Java training

Data types, operators and control statements

© Bogdan Solga

Session overview

1. Data types
2. Operators
3. Control statements
4. Q & A session

Data types

Java uses two types of data:

- Primitives → the most basic data types
  - Built-in into the language
  - Represented by their corresponding keyword
- Classes → composed data types (further presented)
  - Created by aggregating primitive types
  - Defined and used based on the programming needs
  - The JDK contains a lot of predefined classes, including a set of primitives wrappers
    - Further presented & discussed

Primitive types

Eight primitive types:

Category	Types	Size (bits)	Min	Max	Default	Precision	Example
Integer	byte	8	-128	127	0	+127 → -128	byte b = 65;
	char	16	0	2^{16}-1	0	All characters	char c = 'A'; char c = 65;
	short	16	-2^{15}	2^{15}-1	0	+32,767 → -32,768	short s = 65;
	int	32	-2^{31}	2^{31}-1	0	+2,147,483,647 → -2,147,483,648	int i = 65;
	long	64	-2^{63}	2^{63}-1	0l	+9,223,372,036,854,775,807 → -9,223,372,036,854,775,808	long l = 65L;
Floating point	float	32	2^{-149}	$(2-2^{-23})\cdot 2^{127}$	0.0f	3.402,823,5 E+38 → 1.4 E-45	float f = 65f;
	double	64	2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	0.0d	1.797,693,134,862,315,7 E+308 → 4.9 E-324	double d = 65.55;
Other	boolean	--	--	--	false	false, true	boolean b = true;
	void	--	--	--	--	--	--

Data types categories

- Integer
  - `byte`
  - `short`
  - `int`
  - `long`
- Floating point
  - `float`
  - `double`
- `boolean`
- `char`

Wrapper classes

- All the primitive types have a corresponding wrapper class
- The class is used to:
  - Wrap the primitive type
  - Contain:
    - Their min and max values
    - Utility methods for handling the values
- Default (non-initialized value) - `null`
- Primitives and wrapper classes list →

Wrapper classes and their primitive types

Primitive data type	Wrapper class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Hands-on

Operators

- Operators - special symbols that:
  - Perform operations on 1+ operands
  - Return a result
- Three main categories:
  - Assignment, arithmetic and unary operators
  - Equality, relational and conditional operators
  - Bitwise and bit-shift operators

Assignment and arithmetic operators

Operator	Description
+	Additive operator (also used for `String` concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator → modulo computation

Hands-on

Unary operators

Operator	Description
+	Unary plus operator; indicates positive value
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a `boolean`

Hands-on

Ternary operator

- Ternary operator - quick assignment operator
- Syntax: value = <condition> **?** <first> **:** <second>
- Can be seen as a simple:

```
variable var;
if (condition) {
    var = valueIfTrue;
} else {
    var = valueIfFalse;
}
```

Hands-on

Equality, relational & conditional operators

- Equality & relational:
  - == equal to
  - != not equal to
  - > greater than
  - >= greater than or equal to
  - < less than
  - <= less than or equal to
- Conditional:
  - || or
  - && and

Hands-on

Bitwise & bit-shift operators - brief summary

- Bitwise operators:
  - ~ - unary bitwise complement operator → inverts a bit pattern
  - & - performs a bitwise AND operation
  - ^ - performs a bitwise exclusive OR operation
  - | - performs a bitwise inclusive OR operation
- Bit-shift operators:
  - << - signed left shift operator → shifts a bit pattern to the left, and the
  - >> - signed right shift operator → shifts a bit pattern to the right
  - >>> - unsigned right shift operator → shifts a zero into the leftmost position

Control flow statements / control structures

- Statements - lines of code that control the next action to be performed
- Control flow statements - deciding the flow of operation of the following code
- They either:
  - *Pass over*
  - *Advance statements* → change the state of the program
- Method:
  - Decision making
  - Looping
  - Branching statements

Control statement - categories & types

1. Decision-making
    1. `if-then, if-then-else, nested if… else`
    2. `switch`
2. Looping
    1. `while`
    2. `do-while`
    3. `for`
3. Branching
    1. `break`
    2. `continue`
    3. `return`

Decision making statements

- Selection statements for choosing different paths of execution, based on the evaluation of a conditional expression / statement
- Statements:
  - `if (cond) { ... }`
  - `if (cond) { ... } else { … }`
  - Nested `if (cond) { ... } else { … }` → multiple if / else statements
  - `switch`

if (condition) { ... }

```
if (condition) {
    /*executes the following statements if the condition is true */
    statement_1;
    statement_2;
    // …
    statement_n;
}
```

- Avoid nesting if statements, as much as possible
- Try to use the reversed condition → continue on the same indentation level

