CS133: Developing Programming Principles

Lecture 10

Designing Classes

Testing methods

- All methods should be tested individually.
- If method A calls method B, test method B first and make it work properly. Then test method A. This is called **bottom-up testing**.

Driver program

- A simple program that calls a method multiple times with different arguments.
- Only used for testing.
- Includes “test suite” of test cases.
- Allows test cases to be repeated easily.

Testing

- All code should be tested to ensure it works properly.
- Testing code involves:
  - Choosing test cases
  - Running those tests
  - Comparing expected and actual outcomes
  - Modifying code
  - Repeating process with same test cases until satisfied code is “correct”

Example: choosing test cases

```
if (grade == 'A') {
    System.out.println("Excellent!");
} else if (grade == 'B') {
    System.out.println("Good");
} else if (grade == 'F') {
    System.out.println("Failed!");
} else {
    System.out.println("Passed");
}
```

- What possible values of grade should be tested?

Choosing test cases

- Try to cover all different situations, not just typical ones.
- Some guidelines:
  - **Numerical data** – smallest and largest possible, typical, invalid data.
  - **Boolean data** – true, false values.
  - **Character data** – numeric, alphabetic, special characters.
  - **`String` data** – empty **`String`**, **`String`** with one character, and **`String`** with many characters.
  - **Object data** – **`null`** and non-**`null`** values.

Driver program continued

- Example: Testing **`Math.abs(int a)`**

```
public class AbsoluteValueDriver {
  public static void main(String[] args) {
    //Test case 1: positive value
    int answer1 = Math.abs(10);
    System.out.println("Expected: 10, Actual: "+answer1);
    //Test case 2: negative value
    int answer2 = Math.abs(-25);
    System.out.println("Expected: 25, Actual: "+answer2);
    //Test case 3: zero
    int answer3 = Math.abs(0);
    System.out.println("Expected: 0, Actual: "+answer3);
  }
}
```


Top-down design

- The practice of dividing large tasks into smaller, more manageable ones.
- Also known as **stepwise refinement**.

Coding using top-down design

- Each major task identified should be implemented in a method.
- The "original" method will call the other helper methods (which may call other helper methods themselves).
- Helper methods should be **`private`** to their class.

Example

- Given non-negative ints a, b, let **`cat(a, b)`** be the concatenation of their digits
  - e.g., cat(20, 16) = 2016, cat(12, 0) = 120
- Write a method which takes an empty m x n **`Board`** and puts a peg on row r, column c if r or c are factors of **`cat(r, c)`**:
  - Add a yellow peg if one of r, c is a factor
  - Add a blue peg if both of r, c are factors
  - Do not consider factors of 0
- e.g., (11, 0) = yellow, (2,3) = none, (2, 2) = blue

Pseudocode

```
for each row r
   for each column c
      determine number of factors
      if one factor: put yellow peg
      if two factors: put blue peg
   end for c
end for r
```


Helper methods

How to break down code?

- When some step is confusing
- When your methods are getting long and unreadable
- When some subcomputation needs to be performed
- To avoid cuting-and-pasting

Stub

- Simplified version of a method.
- Method will not accomplish task, but will allow code to compile and run.
- Helpful when using top-down design.

Stubs continued

- When writing a stub for a method that returns a value, we must still include a return statement.
- Example:

```
public String reverseString(String s)
{  return "";
   // or return s;
   // or return null;
}
```

- Even a stub must return a value of the correct type.

Multiple constructors

- UW students have a major and may have an option as well.
- Write a class **`UWPlan`**. Include a constructor which sets major and option to provided values.

UWPlan class

```
public class UWPlan {
  private String major;
  private String option;
  // Constructors here
  public String getMajor() {
    return this.major;
  }
  public String getOption() {
    return this.option;
  }
}
```


Example continued

```
public class UWPlan {
  private String major;
  private String option;

  public UWPlan(String aMajor,
                String anOption) {
    this.major = aMajor;
    this.option = anOption;
  }
}
```

Example continued

- Many students have a major, but no option.
- Write another constructor which accepts one parameter for a major.
- It should set the option to be the empty string.

One solution

```
public UWPlan(String aMajor)
{
  this.major = aMajor;
  this.option = "";
}
```

A different solution

```
public UWPlan(String aMajor)
{
  this(aMajor, "");
}
```


Example continued

- Some students take some time before declaring their major.
- Write another constructor that takes no parameters.
- It sets the major to “Undeclared” and the option to be the empty string.

Testing constructors

```
public class UWPlan {
  // rest of class here
  public static void main (String[] args) {
    UWPlan bioinfo = new UWPlan
          ("CS", "Bioinformatics");
    System.out.println(
     "Expected major: CS, Actual major: "
     + bioinfo.getMajor());
    System.out.println("Expected option:"
     +" Bioinformatics, Actual option: "
     + bioinfo.getOption());
```


More tests

```
    UWPlan am = new UWPlan("Applied Math");
    System.out.println("Expected major:"
     + " Applied Math, Actual major: "
     + am.getMajor());
    System.out.println("Expected no option,"
     + " Actual option: " + am.getOption());
```

A third test

```
    UWPlan dontKnowYet = new UWPlan();
    System.out.println("Expected major:"
     + " Undeclared, Actual major: "
     + dontKnowYet.getMajor());
    System.out.println("Expected no option,"
     + " Actual option: "
     + dontKnowYet.getOption());
  }
}
```


Summary

- Testing
- Top-down design
- Overloading constructors

ERROR: undefined
OFFENDING COMMAND:

STACK: