

COMP20220
Programming II (Conversion)

Michael O'Mahony

Chapter 6 Methods

Opening Example

Consider the following problems:

- Find the sum of integers from 1 to 10, inclusive.
- Find the sum of integers from 20 to 30, inclusive.
- Find the sum of integers from 35 to 45, inclusive.

These are similar problems (finding the sum), but the ranges differ...

Opening Example

```
int sum;
```

```
sum = 0;
```

```
for (int i = 1; i <= 10; i++)
```

```
    sum += i;
```

```
System.out.println("Sum 1 to 10 is " + sum);
```

```
sum = 0;
```

```
for (int i = 20; i <= 30; i++)
```

```
    sum += i;
```

```
System.out.println("Sum 20 to 30 is " + sum);
```

```
sum = 0;
```

```
for (int i = 35; i <= 45; i++)
```

```
    sum += i;
```

```
System.out.println("Sum 35 to 45 is " + sum);
```

Opening Example

```
int sum;
```

```
sum = 0;  
for (int i = 1; i <= 10; i++)  
    sum += i;
```

```
System.out.println("Sum 1 to 10 is " + sum);
```

```
sum = 0;  
for (int i = 20; i <= 30; i++)  
    sum += i;
```

```
System.out.println("Sum 20 to 30 is " + sum);
```

```
sum = 0;  
for (int i = 35; i <= 45; i++)  
    sum += i;
```

```
System.out.println("Sum 35 to 45 is " + sum);
```

Solution

```
public static int sum(int n1, int n2) {  
    int sum = 0;  
    for (int i = n1; i <= n2; i++)  
        sum += i;  
    return sum;  
}
```

```
public static void main(String[] args) {  
    System.out.println("Sum 1 to 10 is " + sum(1, 10));  
    System.out.println("Sum 20 to 30 is " + sum(20, 30));  
    System.out.println("Sum 35 to 45 is " + sum(35, 45));  
}
```

Objectives

- To define methods with formal parameters (§6.2).
- To invoke methods with actual parameters (i.e. arguments) (§6.2).
- To define methods with a return value (§6.3).
- To define methods without a return value (§6.4).
- To pass arguments by value (§6.5).
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain (§6.6).
- To use method overloading and understand ambiguous overloading (§6.8).
- To determine the scope of variables (§6.9).
- To apply the concept of method abstraction in software development (§6.10).

Defining and Calling Methods

Example program `TestMax` – demonstrates defining and calling a method `max` to return the largest of two `int` values

The TestMax Class

```
public class TestMax {  
    public static void main(String[] args) {  
        int i = 5;  
        int j = 2;  
        int k = max(i, j);  
  
        System.out.println("The max is " + k);  
    }  
  
    public static int max(int num1, int num2) {  
        int result;  
  
        if (num1 > num2)  
            result = num1;  
        else  
            result = num2;  
  
        return result;  
    }  
}
```

Trace Method Invocation

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

Trace Method Invocation

assign 5 to i

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

Trace Method Invocation

assign 2 to j

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

Trace Method Invocation

invoke method max(i, j)

