Previous Lecture

- Abstraction
  
  Abstraction is the elimination of the irrelevant and the amplification of the essential – Robert C Martin

- Example: Car
  
  To the driver, the car is a steering wheel, gas pedal, brake, and either an automatic or manual transmission.
  
  Everything else is largely irrelevant to the driver of a car

- Abstraction in Programming Languages
  
  Process Abstraction => Subprograms
  
  Data Abstraction => Abstract Data Types

From ADTs to OOP

- Encapsulation
  
  Grouping of data and operations of logically related types
  
  E.g., Stack data structure
    
    Data: list of elements, index to the top element of the list
    
    Operations: create, push, pop, top, isEmpty

- Abstract Data Types (ADTs)
  
  Encapsulation + Information Hiding (using access control)

- Object-Oriented Programming (OOP)
  
  Motivation
  
  Inheritance
  
  Dynamic Binding of Method Calls (Polymorphism)
Object-Oriented Programming (OOP)

- Concept of OOP has its roots in SIMULA 67
  - Not fully developed until Smalltalk 80

- An object-oriented language must provide support for
  - Abstract data types
  - Inheritance
  - Dynamic binding of method calls to methods

Motivation for OOP

- Observations of mid-late 1980s
  - Software reuse may increase productivity
  - ADTs provide encapsulation and access controls, but are still difficult to reuse
  - Not flexible
    - Often requires minor modifications to old types
    - e.g., Square versus Rectangle objects
    - In many cases, modifications require changes to client programs
  - ADTs are independent of each other
    - Cannot organize them according to parent-child or sibling relationships

- Inheritance solves both problems
  - reuse ADTs after minor changes
  - define related classes in a hierarchical structure

Basic Definitions of OOP

- ADTs are often called classes that define
  - format of object
    - (instance variables = data representation)
  - operations on objects (methods = functions)
  - operations for creating/destroying objects

- Object-Oriented Design
  - Calls to methods are sometimes called messages
  - The entire collection of methods of an object is called its message protocol or message interface

Account c;
c.Deposit(10);
### Basic Definitions of OOP

- There are two kinds of variables in a class:
  - **Class variables**
    - one copy for the entire class (e.g., minimum balance)
  - **Instance variables**
    - one copy for each object

- There are two kinds of methods in a class:
  - **Class methods**
    - perform operations on the class, and possibly on objects of the class
  - **Instance methods**
    - perform operations on objects of the class

### Object-Oriented Languages

- **Pure OOP languages**
  - Smalltalk

- **OOP is added to an existing language**
  - C++ (also supports procedural and data-oriented programming)
  - Ada 95 (also supports procedural and data-oriented programming)
  - CLOS (also supports functional programming)
  - Scheme (also supports functional programming)

- **Languages designed for OOP, but still employ basic structure of earlier imperative languages**
  - Eiffel (not based directly on any previous language)
  - Java (based on C++)

### Inheritance

- **Defining an ADT as a variant of another ADT,**
  - may reuse some/all of the entities of original ADT

- **A class that inherits from another class is called a derived class or subclass**

- **The class from which another class inherits is called the parent class or superclass**

- **In the simplest case, a class inherits all entities (variables and methods) of its parent**
Inheritance

- Inheritance can be complicated by access controls to encapsulated entities
- A class can hide entities from its clients (e.g., protected modifier)
- A class can hide entities from its subclasses (e.g., private modifier)
- Derived class can add new entities and modify inherited methods
- The new method is said to override the inherited version

```java
class GraphicalObject {
    protected object_type;
    private int x, y;
    public void draw() { ... }
    public void move(int x, int y) {
        this.x = x; this.y = y;
    }
}
Derived class line extends GraphicalObject {
    public void draw() { ... }
}
```

Inheritance

- Single vs. Multiple Inheritance
  - Single: if the new class is a subclass of a single parent class
  - Multiple: if the new class has more than one parent class

- Disadvantage of Inheritance
  - Creates dependencies among classes
  - Must be careful when making modifications to a class
  - Conflicting requirements: abstract data types vs inheritance

Polymorphism

- polymorphic ("taking many forms") operating on many objects of many forms (but same interface)

- Different kinds of polymorphisms
  - Overloaded subprograms are ad hoc polymorphism
  - Generic subprograms are parametric polymorphism
  - In OOP, another kind of polymorphism due to dynamic binding of messages (calls) to method definitions
Polymorphism

Variables of a parent class are allowed to reference objects of any of the subclasses of that class.

