

Topic 3: Methods and Recursion

Reading: JBD Chapter 4

Methods

- A *method* is a named fragment of code that can be invoked from a Java expression.
- Methods are also sometimes referred to as *functions*
- Methods (optionally) take parameters and (optionally) return a result
- Advantages of methods:
 - Enable code to be reused
 - Support top-down design
 - Make programs more modular (breaking functionality into small pieces enables easier implementation and testing)
 - Each method has its own namespace

Two Types of Methods

- *static methods* are properties of a class

```
Math.sqrt(x)
```

```
Math.random()
```

```
Math.max(a, b)
```

- *instance methods* are properties of an object
- To invoke an instance method, use an expression that returns an object, then a dot, then the method call

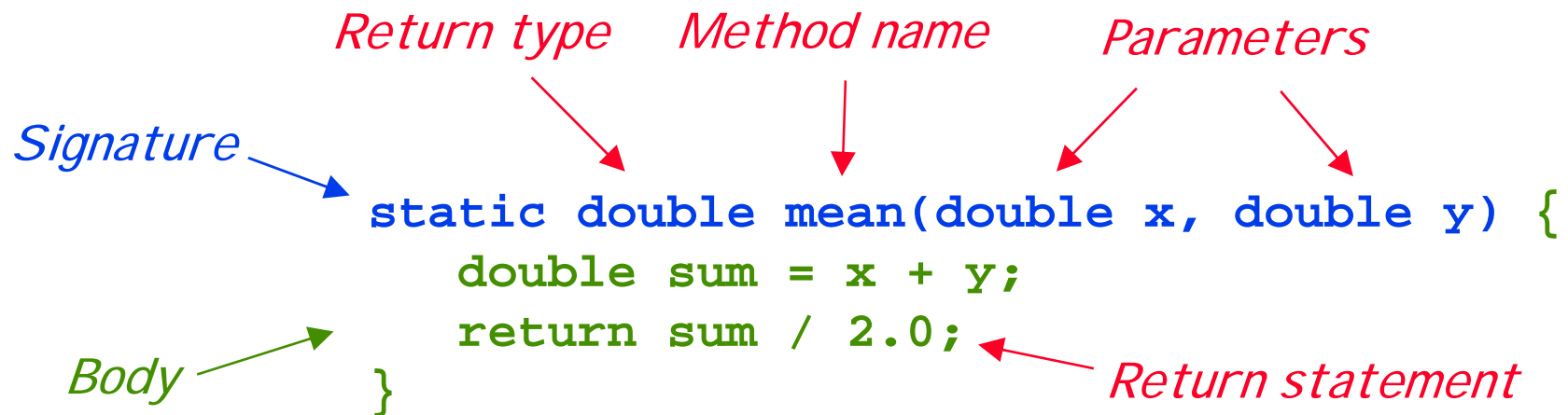
```
string1.length()
```

```
string1.charAt(5)
```

```
string1.equals(string2)
```

```
string1.substring(2, 5).equals(string2)
```

Declaring a Static Method



- The signature specifies:
 - The name of the method
 - The names and types of the parameters (if any)
 - The type of the value returned by the method (if any)

The **return** statement

- Immediately terminates execution of the method
- May or may not have an operand expression

```
return (a + b) / c;
```

```
return;
```

- The value returned (if any) must be compatible with the declared return type of the method (widening conversions are OK)
- If the declared return type is **void**, a return statement is optional, and has no operand
- Otherwise, there must be a return statement on every path through the method

Examples of Methods

- A static method with no parameters

```
static boolean weekend() {  
    // body might read the current date  
    // and return true if it is a weekend  
}
```

- A static method with no results

```
static void logMeIn(String userName) {  
    // body might add the user name and  
    // current time to a log file  
}
```

- A static method with neither parameters nor results

```
static void launchMissiles() {  
    // body might invoke other methods  
}
```

Invoking a Static Method

- A method is executed by a *call* or *invocation*
- The call passes *arguments* to the method
- The argument types must match the declared parameter types (widening conversions are OK)
- Parentheses are required, even if no parameters
- If the method is defined in the same class as the invocation, you do not need the class name

arguments

```
if (mean(a, b) > mean(p, q))  
    launchMissiles();
```

Matching Arguments to Parameters

- The arguments of a method call are bound to the parameters of the method by position (not by name)

```
int x = 5;
int y = 7;
int z = foo(y, x);
```

method call

```
static int foo(int x, int y) {
    // body goes here
}
```

method declaration

- The value returned by the method replaces the method call in the calling environment

What's Wrong Here?

```
class Arithmetic {
    public static void main(String[] args) {
        int x = 1, y = 2;
        int z = add(x, y);
        System.out.println(z);
    }
    int add(int a, int b) {
        return a + b;
    }
}
```

Error: non-static method add(int, int) cannot be referenced from a static context

