Methods

- Parameter Passing
- Scope
- Calling stack

Methods

- Why methods?
- Coding a method
- Calling method
- Method parameters
Procedural Decomposition

- Controlling complexity is a key concept in problem solving and software design
- We deal with complexity by breaking complex tasks into smaller, more manageable tasks
  - And smaller tasks may be divided into yet smaller tasks...
- For example, a recipe for cookies may be decomposed into
  - Preparation (preheat oven, grease pan)
  - Make the dough (measure, mix ingredients, blend)
  - Bake (Measure dough, bake, test, cool)
- The recipe is still ultimately a sequence of individual steps, but the higher level grouping allows us to more readily understand the overall process
  - Procedural decomposition focuses on the steps or actions
  - Object oriented decomposition focuses on the entities (later)

Building Blocks of Programs

- A program consists of many statements
- It is possible to code any logic using just if-else and loops
  - But for a large program, this would be tedious
  - There are likely to be a lot of duplications of logic
- E.g., computing the square root of a number takes a lot of steps
  - If a program needed to compute many square roots, duplicating all these steps would be error prone, hard to read and understand
- Better solution would be to have a “black box” to contain the square root logic
  - Then we could use the black box many times
  - ...and only look inside when we need to fix it or understand how it works
- Methods are the black boxes of programming
  - Similar to the idea of formulas in math
Java Methods

- A sequence of statements can be grouped together as a **method** (sometimes called a function or procedure)
  - We have already been doing that with `main`
  - We can define methods other than main
- Methods that we define may be **called**
  - Changes flow of control – jump to method, execute statements, return to point of call
  - Methods may be called repeatedly (*re-entrant* code)
- Definition of method
  - A method has a **name**
  - A method has a **return type**
  - A method has zero or more **parameters**
  - A method has a **body** (the statements)

Using Java Math methods

- All methods are static
- No import needed since class name is used, e.g.,
  ```java
double val = Math.random();
```  
- Could use static import:
  ```java
import static java.lang.Math.*;
```
  ```java
double val = random();
```  
- There are no Math objects, just static methods
  - Math functions: abs, max, min, sqrt, pow, exp, log
  - Trig functions: sin, cos, tan
  - Rounding functions: ceil, floor, round
  - Useful constants: PI, E
Java API Documentation

- Java classes are documented in the Application Programming Interface (API)
- Web pages, one page for each class
  - Over 7,000 pages!
  - Well indexed, with links between related classes
- For each class, API gives
  - Description of class, inheritance hierarchy
  - List of constructors – arguments, synopsis
  - List of all public methods – arguments, return value, synopsis
  - Sometimes example code
- Available on web at http://java.sun.com/javase/6/docs/api/
  - Math API page

Coding a Method Example

- Simulate coin tosses, ten at a time, until user decides to quit
  - Prompt to start
  - Prompt to continue after each ten tosses
  - Toss.java

- Code to prompt before loop and inside loop can be factored out into a method named prompt

  - Note that loop is terminated by program exiting from the prompt method
Reusing Code

- Blocks of almost identical code

```java
System.out.println("This is a simulation of flipping coins... 
System.out.println("--------------");
String response = scan.nextLine();
if (response.length() > 0)
    System.exit(0);
while (true) {
    ++count;
    if (Math.random() < 0.5) ++won;
    if (count % 10 == 0) {
        System.out.printf("After %d tosses, you have . . . 
System.out.println("--------------");
        String response = scan.nextLine();
        if (response.length() > 0)
            System.exit(0);
    }
}
```

Method Definition and Call

- Static method definition with no return value and no parameters

```java
public static void prompt() {
    // Code of the method
}
```

- Statement to call the method

```java
prompt();
```
Simplify with method call

- Define `prompt` method
  ```java
define prompt method
  public static void prompt() {
    Scanner scan = new Scanner(System.in);
    System.out.println("-----------");
    System.out.println("Press return to continue or Q to quit ");
    String response = scan.nextLine();
    System.out.print("Press return to continue or Q to quit ");
    System.exit(0);
  }

- Replace prompting code blocks with method call
  ```
  System.out.println("This is a simulation of flipping coins. . .
  while (true) {
    ++count;
    if (Math.random() < 0.5) ++won;
    if (count % 10 == 0) {
      System.out.printf("After %d tosses, you have . . .
  prompt();
  }

