今天的内容

- Elements of object model (concepts)
  - Abstraction
  - Encapsulation
  - Modularization
  - Hierarchy
  - Typing
  - Concurrency
  - Persistence
- 复习
- 继续探索

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 between encapsulation and inheritance

- 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.

多继承举例

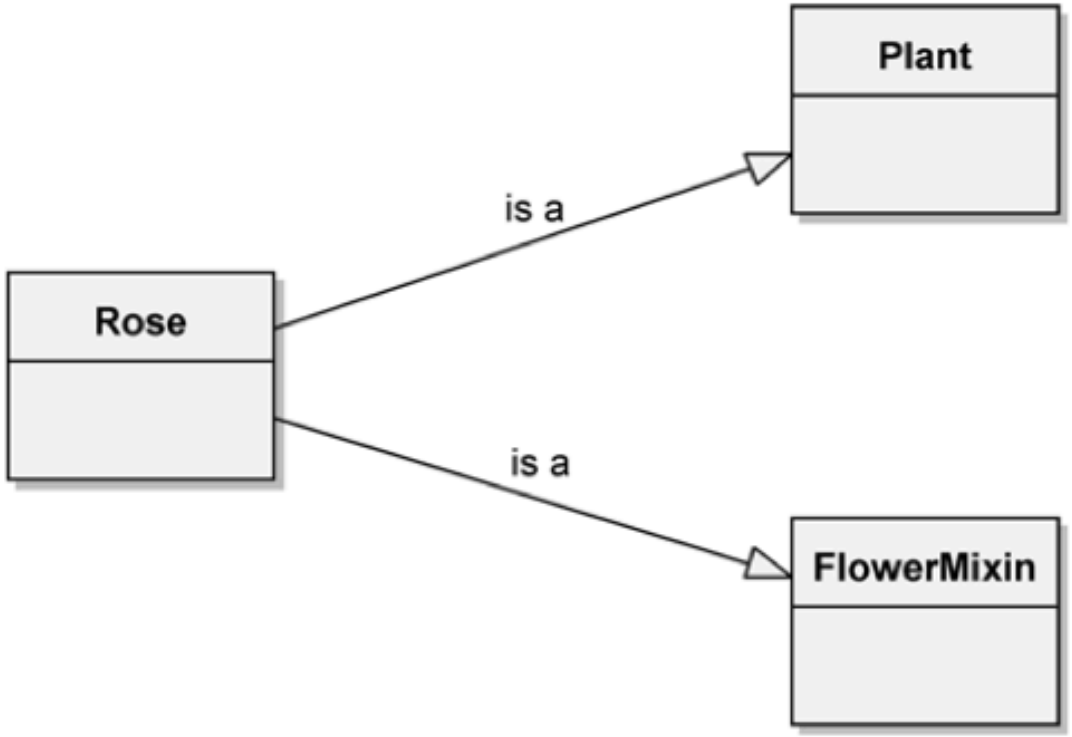
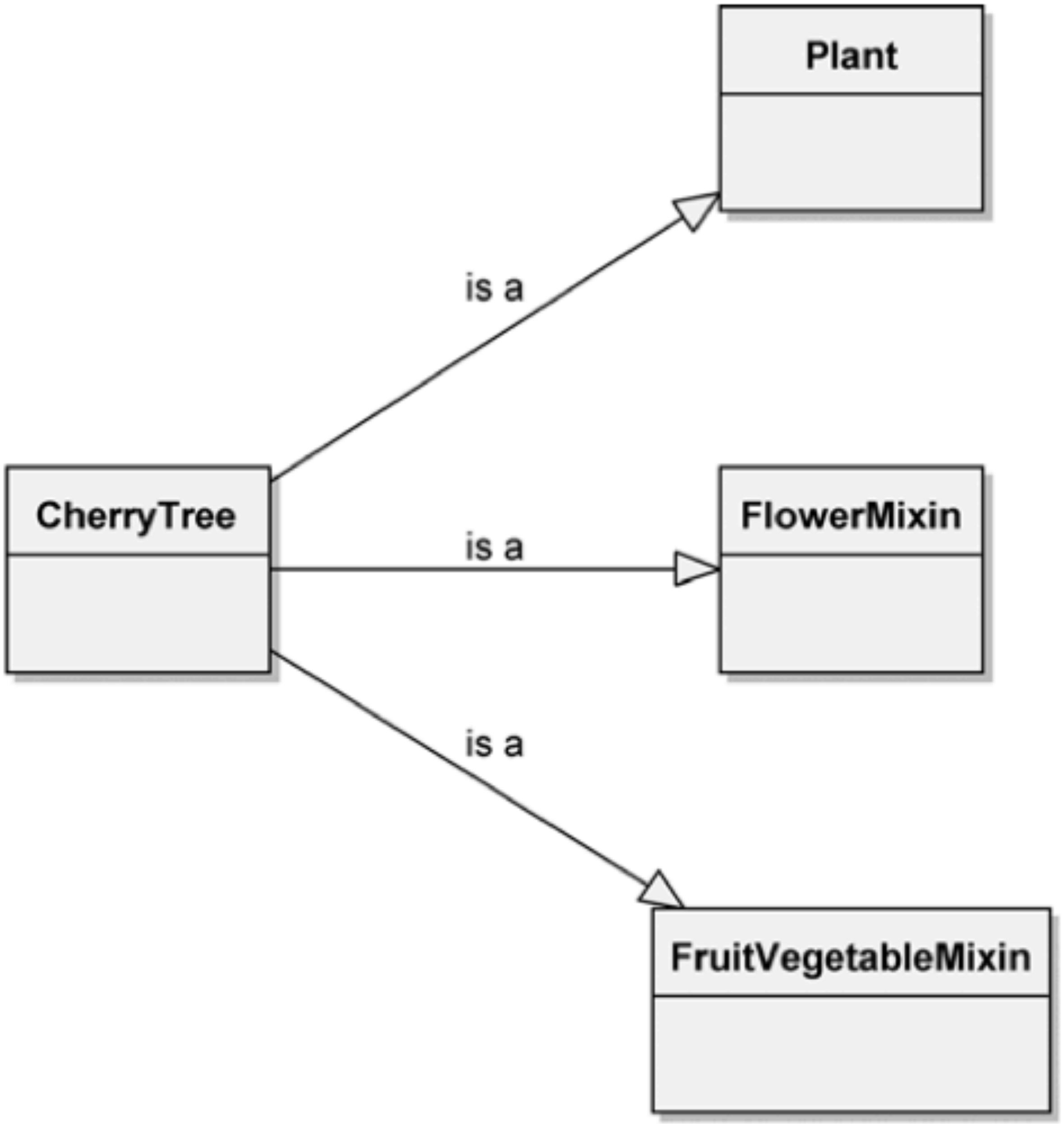


