National University
Of Computer and Emerging Sciences

Theory of Programming Languages

Object Oriented Programming Paradigm

Sajid Anwer

Department of Computer Science,
FAST-NUCES, CFD Campus

National University
Of Computer and Emerging Sciences

Chapter Outline

- Object-Oriented Programming
- Design Issues for OO Languages
- Support for OOP
  - » Smalltalk
  - » C++
  - » Java
  - » C#
  - » Ada 95
  - » Ruby
- Implementation of OO Constructs

Why OOP?

- Pressure on *software productivity* vs. continual reduction in hardware cost
- Productivity increases can come from *reuse*
- Abstract data types (ADTs) for *reuse?*
- Mere ADTs are not enough
  - » ADTs are difficult to reuse
    - Always *need changes* for new uses, e.g., circle, square, rectangle, ...
  - » All ADTs are *independent* and at the same level
    - hard to organize program to match problem space

- More, in addition to ADTs, are needed

What OOP?

- An object-oriented language must provide supports for three key features:
  - » Abstract data types
  - » Inheritance: the central theme in OOP and languages that support it
  - » Polymorphism and dynamic binding

National University
Of Computer and Emerging Sciences

What are Needed?

- Given a collection of *related* ADTs, need to factor out their *commonality* and put it in a new type; *abstraction.*
  - The collection of ADTs then *inherit* from the new type and add their own

```
class Shape {
  private:
    int x; int y;
  public:
    Shape(int a, int b);
    ...
    virtual void draw();
};
```

```
class Circle: public Shape {
  private:
    int radius;
  public:
    Circle(int x1, y2, r1);
    ...
    void draw();
};
```

What are Needed?

- For classes in an inheritance relationship, a method call may need to be bound to a *specific object* of one of the classes *at run time*

```
Shape *shape_list[3]; // array of shape objects
shape_list[0] = new Circle;
shape_list[1] = new Square;
shape_list[2] = new Triangle;
for(int i = 0; i < 3; i++){
  shape_list[i].draw();
}
```

- Polymorphism and *dynamic binding*

OOP Constructs

- ADTs are usually called *classes*, e.g. **`Shape`**
- Class instances are called *objects*, e.g. **`shape_list[0]`**
- A class that inherits is a *derived class* or a *subclass*, e.g. **`Circle, Square`**
- The class from which another class inherits is a *parent class* or *superclass*, e.g. **`Shape`**
- Subprograms that define operations on objects are called *methods*, e.g. **`draw()`**

OOP Constructs

- Calls to methods are called *messages*, e.g. **`Shape_list[i].draw();`**
- Object that made the method call is the *client*
- 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

Inheritance

- Allows new classes defined in terms of *existing ones*, i.e., by inheriting common parts
- Can be complicated by access controls to encapsulated entities
  - » A class can hide entities from its subclasses (private)
  - » A class can hide entities from its clients
  - » A class can also hide entities for its clients while allowing its subclasses to see them
- A class can modify an inherited method
  - » The new one overrides the inherited one
  - » The method in the parent is overridden

Inheritance

- There are two kinds of variables in a class:
  - » *Class variables* - one/class
  - » *Instance variables* - one/object, object state
- 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

```
Shape *shape_list[3]; // array of shapes
shape_list[0] = new Circle;
shape_list[1] = new Square;
shape_list[2] = new Triangle;
for(int i = 0; i < 3; i++){
  shape_list[i].draw();
}
```

Dynamic Binding

- 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, e.g.,

```
shape_list[i].draw();
```

- Allows software systems to be more easily *extended* during both development and maintenance, e.g., new shape classes are defined later

Design Issues for OOP Languages

- The exclusivity of objects
- Are subclasses subtypes?
- Type checking and polymorphism
- 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
- Alternatives: Option 1: Add objects to existing typing system
  - Advantage: fast operations on simple objects
  - Disadvantage: *confusing typing* (2 kinds of entities)
- Option 2: Imperative-style typing system for primitives and everything else objects
  - Advantage: fast operations on simple objects and a relatively small typing system
  - Disadvantage: still confusing by *two type systems*

Are Subclasses Subtypes?

