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
Object-Oriented Programming

- Three key features in OO programming
  - Abstract data types (Ch11)
  - Inheritance
  - Dynamic Binding
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**
- Calls to methods are called **messages**
  - Messages have two parts—a method name and the destination object
- In the simplest case, a class **inherits** all of the entities (data and methods) of its parent class
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 add new entities and modify an inherited method

- The new one **overrides** the inherited one
- The method in the parent is **overridden**
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**
  - Multiple inheritance: more than one parents

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

- Dynamic Binding of messages to method definitions
  - 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
    - `ref_shape` (the example in the next slide)
  - 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
    - During execution, the system should determine which method to be called.
  - Allows software systems to be more easily extended during development and maintenance
Dynamic Binding - Example

```cpp
class shape {
    public:
    virtual void draw() = 0;
    // ...
}
class circle : public shape {
    public:
    virtual void draw() { // ...
    // ...
}

circle c1, c2;
shape &ref_shape = c1;
ref_shape.draw();  // dynamically bound to draw in circle
c2.draw();         // statically bound to draw in circle
```
Dynamic Binding Concepts

- Polymorphism
  - An *abstract method* is one that does not include a definition (it only defines a protocol, no body)
    - Called “pure virtual” method in C++
  - An *abstract class* is one that includes at least one virtual method
  - An abstract class cannot be instantiated (because not all of its methods have bodies)
  - Any subclass of an abstract class that is to be instantiated must provide implementations of all the inherited abstract methods
Design Issues for OOP Languages

- The Exclusivity of Objects
- Subclasses as Types
- Type Checking and Polymorphism
- Single and Multiple Inheritance
- Object Allocation and De-Allocation
- Dynamic and Static Binding
- Nested Classes
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
  - `subtype small_int is integer range -100..100;`
- 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
  - Compatible here means the overriding method can replace the overridden method without causing type error.
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 (type check is performed only when polymorphic variable is used to call the overriding method)
- 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)
    - E.g., subclass C inherits from class A and B. Both A and B define an inheritable method named display.
  - Diamond inheritance (shared inheritance)
  - 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?

- 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
- Problems with subtypes If objects are stack dynamic
  - If class B is a child of class A, then an object (said b1) of B type can be assigned to an object (said a1) of A type.
  - `a1 = b1;` //the value of b1 must be copied to the space of a1.
  - a1 have no sufficient space if B adds data to what it inherited.

Is deallocation explicit or implicit?

- Implicit: to implement reference count, garbage collection
- Explicit: dangling problem.
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 de-allocation is implicit
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)
  - 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
    - Traditional imperative language types
    - Class structure of object-oriented languages
  - 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)
  -Derived classes can modify accessibility of the inherited members (private or public)
    - 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 {
  ...
};
// b and y are protected, 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 to the *instances* of the derived class (due to private derivation) can be declared to be visible using scope operator (::), e.g.,

  ```c++
  class subclass_3 : private base_class {
    base_class :: c;
  }
  ```

  - *Now, instances of subclass_3 can access c.* (as if it is public derivation)

- **One motivation for using private derivation**
  - A class provides members that must be visible (so 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
Support for OOP in C++ (continued)

- Multiple inheritance is supported
  - Allow more than one class to be the parent of a new class. E.g.,
    ```
    class A { ... };
    class B { ... };
    class C : public A, public B { ... };
    ```
  - 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 as \textit{virtual}, meaning that they can be called through polymorphic variables

- A polymorphic variable is a pointer (or reference) that has the type of a base class but is used to point to objects of any class derived from that base class.

- When a polymorphic variable is used to call a virtual function defined in one of the derived class, the call is dynamically bound to correct function definition.

- A \textit{pure virtual} function has no definition at all, and cannot be called. It must be redefined in derived classes.

