今天的内容

- Classes and objects – part II
- Foundations of the object model
- Elements of object model
  - Abstraction
  - Encapsulation
  - Modularization
  - Hierarchy

Class & Objects Part I I

齐琦

Inheritance

- Objects store data in fields; perform actions via operations(methods)
  - An object belongs to a particular category, which called a class
- Class determines form or template for its objects
  - The fields and the methods
- Inheritance (for class re-use, not the only mechanism)
  - Make a class like an existing class but with some variations
  - With some additions and modifications
  - Extends keyword

```
4   class GreatApe {
5     val weight = 100.0
6     val age = 12
7   }
8
9   class Bonobo extends GreatApe
10  class Chimpanzee extends GreatApe
11  class BonoboB extends Bonobo

13  def display(ape:GreatApe) =
14    s"weight: ${ape.weight} age: ${ape.age}"
15  display(new GreatApe) is
16  "weight: 100.0 age: 12"
17  display(new Bonobo) is
18  "weight: 100.0 age: 12"
19  display(new Chimpanzee) is
20  "weight: 100.0 age: 12"
21  display(new BonoboB) is
22  "weight: 100.0 age: 12"
```

- Base – derived class
- Parent – child class
- Superclass - subclass

This works at any level of inheritance

Inheritance, contd.

- Why this works?
  - Inheritance guarantees that anything inherits from GreatApe is a GreatApe
  - Any methods and fields in GreatApe will also in its children
- Enables a method(the display) that works not just with one type, but every class inherits that type
- Creates opportunities for code simplification and reuse

Base Class Initialization

- Ensures that all constructors are called --- guarantees correct object creation
  - Not just constructors for derived-class parts
  - Also constructor for base class
- After creates a object, Scala calls base-class constructor first, then the one for next-derived class, so on.
- Bonobo constructor can call methods in GreatApe class, but a GreatApe cannot know it's a Bonobo or a Chimpanzee

```
4   class GreatApe(
5     val weight:Double, val age:Int)
6
7   class Bonobo(weight:Double, age:Int)
8     extends GreatApe(weight, age)
9   class Chimpanzee(weight:Double, age:Int)
10    extends GreatApe(weight, age)
11  class BonoboB(weight:Double, age:Int)
12    extends Bonobo(weight, age)
13
14  def display(ape:GreatApe) =
15    s"weight: ${ape.weight} age: ${ape.age}"
16
17  display(new GreatApe(100, 12)) is
18  "weight: 100.0 age: 12"
19  display(new Bonobo(100, 12)) is
20  "weight: 100.0 age: 12"
21  display(new Chimpanzee(100, 12)) is
22  "weight: 100.0 age: 12"
```

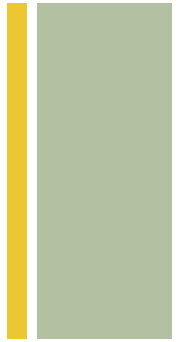
- Derived-class constructor must call primary base-class constructor
  - If there are auxiliary(overloaded) constructors in the base class, you may optionally call one of those
  - Derived-class constructor must pass appropriate arguments to base-class constructor

```scala
// AuxiliaryInitialization.scala
import com.atomicscala.AtomicTest._

class House(val address:String,
  val state:String, val zip:String) {
  def this(state:String, zip:String) =
    this("address?", state, zip)
  def this(zip:String) =
    this("address?", "state?", zip)
}

class Home(address:String, state:String,
  zip:String, val name:String)
  extends House(address, state, zip) {
    override def toString =
      s"$name: $address, $state $zip"
}

class VacationHouse(
  state:String, zip:String,
  val startMonth:Int, val endMonth:Int)
  extends House(state, zip)

class TreeHouse(
  val name:String, zip:String)
  extends House(zip)
```

```scala
val h = new Home("888 N. Main St.", "KS",
  "66632", "Metropolis")
h.address is "888 N. Main St."
h.state is "KS"
h.name is "Metropolis"
h is
"Metropolis: 888 N. Main St., KS 66632"

val v =
  new VacationHouse("KS", "66632", 6, 8)
v.state is "KS"
v.startMonth is 6
v.endMonth is 8

val tree = new TreeHouse("Oak", "48104")
tree.name is "Oak"
tree.zip is "48104"
```

- Home inherits from House, it passes appropriate arguments to primary House constructor
  - It also adds its own val argument
- Derived class can call any overloaded base-class constructors via derived-class primary constructor
- Can not call base-class constructors inside of overloaded derived-class constructors
- Can not inherit from a case class, as in Scala 2.10

Overriding Methods



- Overriding methods --- redefining a method from a base class (do sth different in a derived class)

```
4  class GreatApe {
5    def call = "Hoo!"
6    var energy = 3
7    def eat() = { energy += 10; energy }
8    def climb(x:Int) = energy -= x
9  }
10
11 class Bonobo extends GreatApe {
12   override def call = "Eep!"
13   // Modify the base-class var:
14   energy = 5
15   // Call the base-class version:
16   override def eat() = super.eat() * 2
17   // Add a new method:
18   def run() = "Bonobo runs"
19 }
```

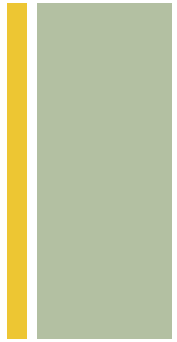
```
21 class Chimpanzee extends GreatApe {
22   override def call = "Yawp!"
23   override def eat() = super.eat() * 3
24   def jump = "Chimp jumps"
25   val kind = "Common" // New field
26 }
```

```
28 def talk(ape:GreatApe) = {
29   // ape.run()  // Not an ape method
30   // ape.jump // Nor this
31   ape.climb(4)
32   ape.call + ape.eat()
33 }
34
35 talk(new GreatApe) is "Hoo!9"
36 talk(new Bonobo) is "Eep!22"
37 talk(new Chimpanzee) is "Yawp!27"
```

- Style convention
  - Call method not change internal state of object, so not use parentheses; eat change internal state, so uses parenthese
- Overriding
  - Create identical method signature in derived class as exists in a base class
  - Substitute behavior defined in base with new behavior
  - Override keyword
- Polymorphism
  - Treat Bonobo or Chimpanzee as ordinary GreatApes in "talk" method, call method produces correct behavior each case
  - Talk somehow knows exact type of object, calls appropriate variation of call.

- Super.eat()
  - What if want to call base-class version
  - If simply call “eat”, calling same method currently inside (recursion)
  - Specify base-class version, short for “superclass”

Enumerations



- Enumeration
  - A collection of names
  - Enumeration class

```
4  object Level extends Enumeration {
5    type Level = Value
6    val Overflow, High, Medium,
7        Low, Empty = Value
8  }
9
10 Level.Medium is "Medium"
11 import Level._
12 Medium is "Medium"
13
14 { for(n <- Range(0, Level.maxId))
15     yield (n, Level(n)) } is
16   Vector((0, Overflow), (1, High),
17     (2, Medium), (3, Low), (4, Empty))
18
19 { for(lev <- Level.values)
20     yield lev }.toIndexedSeq is
21   Vector(Overflow, High,
22     Medium, Low, Empty)
```

