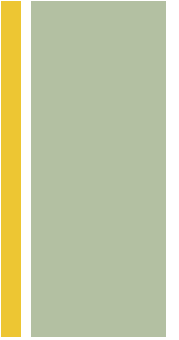
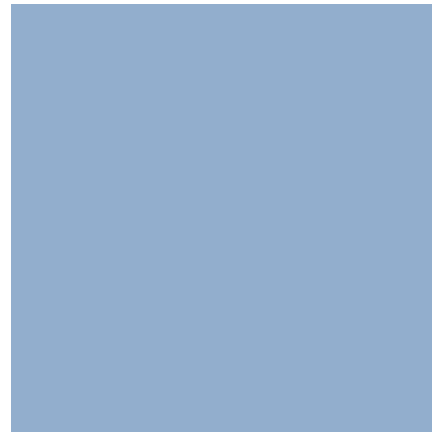




# 今天的内容



- Classes and objects – part IV
- Foundations of the object model
- Elements of object model
  - Abstraction
  - Encapsulation
  - Modularization
  - Hierarchy
  - Typing
  - Concurrency
  - Persistence



# Class & Objects

## Part IV

齐琦

# + Reaching into Java

```
scala> import java.util.Date
import java.util.Date
```

```
scala> val d = new Date
```

```
1 // LinearRegression.scala
2 import com.atomicscala.AtomicTest._
3 import org.apache.commons.math._
4 import stat.regression.SimpleRegression
5
6 val r = new SimpleRegression
7 r.addData(1, 1)
8 r.addData(2, 1.1)
9 r.addData(3, 0.9)
10 r.addData(4, 1.2)
11
12 r.getN is 4
13 r.predict(6) is 1.19
```

- Import Java classes
  - Entire Java standard library available using import like this
- Also can download third-party Java libs and use in Scala
  - E.g. apache common math lib
  - Linear regression usage example
- Rice Java libs, a huge benefit to Scala

# + Applications

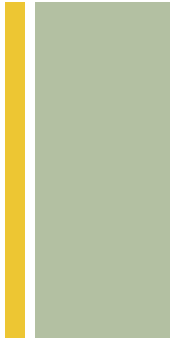
```
3  object WhenAmI extends App {  
4      hi  
5      println(new java.util.Date())  
6      def hi = println("Hello! It's:")  
7  }
```

scalac Compiled.scala

scala WhenAmI

```
1  // CompiledWithArgs.scala  
2  
3  object EchoArgs extends App {  
4      for(arg <- args)  
5          println(arg)  
6  }
```

- Object extends App
  - Constructor statements execute in order
- It doesn't matter what you call the file; the name of the resulting program depends on the name of the object.



- another form that follows a pattern used in older programming languages: you define a method called main, and the method arguments contain the command-line arguments.
- all the arguments come in as Strings. There's no particular reason to use a main other than that it might make the code more familiar to programmers from other languages (Java, in particular).

```
3  object EchoArgs2 {  
4      def main(args:Array[String]) =  
5          for(arg <- args)  
6              println(arg)  
7  }
```



# A little Reflection



- **Reflection** means taking an object and holding it up to a mirror, so it can discover things about itself.
- Example: take an object, find out its class name
  - Create a trait to add a toString to any class, to display the class name

```
2 package com.atomicscala
3 import reflect.runtime.currentMirror
4
5 object Name {
6   def className(o:Any) =
7     currentMirror.reflect(o).symbol.
8     toString.replace('$', ' ').
9     split(' ').last
10 }
11
12 trait Name {
13   override def toString =
14     Name.className(this)
15 }
```

```
3 import com.atomicscala.Name
4
5 class Solid extends Name
6 val s = new Solid
7 s is "Solid"
8
9 class Solid2(val size:Int) extends Name {
10   override def toString =
11     s"${super.toString}($size)"
12 }
13 val s2 = new Solid2(47)
14 s2 is "Solid2(47)"
```

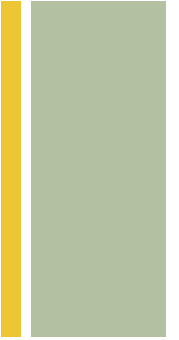
Scala's reflection API is much more powerful and complex than we've shown here.



+

Polymorphism

# + Polymorphism



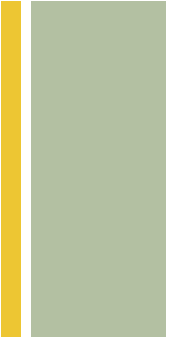
- Greek term, “many forms”
- In programming, it means we can perform the same operation on different types
- If we create a class using another class A along with traits B and C, we can choose to treat that class as if it were only an A or only a B or only a C
  - E.g. , animals, vehicles, mobile trait (ref. to code demo)





# Example: design a game

- Each element in the game will draw itself on screen based on its location
  - When two elements in proximity, they'll interact
  - Sketch a draft making use of polymorphism





```
1 // Polymorphism.scala
2 import com.atomicscala.AtomicTest._
3 import com.atomicscala.Name
4
5 class Element extends Name {
6     def interact(other:Element) =
7         s"$this interact $other"
8 }
9
10 class Inert extends Element
11 class Wall extends Inert
12
13 trait Material {
14     def resilience:String
```

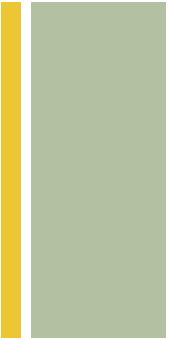
```
15 }
16 trait Wood extends Material {
17     def resilience = "Breakable"
18 }
19 trait Rock extends Material {
20     def resilience = "Hard"
21 }
22 class RockWall extends Wall with Rock
23 class WoodWall extends Wall with Wood
24
25 trait Skill
26 trait Fighting extends Skill {
27     def fight = "Fight!"
28 }
29 trait Digging extends Skill {
30     def dig = "Dig!"
31 }
32 trait Magic extends Skill {
33     def castSpell = "Spell!"
34 }
35 trait Flight extends Skill {
36     def fly = "Fly!"
37 }
```

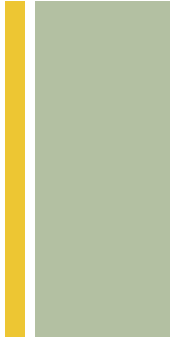
```
38
39 class Character(var player:String="None")
40     extends Element
41 class Fairy extends Character with Magic
42 class Viking extends Character
43     with Fighting
44 class Dwarf extends Character with Digging
45     with Fighting
46 class Wizard extends Character with Magic
47 class Dragon extends Character with Magic
48     with Flight
49
50 val d = new Dragon
51 d.player = "Puff"

52 d.interact(new Wall) is
53 "Dragon interact Wall"
54
55 def battle(fighter:Fighting) =
56     s"$fighter, ${fighter.fight}"
57 battle(new Viking) is "Viking, Fight!"
58 battle(new Dwarf) is "Dwarf, Fight!"
59 battle(new Fairy with Fighting) is
60 "1, Fight!" // Name: $anon$1
61
62 def fly(flyer:Element with Flight,
63     opponent:Element) =
64     s"$flyer, ${flyer.fly}, " +
65     s"${opponent.interact(flyer)}"
66
67 fly(d, new Fairy) is
68 "Dragon, Fly!, Fairy interact Dragon"
```

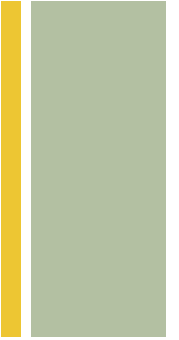


- Interact method on line 6
  - One element interacts with another
- Create different types of elements, and traits to mix in to achieve different effects
  - Traits can inherit from each other
- Skill trait on line 25, classify different traits; can add common fields or methods in Skill
- Characters; constructor argument player on line 39, has a default argument
- Create a Dragon on line 50, change player the name
- Line 52, first example of polymorphism
  - Interact method takes an Element or anything derived from Element. --- a polymorphism

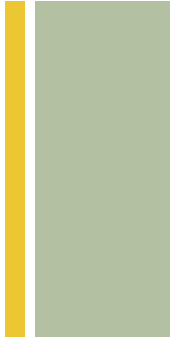




- That's powerful, because now method can also applies to anything that inherits from that type(method's argument type)
  - Transparent and safe, because Scala guarantees that a derived class "is a " base class, by ensuring that derived class has all methods of the base class
- Viking and Dwarf include Fighting trait, so they can be passed to battle --- demonstrates polymorphism; otherwise you must write specific methods
- Polymorphism is a tool allows you to write less code and make it more reusable



- Line 59, the argument passed to battle:
  - New Fairy with Fighting
  - Creates a new class, and immediately make an instance of that class; not give the class a name
- Line 62, argument flyer's type as "Element with Flight"
  - Arguments passed includes both Element and Flight, so fly can call everything it needs to
- "How did you know to do it this way?"
  - The Design challenge
  - Once decide what you want to build, many different ways to assemble it



- Create a base class and add new methods during inheritance, or mix in functionality using traits
- Design decisions
  - Using a combination of experience and observing the way your system is used
- Design process
  - Decide what makes sense based on the requirements of your system
- The pragmatic approach is not to assume that you can get it all right the first time. Instead, write something, get it working, then see how it looks. As you learn, “refactor” your code until the design feels right (don’t settle for the first thing that works).



