Unit 3, Part 1

Methods with Parameters and Return Values

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Static Methods

- We've seen how we can use static methods to:
  1. capture the structure of a program – breaking a task into subtasks
  2. eliminate code duplication
- Thus far, our methods have been limited in their ability to accomplish these tasks.

A Limitation of Simple Static Methods

- For example, in our `DrawTorch` program, there are several `for` loops that each print a series of spaces, such as:

```
for (int i = 0; i < 4 - line; i++) {
    System.out.print(" ");
}

for (int i = 0; i < line - 1; i++) {
    System.out.print(" ");
}
```

- However, despite the fact that all of these loops print spaces, we can't replace them with a method that looks like this:

```
public static void printSpaces() {
    …
```

Why not?

Parameters

- In order for a method that prints spaces to be useful, we need one that can print an *arbitrary number* of spaces.
- Such a method would allow us to write commands like these:

```
printSpaces(5);
printSpaces(4 - line);
```

where the number of spaces to be printed is specified between the parentheses.

- To do so, we write a method that has a *parameter:*

```
public static void printSpaces(int numSpaces) {
    for (int i = 0; i < numSpaces; i++) {
        System.out.print(" ");
    }
}
```

Parameters (cont.)

- A parameter is a special type of variable that allows us to pass information into a method.
- Consider again this method:

```
public static void printSpaces(int numSpaces) {
    for (int i = 0; i < numSpaces; i++) {
        System.out.print(" ");
    }
}
```

- When we execute a method call like

```
printSpaces(10);
```

the expression specified between the parentheses:

- is evaluated
- is assigned to the parameter
- can thereby be used by the code inside the method

Parameters (cont.)

```
public static void printSpaces(int numSpaces) {
    for (int i = 0; i < numSpaces; i++) {
        System.out.print(" ");
    }
}
```

- Here's an example with a more complicated expression:

```
int line = 2;
printSpaces(4 - line);
            4 - 2
              2
```

A Note on Terminology

- The term *parameter* is used for both:
  - the variable specified in the method header
    - known as a *formal* parameter
  - the value that you specify when you make the method call
    - known as an *actual* parameter
    - also known as an *argument*

formal parameter

```
public static void printSpaces(int numSpaces) {
    for (int i = 0; i < numSpaces; i++) {
        System.out.print(" ");
    }
}
```

actual parameter / argument

```
printSpaces(10);
```

Parameters and Generalization

- Parameters allow us to *generalize* a task.
- They allow us to write one method that can perform a family of related tasks – instead of writing a separate method for each separate task.

```
print5Spaces()
print10Spaces()
print20Spaces()
print100Spaces()
…
```

⇒ `printSpaces(`*`parameter`*`)`

Representing Individual Characters

- So far we've learned about two data types:
  - `int`
  - `double`
- The `char` type is used to represent individual characters.
- To specify a `char` literal, we surround the character by single quotes:
  - examples: `'a'` `'z'` `'0'` `'7'` `'?'` `'\\'`
  - can only represent single characters
  - don't use double-quotes!
    `"a"` is a string, not a character

Methods with Multiple Parameters

- Here's a method with more than one parameter:

```
public static void printChars(char ch, int num) {
    for (int i = 0; i < num; i++) {
        System.out.print(ch);
    }
}
```

- Example of calling this method:

```
printChars(' ', 10);
```

- Notes:
  - the parameters (both formal and actual) are separated by commas
  - each formal parameter must be preceded by its type
  - the actual parameters are evaluated and assigned to the corresponding formal parameters

Example of Using a Method with Parameters

```
public static void drawFlame() {
    for (int line = 1; line <= 4; line++) {
        for (int i = 0; i < 4 - line; i++) {
            System.out.print(" ");
        }
        for (int i = 0; i < line; i++) {
            System.out.print("(");
        }
        for (int i = 0; i < line; i++) {
            System.out.print(")");
        }
        System.out.println();
    }
}
```

replace nested loops with method calls

```
public static void drawFlame() {
    for (int line = 1; line <= 4; line++) {
        printChars(' ', 4 - line);
        printChars('(', line);
        printChars(')', line);
        System.out.println();
    }
}
```

