Chapter 12

Support for Object-Oriented Programming
Chapter 12 Topics

- Introduction
- Object-Oriented Programming
- Design Issues for Object-Oriented Languages
- Support for Object-Oriented Programming in Smalltalk
- Support for Object-Oriented Programming in C++
- Support for Object-Oriented Programming in Java
- Support for Object-Oriented Programming in C#
- The Object Model of JavaScript
- Implementation of Object-Oriented Constructs
Introduction

• Many object-oriented programming (OOP) languages
  – Some support procedural and data-oriented programming (e.g., Ada and C++)
  – Some support functional program (e.g., CLOS)
  – Newer languages do not support other paradigms but use their imperative structures (e.g., Java and C#)
  – Some are pure OOP language (e.g., Smalltalk)
Object–Oriented Programming

- Abstract data types
- Inheritance
  - Inheritance is the central theme in OOP and languages that support it
- Polymorphism
Inheritance

• Productivity increases can come from reuse
  – ADTs are difficult to reuse
  – All ADTs are independent and at the same level
• Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts
• Inheritance addresses both of the above concerns—reuse ADTs after minor changes and define classes in a hierarchy
Object–Oriented Concepts

- ADTs are called *classes*
- Class instances are called objects
- A class that inherits is a *derived class* or a *subclass*
- The class from which another class inherits is a parent class or *superclass*
- Subprograms that define operations on objects are called *methods*
Object-Oriented Concepts (continued)

- Calls to methods are called *messages*
- The entire collection of methods of an object is called its *message protocol* or *message interface*
- Messages have two parts--a method name and the destination object
- In the simplest case, a class *inherits* all of the entities of its parent
Object–Oriented Concepts (continued)

• Inheritance can be complicated by access controls to encapsulated entities
  – A class can hide entities from its subclasses
  – A class can hide entities from its clients
  – A class can also hide entities for its clients while allowing its subclasses to see them

• Besides inheriting methods as is, a class can modify an inherited method
  – The new one *overrides* the inherited one
  – The method in the parent is *overridden*
Three ways a class can differ from its parent:

1. The subclass can add variables and/or methods to those inherited from the parent.
2. The subclass can modify the behavior of one or more of its inherited methods.
3. The parent class can define some of its variables or methods to have private access, which means they will not be visible in the subclass.
Object-Oriented Concepts (continued)

- There are two kinds of variables in a class:
  - *Class variables* – one/class
  - *Instance variables* – one/object

- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects

- Single vs. Multiple Inheritance

- One disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance
Dynamic Binding

• A polymorphic variable can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants
• When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic
• Allows software systems to be more easily extended during both development and maintenance
Dynamic Binding Concepts

- An *abstract method* is one that does not include a definition (it only defines a protocol)
- An *abstract class* is one that includes at least one virtual method
- An abstract class cannot be instantiated
Design Issues for OOP Languages

• The Exclusivity of Objects
• Are Subclasses Subtypes?
• Single and Multiple Inheritance
• Object Allocation and Deallocation
• Dynamic and Static Binding
• Nested Classes
• Initialization of Objects
The Exclusivity of Objects

- Everything is an object
  - Advantage – elegance and purity
  - Disadvantage – slow operations on simple objects
- Add objects to a complete typing system
  - Advantage – fast operations on simple objects
  - Disadvantage – results in a confusing type system (two kinds of entities)
- Include an 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
Are Subclasses Subtypes?

