Yet Another OOP

Weng Kai

Python和JS的OOP

8.2类和对象的创建和使用

- Python使用class关键字来定义类，class关键字之后是一个空格，然后是类的名字，再然后是一个冒号，最后换行并定义类的内部实现。Python定义类方法：

```
class Classname :
    initializer
    methods
```

Student类和对象（1）

```
class Student:      #学生类：包含成员变量和成员方法
        def __init__ (self,mname,mnumber):  #构造方法
                self.name = mname          #成员变量
                self.number = mnumber
                self.Course_Grade = {}
                self.GPA = 0
        def getInfo(self):       #成员方法
            print(self.name,self.number)
s1 = Student("wang","317000010")   #创建s1对象
s1.getInfo()
s2=Student("zhang","317000011")            #创建s2对象
s2.getInfo()
```

Student类和对象（2）

- 上述程序定义Student类:
  - 该类包括四个成员变量：name，number，Course_Grade，GPA；
  - 两个成员方法：__init__，getInfo。
  - 实例化了s1与s2两个对象。

__init__方法

- __init__方法是Python类中的一种特殊方法，方法名的开始和结束都是双下划线，该方法称为构造方法，当创建类的对象时，它被自动调用。
- __init__方法中可以声明类所拥有的成员变量，并可为其赋初始值。该方法有一个特点，不能有返回值，因为它是用来构造对象的，调用后实例化了一个该类型的对象。

Self参数

- 类的所有方法都必须至少有一个名为self的参数，并且必须是方法的第一个形参（如果有多个形参的话），self参数代表将来要创建的对象本身。
- 在类的方法中访问实例变量（数据成员）时需要以self为前缀。
- 在外部通过对象调用对象方法时并不需要传递这个参数，如果在外部通过类调用对象方法则需要显式为self参数传值。

创建对象

- 定义了类之后，可以用来实例化对象，并通过“对象名.成员”的方式来访问其中的数据成员或成员方法。

```
s1 = Student(“ji","100001")
s1.getInfo()
```

- 在Python中，可以使用内置方法isinstance()来测试一个对象是否为某个类的实例。

```
>>> isinstance(s1, Student)
True
>>> isinstance(s1, str)
False
```

8.4 封装

- **封装**：
  - 将数据和对数据的操作组合起来构成类，类是一个不可分割的独立单位
  - 类中既要提供与外部联系的接口，同时又要尽可能隐藏类的实现细节。
- Python类中成员分为数据（变量、属性）成员和方法（函数）成员。数据成员有两类：类数据成员（类变量）和实例数据成员（实例变量）；方法成员根据访问特性的不同有：实例方法、类方法、静态方法等

类和对象

- Python类中成员：
  - 数据成员（变量、属性）
    - 类数据成员
    - 实例数据成员
  - 方法（函数）
    - 公有方法
    - 私有方法： 方法名以两个下划线’_’前缀
    - 类方法： @classmethod
    - 静态方法：@staticmethod

类成员变量与实例成员变量

- **实例的成员变量**一般是指在构造方法__init__()中定义的，定义和使用时必须以self作为前缀
- **类的成员变量**是在类中所有方法之外定义的

- 特性：在主程序中（或类的外部），实例变量属于实例(对象)，只能通过对象名访问；而类变量属于类，可以通过类名或对象名都可以访问。

类成员与实例成员

```
class Car:
    price = 100000              #定义类成员变量
    def __init__(self, c):
        self.color = c          #定义实例成员变量

car1 = Car("Red")               #实例化对象
car2 = Car("Blue")
print(car1.color, Car.price)    #查看实例变量和类变量的值
Car.price = 110000              #修改类变量
Car.name = 'QQ'                 #动态增加类变量
car1.color = "Yellow"           #修改实例变量
print(car2.color, Car.price, Car.name)
print(car1.color, Car.price, Car.name)
```

私有成员与公有成员

- 在Python中，以下划线开头的方法名和变量名有特殊的含义，尤其是在类的定义中。
  - _xxx：受保护成员，不能用'from module import *'导入；
  - __xxx__：系统定义的特殊成员；
  - __xxx：私有成员，只有类内自己能访问，不能使用对象直接访问到这个成员。