- Does an "*is-a*" relationship hold between a *parent class object and an object of subclass*?

  `subtype small_Int is Integer range 0 .. 100;`

  - » Every **`small_Int`** variable can be used anywhere **`Integer`** variables can be used

- A derived class is a *subtype* if methods of subclass that override parent class are *type compatible* with the overridden parent methods
  - » A call to overriding method can replace any call to overridden method *without type errors*

National University
Of Computer and Emerging Sciences

Type Checking and Polymorphism

- Polymorphism may require *dynamic type checking* of parameters and the return value
  - » Dynamic type checking is costly and delays error detection
- If overriding methods are restricted to having the same parameter types and return type, the checking can be static

Single and Multiple Inheritance

- Multiple inheritance allows a *new class to inherit from two or more classes*
- Disadvantages of multiple inheritance:
  - » Language and implementation complexity: if class C needs to reference both draw() methods in parents A and B, how to do? If A and B in turn inherit from Z, which version of Z entry in A or B should be ref.?
  - » Potential inefficiency: dynamic binding costs more with multiple inheritance (but not much)
- Advantage:
  - » Sometimes it is quite convenient and valuable

Object Allocation and Deallocation

- From where are objects allocated?
  - » If behave like ADTs, can be allocated *from anywhere*
    - Allocated from the run-time stack
    - Explicitly created 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, assignment of subclass B's object to superclass A's object is value copy, but what if B is larger in space?
    - 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, why?*
- Alternative: allow the user to specify

Nested Classes

- If a new class is needed by *only one class*, no reason to define 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?

Inheritance in C++

- Inheritance
  - » A class need not be the *subclass of any class*
  - » Access controls for members are
    - Private: accessible only from within other members of the same class or from their friends.
    - Protected: accessible from members of the same class and from their friends, but also from members of their derived classes.
    - Public: accessible from anywhere that object is visible
- Multiple inheritance is supported
  - » If two inherited members with same name, they can both be referenced using scope resolution operator

Inheritance in C++

- 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*
    - members in derived class cannot access to any member of the parent class
  - » Public derivation: public and protected members are also public and protected in subclasses

Inheritance in C++

```
class base_class {
  private:
     int a;      float x;
  protected:
     int b;      float y;
  public:
     int c;      float z; };
class subclass_1 : public base_class {…};
// b, y protected; c, z public
class subclass_2 : private base_class {…};
// b, y, c, z private; no access by
  derived
```

Inheritance in C++

- A member that is not accessible in a subclass (because of private derivation) can be visible there using *scope resolution operator (::),* e.g.,

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

- One motivation for using private derivation
  - » A derived class adds some new members, but does not want its clients to see members of the parent class, even though they had to be public in the parent class definition

Dynamic Binding – Abstract Class

- A method can be defined to be **`virtual`**, and 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*

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

```
class Circle: public Shape {
   public:
     ...
    void draw() {...};
};
```

Support for OOP in C++

- Evaluation
  - » C++ provides extensive access controls (unlike Smalltalk)
  - » C++ provides multiple inheritance
  - » Programmer must decide at design time which methods will be statically or dynamically bound
    - Static binding is *faster*, why?
  - » Smalltalk type checking is dynamic (flexible, but somewhat unsafe and is ~10 times slower due to interpretation and dynamic binding

Support for OOP in Java

- Focus on the differences from C++

- General characteristics
  - All data are objects except the *primitive types*
  - All primitive types have *wrapper* classes to store data value, e.g., **`myArray.add(new Integer(10));`**
  - All classes are descendant of the root class, **`Object`**
  - All objects are heap-dynamic, are referenced through reference variables, and most are allocated with **`new`**
  - No destructor, but a *finalize* method is implicitly called when garbage collector is about to reclaim the storage occupied by the object, e.g., to clean locks

Support for OOP in Java

- Single inheritance only, but *interface* provides some flavor of multiple inheritance

- An *interface* like a class, but can include only method declarations and named constants, e.g.,

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

- A class can inherit from another class and “implement” an interface for multiple inheritance

- A method can have an interface as formal parameter, that accepts any class that implements the interface
  → a kind of polymorphism

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

Support for OOP in Java – Nested Classes

- Several varieties of 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, but not a static nested 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 – Evaluation

- Design decisions to support OOP are similar to C++
- No support for procedural programming
- 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#

- C# use same syntax as C++ to represent inheritance, however, did not provide *multiple* inheritance.