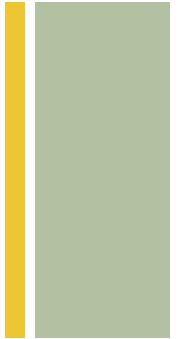
+

Composition





# Suppose modeling a house



```
1 // House1.scala
2
3 trait Building
4 trait Kitchen
5 trait House extends Building with Kitchen
```

- Reads nicely: “A house is a building with a kitchen”
- What if it has two kitchens? Can’t inherit the same trait twice
- Inheritance describes an is-a relationship
  - “A house is a building”
  - When the is-a relationship makes sense, inheritance usually makes sense
- A trait represents a capability, can say it is a *has-ability* relationship
  - “A house has ... kitchen ... ability”

# + Composition



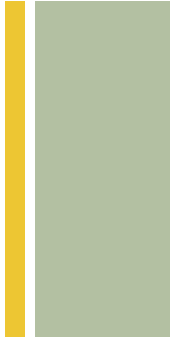
- Most fundamental relationship is not inheritance, nor traits, but *composition*
- Seems too simple to overlooked: just put something inside
- Composition is a has-a relationship
  - “The house has two kitchens”

```
1 // House2.scala
2
3 trait Building
4 trait Kitchen
5
6 trait House extends Building {
7     val kitchen1:Kitchen
8     val kitchen2:Kitchen
9 }
```

```
1 // House3.scala
2
3 trait Building
4 trait Kitchen
5
6 trait House extends Building {
7     val kitchens:Vector[Kitchen]
8 }
```



# Prefer composition to inheritance



- We spend a lot of time and effort understanding inheritance and mixins because they are more complex, but this often gives the impression that they are somehow more important.
- Composition produces simpler designs and implementations
- It's just that we tend to get bound up in those more complicated relationships, and the maxim "prefer composition to inheritance" is a reminder to step back and look at your design and wonder whether you couldn't simplify things using composition. Ultimately, the goal is to produce a good design by properly applying your tools.




- this approach doesn't allow it because you can't inherit a trait twice. Once you have an ability, adding that ability a second time doesn't mean anything.

```
1  // House4.scala
2
3  trait Building
4  trait Food
5  trait Utensils
6  trait Store[T]
7  trait Cook[T]
8  trait Clean[T]
9  trait Kitchen extends Store[Food]
10     with Cook[Food] with Clean[Utensils]
11     // Oops. Can't do this:
12     // with Store[Utensils]
13     // with Clean[Food]
14
15  trait House extends Building {
16     val kitchens:Vector[Kitchen]
17  }
```

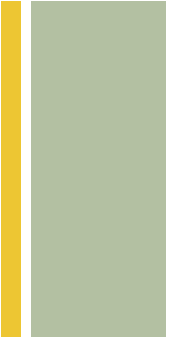


```
1 // House5.scala
2
3 trait Building
4 trait Room
5 trait Storage
6 trait Sink
```

```
7 trait Store[T]
8 trait Cook[T]
9 trait Clean[T]
10 trait Food extends Store[Food]
11   with Clean[Food] with Cook[Food]
12 trait Utensils extends Store[Utensils]
13   with Clean[Utensils] with Cook[Utensils]
14
15 trait Kitchen extends Room {
16   val storage:Storage
17   val sinks:Vector[Sink]
18   val food:Food
19   val utensils:Utensils
20 }
21
22 trait House extends Building {
23   val kitchens:Vector[Kitchen]
24 }
```



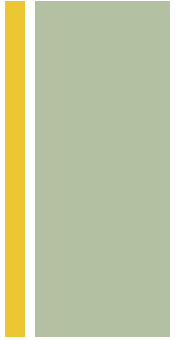
+ Using Traits (more about traits)



- Scala enables you to partition your model into appropriate pieces
- Traits (and the mixins they enable) might be the most powerful of these tools.
  - \* Prefer traits to more concrete types (more abstract == more flexible)
  - \* Divide models into independent pieces
  - \* Delay concreteness



# Difference between traits and abstract classes



- traits cannot have constructor arguments (although they can contain constructor expressions within the trait body)
- because a trait is more of a “capability” than it is a physical thing; a trait is designed for reuse rather than instantiation.





```
1 // AerobicExercise.scala
2 import com.atomicscala.AtomicTest._
3
4 trait Aerobic {
5     val age:Int
6     def minAerobic = .5 * (220 - age)
7     def isAerobic(heartRate:Int) =
8         heartRate >= minAerobic
9 }
10
```

```
11 trait Activity {
12     val action:String
13     def go:String
14 }
15
16 class Person(val age:Int)
17
18 class Exerciser(age:Int,
19     val action:String = "Running",
20     val go:String = "Run!") extends
21     Person(age) with Activity with Aerobic
22
23 val bob = new Exerciser(44)
24 bob.isAerobic(180) is true
25 bob.isAerobic(80) is false
26 bob.minAerobic is 88.0
```

Traits combine their functionality with another object, as **Exerciser** combines **Aerobic** and **Activity** with a **Person**. Note how the **age** field in **Person** satisfies the abstract **age** field in **Aerobic**. The definitions of **action** and **go** on lines 19-20 must be **vals** (with the same names as the fields in **Activity**) to make them fields in the resulting object, and to thus satisfy the requirements from **Activity**.



## + Tagging Traits & Case Objects



- A tagging trait is a way to group classes or objects together.
- an alternative to the approach in Enumerations


```
1 // TaggingTrait.scala
2 import com.atomicscala.AtomicTest._
3
4 sealed trait Color ←
5 case object Red extends Color
6 case object Green extends Color
7 case object Blue extends Color
8 object Color {
9   val values = Vector(Red, Green, Blue)
10 }
11
12 def display(c:Color) = c match {
13   case Red => s"It's $c"
14   case Green => s"It's $c"
15   case Blue => s"It's $c"
16 }
17
18 Color.values.map(display) is
19 "Vector(It's Red, It's Green, It's Blue)"
```

The hallmark of a tagging trait (Color, in this case) is that it only exists to collect some types under a common name, thus it typically has no fields or methods

The sealed keyword: no other subtypes of Color other than the ones you see here

A case object is like a case class except that it produces an object instead of a class

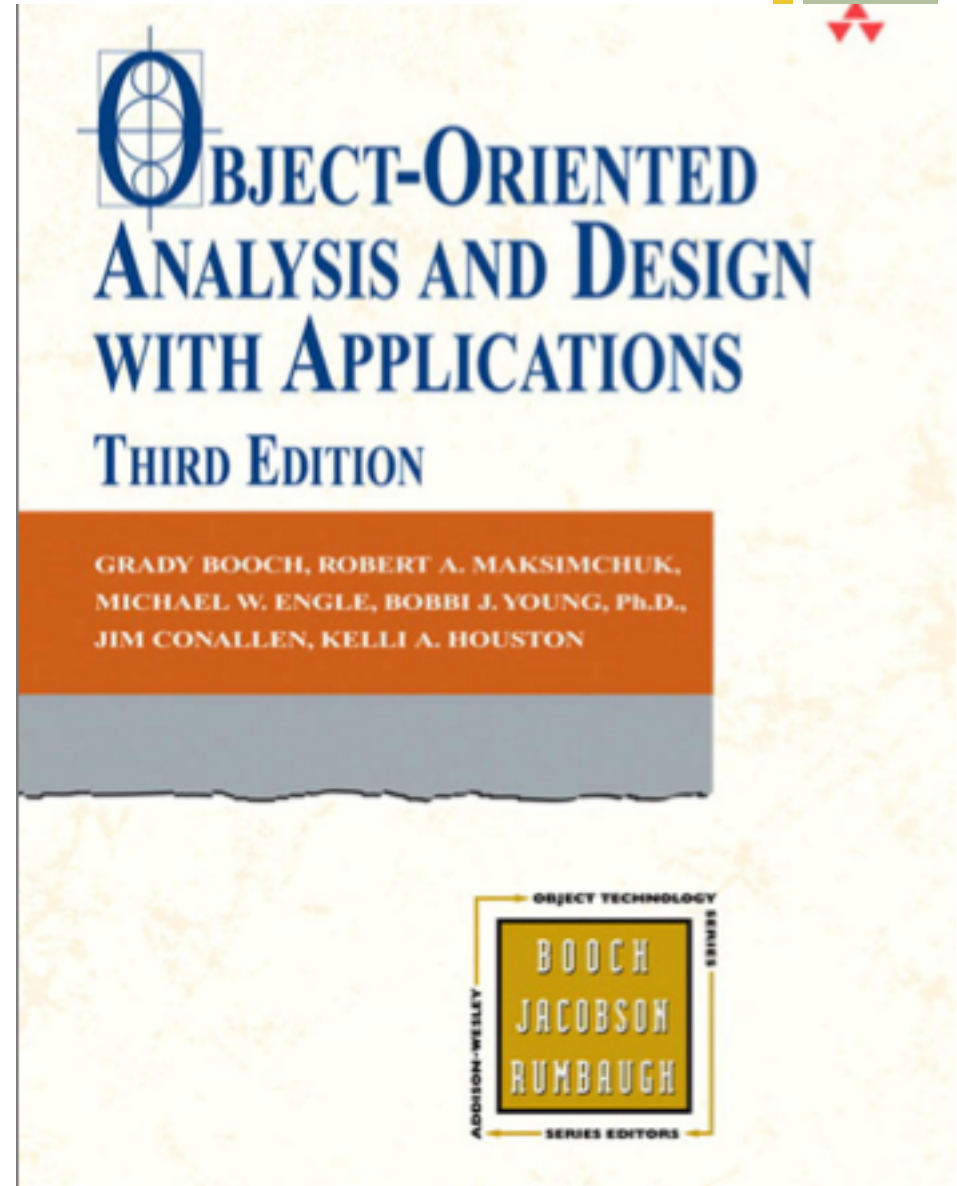
Note that the argument to display is the tagging trait Color. We can refer directly to any of the instances of the case objects

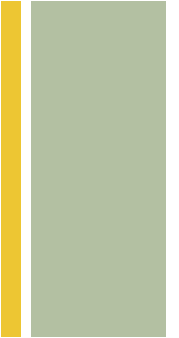


## + Evolution of the Object Model

# + Another Textbook

- “Object-Oriented Analysis and Design with Applications”
  - 3<sup>rd</sup> edition
  - Brady Booch, etc.
  - Addison-Wesley

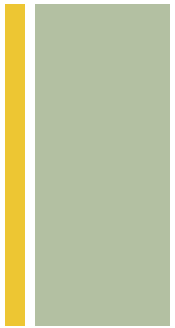




- Two sweeping trends
  1. The shift in focus from programming-in-the-small to programming-in-the-large
  2. The evolution of high-order programming languages
- Complexity in software system prompted applied research in software engineering
  - Decomposition
  - Abstraction
  - Hierarchy
- Needs more 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/I	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. How lowing are worth noting:

C	Efficient; small executables
FORTRAN 77	ANSI standardization

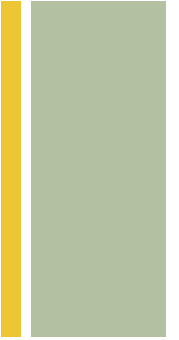
## ■ Object-orientation boom (1980–1990, but few languages survive)

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.

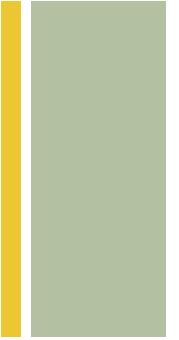
## ■ Second-generation

- Machine gets powerful, business application
- Emphasis on algorithmic abstractions; tell machine what to do

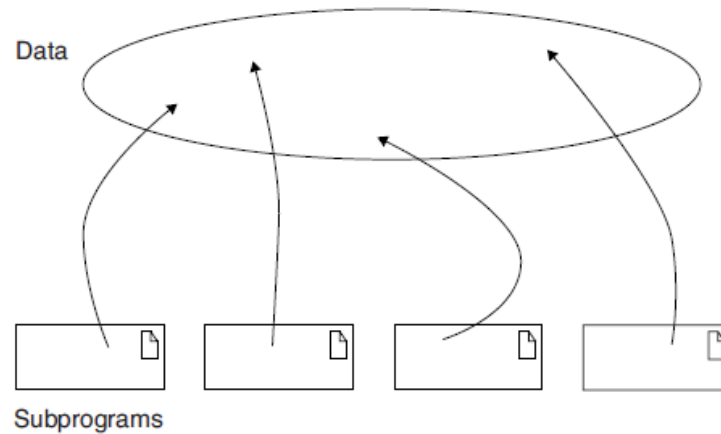
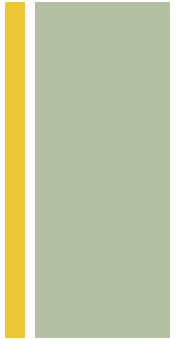
## ■ Third

- Transistors advent; integrated circuit technology; hardware cost dropped
- Demands of data manipulation; Support for data abstraction

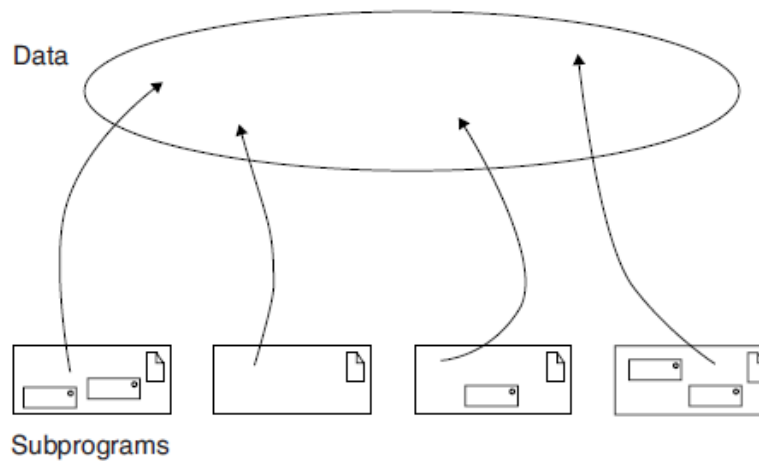




- 70s
  - Thousand of different program languages;
  - Larger programs highlighted inadequacies of earlier languages
  - Few survived; but many concepts introduced adopted by successors
  
- Object-oriented (from 80s, 90s)
  - Object-oriented decomposition of software
  - Main streams: Java, C++, etc.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

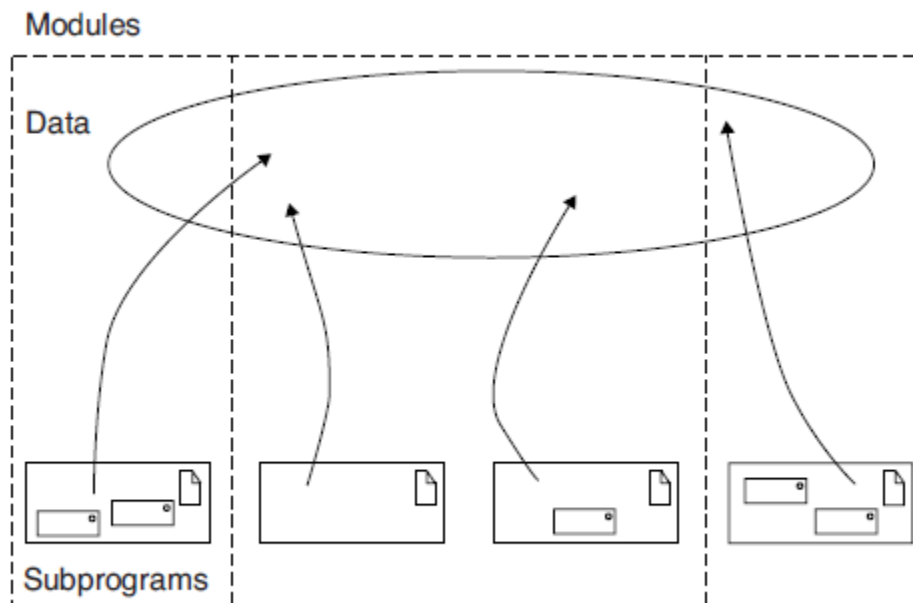
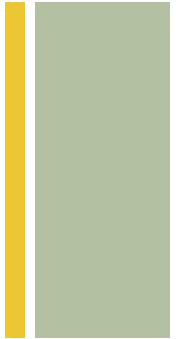