成员的输入显示

- 在IDLE环境中，在对象或类名后面加上一个圆点“.”，稍等则会自动列出其所有公开成员，模块也具有同样的用法。
- 如果在圆点“.”后面再加一个**下划线**“_”，则会列出该对象、类或模块的所有成员，包括私有成员。

JavaScript

对象

- 对象是JavaScript的一种复合数据类型，它可以把多个数据集中在一个变量中，并且给其中的每个数据起名字
- 或者说，对象是一个属性集合，每个属性有自己的名字和值
    - JavaScript并不像其他OOP语言那样有类的概念，不是先设计类再制造对象

创建对象

- var o = new Object();
- var ciclr = {x:0, y:0, radius: 2 };

访问对象属性

- .运算符

  var book = new Object();

  book.title = “HTML5秘籍”;

  book.translator = “李松峰”;

  book.chapter1 = new Object();

  book.chapter1.title = “HTML5简介”

- 即使构造的时候不存在的属性也可以在今后随时增加

删除对象属性

- delete book.chapter1;
- book.chapter1 = null;

遍历所有属性

- for (var x in o) ...

构造方法

- 构造函数

```
function Rect(w,h) {
    this.width = w; this.height = h;
    this.area = function(){ return this.width * this.height;
    };
}
var r = new Rect(5,10); alert(r.area());
```

