CMSC 132:
Object-Oriented Programming II

Object-Oriented Programming & Java Language Constructs

Department of Computer Science
University of Maryland, College Park
Overview

- Object-oriented programming (OOP)
  - Introduction to OOP principles

- Java programming language
  - Review language constructs
  - Introduce new language constructs
    - Many from Java 5.0
Object-Oriented Programming (OOP)

- Approach to improving software
  - View software as a collection of objects (entities)

- Motivated by software engineering concerns
  - To be discussed later in the semester

- OOP takes advantage of two techniques
  - Abstraction
  - Encapsulation
Techniques – Abstraction

Abstraction
- Provide high-level model of activity or data

Procedural abstraction
- Specify what actions should be performed
- Hide algorithms

Data abstraction
- Specify data objects for problem
- Hide representation
Techniques – Encapsulation

Encapsulation

Confine information so it is only visible / accessible through an associated external interface

Approach

For some entity X in program

- Abstract data in X
- Abstract actions on data in X
- Collect data & actions on X in same location

Protects and hides X

Extension of abstraction
Abstraction & Encapsulation Example

Abstraction of a Roster

- **Data**
  - List of student names

- **Actions**
  - Create roster
  - Add student
  - Remove student
  - Print roster

Encapsulation

- Only these actions can access names in roster

<table>
<thead>
<tr>
<th>ROSTER</th>
<th>List of names</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>create( )</td>
</tr>
<tr>
<td></td>
<td>addStudent( )</td>
</tr>
<tr>
<td></td>
<td>removeStudent( )</td>
</tr>
<tr>
<td></td>
<td>print( )</td>
</tr>
</tbody>
</table>
Java Programming Language

Language constructs designed to support OOP

- **Example**
  - Interface – specifies a contract
  - Class – implements/defines contracts, supports encapsulation of implementation

Class libraries designed using OOP principles

- **Example**
  - Java Collections Framework
  - Java Swing
Java Interface

- An Interface defines a contract
  - Collection of
    - Constants
    - Abstract methods; no implementations
  - Can not be instantiated

- Classes can **implement** interfaces
  - Must implement all methods in interface
  - Example
    ```java
class Foo implements Bar { … }
```

- Similar to abstract class
  - But class can “inherit” from multiple interfaces
Java Collections Framework

- **Collection**
  - Object that groups multiple elements into one unit
  - Also called container
  - Example: ArrayList

- **Collection framework** consists of
  - Interfaces
    - Abstract data type
  - Implementations
    - Reusable data structures
  - Algorithms
    - Reusable functionality

- **Collection – Java Interface**
  - See Java API entry for Collection
  - Example (CollectionExample.java)
Overview

- Object-oriented programming (OOP)
  - Introduction to OOP principles

- Java programming language
  - Review language constructs
  - Introduce new language constructs
    - Many from Java 5.0
Review of Java Language Constructs

- **Basic elements**
  - Primitive types, variables, constants, operators
  - If-else, switch, while, for

- **Classes**
  - Object instances
    - Creating objects with `new`
  - Object references
    - The null reference
  - Instance data, class (static) data
  - Methods
    - Parameters, return values, polymorphism
Review of Java Language Constructs

- Inheritance
  - Base class, derived class, super
  - Method overriding (vs. overloading)
  - Abstract methods
  - Up- and down-casting, getClass(), instanceof
    - avoid overuse of these... leads to bad designs
- Interfaces
- 1D Arrays
  - Creating, indexing
- Exceptions
  - Try-catch blocks
New Java Language Constructs

- Autoboxing
- Enumerated types
- Generics
- Enhanced for loop
  - Iterator interface
- Stream input & output
- Scanner class
Enumerated Types

- New type of variable with set of fixed values
  - Establishes all possible values by listing them
  - Supports values(), valueOf(), name(), compareTo()…
  - Can add fields and methods to enums

Example

```java
public enum Color { Black, White } // new enumeration
Color myC = Color.Black;
for (Color c : Color.values()) System.out.println(c);
```

When to use enums

- Natural enumerated types – days of week, phases of the moon, seasons
- Sets where you know all possible values
Enumerated Types

From "Taming the Tiger" presentation by Joshua Bloch and Neal Gafter at Sun's 2004 Worldwide Java Developer Conference

```java
class Card implements Serializable {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
    private final Rank rank;
    private final Suit suit;
    private Card( Rank rank, Suit suit ) {
        this.rank = rank;
        this.suit = suit;
    }
    public Rank rank() { return rank; }
    public Suit suit() { return suit; }
    public String toString() { return rank + " of " + suit; }
}
```
Generics – Motivating Example

Problem

- Utility classes handle arguments as Objects
- Objects must be cast back to actual class
- Casting can only be checked at runtime

Example

class A { ... }
class B { ... }
List myL = new List();
myL.add(new A()); // Add an object of type A
...
B b = (B) myL.get(0); // throws runtime exception
    // java.lang.ClassCastException
Solution – Generic Types

Generic types

- Provides abstraction over types
- Can parameterize classes, interfaces, methods
- Parameters defined using <X> notation

Examples

- public class foo<X, Y, Z> { … }
- List<String> myNames = ...

Improves

- Readability & robustness

Used in Java Collections Framework
Generics – Usage

- Using generic types
  - Specify <type parameter> for utility class
  - Automatically performs casts
  - Can check class at compile time

- Example