Using Method Calls in Expressions

- Result type of method must be compatible with its use (widening conversions are OK)

```
static String greeting(String country) {  
    if (country.equals("Australia"))  
        return "G'day";  
    else return "Hello";  
}
```

```
greeting("Italy") + ", Mario"  
    // OK (value is "Hello, Mario")
```

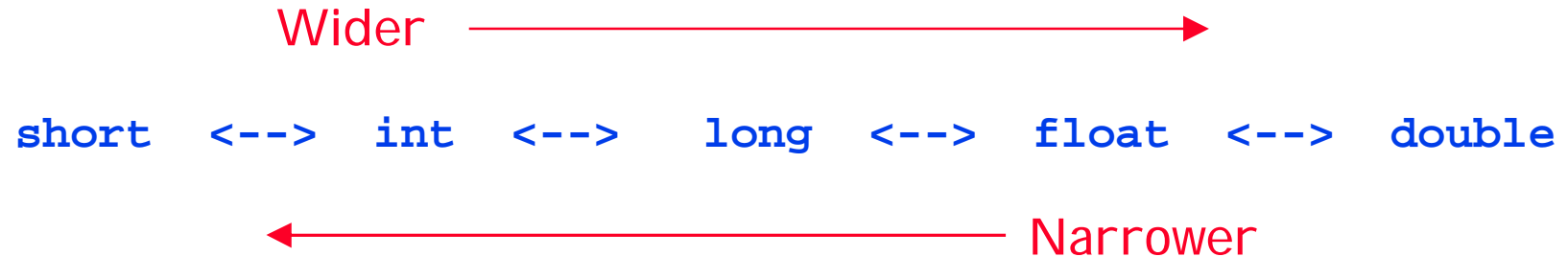
```
Math.sqrt(greeting("Italy"))  
    // Error: method cannot be applied  
    // to this type
```

Overloaded Methods

- Several methods can have the same name but different numbers and types of parameters
- The compiler finds the "best match" for each call (*"method resolution"*)

```
static String foo(int m, int n) {  
    return "I take two ints";  
}  
  
static String foo(boolean q) {  
    return "I take a boolean";  
}  
  
foo(47, 61) // returns "I take two ints"  
foo(true)  // returns "I take a boolean"
```

- If method resolution does not find an exact match, compiler uses *widening conversions*



```
static String foo(float x) {
    return "I take a float";
}

static String foo(double x) {
    return "I take a double";
}

foo(47)    // returns "I take a float"
```

- What happens if the "best match" for a method call is ambiguous?

```
static String foo(int x, double y) {  
    return "I take int and double";  
}  
  
static String foo(double x, int y) {  
    return "I take double and int";  
}  
  
foo(1, 2)    // Error: ambiguous method call
```

Declarations in Method Bodies

- A method body can declare its own local variables (seen only inside the body)

```
class Test {
    public static void main(String[] args) {
        int x = 47;
        System.out.println("main thinks x is " + x);
        foo();
    }
    static void foo() {
        int x = 5;
        System.out.println("foo thinks x is " + x);
    }
}
```

- Output of this program:

```
main thinks x is 47
foo thinks x is 5
```

What happens when you call a method?

- The method body gets a new "stack frame" with its own local variables (including parameters)
- The argument values are copied into the parameters of the new stack frame
- The caller is suspended and the method body is executed
- The result is returned to the caller's stack frame
- The stack frame of the method body goes out of scope (disappears)
- Execution of the caller resumes after the method call

Stack

	Name	Value
main	x	5
	y	0
foo	x	10
	y	2
bar	x	17
	y	7

```
static void main() {  
    int x = 5, y = 0;  
    System.out.println("main: x = " + x);  
    foo(x);  
    System.out.println("main: x = " + x);  
}
```

```
static void foo(int x) {  
    int y = 2;  
    x = x * y;  
    System.out.println("foo: x = " + x);  
    bar(x);  
    System.out.println("foo: x = " + x);  
}
```

```
static void bar(int x) {  
    int y = 7;  
    x = x + y;  
    System.out.println("bar: x = " + x);  
}
```

Stack

	Name	Value
main	x	5
	y	0
foo	x	5 10
	y	2
bar	x	10 17
	y	7

```
static void main() {  
    int x = 5, y = 0;  
    System.out.println("main: x = " + x);  
    foo(x);  
    System.out.println("main: x = " + x);  
}
```

```
static void foo(int x) {  
    int y = 2;  
    x = x * y;  
    System.out.println("foo: x = " + x);  
    bar(x);  
    System.out.println("foo: x = " + x);  
}
```

```
static void bar(int x) {  
    int y = 7;  
    x = x + y;  
    System.out.println("bar: x = " + x);  
}
```

What does this program print?

```
class Test {
    public static void main(String[] args) {
        int x = 1, y = 2;
        System.out.println("Before swap: " + x + y);
        swap(x, y);
        System.out.println("After swap: " + x + y);
    }

    static void swap(int a, int b) {
        int temp;
        temp = a;
        a = b;
        b = temp;
    }
}
```

Lessons:

- Primitive types are passed by value
- A method cannot modify its primitive-type arguments

Can a method modify a String argument?

```
class Test {
    public static void main(String[] args) {
        String s = "Hello";
        System.out.println("Before suffix: " + s);
        suffix(s, " Dolly");
        System.out.println("After suffix: " + s);
    }

    static void suffix(String s1, String s2) {
        s1 = s1 + s2;
    }
}
```

NO, because String is not a mutable type.

