

Object-Oriented Programming

CSCI 334

Stephen Freund

Operator Overloading

```
class Vector {
  int data[10];
  int size;

  // overload * to be dot-product on vectors
  int operator*(Vector other) {
    int result = 0;
    for (int i = 0; i < size; i++) {
      result += data[i] * other.data[i];
    }
    return result;
  }
}

Vector v1;
Vector v2;
...
int dot_prod = v1 * v2;
```

Modular Program Development

- Component
 - logical program unit
- Interface
 - set of operations exported from a component
- Specification
 - description of behavior of component
- Implementation
 - code that defines interface operations (according to spec)

abstype Stack

```
abstype Stack =
  StackRep of int list
with
  val empty = StackRep(nil);
  fun push(n, StackRep(l)) = StackRep(n::l);
  fun top(StackRep(nil)) = 0
    | top(StackRep(n::l)) = n;
  fun pop(StackRep(nil)) = empty
    | pop(StackRep(n::l)) = StackRep(l);
end;
```

Module

- Interface
 - set of name and type declarations
- Implementation
 - code/decls to define names
- Examples:
 - modules in Modula
 - Ada packages
 - ML structures
 - Java packages

Generic Abstractions

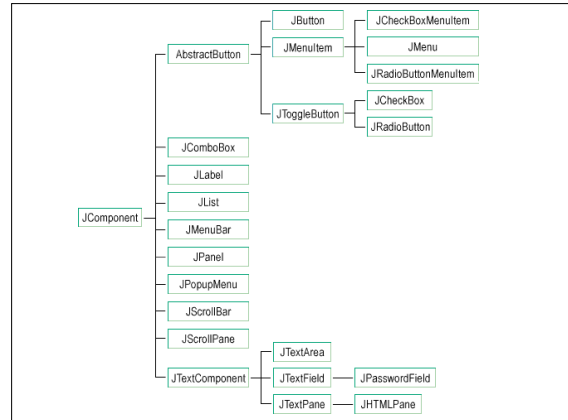
- Parameterize module by types
 - Ada generics, ML functors, C++ STL library
- Example

```
class Stack<T> {
  private Vector<T> elems;
  public boolean empty() { ... }
  public void push(T elem) { ... }
  public T pop() { ... }
}

Stack<String> stringStack;
Stack<Integer> intStack;
```

Generic Constructs (a la STL)

- Collections
 - Iterators
 - Adaptors
 - Algorithms
 - Function Objects
- HW this week: use generic constructs in Java.
- Next few weeks: implement various design patterns to illustrate OOP principles.



Subtyping and Substitutivity

```

class Rectangle {
    private int x,y,w,h;
    void moveTo(int x, int y);
    void setSize(int width, int height);
    void show();
    void hide();
}
    
```

```

class FilledRectangle {
    private int x,y,w,h;
    private Color c;
    void moveTo(int x, int y);
    void setSize(int width, int height);
    void show();
    void hide();
    void setFillColor(Color color);
    Color getFillColor();
}
    
```

Subtyping and Substitutivity

```

void f() {
    Rectangle r =
        new Rectangle();
    r.moveTo(100,100);
    r.hide();
}
    
```

```

void f() {
    Rectangle r =
        new FilledRectangle();
    r.moveTo(100,100);
    r.hide();
}

void g() {
    FilledRectangle r =
        new FilledRectangle();
    r.moveTo(100,100);
    r.setFillColor(Color.red);
    r.hide();
}

void g() {
    FilledRectangle r =
        new Rectangle();
    r.moveTo(100,100);
    r.setFillColor(Color.red);
    r.hide();
}
    
```

Rectangles Revisited

```

void Rectangle {
    int x,y,w,h;
    void moveTo(int x, int y);
    void setSize(int width, int height);
    void show();
    void hide();
}

void FilledRectangle extends Rectangle {
    Color c;
    void setFillColor(Color color);
    Color getFillColor();
}
    
```

OO Program Structure

- Group data and functions
- Class
 - Defines behavior of all objects that are instances of the class
- Subtyping
 - Place similar data in related classes
- Inheritance
 - Avoid reimplementing functions that are already defined

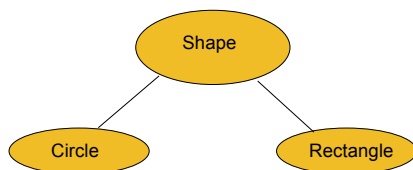
Example: Geometry Library

- Define general concept **Shape**
- Implement two shapes: **Circle, Rectangle**
- Functions on implemented shapes
center, move, rotate, print
- Anticipate additions to library

Shapes

- Interface of every **Shape** must include
center, move, rotate, print
- Different kinds of shapes are implemented differently
 - **Square**: four points, representing corners
 - **Circle**: center point and radius

Subtype Hierarchy



- General interface defined in the **Shape** class
- Implementations defined in **Circle, Rectangle**
- Extend hierarchy with additional shapes

Code Placed In Classes

| | center | move | rotate | print |
|-----------|----------|--------|----------|---------|
| Circle | c_center | c_move | c_rotate | c_print |
| Rectangle | r_center | r_move | r_rotate | r_print |

- Dynamic lookup
 - circle → move(x,y) calls function c_move

Example Use: Processing Loop

```
Vector<Shape> shapes =  
    new Vector<Shape> ();  
...  
for (i = 0; i < shapes.size(); i++) {  
    Shape s = shapes.get(i);  
    s.move(10, 10);  
}
```

Control loop does not know the type of each shape

Object-oriented programming

- Programming methodology
 - organize concepts into objects and classes
 - build extensible systems
- Language concepts
 - encapsulate data and functions into objects
 - subtyping allows extensions of data types
 - inheritance allows reuse of implementation

Object-oriented Method

[Booch]

- Four steps
 - Identify the objects at a given level of abstraction
 - Identify the semantics (intended behavior) of objects
 - Identify the relationships among the objects
 - Implement these objects
- Iterative process
 - Implement objects by repeating these steps
- Not necessarily top-down

This Method

- Based on associating objects with components or concepts in a system
- Why iterative?
 - An object is typically implemented using a number of constituent objects
 - Apply same methodology to subsystems, underlying concepts

Comparison to top-down design

- Similarity:
 - A task is typically accomplished by completing a number of finer-grained sub-tasks
- Differences:
 - Focus of top-down design is on program structure
 - OO methods are based on modeling ideas
 - Combining functions and data into objects makes data refinement more natural (I think)