ICS 111
More about Methods

- Method design
- Stepwise refinement
- Method tracing
- Variable scope

ICS 111
Re-using Methods

- Code reuse is good for programmer efficiency and program correctness:
  - reusing an existing method means we don't have to write it
  - an existing method is less likely to have bugs than a newly-written method
- However, this is only possible if the method is sufficiently general

ICS 111
Method Generality

- There are many choices to be made when designing a method:
  - return type
  - name
  - parameters
  - design of the code
- the return type is often dictated by the computation we want the method to do
- the parameters may be flexible: some choices of parameters may make the method more general

ICS 111
Comparison of two Methods

```
public static void printHello() {
   System.out.println(“Hello world”);
}
public static void printGreeting(String greeting, String to) {
   System.out.println(greeting + “ “ + to);
}
```

- The second method can be reused for different greetings
- Making a method more general often leads to having more parameters
  - but not always!
  - more parameters make the method more complicated and harder to use
- Choice of parameters affects the generality of the method
- The name of the method has also changed to reflect its more general functionality

ICS 111
Method Design

- The method must solve your current needs
- Shorter methods (methods with shorter code) are better than longer methods
  - It's just fine to call other methods from within a method body
- Ideally, methods are units of meaning
  - when they are, they code in the caller is easy to read:

    ```
    name = digitName(number / 100) + “ hundred”;
    ```

    (example from the book, Section 5.7)
  - This turns part of a number (such as 321) into a string, such as “three hundred”

ICS 111
Method Design:
Stepwise Refinement

- Sometimes it's obvious how to break down a solution to a problem, by combining solutions to smaller problems
- when coding, each of the solutions to the smaller problems can be a method

ICS 111
Stepwise Refinement

- Doing an assignment includes:

1. Reading the assignment
2. Doing each of the programming problems
3. Turning in the assignment

- The method for doing step 2 is called more than once
- Now we can write the main method:

```
int numAssignment = 5;
int numProblems = readAssignment(numAssignment);
for (int i = 0; i < numProblems; i++) {
  solution += doProgrammingProblem(numAssignment, i + 1);
}
submitAssignment(numAssignment, solution);
```

ICS 111
Stepwise Refinement: Stubs

- Once the main problem has been subdivided into smaller, easier problems, we can write the methods to solve the smaller problems
- It is a good idea to test the top-level code before writing these lower-level methods
- If so, we can just define the lower-level method to do the minimum that allows the top-level method to still work
- This bare-bones implementation is called a **stub**

```
public static String doProgrammingProblem
   (int assignmentNumber, int problemNumber) {
   return “solution to problem “ + problemNumber + “\n”;
}
```

ICS 111
A real example

- One way to factor a number n is to divide it by every number x < n by which it is divisible
- Printing the factors requires remembering (in a variable) whether we have printed a factor before
  - if this is the first factor, just print it
  - otherwise, print “ * “ before the factor
- Both testing whether a number is divisible by another number, and printing the factor, can be delegated to other methods

ICS 111
Code for Factoring

```
public static void printFactors (int n) {
  int factor = 2;  // two is the first possible factor
  boolean firstPrint = true;
  System.out.println(n + “ = “);  // print the number to be factored
  while (factor <= n) {  // each loop, either increase factor, or make n smaller
    if (isDivisible(n, factor)) {
      printFactor(factor, firstPrint);    // print the factor
      firstPrint = false;         // we've printed one or more factors already
      n = n / factor;             // make n smaller
    } else {  // not divisible: maybe the next int is a factor
      factor++;                   // make factor bigger
    }
  }
  System.out.println();           // after the loop, end the line
}
```

- the two methods isDivisible and printFactor can initially be stubs while we test this code

ICS 111
isDivisible and printFactor stubs

-
  ```
  public static boolean isDivisible (int n, int factor) {
    return true;
  }
  ```

-
  ```
  public static void printFactor(int factor, boolean
  firsttime) {
    System.out.print((factor + “/” + firstTime + “ “);
  }
  ```
- now test the printFactors method:

  10 = 2/true 2/false 2/false
- the factors are wrong, but indeed 10 can be divided by 2, three times, before it is less than two

ICS 111
isDivisible method

- We can use modulo to test if a number n is divisible by another number factor
- If they are divisible, the remainder of the division should be zero

```
public static boolean isDivisible (int n, int factor) {
  return n % factor == 0;
}
```

- 10 = 2/true 5/false
- 100 = 2/true 2/false 5/false 5/false
- our printing isn't exactly what we want yet, but we can see that the results are correct

ICS 111
printFactor method

- printing is just a question of adding or not adding “ * “ before the factor

```
public static void printFactor(int factor, boolean firsttime) {
  System.out.print((firstTime ? “” : “ * “) + factor);
}
```

- and now, we can print the factors of any number!
- 2 = 2
- 10 = 2 * 5
- 100 = 2 * 2 * 5 * 5
- 33 = 3 * 11
- 31 = 31
- 30 = 2 * 3 * 5
- 12345 = 3 * 5 * 823

ICS 111
Summary of Stepwise Refinement

- If we have the high-level view of how to solve a problem, we can write the code for that high-level view
- Any components that we aren't ready to implement will initially be stub methods
- Testing with the stubs can give us confidence that the code for the high-level part is correct
- Once the main part is working for us, we go ahead and implement each stub
  - we test and correct any errors after implementing each stub
- Stepwise refinement makes it easier to identify any problems early, so we know where to look for the solution

ICS 111
Tracing Choices

- Suppose you are tracing this code:

```
if (isDivisible(n, factor)) {
  printFactor(factor, firstPrint);   // print the factor
  firstPrint = false;          // we've printed one or more factors already
  n = n / factor;              // make n smaller
} else {  // not divisible: maybe the next int is a factor
  factor++;                    // make factor bigger
}
```

- When you get to the first method call, what do you do?
  - You can enter the method, and trace the code of the method body
  - or, you can assume that the method does the right thing (return true or false, as appropriate) without going into the details
- Both of these methods of tracing are useful:
  - the first is useful for understanding how each method does what it does
  - the second is more useful (and faster) in understanding the top-level code

ICS 111
Tracing Individual Methods

- Treat parameters as you would variables
  - record their value, track these values when they change
- on a return, record the value returned

ICS 111
Variable Scope and Uniqueness

- We have seen that variables are in scope from their definition to the end of the enclosing block
- It is an error in Java to have two variables with the same name and overlapping scope
- It is OK to have variables with the same name as long as the scopes don't overlap

ICS 111
Uniqueness Examples

```
for (int i = 0; i < 10; i++) {
  for (int i = 77; i < 99; i++) {
```

- the second declaration of i is in the scope of the first and the compiler will complain
- Variables with different scopes:

```
for (int i = 0; i < 10; i++) {
}
for (int i = 77; i < 99; i++) {
}
```

- the two scopes don't overlap

ICS 111
Local and Global Variables

- variables in different methods can have the same name
- we say that variables are **local** to the method
  - as far as scoping is concerned, method parameters like local variables
- variables can also be declared outside methods: these are **global variables**
- global variables can be very useful, but are harder to use correctly, and for now you should not use global variables
  - once you do use them, choose the name carefully so it doesn't conflict with the names of other global variables

Summary

- Carefully designed methods are more likely to be reused
- In stepwise refinement, we create the high-level code first, using stubs for the lower-level methods
- This gives us confidence that the high-level code works, and that we have identified the correct lower-level methods
- In tracing, we can either go into method execution, or assume that methods do what we expect them to do
- Variable names must be unique within the scope of the variable
  - it is a good idea to give variables the smallest scope that still makes them useful