Methods

- Method (a.k.a. function, procedure, or routine):
  - Piece of code that carries a specific computation
  - Can be called (executed) from anywhere in the code (if they are public)
  - Can take one or more parameters (arguments) as input
  - Can return a value (or an array, or any object)

- Local variables:
  - Variables declared inside a method (e.g. s).
  - They are discarded after the method finishes being executed.

Why are methods useful?

- **Code re-use**: a method can be called (executed) as often as we want, from anywhere in the program. No need to duplicate code.

- **Encapsulation**: Allows to think of a piece of code as a black box with a well-defined function. Users don’t need to know how the method works, only what the method does: what are its arguments, what does it return.

- **Makes program much easier to design, understand and debug**

The truth about parameter passing

- **What happens when a method is called?**
  1. The flow of execution of the code calling the method is interrupted.
  2. If the methods takes some arguments, these arguments are allocated in memory (stack). They are initialized with the value of the arguments provided by the caller.
  3. If variables are declared outside the method, they are also put on the stack.
  4. The code of the method is executed. This may include calling other methods.
  5. When the code of the method has been executed, it may return a value to the caller. All local variables and arguments created on the stack are discarded.

- **Summary: Parameters are passed by value**
  - The method called receives a *copy* of the parameters passed
  - Since it is working a copy, the method *can’t change the original* 
  - But watch out with arrays and non-primitive types...

### Parameter passing

<table>
<thead>
<tr>
<th>Memory (stack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static void main(String args[])</td>
</tr>
<tr>
<td>static double circleArea(double r) {</td>
</tr>
<tr>
<td>double a = 3.1416 * r * r;</td>
</tr>
<tr>
<td>return a;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Output: Radius: 2 Area: 12.5664

<table>
<thead>
<tr>
<th>Parameter passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class course</td>
</tr>
<tr>
<td>// prints welcoming statement. Takes no arguments. Returns nothing</td>
</tr>
<tr>
<td>public static void printWelcome() {</td>
</tr>
<tr>
<td>System.out.println(&quot;Welcome to COMP 250&quot;);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Why are methods useful?

- **Code re-use**: a method can be called (executed) as often as we want, from anywhere in the program. No need to duplicate code.

- **Encapsulation**: Allows to think of a piece of code as a black box with a well-defined function. Users don’t need to know how the method works, only what the method does: what are its arguments, what does it return.

- **Makes program much easier to design, understand and debug**

The truth about parameter passing

- **What happens when a method is called?**
  1. The flow of execution of the code calling the method is interrupted.
  2. If the methods takes some arguments, these arguments are allocated in memory (stack). They are initialized with the value of the arguments provided by the caller.
  3. If variables are declared outside the method, they are also put on the stack.
  4. The code of the method is executed. This may include calling other methods.
  5. When the code of the method has been executed, it may return a value to the caller. All local variables and arguments created on the stack are discarded.

- **Summary: Parameters are passed by value**
  - The method called receives a *copy* of the parameters passed
  - Since it is working a copy, the method *can’t change the original* 
  - But watch out with arrays and non-primitive types...

### Parameter passing

<table>
<thead>
<tr>
<th>Memory (stack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static void main(String args[])</td>
</tr>
<tr>
<td>static double circleArea(double r) {</td>
</tr>
<tr>
<td>double a = 3.1416 * r * r;</td>
</tr>
<tr>
<td>return a;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Output: Radius: 2 Area: 12.5664

<table>
<thead>
<tr>
<th>Parameter passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class course</td>
</tr>
<tr>
<td>// prints welcoming statement. Takes no arguments. Returns nothing</td>
</tr>
<tr>
<td>public static void printWelcome() {</td>
</tr>
<tr>
<td>System.out.println(&quot;Welcome to COMP 250&quot;);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Why are methods useful?

- **Code re-use**: a method can be called (executed) as often as we want, from anywhere in the program. No need to duplicate code.

- **Encapsulation**: Allows to think of a piece of code as a black box with a well-defined function. Users don’t need to know how the method works, only what the method does: what are its arguments, what does it return.

- **Makes program much easier to design, understand and debug**

The truth about parameter passing

- **What happens when a method is called?**
  1. The flow of execution of the code calling the method is interrupted.
  2. If the methods takes some arguments, these arguments are allocated in memory (stack). They are initialized with the value of the arguments provided by the caller.
  3. If variables are declared outside the method, they are also put on the stack.
  4. The code of the method is executed. This may include calling other methods.
  5. When the code of the method has been executed, it may return a value to the caller. All local variables and arguments created on the stack are discarded.

