

(4) Basics of the Java Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

- (4) Basics of the Java Programming Language
 - Multi-File Projects with Gradle
 - Procedural Programming
 - Predefined and User defined Methods
 - Definition of Methods
 - Procedural and recursive Method Calling
 - Unit Tests with JUnit
- Cited Literature:
 - Just Java, Peter van der Linden
 - Bruce Eckel, Thinking in Java

Understanding Reusability

- Repetition of code is really bad!
 - We have to copy code to reuse it, or to share it with other developers.
 - If repeated code contains a bug, this bug is present in all occurrences of that code!
 - In general: Don't Repeat Yourself! DRY
- We already discussed the usage of loops to support the DRY idea, e.g. to repeatedly query numbers from the user:

```
Scanner inputScanner = new Scanner(System.in);

System.out.println("Please enter a number.");
System.out.println("The number should be greater than ten.");
int number1 = inputScanner.nextInt();
System.out.println("You entered "+number1+"!");

System.out.println("Please enter a number.");
System.out.println("The number should be greater than ten.");
int number2 = inputScanner.nextInt();
System.out.println("You entered "+number2+"!");

System.out.println("Please enter a number.");
System.out.println("The number should be greater than ten.");
int number3 = inputScanner.nextInt();
System.out.println("You entered "+number3+"!");
```

// This for-loop executes the block below for three times.

```
for (int i = 1; i <= 3; ++i) {
    System.out.println("Please enter a number.");
    System.out.println("The number should be greater than ten.");
    int number = inputScanner.nextInt();
    System.out.println("You entered "+number+"!");
}
```

- But what if another programmer wants to use this code in his own program/java-file and own *main()*-method?
 - In fact we implemented DRY aware code, but it is not yet reusable! This can be achieved by programming own methods!

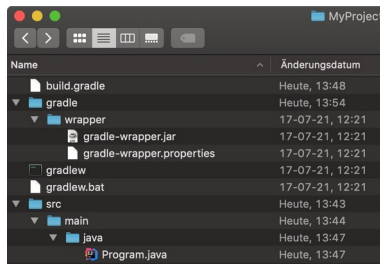
Multi-File Projects, Dependencies and Builds – Part I

- On the next slides we'll discuss Java programs, which need more than one file containing code to work.
 - Esp. will the code contained in one file refer to code defined in another file.
- This is not a problem in Java, h-files or similar are not required. – Java can automatically resolve dependencies.
- But soon, difficulties arise, which make handling programs hard, if they have many dependencies:
 - (1) If we want to use 3rd party libraries: our project depends on other libraries then.
 - (2) If we want to add tests, which usually reside in other files. – The test frameworks mostly reside in 3rd party libraries (see (1)).
 - (3) If we want to package the compiled result to a get an artifact.
 - (4) If we want to publish the artifacts we have created to, e.g., a public or company-internal repository.
 - (5) If we want to explicitly perform dedicated tasks, e.g. "only compile", "compile and tests" etc.
 - (6) If we want to have tasks run in a continuous integration (ci) environment.
- There are basically two ways to resolve at least some of these issues:
 - (1) We can use an IDE, which usually let's us do all the above with support in its user interface.
 - (2) We can use a build management tool, which let's us do all of the above via configuration files and command line execution.

Multi-File Projects, Dependencies and Builds – Part II

- For the time being we'll stick to using a build management tool rather to IDEs.
 - It is a harder to use, but most of the tools work on any platform, so they have foreseeable behavior/requirements on the system.
 - In opposite to IDEs, which look and behave differently on each OS.
- We will use the Gradle build management tool (in short "Gradle") for this course.
 - Gradle assumes a certain directory structure for a Java project.
 - The directory structure, which Gradle accepts by default, was derived from yet another build system, namely Apache Maven.
 - Apache Maven and Gradle are build systems, which exist in parallel as established standards in the Java world.
 - Dependencies and tasks, going beyond simple Java console programs are written in gradle-files (i.e. files with the (gradle-suffix)).
 - A gradle-file essentially contains Groovy code, that expresses what a build should do in a mighty Domain Specific Language (DSL).
 - Gradle's abilities can be extended via plugins, e.g. to create project-files for other IDEs or to compile other JVM languages.
 - E.g. the plugin "idea" adds the ability to create project files for the IntelliJ IDEA IDE with the Java project managed by our Gradle project.
- To start using Gradle we have to download and install Groovy and get the Gradle-wrapper.
 - For upcoming slides we assume this was done.
 - Further we assume Gradle 4.10.2.

The Gradle (Default) Directory Structure



- The Java project we want to discuss resides in the directory "MyProject".
 - Gradle awaits the Java source code in the directory <projectDirectory>/src/main/java, i.e. MyProject/src/main/java.
 - The (very small) Gradle-wrapper (gradlew) subsystem must be put into the directory <projectDirectory>/gradle/wrapper.
 - The Gradle-wrapper will automatically install the correct version of Gradle, when any Gradle task is started.
 - The file gradle-wrapper.properties allows configuration of the Gradle version to use etc., gradlew cares for this.
 - => Those files are so small, that they are meant to be copied around and even committed into source code management systems like git.
 - The gradlew start-up scripts must be directly put into the directory <projectDirectory>.
 - gradlew is the startup-script for unixoid shells and gradlew.bat is the one for the Windows command line.
- The most important file is build.gradle, it must be created/edited according our needs for our project ... let's take a look!

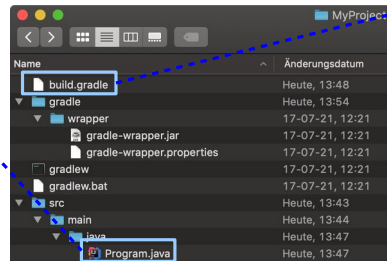
The build.gradle File

- The content of build.gradle is very simple in our case:
 - The plugin "java" is applied, which enables Java compilation.
 - The plugin "application" is applied, it enables running Java applications.
 - For the "application" plugin to work, we have to specify the *mainClassName*.

```
// <build.gradle>
apply plugin: 'java'
apply plugin: 'application'

mainClassName = 'Program'
```

```
// <Program.java>
public class Program {
    public static void main(String[] args) throws Exception {
        System.out.println("Hello World!");
    }
}
```



Running Gradle via Command Line – Part I

- With this Gradle setup we can compile and execute *Program.main()* via gradlew on the command line:

```
Terminal
NicosMBP:MyProject nico$ ./gradlew run
:compileJava
:processResources UP-TO-DATE
:classes
:run
Hello World!
BUILD SUCCESSFUL
Total time: 2.865 secs
NicosMBP:MyProject nico$
```

- When gradlew/gradlew.bat is called, the respective arguments (like run) are automatically relayed to the configured Gradle version.
 - The Gradle system does automatically compile the Java code using javac underneath.
 - Then it starts the program contained in the class *Program* (i.e. the "main-class") using java underneath.
- When we run Gradle with the "run" task, Gradle automatically also compiles the Java code, i.e. there is a dependency.
 - Gradle allows to define own tasks and declare dependencies between them.
 - We will not discuss all possibilities we have with programming own tasks in Gradle, the possibilities are almost without limit.

8

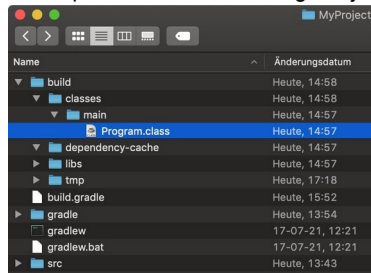
- The very first build with Gradle will download the Gradle-runtime and libraries as specified in the file gradle-wrapper.properties. Therefor we'll see such messages on the command line:

Terminal

```
NicosMBP:MyProject nico$ ./gradlew run
Downloading https://services.gradle.org/distributions/gradle-4.10.2-all.zip
. . . . .
Welcome to Gradle 4.10.2!
NicosMBP:MyProject nico$
```

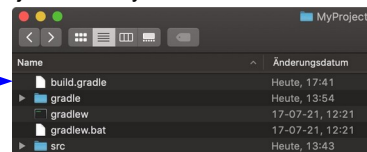

Running Gradle via Command Line – Part II

- Because with the task "run" Gradle also compiles our code, i.e. Program.java, it'll of course yield a respective class-file:



- Gradle creates the directory <projectDirectory>/build by default, it contains resulting class-files and other build output.
 - The Gradle task "build" compiles our code and creates a jar-file, which contains our class(es) in <projectDirectory>/build/lib.
- Gradle also provides the task "clean", which removes the directory <projectDirectory>/build:

```
Terminal
NicosMBP:MyProject nico$ ./gradlew clean
:clean
BUILD SUCCESSFUL
Total time: 2.865 secs
NicosMBP:MyProject nico$
```



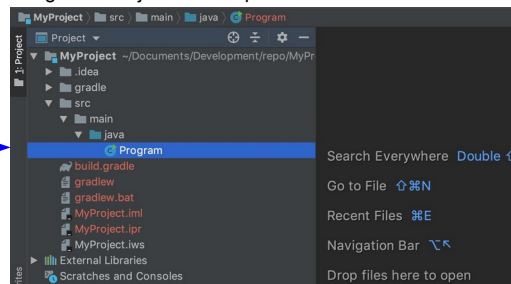
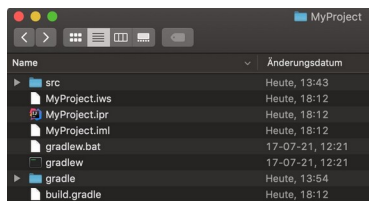
Creating IntelliJ IDEA Project Files with Gradle

- We can use the Gradle plugin `"idea"` and start gradlew with the task `"idea"` to create project files for the IntelliJ IDEA IDE:

```
// <build.gradle>
apply plugin: 'java'
apply plugin: 'application'
apply plugin: 'idea'
mainClassName = 'Program'
```

```
Terminal
NicosMBP:MyProject nico$ ./gradlew idea
:ideaModule
:ideaProject
:ideaWorkspace
:idea
BUILD SUCCESSFUL
Total time: 3.29 secs
NicosMBP:MyProject nico$
```

- After the task `"idea"` ran, we will find the files `<ProjectName>.ipr`, `<ProjectName>.iws` and `<ProjectName>.iml`.
 - ... and we can directly open the project in IntelliJ IDEA by opening the `<ProjectName>.ipr`-file.



10

- The `"idea"` plugin also offers several `"cleanIdeaXXX"`-tasks. E.g. `"cleanIdea"` will remove the IntelliJ IDEA project configuration files, but will keep the `iws`-file, the workspace-file, which contains information about personal settings of the project including run configurations. The `iws`-file can also be removed with the task `"cleanIdeaWorkspace"`.