GraphicalObject obj; // a local variable
Line line; // Line is a subclass of
// GraphicalObject
... // some code that initializes obj and line
obj = line; // ok

If a method in GraphicalObject class is also present in the Line class, obj is bound to the method in the proper class dynamically.

Dynamic binding of method calls

---

```java
import java.io.*;

class A {
    int x = 1;
    public void showValue() {
        System.out.println(x);
    }
}

class B extends A {
    int y = 2;
    public void showValue() {
        System.out.println(y);
    }
}

class mainProgram {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.showValue();
        b.showValue();
        a = b;
        a.showValue();
    }
}
```

---

```java
import java.io.*;

class Shape {
    void draw() {
    }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing circle");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Drawing square");
    }
}

class Rectangle extends Shape {
    void draw() {
        System.out.println("Drawing rectangle");
    }
}

class anyShape {
    Shape s;
    public anyShape (Shape d) { super(); s = d; }
    public void drawShape () { s.draw(); }
}

class buildShapes {
    public static void main(String[] args) {
        Circle c = new Circle();
        Rectangle r = new Rectangle();
        Square s = new Square();
        anyShape d;
        d = new anyShape(c);
        d.drawShape();
        d = new anyShape(r);
        d.drawShape();
        d = new anyShape(s);
        d.drawShape();
    }
}
```

---

```
import java.io.*;

class Shape {
    void draw() {
    }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing circle");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Drawing square");
    }
}

class Rectangle extends Shape {
    void draw() {
        System.out.println("Drawing rectangle");
    }
}

class anyShape {
    Shape s;
    public anyShape (Shape d) { super(); s = d; }
    public void drawShape () { s.draw(); }
}

class buildShapes {
    public static void main(String[] args) {
        Circle c = new Circle();
        Rectangle r = new Rectangle();
        Square s = new Square();
        anyShape d;
        d = new anyShape(c);
        d.drawShape();
        d = new anyShape(r);
        d.drawShape();
        d = new anyShape(s);
        d.drawShape();
    }
}
```
Abstract (Virtual) Class/Method

- Abstract (Virtual) method
  - Includes only the protocol of a method but not the body
  - class Shape : public WindowObject {
    public:
      Shape(SimpleWindow &w, const Position &p, const color c = Red);
      color GetColor() const;
      void SetColor(const color c);
      virtual void Draw(); // virtual function in C++
    private:
      color Color;
    }
  - A class that includes at least one abstract method is called an abstract class
  - Abstract class cannot be instantiated because not all of its methods have bodies
  - Subclass of an abstract class must provide implementations of all of the inherited abstract methods

Java interfaces

- Similar to class but no implementation
- No instance variables
- No method bodies
- Like Ada package specification
- A named collection of method definitions
- Multiple super-interfaces
  - Interface Foo extends Printable, Drawable

Java interfaces

- When to use class vs. interface
  - Class Printable:
    void print() { /* default implementation */ }
  - Class Person extends Printable:
    void print() { /* overriding method */ }
  - interface Printable:
    void print();
  - Class Person implements Printable:
    void print() { /* body of method */ }

Java interfaces

- Classes declare which interfaces they support
- Class can implement many interfaces
- Class Foo implements Drawable, Printable
- Many classes can support an interface
- Class Point implements Printable
- Class Person implements Printable

Java interfaces

- void foo (Printable p) { ... p.print(); ... }
- With Printable as a class
  - Actual parameter must be a subclass
  - Must inherit the implementation of Printable
  - Specify both interface and implementation
- With Printable as an interface
  - Actual parameter must be an instance of a class that implements the Printable interface
  - Specifies only interface
  - Fewer constraints => more reusable