1.不直接制造对象
2.通过this来定义成员
3.没有return

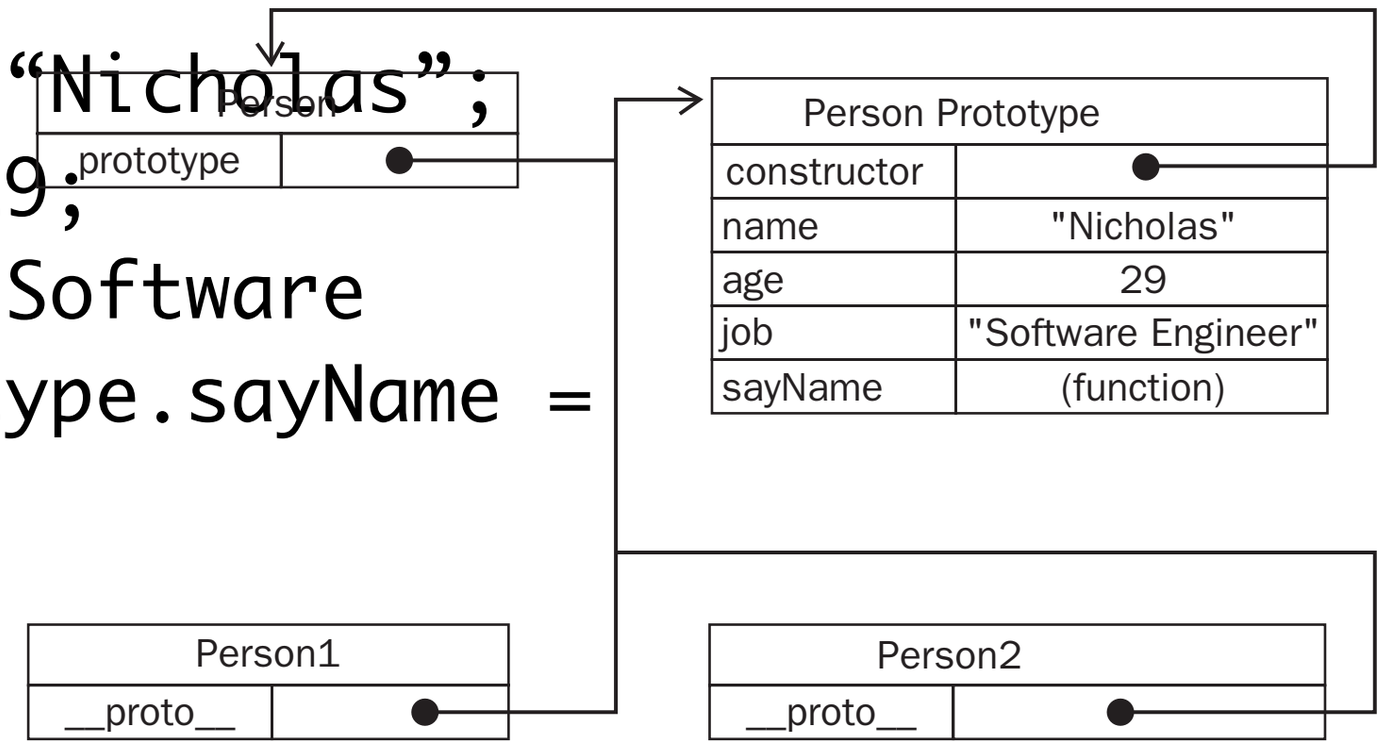
- 一旦定义了构造函数就可以构造任意数量的对象

原型对象

- 对象的prototype属性指定了它的原型对象，可以用.运算符直接读它的原型对象的属性

- 当写这个属性时才在它自己内部产生实际的属性

```
function Person(){ }
Person.prototype.name = “Nicholas”;
Person.prototype.age = 29;
Person.prototype.job = “Software
Engineer”; Person.prototype.sayName =
function(){
    alert(this.name);
};
var person1 = new Person();
person1.sayName(); //”Nicholas”
var person2 = new Person();
person2.sayName(); //”Nicholas”
alert(person1.sayName == person2.sayName);
//true
```



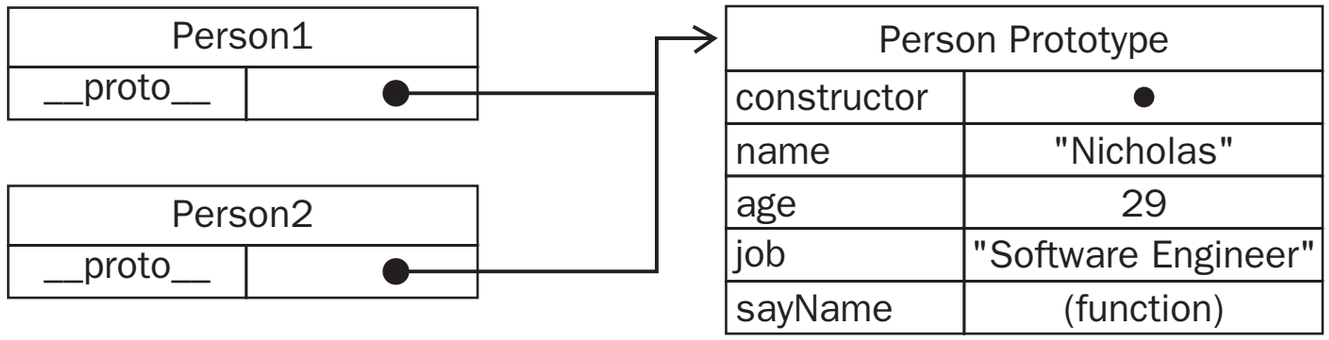
例子来源：Professional JavaScript for Web Developers

```
function Person(){ }
Person.prototype.name = “Nicholas”;
Person.prototype.age = 29; Person.prototype.job =
“Software Engineer”; Person.prototype.sayName =
function(){
    alert(this.name);
};
var person1 = new Person();
var person2 = new Person();
person1.name = “Greg”;
alert(person1.name); //”Greg” - from instance
alert(person2.name); //”Nicholas” - from prototype
```