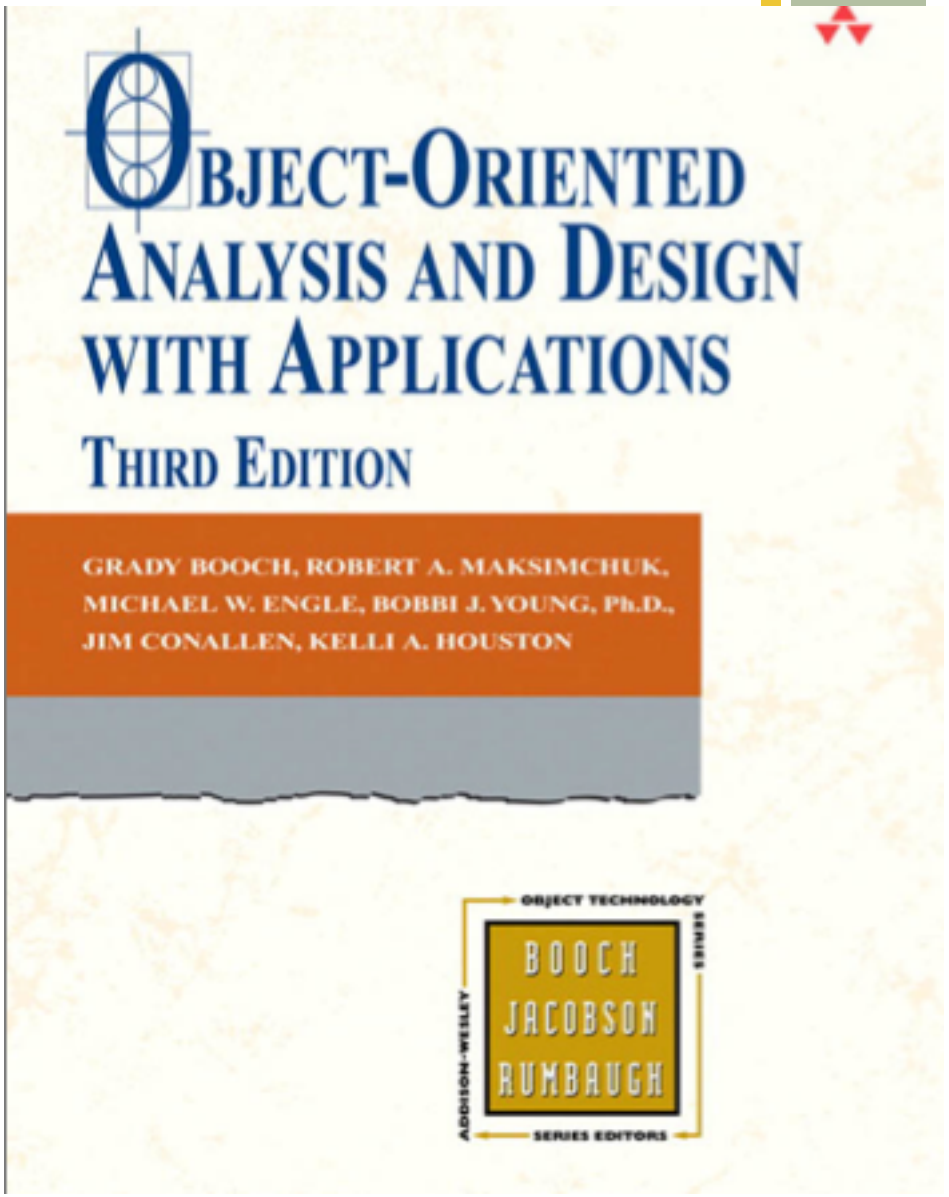
```
23
24 def checkLevel(level:Level)= level match {
25   case Overflow => ">>> Overflow!"
26   case Empty => "Alert: Empty"
27   case other =>  s"Level $level OK"
28 }
29
30 checkLevel(Low) is "Level Low OK"

31 checkLevel(Empty) is "Alert: Empty"
32 checkLevel(Overflow) is ">>> Overflow!"
```

- A new Value is created for each of the vals
- Create an object not create a new type the way creating a class does
- Use type keyword, alias Level to Value
  - Level becomes a new type
- Id field in Value, which incremented each time a new Value created
- Iterate through enumeration names, values field

Evolution of the Object Model

Another Textbook

- "Object-Oriented Analysis and Design with Applications"
  - 3rd edition
  - Brady Booch, etc.
  - Addison-Wesley



- Two sweeping trends

1. The shift in focus from programming-in-the-small to programming-in-the-large
2. The evolution of high-order programming languages

- Complexity in software system prompted applied research in software engineering
  - Decomposition
  - Abstraction
  - Hierarchy
- Needs more expressive programming languages

Generations of programming languages

- First-generation languages (1954–1958)

FORTRAN I	Mathematical expressions
ALGOL 58	Mathematical expressions
Flowmatic	Mathematical expressions
IPL V	Mathematical expressions

- Second-generation languages (1959–1961)

FORTRAN II	Subroutines, separate compilation
ALGOL 60	Block structure, data types
COBOL	Data description, file handling
Lisp	List processing, pointers, garbage collection

- Third-generation languages (1962–1970)

PL/1	FORTRAN + ALGOL + COBOL
ALGOL 68	Rigorous successor to ALGOL 60
Pascal	Simple successor to ALGOL 60
Simula	Classes, data abstraction

- The generation gap (1970–1980)

Many different languages were invented, but few endured. Howe lowing are worth noting:

C	Efficient; small executables
FORTRAN 77	ANSI standardization

- Object-orientation boom (1980–1990, but few languages survive)

Smalltalk 80	Pure object-oriented language
C++	Derived from C and Simula
Ada83	Strong typing; heavy Pascal influence
Eiffel	Derived from Ada and Simula

- Emergence of frameworks (1990–today)

Much language activity, revisions, and standardization have occurred, leading to programming frameworks.

Visual Basic	Eased development of the graphical user interface (GUI) for Windows applications
Java	Successor to Oak; designed for portability
Python	Object-oriented scripting language
J2EE	Java-based framework for enterprise computing
.NET	Microsoft's object-based framework
Visual C#	Java competitor for the Microsoft .NET Framework
Visual Basic .NET	Visual Basic for the Microsoft .NET Framework

- First-generation
  - Primarily for scientific and engineering apps, vocabulary entirely mathematics; write math formulas, freeing from assembly or machine code.
- Second-generation
  - Machine gets powerful, business application
  - Emphasis on algorithmic abstractions; tell machine what to do
- Third
  - Transistors advent; integrated circuit technology; hardware cost dropped
  - Demands of data manipulation; Support for data abstraction

- 70s
  - Thousand of different program languages;
  - Larger programs highlighted inadequacies of earlier languages
  - Few survived; but many concepts introduced adopted by successors
- Object-oriented (from 80s, 90s)
  - Object-oriented decomposition of software
  - Main streams: Java, C++, etc.
  - Emergence of frameworks (e.g. J2EE, .NET)

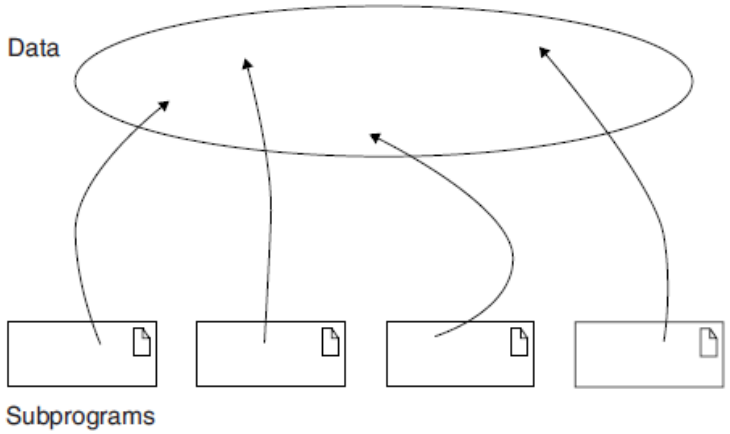


Figure 2–1 The Topology of First- and Early Second-Generation Programming Languages

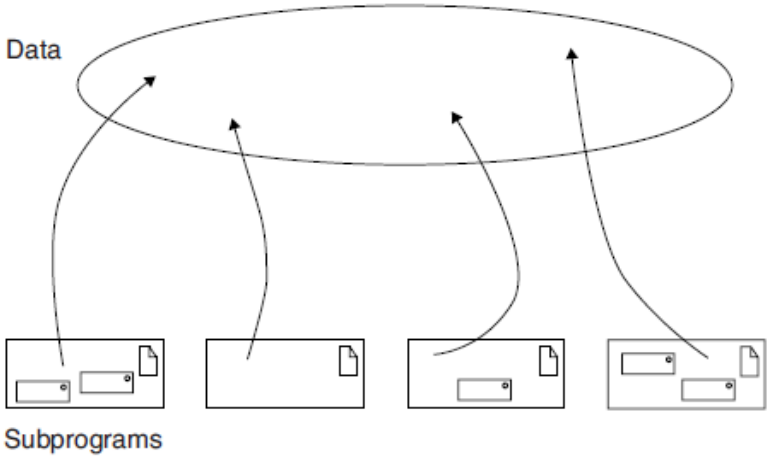


Figure 2–2 The Topology of Late Second- and Early Third-Generation Programming Languages

Subprograms as an abstraction mechanism

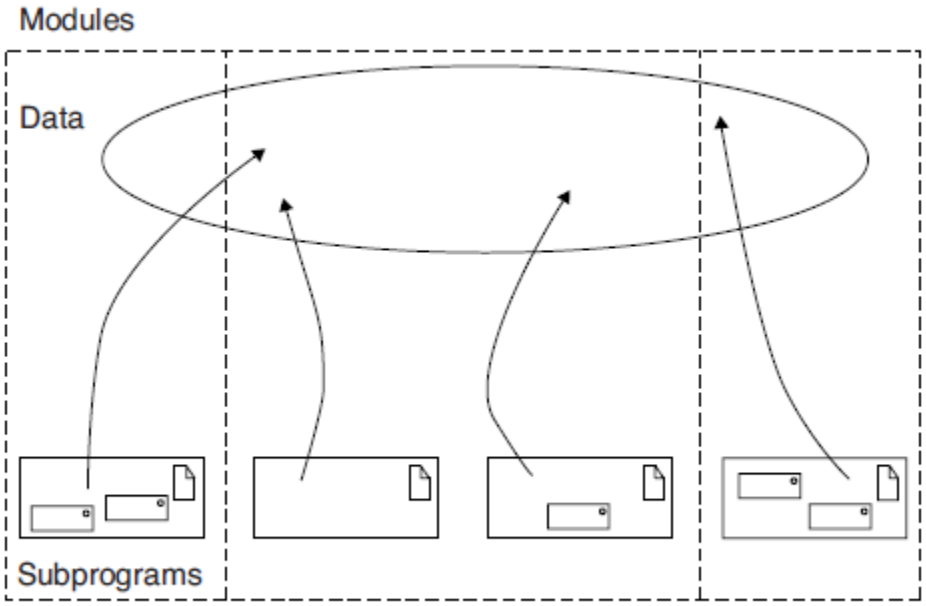


Figure 2–3 The Topology of Late Third-Generation Programming Languages

Modular structure

Most lacked support for data abstraction and strong typing, some errors can only detected during execution of program.

For object-oriented

- Data abstraction important to master complexity of problem.
- Physical building block is module(a logical collection of classes and objects instead of subprograms)

