Programming languages Java

Kitlei Róbert

Department of Programming Languages and Compilers
Faculty of Informatics, ELTE

Tools used in programming

- programming language
- ***libraries*** (sometimes also called packages): easily reusable program parts
  - ◇ standard library: installed with the programming language
  - ◇ third party libraries
  - ◇ package manager: uses a (central) repository to store libraries, and install them on demand
    - Java: Maven, Ant+Ivy, Gradle, jpm4j
- runtime system
  - ◇ sometimes it's the physical machine
  - ◇ sometimes it's a virtual machine
    - it can be run in a browser or HTTP server, making it an ***applet*** or ***servlet*** respectively
- additional tools

Additional tools

- build system: oversees the compilation, testing, installation of a system
- static checking tool (***lint***): checks the source code for errors in compile time
  - practically an extension of the compiler
- debugger: runtime error detecting tool
- profiler: detects which parts of the code run slow or use too much memory
- project management software
  - can track tasks, milestones etc.
  - version control system (VCS): file versioning
  - bug tracker
  - continuous integration: helps put new code in the active system fast
- integrated development environment (IDE): usually has a lot of the above features
  - supports the modification of code (autocompletion, code snippets etc.)
  - helps see the “big picture” (diagrams, navigation etc.)

Standards

- the rules of programming languages are described in ***standards***
  - ◇ hundreds/thousands of pages of highly technical documents
  - ◇ informally referred to with language+year: ***Ada 2012***, ***C11***, ***C++11***, ***Haskell 98***
  - ◇ sometimes they have version numbers: ***Java 8***
    - ▸ Java's numbering is a bit strange: Java 1.8 is the same as Java 8
- most of the rules are very specific
  - ◇ improves platform independence
    - ▸ sometimes the opposite can also improve platform independence…
      - e.g. it can run on a machine with fewer resources
    - ▸ … with worse compatibility
      - e.g. a counter may overflow on one machine, but not on another
  - ◇ it can decrease efficiency

Standards: what do they cover?

- ***lexical rules***: defines the proper format for the ***token***s in the source
  - ◇ e.g. how words, constants, operators look in the code
- ***syntax rules***: defines the structure that can be built out of tokens
  - ◇ is a given text valid source code?
  - ◇ e.g. what is the fully parenthesized form of a complex expression?
  - ◇ a ***syntax tree*** can be built from valid source code
- ***semantics***
  - ◇ other than being well-formed, does the source code make sense?
    - ▸ e.g. does it contain references to undefined methods?
  - ◇ what does the source code mean: how should it be executed?
  - ◇ are there different valid ways to run the code, room for optimisation?

Compilation, execution

- ahead of time compilation (AOT)
  1. ***compile time***: the ***compiler*** turns the ***source code*** into ***object code***
     - ◇ … or ***compile error***s, if the code is invalid
     - ◇ the object code contains ***machine code*** almost ready to run
     - ◇ object files are linked by the ***linker*** to produce the ***executable***
  2. ***runtime***: the machine runs the executable directly
- ***interpreter***: a separate program that compiles and runs the source code step by step
  - ◇ ***scripting language***s are usually run that way
  - ◇ a popular approach: ***REPL*** (read–eval–print loop)
    1. read: the programmer enters a line in the terminal, which contains an expression
    2. eval(uate): the interpreter runs (evaluates) the expression
    3. print: the resulting value is output

Compilation, execution

- JIT (just-in-time compilation)
  - ◇ the runtime system may retranslate parts of the program during execution
  - ◇ ***Java virtual machine***, ***JVM***: usually Java programs are run on this machine
    - ***bytecode***: the machine code of the Java virtual machine

```
javac X.java
java X
java X param1 param2 param3
```

- the Java compiler compiles one file at a time
  - ◇ at most one of the contained classes may be marked as ***public***
    - if there is one present, its name has to match that of the source file (case sensitive)
  - ◇ ***.java***: extension of the source files
  - ◇ the compiler creates a ***.class*** file for all classes in the source