- **Summary: Parameters are passed by value**
  - The method called receives a *copy* of the parameters passed
  - Since it is working a copy, the method *can’t change the original* 
  - But watch out with arrays and non-primitive types...

### Parameter passing

<table>
<thead>
<tr>
<th>Memory (stack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static void main(String args[])</td>
</tr>
<tr>
<td>static double circleArea(double r) {</td>
</tr>
<tr>
<td>double a = 3.1416 * r * r;</td>
</tr>
<tr>
<td>return a;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Output: Radius: 2 Area: 12.5664

<table>
<thead>
<tr>
<th>Parameter passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class course</td>
</tr>
<tr>
<td>// prints welcoming statement. Takes no arguments. Returns nothing</td>
</tr>
<tr>
<td>public static void printWelcome() {</td>
</tr>
<tr>
<td>System.out.println(&quot;Welcome to COMP 250&quot;);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Why are methods useful?

- **Code re-use**: a method can be called (executed) as often as we want, from anywhere in the program. No need to duplicate code.

- **Encapsulation**: Allows to think of a piece of code as a black box with a well-defined function. Users don’t need to know how the method works, only what the method does: what are its arguments, what does it return.

- **Makes program much easier to design, understand and debug**

The truth about parameter passing

- **What happens when a method is called?**
  1. The flow of execution of the code calling the method is interrupted.
  2. If the methods takes some arguments, these arguments are allocated in memory (stack). They are initialized with the value of the arguments provided by the caller.
  3. If variables are declared outside the method, they are also put on the stack.
  4. The code of the method is executed. This may include calling other methods.
  5. When the code of the method has been executed, it may return a value to the caller. All local variables and arguments created on the stack are discarded.

- **Summary: Parameters are passed by value**
  - The method called receives a *copy* of the parameters passed
  - Since it is working a copy, the method *can’t change the original* 
  - But watch out with arrays and non-primitive types...
In assign, a = 2 and b = 2
After fakeAssign:
  a: 1  b: 2  c: 9  // because in fakeAssign, we were working on copies of the original a and b
In stupidIncrement:
  i = 3
After stupidIncrement:
  a: 1  b: 2  i: 9  // the variable i used in fakeAssign has nothing to do with the variable i defined in main
In stupidIncrement:
  i = 10
Again after stupidIncrement:
  a: 1  b: 2  i: 9
After add:
  a: 10  b: 2  i: 9

Parameter passing with arrays

```java
static void changeArray(int a[]) {
    System.out.println("First, a[0] is " + a[0]);
    a[0]=2;
    System.out.println("Then, a[0] is " + a[0]);
    a = new int[2];
    a[0]=3;
    System.out.println("Then, a[0] is " + a[0]);
}
public static void main(String args[]) {
    int[] array;
    array = new int[3];
    array[0] = 1;
    changeArray(array);
    System.out.println("Finally, array[0] is " + array[0] );
}
```

Strings

- Strings store sequences of characters
- Strings behave just like arrays (but they’re more than that)

```java
String s;  // s is a reference to a String. Currently, it’s a null String
String s = "Hello";
char c = s.charAt(1);   // c is 'e'
int l=s.length();   // l is 5
String t = s.substring(1,3);   // t is a new string with "el"
String u = "Hello";
if (u==s) System.out.println("they are ==");       // won’t be printed
if (s.equalsTo(u)) System.out.println("They are equalsTo");  // this will be
```

Memory (stack)             Memory (heap)

Strings

- Complete description of String operations:
  
  https://docs.oracle.com/javase/7/docs/api

Input/Output

- Java has a large number of ways to read in and write out data. We will use only the most basic.
- To import IO libraries, start your code with:
  ```java
  import java.io.*;    // this should be the first line of your code
  ```
- To read data from keyboard:
  ```java
  // First open a stream from which data will be read
  BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in ));
  System.out.println("Enter your name:");
  String name = keyboard.readLine();    // reads one line from the keyboard
  System.out.println("Enter your age:");
  String ageString = keyboard.readLine();
  int age = Integer.parseInt(ageString); // convert the string into an integer
  keyboard.close();  // close the stream when we are done
  ```
Output

To write data from file named "myOutput.txt":
```java
BufferedWriter myFile = new BufferedWriter(new FileWriter("myOutput.txt"));
String line="Hello my friends!";
myFile.writeLine(line);
...
myFile.close();
```

• Good tutorial on IO:

• Full documentation:
  - https://docs.oracle.com/javase/7/docs/api

Programming style and comments

- How?
  - Choose meaningful names for methods and variables. Stick to your conventions.
  - Add comments to clarify any piece of code whose function is not obvious
  - Give a short description of each method:
    - what does it do?
    - what arguments does it expect?
    - what assumptions are made?
    - what does it return?
    - side-effects?
  - Do not overcomment!

