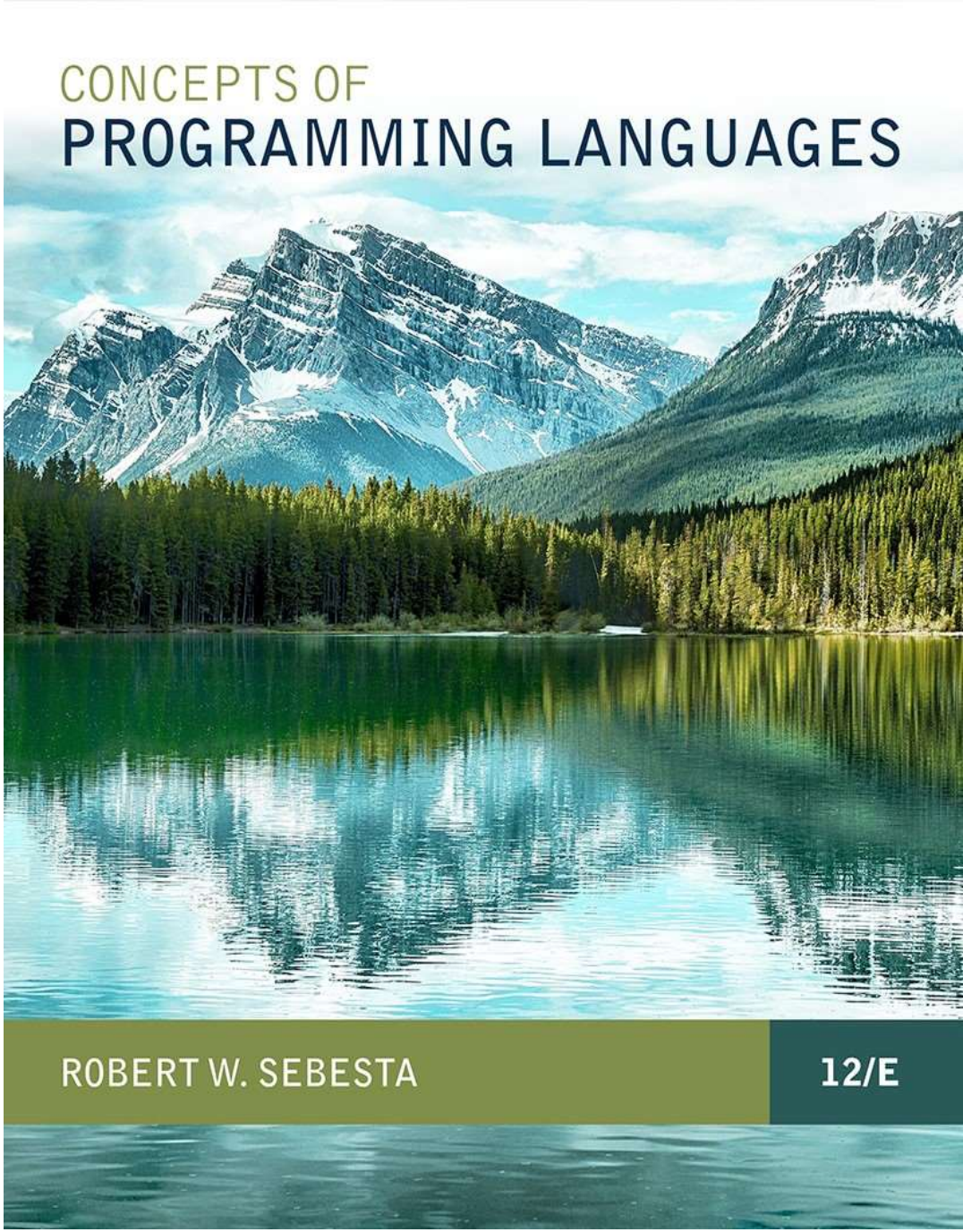
Lecture 9 (Chapter 12)