Figure 2–10 The `Rose` Class, Which Inherits from Multiple Superclasses

+



gure 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

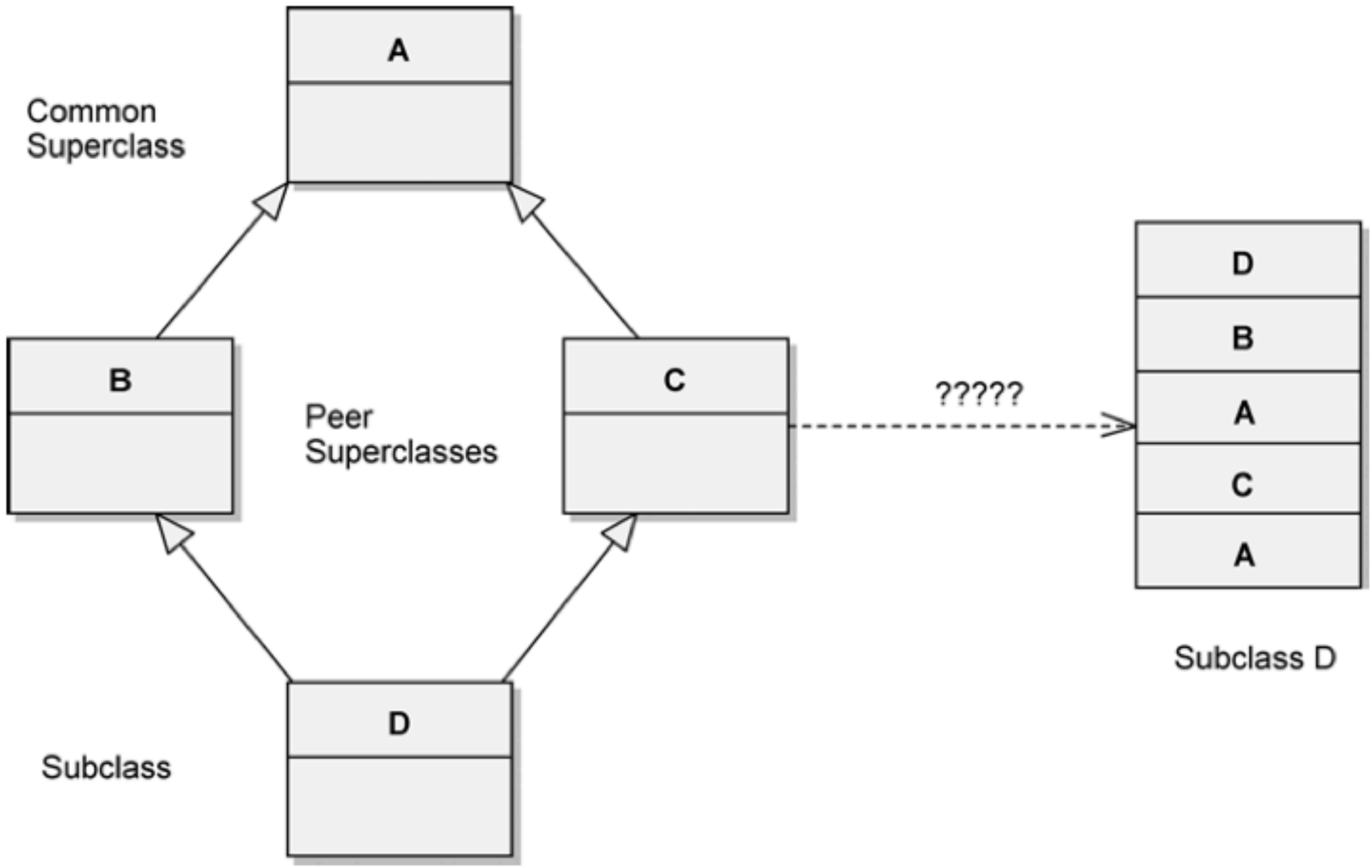


Figure 2–13 The Repeated Inheritance Problem

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.

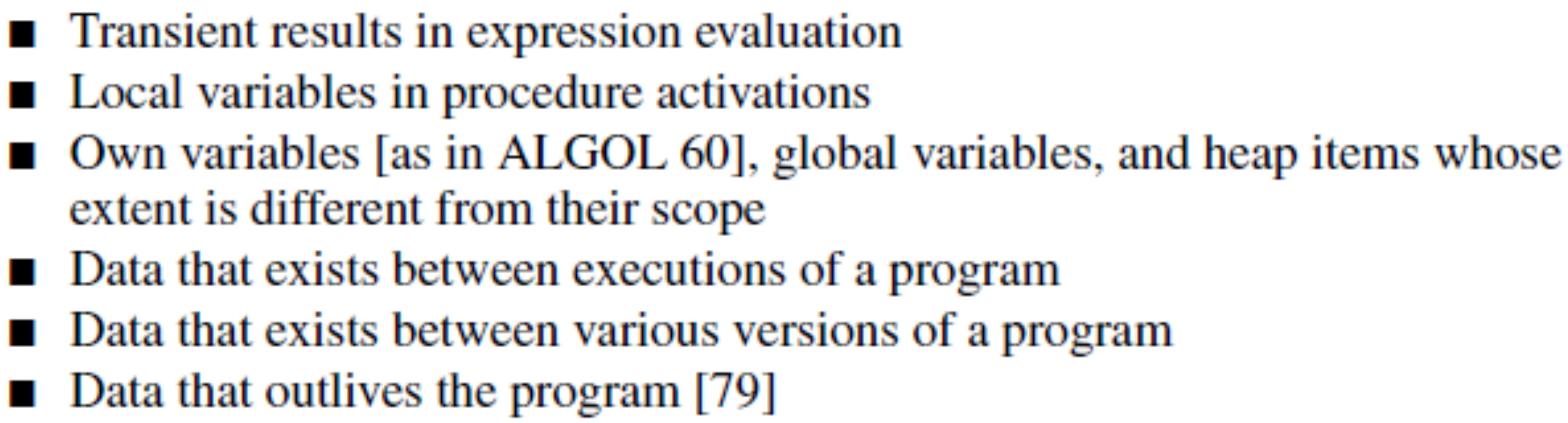
- Second, we may use a class library that implements some form of lightweight processes. Naturally, the implementation of this kind is highly platform-dependent, although the interface to the library may be relatively portable

- 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.