Can a method modify a StringBuffer argument?

```
class Test {
    public static void main(String[] args) {
        StringBuffer s = new StringBuffer("Hello");
        System.out.println("Before suffix: " + s);
        suffix(s, " Dolly");
        System.out.println("After suffix: " + s);
    }

    static void suffix(StringBuffer s1, String s2) {
        s1.append(s2);
    }
}
```

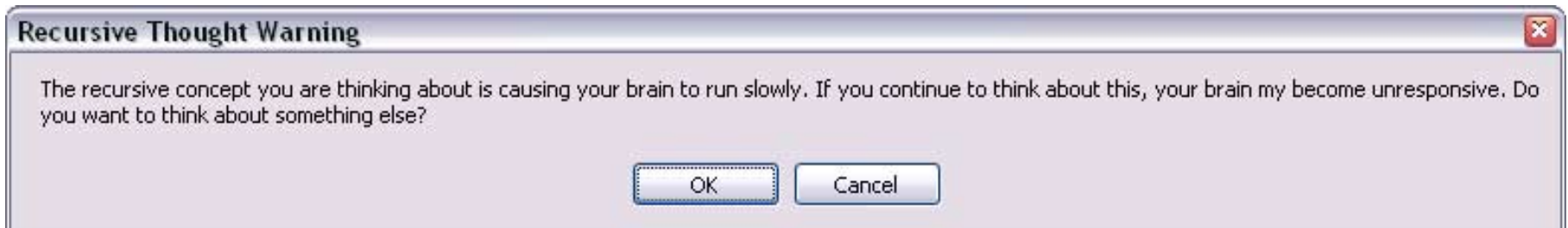
YES, because StringBuffer is a mutable type.

Recursion



Recursion Means "Self-Reference"

- A recursive definition:
*An ancestor is a parent,
or a parent of an ancestor.*
- A recursive acronym:
*GNU stands for
"GNU's Not Unix"*
- Recursion in Nature:



Recursive Methods

- A recursive method is a method that calls itself.
- Recursion can help to solve a problem when:
 - There is one case that is easy to solve (the "base case")
 - Larger cases can be reduced to smaller cases, one step at a time
- The classic example: factorial

```
static int factorial(int n) {  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);  
}
```

Rules for Writing Recursive Methods

Make sure your code:

```
static int fact(int n) {  
    if (n == 1)  
        return 1;  
    else {  
        int f = fact(n-1);  
        return n * f;  
    }  
}
```

- Has a parameter that tends toward a base case
- Identifies the base case and stops the recursion
- Modifies the parameter in each recursive call

Role of the Stack in Recursive Methods

Stack

	Name	Value	
fact(4)	n	4	
	f	6	
fact(3)	n	3	
	f	2	
fact(2)	n	2	
	f	1	
fact(1)	n	1	

```
static int fact(int n) {  
    if (n == 1)  
        return 1;  
    else {  
        int f = fact(n-1);  
        return n * f;  
    }  
}
```

fact(4) = 24

Recursion vs Iteration

- Many simple problems have both iterative and recursive solutions
- Iteration is usually more efficient, but difficult to apply to some kinds of problems

Recursion

```
static int fact(int n) {  
    if (n == 1)  
        return 1;  
    else {  
        int f = fact(n-1);  
        return n * f;  
    }  
}
```

Iteration

```
static int fact(int n) {  
    int f = 1;  
    for (int i = 2;  
         i <= n; i++)  
        f = f * i;  
    return f;  
}
```

What's wrong with this code?

- A recursive method to raise an integer m to the power of another integer n

```
static int power(int m, int n) {  
    return m * power(m, n-1);  
}
```

Don't forget the base case!

```
static int power(int m, int n) {  
    if (n == 0)  
        return 1;  
    else  
        return m * power(m, n-1);  
}
```

Class Exercise

- Write a recursive method that counts the number of times a given character appears in a given string.

```
static int countThisChar(String s, char c) {  
    if (s.length() == 0)  
        return 0;  
    else if (s.charAt(0) == c)  
        return 1 + countThisChar(s.substring(1), c);  
    else  
        return countThisChar(s.substring(1), c);  
}
```