**Figure 2-1** The Topology of First- and Early Second-Generation Programming Languages



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

**Subprograms as  
an abstraction  
mechanism**



Modular structure

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

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

# + 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)

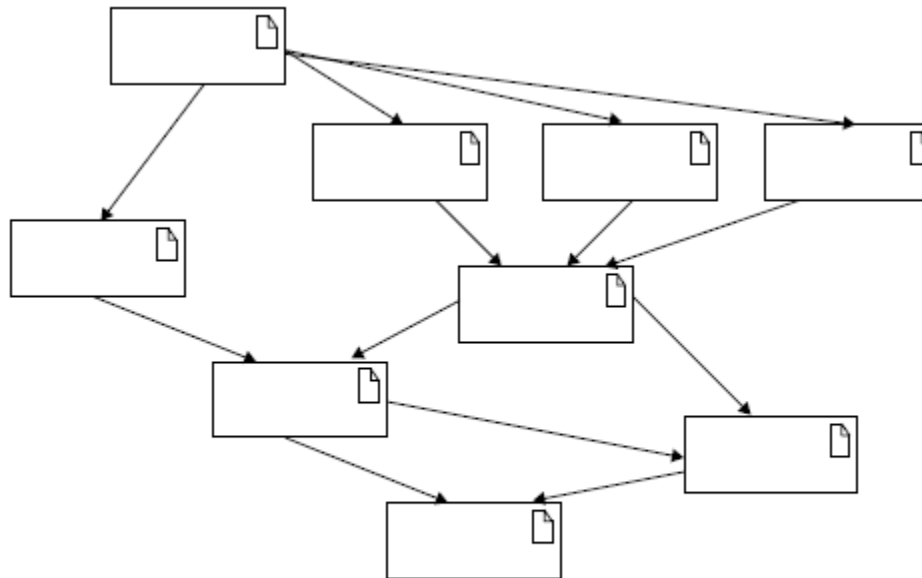
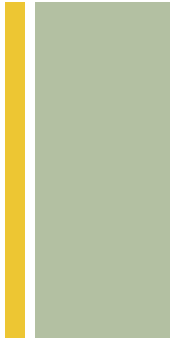


Figure 2–4 The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

# + For object-oriented



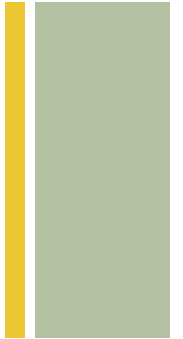
- **Data and operations are united**, that fundamental logical building blocks are no longer algorithms, but classes and objects
- Little or no global data

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)



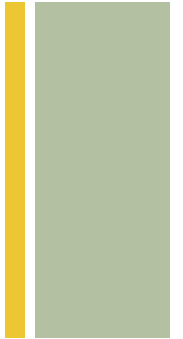
- Object orientation cope with complexity inherent in many different systems
  - Not just to programming languages, user interface design, databases, computer architectures

- 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?



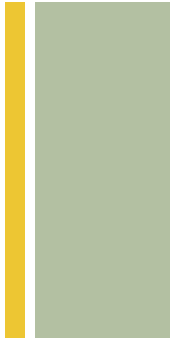
- Cardelli and Wegner say:

[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 concepts

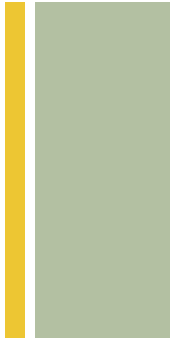


- An object contains encapsulated data and procedures grouped together to represent an entity .
- An object-oriented program is described by the interaction of these objects.
- Information hiding: The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as *private* or *protected* to the owning *class*
- Inheritance: The ability for a *class* to extend or override functionality of another *class*. The so-called *subclass* has a whole section that is derived (inherited) from the *superclass* and then it has its own set of functions and data

# + Object concepts, contd.

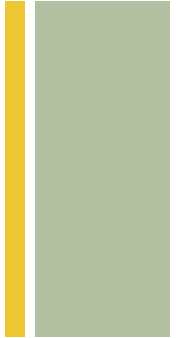
- Interface (object-oriented programming): The ability to defer the implementation of a *method*. The ability to define the *functions* or *methods* signatures without implementing them
- Polymorphism: The ability to replace an *object* with its *subobjects*. The ability of an *object-variable* to contain, not only that *object*, but also all of its *subobjects*

# + 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.



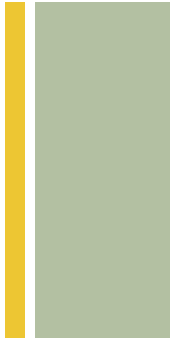
- Object-oriented design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.
- The input for object-oriented design is provided by the output of object-oriented analysis
- Analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process

# + Input for OOD

- Conceptual model: The result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details
- Use case: A description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function
- User interface documentations (if applicable): Document that shows and describes the look and feel of the end product's user interface
- Relational data model (if applicable): A data model is an abstract model that describes how data is represented and used.



# Output of OOD



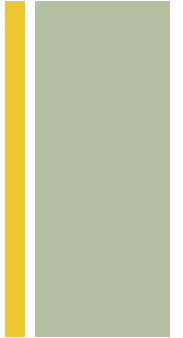
- Class diagram: A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes
- Sequence diagram: Extend the system sequence diagram to add specific objects that handle the system events.
  - A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur

# + 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



- The purpose of any analysis activity in the software life-cycle is to create a model of the system's functional requirements that is independent of implementation constraints
- The main difference between object-oriented analysis and other forms of analysis is that by the object-oriented approach we organize requirements around objects, which integrate both behaviors (processes) and states (data) modeled after real world objects that the system interacts with. In other or traditional analysis methodologies, the two aspects: processes and data are considered separately. For example, data may be modeled by ER diagrams, and behaviors by flow charts or structure charts.

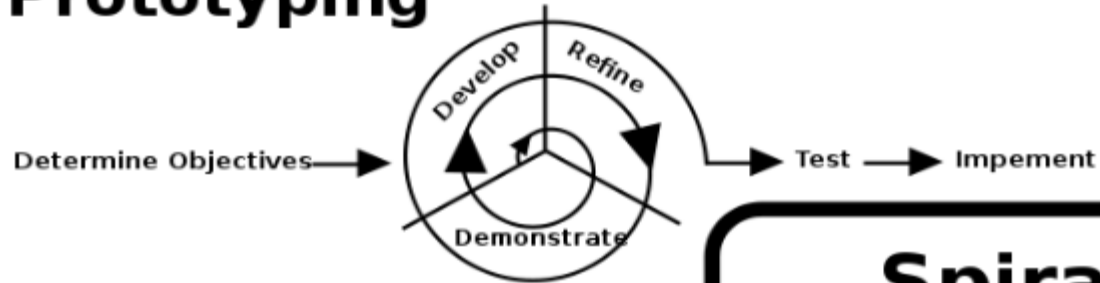


# + Primary tasks in OOA

- Find the objects
- Organize the objects
- Describe how the objects interact
- Define the behavior of the objects
- Define the internals of the objects
- Common models used in OOA are use cases and object models.  
Use cases describe scenarios for standard domain functions that the system must accomplish.
- Object models describe the names, class relations (e.g. Circle is a subclass of Shape), operations, and properties of the main objects

- three basic approaches applied to software development
- + methodology frameworks