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

Trace Method Invocation

invoke method max(i, j)
pass the value of i to num1
pass the value of j to num2

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

Trace Method Invocation

declare variable result

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

Trace Method Invocation

(num1 > num2) is true since
num1 is 5 and num2 is 2

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    return result;  
}
```


Trace Method Invocation

result is now 5

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

Trace Method Invocation

return result, which is 5

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

Trace Method Invocation

return result and assign its value to k

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

animation

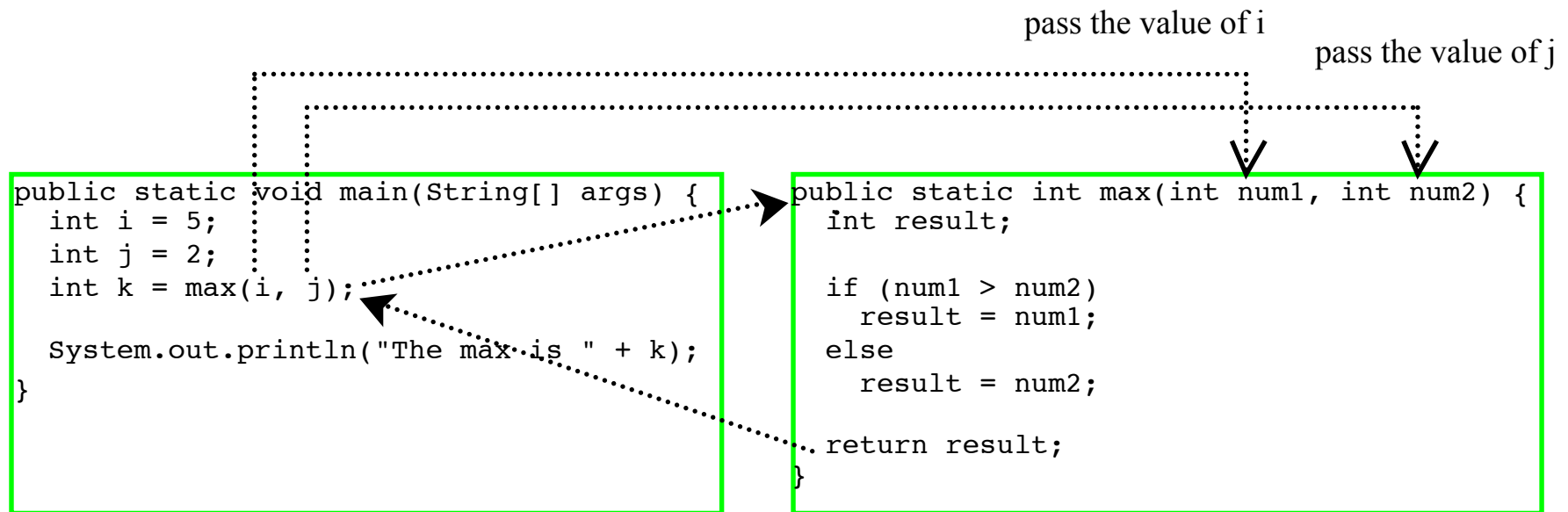
Trace Method Invocation

execute the print statement

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
    System.out.println("The max is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    .  
    return result;  
}
```

Calling Methods



Defining Methods

A method is a collection of statements that are grouped together to perform an operation.

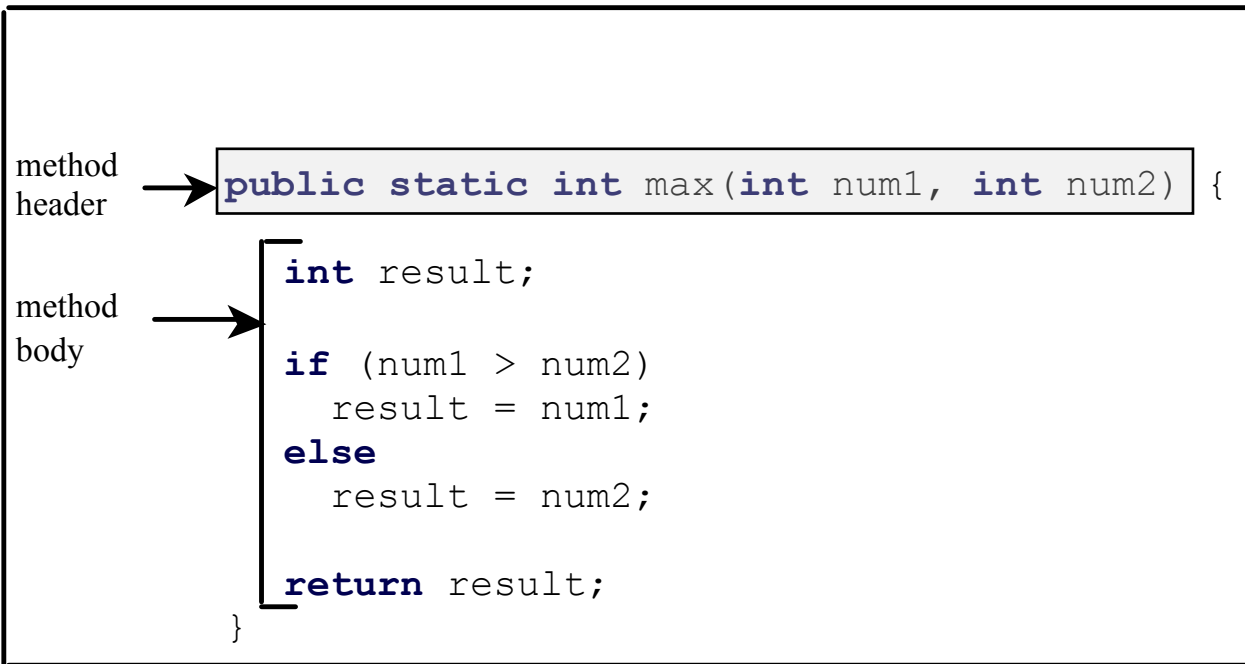
Define a method

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

Defining Methods

A method definition consists of a *method header* and a *method body*.

