CS1007: Object Oriented Design and Programming in Java

Lecture #3
T 9/13
Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

• Feedback
• Scopes
• Static
• Method Overloading
• Exception handling
• Basic classes
• Constructors
• Useful tools
Feedback

• More clarification on THIS

• Practical java examples

• More use of laptop screen for examples

Announcements

• Homework 1 out
  – Start early
  – If you are having problems…you probably have not done HW0
  – Due Sept 27 midnight
This

class student{
    Date dateofBirth;
    int idNumber;

    public void foo(){
       this.dataofBirth
    }

}

Local Variables

• Variables declared within a method are local to that method
  – Local scope
• Variables declared within a class, are called field variables
• Local variable can have the same name as field variables
  – Use this to disambiguate
Instantiated vs static

• When you define a method in a class, every instance of the class has its own copy.

• static methods allows one copy to be accessed by all instances
  – So……..what parts of the class should it be able to access?

Static Fields

• Shared among all instances of a class
• Example: shared random number generator

```java
public class Greeter {
    . . .
    private static Random generator;
}
```

• Example: shared constants

```java
public class Math {
    . . .
    public static final double PI = 3.14159265358979323846;
}
```
Static Methods

- Don't operate on objects
- Example: Math.sqrt
- Example: factory method

```java
public static Greeter getRandomInstance()
{
  if (generator.nextBoolean()) // note: generator is static field
    return new Greeter("Mars");
  else
    return new Greeter("Venus");
}
```

- Invoke through class:

```java
Greeter g = Greeter.getRandomInstance();
```

- Static fields and methods should be rare in OO programs

Pass around

- Can in theory use static variables to pass around values between class instances
- When is this good?
- Why?
- Why Not?
Methods

• Methods are defined by their signatures
  – Return values
  – Arguments values

  public void foo()
  public int foo()

Method Overloading

• We can define two methods with the same name, as long as they have different signatures
  – Different input parameters
  or/and
  – Different return values

  Java will know which one to use
Exceptions

• Object that represents an unusual event or an error
• Attempt to divide by zero
• Array out of bounds
• Null reference

Exception Handling

• Example: NullPointerException

String name = null;
int n = name.length(); // ERROR

• Cannot apply a method to null
• Virtual machine throws exception
• Unless there is a handler, program exits with stack trace

Exception in thread "main" java.lang.NullPointerException
at Greeter.sayHello(Greeter.java:25)
at GreeterTest.main(GreeterTest.java:6)
Checked and Unchecked Exceptions

- Compiler tracks only checked exceptions
- NullPointerException is not checked
- IOException is checked
- Generally, checked exceptions are thrown for reasons beyond the programmer’s control
- Two approaches for dealing with checked exceptions
  - Declare the exception in the method header (preferred)
  - Catch the exception

Declaring Checked Exceptions

- Example: Opening a file may throw FileNotFoundException:

```java
public void read(String filename) throws FileNotFoundException
{
    FileReader reader = new FileReader(filename);
    ...
}
```

- Can declare multiple exceptions

```java
public void read(String filename)
throws IOException, ClassNotFoundException
public static void main(String[] args)
throws IOException, ClassNotFoundException
```
Catching Exceptions

try
{
    code that might throw an IOException
}
catch (IOException exception)
{
    take corrective action
}

• Corrective action can be:
  – Notify user of error and offer to read another file
  – Log error in error report file
  – In student programs: print stack trace and exit

exception.printStackTrace();
System.exit(1);

The `finally` Clause

• Cleanup needs to occur during normal and exceptional processing
• Example: Close a file

FileReader reader = null;
try
{
    reader = new FileReader(name);
    ...
} catch.....
finally
{
    if (reader != null) reader.close();
}
Strings

• Sequence of Unicode characters
  – (Technically, code units in UTF-16 encoding)
• length method yields number of characters
• "" is the empty string of length 0, different from null
• Special class in Java
  – Assigning a string literal to a string reference creates an instance!
• charAt method yields characters:
  char c = s.charAt(i);

String II

• substring method yields substrings:
• "Hello".substring(1, 3) is "el"
• Use equals to compare strings
  if (greeting.equals("Hello"))
• == only tests whether the object references are identical:
  if ("Hello".substring(1, 3) == "el") ... // NO!
String concatenation

- `+` operator concatenates strings:
  - "Hello, " + name
- If one argument of `+` is a string, the other is converted into a string:
  ```java
  int n = 7;
  String greeting = "Hello, " + n;
  // yields "Hello, 7"
  ```
- `toString` method is applied to objects
  ```java
  Date now = new Date();
  String greeting = "Hello, " + now;
  // concatenates now.toString()
  // yields "Hello, Wed Jan 17 16:57:18 PST 2001"
  ```