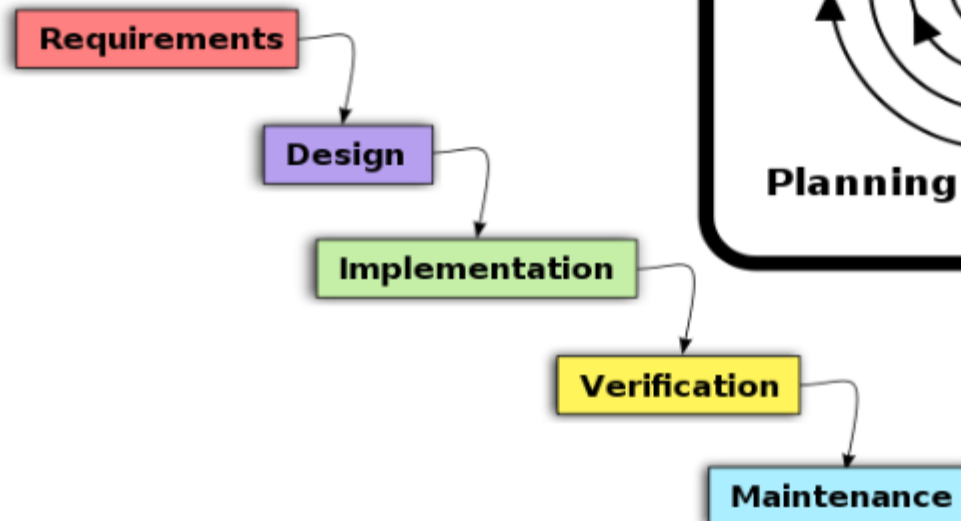
## Prototyping



## Spiral

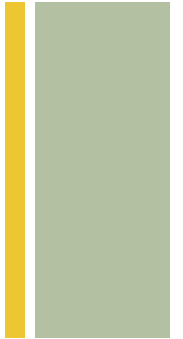


## Waterfall





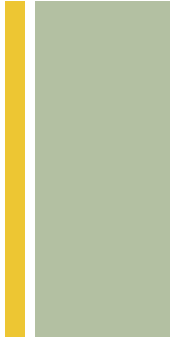
# Object-oriented Programming



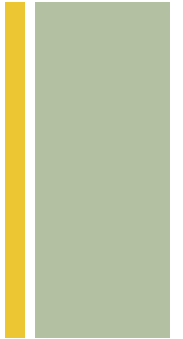
- **Object-oriented programming (OOP)** is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as *attributes*; and code, in the form of procedures, often known as methods
- A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated
- computer programs are designed by making them out of objects that interact with one another
- most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type

# + Criticism

- The OOP paradigm has been criticised for a number of reasons, including not meeting its stated goals of reusability and modularity,<sup>[36][37]</sup> and for overemphasizing one aspect of software design and modeling (data/objects) at the expense of other important aspects (computation/algorithms)
- OOP languages have "extremely poor modularity properties with respect to class extension and modification", and tend to be extremely complex.<sup>[36]</sup> The latter point is reiterated by [Joe Armstrong](#), the principal inventor of [Erlang](#), who is quoted as saying:<sup>[37]</sup>
  - The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.



- In an article Lawrence Krubner claimed that compared to other languages (lisps, functional languages, etc) OOP languages have no unique strengths, and inflict a heavy burden of unneeded complexity
- [Paul Graham](#) has suggested that OOP's popularity within large companies is due to "large (and frequently changing) groups of mediocre programmers." According to Graham, the discipline imposed by OOP prevents any one programmer from "doing too much damage."[\[](#)

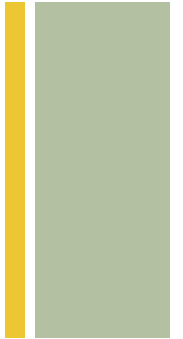


- [Steve Yegge](#) noted that, as opposed to [functional programming](#):  
<sup>[45]</sup>
  - Object Oriented Programming puts the Nouns first and foremost. Why would you go to such lengths to put one part of speech on a pedestal? Why should one kind of concept take precedence over another? It's not as if OOP has suddenly made verbs less important in the way we actually think. It's a strangely skewed perspective
- [Rich Hickey](#), creator of [Clojure](#), described object systems as overly simplistic models of the real world. He emphasized the inability of OOP to model time properly, which is getting increasingly problematic as software systems become more concurrent
- [Eric S. Raymond](#), a [Unix](#) programmer and [open-source software](#) advocate, has been critical of claims that present object-oriented programming as the "One True Solution," and has written that object-oriented programming languages tend to encourage thickly-layered programs that destroy transparency.  
<sup>[46]</sup> Raymond contrasts this to the approach taken with Unix and the [C programming language](#)



## + Elements of the Object Model

# + Programming style



- No single one best for all kinds applications.

- Knowledge base
- Computation-intense operation
- Broadest set of applications

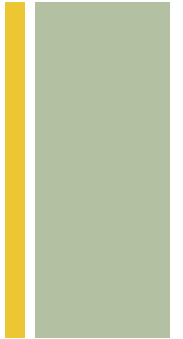
1. Procedure-oriented	Algorithms
2. Object-oriented	Classes and objects
3. Logic-oriented	Goals, often expressed in a predicate calculus
4. Rule-oriented	If-then rules
5. Constraint-oriented	Invariant relationships



# + 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

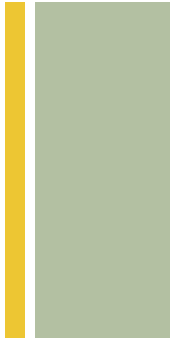


- Define abstraction:

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.

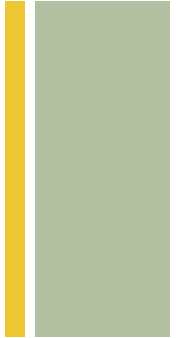
- Focus on outside view of an object, separate object's essential behavior from its implementation
- Decide right set of abstractions for a given domain, is central problem in OOD

# + 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

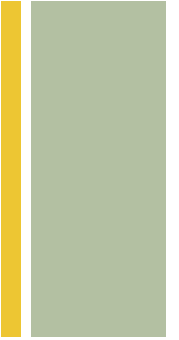
- Farm, maintaining proper greenhouse environment
- A key abstraction is about a sensor
  - A temperature sensor: an object that measures temperature at a location
  - What are responsibilities of a temp sensor? Answers yield different design decisions

<b>Abstraction:</b> Temperature Sensor
<b>Important Characteristics:</b> temperature location
<b>Responsibilities:</b> report current temperature calibrate

**Figure 2–6** Abstraction of a Temperature Sensor

<b>Abstraction:</b> Active Temperature Sensor
<b>Important Characteristics:</b> temperature location setpoint
<b>Responsibilities:</b> report current temperature calibrate establish setpoint

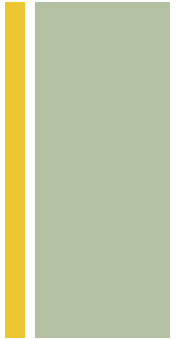
**Figure 2–7** Abstraction of an Active Temperature Sensor



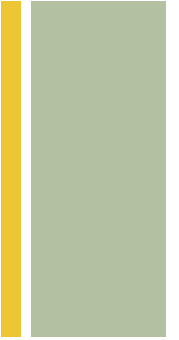
- No objects stands alone; every object collaborates with others to achieve some behavior.
- Design decisions about how they cooperate, define boundaries of each abstraction and the responsibilities and protocol of each object.



# 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



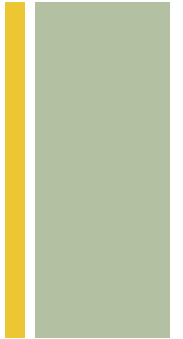
- For abstraction to work, implementations must be encapsulated
  - each class must have two parts: an interface and an implementation
  - Interface – outside view, behavior abstraction
  - Implementation – achieve the behavior
- Define encapsulation
  - “Encapsulation is the process of compartmentalizing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



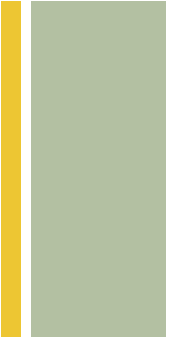
# + Encapsulation contd.

- 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.

