COSC252: Programming Languages:

Abstraction and OOP

Jeremy Bolton, PhD
Asst Teaching Professor

GEORGETOWN UNIVERSITY

Topics

- The Concept of Abstraction
  - Introduction to Data Abstraction
  - Design Issues for Abstract Data Types
  - Language Examples
  - Parameterized Abstract Data Types
  - Encapsulation Constructs
    - Naming Encapsulations
- Object-Oriented Programming
  - Design Issues for Object-Oriented Languages
  - Examples for Object-Oriented Programming in C++
  - Examples for Object-Oriented Programming in Java
- Implementation of Object-Oriented Constructs

Introduction

- Object-oriented programming languages began in the 1960s with Simula
  - Goals were to incorporate the notion of an object, with properties that control its ability to react to events in predefined ways
  - Factor in the development of abstract data type mechanisms
  - Crucial to the development of the object paradigm itself

The Concept of Abstraction

- An *abstraction* is a view or representation of an entity that includes only the most significant attributes

- The concept of abstraction is fundamental in programming (and computer science)

- Nearly all programming languages support process abstraction with subprograms

- Nearly all programming languages designed since 1980 support *data abstraction*

Introduction to Data Abstraction

- An *abstract data type* is a user-defined data type that satisfies the following two conditions:

1. The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition

2. The declarations of the type and the protocols of the operations on objects of the type are contained in a single syntactic unit. Other program units are allowed to create variables of the defined type.

Advantages of Data Abstraction

- Advantages the first condition
  - **Reliability**--by **hiding** the data representations, user code cannot directly access objects of the type or depend on the representation, allowing the representation to be changed without affecting user code
  - **Simplicity.** Reduces the range of code and variables of which the programmer must be aware
  - Name conflicts are less likely

Advantages of Data Abstraction

- Advantages of the second condition
  - Provides a method of program organization
  - Aids modifiability (everything associated with a data structure is together)
  - Separate compilation

Software Reuse and Independence

- Object-oriented programming languages satisfy three important needs in software design:
  - Need to reuse software components as much as possible
  - Need to modify program behavior with minimal changes to existing code
  - Need to maintain the independence of different components
- Abstract data type mechanisms can increase the independence of software components by separating interfaces from implementations

Software Reuse and Independence (cont'd.)

- Four basic ways a software component can be modified for reuse:
  - Extension of the data or operations
  - Redefinition of one or more of the operations
  - Abstraction
  - Polymorphism
- Extension of data or operations:
  - Example: adding new methods to a queue to allow elements to be removed from the rear and added to the front, to create a double-ended queue or deque

Software Reuse and Independence (cont'd.)

- Redefinition of one or more of the operations:
  - Example: if a square is obtained from a rectangle, area or perimeter functions may be redefined to account for the reduced data needed
- Abstraction, or collection of similar operations from two different components into a new component:
  - Example: can combine a circle and rectangle into an abstract object called a figure, to contain the common features of both, such as position and movement

Software Reuse and Independence (cont'd.)

- Polymorphism, or the extension of the type of data that operations can apply to:
  - Examples: overloading and parameterized types
- **Application framework**: a collection of related software resources (usually in object-oriented form) for developer use
  - Examples: **Microsoft Foundation Classes** in C++ and **Swing** windowing toolkit in Java

Software Reuse and Independence (cont'd.)

- Object-oriented languages have another goal:
  - Restricting access to internal details of software components
- Mechanisms for restricting access to internal details have several names:
  - Encapsulation mechanisms
  - Information-hiding mechanisms

Language Requirements for ADTs

- A syntactic unit in which to encapsulate the type definition
- A method of making type names and subprogram headers visible to clients, while hiding actual definitions
- Some primitive operations must be built into the language processor

Design Issues

- Can abstract types be parameterized?
- What access controls are provided?
- Is the specification of the type physically separate from its implementation?