Paradigm

- ***paradigm***: a set of principles one uses when making a program
- ***imperative*** programming
  - emphasises the ***state*** of the program and how it changes during execution
  - the program executes ***statement***s ***in a given order***
    - ***assignment***s make changes in the state
    - the ***control flow*** begins with the first statement, and it is always clear, which way it goes (which instruction is executed next)
- ***structured*** programming
  - ***imperative*** programming where ***control structure***s are restricted
  - ***Böhm-Jacopini theorem*** (1966): all imperative programs can be expressed as a combination of ***sequence***s, ***conditional***s and ***loop***s
  - ***goto*** (directing control flow to a random location) is forbidden
- ***procedural*** programming: subprograms (procedures) are used, their code is structured

Paradigm

- ***declarative*** programming
  - focuses on the structure of the result value instead of control flow ("what" instead of "how")
  - e.g. ***database languages*** (SQL)
- ***logic programming***
  - ***declarative*** programming, where the programmer uses facts and rules to get the result
  - ***Prolog*** (Colmerauer, 1972)

```
% facts (m = mother-of, f = father-of)
m(ed, eva). f(sue, jim). f(jim, ed). f(eva, tom).

% rules
grandfather(X,Y) :- f(X,A), f(A,Y).
grandfather(X,Y) :- m(X,A), f(A,Y).

% queries
?- grandfather(ed, X).
?- grandfather(X, ed).
```

Paradigm

- ***functional*** programming: ***declarative*** programming, where mathematical functions are composed to make the result
  - ◇ same age as imperative programming
    - model: Turing machine (Turing, 1936) ↔ λ-calculus (Church, 1936)
    - language: FORTRAN (Backus, 1957) ↔ LISP (McCarthy, 1958)

```
-- prime numbers in Haskell
prs = 2:filter (\n -> all (\p -> n `rem` p /= 0)
                          (takeWhile (<n`div`2) prs)) [3,5..]
```

- ***event-driven*** programming
  - ◇ the program performs actions when events (e.g. mouse clicks) occur
  - ◇ the code for the actions can be of any mentioned paradigm
- ***object oriented*** programming
  - ◇ we'll talk about this one in detail later on
- many other paradigms exist

The paradigms of Java

- the most important paradigms of Java
  - ◇ imperative; structured; procedural
  - ◇ event-driven
  - ◇ object-oriented
- besides structured constructs, limited variants of goto are available: `break`, `continue`, `return`
  - ◇ full `goto` isn't, though
- some elements of functional programming are making their way into the language, too
  - ◇ many other imperative languages are borrowing functional ideas

Data in the program

- ***value***: data represented in the running program
- values that are conceptually different are said to be of different ***type***s
  - ◇ ***strongly typed*** language: enforces distinction between different types
  - ◇ ***weakly typed*** language: value v of type T can behave as if it was of type T_2
    - ▸ this is a continuity, programming languages lie somewhere between the two extremes
- ***literal***: the form of primitive values in the source code
  - ◇ e.g. `1`, `-52.2623`, `true`, `"abcd\txyz"`
  - ◇ rules for literals are fixed in the standards
    - ▸ `1.` and `.6` can be valid floating point literals in some languages
    - ▸ some languages allow endline characters inside string literals
  - ◇ the types of the values indicated by the literals are fixed

Data in the program

- data can be of primitive types or complex types
  - primitive types of Java
    - character type: ***`char`***
      - technically, it is an integer (the character code)
    - integer types: `byte`, `short`, ***`int`***, `long`
    - logical type: ***`boolean`***
    - floating point types: `float`, ***`double`***
  - all non-primitive values are objects in Java

Data representation

- it might seem that `int`$\equiv \mathbb{Z}$, `double`$\equiv \mathbb{Q}$, or `double`$\equiv \mathbb{R}$
  - … but finite containers (a few bytes) can only hold finite numbers
    - ***overflow***: `(byte)(127+1) == (byte)(-128)`
  - usual representation choices
    - (signed) integer types: ***two's complement***
    - floating point types: ***IEEE 754 standard***
      - `double` values can represent all values of the `int` type
- rounding errors cause inaccuracies, be careful when doing long calculations
  - whenever it is unacceptable (e.g. in financial applications), ***fixed point*** representations are used
- there are types that are arbitrary precision/size, e.g. `BigInteger`
  - less efficient, but only rarely necessary

