

CONCEPTS OF  
PROGRAMMING LANGUAGES

# Chapter 12

## Support for Object-Oriented Programming



ROBERT W. SEBESTA

12/E

ISBN 0-321-49362-1

# Chapter 1 2 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#
- 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)

---

Java



```
public class Car {
    private int speed;

    public void accelerate(int amount) {
        speed += amount;
    }

    public int getSpeed() {
        return speed;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();

        // Sending a message to the myCar object
        myCar.accelerate(20);

        // Sending another message to the myCar object
        int currentSpeed = myCar.getSpeed();
        System.out.println("Current speed: " + currentSpeed);
    }
}
```



# 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 from its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method
  - The new one *overrides* the inherited one
  - The method in the parent is *overridden*

# Object-Oriented Concepts (continued)

---

```
C++  
  
#include <iostream>  
  
class Shape {  
public:  
    void printShape() {  
        std::cout << "This is a shape." << std::endl;  
    }  
};  
  
class Rectangle : public Shape {  
public:  
    void printRectangle() {  
        std::cout << "This is a rectangle." << std::endl;  
    }  
};  
  
int main() {  
    Rectangle rect;  
  
    // Accessing methods from both base and derived classes  
    rect.printShape(); // Output: This is a shape.  
    rect.printRectangle(); // Output: This is a rectangle.  
  
    return 0;  
}
```

# Object-Oriented Concepts (continued)

---

## 1. Public:

```
Java

public class MyClass {
    public static void myPublicMethod() {
        System.out.println("This is a public method.");
    }
}
```

- **Explanation:** The `public` modifier allows the `myPublicMethod()` method to be accessed from any other class.

## 2. Private:

```
Java

public class MyClass {
    private static void myPrivateMethod() {
        System.out.println("This is a private method.");
    }
}
```

- **Explanation:** The `private` modifier restricts the access of `myPrivateMethod()` to within the `MyClass` class only.

## 3. Protected:

```
Java

public class MyClass {
    protected static void myProtectedMethod() {
        System.out.println("This is a protected method.");
    }
}
```

- **Explanation:** The `protected` modifier allows access to `myProtectedMethod()` from within the same class, subclasses, and classes within the same package.

# Object-Oriented Concepts (continued)

## 4. Default (Package-Private):

```
Java

public class MyClass {
    static void myDefaultMethod() {
        System.out.println("This is a default (package-private) method.")
    }
}
```

- **Explanation:** The absence of an access modifier designates the `myDefaultMethod()` as package-private. It can be accessed by any class within the same package.

## 5. Static:

```
Java

public class MyClass {
    public static void myStaticMethod() {
        System.out.println("This is a static method.");
    }
}
```

- **Explanation:** The `static` modifier allows the method to be called on the class itself, without needing to create an object of the class.

## Example Usage:

```
Java

public class Main {
    public static void main(String[] args) {
        MyClass.myPublicMethod();
        // MyClass.myPrivateMethod(); // Error: Private method not acces
        // MyClass.myProtectedMethod(); // Error: Protected method not ac
        // MyClass.myDefaultMethod(); // Error: Default method not acces
        MyClass.myStaticMethod();
    }
}
```

# Object-Oriented Concepts (continued)

---

```
Java

public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void makeSound() {
        System.out.println("Generic animal sound");
    }
}

public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        System.out.println(dog.name); // Allowed, 'name' is protected in
        dog.makeSound(); // Outputs "Woof!"
    }
}
```

# Object-Oriented Concepts (continued)

---

## `public:`

The `Dog` class is declared `public`, meaning it can be accessed from any other class.

## `protected:`

The `name` field in the `Animal` class is declared `protected`, meaning it can be accessed by the `Dog` class (subclass) and any other classes in the same package.

## `@Override:`

The `makeSound` method in the `Dog` class is marked with `@Override`, indicating it is overriding the method in the `Animal` class.

## **No Modifier (Default Access):**

If no access modifier is specified, the class or member has package-private access, meaning it can be accessed only by classes in the same package.