Imperative Programming has Reusability-Limits

```
System.out.println("Please enter a number:");
System.out.println("The number should be greater than ten:");
int number = inputScanner.nextInt();
System.out.println("You entered "+number+"!");
```

- But what if programmer B wants to use this code in his own program/.java-file "ProgramB" and own *main()*-method?
 - The code needs to be transported from our program "ProgramA" to his "ProgramB"
 - The problem of this code: it is not reusable! – Therefore we need to copy the code from "ProgramA" to "ProgramB":

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    public static void main(String[] args) throws Exception {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        System.out.println("You entered "+number+"!");
    }
}
```

```
// <ProgramB.java>
import java.util.Scanner;

public class ProgramB {
    public static void main(String[] args) throws Exception {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        System.out.println("You entered "+number+"!");
    }
}
```

copied
code

- But we have just heard, that repetition of code is really bad:
 - We have to copy code to reuse it, or to share it with other developers. We call this "copy-paste reuse".
 - If repeated code contains a bug, this bug is present in all occurrences of that code!
 - In general: Don't Repeat Yourself! DRY

11

- When a piece of code, which is repeated all over in the code needs a bugfix or other modifications (e.g. so called refactoring) this is sometimes called shotgun refactoring. The term underscores the fact, that pieces of code spread all over the program must be modified, an effect like firing a shotgun, which spreads its pellets.

Now looking back: A Gradle Project with multiple Files

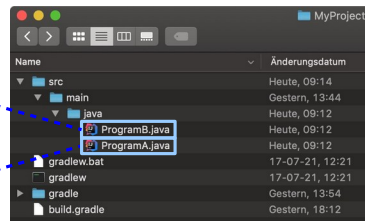
- Now we have a project of multiple java-files, they all go into <projectDirectory>/src/main/java according Gradle standards:

```
// <ProgramB.java>
import java.util.Scanner;

public class ProgramB {
    // pass
}
```

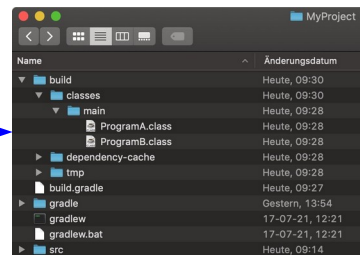
```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    // pass
}
```



- Gradle/gradlew just works as if it was a single-file project.
 - The Gradle task "compileJava" only compiles java-files:

```
Terminal
NicosMBP:MyProject nico$ ./gradlew compileJava
:compileJava
BUILD SUCCESSFUL
Total time: 2.868 secs
NicosMBP:MyProject nico$
```



- Of course tasks like "idea" will also work!

Making Code reusable with Methods – Part I

- We will gradually enhance the reusability of this code during the following slides.
 - We begin with putting the code promoting for user input into a reusable unit:

// This is the code we want to reuse:
System.out.println("Please enter a number.");
System.out.println("The number should be greater than ten.");

```
// <ProgramA.java>  
  
public class ProgramA {  
    static void printPrompt() {  
        System.out.println("Please enter a number.");  
        System.out.println("The number should be greater than ten.");  
    }  
}
```

- We put the code into a so called method with the name printPrompt().
 - In the method *main()* we now call the method *printPrompt()*, which executes the code contained in that method.
 - Methods have names for verbs, activities or actions.

Good to know

Sometime people also say, that methods are "invoked", instead of "called".

```
// <ProgramA.java>  
import java.util.Scanner;  
  
public class ProgramA {  
    static void printPrompt() {  
        System.out.println("Please enter a number.");  
        System.out.println("The number should be greater than ten.");  
    }  
  
    public static void main(String[] args) {  
        for (int i = 1; i <= 3; ++i) {  
            printPrompt();  
            Scanner inputScanner = new Scanner(System.in);  
            int number = inputScanner.nextInt();  
            System.out.println("You entered "+number+"!");  
        }  
    }  
}
```

Making Code reusable with Methods – Part II

- Actually `printPrompt()` is called for three times, i.e. for each "round" of the for loop.
 - So `printPrompt()`'s code is reused for three times.

- So in retrospective, these two programs are functionally equivalent:

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static void printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 3; ++i) {
            printPrompt();
            Scanner inputScanner = new Scanner(System.in);
            int number = inputScanner.nextInt();
            System.out.println("You entered "+number+"!");
        }
    }
}
```

Good to know

Methods are an example of HLL-patterns, that evolved from machine languages/asm. Asm jump-operations, incl. stashing of registers and writing back a result to another register, evolved to subroutines, which eventually evolved to subprograms, functions and methods.

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; ++i) {
            System.out.println("Please enter a number:");
            System.out.println("The number should be greater than ten:");
            Scanner inputScanner = new Scanner(System.in);
            int number = inputScanner.nextInt();
            System.out.println("You entered "+number+"!");
        }
    }
}
```

- A method is a bunch of statements, which can be called using a name. – This makes the reusability!
 - `printPrompt()` represents a bunch of statements printing the prompt to the console.

14

- To be frank, assembly programmers also like the idea of functions (aka sub-programs), not only to reduce DRY but also to save some bytes of code.

Anatomy of Methods

- Let's discuss the definition of the method *doSomething()*:

```
public class Program { // The obligatory class definition.
    static void doSomething () {
        // A block of code (method body).
    }
}
```

method name

- Name of methods, e.g. *doSomething()* or *printPrompt()* obey to the same rules all identifies in Java do.
 - Method names are case sensitive and are not allowed to begin with a digit etc.
 - Method names are identifiers like variables, thus they obey the same rules.
 - A method with a specific name and signature must only occur once in the same class. – We'll discuss this aspect soon.
 - The parentheses do not belong to the method names, they are written to distinguish method names from, e.g., variable names.
 - Method names in Java follow the camelCase naming convention.
 - Method names usually reads like verbs or activities like *doSomething()* or *printPrompt()*.
 - In opposite to variables, which often read like things, values or containers.
 - The name *main()* is somewhat different from this "activity-naming" scheme.


15

- Only **static** methods can be called independently from a belonging to object.

Anatomy of Method Definitions

- Hence, we'll occasionally use complete method definitions in the code snippets.

```
public class Program { // The obligatory class definition.  
    static void doSomething () {  
        // A block of code (method body).  
    }  
}
```

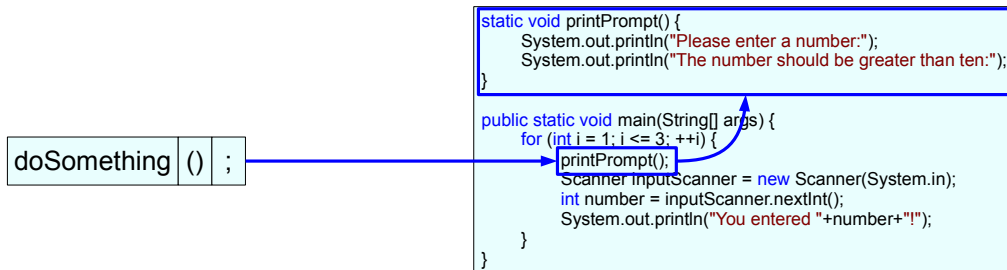


```
static void doSomething () {  
    // A block of code (method body).  
}
```

- (The obligatory `class` definition (*Program* in the example above) will also still be elided.)
 - Braces (i.e. a block) are mandatory to surround method bodies, even if a method body has only one statement!
- For the way we program procedurally in Java, the keywords `static` and `void` are required.
 - Let's just accept this for the time being.
- If the code contained in a method is not of interest, or unknown to us, we just write the comment `// pass` as a placeholder:

```
static void doSomething () {  
    // pass  
}
```


Calling Methods – Part I



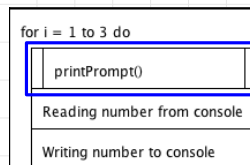
- To make use of the code contained in method, we have to call the method, then its code will be executed.
 - The call itself is just a statement and it can also be used as expression (i.e. evaluating to a value), which we'll discuss later.
 - As can be seen, we can virtually write any code in `printPrompt()` we would have written in `main()`.
 - E.g. we can call `System.out.println()` either in `main()` or `printPrompt()`.
- We call "`printPrompt()` in `main()`" or "from `main()`" -> `main()` is the caller and `printPrompt()` is the callee.
- The call expression is put into effect by just writing the name of the method we want to call and a pair of empty parens.

Calling Methods – Part II

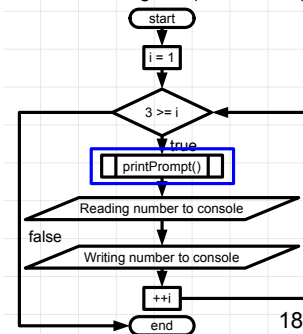
- With the introduction of methods, we leave the ground of flowchart diagrams and NSDs.
 - The idea of those diagram types is to show the imperative code in a method.
 - In other words: a method encapsulates imperative code.
- However, flowchart diagrams and NSDs have symbols to represent method calls from within imperative code:
 - In both diagram types method calls, formally procedure calls, are represented by boxes having left and right borders.

```
static void printPrompt() {  
    System.out.println("Please enter a number.");  
    System.out.println("The number should be greater than ten.");  
}  
  
public static void main(String[] args) {  
    for (int i = 1; i <= 3; ++i) {  
        printPrompt();  
        Scanner inputScanner = new Scanner(System.in);  
        int number = inputScanner.nextInt();  
        System.out.println("You entered "+number+"!");  
    }  
}
```

NSD (method call)



Flowchart Diagram (method call)



The Theory behind Procedural Programming

- In engineering, complex problems are getting separated into smaller problems.
 - In programming, complex code is broken into subroutines, subprograms, procedures or methods.
 - Then we have a piece of code, which can be used to solve a "category" of problems.
 - We call this the "Top-Down" approach, or functional decomposition.

Definition

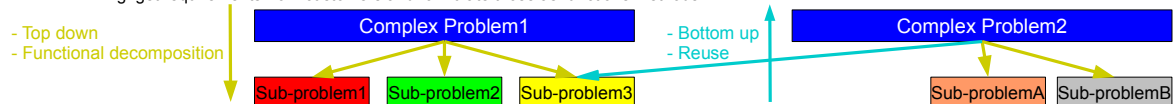
*Procedural programming means to program considering a problem as being expressed by reusable sub-problems, so called **procedures**.*

- Procedural programming is all about the ideas of "Top-Down" and "Bottom-Up".

Good to know:

Procedure from latin *procedere* – to go forward.

- Bottom-Up: Programmers code new functions/methods that solve sub-problems.
 - These functions/methods can then be reused, i.e. called, by other complex code to solve its sub-problems of the same category.
- Top-Down: Programmers decompose their code into functions/methods to make sub problems solvable and make code reusable.
 - E.g. get requirements from customers and formulate those as functions/methods.



- Procedural programming extends imperative programming with the concept of (reusable) functions/methods.
 - Reusability of code from the perspective of procedural programming is the ability to call functions/methods from other functions/methods.
 - In functions/methods, statements can be executed imperatively.
 - And in functions/methods, further functions/methods can be called as well.

The JDK – Using predefined Methods in own Code

- Additionally to coding methods, a programmer's activity is also to collect methods in [classes](#).
 - A collection of methods into [classes](#) and collections of [classes](#) into [packages](#), which solve problems of the same category is called [library](#).
 - We already discussed the [class](#) *Math*, which is a library of math-related methods.
- Java comes with a huge set of libraries, which together make up the [Java Development Kit \(JDK\)](#).
 - As Java's standard library, the JDK [provides a plethora of methods and classes](#) for us.
 - The JDK designers already had their "Top-Down" specs on... :-)
 - These methods and [classes](#) allow [reusing foreign code](#) (the code written by the JDK developers) [to solve our own problems](#).
- The most important JDK method we've used up to now is *System.out.println()*, mind, how we used the dot-notation!