Recall: Variable Scope

- The *scope* of a variable is the portion of a program in which the variable can be used.
- By default, the scope of a variable in Java:
  - begins at the point at which it is declared
  - ends at the end of the innermost block that encloses the declaration

```
public static void printResults(int a, int b) {
    System.out.println("Here are the stats:");

    int sum = a + b;
    System.out.print("sum = ");
    System.out.println(sum);

    double avg = (a + b) / 2.0;
    System.out.print("average = ");
    System.out.println(avg);
}
```

scope of avg

scope of sum

Special Case: Parameters and Variable Scope

- What about the parameters of a method?
  - they do *not* follow the default scope rules!
  - their scope is limited to their method

```
public class MyClass {
    public static void printResults(int a, int b) {
        System.out.println("Here are the stats:");

        int sum = a + b;
        System.out.print("sum = ");
        System.out.println(sum);

        double avg = (a + b) / 2.0;
        System.out.print("average = ");
        System.out.println(avg);
    }

    static int c = a + b;     // does not compile!
}
```

scope of a and b

Practice with Scope

```
public static void drawRectangle(int height) {
    for (int i = 0; i < height; i++) {
        // which variables could be used here?
        int width = height * 2;
        for (int j = 0; j < width; j++) {
            System.out.print("*");
            // what about here?
        }
        // what about here?
        System.out.println();
    }
    // what about here?
}

public static void repeatMessage(int numTimes) {
    // what about here?
    for (int i = 0; i < numTimes; i++) {
        System.out.println("What is your scope?");
    }
}
```

Practice with Parameters

```
public static void printValues(int a, int b) {
    System.out.println(a + " " + b);
    b = 2 * a;
    System.out.println("b" + b);
}

public static void main(String[] args) {
    int a = 2;
    int b = 3;
    printValues(b, a);
    printValues(7, b * 3);
    System.out.println(a + " " + b);
}
```

- What's the output?

A Limitation of Parameters

- Parameters allow us to pass values into a method.
- They *don't* allow us to get a value out of a method.

A Limitation of Parameters (cont.)

- Example: using a method to compute the opposite of a number
- This *won't* work:

```
public static void opposite(int number) {
    number = number * -1;
}

public static void main(String[] args) {
    // read in points from the user

    opposite(points);
    …
}
```

  - the `opposite` method changes the value of `number`, but `number` can't be used outside of that method
  - the method *doesn't* change the value of `points`

Methods That Return a Value

- To compute the opposite of a number, we need a method that's able to *return* a value.
- Such a method would allow us to write statements like this:

```
int penalty = opposite(points);
```

- The value returned by the method would *replace* the method call in the original statement.
- Example:

```
int points = 10;
int penalty = opposite(points);
```

```
int penalty = -10;  // after the method completes
```

Defining a Method that Returns a Value

- Here's a method that computes and returns the opposite of a number:

```
public static int opposite(int number) {
    return number * -1;
}
```

- In the header of the method, `void` is replaced by `int`, which is the type of the returned value.
- The returned value is specified using a `return` statement. Syntax:

```
return <expression>;
```

  - <expression> is evaluated
  - the resulting value replaces the method call in the statement that called the method

Defining a Method that Returns a Value (cont.)

- The complete syntax for the header of a static method is:

```
public static <return type> <name>(<type1> <param1>,
  <type2> <param2>, …)
```

- Note: a method call is a type of expression!
  - it evaluates to its return value

```
int opp = opposite(10);
```

```
int opp = -10;
```

- In our earlier methods, the return type was always `void`:

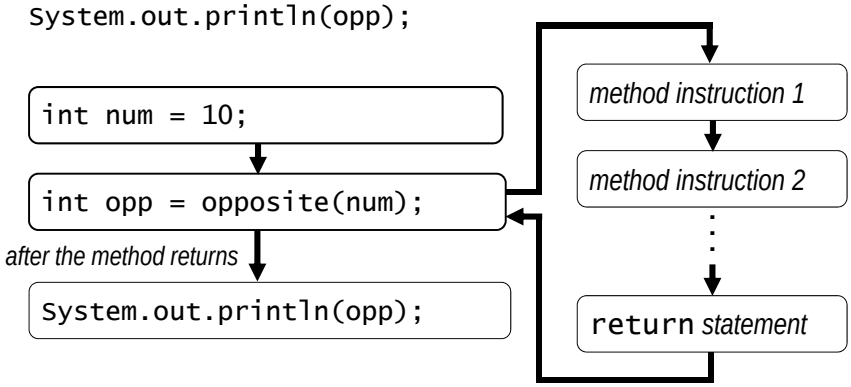
```
public static void printSpaces(int numSpaces) {
    ...
```

This is a special return type that indicates that no value is returned.

