# CS350 Programming Language Design

Spr 21

Ch11&12 : Support for OOP

Dr. Amal Khalifa

Okay, any ideas on how to make women more interested in us?
java
C++
python
Make more exceptions?
Redefine our methods?
Stop treating them like objects?
ANSI C
szym1000 / 9GAG
Java
ANSI C

## Goal

*Discussion of the primary design issues for abstraction, inheritance and dynamic binding*

# Topics

12.1 Introduction

12.2 Object-Oriented Programming

11.2 Data abstraction

11.4 language Examples:

C++, Java , and Ruby

12.3 Design Issues for Object-Oriented Languages

The Exclusivity of Objects

Single vs. multiple inheritance

Allocation of objects

Dynamic vs. static binding

Nested classes

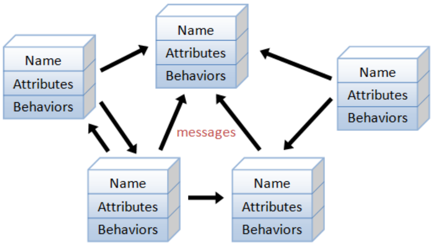
Initialization of objects

12.5 Implementation of Object-Oriented Constructs

C++, Java and Ruby

# OOP

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields/attributes/properties), and code, in the form of procedures/methods/functions.



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

# 12.3 Design Issues for OOP Languages

- Data abstraction and objects
- Single and Multiple Inheritance
- Object Allocation and Deallocation
- Dynamic and Static Binding
- Nested Classes
- Initialization of Objects

OOP
Object
Inheritance
Encapsulation
Abstraction
Polymorphism
Attribute
Method
Class

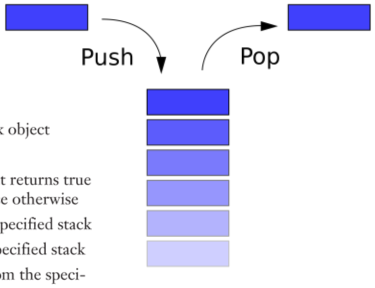
# Data abstraction

# Data abstraction

- An abstraction is a view or representation of an entity that includes only the most significant attributes.
  - allows one to collect instances of entities into groups
- an abstract data type (ADT) is an enclosure that includes:
  - the data representation of one specific data type
  - the subprograms that provide the operations for that type.
- Through access controls, unnecessary details of the type can be hidden from units outside the enclosure that use the type.

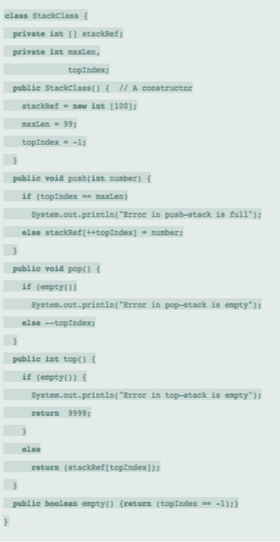
# Example : stack

- a widely applicable data structure that stores some number of data elements and only allows access to the data element at one of its ends, the top

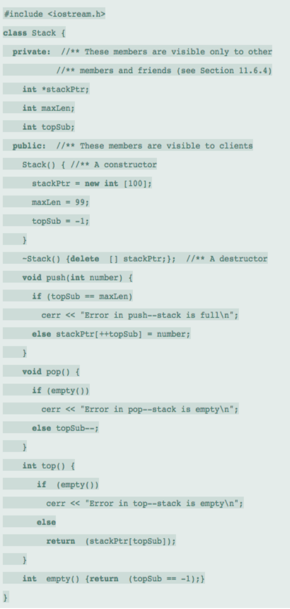


| | |
|---|---|
| create(stack) | Creates and possibly initializes a stack object |
| destroy(stack) | Deallocates the storage for the stack |
| empty(stack) | A predicate (or Boolean) function that returns true if the specified stack is empty and false otherwise |
| push(stack, element) | Pushes the specified element on the specified stack |
| pop(stack) | Removes the top element from the specified stack |
| top(stack) | Returns a copy of the top element from the specified stack |

