CSC 533: Organization of Programming Languages

Spring 2005

OOP in Java

- object-oriented programming
- inheritance & polymorphism
- interfaces
- abstract classes
- Java vs. JavaScript

OOP in Java

recall: Java classes look similar to C++ classes

```
class Person
{
  public:
    Person(string nm, string id,
           char sex, int yrs) {
      name = nm;  ssn = id;
      gender = sex;  age = yrs;
    }

    void Birthday() {
      age++;
    }

    void Display() {
      cout << "Name: " << name
           << endl << "SSN : << ssn << endl
           << "Gender: " << gender << endl
           << "Age: " << age << endl;
    }

  private:
    string name, ssn;
    char gender;
    int age;
};
```

```
public class Person
{
  public Person(String nm, String id,
                char sex, int yrs) {
    name = nm;  ssn = id;
    gender = sex;  age = yrs;
  }

  public void Birthday() {
       age++;
    }

  public void Display() {
    System.out.println("Name: " + name +
        "\nSSN :" + ssn + "\nGender: " +
        gender + "\nAge: " + age);
    }

  private String name, ssn;
  private char gender;
  private int age;
}
```

OOP in Java (cont.)

inheritance in Java looks similar to C++

- recall: all reference types are automatically derived from the Object class

```
class Student : public Person
{
  public:
    Student(string nm, string id, char sex,
            int yrs, string sch, int lvl) :
               Person(nm, id, sex, yrs) {
      school = sch;  grade = lvl;
    }

    void Advance() {
      grade++;
    }

    void Display() {
      Person::Display();
      cout << "School: " << school
           << endl << "Grade: "
           << grade << endl;
    }

  private:
    string school;
    int grade;
};
```

```
public class Student extends Person
{
  public Student(String nm, String id,
                 char sex, int yrs,
                 String sch, int lvl) {
      super(nm, id, sex, yrs);
      school = sch;  grade = lvl;
  }

  public void Advance() {
    grade++;
  }

  private String school;
  private int grade;
}
```

`super` calls the constructor of the parent class

OOP in Java (cont.)

```
class Student extends Person
{
  public Student(String nm, String id,
                 char sex, int yrs,
                 String sch, int lvl) {
      super(nm, id, sex, yrs);
      school = sch;  grade = lvl;
  }

  public void Advance() {
    grade++;
  }

  public void Display() {
    super.Display();
    System.out.println("School: " + school +
        "\nGrade: " + grade);
  }

  private String school;
  private int grade;
};
```

similar to C++, can override existing methods

- specify methods of the parent class using `super`

all (non-private) method calls are dynamically bound in Java

- don't need to specify `virtual`

OOP in Java (cont.)

recall: in C++, could create a structure that mixed parent & derived objects

- needed to utilize pointers to dynamically allocated objects

in Java, all library data structures store reference types

- thus, can automatically mix objects
- since methods are bound dynamically, `Display()` calls the appropriate method

```
ArrayList<Person> people = new ArrayList<Person>();

Person p = new Person("Bjarne", "123-45-6789', 'M', 20);
people.add(p);

Student s = new Student("Grace", "9876-54-321", 'F', 21, "Creighton", 16);
people.add(s);

. . .

for (int i = 0; i < people.size(); i++) {
    people.get(i).Display();
}
```


Advanced OOP in Java

in Java, can disallow inheritance explicitly

- when a method is declared `final`, it can't be overridden

```
public final void Birthday() {
    age++;
}
```

- when a class is declared `final`, it can't be inherited from (e.g., Math, Integer)

no multiple inheritance in Java, but can have multiple *interfaces*

- an *interface* describes the methods required for a derived class

```
public interface List<T> {
  boolean add(T o);
  void add(int index, T o);
  boolean contains(T o);
  T get(int index);
  boolean isEmpty();
  int size();
  ...
}
```

List interface

interfaces are useful for grouping generic classes

- a derived classes *implements* an interface
- can have more than one implementation, with different characteristics

```
public class ArrayList<T> implements List<T>
{
    private T[] items;
    . . .
}

public class LinkedList<T> implements List<T>
{
    private T front;
    private T back;
    . . .
}
```

- using the interface, can write generic code that works on any implementation

```
public numOccur(List<String> words, String desired)
{
    int count = 0;
    for (int i = 0; i < words.size(); i++) {
        if (desired.equals(words.get(i))) {
            count++;
        }
    }
}
```


`Comparable` interface

another useful predefined interface is Comparable

```
public interface Comparable<T> {
    int compareTo(T other)
}
```

- any class of objects that can be compared should implement Comparable

```
public final class Integer implements Comparable<Integer> { … }

public final class String implements Comparable<String> { … }

public class Name implements Comparable<Name> { … }
```

the Collections utility class contains a variety of methods that work on Lists

- e.g., copy, fill, frequency, reverse, shuffle, …
- if elements are Comparable, then additional methods
  e.g., sort, binarySearch, min, max, …

Interface example: HW3-4

```
public interface Statement {
  void Read(Tokenizer program);
  void Execute(VarTable vars);
  String GetType();
  String toString();
}
```

can define an interface that specifies what any kind of statement must do

- in Java, enable output by defining a method that converts an Object to a string – toString method is automatically called when outputting

```
public class AssignStatement implements Statement {
 public AssignStatement() { . . . }
 public void Read(Tokenizer program) { . . . }
 public void Execute(VarTable vars) { . . . }
 public String GetType() { . . . }
 public String toString() { . . . }

 private String lhs;
 private Expression rhs;
}
```

have multiple classes implement the Statement interface

- if derived class fails to implement any method from the interface, then compiler will complain

```
public class OutputStatement implements Statement {
 public OutputStatement() { . . . }
 public void Read(Tokenizer program) { . . . }
 public void Execute(VarTable vars) { . . . }
 public String GetType() { . . . }
 public String toString() { . . . }

 private String constant;
 private Expression rhs;
}
```


Abstract class example: HW3-4

```
public abstract class Statement {
  public abstract void Read(Tokenizer program);
  public abstract void Execute(VarTable vars);
  public abstract String GetType();
  public abstract String toString();

  public static Statement GetNext(Tokenizer program) { . . . }
}
```

Note: an interface does not have any code to be inherited

- so multiple interfaces are OK

```
public class AssignStatement extends Statement {
 public AssignStatement() { . . . }
 public void Read(Tokenizer program) { . . . }
 public void Execute(VarTable vars) { . . . }
 public String GetType() { . . . }
 public String toString() { . . . }

 private String lhs;
 private Expression rhs;
}
```

```
public class OutputStatement extends Statement {
 public OutputStatement() { . . . }
 public void Read(Tokenizer program) { . . . }
 public void Execute(VarTable vars) { . . . }
 public String GetType() { . . . }
 public String toString() { . . . }

 private String constant;
 private Expression rhs;
}
```

if you want to have derived classes inherit some common code, but still ensure methods → define an abstract class

- derived class must implement the abstract methods in order to compile

Java vs. JavaScript

recall: Java took many features from C++, but removed/added features due to different design goals

e.g., platform independence → interpreted+compiled execution model
ease of development over efficiency → dynamic binding, garbage collection
simplicity over expressivity → no goto, no implicit coercions, no operator overload
security → no pointer access, byte code verifier

interesting to consider a third C++ variant: JavaScript

- designed to be a scripting language for execution in and enhancements of a Web browser
- developed at Netscape in 1995, integrated into Navigator
  later adopted by IE under the name JScript (some variations)
- as with Java, chose to keep basic syntax of C++ to aid learning
- different design goals yield different features

JavaScript design

intended to be a scripting language for Web pages

→ JavaScript code is embedded directly into HTML, interpreted by browser
→ a.k.a., client-side scripting

scripting applications are more quick-and-dirty, relatively small

→ variables are bound to type and address dynamically for flexibility
→ do not need to declare variables, functions are not typed either
→ code size is limited by the browser

not expected to develop large applications

→ object-based: lots of useful classes predefined (Array, String, Math, …)
  can define new classes but awkward, no info hiding, no inheritance

user security is important, script code security isn't

→ like Java, JavaScript code can't access local files
→ no way to hide the JavaScript source when download Web page

