CMSC 132:
Object-Oriented Programming II

Object-Oriented Programming & Java Language Constructs

Department of Computer Science

University of Maryland, College Park

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

- You can create your own type with a finite number of values:

```
public enum Color { Black, White } // new enumeration
Color myC = Color.Black;
```

- New type of variable with set of fixed values
  - Supports values(), valueOf(), name(), compareTo()…
  - Can add fields and methods to enums
- When to use enums
  - Sets where you know all possible values

- EXAMPLE: EnumerationExample

Generics – Motivating Example

- **Before Generics…**
  - **Collections using Object class:**
  - **List x = new ArrayList();**
  - **x.add(new Foo());**
  - **Foo f = (Foo) x.get(0);**
    - **Objects must be cast back to actual class**
- **Problem:**

```
x.add(new Bar());
Foo f = (Foo) x.get(1);  // compiles, but…
                         // throws ClassCastException
```

Solution – Generic Types

- **Generic types**
  - **List<Foo> x = new ArrayList<Foo>();**
  - **x.add(new Bar()); // won't compile**
- **Improves**
  - **Readability & robustness**
- **Used in Java Collections Framework**

Autoboxing & Unboxing

- **Recall: Wrapper classes available for primitives.**
- **Java will automatically convert back-and-forth:**

```
List<Integer> a = new ArrayList<Integer>();
a.add(72);              // auto-boxing
int x = a.get(0);       // auto-unboxing
```

Also see example in 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***
  - **Used by Collections.sort( ) & Arrays.sort( )**
  - **Will be used implicitly in certain Collections**
  - **Consistency w/ equals( ) strongly recommended**
    - **x.equals(y) if and only if x.compareTo(y) == 0**
- **Also see Example: ComparableExample**

Comparator Interface

- **Comparator**
  - Use to define orderings beyond the "natural order"
  - Write a separate class for each ordering
  - Classes implement the Comparator Interface:
    - int compare(Object a, Object b)
- **Properties**
  - Supports generics
    - Example: class myC implements Comparator<Foo>{ … }
  - Used in many places in Collections Framework:
    - Example: Collections.sort(myFooList, new myC( ));
- **EXAMPLE: ComparatorExample**

Iterator Interface

- **Iterator**
  - **Common interface for all Collection classes**
  - **Used to process 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: IteratorExample**

Iterable Interface

- **Includes just one prototype:**

  Iterator<T> iterator();

- **Most collections in the Java Collections Framework implement Iterable**

Enhanced For Loop

- **Works for arrays and any class that implements the Iterable interface, including all Collections**

- **For loop handles Iterator automatically**

EXAMPLE: IterableExample

Enhanced For Loop

- **Also works with arrays:**

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

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

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)