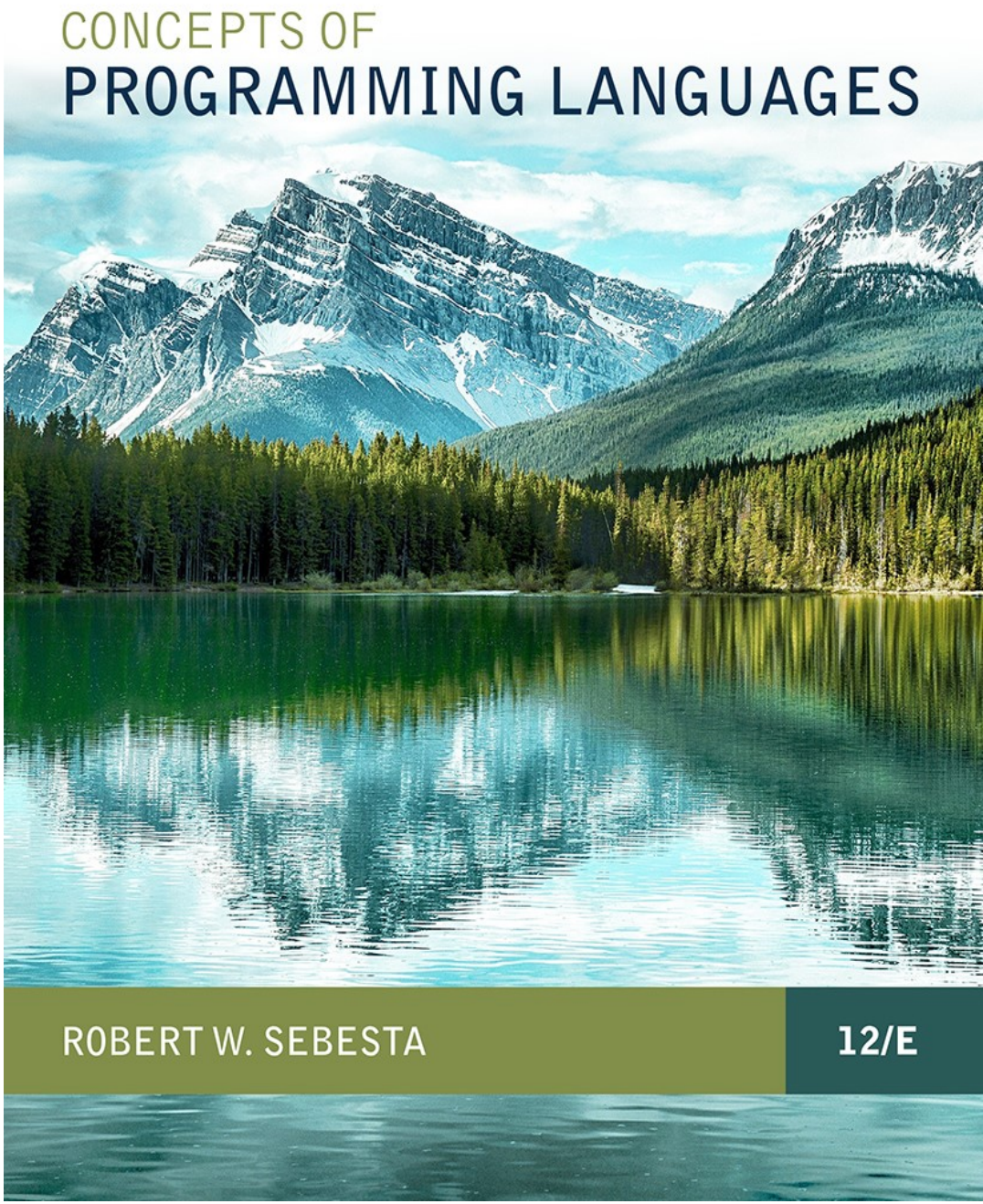
Chapter 12

Support for Object-Oriented Programming



ISBN 0-321-49362-1

Chapter 12 -- Support for OOP

- 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

- Object-oriented programming (OOP) languages
  - Some support procedural and data-oriented programming (e.g., C++)
  - Some support functional program (e.g., Common Lisp Object System)
  - 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).