Design Issues

- The Exclusivity of Objects
  - Everything is an object
    - No distinction between predefined and user-defined classes
    - Advantage – elegance and purity
    - Disadvantage – slow operations on simple objects (e.g., float)
  - Include imperative style typing system for primitives but make everything else objects
    - Advantage
      - Fast operations on simple objects and a relatively small typing system
    - Disadvantage
      - Still some confusion because of the two type systems
      - Need wrapper classes for nonobject types, so that some commonly needed operations can be sent to objects with nonobject type values
        - E.g., in Java, INTEGER objects are wrappers for integer types
Design Issues

- Are Subclasses Subtypes?
  - Does an "is-a" relationship hold between a derived class object and its parent class?
  - If it holds, then all operations of the parent class can be applied to derived class
  - Subtypes in Ada:
    - Subtype Small_int is Integer range –100..100
  - A derived class is called a subtype if
    - it has an is-a relationship with its parent class
    - Subclass can only add variables/methods and override inherited methods in "compatible" ways
      - Compatible means that the overriding method can replace the overridden method without causing type errors (e.g., having identical number of parameters, identical parameter types and return type)
    - Disallows cases where entities in the parent class are not inherited by subclass

Design Issues

- Implementation and Interface Inheritance
  - Interface Inheritance
    - If only the interface of the parent class is visible to the subclass
    - Disadvantage - can result in inefficiencies
  - Implementation Inheritance
    - If both the interface and the implementation of the parent class are visible to the subclass
    - Disadvantage - changes to the parent class require recompilation of subclasses, and sometimes even modification of subclasses

Design Issues

- Type Checking and Polymorphism
  - Polymorphism may require dynamic type checking of parameters and the return value
    - Dynamic type checking is costly and delays error detection
  - If overriding methods are restricted to having the same parameter types and return type,
    - then checking can be static
  - Dynamic type checking is more expensive and delays type error detection
Design Issues

- Single and Multiple Inheritance
  - Advantage of multiple inheritance
    - Sometimes it is more convenient and valuable
  - Disadvantage of multiple inheritance
    - Conflicting definitions (e.g., two or more superclasses define a print method)
    - Repeated inheritance: same class inherited more than once (temporarily)
    - Language and implementation complexity
    - Potential inefficiency - dynamic binding costs more with multiple inheritance
    - More complex dependencies among classes – maintenance could be a problem

- Allocation and Deallocation of Objects
  - From where are objects allocated?
    - If they all live in the heap, advantage is having a uniform method to create and access the objects
  - Is deallocation explicit or implicit?
    - If implicit, some method of storage reclamation is required, e.g., garbage collection

- Dynamic and Static Binding
  - Should all binding of messages to methods be dynamic?
    - Static bindings are faster

Overview of Smalltalk

- Smalltalk is a pure OOP language
  - everything is an object
  - all computation is through objects sending messages to objects
  - it adopts none of the appearance of imperative languages

- The Smalltalk environment
  - the first complete GUI system
  - a complete system for software development
  - all of the system source code is available to the user, who can modify it
Introduction to Smalltalk

Expressions

Four kinds

1. literals (numbers, strings, and keywords)
2. variable names (all variables are references)
3. message expressions
4. block expressions

Message Expressions

Two parts:

the receiver object and
the message itself

Message part specifies:

the method and
possibly some parameters

Replies to messages are objects

Multiple messages to the same object can be strung together, separated by semicolons

Messages

Messages can be of three forms

1. Unary (no parameters)
   e.g., myAngle sin (sends a message to the sin method of the myAngle object)
2. Binary (one parameter, an object)
   e.g., 12 + 17 (sends the message “+17” to the object 12; the object parameter is “17” and the method is “+”)
3. Keyword (use keywords to organize the parameters)
   e.g., myArray at: 1 put: 5 (sends the objects “1” and “5” to the at:put: method of the object myArray)
Methods

General form:
message_pattern | temps | statements

- a message pattern is like the formal parameters of a subprogram
  - for unary messages, it is just the name
  - for others, it lists keywords and formal names

- temps are just names -- Smalltalk is typeless!