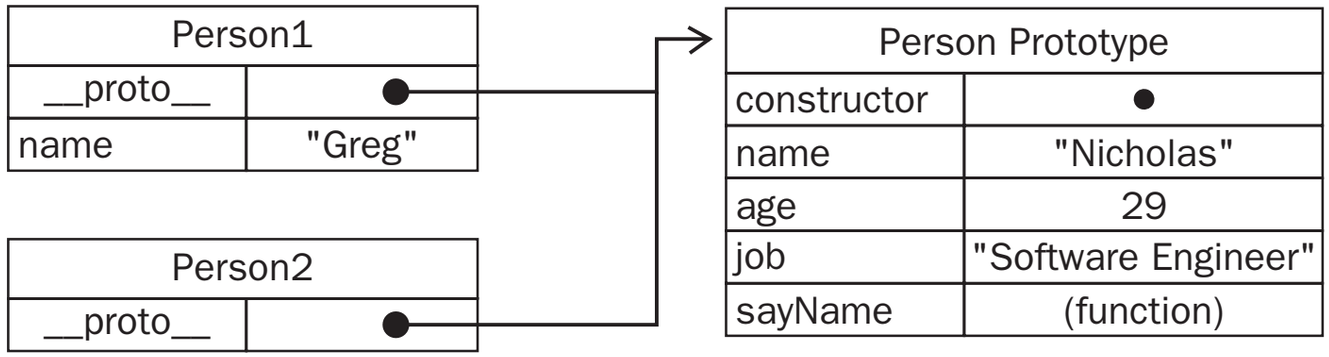
实际赋值的时候会制造相应的属性的

例子来源：Professional JavaScript for Web Developers

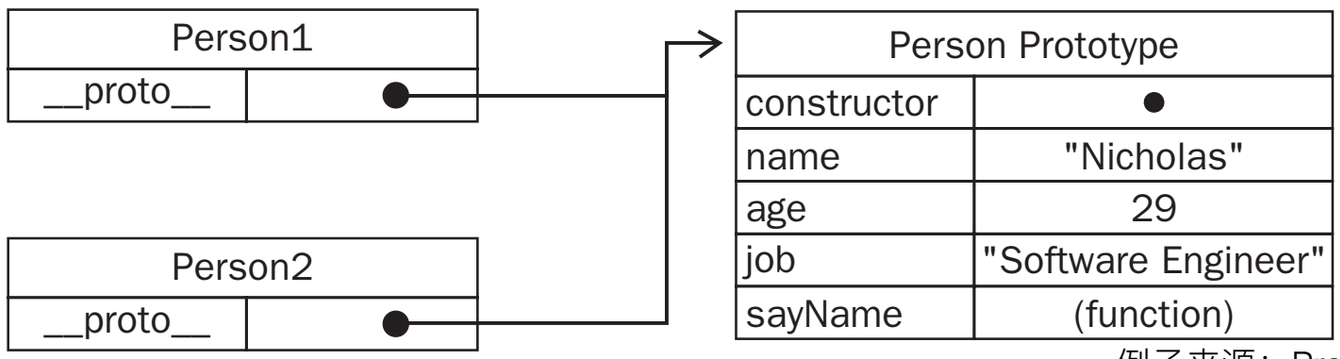
Initially



person1.name = "Greg"



delete person1.name



例子来源：Professional JavaScript for Web Developers

原型的问题

```
function Person(){ }
Person.prototype = {
    constructor: Person,
    name : “Nicholas”,
    age : 29,
    job : “Software Engineer”,
    friends : [“Shelby”, “Court”],
    sayName : function () {
        alert(this.name);
    }
};
var person1 = new Person();
var person2 = new Person();
person1.friends.push(“Van”);
alert(person1.friends); //”Shelby,Court,Van”
alert(person2.friends); //”Shelby,Court,Van”
alert(person1.friends === person2.friends); //
true
```

这里只是改数组里面的内容，没有换数组

例子来源：Professional JavaScript for Web Developers

组合原型和构造方法

```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.friends = [“Shelby”, “Court”];
}
Person.prototype = {
    constructor: Person,
    sayName : function () {
        alert(this.name);
    }
};
var person1 = new Person(“Nicholas”, 29, “Software Engineer”);
var person2 = new Person(“Greg”, 27, “Doctor”);
person1.friends.push(“Van”);
alert(person1.friends); //”Shelby,Court,Van”
alert(person2.friends); //”Shelby,Court”
alert(person1.friends === person2.friends); //false
alert(person1.sayName === person2.sayName); //true
```