Language Examples: C++ class

- Based on C **`struct`** type

- The **class** is the encapsulation device

- A **class** is a type

- All of the class instances of a class share a *single copy of the member functions*

- Each instance of a class has its own copy of the class data members

- Instances can be static, stack dynamic, or heap dynamic

Language Examples: C++ (continued)

- Information Hiding
  - *Private* clause for hidden entities
  - *Public* clause for interface entities
  - *Protected* clause for inheritance

Language Examples: C++ (continued)

- Constructors:
  - Functions to initialize the data members of instances
  - May also allocate storage if part of the object is heap-dynamic
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Can be explicitly called
  - Name is the same as the class name

Language Examples: C++ (continued)

- Destructors:
  - Functions to cleanup after an instance is destroyed; *usually just to reclaim heap storage*
  - Implicitly called when the object's lifetime ends
  - Name is the class name, preceded by a tilde (~)

An Example in C++

```
class Stack {
   private:
    int *stackPtr, maxLen, topPtr;
   public:
    Stack() { // a constructor
        stackPtr = new int [100];
        maxLen = 99;
        topPtr = -1;
    };
    ~Stack () {delete [] stackPtr;};
    void push (int number) {
          if (topSub == maxLen)
            cerr << "Error in push - stack is full\n";
          else stackPtr[++topSub] = number;
       };
    void pop () {…};
    int top () {…};
    int empty () {…};
}
```


GEORGETOWN UNIVERSITY

A `Stack` class header file: Contains protocols / interface

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
class Stack {
private: //** These members are visible only to other
//** members and friends
  int *stackPtr;
  int maxLen;
  int topPtr;
public: //** These members are visible to clients
  Stack(); //** A constructor
  ~Stack(); //** A destructor
  void push(int);
  void pop();
  int top();
  int empty();
}
```


GEORGETOWN UNIVERSITY

The code file for `Stack`

```
// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() { //** A constructor
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;}; //** A destructor
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

Summary of Abstraction

- The concept of ADTs and their use in program design was a milestone in the development of languages
- Two primary features of ADTs are the ***packaging of data*** with their associated operations and ***information hiding***
- C++ data abstraction is provided by classes. Java’s data abstraction is similar to C++
- C++, C#, Java, and Ruby provide naming encapsulations

Introduction OOP

- Many object-oriented programming (OOP) languages
  - Some support procedural and data-oriented programming (e.g., 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 (**3 main benefits**):
  - Abstract data types
    - Encapsulation
    - Information hiding
  - 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

Nomenclature: Object-Oriented Concepts

- ADTs are usually called (implemented as) *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 ***member 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)

- **Inheritance can be complicated by access controls** to encapsulated entities
  - A class can hide entities from its subclasses

- 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 ***overriden***

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 (static)* – one per class
  - *Instance variables* – one per 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: design concerns!
- 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 **<u>reference</u>** (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 Concepts

- Using abstraction concept
- 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 virtual method
- An abstract class cannot be instantiated

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 (primitives)

- 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
  - **Principle of substitution**: a variable can be substituted for that of an ancestor class

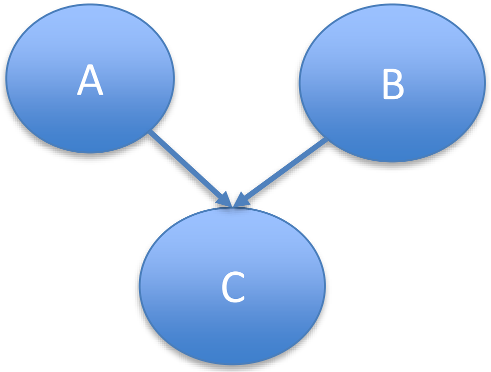
- Subclasses inherit implementation; subtypes inherit interface and behavior

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)
  - Class C inherits from A and B ?!

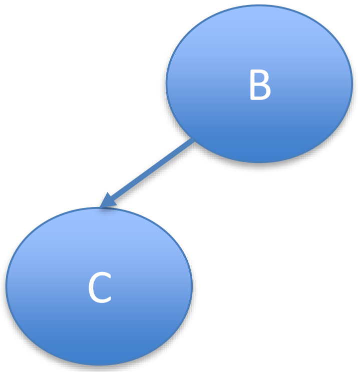
- Advantage: flexibility
  - Sometimes it is quite convenient and valuable



GEORGETOWN UNIVERSITY

Allocation and DeAllocation of Objects

- From where are objects allocated?
  - Options
    - Allocated on 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 (Java)
  - If objects are stack dynamic, there is a problem with regard to subtypes – ***object slicing***
    - What if C objects contain more attributes that B objects



- Is deallocation explicit or implicit on heap?
  - Explicit: user error concerns
  - Implicit: garbage collection overhead