- A class that has at least one pure virtual function is an \textit{abstract class}
Support for OOP in C++ (continued)

- **Evaluation**
  - C++ provides extensive access controls (unlike Smalltalk)
  - C++ provides multiple inheritance (Smalltalk does not)
  - In C++, the programmer can specify at design time which method will be statically bound and which one 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 (e.g., Bool, char, and numeric types) – to make primitive type efficient.
  - Primitive types have **wrapper classes** to store one data value
    - Arrays in Java are objects, such as ArrayList, which can contain only objects. To put a primitive type value, the value must be placed in an object first. E.g.,
      ```java
      myArray.add( new Integer(10) );
      ```
    - After Java 5.0, myArray.add( 10 ); is legal. (by compiler)
  - 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.
Support for OOP in Java

- **Access Modifier**

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>no modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- **Static member:**
  - use the “static” keyword in the member's declaration
  - A member that is declared as static is a class member (variables or methods); otherwise, an instance member.
  - Class variables are shared by all instances of a class and can be accessed through the class name as well as an instance reference.
  - Instances of a class get their own copy of each instance variable, which must be accessed through an instance reference.
Inheritance

- Single inheritance supported only
  - but interface provides benefits of multiple inheritance
- An interface defines a protocol of communication between two objects. It includes only method declarations (no body) and named constants, e.g.,

```java
public interface Comparable {
    int compareTo (Object b);
}
```
- To use an interface, you can write a class that implements the interface. E.g.,

```java
public class myClass implements Comparable {
    int compareTo (Object b) { . . . }
}
```
- A class can implement more than one interface.
Support for OOP in Java (continued)

- Dynamic Binding
  - In Java, all messages are dynamically bound to methods, unless the method is `final` (i.e., it cannot be overridden, therefore dynamic binding serves no purpose)
    - All methods in Java are virtual by default, C++ require clearly defining a method as virtual
  - Static binding is also used if the methods is `static` or `private` both of which disallow overriding
Nested Classes

- Two nested classes: static and non-static.
  - A static nested class is associated with its enclosing class
  - A non-static nested class is associated with an instance of its enclosing class
  - All can be hidden from all classes in their package, except for the nesting class

Why Use Nested Classes?

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- Nested classes can lead to more readable and maintainable code
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`
    - If a method is declared as new but it does not really hide a base method, compiler will issue a warning
  - The parent class version can still be called explicitly with the prefix `base`:
    ```csharp
    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`

```csharp
public class shape {
    public virtual void draw() { ... };
    ...
}

public class circle : shape {
    public override void draw() { ... }
    ...
}
```
Dynamic Binding - Example

- **Dynamic binding**
  - Abstract methods are marked `abstract` and must be implemented in all subclasses.
  - An abstract class cannot be instantiated.

```csharp
public class shape {
    abstract public void draw();
    ...
}
```

- **All C# classes are ultimately derived from a single root class, Object**
  - It defines methods, `ToString`, `Finalize`, `and` `Equals`, which are inherited by all C# types.
Support for OOP in C# (continued)

- **Nested Classes**
  - A C# class that is directly nested in a nesting class behaves like a Java static nested class
  - C# does not support nested classes that behave like the non-static classes of Java
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
The Object Model of JavaScript

- JavaScript was originally designed by Netscape as a scripting language for use in programming Web server or augmenting HTML documents.

- General Characteristics of JavaScript
  - Little in common with Java
    - Similar to Java only in that it uses a similar syntax
  - Dynamic typing
    - No type is implied by variable declaration
    - The type of a variable can change every time a value is assigned to it.
  - No classes or inheritance or polymorphism
  - Variables can reference objects or store primitive type values.
The Object Model of JavaScript

- **JavaScript objects**
  - An object has a collection of properties which are either data properties or method properties
  - 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
  - References to properties are with dot notation

```javascript
var my_car = new Object();
my_car.make = "Ford"; // create and initialize make
my_car.model = "SVT"; // create and initialize model
my_car.engine = new object(); // nested objects
my_car.engine.config = "V6";
```
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
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 de-allocation of objects, and nested classes
- Smalltalk is a pure OOL
- C++ has two distinct type system (hybrid)
- Java is not a hybrid language like C++; it supports only OO programming
- C# is based on C++ and Java
- JavaScript is not an OOP language but provides interesting variations