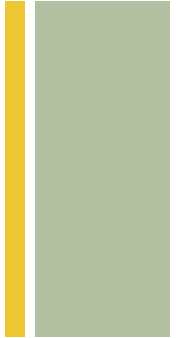
# + Meaning of Modularity



- 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

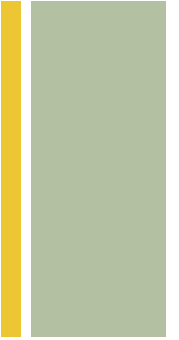


- Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules.
- Deciding on the right set of modules for a given problem is almost as hard a problem as deciding on the right set of abstractions.
- Modules serve as the physical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

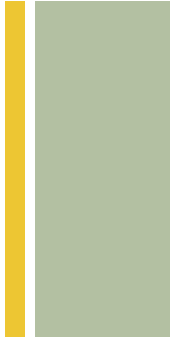


## ■ Guidelines

- 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

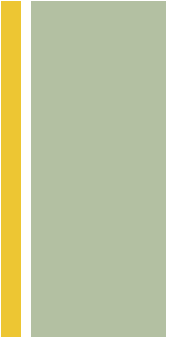


- The developer must therefore balance two competing technical concerns: the desire to encapsulate abstractions and the need to make certain abstractions visible to other modules.
  - strive to build modules that are cohesive (by grouping logically related abstractions) and loosely coupled (by minimizing 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.



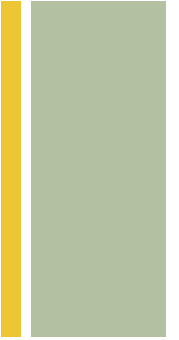
- Additional technical issues may affect modularization decisions
  - Modules as units of a software can be reused across applications; package classes and objects into modules way that makes reuse convenient
  - many compilers generate object code in segments, one for each module; may be practical limits 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

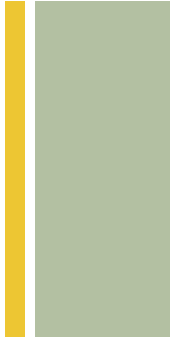


- Single Inheritance
- Multiple Inheritance
- Aggregation

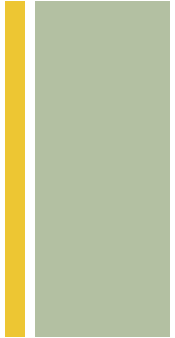




# Single Inheritance



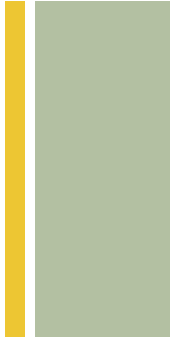
- “is a” hierarchy, relationship
  - A bear “is a” kind of mammal
- Inheritance defines a relationship among classes
  - One class shares structure or behavior defined in on class or more classes (single inheritance or multiple inheritance, respectively)
  - A subclass augments or redefines existing structure and behavior of its super-classes
- Imply a generalization/specialization hierarchy
  - Subclass specializes more general structure or behavior of its superclasses.
  - As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common superclasses



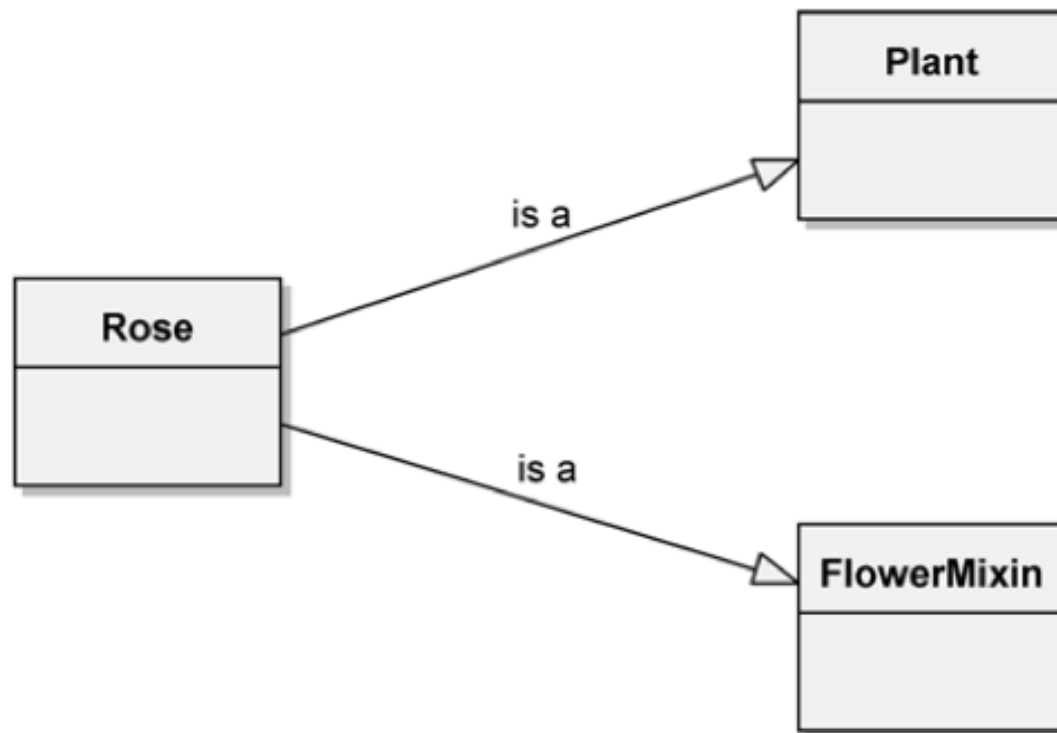
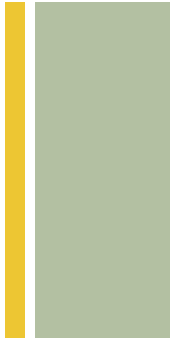
- Trade off support for encapsulation and inheritance
  - Data abstraction attempts to provide an opaque barrier behind which methods and state are hidden; inheritance requires opening this interface to some extent and may allow state as well as methods to be accessed without abstraction
  - C++ and Java offer great flexibility
  - The interface of a class may have three parts:
    - private parts, which declare members that are accessible only to the class itself;
    - protected parts, which declare members that are accessible only to the class and its subclasses;
    - public parts, which are accessible to all clients



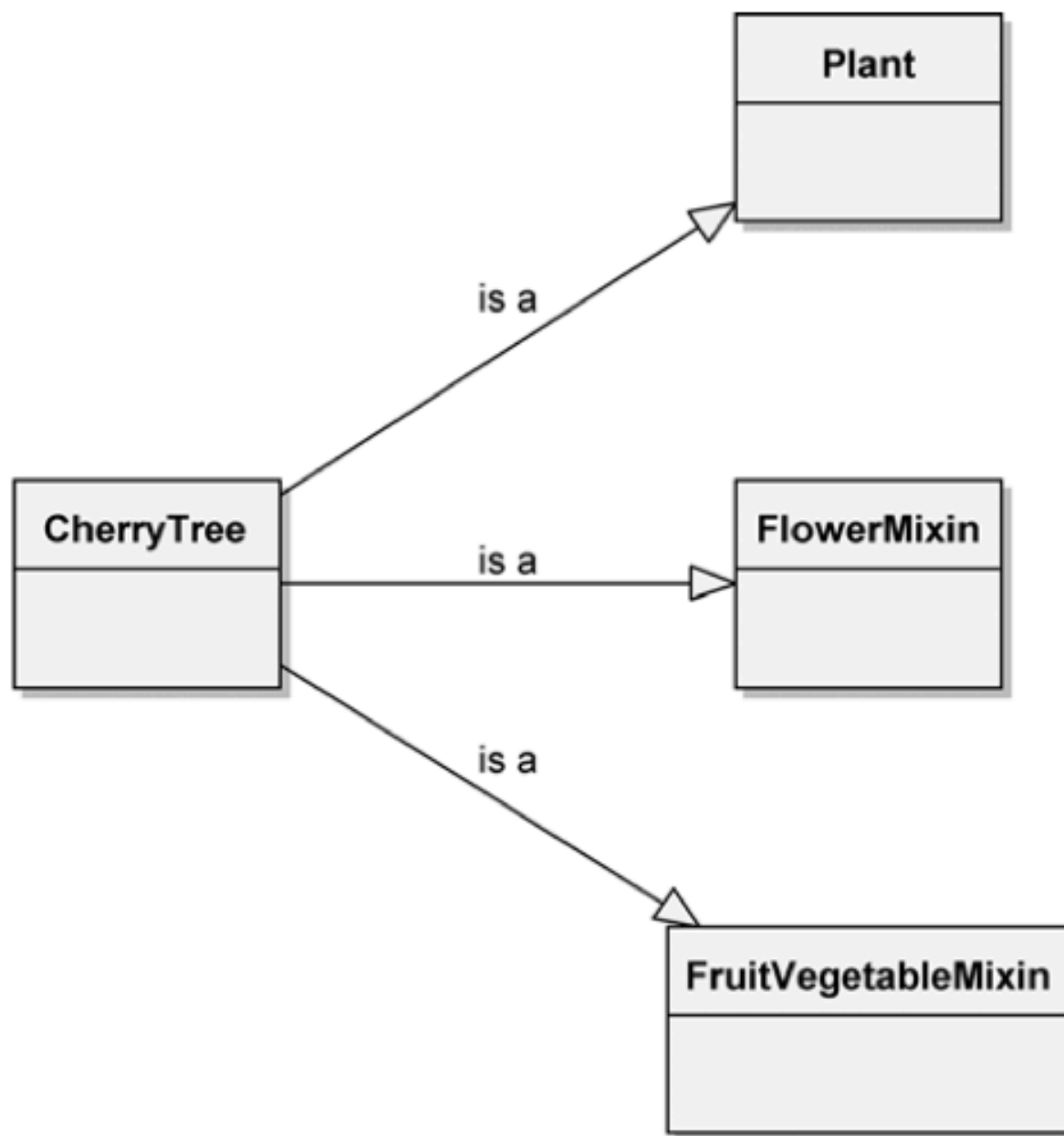
# Examples of Hierarchy: Multiple Inheritance