# Code: C++ & Java



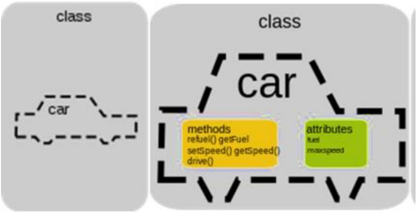
```java
class StackClass {
  private int [] stackRef;
  private int maxLen,
              topIndex;
  public StackClass() {  // A constructor
    stackRef = new int [100];
    maxLen = 99;
    topIndex = -1;
  }
  public void push(int number) {
    if (topIndex == maxLen)
      System.out.println("Error in push—stack is full");
    else stackRef[++topIndex] = number;
  }
  public void pop() {
    if (empty())
      System.out.println("Error in pop—stack is empty");
    else --topIndex;
  }
  public int top() {
    if (empty()) {
      System.out.println("Error in top—stack is empty");
      return  9999;
    }
    else
      return (stackRef[topIndex]);
  }
  public boolean empty() {return (topIndex == -1);}
}
```



```cpp
#include <iostream.h>
class Stack {
  private:  //** These members are visible only to other
           //** members and friends (see Section 11.6.4)
    int *stackPtr;
    int maxLen;
    int topSub;
  public:  //** These members are visible to clients
    Stack() { //** A constructor
      stackPtr = new int [100];
      maxLen = 99;
      topSub = -1;
    }
    ~Stack() {delete  [] stackPtr;};  //** A destructor
    void push(int number) {
      if (topSub == maxLen)
        cerr << "Error in push--stack is full\n";
      else stackPtr[++topSub] = number;
    }
    void pop() {
      if (empty())
        cerr << "Error in pop--stack is empty\n";
      else topSub--;
    }
    int top() {
       if  (empty())
         cerr << "Error in top--stack is empty\n";
       else
         return  (stackPtr[topSub]);
    }
    int  empty() {return  (topSub == -1);}
}
```

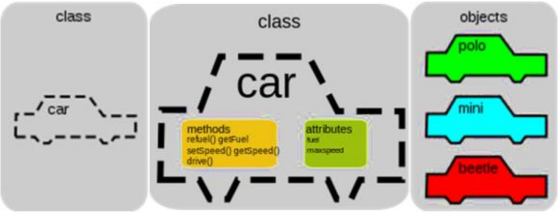
# Classes

- ADTs are usually called classes
- The class is a <u>syntactic</u> unit that encloses the declaration of:
  - the type
  - the prototypes of the subprograms (operations on objects)
  - variables



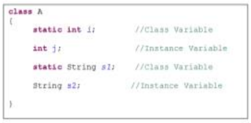
# Objects

- Allow clients to declare variables of the abstract type and manipulate their values.
- Class instances are called objects
- Calls to methods are called messages
  - Messages have two parts--a method name and the destination object

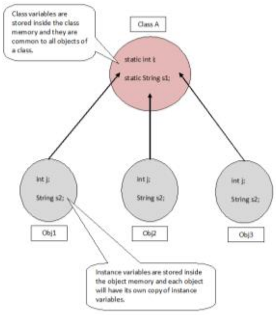


# Object-Oriented Concepts

- 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 – messages to the class
  - Instance methods –messages to objects



```
class A
{
        static int i;           //Class Variable
        int j;                  //Instance Variable
        static String s1;       //Class Variable
        String s2;             //Instance Variable
}
```

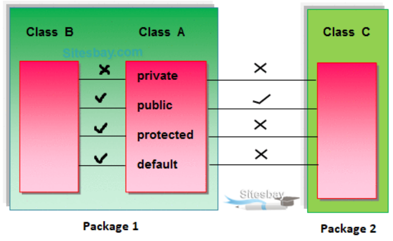


Class variables are stored inside the class memory and they are common to all objects of a class.

Instance variables are stored inside the object memory and each object will have its own copy of instance variables.

# Information hiding

- Although the type name must have external visibility, the type representation must be hidden
  - Elaborate access controls to class entities
  - Access controls for members are
    - Private (visible only in the class and friends)
    - Public (visible in subclasses and clients)
    - Protected (visible in the class and in subclasses, but not clients)



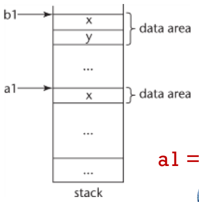
# Design Issues for Abstract data types

- The Exclusivity of Objects
  - Add objects to a complete procedural typing system (e.g., C++)
    - 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 (e.g., Java and C#)
    - Advantage - fast operations on simple objects and a relatively small typing system
    - Disadvantage - still some confusion because of the two type systems
  - Everything is an object (e.g., Smalltalk & Ruby)
    - Advantage - elegance and purity
    - Disadvantage - slow operations on simple objects