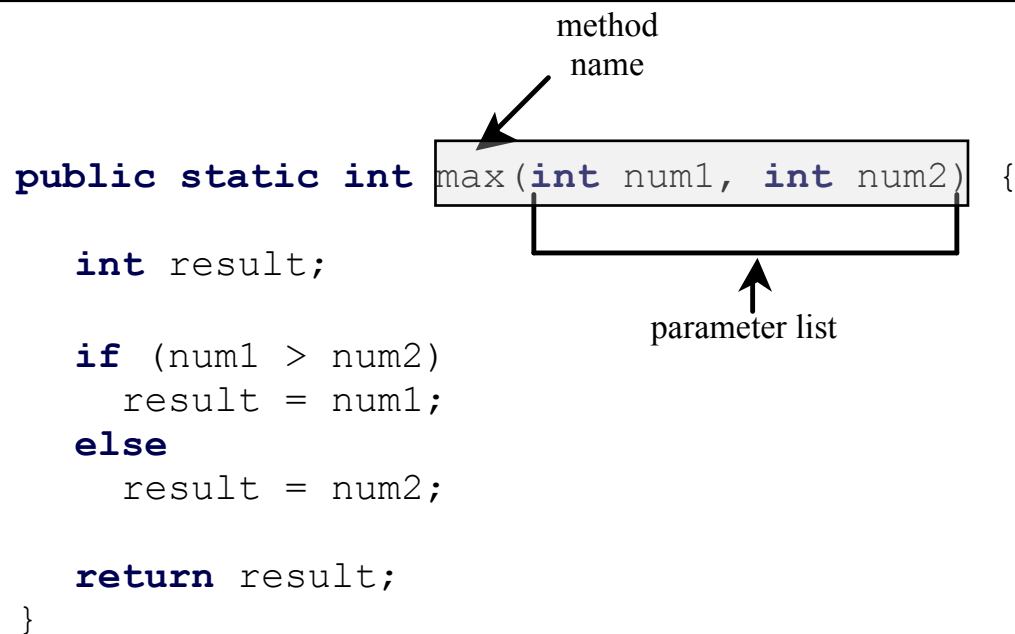
Define a method



Method Signature

The *method signature* is the combination of the *method name* and the *parameter list*.

Define a method



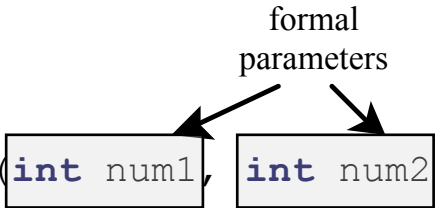
```
public static int max(int num1, int num2) {  
    int result;  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    return result;  
}
```

The diagram illustrates the components of a method signature in the provided Java code. A box labeled "method name" points to the text "max" in the method signature. Another box labeled "parameter list" points to the text "(int num1, int num2)" in the same signature. The rest of the code, including the return type "int", access modifiers "public static", and the method body, is not part of the signature.

Formal Parameters

The variables defined in the method header are known as *formal parameters*.

Define a method



The diagram illustrates the concept of formal parameters in a Java method signature. It shows a code snippet for a method named `max` that takes two integer parameters, `num1` and `num2`. The parameters are highlighted with boxes, and arrows point from the label "formal parameters" to these boxes.

```
public static int max(int num1, int num2) {  
    int result;  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    return result;  
}
```

Return Value Type

A method may return a value. The *return value type* is the data type of the value the method returns.

If a method does not return a value, the return value type is the keyword `void`. For example, the return value type in the `main` method is `void`:

- `main` method header: `public static void main(String[] args)`

Define a method

```
return value
type
↓
public static int max(int num1, int num2) {

    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result; ← return value
}
```

The diagram shows a Java method definition for `max`. The return type `int` is highlighted with a box and an arrow pointing to it from the text "return value type" above. The method body includes a local variable `result` of type `int`, an `if-else` statement to compare `num1` and `num2`, and a `return` statement that returns `result`. An arrow points from the text "return value" to the `result` variable in the `return` statement.

Actual Parameters

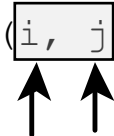
When a method is invoked, value(s) are passed to the parameter(s). These values are referred to as *actual parameters* or *arguments*.

Define a method

```
public static int max(int num1, int num2) {  
  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

Invoke a method

```
int i = 5;  
int j = 2;  
int k = max(i, j);
```



actual parameters
(arguments)

void Methods

void methods do not return values.

Consider the problem of assigning a grade letter to a score (e.g. a grade A is assigned to a score of 95)

The following two programs illustrate the use of methods which implement similar functionality (i.e. assigning grade letters to scores) but use different return value types...

TestVoidMethod

TestReturnMethod

Passing Arguments

```
public static void nPrintln(String message, int n) {  
    for (int i = 0; i < n; i++)  
        System.out.println(message);  
}
```

Suppose you invoke the method using:

```
nPrintln("Welcome to Java", 5);
```

What is the output?

Suppose you invoke the method using:

```
nPrintln("Computer Science", 15);
```

What is the output?

What happens when you invoke the following method?