Support for Object-Oriented Programming



ISBN 0-321-49362-1

Chapter 12 Topics

- Introduction to OOP
- OOP Concepts
  - ADT
  - Inheritance
  - Polymorphism (Dynamic Binding)
- Design Issues for Object-Oriented Languages
- OO vs. PP (supplement)
- Language support for OOP
  - Smalltalk, C++, Java, C#
- Implementation of OO Constructs

Introduction

- Many object-oriented programming (OOP) languages
  - Some support procedural and data-oriented programming (e.g., C++)
  - Newer languages designed with "objects first" concept but still use imperative structures (e.g., Java and C#)
  - Some are pure OOP language (e.g., Smalltalk & Ruby)
  - Multi-paradigm languages are popular now
    - Some OOP languages support functional features while some functional languages also support OOP features
- Three major OOP features:
  - Abstract data types (Lecture 8)
  - Inheritance
    - Inheritance is the central theme in OOP
  - 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: ADT

- 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*
  - 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, e.g. myStk.pop()

Inheritance: Terminologies

A inherits from B;

A derives from B;

A is a subclass of (superclass) B;

A is the child class of (parent class)

C++/C#:	class A : B
Java:	class A extends B { … }
Python:	class A (B) :
Ruby:	class A < B

Object-Oriented Concepts: Inheritance

- Inheritance can be complicated by access controls to encapsulated entities
  - A class can hide entities from its subclasses, e.g. private
  - A class can hide entities from its clients, e.g. private/protected
  - A class can also hide entities for its clients while allowing its subclasses to see them, e.g. protected
- 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 in addition to inherited ones
  2. The subclass can modify the behavior of inherited methods.
  3. The parent class can define some of its variables or methods to have private access, not be visible in the subclass

Access Control in class entities

- C++: private, public, protected (default: private)

```
class Rectangle {
        int width, height;
public:
        void set_values (int,int);
        int area (void);
};
```

- Python: private, public, protected (default: public)

```
class Rectangle():
        def __init__(self, l, w): self.length = l self.width = w
        def rectangle_area(self):
                return self.length*self.width
```

Access Control in Inheritance Base

- C++ complicated access control

  Base of inheritance: private (by default), public, protected

```
class A : B { … }              //private base
class A : public B { . .. }    //public base
class A : protected B { … }    //protected base
```

Base of Inheritance: Example

```
class B {
   private:
          … x…;
          …f() …;
   protected:
          … y …;
          … g() …;
   public:
          … z …;
          …h() …;
};
```

//View from class A

class A : B { … }

x, f() not direct access

y, g(): private

z, h(): private

class A : protected B { … }

x, f() not direct access

y, g(): protected

z, h(): protected

class A : public B { … }

x, f() not direct access

y, g(): protected

z, h(): public

Java: Access control in Inheritance

- Unlike C++, Java doesn't provide an inheritance specifier like public, protected or private.
- Therefore, if some data member is public or protected in base class then it remains public or protected in derived class.

Inheritance: Overriding

- 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 in addition to inherited ones
  2. The subclass can modify the behavior of inherited methods.
  3. The parent class can define some of its variables or methods to have private access, not be visible in the subclass

Overriding/Overridden Example

- Java

```
class Human {
   public void eat() {          //Overridden method
      System.out.println("Human is eating");
   }
}
class Boy extends Human {
   public void eat() {                    //Overriding method
      System.out.println("Boy is eating");
   }
}
```

Overriding vs. Overloading

```
class Human {
   public void eat() {          //Overridden method
      System.out.println("Human is eating");
   }
}
class Boy extends Human {
   public void eat() {                    //Overriding method
      System.out.println("Boy is eating");
   }
   public void eat (int i) {       //overloading method
            
        …
   }
}
```

```
class Human {
    public void eat() {              //Overridden method
       System.out.println("Human is eating");   }}

class Boy extends Human {
      public void eat() {              //Overriding method
         System.out.println("Boy is eating");   }
      public void eat (int i) {       //overloading method
          System.out.println("boy eating version 2");   }}

public class HelloWorld {
       public static void main(String[] args) {
            Human h = new Human();
            Human b = new Boy();
            Boy bb = new Boy();
             h.eat();
             b.eat();       #note: b.eat(4) Error, why??
             bb.eat();
             bb.eat(4);
}}
```