Assignments

Simplest form:
name1 ? name2

- It is simply a pointer assignment

- RHS can be a message expression
e.g., index ? index + 1

Blocks

A sequence of statements, separated by periods, delimited by brackets
e.g.,
[index ? index + 1. sum ? sum + index]

- A block specifies something, but doesn’t do it
- To request the execution of a block, send it the unary message value
e.g., [ ... ] value
Blocks (continued)

- If a block is assigned to a variable, then it is evaluated by sending value to that variable
  
  e.g., addIndex \( \oplus \) \([\text{sum} \oplus \text{sum} + \text{index}]\)
  
  addIndex value

- Blocks can have parameters, as in \([x \ y \mid \text{statements}]\)

- If a block contains a relational expression, then it returns a Boolean object, true or false

Iteration

- Objects true and false have methods for building control constructs

  - "method whileTrue:"

  - may have a pretest logical loop.

  - defined for all blocks that return Boolean objects.

  e.g., \([\text{count} \ ? \ 20] \mid \text{whileTrue} [\text{sum} \ ? \ \text{sum} + \text{count} \mid \text{count} \ ? \ \text{count} + 1] \mid \text{count} \ ? \ 1]"

Iteration (continued)

- "int timesRepeat:" is defined for integers and can be used to build counting loops

  e.g.,

  \(\text{xCube} \ ? \ 1\)

  \(3 \times \text{timesRepeat: [xCube} \ ? \ \text{xCube} * \ x]\)
Selection

- The Boolean objects have the method “ifTrue:” “ifFalse:”, which can be used to build selection

  e.g.,
  ```smalltalk
  total = 0
  ifTrue: [ … ]
  ifFalse: [ … ]
  ```

Large-scale Features of Smalltalk

- Type checking and polymorphism
  all bindings of messages to methods is **dynamic**
  Process:
  - search the object to which the message is sent for the method;
  - if not found, then search the superclass, etc.

  because all variables are typeless, methods are all polymorphic

Large-scale Features of Smalltalk

- Inheritance
  all subclasses are subtypes (nothing can be hidden)
  all inheritance is implementation inheritance
  no multiple inheritance
  methods can be refined – overriding.

  If you want to explicitly reference the superclass version, then the message should have the “super” prefix.
Overview of C++

- C++ is an OO, imperative hybrid
- Designed starting from C
- Initially simply added facilities to support classes to take advantage of benefits of smalltalk features
- Syntax
  - Methods called member functions
  - Instance variables are called data members

C++ Language Design

- Design
  - Organize language in classes (as SIMULA 97)
  - No performance penalty with respect to C
  - Sequential Additions
    - Classes
    - Virtual functions, Operator Overloading, Reference Types
    - Multiple inheritance, abstract classes
    - Templates, parameterized types, exception handling
  - “More a collection of ideas thrown together than the result of an overall language design plan.”

General Characteristics of C++

- Almost completely backward compatible with C
- Memory Allocation
  - Static
  - Stack-dynamic
  - Heap allocated
  - New
  - Delete (no implicit reclamation)
Inheritance

- Multiple
- Inheritance Access Modes
  - Public
  - Private
- Member Function Access Modes
  - Public
  - Protected
  - Private
  - Friend

Inheritance (continued)

- Access Modes
- Build subclasses without subtypes, Example:

```cpp
class stack : private single_linked_list {
public:
    stack();
    void push(int value) {
        single_linked_list::insert_at_head();
    }
    int pop() {
        return single_linked_list::remove_at_head();
    }
};
```