GEORGETOWN UNIVERSITY

Object Slicing Example

- If an object is allocated on the stack, there are memory allocation concerns associated with polymorphic behavior.
  - More specifically: this occurs when handling the variable as opposed to a pointer to the variable

  - Account a;
  - SavingsAcct s;
  - a = s; // copy s to a, what

Dynamic and Static Binding

- Dynamic dispatch: polymorphic methods
- 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

Support for OOP in C++

- 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
```

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 { … }
```

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*

- Implications:
  - Upon method invocation
    1. Determine the actual type of the calling object
    2. Determine which method is actually being invoked based on class def or inheretence hierarchy
    3. Invoke method

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() { ... }
  ...
};
class Square : public Rectangle {
  public:
    void draw() { ... }
  ...
};
```

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

Support for OOP in C++ (continued)

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

```
Square sq;    // Allocates a Square object from the stack
Rectangle rect;  // Allocates a Rectangle object from the stack
rect = sq;   // Copies the data member values from sq object
rect.draw();  // Calls the draw from Rectangle
```


GEORGETOWN UNIVERSITY

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!
  - Smalltalk type checking is dynamic (flexible, but somewhat unsafe)
  - Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++

Implementing OO Constructs

- Two interesting and challenging parts
  - Storage structures for instance variables
  - Dynamic binding of messages to methods

- Your projects will have similar concerns!

Implementing Objects: Instance Data Storage

- Class instance records (CIRs) store the state of an object
  - Static (built at compile time)
  - A template or “cookie-cutter” pattern for all instances of a class.
- 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; 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
    - CIR may store pointers to all dynamically bound methods (inefficient, unnecessary repetition)
    - The storage structure is sometimes called *virtual method tables* (vtable)
    - V-table: a table of pointers to all dynamic methods for a particular (sub)class
    - Method calls can be represented as offsets from the beginning of the vtable

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
- Implementing OOP involves some new data structures
  - CIRs
  - v-tables

Appendix: More OOP Examples

GEORGETOWN UNIVERSITY

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 comparedTo (T b);
}
```

  - Methods can be **`final`** (cannot be overriden)
  - 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 overriden, 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)

- 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

Language Examples: C#

- Based on C++ and Java
- Adds two access modifiers, *internal* and *protected internal*
- All class instances are heap dynamic
- Default constructors are available for all classes
- Garbage collection is used for most heap objects, so destructors are rarely used
- **`struct`**s are lightweight classes that do not support inheritance

Language Examples: C# (continued)

- Common solution to need for access to data members: accessor methods (getter and setter)
- C# provides *properties* as a way of implementing getters and setters without requiring explicit method calls

C# Property Example

```
public class Weather {
   public int DegreeDays { //** DegreeDays is a property
      get {return degreeDays;}
      set {
        if (value < 0 || value > 30)
          Console.WriteLine(
               "Value is out of range: {0}", value);
        else degreeDays = value;}
   }
   private int degreeDays;
   ...
   }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Language Examples – Objective-C

- Interface containers

```
@interface class-name: parent-class {
    instance variable declarations
}
  method prototypes
@end
```

- Implementation containers

```
@implementation class-name
method definitions
@end
```

- Classes are types

Language Examples – Objective-C (continued)

- Method prototypes form

  (+ | –) (return-type) method-name [`:` (formal-parameters)]`;`

  - Plus indicates a class method
  - Minus indicates an instance method
  - The colon and the parentheses are not included when there are no parameters
  - Parameter list format is different
    - If there is one parameter (name is `meth1:`)

      ```
      -(void) meth1: (int) x;
      ```

    - For two parameters

      ```
      -(int) meth2: (int) x second: (float) y;
      ```

      - The name of the method is `meth2::`

Language Examples – Objective-C
(continued)

- Method call syntax

[object-name method-name];

Examples:

```
[myAdder add1: 7];
[myAdder add1: 7: 5: 3];
```

- For the method:

```
-(int) meth2: (int) x second: (float) y;
```

the call would be like the following:

```
[myObject meth2: 7 second: 3.2];
```

Language Examples – Objective-C
(continued)

- Constructors are called *initializers* – all they do is initialize variables
  - Initializers can have any name – they are always called explicitly
  - Initializers always return **`self`**
- Objects are created by calling **`alloc`** and the constructor

```
Adder *myAdder = [[Adder alloc] init];
```

- All class instances are heap dynamic

Language Examples – Objective-C
(continued)

- To import standard prototypes (e.g., i/o)

