JAVA PROGRAMMING: CLASS 4

Rewrite
Methods, APIs, Naming, and Single Responsibility Principle
Packages, Static variables, Imports, Exceptions
Hands-on
Overloading
Hands-on
Initialization
JavaDoc
Assignment 3 & Questions
• A for-loop slide was added to last week’s slides and posted.
• Conditions in “if” and “if-else-if” statements are expressions that can be arbitrarily long. The entire expression must return a boolean result.
• “if-else-if” can have an arbitrary number of “else-if” clauses.
• System.in is an InputStream. If you look at the API for this class, you’ll notice it doesn’t do much more than read raw characters from a data source, in this case the Console. It’s up to you, the programmer, to parse (separate) the input and figure out how to process it into useful program variables. That’s what TextIO and Scanner classes do. They provide read / write methods to simplify reading data that you expect in a particular format.
METHODS

• Methods are a key mechanism of modularizing your program.
• Methods should either compute something and return it or change something then return either nothing or an indication of the new state.
• We’ve already seen the signature and the potential full declaration of a method.
• Methods group units of code under a name that can be reused in other code. This is a crucial aspect of code re-use.
• Classes don’t have to have any methods, but they’re not too useful without them.
• Use ‘return;’ to exit a void method. Use ‘return <expression>’ where expression evaluates to a type that matches the method’s return type.
• Methods are a means of providing a “black box” that you access through its interface by calling it, but without worrying about how it’s implemented.
APIS

• Methods form the basis of an API (Application Programming Interface)
• APIs usage doesn’t require knowledge of how the code is implemented, just of how to call the methods in the API.
• The JavaDoc utility for documenting Java-based APIs will be discussed in a later slide.
• The Java Class Libraries are a large API: the entire Java 7 API is over 200 packages with 4000 classes (per the book).
PASS BY VALUE EXAMPLES

- See programming examples
USE GOOD METHOD NAMES

Should be descriptive of what the method does

double squareRoot( int x )

vs

// a1285: compute the square root a given integer
double a1285( int x )

If you feel the need to add a comment, first ask whether the method is named well. Rename the method instead of adding a comment, it’s easier to read.
SINGLE RESPONSIBILITY PRINCIPLE

• Methods should do one coherent thing at one level of abstraction.
• The print primes up to N method does one coherent thing, but it does it at multiple levels of abstraction.
• We’ll see this principle again when we talk about class design principles.
ABSTRACT ALGORITHM

Print all prime numbers up to a given value of N: Abstract Algorithm

Start with a candidate prime, K at 2 and increment it until we reach N:
  For each value of K:
    Check for a divisor, D, that evenly divides K and is between 2 and the square root of K. If not found, print K.
MULTIPLE RESPONSIBILITY (1ST TRY)

public class MultipleResponsibility {
    void printPrimesUpToN(int n) {
        for (int k = 2; k <= n; k++) {
            int squareRoot = (int) Math.sqrt(k);
            boolean prime = true;
            int j = 2;
            while (prime && j <= squareRoot) {
                prime = (k % j++ != 0);
            }
            if (prime) {
                System.out.println(k);
            }
        }
    }
}

(1) Check each number to see if it’s prime
(2) For a particular k, check for divisors up to its square root
(3) Check if a given number is a divisor of k
(4) Print k if its prime
void printPrimesUpToN(int limit) {  
    for (int k = 2; k <= limit; k++) { // (1)  
        printIfPrime(k);  
    }  
}  

private void printIfPrime(int candidate) {  
    if (isPrime(candidate)) {  
        System.out.println(candidate); // (4)  
    }  
}  

private boolean isPrime(int candidate) {  
    assert candidate >= 2;  
    int squareRoot = (int) Math.sqrt(candidate);  
    return candidate == 2 || !hasDivisorFromTwoToLimit(candidate, squareRoot); // (2)  
}
private boolean hasDivisorFromTwoToLimit(int candidate, int limit) {
    assert candidate > limit && limit >= 2;
    for (int i=2; i <= limit; i++) {
        if (isDivisibleBy( candidate, i )) {
            return true;
        }
    }
    return false;
}

private boolean isDivisibleBy(int value, int possibleDivisor) {
    assert value >= 1 && possibleDivisor >= 1;
    return value % possibleDivisor == 0;
}
SINGLE RESPONSIBILITY PAYS DIVIDENDS

• Smaller methods are easier to read and comprehend.
• Methods tend to be more general than the original, e.g. isPrime(), hasDivisorFromTwoToLimit(), and isDivisibleBy().
• “More general” translates into “more (re)usable”: if you only had the multiple responsibility example available, you’d have to dissect it (usually, cut and paste) to make use of isPrime() and isDivisibleBy() if you needed them elsewhere. Once you have more than one copy, the algorithm becomes harder to maintain.
PACKAGES