```java
class A { ... }
class B { ... }
List<A> myL = new List<A>();
myL.add(new A());       // Add an object of type A
A a = myL.get(0);       // myL element ⇒ class A
...
B b = (B) myL.get(0);   // causes compile time error
```
Generics – Issues

Generics and subtyping

- Even if class A extends class B
- List<A> does not extend List<B>

Example

class B { ... }
class A extends B { ... } // A is subtype of B
B b = new A(); // A used in place of B
List<A> aL = new LinkedList<A>();
List<B> bL = aL; // compile time error

Why?
Subtyping and generic types

Consider what could happen if legal

```java
class B { ... }
class A extends B { ... }  // A is subtype of B
B b = new A();  // A can be used where B expected
List<A> aL = new LinkedList<A>();
List<B> bL = aL;
bL.add(b);
A a = aL.getFirst(); // runtime exception
```

Example (subtyping package)
Subtyping and Arrays

Subtyping works for arrays

```java
class B { ... }
class A extends B { ... }  // A is subtype of B
B b = new A();            // A can be used where B expected
A[] aA = new A[1];
B[] bA = aA;
aA[0] = b;  // won't compile
bA[0] = b;  // get runtime exception
```

Arguably a mistake
Autoboxing & Unboxing

Automatically convert primitive data types

- Data value ↔ Object (of matching class)
- Data types & classes converted
  - Boolean, Byte, Double, Short, Integer, Long, Float

Example

```java
ArrayList<Integer> myL = new ArrayList<Integer>();
myL.add(1);  // previously myL.add(new Integer(1));
int y = mL.getFirst();  // previously int y = mL.getFirst().intValue();
```

Example (SortValues.java)
Comparable Interface

Comparable

- public int compareTo(Object o)
- A.compareTo(B) returns
  - Negative if A < B, 0 if A == B, positive if A > B

Properties

- Referred to as the class's *natural ordering*
- Can sort using Collections.sort() & Arrays.sort()
  - Example: Collections.sort(myList);
- Can use as keys in SortedMap & SortedSet
- Consistency w/ equals() strongly recommended
  - x.equals(y) if and only if  x.compareTo(y) == 0

Example (comparableExample)
Comparator Interface

Comparator

- public int compare(Object A, Object B)
  - Negative if A < B, 0 if A == B, positive if A > B

Properties

- Imposes total ordering on objects of a class
- Provide alternatives to natural ordering
- Supports generics
  - Example: class myC implements Comparator<Foo>{ … }
- Use as parameter for sort function
  - Example: Collections.sort(myFooList, new myC( ));

Example (comparatorExample)
Iterator Interface

**Iterator**
- Common interface for all Collection classes
- Used to examine all elements in collection

**Properties**
- Can remove current element during iteration
- Works for any collection
Iterator Interface

Interface

```java
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove(); // optional, called once per next()
}
```

Example usage

```java
Iterator i = myCollection.iterator();
while (i.hasNext()) {
    myCollectionElem x = (myCollectionElem) i.next();
}
```
Enhanced For Loop

- Works for arrays and any class that implements the `Iterable` interface, including all Collections
  - Has method `iterator()` returns `Iterator<T>` object
  - For loop handles Iterator automatically
    - Test `hasNext()`, then invoke `next()`
- Example 1 // Iterating over a String array

```java
String[ ] roster = {"John", "Mary", "Alice", "Mark"};
for (String student : roster)
    System.out.println(student);
```
**Example 2**

```java
ArrayList<String> roster = new ArrayList<String>();
roster.add("John");
roster.add("Mary");

// using an iterator
for (Iterator<String> it = roster.iterator(); it.hasNext(); )
    System.out.println(it.next());

// using for loop
for (String student : roster)
    System.out.println(student);
```
2-D Arrays of Primitives

- Each row in two-dimensional array is an array
- Rows can have different lengths
- Defining a primitive array where rows have the same length
  
  ```
  int [ ][ ] data = new int[3][4];
  ```

- Defining a primitive data array where rows have different lengths (ragged array)
  
  ```
  int [ ][ ] ragged = new int[2][ ];
  ragged[0] = new int[3];
  ragged[1] = new int[1];
  ```
2-D Arrays of Objects

- Each row in two-dimensional array is an array
- Rows can have different lengths
- Defining an array where rows have the same length

```java
String[][] data = new String[3][4];
```

- Important – Note we have created a 2-D array of references to String objects; no String objects yet exist
- Can also create ragged arrays of objects
- Example (Roster.java)