Module 5: Methods

Ron K. Cytron

Department of Computer Science and Engineering
Washington University in Saint Louis
Thanks to Alan Waldman
for comments that improved these slides

Prepared for 2u
Semester Online

Copyright Ron K. Cytron 2013
5.0 Introduction

- Abstraction is an important idea in computer science
  - We give names to values in our programs
    - The #carbs in a food item, the roll of a die, the amount of blue in a picture, the row of a matrix
  - Now we want to make abstractions out of computational ideas
- Example: A recipe's instructions to "combine ingredients"
  - We could say:
    - Place ingredients a bowl, grab a spoon, move the spoon in a circular motion inside the bowl until the ingredients appear homogenous. Scrape material of spoon and mix that in as well.
    - That's tedious
    - Once the notion of "combine" is defined and understood, we can use it as if it were as basic as pour, open, or set aside, but even those may deserve definitions in terms of smaller, already understood ideas.
- These abstractions allow us to build
  - Bigger ideas from smaller ones
  - New abstractions from older, already defined abstractions
  - Interfaces that make sense and operate safely
5.1 Function Abstraction, An Example

```
combine(sugar, butter)
```
5.1 Function Abstraction, An Example

\texttt{combine(x, y)}

Place \( x \) and \( y \) in a bowl, grab a spoon, move it in a circular motion, ....
5.1 Function Abstraction, An Example

\[ \text{combine}(\text{salt, flour}) \]
5.1 Function Abstraction, An Example

combine(salt, flour)

Doesn't work!

eggs are the wrong data type for combine
5.1 Function Abstraction, An Example

combine(salt, flour)

- eggs
- beat
- vanilla

combine
5.1 Function Abstraction, An Example

\texttt{combine(salt, flour)}

eggs

\begin{align*}
\text{crack} \\
\text{beat} \\
\text{combine}
\end{align*}

vanilla
5.1 Function Abstraction, An Example

combine(salt, flour)

- eggs
- crack
- beat
- combine

Now we have to define crack, beat

vanilla
5.1 Function Abstraction, An Example

combine(salt, flour)

"crack" seems very different from anything we've seen before
5.1 Function Abstraction, An Example

combine(salt, flour)

- eggs
  - crack
  - beat
  - vanilla

However, "beat" seems a lot like "combine"
5.1 Function Abstraction, An Example

- Can we use combine for beat?
  - `combine(sugar, butter)`
    - Takes in two things and mixes them
  - `combine(eggs)`
    - What does this mean?
5.1 Function Abstraction, An Example

• Can we use combine for beat?
  – `combine(sugar, butter)`
    • Takes in two things and mixes them
  – `combine(eggs)`
    • What does this mean?
  – **We could say**
    • `combine(eggs, air)`
5.1 Function Abstraction, An Example

• Can we use combine for beat?
  – combine(sugar, butter)
    • Takes in two things and mixes them
  – combine(eggs)
    • What does this mean?
  – We could say
    • combine(eggs, air)
    • combine(eggs/2, eggs/2)
5.1 Function Abstraction, An Example

• Can we use combine for beat?
  – `combine(sugar, butter)`
    • Takes in two things and mixes them
  – `combine(eggs)`
    • What does this mean?
  – **We could say**
    • `combine(eggs, air)`
    • `combine(eggs/2, eggs/2)` so as to get the same amount after combining
5.1 Function Abstraction, An Example

• Can we use combine for beat?
  – combine(sugar, butter)
    • Takes in two things and mixes them
  – combine(eggs)
    • What does this mean?
  – We could say
    • combine(eggs, air)
    • combine(eggs/2, eggs/2)

• But are combine and beat really the same thing?
5.1 Function Abstraction, An Example

• Can we use combine for beat?
  – combine(sugar, butter)
    • Takes in two things and mixes them
  – combine(eggs)
    • What does this mean?
  – We could say
    • combine(eggs, air)
    • combine(eggs/2, eggs/2)

• But are combine and beat really the same thing?
  – It is a stretch to phrase beat in terms of combine
  – A new abstraction is needed
5.1 Function Abstraction, An Example

combine(salt, flour)

Refactor!
Rethink the abstractions to promote reuse and to clean up code.
5.1 Function Abstraction, An Example

• Beat, Combine
  – What do they have in common?
5.1 Function Abstraction, An Example

• Beat, Combine
  – What do they have in common?
  – What is different about them?
5.1 Function Abstraction, An Example

• Beat, Combine
  – What do they have in common?
  – What is different about them?

• Abstraction above Beat and Combine?
5.1 Function Abstraction, An Example

• Beat, Combine
  – What do they have in common?
  – What is different about them?