• Packages group related, often inter-dependent classes in a logical manner.
• Package names can be arbitrarily nested.
• For the assignments, prefix your package names with javaclass., then the package structure related to your program, e.g. javaclass.assignment1.
CORE JAVA PACKAGES

java.lang – no import statement required
java.util – Collections, String, Math, many others
java.net – Socket, ServerSocket, and more for TCP/IP-based communication
java.io, java.nio – Input/Output streams and many embellishments
java.awt, javax.swing – Java’s GUI toolkits
java.sql – Basic relational database access
// Imports every class in java.util
import java.util.*;

// Import just the HashMap class
import java.util.HashMap;

// Static imports import methods, not classes:
import static java.lang.Math.sqrt;

• A complete class name is: <packageName>.<className> If you need two classes with the same name from two different packages, you’ll have to pick one and use its complete name.

• Cmd/Ctrl –Shift-O will organize imports and import necessary classes if you declare a reference variable of a type that hasn’t yet been imported.

• If a name collision is found you can select the package you want to import the class from.

• Avoid the wildcard and let the IDE do the work for you. This reduces the likelihood of a collision.
COMPILE TIME: STATIC VS NON-STATIC MEMBER VARIABLES

• Static member variables belong to their associated Class. They are globally scoped.

• Non-static member variables belong to an instance of their associated class. We’ll discuss them in next week’s class.
STATIC VS NON-STATIC VISIBILITY

- Static and member variables are visible depending upon their access modifier.
- To access a static variable you can use `ClassName.variableName` in any class, provide the appropriate access modifier is set.
- But, to access a member variable, the access modifier must be set AND it you must have an object reference available if you access it from outside its class.
THROWING AN EXCEPTION

• Usage pattern: throw <exception object>
• For instance: throw new IllegalStateException(“File did not exist”);
• If you throw this exception and the caller of your method does not handle it, it will terminate your thread (and your program if there is just one thread).
• The caller can choose to catch it and recover, instead.
• See ThrowingAnException example.
HANDS-ON

• Write a main() method that opens a FileInputStream for a non-existent file name:
  
  ```java
  FileInputStream aStream = new FileInputStream(new File("file.txt"));
  ```

  • What happens?

• Declare main() to throw an IOException

  • What happens?

• Add a try – catch – finally block instead and in the catch block throw a new
  IllegalStateException("Bad filename given"). Why doesn’t the compiler complain?

  • What happens?

• Instead of throwing a new IllegalStateException, ask the user to enter a different
  file name.
METHOD OVERLOADING

- Method signature:

```java
public double getMyFundsFromBank(String bankName)
```
OVERLOADING

public double getFundsFromMyBank( String bankName ) {
  ...
}

public double getFundsFromMyBank( int id ) {
  ...
}

• Overloading is a form of compile-time polymorphism (‘many forms’) for methods.
• This is different than method overriding that we’ll cover in the next class.
• To overload, use the same method name, but a different length parameter name or the same length with different parameter types.
HANDS-ON

• Create a new class and make several overloaded static methods
• Chain them together in a sequence of calls that returns a value to main() where the value should be printed.
INITIALIZATION

- For loops: can declare the loop variable in the for statement
- Inside a method
  ```java
  int aMethod() {
    int y = 1;  // combination declaration with initialization
  }
  ```
- Outside a method:
  ```java
  public class StaticInitialization {
    static int f = 1;
    // static blocks can only init static member variables
    static {
      if ( f == 1 ) { f++; }
    }
  }
  ```
JAVADOC

Use /** ... */
Well-defined set of @annotations you can use
JavaDoc utility will generate your API documentation
Document public methods.
Especially important if you won’t be releasing source code to others.
Eclipse provides access to JavaDoc.
ASSIGNMENT 3 (DUE: FEB 19, 2015)

• Extend your solution to Assignment 2 to accept user input as follows:
  • Change the computation of all of row 12 and all of column 12 to display a color based on the same color formula as assignment 2, but apply it to the sum of the entire row or column (the other 11 elements in each). Set position (12, 12) to a color based on the sum of the rows or columns.
  • Ask the user to select a valid row and column other than within row 12 or column 12. If the inputs are invalid, output an error message and ask for input again.
  • Once you have a valid row and column entry, ask the user for a new value to be placed in the corresponding array element. Re-compute the color and redraw just the selected square whenever the color has changed.
  • If the user selects position (0, 0) then start a loop and update the array at a random column (1-11) and row (1-11) every 0.25 seconds for 60 seconds (for a total of 240 updates). Update the array value to a randomly selected integer between 1 and 121. After every array update, re-compute the 12th row/column values in the array and then redraw the grid.
  • Use: try { Thread.sleep(250); } catch(InterruptedException e) {} to pause for 0.25 seconds.