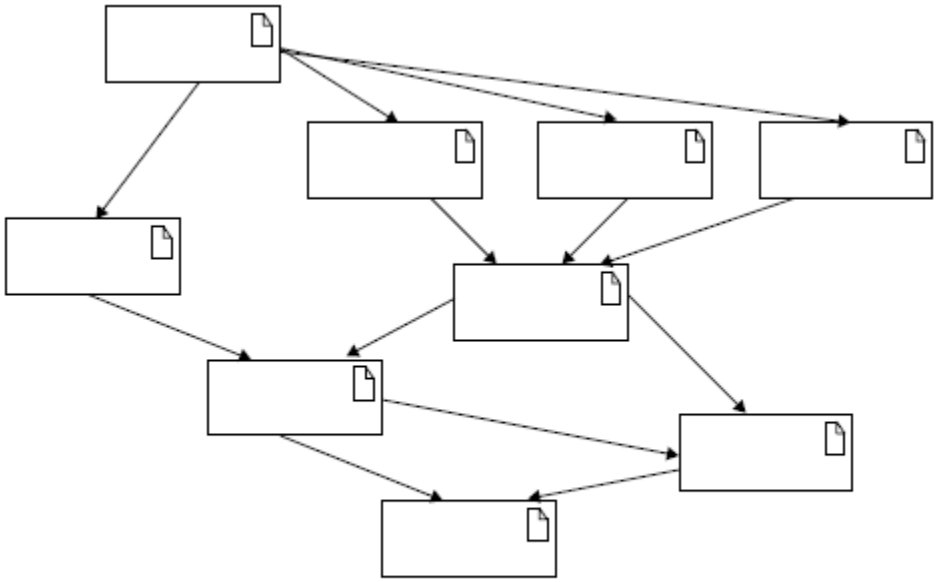


Figure 2–4 The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

For object-oriented

- Data and operations are united, that fundamental logical building blocks are no longer algorithms, but classes and objects
- Little or no global data

guages. To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns" [6]. For this reason, the

+

Foundations of Object Model

Object-oriented programming(OOP)

- Object orientation cope with complexity inherent in many different systems
  - Not just to programming languages, user interface design, databases, computer architectures
- OOP

  > Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

  - Uses objects as building blocks
  - Each object is an instance of some class
  - Classes relates to one another via inheritance

What's object-oriented?

- Cardelli and Wegner say:

[A] language is object-oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have an associated type [class].
- Types [classes] may inherit attributes from supertypes [superclasses]. [34]

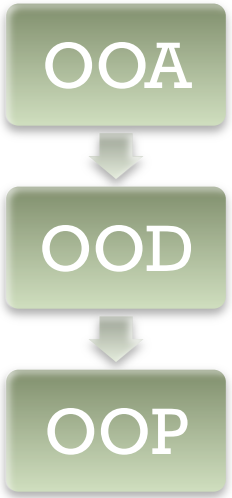
Object-oriented design(OOD)

- Leads to an object-oriented decomposition
- Uses different notations to express different models of logical (class and object structure), and physical (module and process architecture) design of a system

Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

Object-oriented analysis(OOA)

Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.



OOA serves OOD; OOD as blueprints for implementing system using OOP methods

Elements of the Object Model

Programming style

- No single one best for all kinds applications.
  - Knowledge base
  - Computation-intense operation
  - Broadest set of applications

1.	Procedure-oriented	Algorithms
2.	Object-oriented	Classes and objects
3.	Logic-oriented	Goals, often expressed in a predicate calculus
4.	Rule-oriented	If–then rules
5.	Constraint-oriented	Invariant relationships

Elements of object model

- Conceptual framework for object-oriented, is the object model
- Four major elements of this model(a model without any one of these is not object-oriented)
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
- Three minor elements: (useful but not essential)
  - Typing
  - Concurrency
  - Persistence

Meaning of Abstraction

- Define abstraction:

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

- Focus on outside view of an object, separate object's essential behavior from its implementation
- Decide right set of abstractions for a given domain, is central problem in OOD

Spectrum of abstraction

- From most to least useful:

Entity abstraction	An object that represents a useful model of a problem domain or solution domain entity
Action abstraction	An object that provides a generalized set of operations, all of which perform the same kind of function
Virtual machine abstraction	An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
Coincidental abstraction	An object that packages a set of operations that have no relation to each other

- A client is any object that uses resources of another object (known as server).
  - Characterize behavior of an object by considering services it provides to other objects
  - Force to concentrate on outside view of an object, which defines a contract on which other objects may depend, and which must be carried out by inside view
- Protocol: entire set of operations that contributes to the contract, with legal ordering of their invoking
  - Denotes ways that object may act and react, thus constitutes entire outside view of the abstraction.
- Terms: operation, method, member function virtually mean same thing.