- Why?
  - Makes re-use easier (even for you!)
  - Makes finding and solving bugs easier
  - Allows others to use your code
  - Easier to convince your boss (or TA!) that your code is working
  - Easier to analyze the efficiency of the solution

Object-Oriented Programming (OOP)

- Idea: User-defined types to complement primitive types like int, float...
- Definition of a new type is called a class. It contains:
  - Data
  - Methods: Code for performing operations on this data
- Example: the class String contains
  - Data: Sequence of characters
  - Operations: capitalize, substring, compare...
- Example: we could define a class Matrix with
  - Data: an m x n array of numbers
  - Operations: multiply, invert, determinant, etc.

Why OOP?

- Think of a set of classes as a toolbox:
  - You know what each tool does
  - You don’t care how it does it
- OOP allows to think more abstractly:
  - Each class has a well defined interface
  - We can think in terms of functionality rather than in terms of implementation
- The creator of a class can implement it however he/she wants, as long the class fulfills the specification of the interface

A first example

```java
// The new type created is called SportTeam
class SportTeam {

    // The class a four members
    String homeTown;
    int victories, losses, points;

    public static void main(String[] args) {
        // we can declare variables of type SportTeam
        SportTeam expos;

        // this creates an object of typeSportTeam and expos now references it.
        expos = new SportTeam();
        expos.victories = 62;
        expos.homeTown = "Montreal";
        SportTeam alouettes = new SportTeam();
        alouettes.victories = 11;
    }
}
```

```java
class SportTeam {

    String homeTown;
    int victories, losses, points;

    // Constructors are methods used to initialize members of the class
    public SportTeam() { // constructors are declared with no return type.
        victories=losses=points=0;
        homeTown=new String("Unknown");
    }

    // Constructors can have arguments
    public SportTeam(String town) { // we can declare variables of type SportTeam
        victories=losses=points=0;
        homeTown=town;
    }

    public static void main(String[] args) {
        // now we can declare variables of type SportTeam
        SportTeam expos, alouettes;
        expos = new SportTeam();
        alouettes = new SportTeam("Montreal");
    }
}
```
public class SportTeam {
    String homeTown;
    int victories, losses, points;
    public SportTeam() { /* see previous page */}
    public SportTeam(String town) { /* see previous page */}
    // this method returns a string describing the SportTeam
    public String toString() {
        return homeTown + " : " + victories + " victories, " + losses + " losses, for " + points + " points.";
    }
    public static void main(String[] args) {
        // now we can declare variables of type SportTeam
        SportTeam expos, alouettes;
        expos = new SportTeam();
        alouettes = new SportTeam("Montreal");
        expos.victories = 62;
        alouettes.victories = expos.victories - 52;
        String report = alouettes.toString();
        System.out.println(report);
    }
}

Private vs public

• We don't want to let any part of a program access members of a class
  – It might disrupt the internal consistency of the object
    (e.g. one may increase the number of victories without increasing the number of points)
  – We want to hide as much as possible the inside of a class, to enforce abstraction.
• Solution:
  – Make these members private (they can only be used from inside the class)
  – Allow access to these members only through predefined public methods

Assignments and equality testing

Non-primitive types are just references to objects!

public class SportTeam {
    public String homeTown;  // can be changed from within any class
    private int victories, losses, points;  // can only be changed from within this class
    public SportTeam() { /* see previous page */}
    public SportTeam(String town) { /* see previous page */}
    public String toString() {
        // now we can declare variables of type SportTeam
        SportTeam expos, alouettes;
        expos = new SportTeam();
        alouettes = new SportTeam("Montreal");
        expos.victories = 62;
        alouettes.victories = expos.victories - 52;
        String report = alouettes.toString();
        System.out.println(report);
    }
}

public class League {
    int nbTeams;
    public SportTeam teams[];   // an array of SportTeam
    League(int n) {       // constructor
        nbTeams = n;
        for (int i = 0 ; i < n ; i++) teams[i] = new SportTeam();
    }
    public static void main(String[] args) {
        League NHL = new League(30);
        NHL.teams[0].hometown = "Montreal";
        NHL.teams[0].addWin();
        System.out.println(NHL.teams[0].toString());
        System.out.println(NHL.teams[0].toString());
    }
}

This

• Sometimes, it can be useful for an object to refer to itself:
  – the this keyword refers to the current object
• We could rewrite the constructor as:
  public SportTeam() {
    this.victories = this.losses = this.points = 0;
    this.homeTown = new String("Unknown");
  }
• If there was a league object that needed to be updated:
  – league.addTeam(this);
Static members

