaces* of a small number of other objects.
- Are “team players” with a small number of other objects.
- Are as *loosely coupled* with other objects as possible.
- Are well documented, so that others may (re)use them.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 11

Objects (cont.)

- Objects are *instances* of classes, each with a unique *identity*.
- A class defines both the interface(s) and the implementation for a set of objects, which determines their behavior.
- *Abstract classes* are classes that can have no instances.
- Whenever there is an *Abstract classes* such as Pet, there are usually some derived *concrete* classes such as Dog & Cat, which can be instantiated.
- Some OO languages (such as Smalltalk) support the concept of a *metaclass*, which allows the programmer to define a class on-the-fly, and then instantiate it. In this case, the class is an object, whose class is the metaclass.
- An object, once instantiated, cannot change its class.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 12

Characteristics of Objects

- They have unique identity.
- They fall into categories, or classes.
- They fall into hierarchies or aggregations.
- They have well defined behaviors & responsibilities.
- They separate interface from implementation.
- They hide their internal structures.
- They have states.
- They provide services.
- They send messages to other objects.
- They receive messages from other objects, and react appropriately.
- They delegate responsibilities to other objects.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 13

Classes

- A collection of objects, sharing common attributes and behaviors.
- The classification(s) of a collection of objects often depends on the attributes in which you are interested.

Examples: streets, roads, and highways...

Different programs would classify these differently...

- Traffic simulator:
 - one-way, two-way, divided, residential, limited access.
 - location w/ respect to business commuters.
- Maintenance scheduler:
 - surface material.
 - heavy truck traffic.
 - location w/ respect to congressional district.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 14

Classes, cont.

- Classes themselves can have attributes and behaviors.

Example: Class Employee, in a pension management program.

- Total number of Employees.
- How many Employees are fully vested?

- Different languages provide differing support for classes:
 - Smalltalk treats classes as objects (very helpful).
 - C++ provides minimal support (annoying at times).
 - Java is somewhere in between.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 15

Classes, cont.

- Classes are objects too
- A class can have attributes
 - An Employee class may have a list of all its instances
 - A LotteryTicket class may have a seed it uses to generate random ticket numbers; that seed is *shared* by all instances of the class.
- A class can have behaviors
 - An Employee class may have a `getEmployeeBySerialNum` behavior
 - A LotteryTicket class may have a `generateRandomNumber` behavior

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 16

Encapsulation

Exposing only the details that are relevant: the *public interface*.

What? not *How?*

- Hiding the “gears and levers”.
- Exposing only the client’s view of the object’s responsibilities.
- Protects the object from outside interference.
- Protects other objects from relying on details that are likely to change.
- *Information hiding* promotes *loose coupling* between objects and modularity, which promotes flexible design, which promotes reusability.
- Reduces interdependencies in the code.
- “Good fences make good neighbors”.

Example: Your car’s gas pedal.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 17

Encapsulation (cont.)

- Best practice: Objects only speak to each other by method (i.e. function) calls. They never directly access each other’s attributes.

```
class Person {  
    public int age;  
}
```

```
class BetterPerson {  
    private int age; // change to dateOfBirth  
    public int getAge() { return age; }  
}
```

// An even better Person class would have: **private dateOfBirth**

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 18

Abstraction

Abstraction allows *generalizations*.

- Simplify reality - ignore complex details.
- Focus on commonalties, but allow for variations.

Human beings often use generalizations.

When you see a gray German Shepherd named Rex owned by John and Jane Doe..., etc... - Do you think *dog*?

Abstraction can simplify computer programs too.

In software, for example, two important abstractions are *clients* and *servers*.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 19

Abstraction Example

- In a Graphical User Interface (GUI), imagine that the system has to ask the user a series of questions.
 - YesNoQuestions
 - ChooseOneQuestions
 - NumericQuestions
 - FreeTextQuestions
- It simplifies things to treat these all uniformly, each as a specialization of Question; the program will maintain a list of Questions, and invoke the askTheUser() method for each.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 20

Inheritance

Inheritance is a way of describing a class by saying how it differs from another class.

- Example: “Class Y is like Class X except for the following differences...”

Why use inheritance?

- When you have two types where one is necessarily an extension of the other.
- Sometimes (but not all the time) you are going to want to ignore the differences and look only at the what they have in common (the base class). This is called generalization.

Consider a system that can manipulate various kinds of shapes (the classic example):

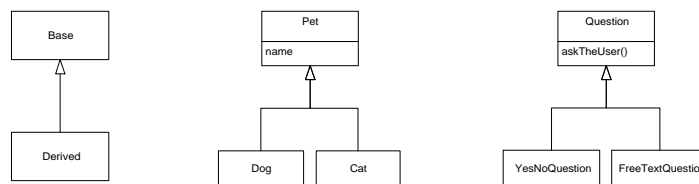
- Sometimes you don’t care what particular kind of shape you have (example: move to a different location).
- Sometimes you do care (example: draw the shape on a display).