More Object-Oriented Concepts

- There are two kinds of variables in a class:
  - *Class variables* – shared by all instances of a class
  - *Instance variables* – belongs to specific instance/object
- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects
- Multiple Inheritance
  - Inherit from multiple parents
- One disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance

Inheritance: Other Features

- There are two kinds of variables in a class:
  - *Class variables* – shared by all instances of a class
  - *Instance variables* – belongs to specific instance/object
- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects

```
public class Person {
    private String name;
    static int count = 0;
    public Person (String myName) {
            name = myName;
            ++count; }
    public int getCount() {
            return count; }
}
```

```
Person p1 = new Person ("Alex");
Person p2 = new Person ("Betty");
//now print p1.getCount()
//and p2.getCount();
```

Polymorphism

- 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

- Polymorphism can be achieved via dynamic binding
  - When the overridden methods called through a polymorphic variable, dynamic binding will bind the method to the class of the object.
    - static binding will bind method to the class of the object reference
    - thus dynamic binding would associate to the correct method of the object

- Dynamic binding allows software systems to be more easily extended during both development and maintenance

Polymorphism Example

```
class Human { …eat() … }
class Boy extends Human { …eat()… }
class Girl extends Human { … }

Human h1 = new Human ();
Boy b1 = new Human () ;
Human h2 = new Girl ();
Girl g1 = new Girl();
Girl g2 = new Human();
Boy b2 = new Boy();
```

Polymorphism Example

```
Human h;                          //h: polymorphic variable
switch (choice) {
case 1: h = new Human (); break;
case 2: h= new Boy (); break;
case 3: h = new Girl (); break;
};

h.eat();
```

This example illustrates/answers:
(1) What is a polymorphic variable?
(2) Why we'd introduce polymorphism?

Dynamic and Static Binding

- Static binding
  - Bound to the type/class in variable declaration
  - Advantage: easy to type checking and fast in binding
- Dynamic binding
  - Bound to the type/class of the objects
  - Advantage: able to invoke specific methods for objects

```
Human h;
switch (choice) {
case 1: h = new Human (); break;
case 2: h= new Boy (); break;
case 3: h = new Girl (); break;
};
h.eat(); //which eat() method would it bound to?
```

Polymorphic Variables: Method Binding

```
class Human { …eat() … }
class Boy extends Human { …eat()… }
class Girl extends Human { … }

Human h;
switch (choice) {
case 1: h = new Human (); break;
case 2: h= new Boy (); break;
case 3: h = new Girl (); break;
};
h.eat()
```

Assume Java is used

if choice == 1 which eat method would h.eat() bound to?

A. eat() in Human
B. eat() in Boy
C. none/error

choice == 2?

choice == 3?

Virtual Function in C++: Example

```
class Human {
        void eat();
        virtual void play();
        virtual void study();
}

class Boy : public Human {
        void eat();
        void play();
}
```

```
Human *h = new Boy();
Boy *b = new Boy();

h->eat();
b->eat();

h->play();
b->play();

h->study();
b->study();
```

Abstract Class and Abstract Methods

- An *abstract method* is one that does not include a definition (it only defines a protocol)
  - Such methods are also called pure virtual methods
- An *abstract class* is one that includes at least one abstract (or pure virtual) method
- An abstract class cannot be instantiated
  - i.e. cannot create objects of an abstract class

Abstract methods: Example

- Java

```
abstract class Human {
        abstract void eat();
        void play();
        void study();
}

class Boy extends Human {
        …
}
Human h = new Boy(); //okay
Human h = new Human() ; //wrong
```

- C++

```
class Human  {
        virtual void eat() = 0;
        virtual void play();
        …
}
```

Q: Which design you think is better? Why?