- Does an “is–a” relationship hold between a parent class object and an object of the subclass?
  - If a derived class is–a parent class, then objects of the derived class must behave the same as the parent class object

- A derived class is a subtype if it has an is–a relationship with its parent class
  - Subclass can only add variables and methods and override inherited methods in “compatible” ways
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, the checking can be static
Single and Multiple Inheritance

- Multiple inheritance allows a new class to inherit from two or more classes
- Disadvantages of multiple inheritance:
  - Language and implementation complexity (in part due to name collisions)
  - Potential inefficiency – dynamic binding costs more with multiple inheritance (but not much)
- Advantage:
  - Sometimes it is extremely convenient and valuable
Allocation and De-Allocation of Objects

• From where are objects allocated?
  – If they behave like the ADTs, they can be allocated from anywhere
    • Allocated from the run-time stack
    • Explicitly create on the heap (via `new`)
  – If they are all heap-dynamic, references can be uniform thru a pointer or reference variable
    • Simplifies assignment – dereferencing can be implicit
  – If objects are stack dynamic, there is a problem with regard to subtypes (copy values)

• Is deallocation explicit or implicit?
Dynamic and Static Binding

• Should all binding of messages to methods be dynamic?
  – If none are, you lose the advantages of dynamic binding
  – If all are, it is inefficient
• Allow the user to specify
Nested Classes

• If a new class is needed by only one class, there is no reason to define so it can be seen by other classes
  - Can the new class be nested inside the class that uses it?
  - In some cases, the new class is nested inside a subprogram rather than directly in another class

• Other issues:
  - Which facilities of the nesting class should be visible to the nested class and vice versa
Support for OOP in Smalltalk

- Smalltalk is a pure OOP language
  - Everything is an object
  - All objects have local memory
  - All computation is through objects sending messages to objects
  - None of the appearances of imperative languages
  - All objects are allocated from the heap
  - All deallocation is implicit

\[ 3 \text{ factorial} + 4 \text{ factorial} \text{ between: 10 and: 100} \]

\[ 10 < (3\times2) + (4\times3\times2) < 100 \implies \text{true} \]
Support for OOP in Smalltalk (continued)

• Type Checking and Polymorphism
  - All binding of messages to methods is dynamic
    • The process is to search the object to which the message is sent for the method; if not found, search the superclass, etc. up to the system class which has no superclass
  - The only type checking in Smalltalk is dynamic and the only type error occurs when a message is sent to an object that has no matching method
Support for OOP in Smalltalk (continued)

• Inheritance
  – A Smalltalk subclass inherits all of the instance variables, instance methods, and class methods of its superclass
  – All subclasses are subtypes (nothing can be hidden)
  – All inheritance is implementation inheritance
  – No multiple inheritance
Support for OOP in Smalltalk (continued)

- **Evaluation of Smalltalk**
  - The syntax of the language is simple and regular
  - Good example of power provided by a small language
  - Slow compared with conventional compiled imperative languages
  - Dynamic binding allows type errors to go undetected until run time
  - Greatest impact: advancement of OOP
Support for OOP in C++

• General Characteristics:
  – Evolved from SIMULA 67
  – Most widely used OOP language
  – Mixed typing system
  – Constructors and destructors
  – Elaborate access controls to class entities
Support for OOP in C++ (continued)

- **Inheritance**
  - A class need not be the subclass of any class
  - Access controls for members are
    - Private (visible only in the class and friends) (disallows subclasses from being subtypes)
    - Public (visible in subclasses and clients)
    - Protected (visible in the class and in subclasses, but not clients)
Support for OOP in C++ (continued)

- In addition, the subclassing process can be declared with access controls (private or public), which define potential changes in access by subclasses
  - Private derivation – inherited public and protected members are private in the subclasses
  - Public derivation public and protected members are also public and protected in subclasses
Inheritance Example in C++

class base_class {
private:
    int a;
    float x;
protected:
    int b;
    float y;
public:
    int c;
    float z;
};

class subclass_1 : public base_class { ... };
// In this one, b and y are protected and
// c and z are public

class subclass_2 : private base_class { ... };
// In this one, b, y, c, and z are private,
// and no derived class has access to any
// member of base_class
Reexportation in C++

- A member that is not accessible in a subclass (because of private derivation) can be declared to be visible there using the scope resolution operator (::), e.g.,

```cpp
class subclass_3 : private base_class {
    base_class :: c;
    ...
}
```
Reexportation (continued)