```
// Well, it's as simple as this:  
System.out.println();
```

- We also used the method *nextInt()* of the type/class *Scanner*, where we also applied the dot-notation.
 - But there was another specially concerning *Scanner* and *nextInt()*: [we had to import the package java.util!](#)

```
import java.util.*; // Import the package java.util to make the type Scanner available!  
  
public static void main(String[] args) throws Exception {  
    Scanner inputScanner = new Scanner(System.in);  
}
```

- Java [classes](#) and [packages](#) are Java's [constructs to organize libraries](#). We'll discuss those in a future lecture!

Calling Methods of other Classes

- Now, we are able to call `printPrompt()` from another Java program/class, e.g. from `ProgramB`:
 - For the time being this is only possible, if `ProgramA.java` and `ProgramB.java` reside in the same directory.

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static void printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
    }

    public static void main(String[] args) throws Exception {
        printPrompt();
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        System.out.println("You entered "+number+"!");
    }
}
```

```
// <ProgramB.java>
import java.util.Scanner;

public class ProgramB {
    public static void main(String[] args) throws Exception {
        ProgramA.printPrompt();
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        System.out.println("You entered "+number+"!");
    }
}
```

- `ProgramB.main()` applies the dot-syntax to prefix the name of the class, `ProgramA`, in which `printPrompt()` is defined.
 - We say, the method name `printPrompt()` is qualified with the name of the class, in which it is defined (`ProgramA`).
 - This is required, because the compiler needs to know, that the method `printPrompt()` is defined in the class `ProgramA`.
 - The qualification is not required, if `printPrompt()` is called from within `ProgramA`.

The local Variable Scope – Part I

- Let's continue to enhance the reusability of our program just by increasing the amount of decomposed code.
 - In the next step, we'll put scanning and reading the user's input into the method *printPrompt()* along with the prompt itself:

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static void printPrompt() {
        System.out.println("Please enter a number.");
        System.out.println("The number should be greater than ten.");
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
    }

    public static void main(String[] args) throws Exception {
        printPrompt();
        System.out.println("You entered " + number + "!");
    }
}
```

- But this code will not compile!
 - The compiler has a problem here, because in *main()* we refer to a variable *number*, which is not visible in *main()*!
 - The compiler message reads "java: cannot find symbol".
 - number* is defined in *printPrompt()* and is only visible in the scope of *printPrompt()*.
- We say, the variable *number* is a local variable (in short "a local") in *printPrompt()*.

The local Variable Scope – Part II

- Up to now, we have only discussed local variables, which are only known to the method, in which they are defined.
 - We can also say, that a variable defined in a method is only visible in the local scope (of that method).
 - Remember, that a scope defines when a variable known to its environment and to which environment is it known.
- Local variables ("locals") are only known in the method, in which they were defined!

```
// <Program.java>
public class Program {
    static void methodA() {
        String name = "My Name";
        int numberA = 41;
        System.out.println(numberA);
        // >41
    }

    static void methodB() {
        methodA();
        int numberB = 234;
        System.out.println(numberB);
        // >234
        System.out.println(numberA); // Invalid! numberA not visible here!
    }
}
```

- Calling *methodA()* will not make its locals name or *numberA* visible to *methodB()*!
- Also the written order of the methods or variables or types is irrelevant for the variables' scope or "locality".

The Idea of "global" Variables – Part I

- We can solve the scoping problem of the local variable *number* by making it a "more global" variable.
- It means, that we simply put *number* into a scope, which is common to *main()* and *printPrompt()*.

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static void printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
    }

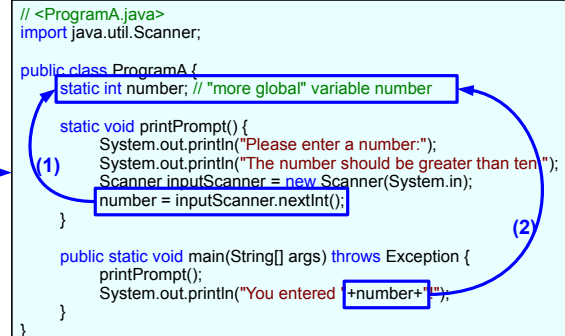
    public static void main(String[] args) throws Exception {
        printPrompt();
        System.out.println("You entered "+number+"!");
    }
}
```

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int number; // "more global" variable number

    static void printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        number = inputScanner.nextInt();
    }

    public static void main(String[] args) throws Exception {
        printPrompt();
        System.out.println("You entered "+number+"!");
    }
}
```



- Now, *number* is visible to *main()* and *printPrompt()*, because *number*, *main()* and *printPrompt()* reside in the same scope.
- (1) *number* is written in *printPrompt()*, i.e. the user's input is stored in *number*. We say *printPrompt()* has a side-effect on *number*.
- (2) And *number* is read in *main()*, where *number* is printed to the console.
- More exactly, *number*, *main()* and *printPrompt()* reside in the same scope now.

The Idea of global Variables – Part II – the class-Scope

- To put `number`, `main()` and `printPrompt()` into the same scope, syntactically
 - we moved the definition of the variable `number` from the method body to the body of the `class`
 - and we prefixed the definition of the variable `number` with the `static` keyword, we'll clarify the need for `static` in a future lecture.
 - Apart from these changes, `number` is still an ordinary `int` variable.

- We can elegantly highlight the common scope, the `class-scope`, of `number`, `main()` and `printPrompt()`:

- The order of symbol definitions in `class-scope` doesn't matter! i.e. `number` could also be defined right below `printPrompt()`, but it would still be visible to `printPrompt()` (and `main()` of course).
- The `class-scope` cannot contain more than one variable of the same name or more than one methods with the same name and signature.
 - If such a "duplicity", a so-called `name clash` happens, we get a compile time message.

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int number;

    static void printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        number = inputScanner.nextInt();
    }

    public static void main(String[] args) throws Exception {
        printPrompt();
        System.out.println("You entered "+number+"!");
    }
}
```

25

- Java doesn't support global variables and methods directly, but `class-scope` (`static`) symbols do exactly work like "globals".

The Idea of global Variables – Part III – the class-Scope

- After establishing the **class-scope** variable *number*, we can reuse *printPrompt()* like so:

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int number;

    static void printPrompt() {
        System.out.println("Please enter a number.");
        System.out.println("The number should be greater than ten.");
        Scanner inputScanner = new Scanner(System.in);
        number = inputScanner.nextInt();
    }

    public static void main(String[] args) throws Exception {
        printPrompt();
        System.out.println("You entered "+number+"!");
    }
}
```

```
// <ProgramB.java>

public class ProgramB {
    public static void main(String[] args) throws Exception {
        ProgramA.printPrompt();
        System.out.println("You entered "+ProgramA.number+"!");
    }
}
```

- We have created a very important "byproduct": the class *ProgramB* doesn't know, that *ProgramA* uses a *java.util.Scanner*!
 - ProgramB* doesn't **import** *java.util.Scanner*, instead it reuses *printPrompt()*, which internally uses *Scanner*.
 - ProgramA.printPrompt()* hides this implementation detail (*Scanner* as local variable) from callers, this is called **information hiding**.
- The current solution looks good, right?
 - Yeah, pretty much, but there is a very serious problem using **class-scope** variable this way!

The Idea of global Variables – Part IV – the class-Scope

- Let's temporarily replace *main()* with the **class**-scope method *printNumber()* in *ProgramA*, it just prints *ProgramA*'s **class**-scope *number*:

```
// <ProgramA.java>
import java.util.Scanner;

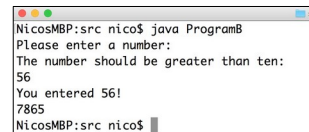
public class ProgramA {
    static int number;

    static void printPrompt() {
        System.out.println("Please enter a number.");
        System.out.println("The number should be greater than ten.");
        Scanner inputScanner = new Scanner(System.in);
        number = inputScanner.nextInt();
    }

    static void printNumber() {
        System.out.println(number);
    }
}
```

```
// <ProgramB.java>

public class ProgramB {
    public static void main(String[] args) throws Exception {
        ProgramA.printPrompt();
        System.out.println("You entered "+ProgramA.number+"!");
        ProgramA.number = 7865;
        ProgramA.printNumber();
    }
}
```



```
NicosMBP:src nico$ java ProgramB
Please enter a number:
56
The number should be greater than ten:
56
You entered 56!
7865
NicosMBP:src nico$
```

- Communication via global symbols (ok: **class**-scope symbols) can be very dangerous!
 - All methods, which can access the **class**-scope can potentially access and modify all symbols of that **class**-scope.
 - I.e. *ProgramB* can access *ProgramA*'s **class**-scope and we can modify *ProgramA.number* directly from *ProgramB*!
- ProgramA.printPrompt()* has a side effect on *ProgramA.number*, and *ProgramB.main()* has a side effect on *ProgramA.number*.
 - What we see here is, that code of different **classes** could overwrite each other's **class**-scope symbols. 27
 - In practice, this way of programming will often evolve to unmanageable code with unpredictable side effects and disasters!

Methods returning a Value – Part I

- One can say, that class-scope variables are a way for methods to communicate with its environment, e.g. its callers.
 - But: Basically the golden path to create manageable procedural code is to reduce side-effects on class-scope/global scope.
 - We just hinted how unstructured access and modification of *ProgramA.number* can lead to problems.
- Another core feature of methods beyond code reuse is the idea of methods providing a result, by returning data.
 - Therefor we're going to discuss methods, which return values, instead of modifying global values.
- Now we'll continue enhancing the reusability of *printPrompt()* by making it a method returning a value:

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        return number;
    }

    public static void main(String[] args) throws Exception {
        int number = printPrompt();
        System.out.println("You entered "+number+"!");
    }
}
```

Methods returning a Value – Part II

- The updated method *printPrompt()* shows two significant modifications:

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt(); // local variable number
        return number;
    }
    // ...
}
```

- (1) The keyword `void` was replaced by the keyword `int`. So, the method *printPrompt()* itself has a type now!
 - (2) There is a new statement in the method body, a `return` statement.
 - *printPrompt()* has no longer a side-effect on a class-scope variable, instead it `returns` a value as a result of its call.
 - So, the `return` statement is one way to get rid off side effects, the side channel of the class-scope variable has vanished!
 - Now, *number* is a local variable in *printPrompt()*.
- 29
- Let's have a closer look at the call of *printPrompt()* to understand what is going on here.

Methods returning a Value – Part III

```
// <ProgramA.java>

public class ProgramA {
    // ...
    public static void main(String[] args) throws Exception {
        int number = printPrompt();
        System.out.println("You entered "+number+"!");
    }
}
```

- When we inspect the call of *printPrompt()*, we'll notice, that its call returns a value, which can be assigned to a variable!
 - In fact, the method call of *printPrompt()* is an expression, which yields a value.
- The type of the method defines the type of the value, which is returned by the method.
 - We call that the return type of the method.

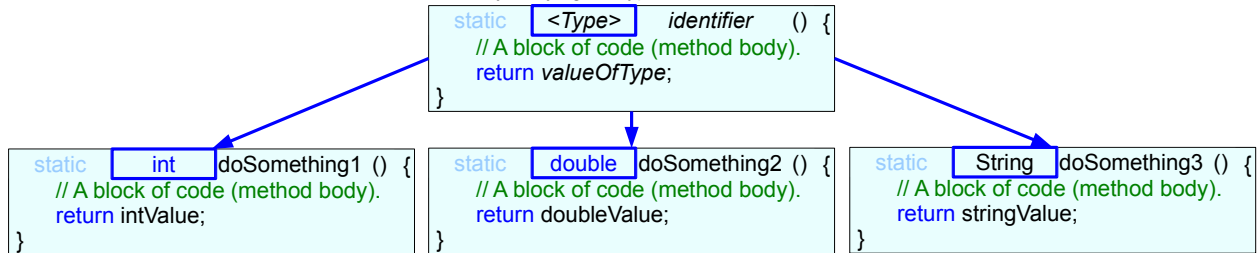
```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        return number; (2)
    }
}

public static void main(String[] args) throws Exception {
    int number = printPrompt(); (1)
    System.out.println("You entered "+number+"!");
}
}
```

Methods returning a Value – Part IV

- The idea of methods with **return** type changes the general anatomy of a method somewhat.
 - Methods can **return** a value and show this by carrying the type in front of the method name:



- In the method body is a **return** statement, which **returns** the result from the method to the caller and **ends** method execution.
 - Any primitive or user defined type can be used as **return** type.
- Methods, which do not return a value show this with the keyword **void** in front of the method name instead of a type:



- void** methods don't have a result, so they might have a side effect to communicate to its environment.

How the return Statement works – Part I

- We begin discussing **return** statements by simplifying the method `printPrompt()`.
 - The **return** statement **returns** a value to its caller.
 - Instead of having the local *number* temp. store the result of `inputScanner.nextInt()`, let's **directly return** the result of `inputScanner.nextInt()`:

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        return number;
    }
}
```

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println("The number should be greater than ten:");
        Scanner inputScanner = new Scanner(System.in);
        return inputScanner.nextInt();
    }
}
```

- Mind, that the caller of `inputScanner.nextInt()` is `printPrompt()` (`printPrompt()`'s caller is `main()`).
- On the other hand a method call is just an expression, which yields a value.
 - This yielded value can be **stored**, i.e. stuck into a variable, or otherwise **consumed**, e.g. in an expression:

```
public static void main(String[] args) throws Exception {
    System.out.println("Please enter a number");
    int enteredNumber = printPrompt();
    System.out.println("Entered number: " + enteredNumber);
}
```

```
public static void main(String[] args) throws Exception {
    System.out.println("Please enter a number");
    System.out.println("Entered number: " + printPrompt());
}
```

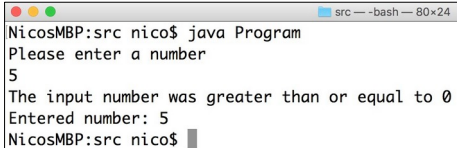

How the return Statement works – Part II

- When control flow within a method reaches a **return** statement, the method will **return** immediately to the caller method.
 - This means, that when a **return** statement is hit, no code beyond that **return** statement will be executed.
 - There are exceptions to this rule exiting the method, with so called **finally** blocks and **try-with-resource-blocks**, which we will discuss in a future lecture.

```
// <Program.java>
import java.util.Scanner;

public class Program {
    static int giveMeAnInt() {
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        if (0 <= number) {
            System.out.println("The input number was greater than or equal to 0");
            return number;
        }
        System.out.println("The input number was less than 0");
        return number;
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Please enter a number");
        System.out.println("Entered number: " + giveMeAnInt());
    }
}
```



```
NicosMBP:src nico$ java Program
Please enter a number
5
The input number was greater than or equal to 0
Entered number: 5
NicosMBP:src nico$
```

- The **int 5** was entered by the user, which is greater than 0, in this case
 - (1) Only "The input number was greater than or equal to 0" was printed to the console.
 - (2) The following **return** statement was executed, it exited from *giveMeAnInt()* and transported the **int 5** to the caller *main()*.³³

How the return Statement works – Part III

- When a value is returned from a method, basically the same like initializing a variable is happening.

```
// <Program.java>

public class Program {
    static int giveMeAnInt() {
        ↑ returning a value ≈
        ↑ implicit assignment
        return 56;
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Number: "+giveMeAnInt());
    }
}
```

- Therefore, Java allows implicit conversions while returning values from methods as we know it from initialization/assignment:
 - Remember, that implicit conversions work from smaller to larger integral types and from a less precise to more precise floaty types.
 - Also remember, that implicit conversions work also from integral to floaty types:

```
// <Program.java>

public class Program {
    static double giveMeADouble() {
        return 56; // This int will be converted to a double value.
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Number: "+giveMeADouble());
    }
}
```

How the return Statement works – Part IV – Edge Cases

- Basically, this example shows, that methods can have more than one return statement.

```
static int giveMeAnInt() {  
    Scanner inputScanner = new Scanner(System.in);  
    int number = inputScanner.nextInt();  
    if (0 <= number) {  
        System.out.println("The input number was greater than or equal to 0");  
        return number;  
    }  
    System.out.println("The input number was less than 0");  
    return number;  
}
```

- But multiple `return` statements are only allowed if written conditionally.
- If an unconditional statement follows a `return` statement, the code won't compile because the statement is unreachable:

```
static int invalidMethod1() {  
    Scanner inputScanner = new Scanner(System.in);  
    int number = inputScanner.nextInt();  
    System.out.println("Before");  
    return number;  
    System.out.println("After");  
    return number;  
}
```

- A method with a declared `return` type must have at least one `return` statement! If it is missing, the method won't compile:

```
static int invalidMethod2() {  
    System.out.println("After");  
}
```

How the return Statement works – Part V – Edge Cases

- A method, that declares a `return` type, must `return` a value on all of its control flow paths, e.g. consider:

```
static int printSomething(boolean b) {  
    if (b) {  
        System.out.println("printing something");  
        return 42;  
    }  
} // Invalid: Missing return statement
```

- In this case the compiler can clearly tell us, that not all paths return a value, here the method only returns if `b` evaluates to `true`.

How the return Statement works – Part VI – void

- The **return** statement can also be applied in methods w/o **return** type, i.e. in **void** methods!
 - As we already know, when control flow reaches the **return** statement, it will **return** immediately to the caller and exit the method.
 - This means, that if a **return** statement is hit, no code beyond that **return** statement will be executed in that method.

```
// <Program.java>
import java.util.Scanner;

public class Program {
    static void enterAnInt() {
        Scanner inputScanner = new Scanner(System.in);
        int number = inputScanner.nextInt();
        if (0 <= number) {
            System.out.println("The input number was greater than or equal to 0");
            return;
        }
        System.out.println("The input number was less than 0");
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Please enter a number");
        enterAnInt();
    }
}
```

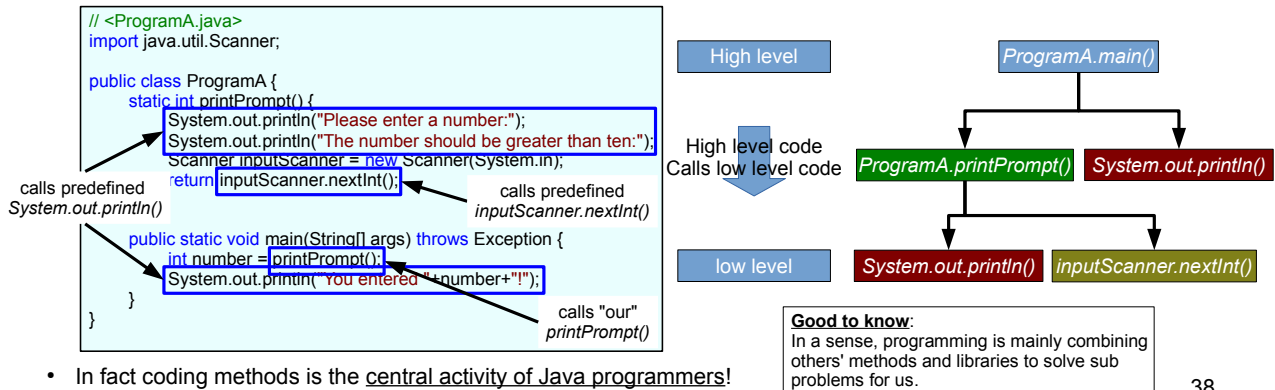
Terminal

```
NicosMBP:Debug nico$ java Program
Please enter a number
5
The input number was greater than or equal to 0
NicosMBP:Debug nico$
```

- The **int** 5 was entered by the user, which is greater than 0, in this case
 - (1) Only "The input number was greater than or equal to 0" was printed to the console.
 - (2) The following return statement was executed, it exited from **enterAnInt()** to the caller **main()**.

Procedural Programming in Java

- Procedural code in Java:
 - Program execution starts in the method *main()*, from where other methods are being called.
 - After the execution of a method ends, control flow will continue in the calling method.
 - Effectively the program ends, when the last method called from *main()* returns to *main()*.
 - I.e. a Java program is basically a hierarchy of method calls!



- In fact coding methods is the central activity of Java programmers!
 - Java programmers write new methods and call them, or they call predefined methods.

Methods with Parameters – Part I

- Let's continue to enhancing the reusability of our Program just by increasing the amount of decomposed code.
- It would be good, if we could exchange the prompted text, which is fixed to "The number should be greater than ten:" now.
 - We can achieve this with a class-scope variable of type *String*, that stores the prompted text:

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static String promptText;
    static int printPrompt() {
        System.out.println("Please enter a number:");
        System.out.println(promptText);
        Scanner inputScanner = new Scanner(System.in);
        return inputScanner.nextInt();
    }

    public static void main(String[] args) throws Exception {
        int number = printPrompt();
        System.out.println("You entered "+number+"!");
    }
}
```

```
// <ProgramB.java>

public class ProgramB {
    public static void main(String[] args) throws Exception {
        ProgramA.promptText = "The number should be greater than ten:";
        ProgramA.printPrompt();
        System.out.println("You entered "+ProgramA.number+"!");
    }
}
```

- This idea works the same, as if we use a class-scope variable to communicate the *number* via a side-effect.
 - The only difference is that we are setting the variable from *ProgramB.main()* and read it from *ProgramA.printPrompt()*.
 - But as we already know: side effects can be really bad via globals. How can we improve the situation?

Methods with Parameters – Part II

- Similar to returning values, methods support accepting values, so called arguments (args), when they are called.
- Therefore Java allows defining methods with parameters (params), which transport arguments into method bodies.
 - Let's rewrite our code to use a method with a parameter to handle flexible prompts.

```
// <ProgramA.java>
import java.util.Scanner;

public class ProgramA {
    static int printPrompt(String promptText) {
        System.out.println("Please enter a number:");
        System.out.println(promptText);
        Scanner inputScanner = new Scanner(System.in);
        return inputScanner.nextInt();
    }

    public static void main(String[] args) throws Exception {
        String promptText = "The number should be greater than ten:";
        int number = printPrompt(promptText);
        System.out.println("You entered " + number + "!");
    }
}
```

```
// <ProgramB.java>

public class ProgramB {
    public static void main(String[] args) throws Exception {
        String promptText = "The # should be greater than ten:";
        ProgramA.printPrompt(promptText);
        System.out.println("You entered " + ProgramA.number + "!");
    }
}
```

Terminal

```
NicosMBP:Debug nico$ java ProgramB
Please enter a number:
The # should be greater than ten:
```

Hint

Similar to return statements, parameters contribute to side effect free programming (e.g. programming without side channels with class-scope ("global") variables).

- Here we pass the argument *promptText* to the method *printPrompt()*.
- The output shows, that the passed argument changes the behavior of the method (console output).
- Parameters allow controlling the behavior of methods from outside by passing arguments.
 - ...in opposite to globals instead of args, where a method's behavior would be controlled via side effects.

Methods with Parameters – Part III

- After the introduction of params, we have to review the anatomy of methods.

- This is the anatomy of the case we've seen so far:

```
static void doSomething1 (String param) {  
    // pass  
}
```

parameter

- Obviously, the new thing in the anatomy is the variable *param*, the parameter, which is written in the method's parentheses.