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

- Object-oriented: still different ways handling objects
- Everything is an object
  - Advantage – elegance and purity
  - Disadvantage – slow operations on simple objects
- Add objects to a complete typing system
  - e.g. defining a class considered as defining a new data type
  - 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
    - e.g. a Cat (subclass) holds "is-a" relation to its parent Pet class s cats behave as pets.
- 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
    - E.g. Cat type could be a subtype of Pet but cat (sub)class can add additional variables/methods, override Pet's methods, …
- Subclasses inherit implementation; subtypes inherit interface and behavior
  - Details about subclasses/subtypes omitted

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?
  - Like ADTs, they can be allocated from anywhere
    - Allocated from the run-time stack or heap
  - If objects 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, with regard to subtypes – *object slicing* problem may occur
    - when an object of a subclass type is copied to an object of superclass type, the storage for superclass may not be able to hold all member variables defined in the subclass
    - e.g. a Cat class may add a "meow" variable in addition to those inherited from Pet
    - Those additional variables will be "sliced off"
- Is deallocation explicit or implicit?

Dynamic and Static Binding

- Should all binding of messages to methods be dynamic?
  - If none (i.e. all static binding), you lose the advantages of flexibility provided by dynamic binding
  - If all are dynamic binding, 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 for it to 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?

(Design issues are open questions, we will look at these issues via case studies, i.e. how the contemporary OO languages handle these issues)

OOP Features: Summary

- OO programming involves three fundamental concepts
  - ADTs
  - Inheritance
  - dynamic binding (i.e. Polymorphism)
- Major design issues
  - exclusivity of objects
  - subclasses and subtypes
  - single and multiple inheritance
  - dynamic binding
  - explicit and implicit de-allocation of objects
  - nested classes
  - …
- Supplement lecture: OO vs. PP (FYI)

Support for OOP in Smalltalk

- Smalltalk is (the first) pure OOP language
  - Everything is an object
    - All objects have local memory
    - All objected are allocated from the heap
    - All deallocation is implicit
  - All computation is through objects sending messages to objects
  - None of the appearances of imperative languages
  - Smalltalk classes cannot be nested in other classes
- 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.
- 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
    - type checking is dynamic
    - the only type error occurs when a message is sent to an object that has no matching method
  - Greatest impact: advancement of OOP

Smalltalk: Example

```
“defining a new class Stack”
Object subclass: #Stack
        instanceVariableNames: 'anArray top’

        classVariableNames: ’ ’
        poolDictionaries: ‘ ‘

“defining methods”
push: item
        top := top+1.
        anArray at: top put: item
pop | item |
        item := anArray at: top.
        top := top-1.
        ^ item
setsize: n
        anArray := Array new: n.
        top := 0.
```

Some client code to test the stack:

```
S := Stack new.
S setsize: 10.
S inspect.
S push: 'hi there’.
S push: 3.14159.
S pop
```

Support for OOP in C++

- General Characteristics:
  - Evolved from C and SIMULA 67
  - Among the most widely used OOP languages
  - constructors and destructors
  - Elaborate access controls to class entities
    - private (visible only in the class and friends)
    - public (visible in subclasses and clients)
    - protected (visible in the class and in subclasses, but not clients)
- Inheritance
  - A class need not be the subclass of any class
  - Access controls for class derivation (i.e inheritance base)
    - 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
    - protected derivation – public and protected members are 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 { … };       //public base inheritance
//      In this one, b and y are protected and
//      c and z are public

class subclass_2 : private base_class { … };    //private base inheritance
//     In this one, b, y, c, and z are private,
//     and no derived class has access to any
//     member of base_class
```

Scope Resolution 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;
          …
}
```

- 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

C++: Multiple Inheritance and Dynamic Binding

- 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 { … }
```

- 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*

C++ Dynamic and Static Binding: Example