# Design Issues for Abstract data types

- Allocation and Deallocation of Objects
  - from the run-time stack (C++)
    - Excess space truncation– object slicing
  - on the heap
    - Sometimes explicitly (via new)
    - references can be uniform thru a pointer or reference variable
    - Simplifies assignment - dereferencing can be implicit
    - Is deallocation explicit or implicit?
      - finalize method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object

```
class A {
  int x;
  . . .
};
class B : A {
  int y;
  . . .
}
```



a1 = b1;

# Design Issues for Abstract data types

- built-in operations should be provided for objects of abstract data types, other than those provided with the type definition.
  - Assignment, equality, comparison
- Non-universal operations for abstract data types
  - Constructors/destructors
  - Iterators / accessors
- Whether abstract data types can be parameterized?
  - structure that could store elements of any type

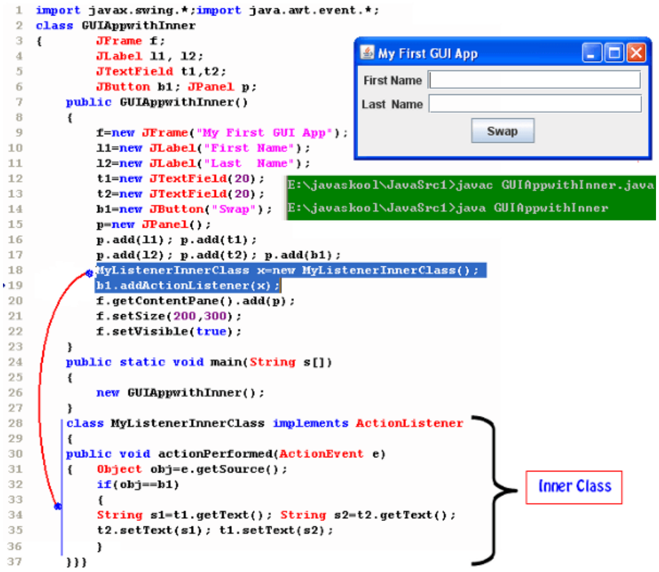
17

# Group exercise

Highlight at least three data abstraction features in *Ruby*.

# 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



```
import javax.swing.*;import java.awt.event.*;
class GUIAppwithInner
{         JFrame f;
          JLabel l1, l2;
          JTextField t1,t2;
          JButton b1; JPanel p;
     public GUIAppwithInner()
     {
          f=new JFrame("My First GUI App");
          l1=new JLabel("First Name");
          l2=new JLabel("Last  Name");
          t1=new JTextField(20);
          t2=new JTextField(20);
          b1=new JButton("Swap");
          p=new JPanel();
          p.add(l1); p.add(t1);
          p.add(l2); p.add(t2); p.add(b1);
          MyListenerInnerClass x=new MyListenerInnerClass();
          b1.addActionListener(x);
          f.getContentPane().add(p);
          f.setSize(200,300);
          f.setVisible(true);
     }
     public static void main(String s[])
     {
          new GUIAppwithInner();
     }
     class MyListenerInnerClass implements ActionListener
     {
     public void actionPerformed(ActionEvent e)
     {    Object obj=e.getSource();
          if(obj==b1)
          {
          String s1=t1.getText(); String s2=t2.getText();
          t2.setText(s1); t1.setText(s2);
          }
     }}}
```

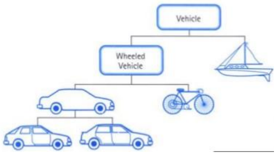
```
E:\javaskool\JavaSrc1>javac GUIAppwithInner.java
E:\javaskool\JavaSrc1>java GUIAppwithInner
```

Inner Class

# Inheritance

# Inheritance

- Productivity increases can come from reuse
  - ADTs 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
  - derived class/subclass
  - parent class/superclass
- One disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance



# Inheritance

- Three ways a class can differ from its parent

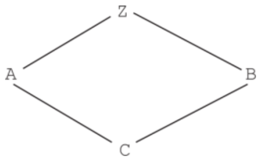
The subclass can add variables and/or methods to those inherited from the parent

The subclass can modify the behavior of one or more of its inherited methods → override

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

# 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)
  - Diamond inheritance
- Advantage:
  - Sometimes it is quite convenient and valuable
- Interfaces can be a good alternative



# Inheritance issues

- 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 for its clients while allowing its subclasses to see them

## Initialization of Objects

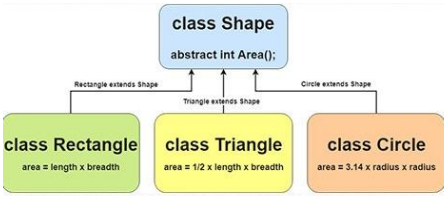
Are objects initialized to values when they are created?