```
nPrintln(15, "Computer Science");
```

Passing Arguments by Value

The arguments are passed by value to parameters when invoking a method.

This is referred to as *pass-by-value*.

The value of the variable which is passed to the parameter is not affected, regardless of the changes made to the parameter inside the method.

Passing Arguments by Value

Example – passing arguments by value to methods...

TestPassByValue

Overloading Methods

Overloading methods enables you to define methods with the same name as long as their *signatures* are different:

- Remember – the method name and the parameter list together constitute the *method signature*.
- The Java compiler determines which method to use based on the method signature.

Overloading Methods

Example – overloading the max method:

```
public static int max(int num1, int num2) {  
    return (num1 > num2) ? num1 : num2;  
}
```

```
public static double max(double num1, double num2) {  
    return (num1 > num2) ? num1 : num2;  
}
```



TestMethodOverloading

Overloading Methods

Both `max(double, double)` and `max(int, int)` are possible matches for `max(3, 4) ...`

The Java compiler finds the method that *best matches* a method invocation:

- Since the method `max(int, int)` is a better match for `max(3, 4)` than `max(double, double)`, it is used to invoke `max(3, 4)`

Ambiguous Invocation

Sometimes there may be two or more possible matches for the invocation of a method, but the compiler cannot determine the most specific match.

This is referred to as *ambiguous invocation*.

Ambiguous invocation is a compile error.

Ambiguous Invocation

Example of ambiguous invocation:

```
public class AmbiguousOverloading {  
    public static void main(String[] args) {  
        System.out.println(max(1, 2));  
    }  
  
    public static double max(int num1, double num2) {  
        return (num1 > num2) ? num1 : num2;  
    }  
  
    public static double max(double num1, int num2) {  
        return (num1 > num2) ? num1 : num2;  
    }  
}
```

Ambiguous Invocation

Solution:

```
public class AmbiguousOverloading {  
    public static void main(String[] args) {  
        System.out.println(max(1, 2));  
    }  
  
    public static double max(int num1, int num2) {  
        return (num1 > num2) ? num1 : num2;  
    }  
  
    public static double max(double num1, double num2) {  
        return (num1 > num2) ? num1 : num2;  
    }  
}
```

Case Study: Generating Random Characters

Suppose we wish to implement a class that generates random characters.

In particular, we wish to:

- Generate a random lowercase letter
- Generate a random uppercase letter
- Generate a random digit character
- Generate any Unicode character at random

How do we proceed?

- Write separate methods for each of the above?
- Do something a little different?

Generating Random Characters

A random digit can be generated by:

```
char ch = (char) ('0' + Math.random() * ('9' - '0' + 1));
```

So we could write a method similar to the above to randomly generate a lowercase letter, an uppercase letter, and any Unicode character...

Generating Random Characters

A random digit can be generated by:

```
char ch = (char) ('0' + Math.random() * ('9' - '0' + 1));
```

So we could write a method similar to the above to randomly generate a lowercase letter, an uppercase letter, and any Unicode character...

A better approach – we can *generalize* the above to generate a random character between any two characters `ch1` and `ch2` as follows:

```
char ch = (char) (ch1 + Math.random() * (ch2 - ch1 + 1));
```


Generating Random Characters

A random digit can be generated by:

```
char ch = (char) ('0' + Math.random() * ('9' - '0' + 1));
```

So we could write a method similar to the above to randomly generate a lowercase letter, an uppercase letter, and any Unicode character...

A better approach – we can *generalize* the above to generate a random character between any two characters `ch1` and `ch2` as follows:

```
char ch = (char) (ch1 + Math.random() * (ch2 - ch1 + 1));
```

Approach:

- Write a *general* method to generate a random character between any two characters
- Write one method each to randomly generate a digit, a lowercase letter, an uppercase letter, and any Unicode character – all of which call the general method

The RandomCharacter Class

```
public class RandomCharacter {  
  
    // Generate a random character between ch1 and ch2, inclusive  
    public static char getRandomCharacter(char ch1, char ch2) {  
        return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));  
    }  
  
}
```

The RandomCharacter Class

```
public class RandomCharacter {  
  
    // Generate a random character between ch1 and ch2, inclusive  
    public static char getRandomCharacter(char ch1, char ch2) {  
        return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));  
    }  
  
    // Generate a random lowercase letter  
    public static char getRandomLowerCaseLetter() {  
        return getRandomCharacter('a', 'z');  
    }  
  
}
```

The RandomCharacter Class

```
public class RandomCharacter {  
  
    // Generate a random character between ch1 and ch2, inclusive  
    public static char getRandomCharacter(char ch1, char ch2) {  
        return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));  
    }  
  
    // Generate a random lowercase letter  
    public static char getRandomLowerCaseLetter() {  
        return getRandomCharacter('a', 'z');  
    }  
  
    // Generate a random uppercase letter  
    public static char getRandomUpperCaseLetter() {  
        return getRandomCharacter('A', 'Z');  
    }  
  
}
```

The RandomCharacter Class

```
// RandomCharacter.java: Generate random characters
public class RandomCharacter {
    // Generate a random character between ch1 and ch2, inclusive
    public static char getRandomCharacter(char ch1, char ch2) {
        return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
    }

    // Generate a random lowercase letter
    public static char getRandomLowerCaseLetter() {
        return getRandomCharacter('a', 'z');
    }

    // Generate a random uppercase letter
    public static char getRandomUpperCaseLetter() {
        return getRandomCharacter('A', 'Z');
    }

    // Generate a random digit character
    public static char getRandomDigitCharacter() {
        return getRandomCharacter('0', '9');
    }

    // Generate a random character
    public static char getRandomCharacter() {
        return getRandomCharacter('\u0000', '\uFFFF');
    }
}
```

RandomCharacter

TestRandomCharacter1

TestRandomCharacter2

Scope of Local Variables

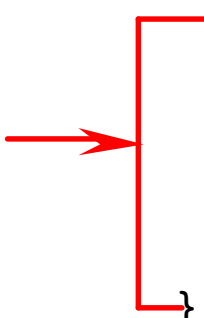
Local variable: a variable defined inside a method.

Scope: the part of the program where the variable can be referenced.

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.

```
public static void method1() {  
    .  
    .  
    int i = 1;  
    .  
    .  
    .  
    .  
    .  
}
```

scope of i →



Scope of Local Variables

A variable declared in the initial action part of a `for` loop header has its scope in the entire loop.

A variable declared inside a `for` loop body has its scope limited to the loop body from its declaration and to the end of the block that contains the variable.

```
public static void method1() {  
    .  
    .  
    for (int i = 1; i < 10; i++) {  
        .  
        .  
        int j;  
        .  
        .  
        .  
    }  
    .  
    .  
}
```

The diagram illustrates the scope of variables `i` and `j` in the provided code snippet. A red bracket on the left side of the `for` loop header and its body indicates the scope of variable `i`, which is the entire loop. A green bracket on the left side of the loop body indicates the scope of variable `j`, which is limited to the loop body. Arrows point from the text labels "scope of i" (in red) and "scope of j" (in green) to their respective brackets.

Scope of Local Variables

Variable `i` **can** be declared in two **non-nesting** blocks:

```
public static void method1() {  
    int x = 1;  
    int y = 1;  
  
    for (int i = 1; i < 10; i++) {  
        x += i;  
    }  
  
    for (int i = 1; i < 10; i++) {  
        y += i;  
    }  
}
```

Variable `i` **cannot** be declared in two **nesting** blocks:

```
public static void method2() {  
    int i = 1;  
    int sum = 0;  
  