Converting Strings to Numbers

- Use static methods
  - `WHY???
    - `Integer.parseInt`
    - `Double.parseDouble`
- Example:
  ```java
  String input = "7";
  int n = Integer.parseInt(input);
  // yields integer 7
  ```
- **NOTE:**
  - If string doesn't contain a number, throws a NumberFormatException (unchecked)
Reading Input

- # Construct Scanner from input stream (e.g. System.in)
- Scanner in = new Scanner(System.in)
- # nextInt, nextDouble reads next int or double
- int n = in.nextInt();
- # hasNextInt, hasNextDouble test whether next token is a number
- # next reads next string (delimited by whitespace)
- # nextLine reads next line

Example

01: import java.util.Scanner;
02: 
03: public class InputTester
04: {
05:     public static void main(String[] args)
06:     {
07:         Scanner in = new Scanner(System.in);
08:         System.out.print("How old are you?");
09:         int age = in.nextInt();
10:         age++;
11:         System.out.println("Next year, you'll be " + age);
12:     }
13: }
The ArrayList<E> class

- Generic class: ArrayList<E> collects objects of type E
- E cannot be a primitive type
- add appends to the end

```java
ArrayList<String> countries = new ArrayList<String>();
countries.add("Belgium");
countries.add("Italy");
countries.add("Thailand");
```

II

- get gets an element; no need to cast to correct type:
  ```java
  String country = countries.get(i);
  ```
- set sets an element
  ```java
  countries.set(1, "France");
  ```
- size method yields number of elements
  ```java
  for (int i = 0; i < countries.size(); i++) . . .
  ```
- Or use "for each" loop
  ```java
  for (String country : countries) . .
  ```
Arrays drawback

- Can insert and remove elements in the middle
  
  countries.add(1, "Germany");
  countries.remove(0);

- Not efficient--use linked lists if needed frequently

Linked List

- What?
  - Efficient insertion and removal

- add appends to the end

  LinkedList<String> countries = new LinkedList<String>();
  countries.add("Belgium");
  countries.add("Italy");
  countries.add("Thailand");

- Use Listiterators to edit in the middle
  - Iterator points between list elements
List Iterators

- next retrieves element and advances iterator
  ```java
  ListIterator<String> iterator = countries.listIterator();
  while (iterator.hasNext()) {
    String country = iterator.next();
    . . .
  }
  ```
- Or use "for each" loop:
  ```java
  for (String country : countries)
  ```
- add adds element before iterator position
- remove removes element returned by last call to next

Arrays

- Drawback of array lists: can't store numbers in a simple manner
- Arrays can store objects of any type, but their length is fixed
  ```java
  int[] numbers = new int[10];
  ```
- Array variable is a reference

```
ref
```
Arrays

• Array access with [] operator:
  int n = numbers[i];
• length member yields number of elements

  for (int i = 0; i < numbers.length; i++)

• Or use "for each" loop
  for (int n : numbers)

Arrays

• # Can have array of length 0; not the same as null:

  • numbers = new int[0];
  • # Multidimensional array

  • int[][] table = new int[10][20];
  • int t = table[i][j];
main

• The main method is declared public, static and void.
• Because it is static we often need to create an instance of the class inside its own main.
• Why?

main

• Every class can have a main method. If you five classes, with each one having a main, you need to tell java which one to run…
• How is this done?
• Can also use individual mains as testing areas, will be ignored when not run
Default Values

- By Default java assigns the following values:
  - boolean false
  - char 0
  - byte, int 0
  - float +0.0F
  - double +0.0
  - reference null

Constructor

- A constructor is a method that gets called when an object is created using `new`.
- We can use the constructor to initialize the fields of the object.
- A constructor can have as many parameters as necessary, but can not have a return type.

```java
public class Moo
{
    private int x;

    public Moo(int x)
    {
        this.x = x;
    }
}
```
Default Constructor

• If we don’t define a constructor the default constructor with no parameters will be created.

• So we can say:
  Moo m = new Moo();

• Like other methods, the constructor can also be overloaded.
• Can call one constructor from another
  – this(something);
  – Must be the first statement in the method

Remember

• Object: Three characteristic concepts
  – State
  – Behavior
  – Identity

• Class: Collection of similar objects
Program Design

• Analysis  
• Design  
• Implementation

Analysis Phase

• Functional Specification
  – Completely defines tasks to be solved  
  – Free from internal contradictions  
  – Readable both by domain experts and software developers  
  – Reviewable by diverse interested parties  
  – Testable against reality
Design Phase

• Goals
  – Identify classes
  – Identify behavior of classes
  – Identify relationships among classes

• Artifacts
  – Textual description of classes and key methods
  – Diagrams of class relationships
  – Diagrams of important usage scenarios
  – State diagrams for objects with rich state

Implementation Phase

• Implement and test classes
• Combine classes into program
• Avoid "big bang" integration
• Prototypes can be very useful