例子来源：Professional JavaScript for Web Developers

Design Issues of OOP

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

- Option 1: Everything is an object
  - Advantage: elegance and purity
  - Disadvantage: slow operations on simple objects
- Option 2: Add objects to existing typing system
  - Advantage: fast operations on simple objects
  - Disadvantage: confusing typing (2 kinds of entities)
- Option 3: 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?
  - If a derived class is-a parent class, then objects of derived class behave same as parent class object

```
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

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?
- 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
- 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?

Outline

- Object-Oriented Programming (Sec. 12.2)
- Design Issues for OO Languages (Sec. 12.3)
- Support for OOP (Sec. 12.4 ~ 12.9)
  - Smalltalk
  - C++
  - Java
  - C#
  - Ada 95
  - Ruby
- Implementation of OO Constructs (Sec. 12.10)

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++ (cont.)

- 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 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 {…};
// b, y protected; c, z public
class subclass_2 : private base_class {…};
// b, y, c, z private; no access by derived
```

Inheritance Example 2 in C++

```
class base{
     int x;
  public:
     void setx(int n)  {x = n;}
     void showx(void)  {cout << x << "\n";}};
class derived : private base{
     int y;
  public:
     void setxy(int n) {y = n; x = n; }
     showxy(void) {
       cout<<x<<"\n"; cout<<y<<"\n"; } };
```

Where are the errors?

(http://www.umsl.edu/~subramaniana/private_inherit.html)

Inheritance Example 2 (cont.)

```
class base{
     int x;
  public:
     void setx(int n)  {x = n;}
     void showx(void)  {cout<<x<< "\n"; } };
class derived : private base{
     int y;
  public:
     void setxy(int n, m)  {setx(n); y = m; }
     void showxy(void)  {showx(); cout<<y; }
};
```

(http://www.umsl.edu/~subramaniana/private_inherit.html)

Reexportation 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

Another Example in C++

```
class single_LL {
  private:
     class node {
       public: node *link;  int contents; };
     node *head;
public:
     single_LL() {head = 0};
     void insert_at_head(int);
     void insert_at_tail(int);
     int remove_at_head();
     int empty();  }
```

Another Example in C++ (cont.)

```
class stack: public single_LL {
  public:
     stack() {};
     void push(int value) {
       single_LL::insert_at_head(value); };
    int pop() {
       return single_LL::remove_at_head();};
};
class queue: public single_LL {
  ... enqueue(int value) ... dequeue() ...
};
```

Problem with Public Derivation

- Clients of **`stack`** can access all of the public members of the parent class, **`single_LL`**, including **`insert_at_tail`** → not desirable
  - Public derivation is used when one wants subclass to inherit the entire interface of the base class
  - Let subclass inherit only the implementation of the base class and make it not subtype of the parent

Private Derivation in C++

```
class stack2: private single_LL {
  public:
     stack2() {};
     void push(int value) {
       single_LL::insert_at_head(value); };
    int pop() {
      return single_LL::remove_at_head();};
    single_LL::empty(); //reexport
};
```

Private Derivation in C++ (cont.)

```
class queue2: private single_LL {
  public:
     queue2() {};
     void enqueue(int value) {
       single_LL::insert_at_tail(value); };
    int dequeue() {
       return single_LL::remove_at_head();};
    single_LL::empty(); //reexport
};
```