    for (int i = 1; i < 10; i++) {  
        sum += i;  
    }  
}
```


Modularizing Code

Methods can be used to reduce redundant coding and enable code reuse.

Methods facilitate the modularization of code, thereby improving the quality of the program.

Separates the functionality of a program into independent modules, such that each module contains everything necessary to execute only one aspect of the desired functionality.

Modularizing Code

By writing a method to obtain e.g. the maximum value of two integers, a number of advantages accrue:

- It isolates the problem for computing the maximum value from the rest of the code in the `main` method. Thus, the logic becomes clear and the program is easier to read.
- Any errors in computing the maximum value are confined to the `max` method, which narrows the scope of debugging.
- The `max` method can also be reused by other programs.

Method Abstraction

The key to developing software is to apply the concept of abstraction.

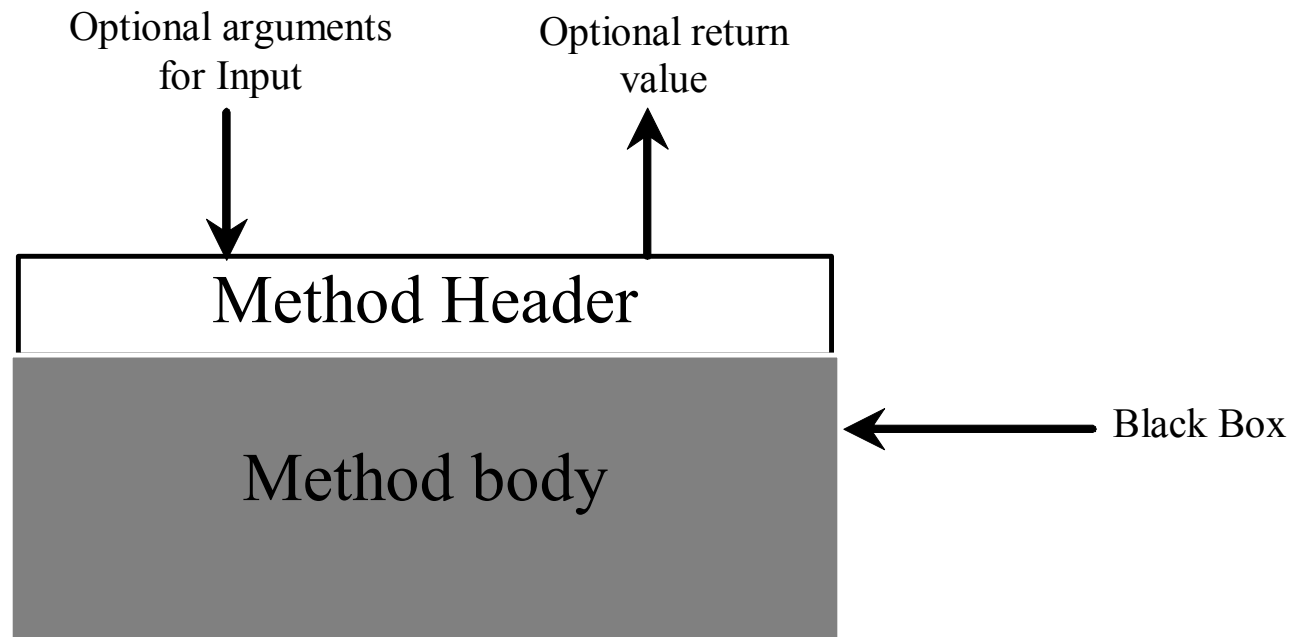
Method abstraction is achieved by separating the *use* of a method from its *implementation*.

The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is also known as *information hiding* or *encapsulation*.

If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature.

Method Abstraction

Think of the method body as a black box that contains the detailed implementation for the method.



Benefits of Methods

Write a method once and reuse it anywhere.

Information hiding – hide the method implementation from the user.

Reduce complexity.

Reuse Methods from Other Classes

One of the benefits of methods is for reuse.

Static methods implemented in one class can be invoked from any class.

We have already seen this...

- For example, calling various methods in the `Math` and `Character` classes:

```
Math.abs (...), Math.pow (...,...) ...
```

```
Character.isDigit (...), Character.toUpperCase (...)
```

- And of course calling methods in our `RandomCharacter` class:

```
RandomCharacter.getRandomDigitCharacter (),
```

```
RandomCharacter.getRandomCharacter () ...
```

Stepwise Refinement

The concept of method abstraction can be applied to the process of developing programs.

When writing a large program to solve a complex problem:

- Use the *divide and conquer* strategy, also known as *stepwise refinement*, to decompose the problem into sub-problems...
- Sub-problems can be further decomposed into smaller, more manageable problems...
- A *design diagram* (aka *structure chart*) can be used to visualize this process.

The idea is not to focus on *implementation details*, rather to decompose and identify the successive steps involved in writing the program.

Case Study: Validating Credit Card Numbers

Case study to demonstrate stepwise refinement. Consider a program to validate credit card numbers.

Valid credit card numbers must satisfy certain criteria:

- 1) The number must have between 13 and 16 digits.
- 2) The number must have a valid prefix – i.e. it must start with any of the following:
 - 4 (for Visa cards)
 - 5 (for Master cards)
 - 37 (for American Express cards)
 - 6 (for Discover cards)
- 3) The number must satisfy the *Luhn check* (aka the *Mod 10 check*).

Validating Credit Card Numbers

The *Luhn check*:

- (i) Double every second digit from right to left.

If doubling of a digit gives a two-digit number, add the two digits to get a single-digit number.

Add all single-digit numbers:

$$4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37$$

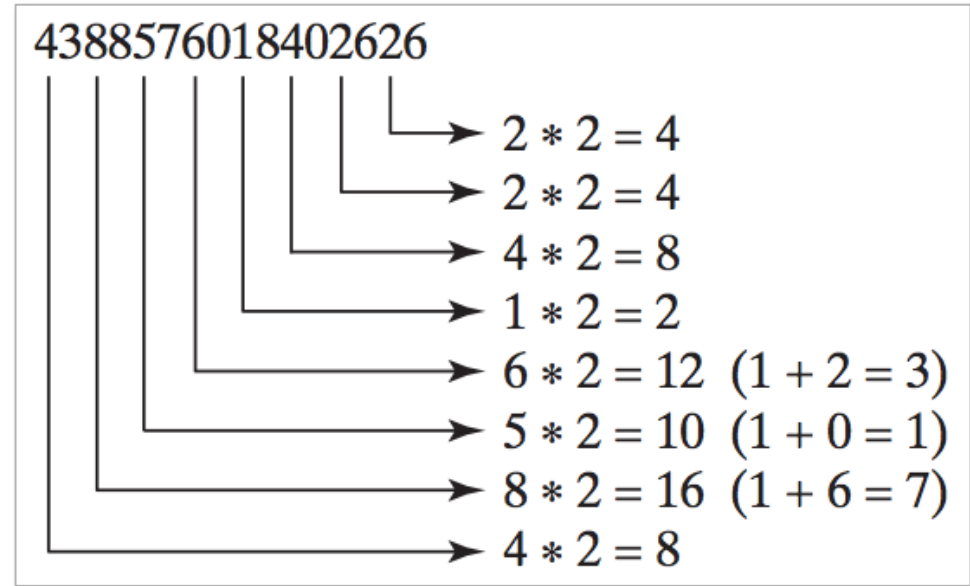
- (ii) Add all digits in the odd places from right to left in the card number:

$$6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38$$

Sum the results from (i) and (ii) above:

$$37 + 38 = 75$$

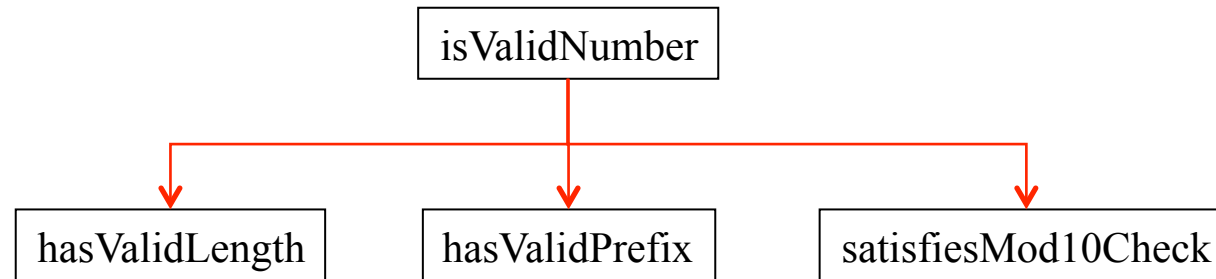
If the sum is divisible by 10, the card number is valid; otherwise, it is invalid.



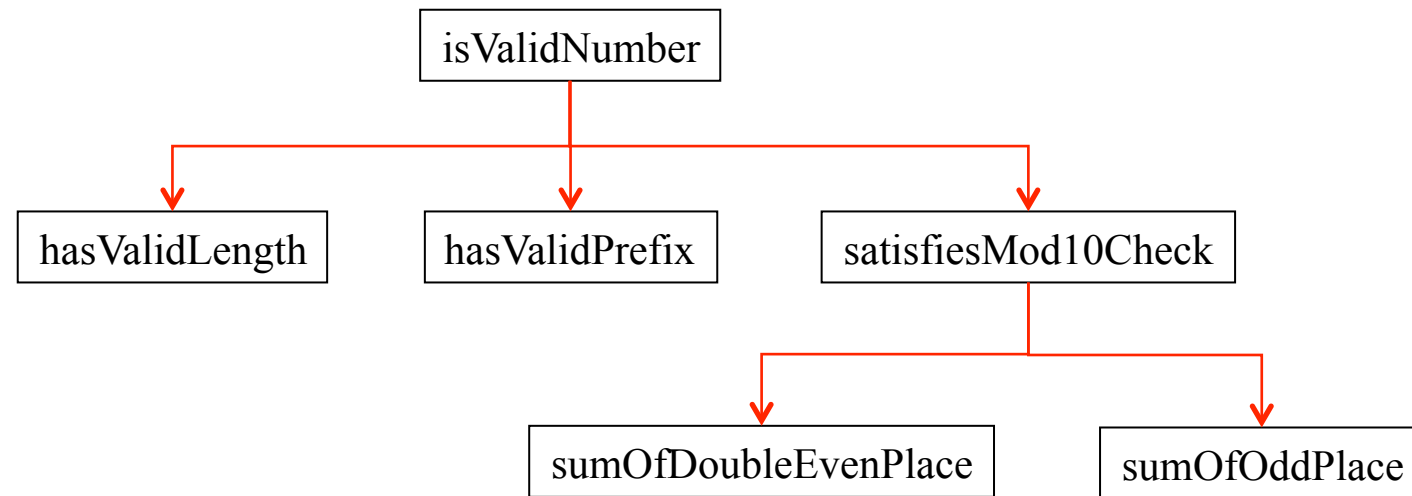
Design Diagram

isValidNumber

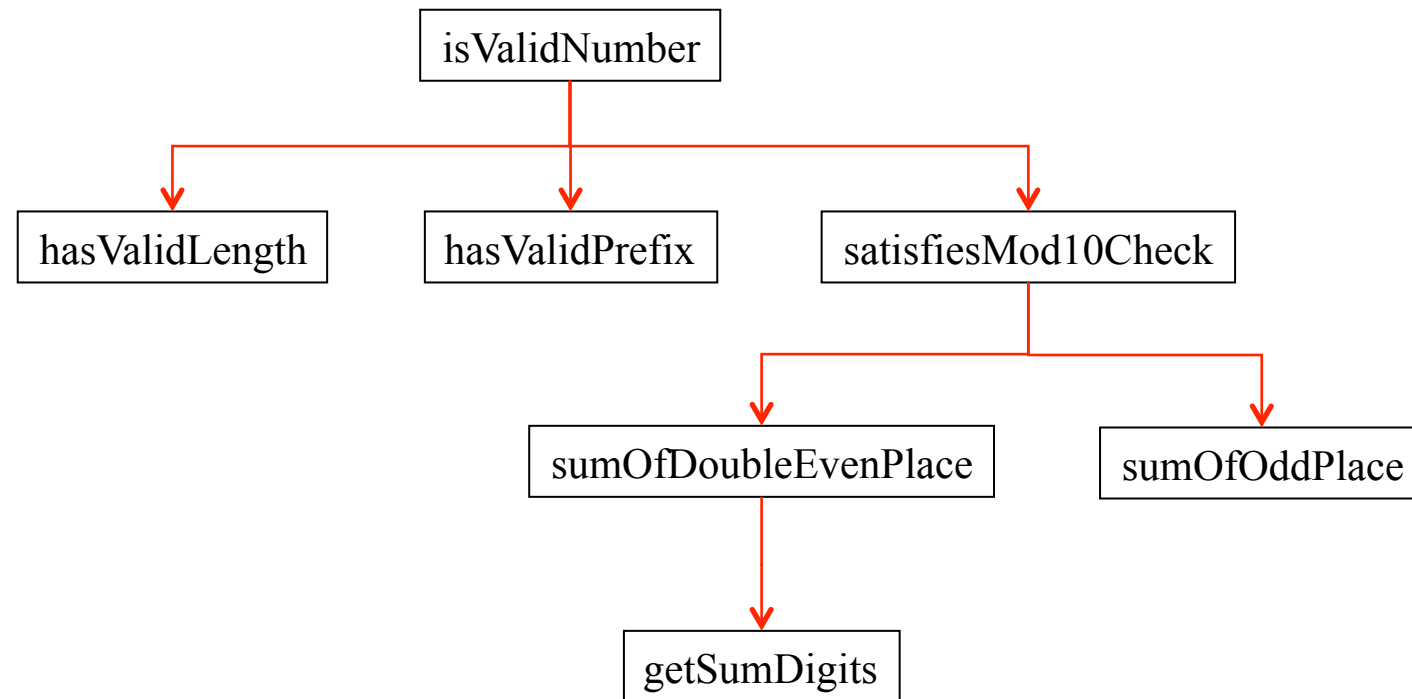
Design Diagram



Design Diagram



Design Diagram



Methods