## **Other Modifiers:**

- `final`: Prevents a class from being subclassed or a method from being overridden.
- `abstract`: Declares an abstract class that cannot be instantiated directly or an abstract method that must be implemented by subclasses.

# 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



# Object-Oriented Concepts (continued)

---

Instance variables are defined with (self)

Code

```
class Employee:

    company_name = "Acme Inc" # Class variable

    def __init__(self, name, salary):

        self.name = name # Instance variable

        self.salary = salary # Instance variable

emp1 = Employee("John", 50000)

emp2 = Employee("Jane", 60000)

print(emp1.company_name) # Outputs "Acme Inc" (same for all employees)

print(emp2.name) # Outputs "Jane" (specific to Jane's object)
```

# Object-Oriented Concepts (continued)

---

## Object methods

Java



```
public class Car {  
    private String color;  
  
    public void setColor(String color) {  
        this.color = color;  
    }  
  
    public String getColor() {  
        return this.color;  
    }  
}
```

*// Usage:*

```
Car myCar = new Car();  
myCar.setColor("Red");  
System.out.println(myCar.getColor());
```

# Object-Oriented Concepts (continued)

---

## Class methods - Defined as static in Java

Java



```
public class MathUtils {  
    public static int square(int num) {  
        return num * num;  
    }  
}
```

*// Usage:*

```
int result = MathUtils.square(5); // No object creation required
```

# 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 - Polymorphism example

---

All java classes are virtual classes

```
Java

class Animal {
    public void sound() {
        System.out.println("The animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("The dog barks");
    }
}

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("The cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();

        animal1.sound(); // Output: The dog barks
        animal2.sound(); // Output: The cat meows
    }
}
```

# Dynamic Binding - Polymorphism example

---

## All java classes are virtual classes except static

In Java, the statement that "all functions are virtual" reflects the language's approach to method overriding and dynamic method dispatch. Here's a breakdown of what this means:

### Key Points

1. **Method Overriding:** In Java, when a subclass provides a specific implementation of a method that is already defined in its superclass, this is known as method overriding. The overridden method in the subclass can be called through a reference of the superclass type.
2. **Dynamic Method Dispatch:** Java uses dynamic method dispatch to determine which method implementation to execute at runtime. This means that the method that gets called is based on the object being referred to, not the type of the reference. This behavior is a hallmark of polymorphism.
3. **No Explicit :** Unlike C++ where you have to explicitly declare a method as `virtual` to enable polymorphic behavior, in Java, all non-static, non-private methods are implicitly considered virtual. This means that they can be overridden by subclasses, and their behavior can be determined at runtime.
4. **Static and Private Methods:** It's important to note that static methods and private methods do not exhibit this behavior. Static methods are bound at compile time, and private methods cannot be overridden.

# Dynamic Binding

C++

```
// C++ Program to Demonstrate the Concept of Dynamic binding
// with the help of virtual function

#include <iostream>
using namespace std;

class GFG {
public:
    // function that call print

    void call_Function() { print(); }
    // the display function
    virtual void print()
    {
        cout << "Printing the Base class Content" << endl;
    }
};

// GFG2 inherited publicly
class GFG2 : public GFG {
public:
    void print() // GFG2's display
    {
        cout << "Printing the Derived class Content"
            << endl;
    }
};

int main()
{
    GFG geeksforgeeks; // Creating GFG's object
    geeksforgeeks.call_Function(); // Calling call_Function
    GFG2 geeksforgeeks2; // calling GFG2
    geeksforgeeks2.call_Function();
    return 0;
}
```

# 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 abstract method
- An abstract class cannot be instantiated



# Abstract class example

---

```
C++  
  
#include <iostream>  
  
class Shape {  
public:  
    virtual ~Shape() {} // Virtual destructor  
    virtual void draw() = 0; // Pure virtual function  
};  
  
class Circle : public Shape {  
public:  
    void draw() override {  
        std::cout << "Drawing a circle" << std::endl;  
    }  
};  
  
class Square : public Shape {  
public:  
    void draw() override {  
        std::cout << "Drawing a square" << std::endl;  
    }  
};  
  
int main() {  
    // Shape s; // Error: Cannot instantiate abstract class  
  
    Circle c;  
    c.draw();  
  
    Square s;  
    s.draw();  
  
    return 0;  
}
```

# 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 like the ADTs, they can be allocated from anywhere
    - Allocated from the run-time stack
    - Explicitly create on the heap (via `new`)
  - If they are all heap-dynamic, references can be uniform thru a pointer or reference variable
    - Simplifies assignment - dereferencing can be implicit
  - If objects are stack dynamic, there is a problem with regard to subtypes – *object slicing*
- Is deallocation explicit or implicit?