Dynamic Binding in C++

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

Dynamic Binding in C++ (cont.)

```
square* sq = new square;
rectangle* rect = new rectangle;
shape* shape_ptr;
shape_ptr = sq;
shape_ptr->draw(); // dynamically bound
rect = sq;
rect->draw(); // statically bound
square sq1; // sq1 on stack
rectangle rect1;
rect1 = sq1; // copy data member values
rect1.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!
  - 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

Inheritance 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)

Dynamic Binding in Java

- In Java, all messages are dynamically bound to methods, unless the method is **`final`** (i.e., it cannot be overridden, and thus dynamic binding serves no purpose)
- Static binding is also used if the method is **`static`** or **`private`**, both of which disallow overriding

Nested Classes in Java

- 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

Evaluation of Java

- 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 Ruby

- General characteristics
  - Everything is an object and 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

Outline

- Object-Oriented Programming (Sec. 12.2)
- Design Issues for OO Languages (Sec. 12.3)
- Support for OOP (Sec. 12.4 ~ 12.9)
  - Smalltalk
  - C++
  - Java
  - C#
  - Ada 95
  - Ruby
- Implementation of OO Constructs (Sec. 12.10)

Implementing OO Constructs

- Most OO constructs can be implemented easily by compilers
  - Abstract data types → scope rules
  - Inheritance
- 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 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

Dynamic Binding of Methods Calls

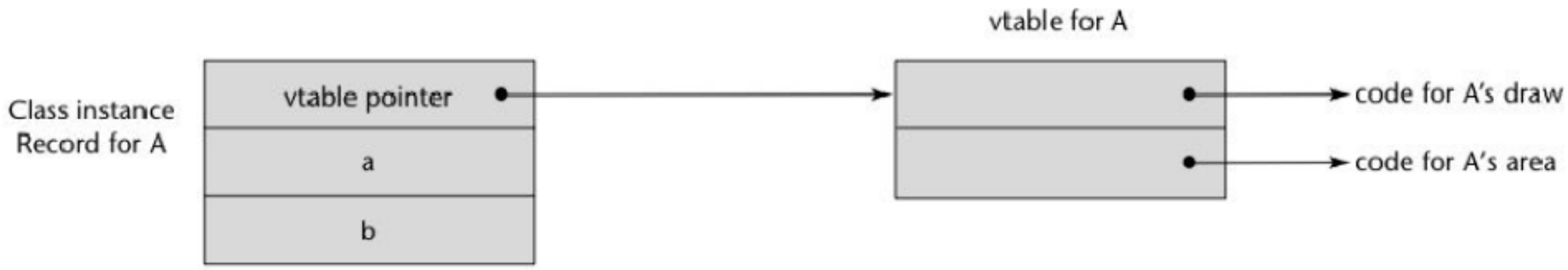
- Methods in a class that are statically bound need not be involved in CIR; methods that are dynamically bound must have entries in the CIR
  - Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  - Storage structure for the list of dynamically bound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

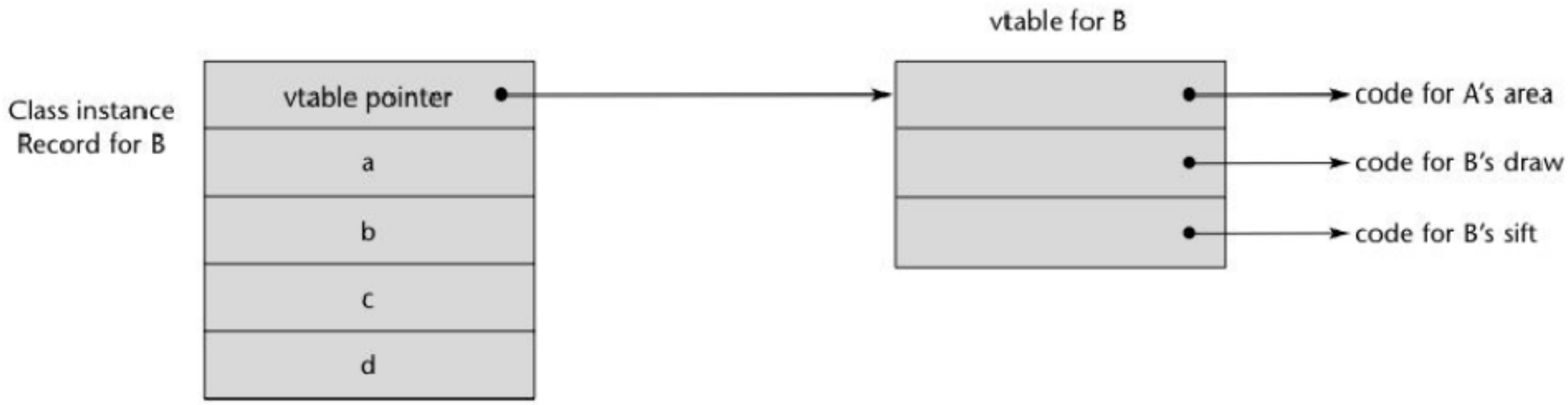
Example of CIR in Java

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

Example CIR


vtable for A
Class instance
Record for A
vtable pointer
a
b
code for A's draw
code for A's area


vtable for B
Class instance
Record for B
vtable pointer
a
b
c
d
code for A's area
code for B's draw
code for B's sift

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
- C++ has two distinct type system (hybrid)
- Java is not a hybrid language like C++; it supports only OO programming