© Department of Computer Science UMD

CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Object-Oriented Programming Intro

Department of Computer Science
University of Maryland, College Park

© Department of Computer Science UMD

Object-Oriented Programming (OOP)

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

- OOP takes advantage of two techniques
  - Abstraction
  - Encapsulation

© Department of Computer Science UMD

Techniques – Abstraction

- **Abstraction**
  - Provide high-level model of activity or data
  - Don't worry about the details. What does it do, not how
  - *Example from outside of CS*: Microwave Oven

- **Procedural abstraction**
  - Specify what actions should be performed
  - Hide algorithms
  - *Example:* Sort numbers in an array (is it Bubble sort? Quicksort? etc.)

- **Data abstraction**
  - Specify data objects for problem
  - Hide representation
  - *Example:* List of names

- **Abstract Data Type (ADT)**
  - Implementation independent of interfaces
  - *Example:* The ADT is a map (also called a dictionary). We know it should associate a key with a value. Can be implemented in different ways: binary search tree, hash table, or a list.

© Department of Computer Science UMD

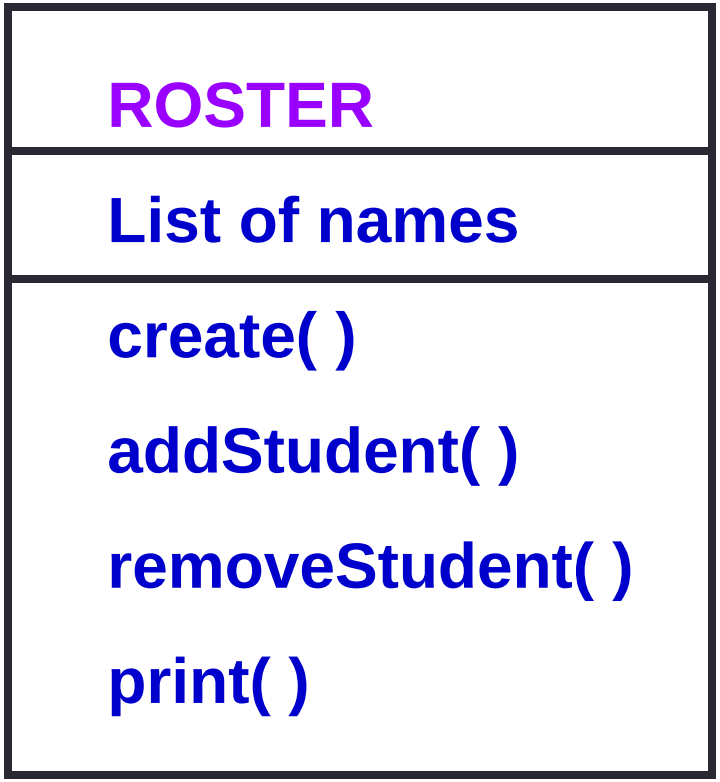
Techniques – Encapsulation

- Encapsulation
  - **Definition:** A design technique that calls for hiding implementation details while providing an interface (methods) for data access
  - Example: use the keyword **private** when designing Java classes
  - Allow us to use code without having to know its implementation (supports the concept of abstraction)
  - Simplifies the process of code modification and debugging
    - You can make changes to your code without breaking code of others that are using your class. Change the internals all you want, but just keep the interface constant

© Department of Computer Science UMD

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



© Department of Computer Science UMD

Java Programming Language

- Language constructs designed to support OOP
  - **Interfaces**
    - Specifies a contract. Allows us to express an ADT. What should it do, not how
    - Provides abstract methods (*usually no implementation*)
    - Defines an IS-A relationship
  - **Class**
    - Blue print for an object
      - Object – instance of a class
    - Can be used to ***implement*** an interface (How will it do what the interface promised)
    - Classes can ***extend*** other classes
      - Allows new class to inherit from original class
      - Defines an IS-A relationship