Object-Oriented Programming

- Three major language features:
  - Abstract data type (ADT)
  - Inheritance
  - Polymorphism.

Inheritance

- Reuse:
  - increase productivity
  - ADTs are difficult to reuse
  - all ADTs are independent and at the same level
- Inheritance:
  - allows new classes defined in terms of the existing ones, i.e., by allowing them to inherit common parts
  - addresses the concerns above
    - reuse ADTs after minor changes
    - define classes in a hierarchy.

Object-Oriented Concepts

- Class:
  - ADTs are usually called classes
  - object: instance of a class
  - *subclass (derived/child class)*:
    - class that is derived from another class
  - *superclass (parent class)*:
    - class from which the subclass is derived
  - subclass inherits all the *members* (fields, methods, nested classes) from its superclass
- Methods:
  - define operations on objects.

Object-Oriented Concepts

- Message:
  - calls to methods
- *message protocol* (*message interface*) :
  - the entire collection of methods of an object
- Message has two parts
  - method name
  - destination object.

Object-Oriented Concepts

- Inheritance:
  - can be *complicated* by *access controls* to encapsulated entities
  - class can hide entities
    - from its subclasses and clients
    - from its clients while allowing its subclasses to see them
  - class can modify an inherited method
    - The new one *overrides* the inherited one
    - The method in the parent is *overridden.*

Object-Oriented Concepts

- a subclass can differ from its parent class:
  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

- Variables in a class:
  - *Class variables* – one per class
  - *Instance variables* – one per object
- 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 inter-dependencies among classes that complicate maintenance.

Dynamic Binding

- A *polymorphic variable* defined in a class can refer to:
  - *objects* of the class, OR
  - *objects* of any of its descendants (child classes)
- When an override method is called through a polymorphic variable, the binding to the correct method will be *dynamic.*

Dynamic Binding: C++ example

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

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;     // polymorphic variable

rect->draw();         // Statically bound to
                      // draw in Rectangle

ptr_shape = sq;       // points to a Square
ptr_shape ->draw();   // Dynamically bound
                      // to draw in Square.
```

Dynamic Binding Concepts

- *Abstract Method*:
  - does not include a definition (it only defines a protocol)
  - E.g., Java: public abstract class CSCI {

```
// declare fields here
// declare non-abstract methods here
abstract void draw(); }
```

- *Abstract Class*
  - includes at least one *abstract* method
  - cannot be instantiated;
  - a subclass of an abstract class implements the abstract method.
  - E.g., class CSCI6221 extends CSCI { …. } .

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 the subclass object?
  - If a derived class is-a parent class, then objects of the derived class must behave the same as the parent class object
- A subclass/derived class:
  - is a subtype if it has an is-a relationship with its parent class
  - can only add variables and methods and override inherited methods in "compatible" ways
- Subclasses inherit implementation; subtypes inherit interface and behavior.

Single vs. Multiple Inheritance

- Multiple inheritance:
  - allows a new class to inherit from two or more classes
- Advantage of multiple inheritance:
  - quite convenient and valuable
- 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).

Allocation and Deallocation of Objects

- From where are objects allocated?
  - If objects behave like the ADTs:
    - can be allocated from anywhere
    - allocated from the **run-time memory stack**
    - explicitly create on the **heap** (via **`new`**)
  - If objects are all heap-dynamic:
    - references can be thru a pointer/reference variable
    - dereferencing can be implicit
- deallocation of heap object explicitly
  - dangling pointers could be created.

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.

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

Support for OOP: Smalltalk

- 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
  - E.g., (anObject isKindOf: aClass)

"check if anObject is an instance of aClass or one of it subclasses".

```
Object subclass: #Account.
```

"create a new class *Account*".

Support for OOP: Smalltalk

- 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—*Object*, which has no superclass
  - The type checking in Smalltalk is dynamic
  - The type error occurs when a message is sent to an object that has no matching method.

Support for OOP: Smalltalk

- Evaluation of Smalltalk (1980)
  - 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: 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: C++

- Inheritance
  - A class need not be the subclass of any class
  - ***Access controls*** for class 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)

- Note: Subtyping is useful in supporting reuse externally, giving rise to a form of polymorphism. Once a data type is determined to be a subtype of another, any function that could be applied to elements of the **supertype** can also be applied to elements of the **subtype**. https://www.cs.princeton.edu/courses/archive/fall98/cs441/mainus/node12.html.

Support for OOP: C++

- The subclass 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* – inherited public and protected members are public and protected in the 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.
//     subclass_2 sub2; sub2.c = 10; //compiler error
//     private: default derivation.
```