• Abstraction above Beat and Combine?
  – What should it take as input?
5.1 Function Abstraction, An Example

• Beat, Combine
  – What do they have in common?
  – What is different about them?

• Abstraction above Beat and Combine?
  – What should it take as input?

• How do we use the new abstraction to
  – Rephrase Combine
  – Rephrase Beat
5.1 Function Abstraction, An Example

• New abstraction: mixWithSpoon
  – What does it need as input?
5.1 Function Abstraction, An Example

• New abstraction: mixWithSpoon
  – What does it need as input?
    • An ingredient
    • Another ingredient
    • A speed at which the ingredients should be mixed
5.1 Function Abstraction, An Example

• New abstraction: mixWithSpoon
  – What does it need as input?
    • An ingredient
    • Another ingredient
    • A speed at which the ingredients should be mixed
  – mixWithSpoon(x, y, speed)
5.1 Function Abstraction, An Example

- **New abstraction: mixWithSpoon**
  - **What does it need as input?**
    - An ingredient
    - Another ingredient
    - A speed at which the ingredients should be mixed
  - `mixWithSpoon(x, y, speed)`

- **combine(x,y) becomes**
  - `mixWithSpoon(x, y, slow)`
5.1 Function Abstraction, An Example

- **New abstraction: mixWithSpoon**
  - What does it need as input?
    - An ingredient
    - Another ingredient
    - A speed at which the ingredients should be mixed
  - `mixWithSpoon(x, y, speed)`

- **combine(x,y) reduces to**
  - `mixWithSpoon(x, y, slow)`

- **beat(z) [ could also say beat(x) ] reduces to**
  - `mixWithSpoon(x, air, fast)`
5.2 Exercise

• Video intro
  – Let's take the cooking abstractions a step further

• Consider the following recipe excerpt
  – Heat 200 cc water to 100 degrees
  – Stir in 5 grams of yeast
  – Melt butter and stir in the vanilla with the butter
  – After 5 minutes, combine yeast with butter, sugar, and flour
  – Place the result in a 110 degree oven to rise for 45 minutes
  – Divide the mixture into 10 equally-sized portions and cook for 45 minutes at 350 degrees
5.2 Exercise

• Question card
  – Define abstractions for the recipe
  – Rephrase the instructions in terms of the abstractions

• Check other students' responses

• What abstractions did others find that are
  – Similar to yours
  – Different from yours
5.2 Exercise

• Video response
  – Abstractions:
    • applyHeat(thing, degrees, time)
    • mixWithSpoon(x, y, speed)
      – Now use it for stir as well as other things
5.3 Syntax, anatomy, design of a method

`combine(flour, water)`

Care about:
- Types of inputs
5.3 Syntax, anatomy, design of a method

\texttt{combine(fLOUR, \textit{water})}

\begin{itemize}
\item Types of inputs
\item Type of output
\end{itemize}

\texttt{dough}
5.3 Syntax, anatomy, design of a method

\texttt{mpy}(x, y)

Care about:
- Types of inputs
5.3 Syntax, anatomy, design of a method

\texttt{mpy(x, y)}

Care about:
- Types of inputs
- Type of output
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
5.3 Syntax, anatomy, design of a method

The signature contains

• The name of the method
5.3 Syntax, anatomy, design of a method

The signature contains

- The name of the method
- The incoming parameters' types (names do not matter)

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```
5.3 Syntax, anatomy, design of a method

The signature contains

- The name of the method
- The incoming parameters' types
- The return type

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```

`mpy(5, 2)`
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```

`mpy(5,2)`
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```

`mpy(5,2)`
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}

mpy(5,2)
```
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + 5;
    }
    return sum;
}
```

mpy(5,2)
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + 5;
    }
    return sum;
}

mpy(5,2)
```
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```

mpy(5,2)
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method:

```java
mpy(5, 2)
```

The method continues until it hits a `return` statement.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < 2; ++i) {
        sum = sum + 5;
    }
    return sum;
}
```
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```
mpy(5, 2)
```

The method continues until it hits a `return` statement.

```
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i = 0; i < 2; ++i) {
        sum = sum + 5;
    }
    return 10;
}
```
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method:

```java
public static int mpy(int c, int d) {
    if (d < 0) return 0;
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```

The method continues until it hits a return statement.

Example:

```
mpy(5,-1)
```
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

\[
\text{mpy}(5, -1)
\]

The method continues until it hits a return statement.

```java
public static int mpy(int c, int d) {
    if (d < 0) return 0;
    int sum = 0;
    for (int i = 0; i < d; ++i) {
        sum = sum + 5;
    }
    return sum;
}
```
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```
mpy(5,-1)
```

The method continues until it hits a return statement.

