Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 2:
Review of Object Orientation

2.1 What is Object Orientation?

Procedural paradigm:

- Software is organized around the notion of *procedures*
- *Procedural abstraction*
  - —Works as long as the data is simple
- *Adding data abstractions*
  - —Groups together the pieces of data that describe some entity
  - —Helps reduce the system's complexity.
    - Such as *Records* and *structures*

Object oriented paradigm:

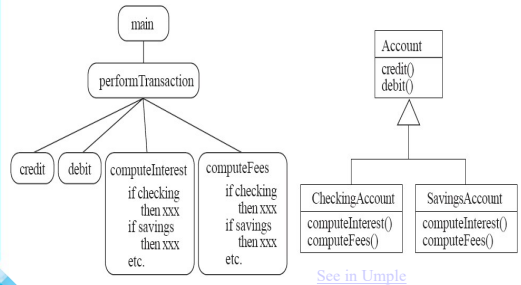
- Organizing procedural abstractions in the context of data abstractions

Object Oriented paradigm

An approach to the solution of problems in which all computations are performed in the context of objects.

- The objects are instances of classes, which:
  - —are data abstractions
  - —contain procedural abstractions that operate on the objects
- A running program can be seen as a collection of objects collaborating to perform a given task

A View of the Two paradigms



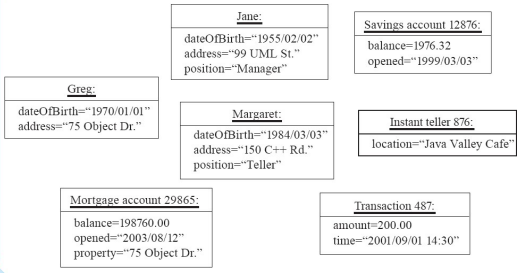
See in Umple

2.2 Classes and Objects

Object

- A chunk of structured data in a running software system
- Has *properties*
  - —Represent its state
- Has *behaviour*
  - —How it acts and reacts
  - —May simulate the behaviour of an object in the real world

Objects



Classes

A class:

- A unit of abstraction in an object oriented (OO) program
- Represents similar objects
  - —Its *instances*
- A kind of software module
  - —Describes its instances' structure (properties)
  - —Contains *methods* to implement their behaviour

Is Something a Class or an Instance?

- Something should be a *class* if it could have instances
- Something should be an *instance* if it is clearly a *single* member of the set defined by a class

Film

- Class; instances are individual films.

***Reel of Film*:**

- Class; instances are physical reels

Film reel with serial number SW19876

- Instance of `ReelOfFilm`

Science Fiction

- Instance of the class `Genre`.

Science Fiction Film

- Class; instances include 'Star Wars'

***Showing of 'Star Wars' in the Phoenix Cinema at 7 p.m.*:**

- Instance of `ShowingOfFilm`

Naming classes

- Use *capital* letters
  - —E.g. `BankAccount` not `bankAccount`
- Use *singular* nouns
- Use the right level of generality
  - —E.g. `Municipality`, not `City`
- Make sure the name has only *one* meaning
  - —E.g. 'bus' has several meanings

2.3 Instance Variables

Variables defined inside a class corresponding to data present in each instance

- Also called *fields* or *member variables*
- Attributes
  - —Simple data
  - —E.g. `name`, `dateOfBirth`
- Associations
  - —Relationships to other important classes
  - —E.g. `supervisor`, `coursesTaken`
  - —More on these in Chapter 5

Variables vs. Objects

A variable

- *Refers* to an object
- May refer to different objects at different points in time

An object can be referred to by several different variables at the same time

***Type* of a variable**

- Determines what classes of objects it may contain

Class variables

A *class variable's* value is *shared* by all instances of a class.

- Also called a *static* variable
- If one instance sets the value of a class variable, then all the other instances see the same changed value.
- Class variables are useful for:
  - —Default or 'constant' values (e.g. PI)
  - —Lookup tables and similar structures

Caution: *do not over-use class variables*

2.4 Methods, Operations and Polymorphism

Operation

- A higher-level procedural abstraction that specifies a type of behaviour
- Independent of any code which implements that behaviour
  - —E.g. calculating area (in general)

Methods, Operations and Polymorphism

Method

- A procedural abstraction used to implement the behaviour of a class
- Several different classes can have methods with the same name
  - —They implement the same abstract operation in ways suitable to each class
  - —E.g. calculating area in a rectangle is done differently from in a circle

Polymorphism

A property of object oriented software by which an *abstract operation may be performed in different ways* in different classes.