Toss2.java

Week 3  CIS 210  Fall 2009  11

Benefits of Methods

- A method is an **abstraction**
  - Emphasis on operation
  - Hides the details of the implementation

- Methods facilitate the **reuse** of code
  - Don't have to re-design the implementation
  - Makes software maintenance easier
  - Change the method implementation, and all calls will use the changes

Week 3  CIS 210  Fall 2009  12
Static Methods

- A class may contain one or more static methods
  - `main` is a static method when that class is the starting point for the whole program

- Static methods may be called from any other method
  - If called from a method in the same class, we only need the method name, e.g., `prompt()`
  - If called from a method in a different class, then the method name must be qualified by the class name, e.g., `Toss2.prompt()`
  - No objects need to be created in order to call static methods
  - The `Math` class contains many useful static methods

- In other languages, static methods would be called procedures or functions

More about Methods

- Methods can be more versatile when they are parameterized and return values

- Method Parameters
  - Parameter values are passed to methods
  - Code of the method is like a formula that is applied to the parameter values
  - Method definition must declare each parameter type
  - Method call must provide values that match the formal parameters of the method definition

- Return values
  - A method can return a single value
  - The type of the value returned must be given in the method definition
  - The value is returned by using the `return` statement
Method to Compute Maximum

- Static method **definition** with two integer parameters and an integer return value

```java
public static int max(int num1, int num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

- **MaxTest.java**

Method Control Flow

| x 5 | y 13 | m 13 |

```java
void main(...) {
    ...
    int x = 5;
    int y = 13;
    int m = max(x, y);
    System.out.println(m);
}
```
Variable Scope

- The **scope** of a variable is the places in the Java source code where it is legal to use the variable
  - Scope also applies to all identifiers: method names, parameter names, class names
- The scope of a local variable is all the lines from the line of declaration up to the end of the enclosing block
  - For a local variable, you can think of the variable as coming into existence at the point of declaration and disappearing at the end of the block
  - A variable declared in a for loop has scope restricted to the body of the loop
- Variables may be declared in any block
  - The same names may be used for variables in non-overlapping blocks
- Parameters to a method are like local variables
  - The point of declaration is the method header
  - The scope extends to the end of the method body

```java
int power(int x, int y) {
    int result = 1;
    for (int i = 0; i < y; ++i) {
        int tmp = result;
        for (int j = 1; j < x; ++j)
            result = result + tmp;
    }
    return result;
}

void main(String[] args) {
    int x = 2;
    int y = 5;
    for (int i = 0; i <= y; ++i) {
        System.out.println(x + " to the " + i + " power is " + power(x, i));
    }
}
```
Method Stacks

- Each call to a method is like placing an order and waiting for it to be filled
  - To execute the code of the method, an environment is needed for the method parameters, the local variables of the method, and the return value
- The code of one method may call another method
  - The first method "waits" for the second method it calls to finish
- The order must be preserved
  - The order is **last in, first out** (the last method called must finish and return before the method that called it can continue)
- Method calls form a **stack**
  - The first method is on the bottom
  - The last method called is on the top
  - When the last method called finishes, it is popped from the stack
  - The environment for each call is called a **stack frame**

Tracing the Method Stack

```java
int max(int n1, int n2) {
    if (n1 > n2)
        return n1;
    else
        return n2;
}

int max3(int n1, int n2, int n3) {
    return max(max(n1,n2), n3);
}

void main() {
    m = max3(x,y,z);
}
```

```
x         89
y        117
z         91
m

main

max
n1        117
n2        91
max
n1        89
n2        117
max
n1        89
n2        117
return
n3        91
max
n1        89
n2        117
n3        91
max
n1        89
n2        117
n3        91
return
m = max3(x,y,z);
```

Max3Test.java

```
stack growth
```

Week 3  CIS 210  Fall 2009
Overloading Methods

- Different methods should have different names
- But methods that implement similar abstractions and only differ by their parameters may have the same name
  - This is called **method overloading**
  - The **number of parameters** may be different
  - Or, the **types of the parameters** may be different
  - Java figures out which method to call based on the number and types of the actual parameters used
- The method name, along with the number of parameters and their types, is called the **method signature**
  - It uniquely identifies the method

MaxOverload.java