```
// Return true if the card number is valid
public static boolean isValidNumber(String number)

// Return true if the card number has between 13 and 16 digits
public static boolean hasValidLength(String number)

// Return true if the card number has a valid prefix
public static boolean hasValidPrefix(String number)

// Return true if the Mod 10 check is satisfied
public static boolean satisfiesMod10Check(String number)

// Double every second digit from right to left and return sum
public static int sumOfDoubleEvenPlace(String number)

// Return sum of digits in odd places from right to left
public static int sumOfOddPlace(String number)

// Return this number if it is a single digit;
// otherwise return the sum of the two digits
public static int getSumDigits(int number)
```

Implementation: Top-Down

The top-down approach is to implement one method in the design diagram (structure chart) at a time from the top to the bottom.

Stubs can be used for the methods waiting to be implemented. A stub is a simple but incomplete version of a method.

The use of stubs enables you to test invoking the method from a caller.

Implement the top method first and then use a stub for the other methods...

For example, your program to validate credit card numbers might begin like this...

ValidateCCStubs

Implementation: Bottom-Up

The bottom-up approach is to implement one method in the design diagram (structure chart) at a time from the bottom to the top.

For each method implemented, write a test program to test it.

Both top-down and bottom-up methods are fine. Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy.

Benefits of Stepwise Refinement

Simpler programs

Reusing methods

Easier developing, debugging, and testing

Better facilitating teamwork

Next Topics

Chapters 7 & 8:

- Single-dimensional & multidimensional arrays...