- Java also supports methods with multiple parameters, a so called parameter list, parameter set or just parameters/params.

```
static void doSomething2 (String param, String param2) {  
    // pass  
}
```

parameter list

Hint

The params are separated by commas, not semicolons!

- Java doesn't support the variable-list notation for parameters!

```
static void doSomething2 (String param, param2) {  
    // pass  
}
```

variable-list not supported

Methods with Parameters – Part IV

- When we strip a method down to its name and ordered list of types in the param list, we get the signature of this method.

```
static void doSomething1 (String param) {  
    // pass  
}
```

signature ↓

doSomething1(String)

```
static void doSomething1 (String param, String param2) {  
    // pass  
}
```

signature ↓

doSomething2(String, String)

- So we come to a more complete, i.e. still incomplete and preliminary anatomy of a method:

```
static <Type1> identifier ([<Type2> paramIdentifier1[, <Type3> param2Identifier, ...]]) {  
    // A block of code (method body).  
    [return valueOfType1;]  
}
```

return type identifier parameter list method body

- And of its signature (the **return** type is no part of the signature):

```
identifier ([<Type2> paramIdentifier1[, <Type3> param2Identifier, ...]])
```

identifier parameter list

Features of Methods – Parameters and Arguments – Part I

- Often the terms argument and parameter are used interchangeably, but this is completely wrong!
- An argument is the value, we actually pass to a method on the caller side.
 - It doesn't matter, whether we pass a variable, a value as result of an expression or a literal: in the end only a value is passed.

```
// Variables as arguments:  
int takenPressureA = 15, takenPressureB = 78;  
int result1 = Program.sum(takenPressureA, takenPressureB); // Arguments: 15 and 78
```

```
// Results of evaluated expressions as arguments:  
int toleranceA = 5, toleranceB = 17;  
int result2 = Program.sum(takenPressureA + toleranceA, takenPressureB + toleranceB); // Arguments: 20 and 95
```

```
// Literal values as arguments :  
int result3 = Program.sum(12, 31); // Arguments: 12 and 31
```

Good to know

To drive people completely crazy, arguments are sometimes called actual parameters and parameters are sometimes called formal arguments.

- Accepting passed values on the callee side (i.e. putting them into variables) is basically the same like initializing variables.
- Java allows implicit conversions while accepting arguments and storing them in parameters.
- A parameter is a variable in a method, which holds the value of an argument, that was passed to the method.
 - When a method is called, the arguments' values are copied into its parameters.

```
static int sum(int x, int y) {  
    return x + y;  
}
```

- `sum()`'s `x` and `y` will hold the values 15 and 78 to compute `result1`, 20 and 95 to compute `result2` and 12 and 31 to compute `result3`.
- Parameters allow controlling the behavior of methods, arguments do actually control the behavior of methods. 43
 - In `sum()`, `x` and `y` allow specifying summands and the arguments are the actual summands, which control the result of `sum()`.

Features of Methods – Parameters and Arguments – Part II

- To understand params, it makes a lot of sense to discuss the caller side apart from the callee side.
- We called `printPrompt()` with an argument, i.e. the variable `promptText`. We say, that we pass `promptText` to `printPrompt()`.

```
String promptText = "The number should be greater than ten.";
int number = printPrompt(promptText);
```

- Mind, how the code reads like prosaic English: "print prompt with the prompt text"
 - This works, because the arg `promptText` is of type `String` and `printPrompt()` awaits (or accepts) an argument of type `String`.
- The flexible syntax of Java allows replace passing variables with passing literals or even calculated values.

```
int number = printPrompt("The number should be greater than ten.");
```

 - Mind, that we have called `System.out.println()` with literal `Strings` as values in many examples already!

- If a method accepts multiple params, we have to pass a list of argument values separated with commas.
 - Here the `double` values 10 and 2 are passed as a comma separated list of arguments to `Math.pow()`:

```
public class Math { // simplified, many methods hidden
    public static double pow(double base, double exponent)
        // pass
    }
}
```

```
double result1 = Math.pow(10, 2);
```

- If a method specifies params, it is required to pass all arguments to this method to satisfy all parameters!

```
double result1 = Math.pow(10); // Won't compile: actual and formal argument lists differ in length
```

Features of Methods – Parameters and Arguments – Part III

- All right, now to the callee side! So, methods can accept a list of arguments as data to control their behavior.
 - The methods' parameters store the values, which were passed as arguments, when they were called.

- Let's discuss the method `sum()`, which has two params:

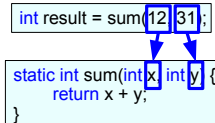
```
static int sum(int x, int y) {  
    return x + y;  
}
```

- `sum()` calculates the sum of two `ints` and `returns` that sum to the caller.

- The accepted arguments are declared as comma separated variables in the parentheses of the method.
 - And ... these variables are called parameters.
 - When a method is called, we have to pass the arguments in exactly the order, in which the "matching" params were declared.

```
int result = sum(12, 31);
```

```
static int sum(int x, int y) {  
    return x + y;  
}
```



- This is called positional argument passing: the position of the arguments must match the params by their position in the param list.

Features of Methods – Parameters and Arguments – Part IV

- Params act like local variables in the method.

- The name of the argument variable has no connection to the name of the param, it can be completely different:

```
static int printPrompt(String promptText) {  
    // pass  
}
```

```
String blaBlaBla = "The number should be greater than ten:";  
// The passed argument is a variable of name blaBlaBla, but the method's param is named promptText:  
int number = printPrompt(blaBlaBla);
```

- It means we cannot have two params with the same name! E.g. the params of the method need to have differing names:

```
static int sum(int x, int x) { // variable x is already defined in method sum  
    return x + x;  
}
```

- It also means that the params' names need to be different from the names of any other locals:

```
static int printPrompt(String promptText) {  
    String promptText = "example text"; // variable promptText is already defined in method printPrompt  
    // pass  
}
```

- Parameters can be of any Java type, i.e. primitive as well as user defined types.

- The types of params in the parameter-list can also be mixed freely, e.g. assume the method sum accepting `int` and `short`:

```
static int sum2(int x, short y) {  
    return x + y;  
}
```

- Programming languages running on the JVM like Java have to support methods with at least 255 params.

- Mind that we have to write a pair of empty parens, when we want to call a method without params!

Features of Methods – Call by value

- When a method is called, the passed arguments are copied into "their" parameters.
 - I.e. the variables used as arguments have no "connection" to the values in the method's params, this is called call by value!
 - Java does only support call by value!
 - The general opposite of call by value is call by reference.

- Practical example: Java's ability to only support call by value leads a method like `swap()` to fail:

```
// Suspicious implementation of swap()!  
static void swap(int first, int second) {  
    int temp = first;  
    // Assignment affects only the parameter!  
    first = second;  
    // Assignment affects only the parameter!  
    second = temp;  
}  
  
int a = 12, b = 24;  
swap(a, b);  
// ... but the values of a and b won't be swapped:  
System.out.println("a: "+a+", b:"+b);  
//>a: 12, b: 24
```

- Assigning to params in the method `swap()` won't affect the values in the variables, which were passed as arguments!

- Btw: `swap()` could be implemented with class-scope variables and usage of side effects!

Features of Methods – final Parameters

- To avoid "misleading" assignments to parameters, parameters can be declared as **final** variables in the parameter list.

```
// This silly method tries to modify parameters:  
static void sillyMethod(final int number) {  
    number = 80; // Won't compile: final parameter x may not be assigned  
}
```

- Mind, that **final** variables define run time constants in Java, so **final** parameters are just constants.
 - ... and we cannot assign to constants!
- final** parameters are not part of a method's signature!

```
static void sillyMethod(final int number) {  
    // pass  
}
```

signature ↓

```
sillyMethod(int)
```


Features of Methods – Method Overloading – Part I

- Quite often we want to extend existing methods, so that they support more parameters to control their behavior.
 - `sum()` is a valid example here, up to now, we can only add two summands.

- On the other hand, we can easily write a new method, `sum3()`, which adds the three passed `int` arguments together:

```
static int sum3(int x, int y, int z) {  
    return x + y + z;  
}
```

- Soon, more `sumX()`-like methods will be needed to add more and more `ints`, with a certain method naming schema:

```
static int sum4(int w, int x, int y, int z) {  
    return w + x + y + z;  
}  
static int sum5(int v, int w, int x, int y, int z) {  
    return v + w + x + y + z;  
}  
static int sum6(int u, int v, int w, int x, int y, int z) {  
    return u + v + w + x + y + z;  
}
```

```
// We have multiple sumX() methods, which carry different param lists:  
int result1 = sum(12, 13);  
int result2 = sum3(12, 13, 14);  
int result3 = sum4(12, 13, 14, 15);  
int result4 = sum5(12, 13, 14, 15, 16);  
int result5 = sum6(12, 13, 14, 15, 16, 17);
```

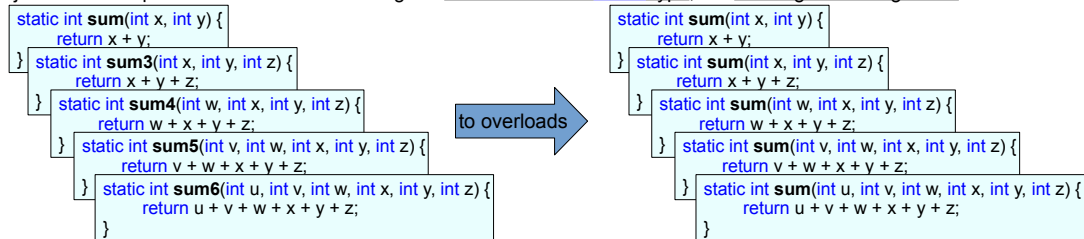
- The idea is just to encode the count of parameters in the signature in the method name.
- Among other improvements we could do to this method, we can simplify its naming schema with so called overloads.
 - In this lecture we'll only discuss overloads, but in a future lecture we'll learn about more means to esp. enhance methods like `sum()`.
- Ok, so let's discuss method overloads.

49

- Btw. selecting method overloads at run time can also be implemented in with so called "multiple dispatch".

Features of Methods – Method Overloading – Part II

- The syntactic, or declarative format of method overloads is really simple to get!
 - We'll just have multiple methods in a `class` having the same name and return type, but differing in their signature:



- Instead of differently named `sumX()` methods we have five overloads of a method with the same name: `sum()`.
 - We really only have different variants of the `sum()` method, which only differ in params, their behavior is principally the same.
 - Overloads allows methods having the same name but different parameter lists!
- Therefor overloaded method read very natural, they are simple to use, or "discover" and program:

```
// Now we call, or "reach" the method sum() having five overloads:  
int result1 = sum(12, 13);  
int result2 = sum(12, 13, 14);  
int result3 = sum(12, 13, 14, 15);  
int result4 = sum(12, 13, 14, 15, 16);  
int result5 = sum(12, 13, 14, 15, 16, 17);
```

Features of Methods – Method Overloading – Part III

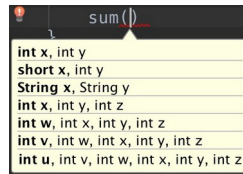
- A popular programming practice to implement overloads is delegation.
 - In this case delegation means, that an overloaded method just calls one or many of its own overloads.

```
static int sum(int u, int v, int w, int x, int y, int z) {  
    return u + v + w + x + y + z;  
}
```

reimplemented
with delegation

```
static int sum(int w, int x, int y) {  
    return w + x + y;  
}  
  
static int sum(int u, int v, int w, int x, int y, int z) {  
    return sum(u, v, w) + sum(x, y, z);  
}
```

- Usually, IDEs support a kind of "fanning out" view to show the overloads of a certain method.
 - For example IntelliJ IDEA:



- Actually, the possibilities we have using overloads don't end here!

Features of Methods – Method Overloading – Part IV

- Up to now we've only discussed overloads of methods having more and more parameters, i.e. differing param counts.
- Java also allows overloading methods with completely different parameter types.
 - E.g. let's introduce an additional overload of `sum()` accepting an `int` and a `short` argument:

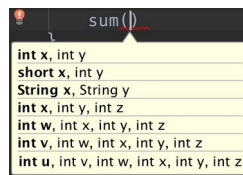
```
static int sum(int x, int y) {  
    return x + y;  
}  
static int sum(short x, int y) {  
    return x + y;  
}
```

```
int summand1 = 12, summand2 = 34;  
int result = sum(summand1, summand2);  
short shortSummand1 = 65;  
int result1 = sum(shortSummand1, summand2);
```

- Theoretically, we could really go wild about `sum()`'s overloads and add a (pointless) overload adding `String` lengths:

```
static int sum(String x, String y) {  
    return x.length() + y.length();  
}
```

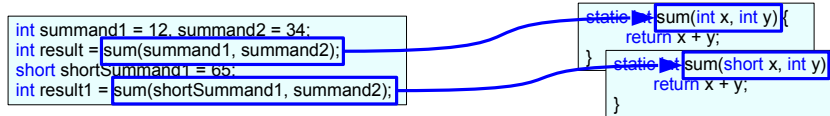
- So, we end up with a lot of `sum()` overloads:



```
sum()  
int x, int y  
short x, int y  
String x, String y  
int x, int y, int z  
int w, int x, int y, int z  
int v, int w, int x, int y, int z  
int u, int v, int w, int x, int y, int z
```

Features of Methods – Method Overloading – Part V

- The correct overloaded method to call is determined by the compiler.
 - The compiler selects the matching overload by comparing the list of passed arguments with the available signatures



- The way the compiler "discovers" the correct overloads introduces some restrictions.
- return types are no part of a method's signature! We cannot overload methods based on return types:

```
static int sum(int x, int y) {  
    return x + y;  
}  
  
// Won't compile: method sum(int,int) is already defined in class Program  
static double sum(int x, int y) {  
    return x + y;  
}
```

- final modifiers are no part of a method's signature! We cannot overload methods based on final modifiers:

```
static int sum(int x, int y) {  
    return x + y;  
}  
  
// Won't compile: method sum(int,int) is already defined in class Program  
static int sum(final int x, int y) {  
    return x + y;  
}
```

- That overloads are resolved by the compiler is very important. It means that the selection of overloads happens in a very early build phase.

Features of Methods – Method Overloading – Part VI

- Sometimes, the compiler performs unexpected overload resolutions. Let's reconsider having following *sum()* overloads:

```
static int sum(int x, int y) {  
    return x + y;  
}  
static int sum(short x, int y) {  
    return x + y;  
}
```

- Which overload of *sum()* will be called in this example?

```
int result = sum(65, 37);
```

```
static int sum(int x, int y) {  
    return x + y;  
}
```

- Indeed the overload *sum(int, int)* is called!
- Why? The arguments 65 and 37 match better with this overload, because they are *int* literals!
- The overload *sum(short, int)* is not chosen, because it would require the compiler to implicitly convert the *int* 65 to a *short*.

- The compiler primary resolves overloads, which require no implicit type conversion!

- But we can force the compiler to resolve to *sum(short, int)*!

- We do so by explicitly converting 65 to *short* with a cast:

```
int result = sum((short)65, 37);
```

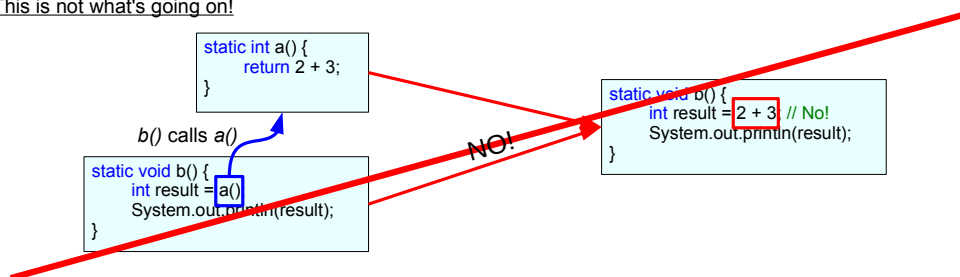
```
static int sum(short x, int y) {  
    return x + y;  
}
```

Features of Methods – Method Overloading – Summary

- So, the combination of count, order and types of a method's params makes the signature of the method.
 - With the introduction of overloads, a method can have multiple signatures.
 - Overloads of a specific method should represent similar algorithms working with different types having the same name.
- Vice versa this means, that each method overload needs to have a different signature!
 - The return type does not contribute to the signature in Java! => Overloading cannot be based on different return types!
 - The parameter names play no role for a method's naked signature.
 - The final modifier does not contribute to the signature.
- Java does not support overloading of present operators like C++.
 - Java solves the functionalities, which are solved by C++' operator overloading, with method- and interface-implementation.
 - However, Java automatically overloads the ==-operator, which compares fundamental types and all non-fundamental types!
 - The ==-operator for non-fundamental types, which are generally reference-types, performs a reference-comparison.
- Tips
 - The set of overloaded methods should really just be special cases of each other, but do not do completely different things!
 - Overloads are esp. important for so called constructors, we will discuss in a future lecture.

Methods are called – there is no Code Replacement

- Methods are called, there is no code replacement!
- A common misunderstanding: a method call leads to "code replacement" at compile time.
 - This is not what's going on!

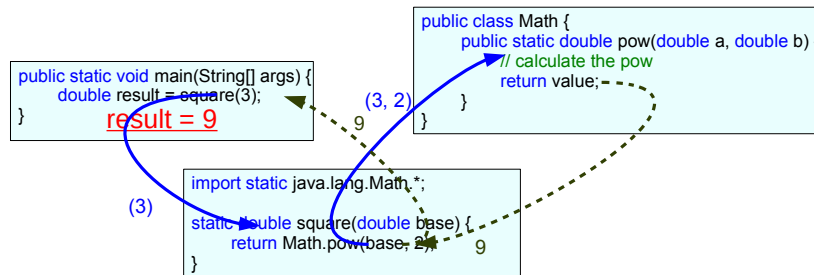


- In `b()` the call of `a()` is not replaced by the code contained in `a()` after compilation finished!
- Correct: The methods `a()` and `b()` do still exist after compilation, their code is not "merged" somehow.

- There is no code replacement going on, it would lead to code bloat, and thus bigger executables after compilation.

Procedural Calling of Methods in Action

- We already discussed that methods can call other methods.
 - When a method completes, execution returns to where the method was called from.
 - "Top-level" methods call "lower-level" methods. So, as we already recognized, there is a call hierarchy.



- There is also an alternative of procedural calling of methods: recursive calling of methods! Let's understand recursion...

Recursive Calling of Methods

- Some problems can be described as smaller instances of the same problem.
 - In maths such descriptions are called recursive descriptions.
 - For example the factorial function $n!$ ($1 \times 2 \times 3 \times 4 \dots \times n$):

$$n! = \begin{cases} 1; & n=0 \\ n \cdot (n-1)!; & n>0 \end{cases} \quad n \in \mathbf{N}$$

Good to know

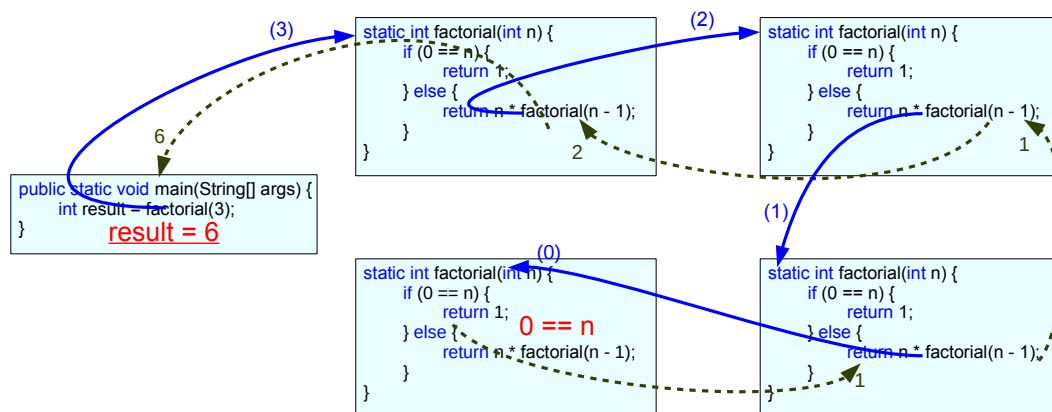
Some folks use the blackboard bold notation to name standard number sets, e.g. \mathbf{N} . Virtually this notation is only meant to be used on the board and/or with handwriting in order to simulate bold face. In print, the names of the standard number sets are just written in bold face. We'll stick to the bold face convention (**N**) instead of the blackboard bold notation (\mathbf{N}).

- Factorial can be expressed as a recursive method in Java.
 - In maths, recursion means that an equation is defined in terms of itself.
 - In Java, recursion means that a method calls itself.
- How to write a recursive method?
 - A subproblem of the whole problem must be used as argument for recursive calls.
 - And it is needed to consider the base case, when the recursion terminates!
 - (The progression of the recursive calls must tend to the base case.)
 - If we fail to consider one of these cases, we may end with an infinite recursion!

58

- Where to use the "factorial" function?
 - E.g. the factorial is used to calculate the count of permutations of a set of elements.
- Infinite recursions run as long as all the stack memory of the executing thread is exhausted.

Recursive Calling of Methods in Action



59

- In the beginning, recursion is difficult to understand for most people. – They think about all the calls of a recursive function, in order to understand, if it is correct or not. They rather understand a function as a "machine" that processes inputs and returns outputs. → A better way to understand a function is that a function is a process. E.g. "process all documents and process all documents referred to in these documents".

Various Implementations of the Factorial Algorithm

$$n! = \begin{cases} 1; & n=0 \\ n \cdot (n-1)!; & n>0 \end{cases} \quad n \in \mathbb{N}$$

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

```
static int factorialIterative(int n) {  
    int result = 1;  
    for (int i = n; i > 0; --i) {  
        result *= i;  
    }  
    return result;  
}
```

```
static int factorialNiceRecursive(int n) {  
    return (0 == n)  
        ? 1  
        : n * factorialNiceRecursive(n - 1);  
}
```

Good to know

Factorial could also be implemented with a table, which associates each n with $N!$. Such an implementation wouldn't use a loop/iteration or recursion. – The downside of this approach: we as programmers have to calculate the results and fill in the table. And btw. this table would be really large, ideally infinite in size, but practically limited by the memory of the system.

- Mind, that these implementations do not deal with negative ns ($n!$ is only defined for \mathbb{N})!
 - We'll solve this problem, i.e. fix this bug, in a future lecture!

Recursion and Recursive Functions in the Wild

- Recursion is useful to operate on recursive data. So called data trees!
 - E.g. a graphical TreeView contains trees, that contain subtrees, that contain subtrees...
 - XML and JSON data is also a cascading tree of trees.
 - Directory and file hierarchies.
 - Network analyses, e.g. calculation of network efficiency and network load.
- Complex problems and puzzles can be solved easily with recursion.
 - Maths: permutations of a set, numeric differentiation/integration
 - Games: Towers of Hanoi, Sudoku, Chess puzzles etc.
- Definition of subproblems to be solved by multiple CPUs independently.
 - The idea is to use CPU resources most efficiently -> Divide and conquer!
 - Filters in graphics programming.
 - Data analyses in networks and databases.
- Warning: recursive code is often elegant, but it comes with its payoffs.
 - Concretely, too many recursive calls will end in an exhaustion of stack memory (e.g. in Java), a so called stack overflow. 61
 - Recursive code is tricky to get for humans, because people rather think iteratively.

Functional Programming – Part I

- It makes sense to take another look to the iterative and recursive implementations of $n!$

```
static int factorialIterative(int n) {  
    int result = 1;  
    for (int i = n; i > 0; --i) {  
        result *= i;  
    }  
    return result;  
}
```

```
static int factorialRecursive(int n) {  
    return (0 == n)  
        ? 1  
        : n * factorialRecursive(n - 1);  
}
```

- `factorialIterative()`, i.e. the iterative $n!$ uses a `for` loop statement to "drive" the calculation.
 - This loop statement internally modifies the state of the variables `result` and `i`. It uses side effects on those local variables.
- The recursive $n!$ just conditionally calls itself with new values as arguments and consumes its own return values.
 - It is not driven by multiple statements. It uses one single expression. It looks like having no "moving parts".
 - For god's sake method calls are not replaced by their method body: that would be a disaster when recursion is used!
- The idea to code methods without side effects is an aspect of the paradigm of functional programming (fp).
 - Recursion is associated with fp, because recursion is used in fp to express loops without side effects.
 - Mind, that "classical", i.e. imperative loops always deal with side effects, e.g. a `for` loop incrementing its counter variable.

Functional Programming – Part II – Relation to Procedural Programming

- Procedural programming:
 - Makes us think in terms of sub programs, which modify data locally or globally.
 - It rather has a focus on side effects and variables.
 - Mentally, it rather makes us think in mathematical procedures/algorithms.
- Functional programming:
 - Makes us think in terms of functions, which are kind of closed and never modify global state.
 - It rather has a focus on argument values and return values, side effects are not existent.
 - Mentally, it rather makes us think in mathematical formulas.
- Virtually fp is a very early programming concept, which was used in the first programming languages.
 - A notable example is Lisp, which was introduced in 1958.
 - With the advent of procedural languages, fp was almost pushed away from the mainstream.
- However, today's languages re-adopted fp technologies into their programming idioms.
 - Most notable Java 8 and the .NET framework 3.5 with the LINQ technology in the languages C#3, VB9.0 and F#.
- The topic of fp requires is own, maybe multiple lectures.

Functional Programming – Part III

- We have just repeated the fact, that methods can have side effects and fp can effectively reduce side effects.
 - Hm, why is that a relevant or important topic?
- One relevant aspect is, that fp gives to us better tools to exploit multi core/CPU computing power.
 - But to understand the ideas behind programming multi-processing code with so called multi-threaded programming.
- Another aspect is, that side effects in methods can introduce code, which behaves surprisingly or even wrongly.

Functional Programming – Part IV

- Let's discuss this code, in which *a()* has a side effect on *globalVariable* and *b()* accesses *globalVariable*:

```
// <Program.java>

public class Program {
    static int globalValue;

    static int a() {
        globalValue = 1;
        return 42;
    }

    static double b(int x) {
        return globalValue * x;
    }
}
```

Definition

The paradigm of functional programming demands, that calling a specific function (read: method) must yield the same result, when called with the same arguments no matter when or how often the function is called. This principle is called referential transparency. Referential transparency cannot be guaranteed, when functions have side effects. A function obeying referential transparency is called pure function.

- With this constellation, we can program two version of *main()* using *a()* and *b()* for a calculation:

```
// Version 1:
public static void main(String[] args) throws Exception {
    double result = a() * b(15);
    // result = 630.0
}
```

```
// Version 2:
public static void main(String[] args) throws Exception {
    double result = b(15) * a();
    // result = 0.0
}
```

- The result, i.e. the value of the expression with the same methods and arguments, depends on the order of method calls!
- Mind, that side effects could even be more difficult to predict, when we use multiple operator expressions with mixed precedences...
- For god's sake the order of evaluation of method calls is always from left to right, i.e. *a() * b()* really means *a() * b()*.
 - In C/C++ result's value would be completely unpredictable in both versions, because in C/C++ the order of expression evaluation is undefined.

Documentation of Methods

- To make methods really reusable, it is required to document how these methods work, esp.:
 - What does the method require to work properly: arguments and preconditions.
 - What does the method deliver after the work was done: returned values and postconditions.
- Basically there are two things that help to document something about a method:
 - (1) The most important one: Name the method as well as its parameters in a meaningful/self-explanatory way!
 - (2) Provide a prosaic documentation of what the method does.
- Let's think about the method *factorial()*:
 - The name of the method as well as the name of the parameter (em, just *n*), is already meaningful enough.

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

Good to know:

The method I wrote will be used by other developers, therefore I must document how it has to be used.

- But, do you remember that the passed argument needs to be a positive number? (n needs to be a natural number)
- => We should communicate this precondition to potential users of our method!
- Java provides a convenient means to create prosaic documentations whilst programming: Javadoc!

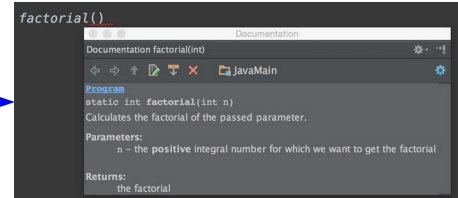
Documentation of Methods – Javadoc

- The JDK provides the `javadoc` tool to generate a documentation from Java source code!
 - By default, javadoc generates HTML 4 pages.
- As programmers we can help Javadoc to generate detailed documentations by providing comments with special markup:

```
/**
 * Calculates the factorial.
 * @param n the <b>positive</b> integral number for which
 * we want to get the factorial
 * @return the factorial
 */
static int factorial(int n) {
    // pass
}
```

Mind

The method, I wrote might also be used by other developers, therefore it must be documented to support the DRY principle!



- Javadoc comments are fringed with `/** */` (not `/* */`) and allow the usage of @-prefixed tags for special documentation aspects.
 - The first lines of a Javadoc comment can be used for a free prosaic text "Calculates the factorial.". The following lines can be used for the tags.
 - Important javadoc (block-)tags for methods: `@param`, `@return`, `@see`, `@since` and `@throws`.
 - It is also possible to embed HTML 4 markup to style the output documentation.
- The result of these comments can also be used in the IDE directly, without generating HTML pages explicitly.
 - Thus we can use the Javadoc comments to have an inline documentation during development in the IDE.

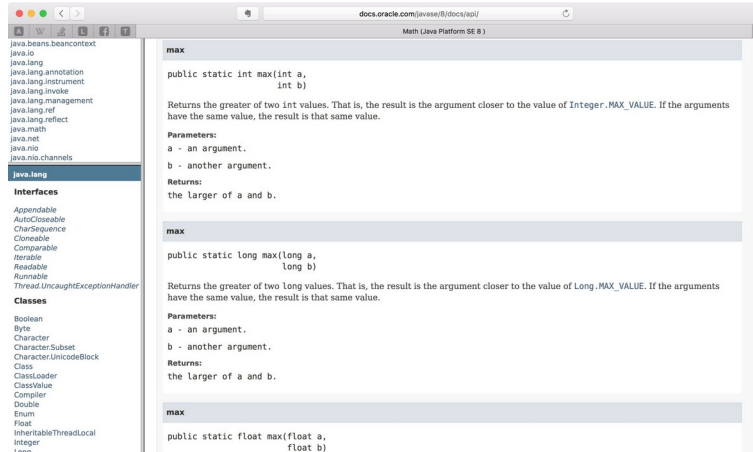
- Writing code and documentation in the same sources is called "literate programming" after Donald E. Knuth.

67

- Besides Javadoc's descriptive block-tags, which require an associated value like `@param` and `@return`, there also exist inline-tags like `{@code}` or `{@link}`, which can be placed freely in a comment text.
- We should avoid adding too many tags and embedded markup to make the generated documentation look awesome. Why? Well, because the comment itself should remain readable as well!
- In the example it is shown, how IntelliJ IDEA shows the documentation of a method from the Javadoc comment. This view can be triggered by `ctrl+Q` on Windows or `ctrl+J` on macOS.
- We can generate an HTML 4 or HTML 5 (since Java 9) documentation from the source code (not from class-files!) ourselves by using the command line tool `javadoc`. It is possible to extend javadoc to use own target formats with so called doclets.
 - By default javadoc only processes comments on `public` and `protected` types and members.
 - To also add package-private and private items, the flag `-private` must be specified on the command line.

The Documentation of the JDK also uses Javadoc

- The JDK's APIs have also been documented with javadoc and Java programmers use it every day.
 - <http://docs.oracle.com/javase/8/docs/api/>
 - Here the javadoc documentation showing some overloads of the method *Math.max()*:



- The quality of the JDK documentation should guide the quality of our own documentation!

Testing Methods

```
/**
 * Calculates the factorial.
 * @param n the <b>positive</b> integral number for which
 * we want to get the factorial
 * @return the factorial
 */
static int factorial(int n) {
    // pass
}
```

- With javadoc comments at hand, a 3rd party developer should have enough knowledge to use the method.

```
int result = Program.factorial(16);
System.out.println("Result: "+result);
```

- No problem at all, but what happens, if the developer explicitly hurts the documentation's hint to pass positive numbers:

```
int value = -11;
result = Program.factorial(value % 2) // Oops!
System.out.println("Result: "+result);
```

Terminal

```
NicosMBP:src nico$ java Program
Exception in thread "main" java.lang.StackOverflowError
  at Program.factorial(Program.java:6)
  at Program.factorial(Program.java:6)
  at Program.factorial(Program.java:6)
  ...
```

- The resulting *StackOverflowError* isn't a surprise: recursion doesn't stop for a negative n, which quickly exceeds the stack.
- But it is only no surprise for us, because we know, that *factorial()* is implemented with recursion!

Testing Methods

- That *Program.factorial()* must somehow fail with a negative argument should be clear and cannot be "corrected" simply.
 - In one of the earlier slides in this lecture, we stated, that we will fix this "bug" soon. – We didn't, instead we documented it!
- We didn't explicitly discuss the issue of how *Program.factorial()* can handle errors, because it is barely a theoretical topic.
 - Normally, developers, e.g. to program *Program.factorial()*, make decisions while they go, without further looking back.
 - Here, we as developers just added a hint to *Program.factorial()*'s documentation, but didn't say, what will happen, when ignored.
- Nowadays, developers use another approach, when programming new methods: test-driven development (TDD).
- To enable test-driven development, we have to add some abilities to our Gradle-project. Several additions are required:
 - Add a dependency to a test-library into the project.
 - Add tests, i.e. code, which uses or code (*Program.factorial()*) into the Gradle project obeying Gradle's directory structure's standard.
 - import the the required test-library into our test code and use the test-library's special methods and annotations to get test reporting.
- Actually, we have to do quite a lot, let's discuss each step ...

Updating build.gradle to support Unit Tests

- We have to modify build.gradle, so that it retrieves the needed test-libraries for us and enable test runs as build task.

- (1) We have to tell Gradle, where to search for libraries in general.
 - The places where libraries are searched are called repository (repo).
 - Many quasi-standard libraries can be found in the public Maven repository at <https://repo.maven.apache.org/maven2/>
 - We can also specify private repositories, which we manage on our own.
- (2) We have to tell Gradle, that our project has a dependency to a specific library, **junit:junit:4.12** in this case.
 - Additionally, we specify a test runtime library, which is used to actually execute our tests.
 - Gradle downloads and caches the required libraries automatically from the repos specified in (1).
 - The format to specify libraries as <groupId>:<artifactID>:<version> allows to exactly define, which version of the library to get from the repo.

```
// <build.gradle>
apply plugin: 'java'
apply plugin: 'application'
apply plugin: 'idea'

mainClassName = 'Program'

repositories { // (1)
    mavenCentral()
}

dependencies { // (2)
    testImplementation 'junit:junit:4.12'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.4.2'
}

test {
    testLogging { // (3)
        events 'passed', 'failed'
    }
}
```

- (3) Additionally, we tell Gradle, that we want to see console output for passed and failed test.

Writing Unit Tests – Part I

- We have already mentioned the term unit test. – What does it mean?
- The idea is to test a certain unit of programmed functionality in isolation as far as possible.
 - Unit tests are written by the programmer, who created the functionality to be tested. – I.e. they are not created by QA-personnel!
 - The test-library JUnit provides special tools to support writing unit tests in a simple manner.
- We only have one unit, namely the class *Program*, which only provides one method *Program.factorial()* to test.
 - A unit test should test normal cases and edge cases of the component in question.
 - So, we are interested in testing *Program.factorial()* for if it behaves in an expected manner when used with different arguments.
- Without further ado, here our first unit test, which checks three simple cases/arguments for *Program.factorial()*:

```
import org.junit.*;

public class MyTests {
    @Test
    public void validFactorialResultsTest () {
        Assert.assertTrue(6 == Program.factorial(3));
        Assert.assertTrue(1 == Program.factorial(1));
        Assert.assertTrue(1 == Program.factorial(0));
    }
}
```

72

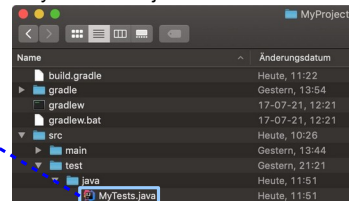
- Now, we going to explain, how this is setup in the project, and of course how the JUnit library works!

Writing Unit Tests – Part II

- By default, Gradle awaits the test code (written in Java) in the directory <projectDirectory>/src/test/java.
 - Mind, that this is a directory, which exists in parallel to <projectDirectory>/src/main/java.

```
import org.junit.*;

public class MyTests {
    @Test
    public void validFactorialResultsTest() {
        Assert.assertTrue(6 == Program.factorial(3));
        Assert.assertTrue(1 == Program.factorial(1));
        Assert.assertTrue(1 == Program.factorial(0));
    }
}
```



- The directory naming should give a hint, that Gradle is able to
 - (1) compile, run and test code written in different languages stored in <projectDirectory>/src/main/<languageName>
 - To make this work, it is of course required to have respective compilers and interpreters installed on the build machine for the languages in question.
 - (2) and to run tests written in any suitable language stored in <projectDirectory>/src/test/<languageName>.
 - E.g. tests written in Groovy with yet more specific libraries like Spock.

Writing Unit Tests – Part III

```
import org.junit.*;

public class MyTests {
    @Test
    public void validFactorialResultsTest() {
        Assert.assertTrue(6 == Program.factorial(3));
        Assert.assertTrue(1 == Program.factorial(1));
        Assert.assertTrue(1 == Program.factorial(0));
    }
}
```

- The test code starts with the statement importing the test-library into our test.
 - After that, all types, annotations and methods provided by *org.junit* can be used in our unit test.
 - With the Gradle directory-structure, the test code in *MyTest* is guaranteed to "know" our *class Program* and *Program.factorial()*.
- In the test method *validFactorialResultsTest()* we have three statements, which check so called assertions.
 - JUnit's method *Assert.assertTrue()* accepts a *boolean* value.
 - If this *boolean* argument evaluates to *true* all is fine: the assertion is correct and *validFactorialResultsTest()* will continue.
 - If any *boolean* argument evaluates to *false*: the assertion is incorrect and *validFactorialResultsTest()* will fail immediately.
- Each test method must be marked with the *public* keyword, to be discoverable by the unit test runner.
- Each method, which should be "discovered" as unit test method must also be annotated with the *@Test* annotation
 - Annotations are an advanced Java construct, which allows to put special information for the compiler and the runtime into the code.

Writing Unit Tests – Part IV

- Having all in place, let's execute our unit test with Gradle just calling the "test" task:

```
Terminal
NicosMBP:MyProject nico$ ./gradlew test
> Task :test
MyTests > validFactorialResultsTest PASSED
BUILD SUCCESSFUL
3 actionable tasks: 3 executed
NicosMBP:MyProject nico$
```

- The colorful output of the test immediately tells us, that "all is green", the tests have succeeded.
- The "test" task is also implicitly executed, when we call the "build" task: after compilation the tests are getting executed.
 - This makes perfect sense: if we had programmed a bug into *Program.factorial()*, we would see it, when we try to build the code!
- JUnit analyses the test code and calls all methods, which are annotated with `@Test`.
 - This is the key to managing multiple test methods in the same test class.
- *validFactorialResultsTest()* only tested the happy case, i.e. expected correct results of *Program.factorial()*.
 - Let's add another test method to check for a negative case, i.e. what we expect the result not to be!

Writing Unit Tests – Part V

```
import org.junit.*;

public class MyTests {
    @Test
    public void validFactorialResultsTest() {
        Assert.assertTrue(6 == Program.factorial(3));
        Assert.assertTrue(1 == Program.factorial(1));
        Assert.assertTrue(1 == Program.factorial(0));
    }
    @Test
    public void invalidFactorialResultTest() {
        Assert.assertTrue(1 != Program.factorial(3));
    }
}
```

- We added a new method *invalidFactorialResultTest()* and annotated it with *@Test*, so that JUnit can discover it.
 - The contained assertion just states, that the factorial of 3 should not be 1.
 - Both tests will now be executed, when running the task "test":

```
Terminal
NicosMBP:MyProject nico$ ./gradlew test
> Task :test
MyTests > invalidFactorialResultsTest PASSED
MyTests > validFactorialResultsTest PASSED
BUILD SUCCESSFUL
3 actionable tasks: 3 executed
NicosMBP:MyProject nico$
```

- But this isn't a useful test, it's already covered in *validFactorialResultTest()*, a more useful test concerns negative arguments!

Writing Unit Tests – Part VI

- Ok, so let's add another test, which checks *Program.factorial()*'s behavior with negative arguments. – But, what to expect?
 - The usual practice to start, i.e. we do not know, what will happen, is just to put the code in question into a test method:

```
import org.junit.*;

public class MyTests {
    // ...
    @Test
    public void invalidFactorialResultTestWithNegativeArgument() {
        Program.factorial(-3);
    }
}
```

- When we run the "test" task, we'll see how the new test method fails with a *StackOverflowError*:

```
Terminal
NicosMBP:MyProject nico$ ./gradlew test
> Task :test FAILED
MyTests > invalidFactorialResultTest PASSED
MyTests > validFactorialResultsTest PASSED
MyTests > invalidFactorialResultTestWithNegativeArgument FAILED
    java.lang.StackOverflowError
3 tests completed, 1 failed
BUILD FAILED in 1s
NicosMBP:MyProject nico$
```

- But, actually, this not an error! – We expect a *StackOverflowError* to happen, if the argument is negative.
- This fact was also clearly given in the javadoc comment of *Program.factorial()*!
- We should change the test! – Getting a *StackOverflowError* is expected: in this case and the test must succeed!

Writing Unit Tests – Part VII

- With JUnit it is super easy to declare a test to expect a certain *Exception*, without further ado, here the solution:

```
import org.junit.*;

public class MyTests {
    // ...
    @Test(expected = StackOverflowError.class)
    public void invalidFactorialResultTestWithNegativeArgument() {
        Program.factorial(-3);
    }
}
```

- As can be seen, `@Test` can be configured with arguments itself! We just tell it, that *StackOverflowError* must be thrown in this test.
- This type of programming, we just annotate, what we want and do not program it explicitly is called declarative programming.
 - So, annotations are a means to allow declarative programming in Java.
- Executing this updated test, yields the result we wanted to see, *StackOverflowError* is ok for negative arguments:

```
Terminal
NicosMBP:MyProject nico$ ./gradlew test
> Task :test
MyTests > invalidFactorialResultTest PASSED
MyTests > validFactorialResultsTest PASSED
MyTests > invalidFactorialResultTestWithNegativeArgument PASSED
BUILD SUCCESSFUL in 1s
NicosMBP:MyProject nico$
```

When Unit Tests break down

- Now that we've a couple of unit tests for *Program.factorial()* in avail, we can make experiments, e.g. make it iterative:

```
public class Program {  
    static int factorial(int n) {  
        int result = 1;  
        for (int i = n; i > 0; --i) {  
            result *= i;  
        }  
        return result;  
    }  
}
```

```
import org.junit.*;  
  
public class MyTests {  
    // ...  
    @Test(expected = StackOverflowError.class)  
    public void invalidFactorialResultTestWithNegativeArgument() {  
        Program.factorial(-3);  
    }  
}
```

- But after this change, our unit tests break!

```
Terminal  
NicosMBP:MyProject nico$ ./gradlew test  
> Task :test FAILED  
MyTests > invalidFactorialResultTest PASSED  
MyTests > validFactorialResultsTest PASSED  
MyTests > invalidFactorialResultTestWithNegativeArgument FAILED  
    java.lang.AssertionError  
NicosMBP:MyProject nico$
```

- The reason is, that *Program.factorial()* will no longer raise *StackOverflowError*, if its argument is negative.
- The iterative implementation will stop the loop, if the operating number is zero or less than zero, which is the case for -3.
- Instead *Program.factorial()* does now actually return a result, namely 1, which is of course mathematically wrong for -3!

79

- This example ends our journey through the topic of unit test for the time being.

- To be frank, we didn't find a proper way to handle *Program.factorial()* with negative arguments. The answer is that there is no good solution with "traditional" programming practices.
- The failed test showed another interesting fact: the test was highly depended on how *Program.factorial()* was implemented, namely with recursion, expecting to deal with *StackOverflowErrors*. And this expectation broke down, when we changed *Program.factorial()*'s implementation to an iterative implementation. The takeaway is, that potential 3rd party users of our code had an implicit dependency to an implementation detail. – This is usually bad, not only could a changed implementation easily break 3rd party users' code, but it also unleashes how we have implemented stuff. – For God's sake, we have a unit test for this aspect! ;)
- In a future lecture, we will at least present a common way for each thinkable implementation of *Program.factorial()* to deal with negative numbers and signal errors to the caller. The approach is to take *Exceptions* in our own hands.

Remarks on Unit Tests

- Unit tests are written by developers, but their aim should be to break the component.
- Advantages having unit tests:
 - They can supplement documentation.
 - They give confidence in refactoring code: we can modify the code, as long as unit tests don't break: all is fine.
- Designing unit tests:
 - The naming and granularity of the unit tests is very crucial.
 - Mind, that the data, i.e. variations in the data and the sheer amount of data is also relevant for unit tests!
 - This data can act as a basis for so called data driven tests (DDTs).
- Test driven development (TDD) further formalizes development and testing, because it demands writing tests first!
- Test enthusiasts count code not being under unit test as legacy code.

Thank you!