Methods in Java

Selim Aksoy
Bilkent University
Department of Computer Engineering
saksoy@cs.bilkent.edu.tr

Data Scope

- The *scope* of data is the area in a program in which that data can be used (referenced)
- Data declared at the class level can be used by all methods in that class
- Data declared within a method can be used only in that method
- Data declared within a method is called *local data*

Local and Class Scope

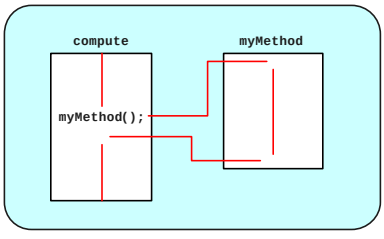
```
public class X {
    public static int a; // a has class scope, can be seen from
                         // anywhere inside the class
    ....

    public static void m() {
          a = 5;         // no problem
          int b = 0;     // b is declared inside the method, local scope
    .....
    } // here variable b is destroyed, no one will remember him

    public static void m2() {
          a = 3;         // ok
          b = 4;         // who is b? compiler will issue an error
    }
}
```


Method Declarations

- A *method declaration* specifies the code that will be executed when the method is invoked (or called)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method is defined

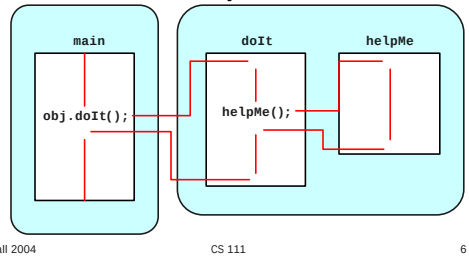
Method Control Flow

- The called method can be within the same class, in which case only the method name is needed



Method Control Flow

- The called method can be part of another class or object



Visibility Modifiers

- Classes support encapsulation: encouraging separation between operations and their implementations
- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data value
- We have used the modifier `final` to define a constant
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss in CS 112

Visibility Modifiers

- Members of a class that are declared with *public visibility* can be accessed from anywhere
- Members of a class that are declared with *private visibility* can only be accessed from inside the class
- Members declared without a visibility modifier have *default visibility* and can be accessed by any class in the same package

Visibility Modifiers

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

The static Modifier

- Static methods can be invoked through the class name rather than through a particular object
- To write a static method, we apply the `static` modifier to the method definition
- The `static` modifier can be applied to variables as well
- It associates a variable or method with the class rather than with an object

Method Header

- A method declaration begins with a *method header*

```
char calc (int num1, int num2, String message)
```

return type — **method name** — **parameter list**

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal argument*

Method Body

- The method header is followed by the *method body*

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

sum and result are local data

They are created each time the method is called, and are destroyed when it finishes executing

The return expression must be consistent with the return type

The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a `void` return type
- A *return statement* specifies the value that will be returned

```
return expression;
```

- Its expression must conform to the return type

Parameters

- Each time a method is called, the *actual parameters* in the invocation are copied into the formal parameters

```
ch = obj.calc (25, count, "Hello");
```

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```


Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler determines which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature

Overloading Methods

Version 1

```
float tryMe (int x)
{
   return x + .375;
}
```

Version 2

```
float tryMe (int x, float y)
{
   return x*y;
}
```

Invocation

```
result = tryMe (25, 4.32)
```


Overloaded Methods

- The `println` method is overloaded:

```
println(String s)
println(int i)
println(double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println("The total is:");
System.out.println(total);
```


Method Decomposition

- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A service method of an object may call one or more support methods to accomplish its goal
- Support methods could call other support methods if appropriate

The StringTokenizer Class

- The elements that comprise a string are referred to as *tokens*
- The process of extracting these elements is called *tokenizing*
- Characters that separate one token from another are called *delimiters*
- The `StringTokenizer` class, which is defined in the `java.util` package, is used to separate a string into tokens

The StringTokenizer Class

- The default delimiters are space, tab, carriage return, and the new line characters
- The `nextToken` method returns the next token (substring) from the string
- The `hasMoreTokens` returns a boolean indicating if there are more tokens to process

Pig Latin Translation Example

- Translating an English sentence into Pig Latin can be decomposed into the process of translating each word
- The process of translating a word can be decomposed into the process of translating words that
  - begin with vowels
  - begin with consonant blends (sh, cr, tw, ...)
  - begins with single consonants

Pig Latin Translation Example

```
String st = "A method should be relatively
            small, so that it can be readily
            understood as a single entity";
String result = PigLatinTranslator.translate(st);
System.out.println(result);
```

output:

```
ayay ethodmay ouldshay ebay elativelyray
all,smay osay atthay ityay ancay ebay
eadilyray understoodyay asyay ayay
inglesay entityyay
```


PigLatinTranslator

```
import java.util.StringTokenizer;

public class PigLatinTranslator {

    //-----------------------------------------------------------
    //  Translates a sentence of words into Pig Latin.
    //-----------------------------------------------------------
    public static String translate (String sentence) {

        String result = "";

        sentence = sentence.toLowerCase();

        StringTokenizer tokenizer = new StringTokenizer (sentence);
        while ( tokenizer.hasMoreTokens() ) {
            result += translateWord( tokenizer.nextToken() );
            result += " ";
        }

        return result;
    }
```


PigLatinTranslator

```
    //-----------------------------------------------------------
    //  Translates one word into Pig Latin. If the word begins with a
    //  vowel, the suffix "yay" is appended to the word.  Otherwise,
    //  the first letter or two are moved to the end of the word,
    //  and "ay" is appended.
    //-----------------------------------------------------------
    private static String translateWord (String word) {

        String result = "";

        if ( beginsWithVowel(word) ) {
            result = word + "yay";
        }
        else {
            if ( beginsWithBlend(word) ) {
                result = word.substring(2) + word.substring(0,2) + "ay";
            }
            else {
                result = word.substring(1) + word.charAt(0) + "ay";
            }
        }

        return result;
    }
```

PigLatinTranslator

```
    //-----------------------------------------------------
    //  Determines if the specified word begins with a vowel.
    //-----------------------------------------------------
    private static boolean beginsWithVowel (String word) {

        String vowels = "aeiou";

        char letter = word.charAt(0);

        return ( vowels.indexOf(letter) != -1 );
    }
}
```