• One motivation for using private derivation
  – A class provides members that must be visible, so they are defined to be public members; a derived class adds some new members, but does not want its clients to see the members of the parent class, even though they had to be public in the parent class definition
Support for OOP in C++ (continued)

• Multiple inheritance is supported
  – If there are two inherited members with the same name, they can both be referenced using the scope resolution operator
• Dynamic Binding
  – A method can be defined to be *virtual*, which means that they can be called through polymorphic variables and dynamically bound to messages
  – A pure virtual function has no definition at all
  – A class that has at least one pure virtual function is an *abstract class*
Support for OOP in C++ (continued)

• Evaluation
  – C++ provides extensive access controls (unlike Smalltalk)
  – C++ provides multiple inheritance
  – In C++, the programmer must decide at design time which methods will be statically bound and which must be dynamically bound
    • Static binding is faster!
  – Smalltalk type checking is dynamic (flexible, but somewhat unsafe)
  – Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++
Support for OOP in Java

• Because of its close relationship to C++, focus is on the differences from that language

• 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
  – A finalize method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object
Support for OOP in Java (continued)

• Inheritance
  - Single inheritance supported 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, e.g.,
    public interface Comparable {
      public int compareTo (Object b);
    }
  - Methods can be final (cannot be overridden)
Support for OOP in Java (continued)

• Dynamic Binding
  – In Java, all messages are dynamically bound to methods, unless the method is \texttt{final} (i.e., it cannot be overridden, therefore dynamic binding serves no purpose)
  – Static binding is also used if the methods is \texttt{static} or \texttt{private} both of which disallow overriding
Support for OOP in Java (continued)

- Several varieties of nested classes
- All can be hidden from all classes in their package, except for the nesting class
- Nested classes can be anonymous
- A local nested class is defined in a method of its nesting class
  - No access specifier is used

```java
myButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        // do stuff here...
    }
});
```
Support for OOP in Java (continued)

• Evaluation
  - Design decisions to support OOP are similar to C++
  - No support for procedural programming
  - No parentless classes
  - Dynamic binding is used as “normal” way to bind method calls to method definitions
  - Uses interfaces to provide a simple form of support for multiple inheritance
Support for OOP in C#

• General characteristics
  – Support for OOP similar to Java
  – Includes both classes and structs
  – Classes are similar to Java’s classes
  – structs are less powerful stack–dynamic constructs
Support for OOP in C# (continued)

• Inheritance
  – Uses the syntax of C++ for defining classes
  – A method inherited from parent class can be replaced in the derived class by marking its definition with `new`
  – The parent class version can still be called explicitly with the prefix `base`:
    ```
    base.Draw()
    ```
Support for OOP in C#

- Dynamic binding
  - To allow dynamic binding of method calls to methods:
    - The base class method is marked `virtual`
    - The corresponding methods in derived classes are marked `override`
  - Abstract methods are marked `abstract` and must be implemented in all subclasses
  - All C# classes are ultimately derived from a single root class, `Object`
Support for OOP in C#

• Evaluation
  – C# is the most recently designed C–based OO language
  – The differences between C#’s and Java’s support for OOP are relatively minor
Support for OOP in Ruby

• General Characteristics
  - Everything is an object
  - All computation is through message passing
  - Class definitions are executable, allowing secondary definitions to add members to existing definitions
  - Method definitions are also executable
  - All variables are type-less references to objects
  - Access control is different for data and methods
    • It is private for all data and cannot be changed
    • Methods can be either public, private, or protected
      • Method access is checked at runtime
  - Getters and setters can be defined by shortcuts
Support for OOP in Ruby (continued)

• Inheritance
  – Access control to inherited methods can be different than in the parent class
  – Subclasses are not necessarily subtypes

• Dynamic Binding
  – All variables are typeless and polymorphic

• Evaluation
  – Does not support abstract classes
  – Does not fully support multiple inheritance
  – Access controls are weaker than those of other languages that support OOP
The Object Model of JavaScript