JavaScript example

```
<html>
<head>
  <title> Dave's Hello World Page </title>

  <script type="text/javascript">
    function SayHello(name)
    {
      if (name == "Dave") {
        document.write("It's an honor, Dave!");
      }
      else {
         for (var i = 0; i < 10; i++) {
           document.write("Hello <i>" + name +
                          "</i>, glad to meet you! <br />");
         }
      }
    }
  </script>
</head>

<body>
  <script type="text/javascript">
    userName = prompt("Enter your name:", "Your name");
    SayHello(userName);
  </script>
</body>
</html>
```

JavaScript code can be embedded in the page using SCRIPT tags

- code is executed by the browser, output is inserted into the page
- `document.write` is JavaScript's output routine

control structures are similar to C++/Java

no type for function, parameters

can declare local variables

JavaScript objects

many useful objects w/ methods are predefined

e.g., `Math.abs`
`document.lastModified`

can define new classes

can even utilize inheritance

- very awkward syntax
- no info hiding

can add data/methods dynamically

- die1.owner = "Dave"

also, can put useful code in a separate file, load using SCRIPT tag with SRC attribute

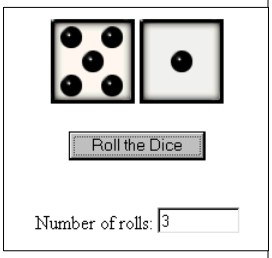
```
// Die.js

function Die(sides) {
  this.numSides = sides;
  this.numRolls = 0;
  this.Roll =
    function () {
      this.numRolls++;
      return Math.floor(Math.random()*this.numSides)+1;
    }
}

function ColoredDie(sides, color)
{
  this.dieColor = color;
  this.Die(sides);
}
ColoredDie.prototype = new Die;
ColoredDie.prototype.Die = Die;
```

```
<html>
<head>
  <title> Roll two dice</title>
  <script type="text/javascript" src="Die.js">
  </script>
</head>

<body>
  <script type="text/javascript">
    die1 = new Die(6);
    die2 = new ColoredDie(6, "blue");

    roll1 = die1.Roll();
    roll2 = die2.Roll();

    document.write("I rolled " + roll1 + " and " +
        roll2 + " ==> " + (roll1+roll2) + "<br />");
  </script>
</body>
</html>
```

JavaScript form processing

main uses of JavaScript

- processing form elements (e.g., text box)
- reacting to user-controlled events (e.g., button click)




```
<html>
 <head>
   <title> Dice Stats </title>
   <script type="text/javascript" src="Die.js"></script>
   <script type="text/javascript">
     function DoIt()
     {
       var die1 = new Die(6);
       var die2 = new Die(6);

       document.images.die1.src = "Images/die"+die1.Roll()+".gif";
       document.images.die2.src = "Images/die"+die2.Roll()+".gif";

       document.DiceForm.numRolls.value =
           parseFloat(document.DiceForm.numRolls.value) + 1;
     }
   </script>
 </head>

 <body>
   <div style="text-align:center">
     <img name="die1" src="Images/die1.gif" />
     <img name="die2" src="Images/die1.gif" />
     <p>
     <form name="DiceForm">
       <input type="button" value="Roll the Dice" onClick="DoIt();">
       <p>
       Number of rolls:
       <input type="text" name="numRolls" size=8 value=0>
     </form>
     </p>
   </div>
 </body>
</html>
```


JavaScript vs. Java implementations

consider a text-manipulation example: convert English text to pirate talk

Java application

- divided into 2 classes:
  - `Translator.java` : a generic language translator
  - `PirateTalk.java` : GUI front-end, displays label & text area & button
- translation vocabulary is read in from a file, so easily changed

JavaScript program (i.e., interactive Web page)

- all contained in a single Web page
  - `PirateTalk.html`
- since can't read external files, vocabulary must be embedded in the page
- GUI elements are much simpler (since HTML), easy to embed images