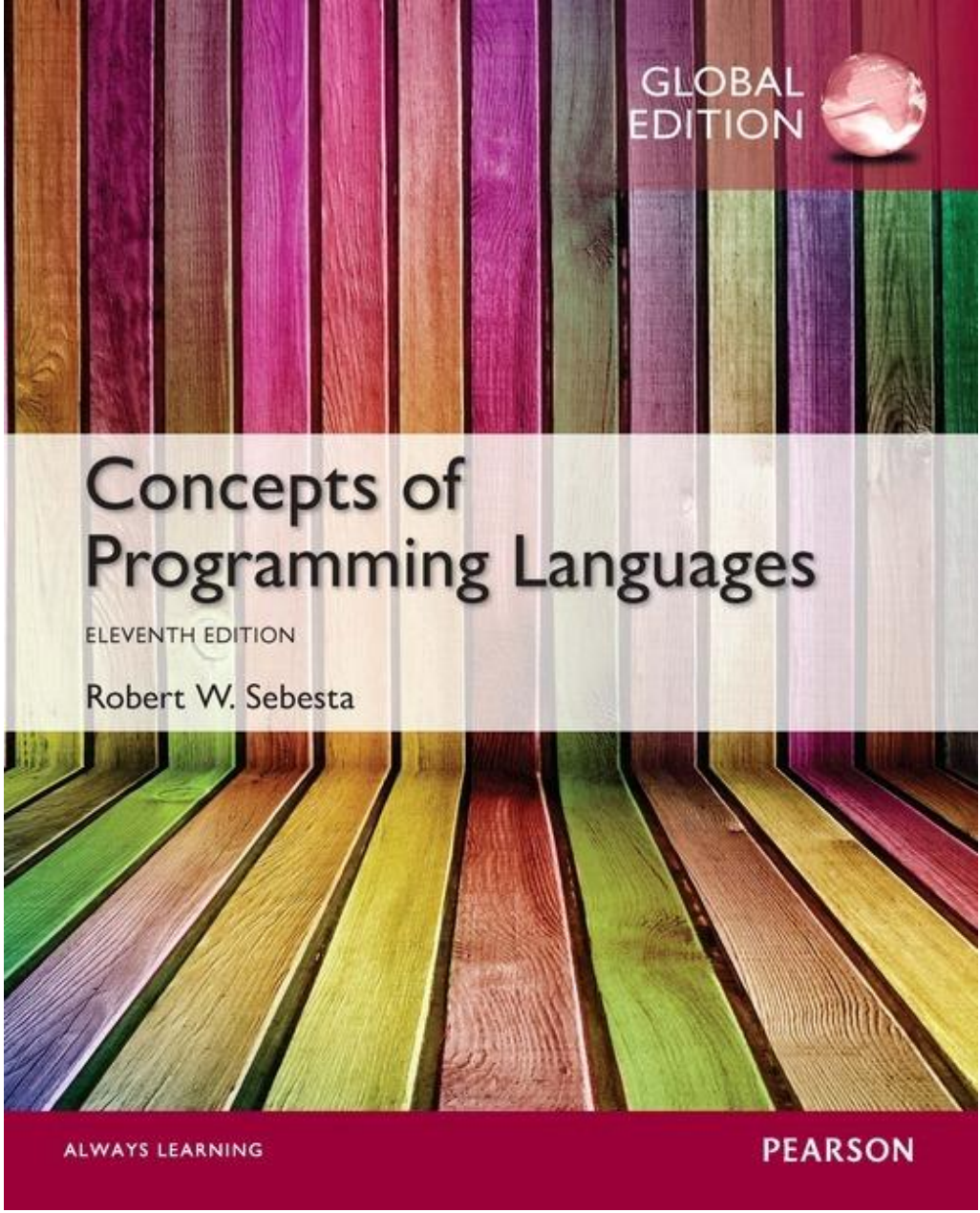
Chapter 12

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 Objective-C
- Support for Object-Oriented Programming in Java
- Support for Object-Oriented Programming in C#
- Support for Object-Oriented Programming in Ruby
- Implementation of Object-Oriented Constructs
- Reflection

Introduction

- Many object-oriented programming (OOP) languages
  - Some support procedural and data-oriented programming (e.g., 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 & Ruby)
  - Some functional languages support OOP, but they are not discussed in this chapter

Object-Oriented Programming

- Three major language features:
  - Abstract data types (Chapter 11)
  - 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—always need changes
  - 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 usually 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 *overriden*

Object-Oriented Concepts (continued)

- 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
- Subclasses inherit implementation; subtypes inherit interface and behavior

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 quite convenient and valuable

Allocation and DeAllocation of Objects

- From where are objects allocated?
  - If they behave line 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 – *object slicing*
- Is deallocation explicit or implicit?

Allocation and DeAllocation of Objects

- *Object slicing*
  - For example, if b1 is a variable of B type and a1 is a variable of A type, then

    a1 = b1;

    is a legal statement.
  - However, if a1 and b1 are stack dynamic, then they are value variables and, if assigned the value of the object, must be copied to the space of the target object. If B adds a data field to what it inherited from A, then a1 will not have sufficient space on the stack for all of b1. The excess will simply be truncated, which could be confusing to programmers who write or use the code. This truncation is called **object slicing**.