期末考试的范围和题型

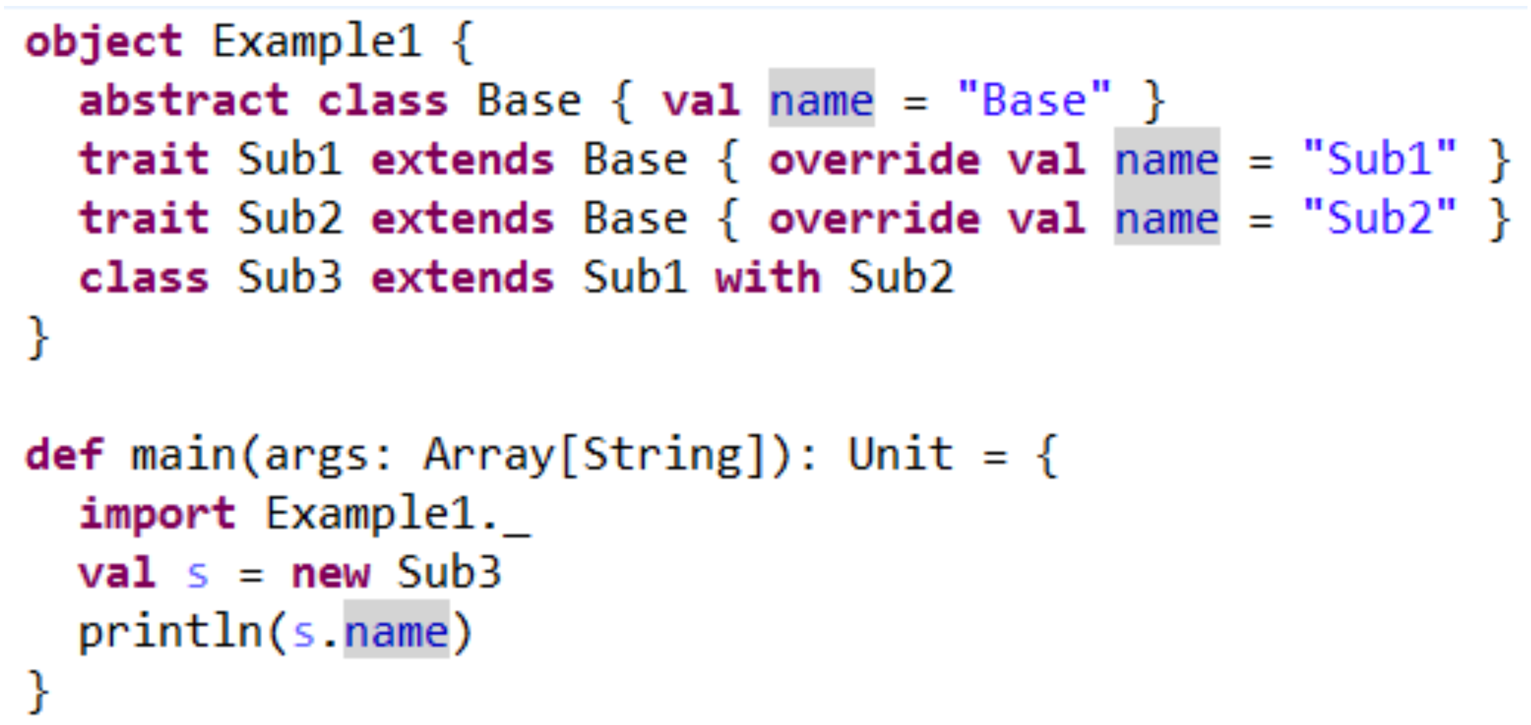
- 题型
  - 1 选择题 （3道）
  - 2 判断对错 （9）
  - 3 分析题 （2）
  - 4 简答题 （8）
- 范围
  - 主要覆盖前5次课，后6次课的内容
  - Scala里的面向对象编程
  - 对象模型基础和元素

重要的概念

- 继承(inheritance)；多继承，重复继承
- 多态(Polymorphism)
- 覆盖，重载
- 抽象类
- 对象模型的主要元素
  - 抽象，封装，组件化，层次体系
- 面向对象的要求
- 面向对象分析 -> 面向对象设计 -> 面向对象编程
- 类型(type)；强类型；静态类型，动态类型绑定

Review: 重复继承

- 重复继承例子



```
object Example1 {
  abstract class Base { val name = "Base" }
  trait Sub1 extends Base { override val name = "Sub1" }
  trait Sub2 extends Base { override val name = "Sub2" }
  class Sub3 extends Sub1 with Sub2
}

def main(args: Array[String]): Unit = {
  import Example1._
  val s = new Sub3
  println(s.name)
}
```

+

Quora: crowd resourcing

- www.quora.com
- Questions and answers in almost all the fields

What are some best practices in Scala object-oriented programming?