- Dynamic Binding

```
public class Shape {
  public virtual void Draw() { . . . }
  . . .
}
public class Circle : Shape {
  public override void Draw() { . . . }
  . . .
}
public class Rectangle : Shape {
  public override void Draw() { . . . }
  . . .
}
```

Support for OOP in C# -- Evaluation

- The differences between C#’s support for object-oriented programming and that of Java are *relatively minor.*

- The availability of *structs* in C#, which Java does not have, can be considered an *improvement*.

Implementing OO Constructs

- Most OO constructs can be implemented easily by compilers
  - » Abstract data types
    - scope rules of primitive data types
  - » Inheritance
- Two interesting and challenging parts:
  - » Storage structures for *instance variables*
  - » Dynamic binding of messages to methods

Implementing OO Constructs

- *Class instance records* (CIRs) store the state of an object
  - *Static* (built at compile time and used as a template for the creation of data of class instances)
  - Every class has its own CIR
- CRI for the *subclass* is a copy of that of the parent class, with entries for the new instance variables added at the end
- Because CIR is static, access to all instance variables is done by constant offsets from beginning of CIR

Implementing OO Constructs

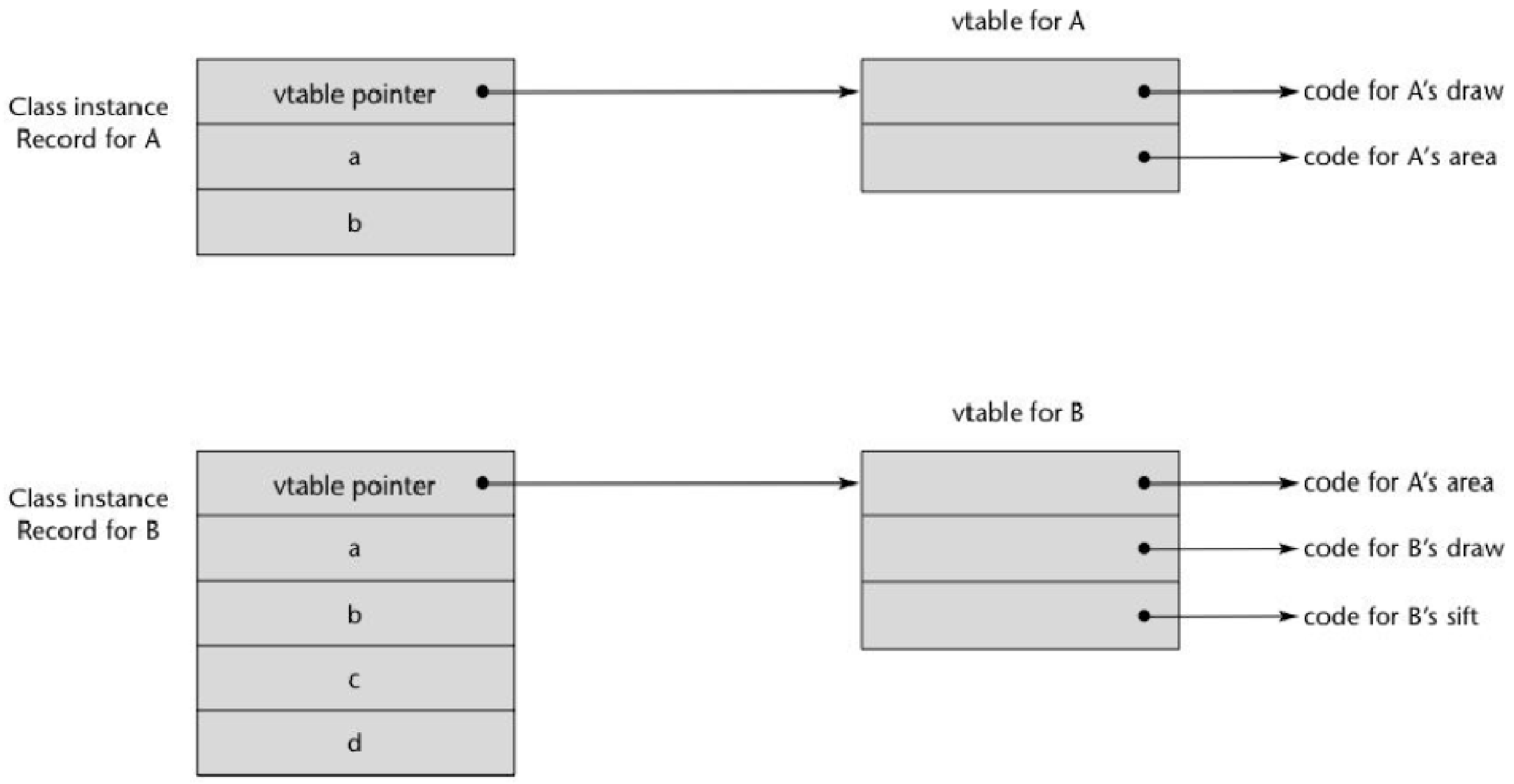
- Methods in a class that are statically bound need not be involved in CIR; methods that are dynamically bound must have entries in the CIR
  - » Calls to dynamically bound methods can be connected to the *corresponding code* thru a pointer in the CIR
  - » Storage structure for the list of dynamically bound methods is called *virtual method tables* (vtable)
  - » Method calls can be represented as *offsets from the beginning* of the vtable