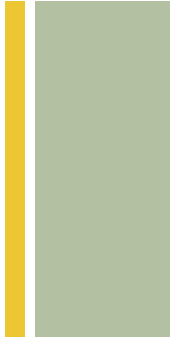
- Inheritance from multiple super-classes
- Flowering plant, fruits and vegetables plant example
  - classes that independently capture the properties unique to flowering plants and to fruits and vegetables ;
  - They have no superclass; they stand alone. These are called *mixin classes* because they are meant to be mixed together with other classes to produce new subclasses.



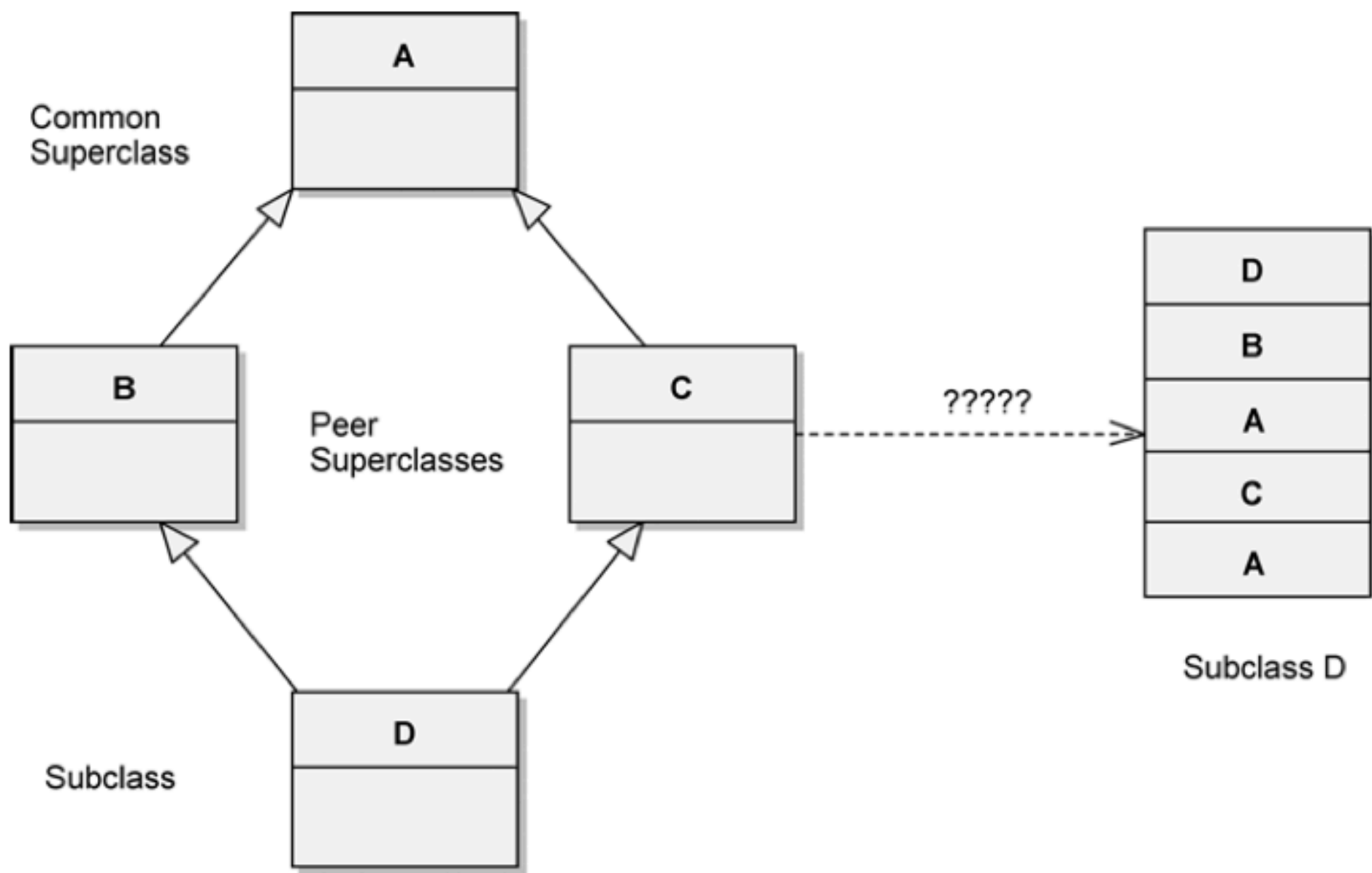
**Figure 2–10** The `Rose` Class, Which Inherits from Multiple Superclasses



**Figure 2-12** The `CherryTree` Class, Which Inherits from Multiple Superclasses



- Languages must address two issues: clashes among names from different superclasses and repeated inheritance.
  - Repeated inheritance occurs when two or more peer superclasses share a common superclass
  - question arises, does the leaf class (i.e., subclass) have one copy or multiple copies of the structure of the shared superclass?
  - Some languages prohibit repeated inheritance, some unilaterally choose one approach, and others, such as C++, permit the programmer to decide
  - In C++, virtual base classes are used to denote a sharing of repeated structures, whereas nonvirtual base classes result in duplicate copies appearing in the subclass



**Figure 2-13** The Repeated Inheritance Problem

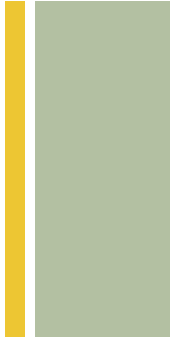
# + Examples of Hierarchy: Aggregation

- “is a” hierarchies denote generalization/specialization relationships,
- “part of” hierarchies describe aggregation relationships.
  - E.g. , a garden consists of a collection of plants with a growing plan
  - i.e. , plants are “part of ” the garden, growing plan is “part of ” garden
- combination of inheritance with aggregation is powerful: Aggregation permits the physical grouping of logically related structures, and inheritance allows these common groups to be easily reused among different abstractions.



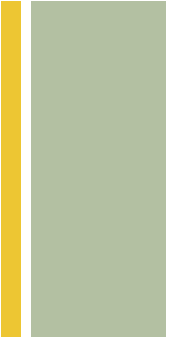


# Aggregation raises issue of ownership

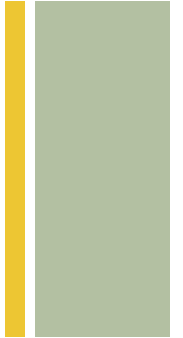


- replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants
- lifetime of a garden and its plants are independent
- In contrast, GrowingPlan object is intrinsically associated with a Garden object and does not exist independently

# + Meaning of Typing

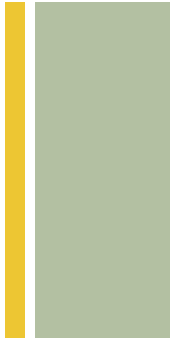


- concept of a type derives primarily from the theories of abstract data types
- Deutsch suggests, “A type is a precise characterization of structural or behavioral properties which a collection of entities all share”
- Though type and class similar, type places a different emphasis on meaning of abstraction
  - Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways



- Strongly typed, weakly typed, untyped
- Strong typing enforces certain design decisions; but introduces semantic dependencies such that even small changes in interface of a base class require recompilation of all subclasses
- Two general solutions
  - First, we could use a type-safe container class that manipulates only objects of a specific class. This approach addresses the first problem, wherein objects of different types are incorrectly mingled
  - Second, we could use some form of runtime type identification; this addresses the second problem of knowing what kind of object you happen to be examining at the moment.