© Department of Computer Science UMD

Review on Interfaces

- Defines a new reference type
- Represents an API (Application Programming Interface)
- Can not be instantiated (you can only create an instance of a class that implements the interface)
- An Interface can contain the following public members:
  - static final constants
  - abstract methods (no body)
  - default methods (with code in the body) – added in Java 8 to support backward compatibility
  - static methods
  - static nested types
- **Example:** animalExample package

© Department of Computer Science UMD

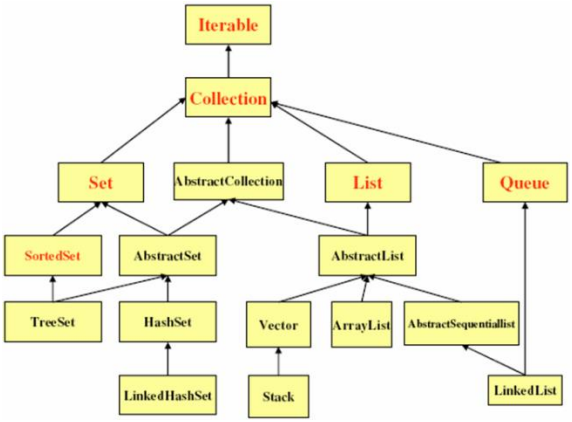
Java Collections Framework

- **Collection**
  - Object that groups multiple elements into one unit
  - Also called container
  - An example of a collection you used in CMSC 131 is an ArrayList (nice array ☺)
- Java Collections Framework (JCF) consists of
  - Interfaces
  - Implementations

© Department of Computer Science UMD

Java Collections Framework

- **Collection** → Java Interface
  - See Java API entry for Collection
    - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html
    - **Example:** CollectionExample.java



Interface (red)
Class (black)

- **Collections** → Class
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collections.html

© Department of Computer Science UMD

Generics (Motivating Example)

- Problems before Generics (Introduced in Java 5)
  - 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 ArrayList();   //raw type
myL.add(new A());             // Add an object of type A
…
B b = (B) myL.get(0);      // throws runtime exception
                           // java.lang.ClassCastException
```

© Department of Computer Science UMD

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

© Department of Computer Science UMD

Generics (Usage)

- Using generic types
  - Specify <type parameter> when creating an instance
  - Automatically performs casts
  - Can check class at compile time
- Example

```
class A { … }
class B { … }
List<A> myL = new ArrayList<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
```

Example: ArrayListExample.java

© Department of Computer Science UMD

Autoboxing & Unboxing

- Automatically convert primitive data types
  - Data value ⇔ Object (of matching class)
  - Wrapper Classes:
    - Character, Boolean, Byte, Double, Short, Integer, Long, Float
- Example

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

Example: SortValues.java

© Department of Computer Science UMD

Iterable and Iterator Interfaces

- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Iterable.html
- Allows you to use enhanced for loop (see next slide)
- Note that it only has one mandatory method that needs an implementation:
  - `Iterator<T> iterator()`
- So what is an *Iterator*? Another interface
  - See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Iterator.html
- Note that it only has two mandatory methods that need an implementation:
  - `boolean hasNext(); // true if there is another element`
  - `E next(); // returns the next element of type E`

© Department of Computer Science UMD

Iterable and Iterator Interfaces

- All Java Collection classes are iterables (**note a Map is not a collection**). Therefore, you can call the iterator method to get an Iterator and use an enhanced for loop to visit elements in the collection
- Example:

```
ArrayList<String> L = new ArrayList<String>();
L.add("Mary");
L.add("Pete");
Iterator<String> i = L.iterator();
while (i.hasNext())
  System.out.println(i.next());
```

- We will make classes that implement **Iterator** later in the course. For now, we just use the ones in the JCF

© Department of Computer Science UMD

Enhanced For Loop

- Works for arrays and any class that implements the Iterable interface, including all collections
  - Recall that iterables have an **`iterator()`** method that returns an **`Iterator<T>`** object
- Enhanced for loop handles Iterator automatically
  - Test **`hasNext()`**, then invoke **`next()`**
- /* Iterating over a String array */

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

© Department of Computer Science UMD

Enhanced For Loop

```
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);
```