# Dynamic and Static Binding

---

- Should all binding of messages to methods be dynamic?
  - If none are, you lose the advantages of dynamic binding
  - If all are, it is inefficient
- 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 objects 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  
{ ... }
```

# Multiple inheritance example

```
C++  
  
#include <iostream>  
  
using namespace std;  
  
// Base class 1  
class Shape {  
public:  
    void draw() {  
        cout << "Drawing a shape" << endl;  
    }  
};  
  
// Base class 2  
class Color {  
public:  
    void setColor(string c) {  
        color = c;  
    }  
  
    void printColor() {  
        cout << "Color: " << color << endl;  
    }  
  
private:  
    string color;  
};  
  
// Derived class inheriting from both Shape and Color  
class ColoredCircle : public Shape, public Color {  
public:  
    void draw() {  
        cout << "Drawing a colored circle" << endl;  
    }  
};  
  
int main() {  
    ColoredCircle cc;  
  
    cc.draw();    // Calls the draw() method from ColoredCircle  
    cc.setColor("red");  
    cc.printColor();  
}
```

# 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*

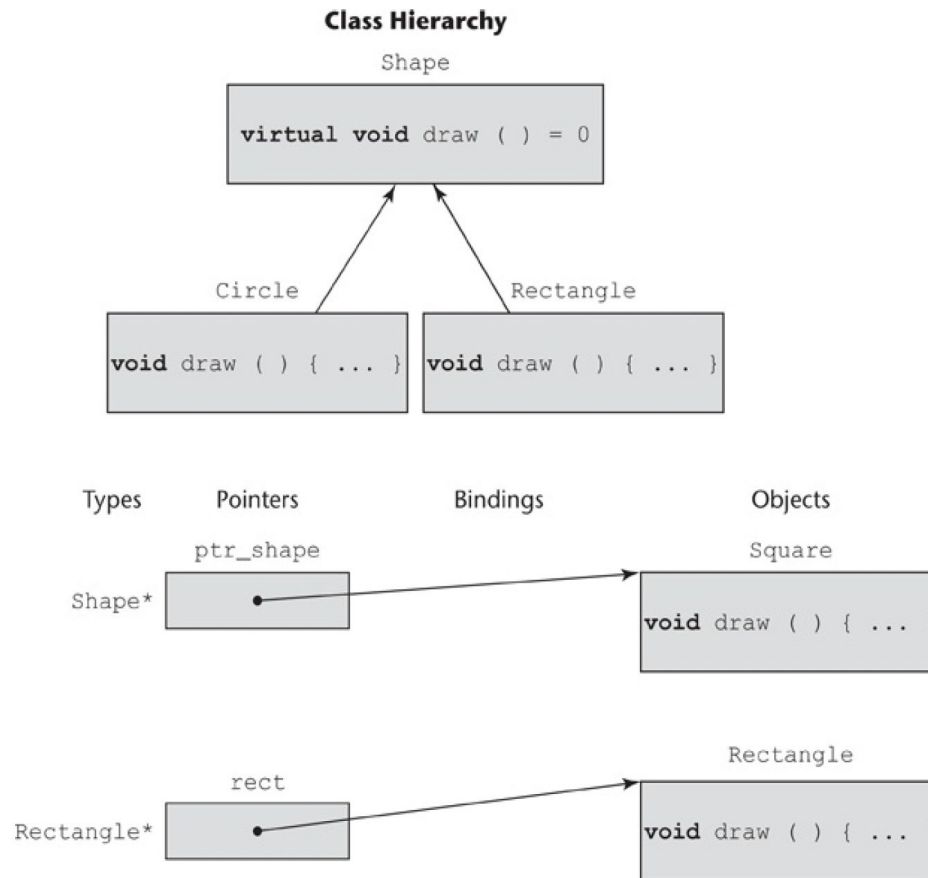
# Support for OOP in C++ (continued)

---

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

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

# Support for OOP in C++ (continued)



**Figure 12.6 Dynamic binding**



# Support for OOP in C++ (continued)

---

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