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
- Maybe the design should 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

Initialization of Objects

- Are objects initialized to values when they are created?
  - Implicit or explicit initialization
- How are parent class members initialized when a subclass object is created?

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 objected are allocated from the heap
  - All deallocation is implicit
  - Smalltalk classes cannot be nested in other classes

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)

- Dynamic Binding
  - 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)

- 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
  - Introduced the graphical user interface
  - Greatest impact: advancement of OOP

Support for OOP in C++

- General Characteristics:
  - Evolved from C and SIMULA 67
  - Among the most widely used OOP languages
  - 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.,

```
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 (::)

```
class Thread { ... }
class Drawing { ... }
class DrawThread : public Thread, public Drawing
  { … }
```

Support for OOP in C++ (continued)

- 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)

```
class Shape {
  public:
    virtual void draw() = 0;
    ...
};
class Circle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Rectangle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Square : public Rectangle {
  public:
    void draw() { ... }
  ...
};
```

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;
ptr_shape = sq;  // points to a Square
ptr_shape ->draw();  // Dynamically
                     // bound to draw in Square
rect->draw();  // Statically bound to
               // draw in Rectangle
```

Support for OOP in C++ (continued)

- If objects are allocated from the stack, it is quite different

```
Square sq;    // Allocates a Square object from the stack
Rectangle rect;  // Allocates a Rectangle object from the stack
rect = sq;   // Copies the data member values from sq object
rect.draw();  // Calls the draw from Rectangle
```

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 Objective-C

- Like C++, Objective-C adds support for OOP to C
- Design was at about the same time as that of C++
- Largest syntactic difference: method calls
- Interface section of a class declares the instance variables and the methods
- Implementation section of a class defines the methods
- Classes cannot be nested

Support for OOP in Objective-C (continued)

- Inheritance
  - Single inheritance only
  - Every class must have a parent
  - NSObject is the base class

```
@interface myNewClass: NSObject { … }
    …
@end
```

  - Because all public members of a base class are also public in the derived class all subclasses are subtypes
  - Any method that has the same name, same return type, and same number and types of parameters as an inherited method overrides the inherited method
  - An overriden method can be called through **`super`**
  - All inheritance is public (unlike C++)

Support for OOP in Objective-C (continued)

- Inheritance (continued)
- Objective-C has two approaches besides subclassing to extend a class
  - A *category* is a secondary interface of a class that contains declarations of methods (no instance variables

```
#import "Stack.h"
@interface Stack (StackExtend)
  -(int) secondFromTop;
  -(void) full;
@end
```

  - A category is a *mixin* – its methods are added to the parent class
  - The implementation of a category is in a separate implementation: `@implementation Stack (StackExtend)`

Support for OOP in Objective-C (continued)

- Inheritance (continued)
  - The other way to extend a class: protocols
  - A protocol is a list of method declarations

```
@protocol MatrixOps
  -(Matrix *) add: (Matrix *) mat;
  -(Matrix *) subtract: (Matrix *) mat;
@optional
  -(Matrix *) multiply: (Matrix *) mat;
@end
```

  - `MatrixOps` is the name of the protocol
  - The `add` and `subtract` methods must be implemented by class that uses the protocol
  - A class that adopts a protocol must specify it

```
@interface MyClass: NSObject <YourProtocol>
```

Support for OOP in Objective-C (continued)

- Dynamic Binding
  - Different from other OOP languages – a polymorphic variable is of type `id`
  - An `id` type variable can reference any object
  - The run-time system keeps track of the type of the object that an `id` type variable references
  - If a call to a method is made through an `id` type variable, the binding to the method is dynamic

Support for OOP in Objective-C (continued)

- Evaluation
  - Support is adequate, with the following deficiencies:
  - There is no way to prevent overriding an inherited method
  - The use of `id` type variables for dynamic binding is overkill – these variables could be misused
  - Categories and protocols are useful additions

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 <T> {
        public int comparedTo (T b);
}
```

  - Methods can be **`final`** (cannot be overriden)
  - All subclasses are subtypes

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 overriden, therefore dynamic binding serves no purpose)
  - Static binding is also used if the methods is `static` or `private` both of which disallow overriding

Support for OOP in Java (continued)

- Nested Classes
  - All are hidden from all classes in their package, except for the nesting class
  - Nonstatic classes nested directly are called *innerclasses*
    - An innerclass can access members of its nesting class
    - A static nested class cannot access members of its 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

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 (e.g., no inheritance)

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()
```

  - Subclasses are subtypes if no members of the parent class is private
  - Single inheritance only

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# (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 a relatively 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

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

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

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

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

```
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`

Reflection in C#

- In the .NET languages the compiler places the intermediate code in an assembly, along with metadata about the program
- `System.Type` is the namespace for reflection
- `getType` is used instead of `getClass`
- `typeof` operator is used instead of `.class` field
- `System.Reflection.Emit` namespace provides the ability to create intermediate code and put it in an assembly (Java does not provide this capability)

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 de-allocation 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 types languages