• Normally, each object has its own copy of all the members of the class, but...
• Sometimes we want to have members that shared by all objects of a class
• The **static** qualifier in front of a member (or method) means that all objects of that class share the same member

```java
public class SportTeam {
    public String homeTown;
    private int victories, losses, points;
    static public double exchangeRate; /* all objects of type SportTeam share the same exchangeRate */
    public SportTeam() { /* see previous page */}
    public SportTeam(String town) { /* see previous page */}
    public SportTeam(town) { /* see previous page */}
    public void addWin() { /* see previous page */}
    public static void main(String[] args) {
        // now we can declare variables of type SportTeam
        SportTeam expos, alouettes;
        expos.exchangeRate = 1.57; /* static members can be used without an actual object */
        expos = new SportTeam();
        alouettes = new SportTeam("Montreal");
        expos.exchangeRate = 1.58; // or from one particular object
        System.out.println("Rate from expos: " + expos.exchangeRate);
        System.out.println("Rate from alouettes: " + alouettes.exchangeRate);
    }
}
```

Inheritance

• Suppose you need to write a class X whose role would be very similar to an existing class Y. You could
  – Rewrite the whole code anew
    • Time consuming, introduces new bugs, makes maintenance a headache
  – Copy the code of Y into X, then make your changes
    • Maintenance problem: you need to maintain both X and Y
  – Inherit the code from Y, but override certain methods
    • Code common to X and Y is kept in Y. New methods are added in X

Inheritance - Example

• You want to extend SportTeam to make it specific to certain sports
  – HockeyTeam
    • Has all the members defined in sportTeam, but also number of ties.
    • Number of points = 3 * victories + 1 * ties
  – BaseballTeam
    • Has all the members defined in SportTeam, but also number of homeruns

```java
public class HockeyTeam extends SportTeam {
    private int ties;
    public HockeyTeam() { // constructor for HockeyTeam
        super(); // super() calls the constructor of the superclass
        ties = 0;
    }
    public void addWin() {
        super.addWin(); // This calls the addWin method provided by the parent class
        points++; // Since points is private, this wouldn't compile. We need to declare points as "protected" instead of private to allow access to subclasses
    }
    public void addTie() {
        ties++;
        points++;
    }
}
```
Types and dispatch

```java
public static void main(String[] args) {
    HockeyTeam habs;
    habs = new HockeyTeam();
    habs.hometown = "Montreal";
    habs.addWin();  // The addWin method called is the one from HockeyTeam. habs.points is 3
    System.out.println(habs.toString());  // This is legal because HockeyTeam is a subtype of SportTeam
    SportTeam bruins = new HockeyTeam();  // This is legal because HockeyTeam is a subtype of SportTeam
    bruins.addWin();  // bruins.points is now 3
    HockeyTeam leafs = new SportTeam();  // This is NOT legal because SportTeam is not a subtype of HockeyTeam
}
```

Exceptions - When things go wrong

- Some things are outside programmer's control:
  - User types "Go expos" when asked to enter number of victories
  - Try to open a file that doesn't exist
  - Try to compute sqrt(-1)
- Exception mechanism allows to deal with these situations gracefully
  - When problem is detected, the code throws an exception
  - The execution of the program stops. JVM looks for somebody to catch the exception
  - The code that catches the exception handles the problem, and execution continues from there
  - If no code catches exception, the program stops with error message
- An exception is an object that contains information about what went wrong.

Throwing exceptions

```java
static double mySqrt(double x) {
    try {
        if (x<=0) throw new ArithmeticException("Sqrt is defined only for positive numbers");
    } catch (ArithmeticException e) {
        System.out.println("The mySqrt operation failed with error:") + e
        return 0;
    }
}
```

Methods throwing exceptions

- Sometimes, it is not appropriate for a method to handle the exception it throws
- Methods can throw exceptions back to the caller

```java
public static void main(String[] args) throws ArithmeticException {
    double x = 0, y = 0, z = 0;
    try {
        x = mySqrt(10);
        y = mySqrt(-2);
        z = mySqrt(100);
    } catch (ArithmeticException e) {
        System.out.println(e.toString());
    }
    // what is the value of x, y, z now?
    // x is 1, y and z are zero
}
```

Java resources

- Java Application Programming Interface (API)
  - http://docs.oracle.com/javase/7/docs/api/
- Java books: 1594 different books on Amazon
  - The Java Programming Language -- by Ken Arnold (Author), et al; By the authors of Java itself. The ultimate reference. Not easy to read for beginners.
  - Java in a Nutshell, Fourth Edition, by David Flanagan; A text version of the Java API