Methods

15-110 Summer 2010

Margaret Reid-Miller

Methods

- A *method* is a group of programming statements that has a name, e.g., `main()`
- A *method definition* includes the *method header* and *method body.*
- Flow of control:
  - When a method is *invoked* (called), program execution transfers to that method and the body of the method is executed.
  - When the method finishes program execution returns to the place from where the method was called.

Review: Calling Methods

- To call a method defined in the same class, then use the method name only:

  e.g., **`displayQuestion();`**
- To call a method defined in a different class and is not static, then use an **object** variable of that class:

  e.g., **`console.next();`**
- To call a method defined in a different class and is static, then use the **class** name:

  e.g., **`Math.round(3.6);`**

Static Methods Definitions

- A *static method* definition has the following form:

header

```
public static <return-type> <method-name> (<parameter-list>) {
        <method-body>
}
```

- The ***parameter-list*** is zero, one, or more variables (type and name) that holds the data passed to the method when the method is called.
- The ***return-type*** specifies the type of the data that method returns to the instruction that called this method.
- The ***method-body*** is the list of instructions that define how this method performs its action.

Void-Method Definitions

- When a method perform some action and does not return a value, its return type is specified as `void`.

Example: return type parameter list

```
public static void displayQuestion() {
   System.out.println
           ("What does Homer like to eat?");
}
```

Calling Void Methods

Example: In a program we might write, on a line by itself, the following:

```
      displayQuestion();
```

This call invokes the **`displayQuestion`** method and the method body is executed.

What is the return type for the `println` method?

```
      System.out.println("DONUTS");
```

Parameters

- Suppose we want to display the question for different members of the Simpson family:

  ```
  displayQuestion("Bart");
  displayQuestion("Marge");
  ```

- To be able to use different person's names, we need to *parameterize* the `displayQuestion` method
- To parameterize a method requires 2 changes:
  - **Define** the method to have one or more *parameter* <u>variables</u> that accept data from the caller.
  - **Call** the method with actual <u>values</u> (*arguments*) to pass to the method.

Method with One Parameter

argument

```
    displayQuestion("Bart");
```

```
public static void displayQuestion(String person) {
   System.out.println("What does " + person
          + " like to eat?");
}
```

parameter

- The parameter `person` is a ***local variable*** (available in the method only) but it gets its initial value from the caller.
- When we call `displayQuestion("Bart")`, it is as if we started the method with

  ```
  String person = "Bart";
  ```

Parameters and Arguments

- A ***parameter*** (or *formal parameter*) in the method header declares the **type** and **name** of a variable that generalizes the method behavior; It is a placeholder for some unspecified value.

```
public static void displayQuestion(String person)
```

- An ***argument*** (or *actual parameter*) is the **actual value** *passed* by the caller to the method when it invokes the method. It indicates the specific behavior of the method.

```
displayQuestion("Bart");
```

Method with Two Parameters

```
printRectangleArea(4.5, 3.2);
```

```
public static void
      printRectangleArea (double width, double height) {
   System.out.println("Area of rectangle with width " +
        width + " and height " + height +
        " is " + width * height);
}
```

Output:

```
Area of a rectangle with width 4.5 and height 3.2 is 14.4
```

Method that returns a value

double taxOwed = computeTax(300.0, 12.0);

return type

```
public static double computeTax
                    (double amount, double rate) {
    double tax = amount * rate / 100.0;
    return tax;
   }
```

return statement

This expression must have the same type as the return type.

The `return` Statement

`return` *<expression>*;

- The ***return statement*** returns the *expression value* to the statement that called this method.
- It can return primitive value or an object. The type must match the return type specified in the method header.
- If a return statement is executed, control returns to the statement that called this method immediately. (Any statements following the return statement in the method are not executed.)