```
#import <Foundation/Foundation.h>
```

- The first thing a program must do is allocate and initialize a pool of storage for its data (pool's variable is `pool` in this case)

```
NSAutoreleasePool * pool =
        [[NSAutoreleasePool alloc] init];
```

- At the end of the program, the pool is released with:

```
[pool drain];
```


GEORGETOWN UNIVERSITY

Language Examples – Objective-C *(continued)*

- Information Hiding
  - The directives `@private` and `@public` are used to specify the access of instance variables.
  - The default access is protected (private in C++)
  - There is no way to restrict access to methods
  - The name of a getter method is always the name of the instance variable
  - The name of a setter method is always the word set with the capitalized variable's name attached
  - If the getter and setter for a variable does not impose any constraints, they can be implicitly generated (called *properties*)

Language Examples – Objective-C (continued)

```
// stack.m - interface and implementation for a simple stack
#import <Foundation/Foundation.h>
@interface Stack: NSObject {
  int stackArray[100], stackPtr,maxLen, topSub;
}
  -(void) push: (int) number;
  -(void) pop;
  -(int) top;
  -(int) empty;
@end
@implementation Stack
  -(Stack *) initWith {
    maxLen = 100;
    topSub = -1;
    stackPtr = stackArray;
    return self;
}
```

Language Examples – Objective-C
(continued)

```
// stack.m - continued
  -(void) push: (int) number {
    if (topSub == maxLen)
      NSLog(@"Error in push - stack is full");
    else
      stackPtr[++topSub] = number;
  ...
}
```

Language Examples – Objective-C
(continued)

- An example use of `stack.m`
  - Placed in the `@implementation` of `stack.m`

```
int main (int argc, char *argv[]) {
  int temp;
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  Stack *myStack = [[Stack alloc] initWith];
  [myStack push: 5];
  [myStack push: 3];
  temp = [myStack top];
  NSLog(@"Top element is: %i", temp);
  [myStack pop];
  temp = [myStack top];
  NSLog(@"Top element is: %i", temp);
  temp = [myStack top];
  myStack pop];
  [myStack release];
  [pool drain];
  return 0;
}
```

Support for OOP in C#

- General characteristics
  - Support for OOP similar to Java
  - Includes both classes and **`struct`**s
  - Classes are similar to Java's classes
  - **`struct`**s 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

Support for OOP in Objective-C

- Like C++, Objective-C adds support for OOP to C
- Design was at about the same time as that of C++
- Largest syntactic difference: method calls
- Interface section of a class declares the instance variables and the methods
- Implementation section of a class defines the methods
- Classes cannot be nested

Support for OOP in Objective-C (continued)

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

```
@interface myNewClass: NSObject { … }
    …
 @end
```

  - Because all public members of a base class are also public in the derived class all subclasses are subtypes
  - Any method that has the same name, same return type, and same number and types of parameters as an inherited method overrides the inherited method
  - An overriden method can be called through **`super`**
  - All inheritance is public (unlike C++)

Support for OOP in Objective-C
(continued)

- Inheritance (continued)
- Objective-C has two approaches besides subclassing to extend a class
  - A *category* is a secondary interface of a class that contains declarations of methods (no instance variables

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

  - A category is a *mixin* – its methods are added to the parent class
  - The implementation of a category is in a separate implementation: `@implementation Stack (StackExtend)`

Support for OOP in Objective-C
(continued)

- Inheritance (continued)
  - The other way to extend a class: protocols
  - A protocol is a list of method declarations

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

  - `MatrixOps` is the name of the protocol
  - The `add` and `subtract` methods must be implemented by class that uses the protocol
  - A class that adopts a protocol must specify it

```
@interface MyClass: NSObject <YourProtocol>
```

Support for OOP in Objective-C (continued)

- Dynamic Binding
  - Different from other OOP languages – a polymorphic variable is of type `id`
  - An `id` type variable can reference any object
  - The run-time system keeps track of the type of the object that an `id` type variable references
  - If a call to a method is made through an `id` type variable, the binding to the method is dynamic

Support for OOP in Objective-C
(continued)

- Evaluation
  - Support is adequate, with the following deficiencies:
  - There is no way to prevent overriding an inherited method
  - The use of `id` type variables for dynamic binding is overkill – these variables could be misused
  - Categories and protocols are useful additions

Support for OOP 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 objected 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