Examples of Abstraction

- Farm, maintaining proper greenhouse environment
- A key abstraction is about a sensor
  - A temperature sensor: an object that measures temperature at a location
  - What are responsibilities of a temp sensor? Answers yield different design decisions

Abstraction: Temperature Sensor
Important Characteristics: temperature location
Responsibilities: report current temperature calibrate

Figure 2–6 Abstraction of a `Temperature Sensor`

Abstraction: Active Temperature Sensor
Important Characteristics: temperature location setpoint
Responsibilities: report current temperature calibrate establish setpoint

Figure 2–7 Abstraction of an `Active Temperature Sensor`

- No objects stands alone; every object collaborates with others to achieve some behavior.
- Design decisions about how they cooperate, define boundaries of each abstraction and the responsibilities and protocol of each object.

Meaning of Encapsulation

- Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.
- Encapsulation is most often achieved through information hiding (not just data hiding)
- Whereas abstraction "helps people to think about what they are doing," encapsulation "allows program changes to be reliably made with limited effort"
- Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns.
  - DB application, programs depend on a schema(data's logical view),not care physical data representation

- For abstraction to work, implementations must be encapsulated
  - each class must have two parts: an interface and an implementation
  - Interface – outside view, behavior abstraction
  - Implementation – achieve the behavior
- Define encapsulation
  - "Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."

Encapsulation contd.

- Encapsulates implementation details; no client need know about the implementation decisions
  - Because it not affect observable behavior of class
- As system evolves, implementation often changed to use more efficient algorithms
- Ability to change the representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

Meaning of Modularity

- Partition a program into individual components
  - Reduce complexity
  - Creates well-defined, documented boundaries within program
  - Examples
    - Smalltalk – class
    - Java – packages containing classes
    - C++, Ada – module construct
- Classes and objects form logical structure a system; place them in modules to produce system's physical architecture
  - Hundreds classes, to help manage complexity

- Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules.
- Deciding on the right set of modules for a given problem is almost as hard a problem as deciding on the right set of abstractions.
- Modules serve as the physical containers in which we declare the classes and objects of our logical design.

- Guidelines
  - overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently
  - In practice, the cost of recompiling the body of a module is relatively small: Only that unit need be recompiled and the application relinked
  - a module's interface should be as narrow as possible, yet still satisfy the needs of the other modules that use it.
    - cost of recompiling the interface of a module is relatively high
  - hide as much as we can in the implementation of a module

- The developer must therefore balance two competing technical concerns: the desire to encapsulate abstractions and the need to make certain abstractions visible to other modules.
  - strive to build modules that are cohesive (by grouping logically related abstractions) and loosely coupled (by minimizing the dependencies among modules)
- Define modularity
  - Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

- Additional technical issues may affect modularization decisions
  - Modules as units of a software can be reused across applications; package classes and objects into modules way that makes reuse convenient
  - many compilers generate object code in segments, one for each module; may be practical limits on the size of individual modules
- Modules also serve as the unit of documentation and configuration management. (more modules more docs)
- Identification of classes and objects is part of the logical design of the system, but identification of modules is part of the system's physical design.
  - These design decisions happen iteratively

Meaning of Hierarchy

- Encapsulation helps manage this complexity by hiding the inside view of our abstractions ; Modularity helps also, by giving us a way to cluster logically related abstractions.
- Define Hierarchy
  - Hierarchy is a ranking or ordering of abstractions.
- Two most important hierarchies in a complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

Examples of Hierarchy

- Single Inheritance
- Multiple Inheritance
- Aggregation