- Requires that there be *multiple methods of the same name*
- The choice of which one to execute depends on the object that is in a variable
- Reduces the need for programmers to code many `if-else` or `switch` statements

2.5 Organizing Classes into Inheritance Hierarchies

Superclasses

- Contain features common to a set of subclasses

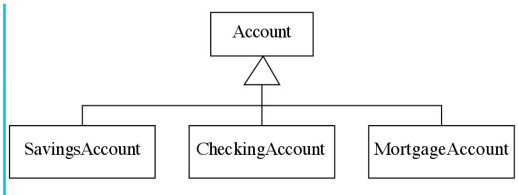
Inheritance hierarchies

- Show the relationships among superclasses and subclasses
- A triangle shows a *generalization* △

Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

An Example Inheritance Hierarchy



See in Umple

Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

The Isa Rule

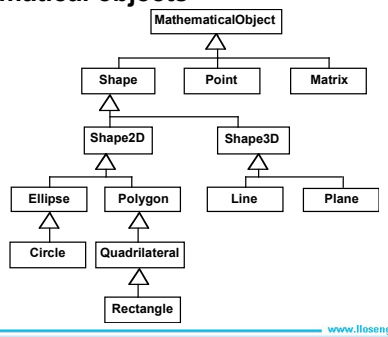
Always check generalizations to ensure they obey the isa rule

- "A checking account ***is an*** account"
- "A village ***is a*** municipality"

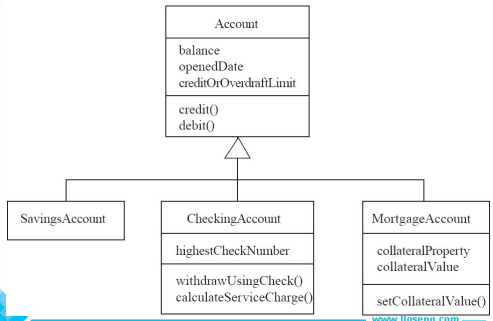
Should 'Province' be a subclass of 'Country'?

- No, it violates the isa rule
  - —"A province ***is a*** country" is invalid!

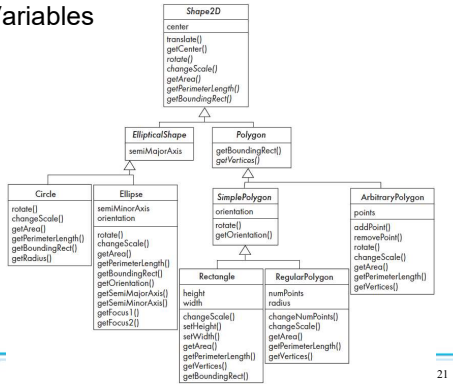
A possible inheritance hierarchy of mathematical objects



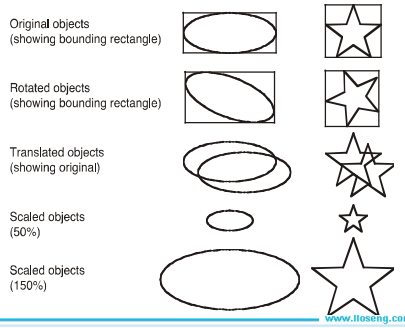
Make Sure all Inherited Features Make Sense in Subclasses



2.6 Inheritance, Polymorphism and Variables



Some Operations in the Shape Example



Abstract Classes and Methods

An operation should be declared to exist at the highest class in the hierarchy where it makes sense

- The *operation* may be *abstract* (lacking implementation) at that level
- If so, the *class* also must be *abstract*
  - —No instances can be created
  - —The opposite of an abstract class is a *concrete* class
- If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
  - —Leaf classes must have or inherit concrete methods for all operations
  - —Leaf classes must be concrete

Overriding

A method would be inherited, but a subclass contains a new version instead

- For restriction
  - —E.g. `scale(x,y)` would not work in `Circle`
- For extension
  - —E.g. `SavingsAccount` might charge an extra fee following every debit
- For optimization
  - —E.g. The `getPerimeterLength` method in `Circle` is much simpler than the one in `Ellipse`

How a decision is made about which method to run

1. **If there is a concrete method for the operation in the current class, run that method.**
2. **Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.**
3. **Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.**
4. **If no method is found, then there is an error**
   - In Java and C++ the program would not have compiled

Dynamic binding

Occurs when decision about which method to run can only be made at *run time*

- Needed when:
  - —A variable is declared to have a superclass as its type, and
  - —There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses

2.7 Concepts that Define Object Orientation

The following are necessary for a system or language to be OO

- Identity
  - —Each object is *distinct* from each other object, and *can be referred to*
  - —Two objects are distinct *even if they have the same data*
- Classes
  - —The code is organized using classes, each of which describes a set of objects
- Inheritance
  - —The mechanism where features in a hierarchy inherit from superclasses to subclasses
- Polymorphism
  - —The mechanism by which several methods can have the same name and implement the same abstract operation.