```
public static int mpy(int c, int d) {
    if (d < 0) return 0;
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + 5;
    }
    return sum;
}
```
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    if (d < 0) return 0;
    int sum = 0;
    for (int i = 0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```

The method continues until it hits a `return` statement.

`mpy(5,-1)`
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    if (d < 0) return 0;
    int sum = 0;
    for (int i = 0; i < d; ++i) {
        sum = sum + 5;
    }
    return sum;
}
```

The method continues until it hits a `return` statement.

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
mpy(5, -1)
```

The method continues until it hits a `return` statement.
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    if (d < 0) return 0;
    int sum = 0;
    for (int i = 0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```

The method continues until it hits a `return` statement.

The first return statement executed causes the method to terminate, returning the indicated value.

`mpy(5,-1)`
5.3 Syntax, anatomy, design of a method

The method body contains the instructions that execute when the method is called.

At that time, values are available for the parameters, and these are substituted throughout the method.

```java
public static int mpy(int c, int d) {
    if (-1 < 0) return 0;
    int sum = 0;
    for (int i = 0; i < -1; ++i) {
        sum = sum + 5;
    }
    return sum;
}
```

The method continues until it hits a `return` statement.

The `first return` statement executed causes the method to terminate, returning the indicated value.

`mpy(5, -1)`
5.3 Syntax, anatomy, design of a method

- Double a number

```java
public static int doubleIt(int in) {
    return 2 * in;
}
```
5.3 Syntax, anatomy, design of a method

- Double a number

```java
public static int doubleIt(int in) {
    return mpy(in, 2);
}
```

Or, we can reduce the problem of doubling a number to the problem of multiplying it by 2.
5.3 Syntax, anatomy, design of a method

• Double a number

```java
public static int doubleIt(int in) {
    return mpy(in, 2);
}
```

Or, we can reduce the problem of doubling a number to the problem of multiplying it by 2. In this way we reuse the mpy method we have already defined.
5.3 Syntax, anatomy, design of a method

• Represent a double with 2 digits after the decimal point

```java
public static String twoDigs(double d) {
    int t100 = (int) (d*100);
    double result = t100/100.0;
    return "" + result;
}
```

It would be tedious to type this code repeatedly
5.3 Syntax, anatomy, design of a method

• Represent a double with 2 digits after the decimal point

```java
public static String twoDigs(double d) {
    int t100 = (int) (d*100);
    double result = t100/100.0;
    return "" + result;
}
```

It would be tedious to type this code repeatedly. Instead:

• Get it right once
5.3 Syntax, anatomy, design of a method

• Represent a double with 2 digits after the decimal point

```java
public static String twoDigs(double d) {
    int t100 = (int) (d*100);
    double result = t100/100.0;
    return "" + result;
}
```

It would be tedious to type this code repeatedly. Instead:

• Get it right once
• Define it in a method
• Call the method as needed
5.4 Encapsulation mechanism: Scope

• A method exposes its signature
  – But it hides everything else inside is braces { ..... }
  – Local variables
  – Parameters

• Scope is an encapsulation mechanism
  – Hide unnecessary detail
  – Prevent unwanted interactions

• Analogy
  – Our stomach hosts digestive juices
  – These are encapsulated and should not come out
  – But they release nutrition that is allowed out
5.4 Encapsulation mechanism: Scope

- We have seen scope before with iteration

```c
for (int i=0; i < N; ++i) {
    int j = i + 3;
}
```

// i is not available here
// only inside the braces
// (even though it is declared
// slightly outside of them)
// j also is not available here
5.4 Encapsulation mechanism: Scope

• Good software practice:
  – Contain variables in scopes
  – Declare variables close to where they are needed

• Example of bad style:

```c
int i;
//
// 5 pages of code later…
//
while (i < 5) {
...
}
```
5.4 Encapsulation mechanism: Scope

All variables declared within the method's body are local to the method and will disappear when the method returns.

We say such variables are limited to the scope of the method: the text between its braces.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```
5.4 Encapsulation mechanism: Scope

All variables declared within the method's body are *local* to the method and will disappear when the method returns.

sum is such a variable

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```
5.4 Encapsulation mechanism: Scope

All variables declared within the method's body are local to the method and will disappear when the method returns.

\[ \text{sum} \] is such a variable, but so are \( c \) and \( d \)

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```
5.4 Encapsulation mechanism: Scope

All variables declared within the method's body are local to the method and will disappear when the method returns. Similarly, the variable i is available only within the loop's braces.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```
5.4 Encapsulation mechanism: Scope

All variables declared within the method's body are local to the method and will disappear when the method returns.

Similarly, the variable i is available only within the loop's braces.

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = sum + c;
    }
    return sum;
}
```
5.4 Encapsulation mechanism: Scope