Flow of Control with Methods That Return a Value

- The flow of control jumps to a method until it returns.
- The flow jumps back, and the returned value replaces the call.
- Example:

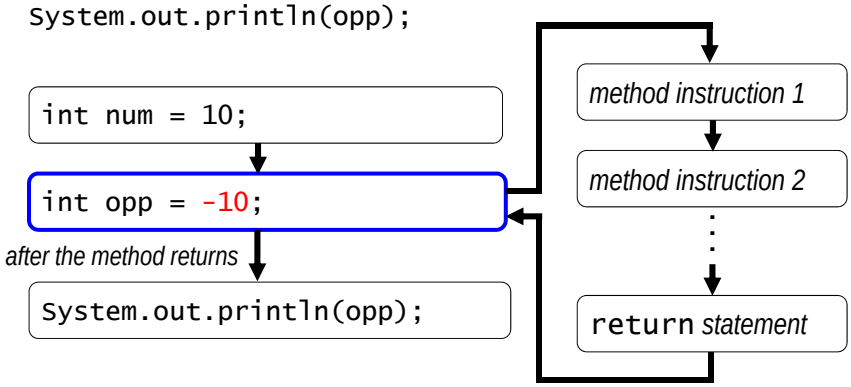
```
int num = 10;
int opp = opposite(num);
System.out.println(opp);
```



Flow of Control with Methods That Return a Value

- The flow of control jumps to a method until it returns.
- The flow jumps back, and *the returned value replaces the call*.
- Example:

```
int num = 10;
int opp = opposite(num);
System.out.println(opp);
```



Returning vs. Printing

- Instead of returning a value, we could write a method that prints the value:

```
public static void printOpposite(int number) {
    System.out.println(number * -1);
}
```

- However, a method that returns a value is typically more useful.
- With such a method, you can still print the value by printing what the method returns:

```
System.out.println(opposite(num));
```

  - the return value replaces the method call and is printed

- In addition, you can do other things besides printing:

```
int penalty = opposite(num);
```

Practice: Computing the Volume of a Cone

- volume of a cone = $\frac{\text{base} * \text{height}}{3}$
- Let's write a method named `coneVol` for computing it.
  - parameters and their types?
  - return type?
  - method definition:

```
public static ________ coneVol(_____________________________) {



}
```

The `Math` Class

- Java's built-in `Math` class contains static methods for mathematical operations.
- These methods return the result of applying the operation to the parameters.
- Examples:
  - `round(double value)` – returns the result of rounding `value` to the nearest integer
  - `abs(double value)` – returns the absolute value of `value`
  - `pow(double base, double expon)` – returns the result of raising `base` to the `expon` power
  - `sqrt(double value)` – returns the square root of `value`
- Table 3.2 in the textbook includes other examples.

The `Math` Class (cont.)

- To use a static method defined in another class, we need to use the name of the class when we call it.
- We use what's known as *dot notation.*
- Syntax:

  <class name>.<method name>(<param1>, <param2>, …)
- Example:

```
double maxVal = Math.pow(2, numBits - 1) - 1;
```

class name — method name — actual parameters

*** Common Mistake ***

- Consider this alternative `opposite` method:

```
public static int opposite(int number) {
    number = number * -1;
    return number;
}
```

- What's wrong with the following code that uses it?

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        opposite(number);
        System.out.print("opposite = ");
        System.out.println(number);
    }
```

Keeping Track of Variables

- Consider again the alternative `opposite` method:

```
public static int opposite(int number) {
    number = number * -1;
    return number;
}
```

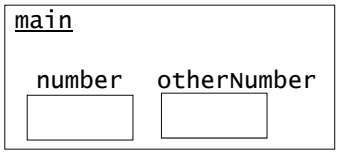
- Here's some code that uses it correctly:

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(number);
        ...
    }
```

- There are two different variables named `number`.
  How does the runtime system distinguish between them?
- More generally, how does it keep track of variables?

Keeping Track of Variables (cont.)

- When you make a method call, the Java runtime sets aside a block of memory known as the *frame* of that method call.

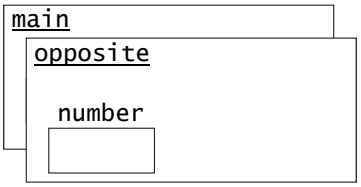


note: we're ignoring `main`*'s parameter for now*

- The frame is used to store:
  - the formal parameters of the method
  - any local variables – variables declared within the method
- A given frame can only be accessed by statements that are part of the corresponding method call.

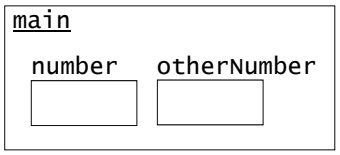
Keeping Track of Variables (cont.)

- When a method (*method1*) calls another method (*method2*), the frame of *method1* is set aside temporarily.
  - *method1*'s frame is "covered up" by the frame of *method2*
  - example: after `main` calls `opposite`, we get:



- When the runtime system encounters a variable, it uses the one from the current frame (the one on top).
- When a method returns, its frame is removed, which "uncovers" the frame of the method that called it.

Example: Tracing Through a Program

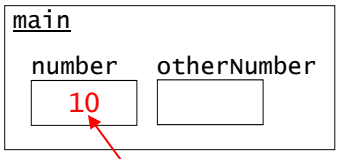


- A frame is created for the `main` method.