Implementing OO Constructs

```
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 int sift() {...}
}
```

Implementing OO Constructs – Example CIR



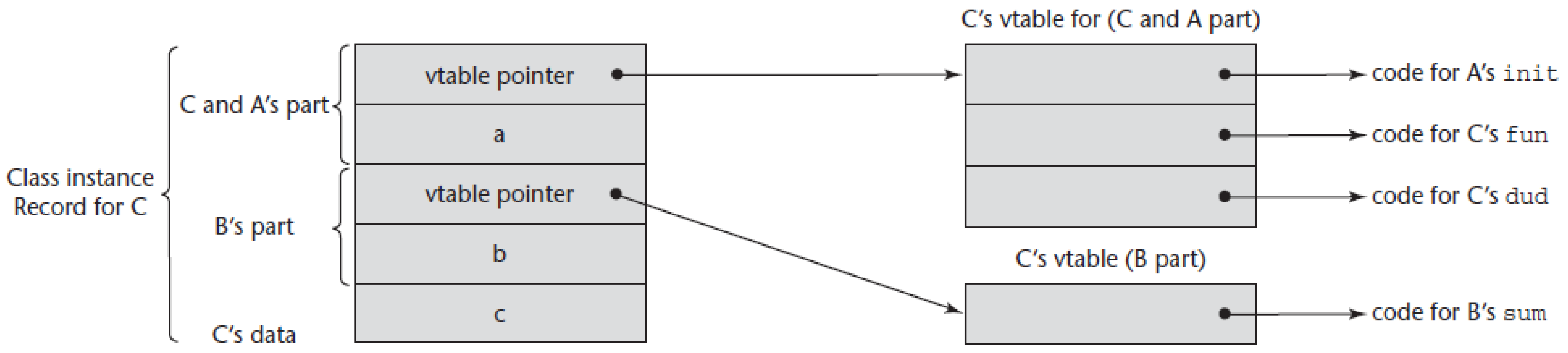
Implementing OO Constructs – Multiple Inheritance CIR

```
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() { . . . }
  };
```

National University
Of Computer and Emerging Sciences

Implementing OO Constructs – Multiple Inheritance CIR


Class instance Record for C
C and A's part
B's part
C's data
vtable pointer
a
vtable pointer
b
c
C's vtable for (C and A part)
code for A's init
code for C's fun
code for C's dud
C's vtable (B part)
code for B's sum