Other Key Concepts

Abstraction

- Object -> something in the world
- Class -> objects
- Superclass -> subclasses
- Operation -> methods
- Attributes and associations -> instance variables

Modularity

- Code can be constructed entirely of classes

Encapsulation

- Details can be hidden in classes
- This gives rise to *information hiding*:
  - —Programmers do not need to know all the details of a class

The Basics of Java

History

- The first object oriented programming language was Simula-67
  - —designed to allow programmers to write simulation programs
- In the early 1980's, Smalltalk was developed at Xerox PARC
  - —New syntax, large open-source library of reusable code, bytecode, platform independence, garbage collection.
- late 1980's, C++ was developed by B. Stroustrup,
  - —Recognized the advantages of OO but also recognized that there were tremendous numbers of C programmers
- In 1991, engineers at Sun Microsystems started a project to design a language that could be used in consumer 'smart devices': Oak
  - —When the Internet gained popularity, Sun saw an opportunity to exploit the technology.
  - —The new language, renamed Java, was formally presented in 1995 at the SunWorld '95 conference.

Java documentation

Looking up classes and methods is an essential skill

- Looking up unknown classes and methods will get you a long way towards understanding code

Java documentation can be automatically generated by a program called Javadoc

- Documentation is generated from the code and its comments
- You should format your comments as shown in some of the book's examples
  - —These may include embeded html

Overview of Java

The next few slides will remind you of several key Java features

- Not in the book
- See the book's web site for
  - —A more detailed overview of Java
  - —Pointers to tutorials, books etc.

Characters and Strings

`Character` is a class representing Unicode characters

- More than a byte each
- Represent any world language

`char` is a primitive data type containing a Unicode character

`String` is a class containing collections of characters

- + is the operator used to concatenate strings

Arrays and Collections

Arrays are of fixed size and lack methods to manipulate them

`ArrayList` is the most widely used class to hold a *collection* of other objects

- More powerful than arrays, but less efficient

`Iterators` are used to access members of `Vectors`

- Enumerations were formally used, but were more complex

```
a = new ArrayList();
Iterator i = a.iterator();
while(i.hasNext())
{
  aMethod(i.next());
}
```


Casting

Java is very strict about types

- If variable v is declared to have type X, you can only invoke operations on v that are defined in X or its superclasses
  - —Even though an instance of a *subclass* of X may be actually stored in the variable
- If you *know* an instance of a subclass is stored, then you can *cast* the variable to the subclass
  - —E.g. if I know a `Vector` contains instances of `String`, I can get the next element of its `Iterator` using:

```
(String)i.next();
```

  - —To avoid casting you could also have used templates::

```
a = ArrayList<String>; i=a.iterator();
i.next()
```

using

Exceptions

Anything that can go wrong should result in the raising of an Exception

- **Exception** is a class with many subclasses for specific things that can go wrong

Use a try - catch block to trap an exception

```
try
{
  // some code
}
catch (ArithmeticException e)
{
  // code to handle division by zero
}
```


Interfaces

Like abstract classes, but cannot have executable statements

- Define a set of operations that make sense in several classes
- Abstract Data Types

A class can implement any number of interfaces

- It must have concrete methods for the operations

You can declare the type of a variable to be an interface

- This is just like declaring the type to be an abstract class

Important interfaces in Java's library include

- Runnable, Collection, Iterator, Comparable, Cloneable

Packages and importing

A package combines related classes into subsystems
- All the classes in a particular directory

Classes in different packages can have the same name
- Although not recommended

***Importing* a package is done as follows:**
import finance.banking.accounts.*;

Access control

Applies to methods and variables
- `public`
  —Any class can access
- `protected`
  —Only code in the package, or subclasses can access
- (blank)
  —Only code in the package can access
- `private`
  —Only code written in the class can access
  —Inheritance still occurs!

Threads and concurrency

Thread:
- Sequence of executing statements that can be running concurrently with other threads

To create a thread in Java:
- 1. Create a class implementing `Runnable` or extending `Thread`
- 2. Implement the `run` method as a loop that does something for a period of time
- 3. Create an instance of this class
- 4. Invoke the `start` operation, which calls `run`

Programming Style Guidelines

Remember that programs are for people to read
- Always choose the simpler alternative
- Reject clever code that is hard to understand
- Shorter code is not necessarily better

Choose good names
- Make them highly descriptive
- Do not worry about using long names

Programming style …

Comment extensively
- Comment whatever is non-obvious
- Do not comment the obvious
- Comments should be 25-50% of the code

Organize class elements consistently
- Variables, constructors, public methods then private methods

Be consistent regarding layout of code

Programming style …

Avoid duplication of code
- Do not 'clone' if possible
  —Create a new method and call it
  —Cloning results in two copies that may both have bugs
  - When one copy of the bug is fixed, the other may be forgotten

Programming style ...

Adhere to good object oriented principles

- E.g. the 'isa rule'

Prefer `private` as opposed to `public`

Do not mix user interface code with non-user interface code

- Interact with the user in separate classes
  - —This makes non-UI classes more reusable

2.10 Difficulties and Risks in Object-Oriented Programming

Language evolution and deprecated features:

- Java is evolving, so some features are 'deprecated' at every release
- But the same thing is true of most other languages

Efficiency can be a concern in some object oriented systems

- Java can be less efficient than other languages
  - —VM-based
  - —Dynamic binding