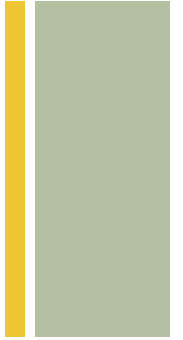
# + Benefits using strongly typed languages



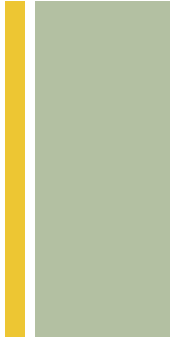
- Without type checking, a program in most languages can ‘crash’ in mysterious ways at runtime.
- In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable.
- Type declarations help to document programs.
- Most compilers can generate more efficient object code if types are declared. [72]



# Static and dynamic typing



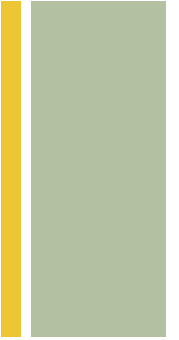
- Strong and weak typing refers to *type consistency*, whereas static and dynamic typing refers to the *time when names are bound to types*
- Static typing (also known as *static binding* or *early binding*) means that the types of all variables and expressions are fixed at the time of compilation;
- dynamic typing (also known as *late binding*) means that the types of all variables and expressions are not known until runtime
- A language may be both strongly and statically typed (Ada), strongly typed yet supportive of dynamic typing (C++, Java), or untyped yet supportive of dynamic typing (Smalltalk).



- *Polymorphism* is a condition that exists when the features of dynamic typing and inheritance interact.
- Polymorphism represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass
- opposite of polymorphism is *monomorphism*, which is found in all languages that are both strongly and statically typed
- Polymorphism is the most powerful feature of object-oriented programming languages next to their support for abstraction
  - is what distinguishes object-oriented programming from more traditional programming with abstract data types.

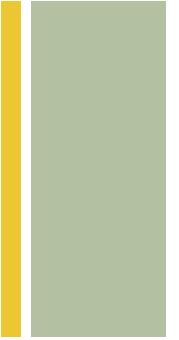
# + Meaning of Concurrency

- an automated system may have to handle many different events simultaneously. Other problems may involve so much computation that they exceed the capacity of any single processor
- natural to consider using a distributed set of computers for the target implementation or to use multitasking
- System involving concurrency may have many threads of control, some are transitory, others last entire lifetime of the system's execution
- Systems across multiple CPUs allow for truly concurrent threads of control, whereas running on a single CPU only achieve illusion of concurrent threads of control, by means of time-slicing algorithm



- designing one that encompasses multiple threads of control is much harder because one must worry about such issues as deadlock, livelock, starvation, mutual exclusion, and race conditions.
- Black et al. therefore suggest that “an object model is appropriate for a distributed system because it implicitly defines (1) the units of distribution and movement and (2) the entities that communicate” [77]
- Object-oriented programming focuses on data abstraction, encapsulation, and inheritance, concurrency focuses on process abstraction and synchronization [78]



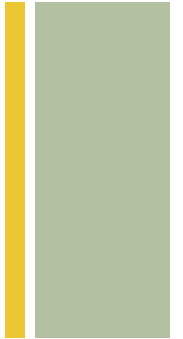


- Active Objects

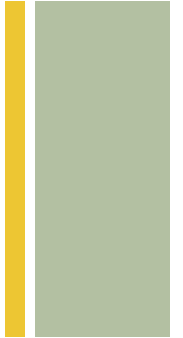
- object is a concept that unifies these two different viewpoints:  
Each object (drawn from an abstraction of the real world) may represent a separate thread of control (a process abstraction).
- In a system based on an object-oriented design, we can conceptualize the world as consisting of a set of cooperative objects, some of which are active and thus serve as centers of independent activity
- Define concurrency as follows:
  - Concurrency is the property that distinguishes an active object from one that is not active



# Approaches to concurrency in OO-design



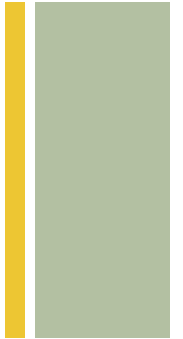
- First, concurrency is an intrinsic feature of certain programming languages, which provide mechanisms for concurrency and synchronization. So can use it to create an active object.
- Second, we may use a class library that implements some form of lightweight processes. Naturally, the implementation of this kind is highly platform-dependent, although the interface to the library may be relatively portable
- Third, we may use interrupts to give us the illusion of concurrency.
  - knowledge of certain low-level hardware details
  - might have a hardware timer that periodically interrupts the application



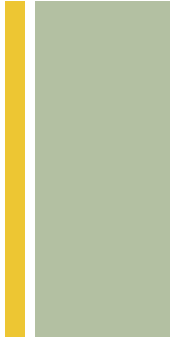
- you must consider how active objects **synchronize** their activities with one another as well as with objects that are purely sequential
  - E.g. , if two active objects try to send messages to a third object, we must be certain to use some means of mutual exclusion, so that the state of the object being acted on is not corrupted
  - This is the point where the ideas of abstraction, encapsulation, and concurrency interact
- In the presence of concurrency, it is not enough simply to define the methods of an object; we must also make certain that the semantics of these methods are preserved in the presence of multiple threads of control.



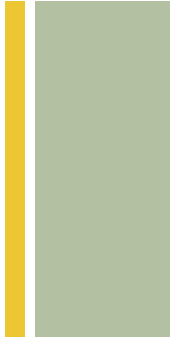
# Meaning of Persistence



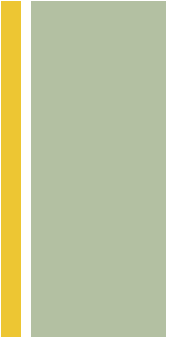
- An object in software takes up some amount of space and exists for a particular amount of time
- Spectrum of object persistence encompasses the following:
  - Transient results in expression evaluation
  - Local variables in procedure activations
  - Own variables [as in ALGOL 60], global variables, and heap items whose extent is different from their scope
  - Data that exists between executions of a program
  - Data that exists between various versions of a program
  - Data that outlives the program [79]



- “Data that outlives the program” is the case of Web applications where the application may not be connected to the data it is using through the entire transaction execution.
- introducing the concept of persistence to the object model gives rise to object-oriented databases
  - offer to the programmer the abstraction of an object-oriented interface, through which database queries and other operations are completed in terms of objects whose lifetimes transcend the lifetime of an individual program.
  - it allows us to apply the same design methods to the database and nondatabase segments of an application



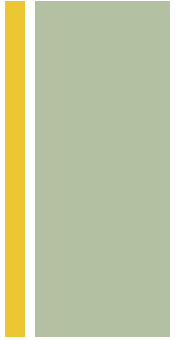
- Some object-oriented programming languages provide direct support for persistence.
  - Java provides Enterprise Java Beans (EJBs) and Java Data Objects.
- However, streaming objects to flat files is a naive solution to persistence that does not scale well.
- typical approach to persistence is to provide an object-oriented skin over a relational database
- Customized object-relational mappings can be created by the individual developer, but challenging to do well
  - Frameworks available to ease this task, e.g., Hibernate



- for systems that execute on a distributed set of processors, we must sometimes be concerned with persistence across space
- Define persistence as follows:
  - Persistence is the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created).



# Benefits of Object Model



- First, the use of the object model helps us to exploit the expressive power of object-based and object-oriented programming languages.
- Second, the use of the object model encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frameworks
- Third, the use of the object model produces systems that are built on stable intermediate forms, which are more resilient to change.
- Fourth, object model's guidance in designing an intelligent separation of concerns also reduces development risk and increases our confidence in the correctness of our design.