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(number);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

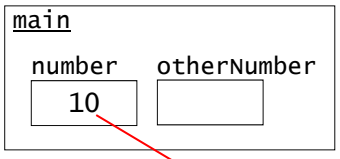
Example: Tracing Through a Program




```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(number);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

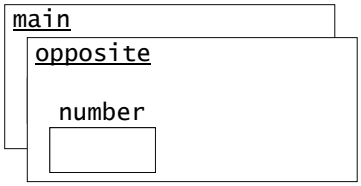
Example: Tracing Through a Program




```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(number);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

Example: Tracing Through a Program

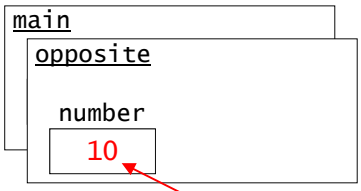


- A frame is created for the `opposite` method, and that frame "covers up" the frame for `main`.

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

Example: Tracing Through a Program

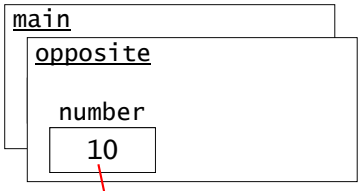


- The actual parameter is passed in and is assigned to the formal parameter.

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

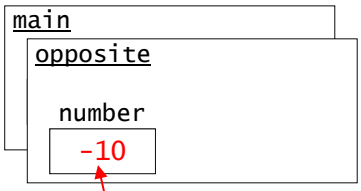
Example: Tracing Through a Program




```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

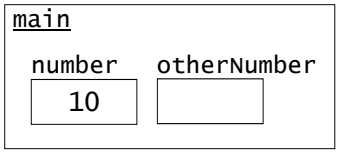
Example: Tracing Through a Program




```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return number;
    }
}
```

Example: Tracing Through a Program

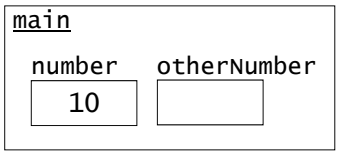


- `opposite` returns, which removes its frame.
- *The variable* `number` *in* `main`*'s frame hasn't been changed!*

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

Example: Tracing Through a Program

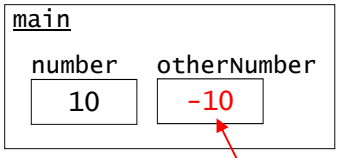


- The returned value replaces the method call.

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

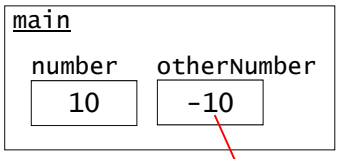
Example: Tracing Through a Program




```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = -10;
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

Example: Tracing Through a Program




```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = -10;
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

Example: Tracing Through a Program

- `main` returns, which removes its frame.

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = -10;
        System.out.print("opposite = ");
        System.out.println(-10);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

Practice

- What is the output of the following program?

```
public class MethodPractice {
    public static int triple(int x) {
        x = x * 3;
        return x;
    }

    public static void main(String[] args) {
        int y = 2;
        y = triple(y);
        System.out.println(y);
        triple(y);
        System.out.println(y);
    }
}
```

More Practice

foo
x | y

```
public class Mystery {
    public static int foo(int x, int y) {
        y = y + 1;
        x = x + y;
        System.out.println(x + " " + y);
        return x;
    }

    public static void main(String[] args) {
        int x = 2;
        int y = 0;

        y = foo(y, x);
        System.out.println(x + " " + y);

        foo(x, x);
        System.out.println(x + " " + y);

        System.out.println(foo(x, y));
        System.out.println(x + " " + y);
    }
}
```

main
x | y

output

From Unstructured to Structured

```
public class TwoTriangles {
    public static void main(String[] args) {
        char ch = '*';      // character used in printing
        int smallBase = 5;  // base length of smaller triangle

        // Print the small triangle.
        for (int line = 1; line <= smallBase; line++) {
            for (int i = 0; i < line; i++) {
                System.out.print(ch);
            }
            System.out.println();
        }

        // Print the large triangle.
        for (int line = 1; line <= 2 * smallBase; line++) {
            for (int i = 0; i < line; i++) {
                System.out.print(ch);
            }
            System.out.println();
        }
     }
}
```

From Unstructured to Structured (cont.)

```
public class TwoTriangles {
    public static void main(String[] args) {
        char ch = '*';      // character used in printing
        int smallBase = 5;  // base length of smaller triangle

        // Print the small triangle.

        printTriangle(_____________________________________);

        // Print the large triangle.

        printTriangle(_____________________________________);
    }

    public static void printTriangle(_____________________) {









    }
}
```