Public, Protected, Private Inheritance

- Class A declares 3 variables
  - i is public to all users of class A
  - j is protected, i.e., it can only be used by methods in class A or its derived classes
  - k is private, i.e., it can only be used by methods in class A
- Class B inherits publicly from A
  - i is again public to all users of class B
  - j is again protected, i.e., it can be used by methods in class B or its derived classes
- Class C uses protected inheritance from A
  - i is now protected in C, so the only users of class C that can access i are the methods of class C
  - j is again protected, i.e., it can be used by methods in class C or its derived classes
- Class D uses private inheritance from A
  - i and j are private in D, so users of D cannot access them, only methods of D itself
Java

- General Characteristics
  - All data are objects except the primitive types
  - All primitive types have wrapper classes that store one data value
  - All objects are heap-dynamic, are referenced through reference variables, and most are allocated with new
  - Inheritance
    - Single inheritance only, but there is an abstract class category that provides some of the benefits of multiple inheritance (interface)
    - An interface can include only method declarations and named constants

```java
public class Clock extends Applet
    implements Runnable
```

Ada 95

- General Characteristics
  - OOP was one of the most important extensions to Ada 83
  - Encapsulation container is a package that defines a tagged type
  - A tagged type is one in which every object includes a tag to indicate during execution its type
  - Tagged types can be either private types or records
  - No constructors or destructors are implicitly called

```ada
package Person_Pkg is
    type Person is tagged private;
    procedure Display(P : in person);
private
    type Person is tagged record
        Name : String (1..30);
        Address : String (1..30);
        Age : Integer;
    end record;
end Person_Pkg;
```
Ada 95

- Inheritance
  - Subclasses are derived from tagged types
  - New entities in a subclass are added in a record

```ada
with PERSON_PKG; use PERSON_PKG;
package STUDENT_PKG is
  type STUDENT is new PERSON with
    record
      GRADE_POINT_AVERAGE : FLOAT;
      GRADE_LEVEL : INTEGER;
    end record;
  procedure DISPLAY (ST: in STUDENT);
end STUDENT_PKG;
```

- DISPLAY is being overridden from PERSON_PKG
- All subclasses are subtypes
- Single inheritance only, except through generics

Ada 95

- Dynamic Binding
  - Dynamic binding is done using polymorphic variables called classwide types
  - e.g., for the tagged type PERSON, the classwide type is PERSON'class
  - Other bindings are static
  - Any method may be dynamically bound

Eiffel

- General Characteristics
  - Has primitive types and objects
  - All objects get three operations, copy, clone, and equal
  - Methods are called routines
  - Instance variables are called attributes
  - The routines and attributes of a class are together called its features
  - Object creation is done with an operator (!!)
  - Constructors are defined in a creation clause, and are explicitly called in the statement in which an object is created
Eiffel

- Inheritance
  - The parent of a class is specified with the inherit clause
- Access control
  - feature clauses specify access control to the entities defined in them
  - Without a modifier, the entities in a feature clause are visible to both subclasses and clients
  - With the child modifier, entities are hidden from clients but are visible to subclasses
  - With the none modifier, entities are hidden from both clients and subclasses
- Inherited features can be hidden from subclasses with undefine
- Abstract classes can be defined by including the deferred modifier on the class definition

Eiffel

- Dynamic Binding
  - Nearly all message binding is dynamic
  - An overriding method must have parameters that are assignment compatible with those of the overriden method
  - All overriding features must be defined in a redefine clause
  - Access to overriden features is possible by putting their names in a rename clause
- Evaluation
  - Similar to Java in that procedural programming is not supported and nearly all message binding is dynamic
  - Elegant and clean design of support for OOP

Threads

- Java allows for some level of multiprocessing through threads.
- Threads are defined by the new Thread(object) command.
- The synchronized attribute (synchronized void startSort()) on multiple-method definitions prevents more than one of them from executing at a time, and thus avoids deadlock situations.
Threads -2

- Created by the new command
- Executed by the run() method;
  - started with start()
- Executes until the stop() method or run() completes execution
- Threads may suspend() and resume() execution.

Synchronization:
- Use monitors with synchronized attribute:
  public synchronized void methodName(int value)
  {val = returned_value;}

Organization of Programming Languages -Cheng (Fall 2004)