Implicit or explicit

How are parent class members initialized when a subclass object is created?

## Interfaces and Abstract classes

- 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
- An Interface is a pure Abstract class.



class Shape

abstract int Area();

Rectangle extends Shape

Triangle extends Shape

Circle extends Shape

class Rectangle

area = length x breadth

class Triangle

area = 1/2 x length x breadth

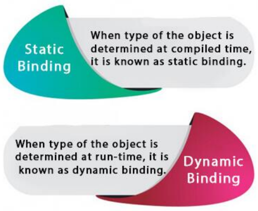
class Circle

area = 3.14 x radius x radius

# Polymorphism

## Polymorphism

- A polymorphic variable can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants

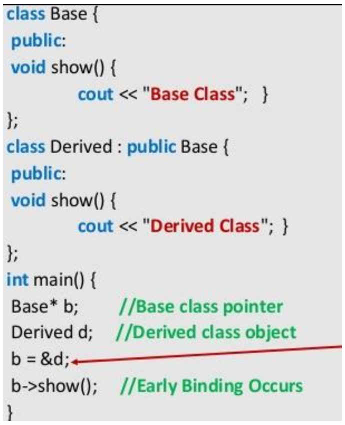


Static Binding

When type of the object is determined at compiled time, it is known as static binding.

Dynamic Binding

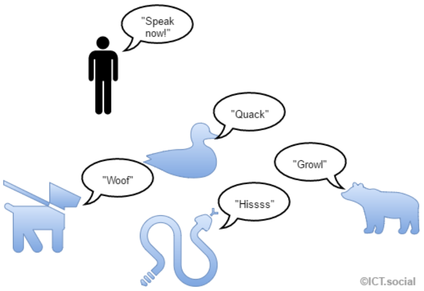
When type of the object is determined at run-time, it is known as dynamic binding.



```
class Base {
 public:
 void show() {
            cout << "Base Class";  }
};
class Derived : public Base {
 public:
 void show() {
            cout << "Derived Class"; }
};
int main() {
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Early Binding Occurs
}
```

# Polymorphic behavior

- 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



# Extensible Programming

- Allows software systems to be more easily extended during both development and maintenance

## Dynamic vs. static Binding

- Design issue: 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

## 12.4 Support for OOP

# Support for OOP, Inheritance

## C++

- A class need not be the subclass of any class
- the subclassing process can be declared with access controls
  - 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
- Multiple inheritance is supported
  - If there are two inherited members with the same name, they can both be referenced using the scope resolution operator (::)

## Java

- Object class, default superclass
- Single inheritance
- Initialization of Objects from subclass is done either implicitly or explicitly
- Methods can be final (cannot be overridden)

# Support for OOP, Dynamic Binding

## C++

- A method can be defined to be virtual, which means that they can be called through polymorphic variables and dynamically bound to messages

## Java

- all messages are dynamically bound to methods, unless the method is :
  - final
  - static or
  - private

  (i.e., it cannot be overridden, therefore dynamic binding serves no purpose)

# Support for OOP, abstract classes/interfaces

## C++

- A class that has at least one pure virtual function is an abstract class
- A pure virtual function has no definition at all

## Java

- Interfaces are pure abstract classes that provide some of the benefits of multiple inheritance
- An interface can include only method declarations and named constants, e.g.,

```
public interface
Comparable{
public int comparedTo
 (object b);
    }
```

- A class <u>implements</u> an interface

# Support for OOP, Evaluation

## C++

- C++ provides extensive access controls
- 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!

## 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

# Group exercise

Highlight at least three OOP features in *Ruby*.

# Summary

- OO programming involves three fundamental concepts: ADTs, inheritance, dynamic binding
- Major design issues: exclusivity of objects, 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 systems (hybrid)
- Java is not a hybrid language like C++; it supports only OOP

# ANY Q??