Re-Exportation 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 {
        int t = base_class::c + 10;
        ...
}.
```

Re-Exportation in C++

- **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
  - e.g., *int c;* and *float z*; defined in base_class class.

Support for OOP: C++

- Multiple inheritance is supported
  - If there are two inherited members with the same name, they can both be referenced using the scope resolution operator (::)

```
e.g.,
class Thread { ... }
class Drawing { ... }
class DrawThread : public Thread, public Drawing
  { … }
// multiple inheritance from two parent classes.
```

Support for OOP: C++

- 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
  - *Pure virtual function*
    - has no definition at all
  - *Abstract class :*
    - has at least one pure virtual function

Support for OOP: C++

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

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;

rect->draw();          // Statically bound to
                       // draw in Rectangle

ptr_shape = sq;        // points to a Square
ptr_shape ->draw();    // Dynamically bound
                       // to draw in Square.
```

Support for OOP: C++

- 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: C++

- Evaluation
  - C++ provides extensive access controls (unlike Smalltalk)
  - C++ provides multiple inheritance
  - 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: 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 instance variables and methods
- Implementation section of a **class**
  - defines the methods
- Classes **cannot** be nested.

Support for OOP: Objective-C

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

```
@interface myNewClass: NSObject { … }

    …
@end
```

  - All subclasses are **subtypes** because all public members of a base class are also public in the derived class
  - 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 overridden method can be called through **`super`**
  - All inheritance is public (unlike C++).

Support for OOP: Objective-C

- Objective-C has two ways to extend a class
  - category
  - protocol.

Support for OOP: Objective-C

- *Category:* adds methods to *existing* classes; is a secondary interface of a class.

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

  - *Interface*: where to define attributes, methods of a **class**
  - A *mixin* is a class that contains a combination of methods from other classes. A *category*—`StackExtend`, is a *mixin*.
  - The implementation of a category is in a separate implementation: `@implementation Stack (StackExtend)`
  - Class interface declares the methods and properties associated with that class.

Support for OOP: Objective-C

- *Protocol* : a list of methods and properties

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

  - 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: Objective-C

- 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
  - E.g., NSInteger myFunc(id left, id right, void *context) { ...; }.

Support for OOP: Objective-C

- 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: 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, e.g., Integer, 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: Java

- Inheritance
  - Single inheritance supported only,
  - *abstract class* category provides some of the benefits of multiple inheritance (***interface***)
  - An interface can include only method declarations and named constants, e.g.,

```
e.g., public interface Comparable <T> {
          public int comparedTo (T b);
      }
```

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

Support for OOP: Java

- Dynamic Binding
  - all messages are dynamically bound to methods, unless the method is `final` (i.e., it cannot be overridden, 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: Java

- Nested Classes
  - hidden from all classes in their package, except for the nesting class
  - Non-static 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: Java

- Evaluation
  - Design decisions to support OOP are similar to C++
  - No support for procedural programming (e.g., Fortran, Pascal, C)
  - No parentless classes--every class has Object as a superclass (https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)
  - 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: 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: C#

- 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**. **E.g.,** public **new** void Method1() { … }
  - The parent class version can still be called explicitly with the prefix **base:**

    e.g., base. Draw()
  - Subclasses are subtypes if no members of the parent class is private
  - Single inheritance only.

Support for OOP: 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: C#

- 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: 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: Ruby

- General Characteristics
  - Everything is an object
  - 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
  - E.g., class Car

```
        attr_accessor :velocity
    end
    my_car = Car.new
    my_car.velocity = 65                    # setter
    my_velocity = my_car.velocity           # getter.
```

Support for OOP: Ruby

- 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.

Support for OOP: Ruby

```
class Foo
        def  self.inherited(subclass)
                puts "New subclass: #{subclass}"
        end
end

class Bar < Foo
end

class Baz < Bar
end

## result
New subclass: Bar
New subclass: Baz
```

ref: http://ruby-doc.org/core-2.0.0/Class.html.