Subroutines as values

- in some languages, subroutines are values (“first-class citizens”)
  - can be assigned to a variable, can act as a return value etc.
  - ***closure***: subprogram that can refer to its environment (in the example, F refers to the variable C)

```
f(Par1, ParFun) ->  % function in the Erlang language
  C = 36,
  F = fun(X) -> C*X end, % the fun...end is the first-class c.
  F(ParFun(Par1)). % using the function bound by the variable
```

- in Java, subroutines are not first-class
  - ... but they are *almost*, starting with Java 8
    - ***syntactic sugar***: a more complex construct appears in a simplified form in the source code

```
int f(int par1, Function<Integer, Integer> parFun) {
    int c = 36;
    Function<Integer, Integer> f = x -> c*x;
    return f.apply(parFun.apply(par1));
}
```

Program structure

- the basic structure of Java programs is the following (other languages are more or less similar):
  - package → class → method → statement → expression
  - package → class → field → initializer (expression)

```
package hu.site.pkg;
class SomeCl {  // fully qualified name: hu.site.pkg.SomeCl
    int  field;
    void thisIsAMethod() {
        field = 28 * m2() + 456; // statement
             // --   ----   ---
             // ---------       the underlined
             // --------------- parts are all
     // ----------------------- (sub)expressions
    }
    int m2() { return 123; }
}
```

Program structure

- the program consists of ***package***s
  - dots act as separators in the name of the package
  - packages related to the site usually start with the domain name in reverse (in the example, `site.hu`)
  - the directory structure has to follow the class structure
    - e.g. files related to the package `abc.def.gh.ij` have to go in the directory `abc/def/gh/ij`
  - the ***package*** directive gives the name of the file
    - can only appear at the top of the `.java` file
    - e.g. `package abc.def.gh.ij;`
    - if no package is indicated, the file belongs to the ***default package***, and it has to go to the root directory of the source files

Program structure

- packages contain ***class***es
  - the classes describe the structure of the objects
    - they contain ***data field***s or simply ***field***s
  - they also describe the ***operation***s that the objects support
    - ***subprogram*** or ***(sub)routine***: a part of the code that can take arguments and be run
    - ***method*** or ***member function***: the operation associated with a class
    - ***method invocation*** is usually a call to the appropriate subroutine on the local machine
      - from a different point of view: by calling a method, we are sending a ***message*** to the object; when it is received, the subroutine is run, and then we get the return value in another message
    - methods can also be invoked from a different computer (remote method invocation) using network communication
      - in this case, the messages are explicit

Program structure

- the code of the subroutines consists of ***statement***s
  - ◇ statements are ***execute***d when the program is run
  - ◇ statements can be ***simple*** (e.g. declaration, return, break, continue) or ***compound*** (conditionals, loops)
  - ◇ expressions with a semicolon (;) are statements
- ***expression***s are generally used to compute a value
  - ◇ when execution reaches an expression, it is ***evaluate***d: its value is determined
  - ◇ expressions may have ***subexpression***s which usually have to be evaluated to compute the value of the full expression
- in many languages there is no distinction, there are only expressions

Words

- ***keyword***: a lexical element, a (usually readable) word that has a specific purpose
  - ◇ its meaning is fixed by the standard, it cannot be changed
  - ◇ `class`, `new`, `final`, `static` etc. are keywords in Java
  - ◇ it is generally "stronger" than other lexical elements

```
int class = 3;  // forbidden: "class" is a keyword
```

- ***identifier***: the programmer introduces a name for a construct (package, class, method, field, variable)
  - ◇ when the name is used, it is understood to refer to this construct

Program layout

- ***whitespace***: space, line break, tab (and sometimes also other) characters in the source code
  - ◇ Java is ***free-form***: whitespace is only used for separation purposes, mostly ignored by the compiler
  - ◇ programs could be written on a single line
- ***indentation***: whitespace placed on the beginning of the line
  - ◇ ***nested*** constructs (those that contain other constructs: classes, methods, complex statements) increase indentation
    - ▸ typically, 4 (or 2, 3, 8) spaces are added on each level
  - ◇ helps with the perception of program structure
  - ◇ the indented code parts are sometimes surrounded by opening/closing symbols (e.g. *{*, *}*)
    - ▸ there are several options where to place them (end of last line, on new line with/without indentation)
- it is advisable to separate bigger units (classes, methods) by newlines
- ***indentation-based*** language: indentation can modify the meaning
  - ◇ e.g. Python, Haskell

Coding conventions

- ***coding convention***: using stricter rules than enforced by the compiler
  - goal: better quality source code
- coding conventions are applied to
  - names of classes (`CamelCase`), identifiers and methods (`camelCase`), `final` variables (`ALL_CAPS`)
  - how whitespace should be placed, e.g. indentation
  - ***software metrics***: values measured on the code, indicates its quality
    - lines of code (of files or methods): should not be exceedingly lengthy
    - depth of nesting: how many `if`s, `for`s etc. are inside each other
      - more than 2 or 3 makes the program hard to follow
    - tools monitoring metrics can immediately show if the code needs improvement
    - fixing the above problems: code can be extracted to a new method
      - ***refactoring***: reorganisation of the code (with tool support)

Expressions: operators

- ***arity***: number of operands
  - ***binary***: most operators have two operands
  - ***unary***: `!x`, `+x`, `-x`, `~x`, `++x`, `x++`
  - ***ternary***: `x?a:b`
- ***fixity***: where the operator is placed
  - ***prefix*** (`++x`), ***postfix*** (`x--`), ***infix*** (`x+y`), ***mixfix*** (`b?x:y`)
- ***precedence***: what does expr1 $\oplus$ expr2 $\odot$ expr3 mean?
  - expr1 $\oplus$ (expr2 $\odot$ expr3): $\odot$ has higher precedence
  - (expr1 $\oplus$ expr2) $\odot$ expr3: $\oplus$ has higher precedence
- ***associativity***: what does expr1 $\oplus$ expr2 $\oplus$ expr3 mean?
  - (expr1 $\oplus$ expr2) $\oplus$ expr3: $\oplus$ is ***left associative***
    - most operators are left associative
  - expr1 $\oplus$ (expr2 $\oplus$ expr3): $\oplus$ is ***right associative***
    - the assignment operator is right associative in most languages
    - … in some languages, it is not an operator, it is a statement

Expressions: purity/impurity

- expressions and operations are similar in many ways
  - both have names (for expressions: the operator)
  - both can have arguments (for expressions: the operands)
  - some programming languages barely make a distinction
- their behaviour can be described as...
  - ***pure***: computes a value
    - gets arguments, using them produces a return value
    - like functions in mathematics, they always do the same thing: for the same input, they give the same output
    - they can use temporary values (e.g. the values of the subexpressions), but not store them when the computation is done
  - has ***side effect***: performs an action
    - changes the ***state*** of the program
    - ... or communicates with the environment of the program
- ***procedure/subroutine***: name for an operation with side effects
- ***function***: name for a pure operation
  - commonly, "function" can refer to both pure and impure operations

Expressions: purity/impurity

- pure code is much easier to handle than impure
  - ◇ you should try to keep subprograms pure if possible
  - ◇ makes code much easier to test
- if a code part is impure, make it clearly visible
  - ◇ one expression should not have more than one side effect
  - ◇ procedures should contain few if any pure parts; move those to separate functions
- common sorts of side effects
  - ◇ writing/reading global/static variables
  - ◇ outputting values through a method's arguments
  - ◇ I/O operations (reading/writing standard output, files etc.)
  - ◇ network communication
  - ◇ calling another impure function

Expressions: assignment, ++

- for prefix ++, the expression evaluates to the variable's incremented value
- for postfix ++ (and --), it is the original value

```
                 // i   a   b
int i  = 4;      // 4           // declaration with initialisation
    i += 2;      // 6
    i -= 3;      // 3
int a  = ++i;    // 4   4
int b  = i++;    // 5       4
```

- the assignment operator is impure: it sets the value of the variable
  - ◇ assignment is an expression: it has a value (same as that of its subexpression)
  - ◇ it is right associative, unlike most other operators

```
int i;
int j;
i = j = 3 * f() + g();
i = (j = ((3 * f()) + g()));  // fully parenthesized form
```

Expressions: evaluation order

- Java: subexpressions are evaluated left to right
- example: `i+++ t[i]`
  - ◇ fully parenthesized form: `(i++) + t[i]`
  - ◇ `i` starts as 6 and `t[7]` as 15

1. `i`, evaluates to: 6
2. `i++`, evaluates to: 6 (side effect: `i` new value: 7)
3. `t[i]`, evaluates to: `t[7]`, which evaluates to 15
4. full expression evaluates to: `6 + 15`, that is, 21

- example: `t[i] = i = 0`
  - ◇ fully parenthesized form: `t[i] = (i = 0)`
  - ◇ `i` starts as 1

1. `i` in the expression `t[i]`, evaluates to: 1
2. `0`, evaluates to: 0
3. `i = 0`, evaluates to: 0 (side effect: `i` new value: 0)
4. full expression evaluates to: 0 (side effect: `t[1]` new value: 0)

- ◇ it is hard to follow side effects (it is not `t[0]` that gets assigned to)

Expressions: evaluation order

- evaluation order is very important if the subexpressions have side effects
    - ◇ in some languages, the standard does not fix evaluation order in some cases
        - the same source can be validly compiled to two codes that yield different results
        - therefore, in those cases the meaning of the code is *undefined by definition*

```
i = i++ + 1;
a = i++ + ++i;
b = f() + g();    // supposing f and g are impure
```

- it is bad practice to write such code in Java, too
    - ◇ if the programmer uses other languages as well, it's hard to jump back and forth between rulesets
    - ◇ side effects are hard to perceive and follow
        - reading and writing the same variable: are we using the new value?
        - clashing side effects: in what order are they executed?
    - ◇ solution: split it up into smaller statements

Expressions: laziness/eagerness

- we expect pure expressions to produce output from their input values like mathematical functions do
  - ... but they can throw ***exception***s
  - ... and they can end up in an ***infinite loop***
    - this is usually denoted $\perp$ (bottom) or ∞
- compound expressions are affected through their subexpressions
- ***lazy evaluation***: the expression is evaluated only when/if we require their values
  - ***short-circuit***: `false` $\&\& \_ \Rightarrow$ `false`, even if $\_$ would evaluate to $\perp$ or throw an exception
- ***eager/strict evaluation***: evaluates both subexpressions each time
  - `false` $\& X \Rightarrow X$ (in case of $\perp$ and exception, too)
- the operator & is applicable to numbers as a *binary and*: `13&24` $\Rightarrow$ `8`

Expressions: typing

- ***statically typed***: the type of each variable and expression is determined in compile time
  - ◇ detects many types of incorrect usage statically
  - ◇ ***manifest typing***: the types have to be explicitly stated
  - ◇ ***type inference***: the compiler finds out (infers) the types of variables/expressions/methods
    - most functional languages support it explicitly
      - using them, types almost always can be omitted
    - imperative languages have started taking up this feature
- ***dynamically typed***: the variables/expressions are not typed, only the values that they take
  - ◇ a variable can take values of completely different types
  - ◇ no compile time protection
    - usually there are third party tools with partial support for static typing
  - ◇ may improve development time somewhat
- static/dynamic typing is orthogonal to strong/weak typing!

Statements: for

```
for (int i = 0; i < 10; ++i) {
    System.out.printf("value of i: %d%n", i);
}
```

- the loop counter (i) is local to the loop
- it is possible to directly manipulate the loop counter inside the loop, but that is a *very* bad idea

```
for (int i = 0; i < 10; i++) {
    System.out.printf("value of i: %d%n", i);
}
```

- the compiler produces the same code as for the one above
  - it detects that the value of ++i and i++ is unused

Statements: for and while

- for and while loops can be transformed into one another

```
for (initialisation; condition; stepping)  body
 ↓  ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
initialisation;
while (condition) { body; stepping; }

while (condition)    body
 ↓  ↓   ↓   ↓   ↓   ↓  ↓   ↓   ↓   ↓
for (; condition; )   body
```

- infinite loop

```
while (true) body
for (;;)     body

while (1)    body   // hack in weakly typed languages
```

Statements: for and while

- which one should you use? (to increase readability)
  - `for`: iteration over a fixed interval/data structure
    - we know beforehand, how many steps we will take
    - we go over all elements of a data structure (array, list, tree etc.)
  - `while`: possibly infinite loop with exit condition
    - the maximum number of steps is unknown beforehand
    - e.g. the user inputs data, we don't know when he will stop
  - `do`..`while`: loop with condition after the body
    - harder to read than the other two

Statements: foreach

- iteration through a data structure (technically, an `Iterable` or an array)

```
for (String arg: args) {
    System.out.println(arg);
}
```

- new language element in Java 8: stream
  - ◇ technically this is not a statement

```
Stream<String> argStream = Arrays.stream(args);

argStream.forEach((String arg) -> System.out.println(arg));
argStream.forEach(arg -> System.out.println(arg));
argStream.forEach(System.out::println);
```

- streams have a lot of potential

```
int sumOfWeights = widgets.parallelStream()
                          .filter(w -> w.getColor() == RED)
                          .mapToInt(w -> w.getWeight())
                          .sum();
```

Statements: break and continue

- ***break***: stop a loop, execution continues after it
- ***continue***: the rest of the body is skipped, the loop begins its next iteration
- most of the time, they apply to the innermost loop
- if you label outer loops, they can also apply to them as well
  - there is no general goto, you cannot jump at labels at will

```
outer:
for (.....) {  // <-- execution goes here after (3)
    for (.....) { // <-- execution goes here after (4)
        if (.....)   break outer;    // (1)
        if (.....)   break;          // (2)
        if (.....)   continue outer; // (3)
        if (.....)   continue;       // (4)
        // code here runs only if (1)-(4) didn't fire
    }
    // <-- execution goes here after (2)
}
// <-- execution goes here after (1)
```

Statements: block

- groups statements
- limits the visibility of variables declared inside
- most commonly used as the body of `if`s and loops
  - the body of functions is technically not a block statement (although surrounded by braces)
- another use: you can limit the visibility of variables

```
{
    int sumOfTenNumbers = 0;
    for (int i = 0; i < 10; ++i) {
        sumOfTenNumbers += inputNumberFromUser();
    }
    // sumOfTenNumbers is visible here
}

// sumOfTenNumbers is not usable anymore
```

Statements: if

```
if (f1) if (f2) prg1 else prg2
```

- ***dangling else problem***: more `if`s are present in the source code than `else`s
  - question: which `if` should have the `else` clause?
  - universally, the answer is: it should belong to the closer `if` (see left)
  - if we would like it to belong to the outer `if`, we have to use a block (see right)

```
if (f1)
    if (f2)
        prg1
    else
        prg2
```

```
if (f1) {
    if (f2)
        prg1
} else
    prg2
```

Statements: if, for beginners' errors

- the empty statement is written as a single semicolon

```
// wrong              // real meaning
if (...); {           if (...)     ;
    code
}                     {   code   } // runs no matter what

// wrong              // real meaning
for (...); {          for (...)    ;   // runs the empty statement
    code                               // many times over
}                     {   code   } // runs exactly once
```

- without the block, the code has a different meaning, it's easy to miss
  - ◇ if there is any doubt about what goes where, use explicit blocks, parentheses etc.

```
// wrong              // real meaning
for (...)             for (...)
    statement1                statement1     // this is the body
    statement2          statement2   // runs exactly once
```

Statements: switch

- it is "compulsory" to put a **break** at the end of each case
  - ◇ we almost never want the execution to "fall through" to the next case's code
  - ◇ … except if we want the same code to run for several cases

```
switch (expression) {
case value1: prg1;
             break;
case value2: prg2;
             break;
default:     prgDef;
             break;
case value3:
case value4: prg3;
}
```

- the **default** case is optional, and not necessarily the last case