

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**
 1. **Abstraction**
 2. **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

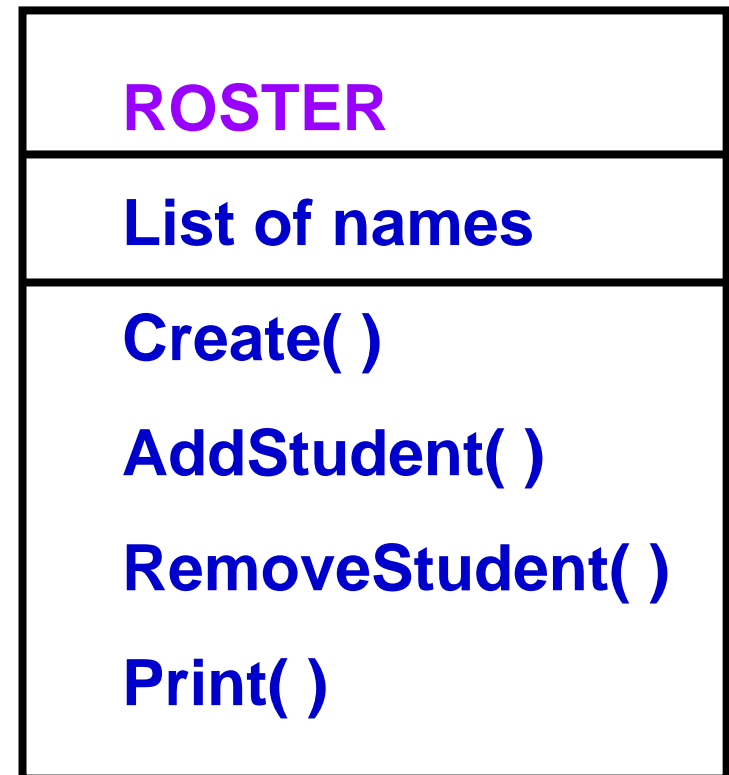
- List of student names

■ Actions

- Create roster
- Add student
- Remove student
- Print roster

■ Encapsulation

- Only these actions can access names in roster



Java Programming Language

■ Language constructs designed to support OOP

■ Example

- Interface – supports procedure abstraction
- Class – supports encapsulation

■ Class libraries designed using OOP principles

■ Example

- Java Collections Framework
- Java Swing

Java Interface

■ Interface

- Collection of

 - Constants

 - Abstract methods

- Can not be instantiated

■ Classes can **implement** interface

- Must implement **all** methods in interface

- Example

 - `class foo implements bar { ... } // interface 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

■ Collection **framework** consists of

- Interfaces
 - Abstract data type
- Implementations
 - Reusable data structures
- Algorithms
 - Reusable functionality

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**
- 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**
- **Annotations**
- **BitSet class**

Autoboxing & Unboxing

■ Automatically convert primitive data types

- Data value \Leftrightarrow Object (of matching class)

- Data types & classes converted

 - Boolean, Byte, Double, Short, Integer, Long, Float

■ Example

```
ArrayList myL = new ArrayList();
```

```
myL.add(1);      // previously myL.add(new Integer(1));
```

```
Integer X = new Integer(2);
```

```
int y = X;      // previously int y = X.intValue();
```

Also see example in `SortValues.java`

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**

```
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

```
public 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> { ... }`
- `public class List<String> { ... }`

■ 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

```
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

■ Example

```
class B { ... }  
class A extends B { ... } // A is subtype of B  
B b = new A(); // A used in place of B  
List<B> myL = new List<A>(); // compile time error  
// List<A> used in place of List<B>  
// List<A> is not subtype of List<B>
```

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
 - If `x.equals(y)`, then `x.compareTo(y) == 0`

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());`

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

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

■ Example usage

```
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
 - Has method `iterator()` returns `Iterator<T>` object
- For loop handles `Iterator` automatically
 - Test `hasNext()`, then get & cast `next()`
- **Example 1 // Iterating over a String array**

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

Enhanced For Loop

■ Example 2

```
ArrayList<String> roster = new ArrayList<String>( );  
roster.add("John");  
roster.add("Mary");  
Iterator it = roster.iterator( ); // using an iterator  
while (it.hasNext( ))  
    System.out.println(it.next( ));  
for (String student : roster) // using for loop  
    System.out.println(student);
```

Stream Input/Output

■ Stream

- A connection carrying a sequence of data (ordered sequence of bytes)

■ Streams can be associated with

- Files, memory, other Strings

■ Many Java classes for handling streams

- Data consisting of characters (e.g., text files)
- Data consisting of raw bytes (e.g., binary files)
- Can buffer information

■ Combining different classes

- Can define stream with desired characteristics

Using Streams

■ Opening a stream

- Connects program to external data
- Location of stream specified at opening
- Only need to refer to stream

■ Usage

1. `import java.io.* ;`
2. Open stream connection
3. Use stream → read and / or write
 - Catch exceptions if needed
4. Close stream

■ Examples

- See `fileExamples` package

Standard Input/Output

■ Standard I/O

- Provided in System class in java.lang

- System.in

 - An instance of InputStream

- System.out

 - An instance of PrintStream

- System.err

 - An instance of PrintStream

Scanner Class

■ Scanner

- Read primitive types & strings from input stream
 - Including `System.in` (standard input)
- Provides methods to treat input as `String`, `Integer`...
- Supports `String nextLine()`, `int nextInt()`...
- Throws `InputMismatchException` if wrong format

Scanner Class Examples

■ Example 1

// old approach to scanning input

```
BufferedReader br = new BufferedReader( new  
    InputStreamReader(System.in));
```

```
String name = br.readLine( );
```

// new approach using scanner

```
Scanner in = new Scanner(System.in);
```

```
String name = in.nextLine( ); int x = in.nextInt( );
```

■ Example 2

- **See ScannerExample.java**

- **Note use of printf**

Annotations

- **Annotation – Java construct that allow us to add validity constraints to Java Classes**
- **Validity constraint example**
 - **A instance variable cannot assume a negative value**
 - **A parameter can not be null**
 - **A method in a class must override a method in its superclass**
- **Syntax**
 - **at-sign (@) followed by annotation type and a parenthesized list of element-value pairs**
- **Example**
 - **`@DefaultAnnotationForParameters(NonNull.class)`**
- **You can ignore annotations in code distribution for class projects**

Reviewing Bit-Operations

■ Java Bitwise operators

- & and
- | or
- ^ exclusive or (xor)
- ~ complement

■ and

- x 11010
- y 10110
- x & y 10010

■ or

- x 11010
- y 10110
- x | y 11110

■ xor

- x 11010
- y 10110
- x ^ y 01100

BitSet Class

- Implements a set of bits where the bits of the set are indexed by nonnegative integers

- **Methods**

- **BitSet()** – New bit set
- **BitSet(int nbits)** – Bit set large enough to represent bits with indices from 0 through nbits – 1
- **and(BitSet set)** – Performs logical **and** between the current object and the set parameter (current object is updated with the result)
- **or(BitSet set)** – Performs logical **or** between the current object and the set parameter (current object is updated with the result)
- **cardinality()** – Returns number of bits set to 1
- **flip(int bitIndex)** – Sets the bit at the specified index
- **get(int bitIndex)** – Returns true if the bit at bitIndex is set; false otherwise
- **length()** – Index of the highest set bit + 1. It returns zero if the BitSet contains no bits set.
- **size()** – Number of bits space used by the BitSet to represent bit values
- **toString()** – For every bit set, the decimal representation of that index is included in the result.

- **Example (See Computers.java)**

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

```
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 (See Roster.java)