```
Circle circ;    // Allocates a Circle object from the stack
Rectangle rect; // Allocates a Rectangle object from the stack
rect = circ;    // Copies the data member values from Circle
                  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!

# 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 compareTo (T b);  
}
```

- Methods can be **final** (cannot be overridden)
- 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 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 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)

---

## 1. Static Nested Classes:

- Declared with the `static` keyword.
- Similar to a regular class, but it is scoped within the enclosing class.
- Can only access static members of the enclosing class.
- To access a static nested class, you don't need an instance of the outer class.


Java

```
class OuterClass {
    static class StaticNestedClass {
        void printMessage() {
            System.out.println("This is a static nested class.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
        nestedObject.printMessage();
    }
}
```

# Support for OOP in Java (continued)

## 2. Inner Classes (Non-static Nested Classes):

- Without the `static` keyword.
- Has access to all members (including private) of the enclosing class. 
- Requires an instance of the enclosing class to be created.
- Has an implicit reference to the enclosing class instance.

Java

```
class OuterClass {
    int x = 10;

    class InnerClass {
        void printMessage() {
            System.out.println("This is an inner class, x = " + x);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new InnerClass();
        innerObject.printMessage();
    }
}
```



# 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

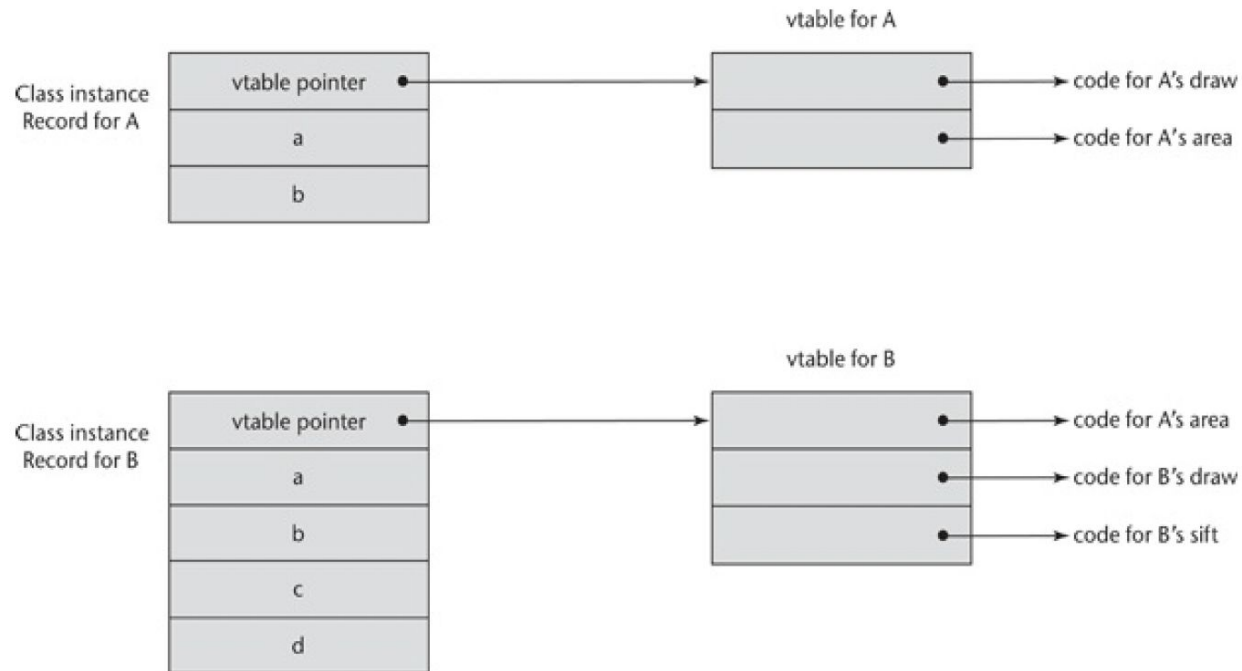
# Instance Data Storage

---