Exercises:

1. Define the following method.

```
   // Returns the maximum of a and b
public static int findMax(int a, int b) {



   }
```

2. Write a code fragment to find the max of three numbers, `n1`, `n2`, and `n3`, using `findMax` method.

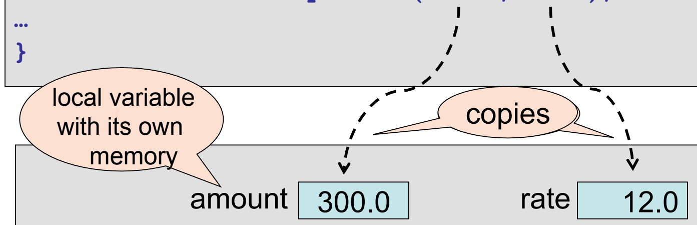
```
public static void main(String[] args) {
…

double taxOwed = computeTax(300.0, 12.0);
…
}
```

f(3,2) evaluates to 17

f(x,y) = 3x+2y+4

```
public static double computeTax
                  (double amount, double rate){
    double tax = amount * rate / 100.0;
    return tax;
}
```

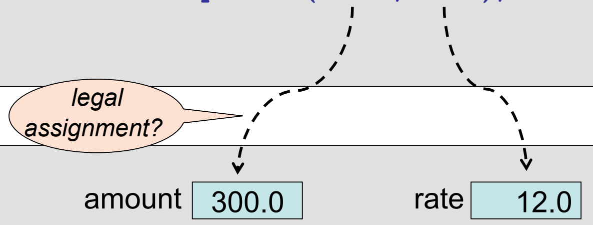
```
public static void main(String[] args) {
…

double taxOwed = computeTax(300.0, 12.0);
…
}
```



```
public static double computeTax(double amount,
                                double rate){
    double tax = amount * rate / 100.0;
    return tax;
}
```

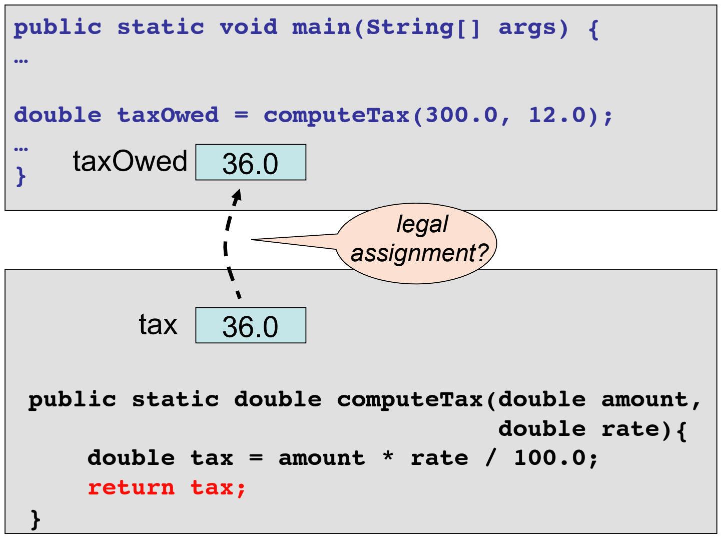
```
public static void main(String[] args) {
…

double taxOwed = computeTax(300.0, 12.0);
…
}
```

***Argument value**: literal, variable, expression*



```
public static double computeTax(double amount,
                                double rate){
    double tax = amount * rate / 100.0;
    return tax;
}
```

***Parameter:** declares a local variable*



```
public static void main(String[] args) {
…

double taxOwed = computeTax(300.0, 12.0);
…
}
```

```
 public static double computeTax(double amount,
                                 double rate){
     double tax = amount * rate / 100.0;
     return tax;
 }
```

```
public static void main(String[] args) {


double amount = 300.0;          // assign the arguments
double rate = 12.0;             // to the parameters

double tax = amount * rate/100; // body of the method

double taxOwed = tax;           // value returned by
                                // computTax()
```

Calling the computeTax() method is as if we had executed the code above.

Local Variables

- A variable declared in the method is called a *local variable*. It can be used only inside the method.

```
public static double computeTax
                  (double amount, double rate) {
    double tax = amount * rate / 100.0;
    return tax;
  }
```