```
class Shape {
 public:
   virtual void draw() = 0;
   ...        //virtual function indicates
              //dynamic binding
   void display()   //static binding
          { …}
};
class Circle : public Shape {
  public:
    void draw() { ... }
    void display() { …}
  ...
};
class Rectangle : public Shape {
  public:
    void draw() { ... }
  ...
};
```

```
Circle *circ = new Circle;
Rectangle* rect = new Rectangle;
Shape* ps;

ps= circ;  // points to a Circle
ps ->draw();  // dynamically bound to
                //draw()  in Circle
ps->display(); //statically bound to display()
                 //in Shape
circ->draw();  //statically bound to draw()
                 //in Circle
rect->draw();  // Statically bound to
                  // draw() in Rectangle
```

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

```
Circle circ;    //allocates on stack
Shape sp;
sp = circ;      //possible object slicing
sp.draw();  //bound to draw() in Circle
```

Recent development in OOP discourages objects allocated on stack.

OOP in C++: Evaluation

- C++ provides extensive access controls
  - Access control to members as well as to inheritance base
  - Too complicated
- 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
  - By default is static binding (static binding is faster!)
  - Dynamic binding if programmer declares a method as virtual
- C++ is much faster than Smalltalk
  - 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

Java: Inheritance

- Single inheritance supported only
- The benefits of multiple inheritance can be achieved via 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 overridden)

- All subclasses are subtypes
- The super keyword
  - a reference variable that is used to refer parent class objects.

Java: Dynamic Binding and Nested Classes

- 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)
  - Static binding is also used if the methods is static or private both of which disallow overriding
- Nested Classes
  - hidden from all classes in the 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

OOP in Java: Evaluation

- Design decisions to support OOP are similar to C++
- No support for procedural programming
- No parentless classes
  - The Object class is the root class in Java
  - It is the super class of every user-defined/predefined class
- Dynamic binding is used as "normal" way to bind method calls to method definitions
  - Dynamic binding by default
- 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
    - structs are less powerful stack-dynamic constructs (e.g., no inheritance)
- Inheritance
  - Syntax: similar to C++
  - A method in parent class can be overridden in the derived class by marking its definition with new
    - The parent class version can still be called with the prefix base:

      base.Draw()
  - Single inheritance only

Support for OOP in C# (continued)

- 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
- 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

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

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;
  - access to methdos via its declared (class) type
- 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
    - access to methods based on object via CIR
  - The storage structure is sometimes called *virtual method tables* (vtable)

DESIGN ISSUE/ LANGUAGE	SMALLTALK	C++	JAVA	C#	RUBY
Exclusivity of objects	All data are objects	Primitive types plus objects	Primitive types plus objects	Primitive types plus objects	All data are objects
Are subclasses subtypes?	They can be and usually are	They can be and usually are if the derivation is public	They can be and usually are	They can be and usually are	No subclasses are subtypes
Single and multiple inheritance	Single only	Both	Single only, but some effects with interfaces	Single only, but some effects with interfaces	Single only, but some effects with modules
Allocation and deallocation of objects	All objects are heap allocated; allocation is explicit and deallocation is implicit	Objects can be static, stack dynamic, or heap dynamic; allocation and deallocation are explicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit
Dynamic and static binding	All method bindings are dynamic	Method binding can be either	Method binding can be either	Method binding can be either	All method bindings are dynamic
Nested classes?	No	Yes	Yes	Yes	Yes
Initialization	Constructors must be explicitly called	Constructors are implicitly called	Constructors are implicitly called	Constructors are implicitly called	Constructors are implicitly called

OO Languages: Summary

- Smalltalk is a pure OOL
- C++ is a hybrid language supporting both procedural/imperative programming and OOP
- Java is not a hybrid language like C++; it supports only OOP
  - Though it uses imperative-based syntax
- C# is based on C++ and Java
- Implementing OOP involves some new data structures
- OOP now is often the most popular and pragmatic programming approach