• Summary
  – Abstraction
    • Generalize an idea to make it more widely applicable
  – Encapsulation
    • Hide details and contain variables and code to avoid unwanted interactions
5.5 Exercise

• Write a method that takes in a String and returns that string concatenated with itself
• Write a method that takes in a String, an int \( n \), and returns \( n \) copies of the string concatenated together
• Rewrite your first method in terms of your second one
• Look at split(" ") as a method on a string that splits a String into an array containing the parts split by the " ">
• Write a join(String[] array, String joiner) that is the inverse of split
• Rewrite your second method in terms of that one
5.5 Exercise

- Video response to go over code
5.6 Execution of Methods

• We define methods using parameters, code, and return values

• But how do they actually execute?
  – Input values are prepared and placed on top of a stack
  – The called method receives those values, computes a result (if non-void) and places the answer on the stack as it returns

• The stack allows one method to call another
  – Waiting for that other method to finish and provide its answer before continuing

• Let's examine this in greater detail
  – The Java debugger can help us see what is happening
public static int add(int a, int b) {
    return a + b;
}

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))
```

The 1 is placed on the stack for the first parameter to add.
5.6 Execution of Methods

```
int x = 5;
int y = 2;
add(1, mpy(x, y))
```

Copies of x and y are placed on the stack
"call by value"
5.6 Execution of Methods

```java
int x = 5;
int y = 2;
add(1, mpy(x, y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i = 0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

Copies of x and y are placed on the stack
5.6 Execution of Methods

```
int x = 5;
int y = 2;

add(1, mpy(x,y))

These are picked up
by the called method
for its parameters' values

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x,y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

The method executes
5.6 Execution of Methods

```
int x = 5;
int y = 2;

add(1, mpy(x,y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

The method executes

```
5 2
```

<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x,y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

The method executes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
5.6 Execution of Methods

int x = 5;
int y = 2;

add(1, mpy(x,y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
5.6 Execution of Methods

```
int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int add(int a, int b) {
    return a + b;
}
```

Parameters are picked up by add
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int add(int a, int b) {
    return a + b;
}
```

The result is computed and stored on the stack for the caller to retrieve.
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int add(int a, int b) {
    return a + b;
}
```

The result is computed and stored on the stack for the caller to retrieve.
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

The return value is picked up

```
c d i sum  
5 2 0 0
```
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

The return value is picked up
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))
```

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

The return value is picked up

<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i = 0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i = 0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

Values of sum and c are pushed on the stack for the call to add

```
<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
```
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

Parameters are picked up by add

add(1, mpy(x, y))

5 5

public static int add(int a, int b) {
    return a + b;
}
```
5.6 Execution of Methods

```java
public static int add(int a, int b) {
    return a + b;
}
```
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int add(int a, int b) {
    return a + b;
}
```

The result is computed and stored on the stack for the caller to retrieve.
5.6 Execution of Methods

int x = 5;
int y = 2;

add(1, mpy(x, y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
5.6 Execution of Methods

```java
int x = 5;
int y = 2;
add(1, mpy(x, y))
```

The return value is picked up

```java
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
5.6 Execution of Methods

\[\text{int } x = 5;\]
\[\text{int } y = 2;\]

\[\text{add}(1, \text{mpy}(x,y))\]

The return value is picked up

\[
\begin{array}{|c|c|c|c|}
\hline
c & d & i & \text{sum} \\
\hline
5 & 2 & 0 & 10 \\
\hline
\end{array}
\]
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x,y))

public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

The loop is over

<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))
```

The return instruction pushes the value onto the stack for the caller to retrieve

```
public static int mpy(int c, int d) {
    int sum = 0;
    for (int i=0; i < d; ++i) {
        sum = add(sum, c);
    }
    return sum;
}
```

<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

private int add(int a, int b) {
    return a + b;
}
```

Parameters are picked up by `add`
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x, y))

private int add(int a, int b) {
    return a + b;
}
```

The result is computed and stored on the stack for the caller to retrieve.
5.6 Execution of Methods

```java
int x = 5;
int y = 2;

add(1, mpy(x,y))

private int add(int a, int b) {
    return a + b;
}
```

The result is computed and stored on the stack for the caller to retrieve.
5.7 Roundtable

• Use the debugger in eclipse to step through the program just as I did
  – How can you step into a method?
  – How can you step over a statement?
• Explain how to look at local variables and parameters
• Introduce a bug into the code (<= instead of <)
• Have the student find the bug
• Explain how to view the stack
5.8 Conclusion

• Methods allow us to
  – Make an idea abstract and reusable
  – Hide details and encapsulate names

• Methods take parameters
  – They are passed "by value"
  – But arrays, when passed by value, allow the receiving method to access their contents!

• Methods can return results
  – void methods do not
  – Other methods do

• The mechanism used to track method behavior is a stack