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 21

Inheritance (cont.)



- The Derived class *inherits* from the Base class; the Derived class *extends* the Base class; the Derived class is a *specialization* of the Base class.
- The Derived class may provide additional *state* (member data), or additional *behavior* (member functions/methods), or it may *override* the implementation of inherited methods.
- The Base class is a *generalization* of all its Derived classes; one could say: in general, all Pets have names.
- *Base class = parent class = superclass.*
- *Derived class = child class = subclass.*

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 22

Inheritance (cont.)

- Implied by (some, but not all) **is-a** or **is-a-kind-of** relationships.
 - A Square is-a-kind-of Shape (uses inheritance).
 - Leroy is-a Dog (doesn't use inheritance).
- Has no good analog in traditional procedural analysis and design.
- Very powerful mechanism.
 - Enables focusing on the general, rather than the specific.
 - Allows code reuse and resilience to change.

Implementation inheritance: Derived classes inherit the *attributes and behaviors* (code) of their base classes.

Interface inheritance: Classes implement the abstract *interface* methods, preserving the intended semantics.

- In C++ it is possible to have multiple implementation inheritance.
- Java allows a Derived class to inherit the *implementation* of only one Base class, but it is fine to inherit multiple *interfaces*.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 23

Polymorphism

“The ability of two or more classes of objects to respond to the same message, each in its own way.”

“The ability to use any object which implements a given interface, where the specific class name need not be specified.”

Example:

– `question.askTheUser();`

- To be useful, the responses should be similar in nature.
- Made possible via *interface inheritance & dynamic (run-time) binding*.

Liskov substitution principle:

– If Y is a subclass of X, then it should be possible to use any instance of Y wherever any instance of X is used.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 24

Java code to demonstrate polymorphism

```
// File: question/QuestionTest.java
// What will the following Java code output to the screen?
// Refer to the Beginning Java link on the course web site.
package question;

abstract class Question // Full class name is question.QuestionTest
{
    public Question( String _text ) // Constructor
    {
        theText = _text;
    }

    public abstract void askTheUser();
    protected String theText;
}

```

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 25

Java code to demonstrate polymorphism (cont.)

```
class YesNoQuestion extends Question
{
    public YesNoQuestion( String _text ) { super( _text ); }
    public void askTheUser()
    {
        System.out.println( theText );
        System.out.println( "YES or NO ...?" );
    }
}
class FreeTextQuestion extends Question
{
    public FreeTextQuestion( String _text ) { super( _text ); }
    public void askTheUser()
    {
        System.out.println( theText );
        System.out.println( "Well...? What's the answer...?" );
    }
}

```

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 26

Java code to demonstrate polymorphism (cont.)

```
public class QuestionTest
{
    public static void main( String[] args )
    {
        Question[] questions = getQuestions();
        for( int i = 0; i < questions.length; i++ )
        {
            questions[ i ].askTheUser(); // Polymorphism !!!
        }
    }
    private static Question[] getQuestions()
    {
        Question[] qs = new Question[ 2 ];
        qs[0] = new YesNoQuestion( "Do you understand polymorphism?" );
        qs[1] = new FreeTextQuestion( "Why is polymorphism good?" );
        return qs;
    }
}
```

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 27

More Example Java code

```
// File: Derived.java
// What will the following Java code output to the screen?
class Base {
    final void foo() { System.out.println( "Base foo" ); }
    void bar() { System.out.println( "Base bar" ); }
}
public class Derived extends Base {
    void bar() { System.out.println( "Derived bar" ); }
    public static void main( String[] args ) {
        Derived d = new Derived();
        d.foo();
        d.bar();
        Base b = d;
        b.bar();
    }
}
```

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 28

Why OO Works: Reducing Complexity

- **Encapsulation** exposes only the public interface, thereby hiding the complex implementation details, and avoiding complex interdependencies in the code.
- **Polymorphism** allows different classes with the same interface to be interchangeable, thereby reducing code complexity.
- **Inheritance** from abstract classes and/or interfaces serves to reduce complexity by allowing *generalizations*.
- **Delegation** reduces complexity by building more complete or higher-level services from smaller, encapsulated ones. Delegation also provides increased *run-time* flexibility.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 29

Why OO Works: Linguistics & Cognition

Nouns are the primary words we use. We then qualify them with modifiers and attributes. Finally, we associate them with verbs.

- Object-oriented design follows this pattern.
- Procedural design does not.

This is one reason why people often find objects easier to understand.

We can form good Subject-verb-object sentences from an object model:

- People own pets.
- David owns Leroy.

Try doing *that* with functional decomposition!

- Furthermore, humans make heavy use of abstractions & generalizations...

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 30

OO As a Natural Step in Programming Evolution

- At first there was Machine Language.
- Then Assembly Language improved on that, providing symbols.
- High-level languages were introduced: Fortran, Pascal, C, etc... These provided “structure” to the graph of program statements.
- The use of “gotos” has gradually declined. This has helped to simplify program structure.
- Data structures and algorithms provide reusable patterns of program structure. This has promoted thinking at higher levels of abstraction.
- Object-oriented abstractions are primarily based on the problem to be solved rather than on the machine.
- The graph of program statements has become a less complicated graph of object collaborations, at a higher-level of abstraction.
- Design Patterns provide reusable object structures . . .

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 31

More OO Benefits

- Components are good... code reuse.
- Design patterns are good... design reuse.
- Interfaces are good... flexible and robust code.
- Infrastructure and reusable services are also good.
- Interfaces are ideal for partitioning individual & team responsibilities, increasing team efficiency.
- Loose coupling & modularity facilitate extensibility, flexibility, scalability & reuse.
- Logical changes are naturally isolated thanks to object modularity and information hiding (encapsulation). This leads to faster implementation of such changes, and easier maintenance.
- OO middleware offers location, platform & language transparencies.
- A well-designed set of objects allows new functionality to be added incrementally & non-invasively.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 32

Good Object-Oriented Design

- More art than science.
- No model that solves the problem at hand is intrinsically wrong, but some models are better than others because they prove to be more useful, flexible, extensible, easier to understand, less complex...
- The first design is seldom the best one. It is not always easy to find the best abstractions for modeling a problem. Experience helps.
- It sometimes takes several tries to see where to best draw boundaries between the parts of a system. What interface does each part show to the other parts?
- Be architecture-centric, rather than feature-centric. Focus on the general; defer the specific. If you get this right first, the rest is much more likely to be good.
- Predictable extensions should be designed to be accomplished addedly and non-invasively; two different designs might yield identical functionality and yet be very different in this respect.
- Strive to generalize service-oriented infrastructure for reuse, so that the code can quickly & easily evolve with the inevitable changes in requirements.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 33

Responsibility

- The most widely adopted approach to OO analysis.
 - Based on “**client / server**” relationships.

There are 2 common interpretations of “client / server”:

- Used in distributed architectures where the server provides access to shared resources (such as a database) and the clients provide the user interface.
- Used in the object-oriented vernacular where the server is an object that provides services; this is the meaning we are using here.

Clients *collaborate* with servers (by sending “messages”).

- An object may be a client in one collaboration, and a server in another.

Servers are *responsible* for providing certain services.

- In general, an object is *responsible* for behaving in a certain, well-defined way.

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 34

Overview of the Design Process

- Look at a domain and identify the objects and classes
 - Objects are often the first to be identified
 - Classes are found by grouping objects
- Identify the relationships between the objects and between the classes
 - Structural relationships
 - Collaborative relationships
- Assign responsibilities
 - Based on collaborative relationships
- Iterate, iterate, iterate, iterate, iterate, iterate, iterate, iterate, iterate.
- Begin by modeling the domain, not the solution!

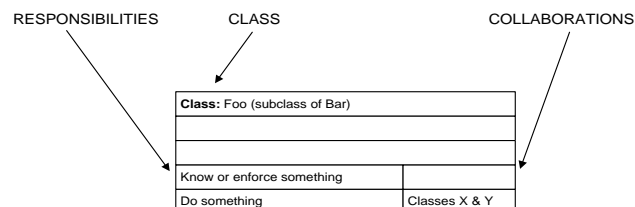
Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 35

CRC Cards

- The CRC approach uses 3 x 5 index cards, one per *class*, which shows its *responsibilities* and with which other class(es) it must *collaborate* in order to fulfill each responsibility. Write a brief description of the class on the back of the card.
- In this example, class Foo must collaborate with (send messages to) classes X & Y in order to fulfill its responsibility to be able to “do something.”



Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 36

Example: The “Sticks” Game

A program is to be written that allows two people to play a game against each other on a computer. The game consists of a layout with a number of sticks arranged in rows. When the game starts, they are arranged as shown here:

```
1: |
2: | |
3: | | |
4: | | | |
```

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 37

Rules of the Game

- Players alternate turns. During ones turn, a player removes one or more sticks from any non-empty row. The player who removes the last stick loses.
- At the start of the game, and after each move, the program will display the state of the game, indicate which player is to move, and prompt that player for the number of the row from which he/she wishes to remove sticks , and the number of sticks he/she wishes to remove.
- The program will tell the user when the specified move is invalid (i.e., asking to remove more sticks from a row than there are currently in that row).

Find the objects and classes...

Document using CRC cards.

Bonus Question: Who keeps track of whose turn it is?

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 38

CRC card for Row

RESPONSIBILITIES

CLASS

COLLABORATIONS

Class: Row	
Display	Stick
RemoveSticks	Stick

Copyright © 1996 - 2008

David Leberknight and Ron LeMaster. All rights reserved.

(I) Object- Oriented Fundamentals - 39