```
public class A {  
    public int a, b;  
    public void draw() { . . . }  
    public int area() { . . . }  
}  
public class B extends A {  
    public int c, d;  
    public void draw() { . . . }  
    public void sift() { . . . }  
}
```

# Instance Data Storage

---



**Figure 12.7 An example of the CIRs with single inheritance**

# 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

# Dynamic Binding of Methods Calls

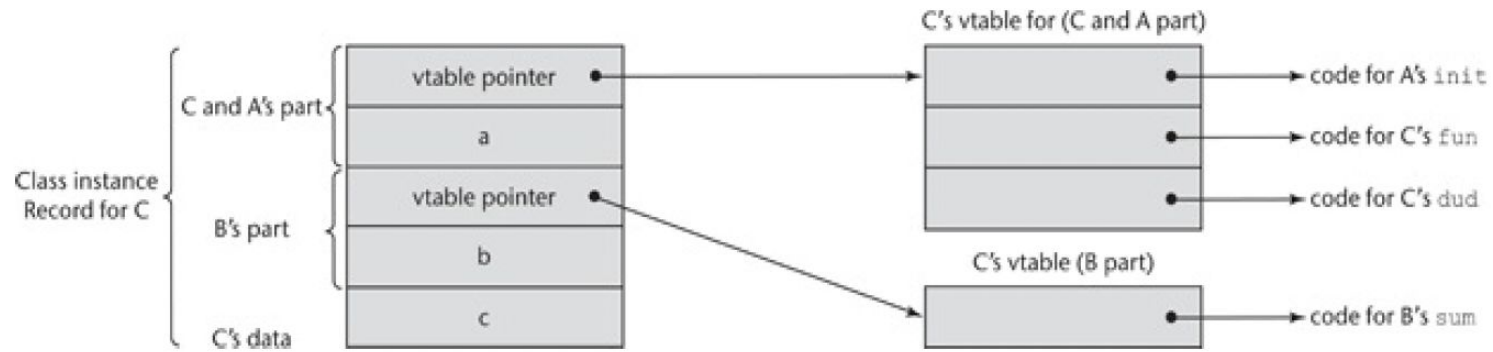
---

Virtual solves the diamond inheritance issue in C++

```
class A {
public:
    int a;
    virtual void fun() { . . . }
    virtual void init() { . . . }
};
class B {
public:
    int b;
    virtual void sum() { . . . }
};
class C : public A, public B {
public:
    int c;
    virtual void fun() { . . . }

    virtual void dud() { . . . }
};
```

# Dynamic Binding of Methods Calls



**Figure 12.8 An example of a subclass CIR with multiple parents**

# 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 Java (continued)

---

Java



```
import java.lang.reflect.*;

public class ReflectionExample {
    public static void main(String[] args) {
        try {
            Class<?> clazz = Class.forName("java.lang.String");

            // Get the name of the class
            System.out.println("Class Name: " + clazz.getName());

            // Get the methods of the class
            Method[] methods = clazz.getDeclaredMethods();
            for (Method method : methods) {
                System.out.println("Method: " + method.getName());
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

# method.invoke() example

---

```
Java

import java.lang.reflect.Method;

public class ReflectionExample {

    public static void main(String[] args) {
        try {
            // Get the Class object for the MyClass class
            Class<?> myClass = MyClass.class;

            // Get the method named "myMethod" with a single String parameter
            Method myMethod = myClass.getMethod("myMethod", String.class);

            // Create an instance of MyClass
            MyClass obj = new MyClass();

            // Invoke the method using reflection
            Object result = myMethod.invoke(obj, "Hello, Reflection!");

            // Print the result
            System.out.println(result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

class MyClass {
    public String myMethod(String message) {
        return "Message: " + message;
    }
}
```

# 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