• General Characteristics of JavaScript
  – Little in common with Java
    • Similar to Java only in that it uses a similar syntax
  – Dynamic typing
  – No classes or inheritance or polymorphism
  – Variables can reference objects or can directly access primitive values
The Object Model of JavaScript

- **JavaScript objects**
  - An object has a collection of properties which are either data properties or method properties
  - Appear as hashes, both internally and externally
  - A list of property/value pairs
  - Properties can be added or deleted dynamically
  - A bare object can be created with `new` and a call to the constructor for `Object`
    ```javascript
    var my_object = new Object();
    ```
  - References to properties are with dot notation
JavaScript Evaluation

- Effective at what it is designed to be
  - A scripting language
- Inadequate for large scale development
- No encapsulation capability of classes
  - Large programs cannot be effectively organized
- No inheritance
  - Reuse will be very difficult
Implementing OO Constructs

• Two interesting and challenging parts
  – Storage structures for instance variables
  – Dynamic binding of messages to methods
Instance Data Storage

- Class instance records (CIRs) store the state of an object
  - Static (built at compile time)
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Because CIR is static, access to all instance variables is done as it is in records
  - Efficient
Dynamic Binding of Methods Calls

• Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR
  – Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  – The storage structure is sometimes called *virtual method tables* (vtable)
  – Method calls can be represented as offsets from the beginning of the vtable
CIR with single inheritance (1)

```java
public class A {
    public int a, b;
    public void draw () { …}
    public void area () { …}
}

public class B extends A {
    public int c, d;
    public void draw () { …}
    public void sift () { …}
}
```
CIR with single inheritance (2)

**Figure 12.2**
An example of the CIRs with single inheritance
Reflection

- A programming language that supports reflection allows its programs to have runtime access to their types and structure and to be able to dynamically modify their behavior.
- The types and structure of a program are called *metadata*.
- The process of a program examining its metadata is called *introspection*.
- Interceding in the execution of a program is called *intercession*.
Reflection (continued)

- **Uses of reflection for software tools:**
  - Class browsers need to enumerate the classes of a program
  - Visual IDEs use type information to assist the developer in building type correct code
  - Debuggers need to examine private fields and methods of classes
  - Test systems need to know all of the methods of a class
Reflection in Java

- Limited support from `java.lang.Class`
- Java runtime instantiates an instance of `Class` for each object in the program
- The `getClass` method of `Class` returns the `Class` object of an object

```java
float[] totals = new float[100];
Class fltlist = totals.getClass();
Class stg = "hello".getClass();
```

- If there is no object, use `class` field

```java
Class stg = String.class;
```
Reflection in Java (continued)

- **Class** has four useful methods:
  - `getMethod` searches for a specific public method of a class
  - `getMethods` returns an array of all public methods of a class
  - `getDeclaredMethod` searches for a specific method of a class
  - `getDeclaredMethods` returns an array of all methods of a class
Reflection in Java (continued)

• The `Method` class defines the `invoke` method, which is used to execute the method found by `getMethod`
Downsides of Reflection

- Performance costs
- Exposes private fields and methods
- Voids the advantages of early type checking
- Some reflection code may not run under a security manager, making code nonportable
Summary

- **OO programming involves three fundamental concepts:** ADTs, inheritance, dynamic binding
- **Major design issues:** exclusivity of objects, subclasses and subtypes, type checking and polymorphism, single and multiple inheritance, dynamic binding, explicit and implicit deallocation of objects, and nested classes
- Smalltalk is a pure OOL
- C++ has two distinct type systems (hybrid)
- Java is not a hybrid language like C++; it supports only OOP
- C# is based on C++ and Java
- Ruby is a relatively recent pure OOP language; provides some new ideas in support for OOP
- Implementing OOP involves some new data structures
- Reflection is part of Java and C#, as well as most dynamically typed languages
Class A {
    static int Sv = 0;
    int Nv = 0;
    public static void Sm() {}
    public void Nm() {}
}

A.Sm();

......
A v1 = new A();
v1.Nm();
A v2 = new A();
V1.Nv = 1;

......