- http://www.quora.com/What-are-some-best-practices-in-Scala-object-oriented-programming
- Trait for mixin inheritance
- Abstract type and abstract val
- Object-based module system
- Path-dependent types

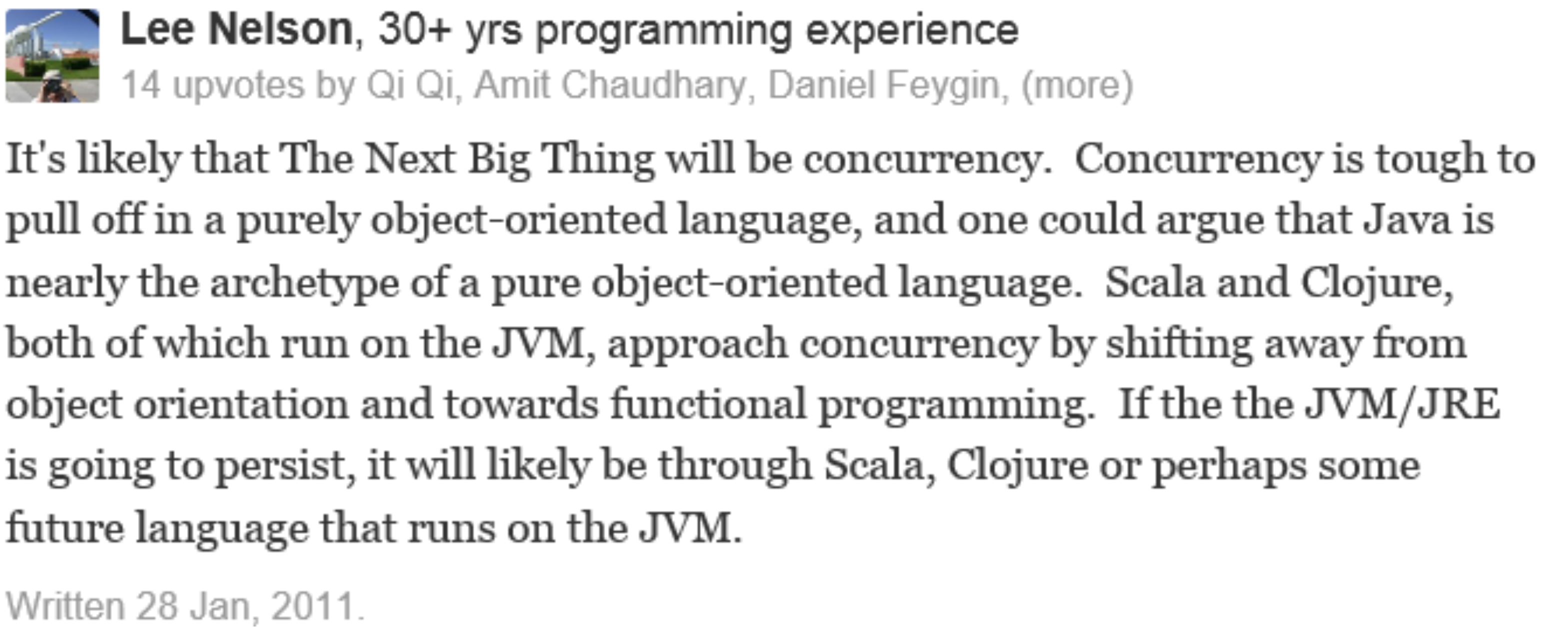
What are the advantages of Functional Programming over Object-Oriented Programming?

- http://www.quora.com/What-are-the-advantages-of-Functional-Programming-over-Object-Oriented-Programming

+

What's the future of Java and the Java community?

- http://www.quora.com/Whats-the-future-of-Java-and-the-Java-community



Lee Nelson, 30+ yrs programming experience
14 upvotes by Qi Qi, Amit Chaudhary, Daniel Feygin, (more)

It's likely that The Next Big Thing will be concurrency. Concurrency is tough to pull off in a purely object-oriented language, and one could argue that Java is nearly the archetype of a pure object-oriented language. Scala and Clojure, both of which run on the JVM, approach concurrency by shifting away from object orientation and towards functional programming. If the the JVM/JRE is going to persist, it will likely be through Scala, Clojure or perhaps some future language that runs on the JVM.

Written 28 Jan, 2011.