local variable

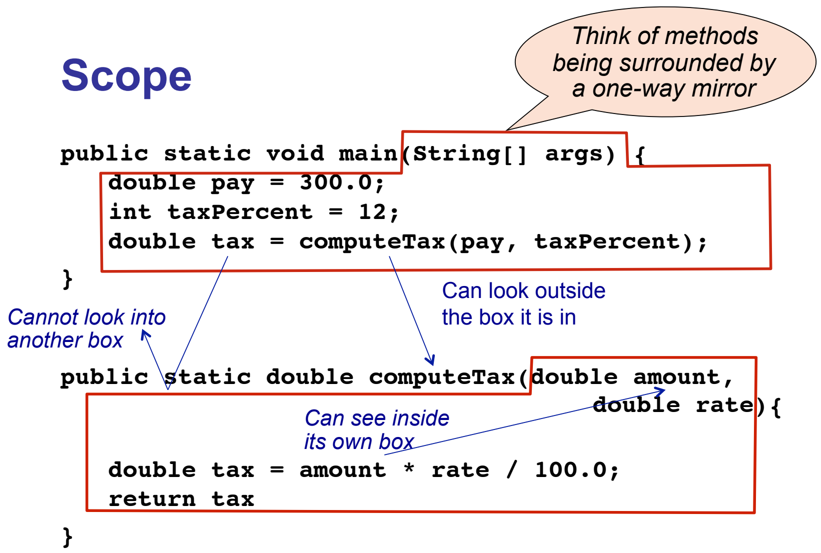
- Different methods can have local variables with same name!

  Are they the same variable?
  Are parameters local variables?
  Can you assign a new value to a parameter?

Scope

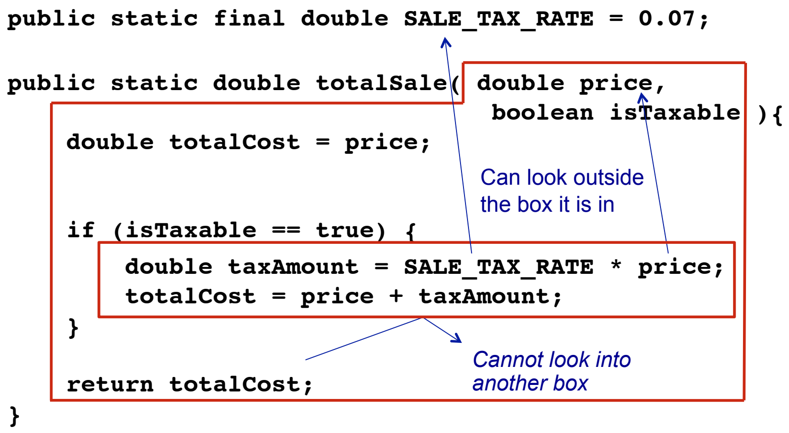
- The ***scope*** of a variable determines where the variable can be referenced, that is, where the variable is visible.
- A related concept is the ***life*** of the variable, which is when, during the execution of the program, a variable has memory space allocated to it and its data can be used.
- The scope of a **local variable** starts from where the variable is declared to the end of the block in which it is declared.
- The scope of a **method parameter** is the method body.

Scope


Think of methods being surrounded by a one-way mirror
Cannot look into another box
Can look outside the box it is in
Can see inside its own box

```
public static void main(String[] args) {
    double pay = 300.0;
    int taxPercent = 12;
    double tax = computeTax(pay, taxPercent);
}

public static double computeTax(double amount,
                                double rate){

    double tax = amount * rate / 100.0;
    return tax
}
```

Scope (cont'd)


Can look outside the box it is in
Cannot look into another box

```
public static final double SALE_TAX_RATE = 0.07;

public static double totalSale( double price,
                                boolean isTaxable ){
    double totalCost = price;

    if (isTaxable == true) {
        double taxAmount = SALE_TAX_RATE * price;
        totalCost = price + taxAmount;
    }

    return totalCost;
}
```

Limiting Scope

- Generally, we want to declare variables in the **most local scope possible** because it provides more security. That is, declare variables at the point you need them.
- If methods have their own local variables to use, then you don't have to consider possible interference from or changes to other parts of the program.
- CAREFUL: Don't limit scope too much:

```
if (age >= 12) {
   int fare = 2;
}
else {
   int fare = 5;
}
System.out.println("Fare is " + fare);
```

outside the scope of `fare`; `fare` is undefined

Overloading Methods

- ***Overloading:*** Two or more methods with the same name but different signatures. Example:
  `String substring(int startIndex, int EndIndex)`
  `String substring(int startIndex)`
- ***Signature:*** The name of the method and the number and type of the parameters.
- Java can figure out which method you are calling based on the number or the types of the arguments supplied in the call to the method. Example:
  `str.substring(3, 6)`
  `str.substring(3)`
- Note: The names of the parameters and the return type do not distinguish two methods, as calls to either method could be the same.