Hands-on

if (condition) { ... }

```
if (condition) {
    /*executes the following statements if the condition is true */
    statement_1;
    // …
    statement_n;
} else {
    /*executes the following statements if the condition is false */
    statement_m;
    // …
    statement_x;
}
```

Hands-on

switch (statement)

- `switch` - executing statements based on multiple possible values
- The selection - determined by the expression between the parenthesis
- Usable types:
  - Primitives: `int, char, byte and short`
  - Classes: enumerations, `String` & other wrapper classes

```
switch (weekDay) {
    case FRIDAY: // plan for the weekend
    case SUNDAY: // plan for the next week
}
```

Hands-on

switch (statement) - default

- `default` keyword - specifies the fallback statement(s) to be executed if the condition is not matched

```
switch (weekDay) {
    case FRIDAY: // plan for the weekend
    case SUNDAY: // plan for the next week
    default:     // enjoy what you are doing
}
```

Hands-on

switch (statement) - break

- `break` keyword - interrupting the sequence of executed statements

```
switch (weekDay) {
    case FRIDAY:
        planForTheWeekend();
        break;
    case SUNDAY:
        planForNextWeek();
        break;
}
```

Hands-on

Looping statements

- *Loop* - repeating a set of statements until a condition is satisfied
- Usefulness - avoiding writing repetitive code
  - Repeating the same piece of code - also called an *iteration*
- Three kinds of looping/iteration statements
  - `while`
  - `do-while`
  - `for`
    - Index based: `for (int i = 0; i < 10; i++) { … }`
    - Item based: `for (String word : words) { … }`

Hands-on

while (condition) { … }

- Usefulness - repeat 1+ statements while the condition is `true`
- Behavior:
  - Evaluates the condition in the parenthesis
  - Executes the statement(s) in the {} block until the condition is `false`

```
while (condition) {
    statement_1;
    ...
    statement_n;
}
```

Hands-on

do { … } while (condition)

- Behavior:
  - 1. Evaluates the statements inside the body
  - 2. Evaluates the while condition (specified in parenthesis)
- If the condition is true - the next iteration is executed
- Else - no iteration takes place

```
do {
    statement_1;
    ...
    statement_n;
} while (condition);
```

Hands-on

for (iteration mode / items)

- Usefulness:
  - Iterate over a set of items - array or collection, usually
  - Perform a set of common operations on them
- Two types of for loops:
  - Index based: uses a variable value to iterate over the items

```
for (int i = 0; i < 10; i++) { … }
```

  - Item based: iterates over each item

```
for (String value : sentence) { … }
```

Index based `for` iteration

The syntax consists of three segments

- Initialization - initializes the loop value
- Termination condition - terminates the loop execution when `false`
- Change of state expression - changes the expression's value
  - Invoked after every loop iteration
  - Usually an increase or decrease of the expression

```
for (int i = 0; int < 10; i ++) {
    System.out.println(i);
}
```

Hands-on

Item based `for` iteration

The syntax consists of two segments:

- Variable type and name
- Iterated object/array

```
for (String word : words) {
    System.out.println(word);
}
```

Hands-on

Branching statements

- Usefulness - unconditionally shift the control to another statement
  - Also known as ‘jumping statements’
- Branching statements:
  - `break`
  - `continue`
  - `return`

The `break` statement

- Usefulness - instructs the control flow to:
  - *Terminate the current block execution*
  - Continue with the first statement after the block
- Used in two forms:
  - Unlabelled → for looping:
    - Statement blocks
    - The switch statement(s)
  - Labeled

The `break` statement - unlabeled & labeled

- Unlabeled: → the most frequent & recommended

```
for (int i = 0; i < 5 ; i++) {
    System.out.println("The index is " + i);
    if (i == 3) break;
}
```

- Labeled: → the least frequent & least recommended

```
Label:
for (int i = 0; i < 5 ; i++) {
    if (i == 3) break Label;
}
```

Hands-on

The `continue` statement

- Usefulness - instructs the control flow statements to:
  - *Skip the execution* of the current line of code
  - Move to the next block of code
- Used in two the same two forms:
  - Unlabelled
  - Labeled

The `continue` stmt. - unlabeled & labeled

- Unlabeled: → the most frequent & recommended

```
for (int i =0; i < 5 ; i++) {
    System.out.println(“The index is “ + i);
    if (i == 3) continue;
}
```

- Labeled: → the least frequent & least recommended

```
Label:
for (int i =0; i < 5 ; i++) {
    if (i == 3) continue Label;
}
```

Hands-on

The `return` statement

- Usefulness - instructs the block of code to:
  - Exit
  - Return to the method it was invoked from
- Two forms:
  - 1. Returns a value → used in methods that return a non-`void` value
  - 2. Doesn't return a value → used in `void` returning methods

Q & A session

- Please ask any remaining questions