

CSEN 202

Introduction to Computer Programming

Lecture 2: Data Types

Prof. Dr. Slim Abdennadher and
Dr Mohammed Abdel Megeed Salem,
slim.abdennadher@guc.edu.eg

German University Cairo, Department of Media Engineering and Technology

February 10/15, 2018

1 Synopsis

1.1 Lecture 1

What you already know

- The *origin* of Java as a programming language
- The path from the *source* to the *executable* in Java
- A *minimum* Java program and its structure
- Further, you know...

What you already know

- ... The structure of *comments*
 - Normal comment: `/* ... */`
 - Single line comment: `// ...`
 - Javadoc comment: `/** ... */`
- ... The classification of *errors*
 - Syntax error
 - Logical error
 - Runtime error
- The format of *identifiers*

- Starts with letter, can include letters, digits, '\$', '_'
- Is case sensitive
- Must not be a *reserved word*

Just for the record

Reserved words in Java:

abstract, assert, boolean, break, byte, case, catch, char, class, const¹, continue, default, double, do, else, enum, extends, false, final, finally, float, for, goto¹, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while

2 Outline

2.1 Lecture 2

Today's lecture

- Primitive datatypes and their aspects
- Special values
- Literals and assignments
- Strings
- Composite expressions

3 Primitive data types

Summary

Obviously, we need to clarify:

- How a piece of data in the memory is to be *interpreted*
- How data can be *stored* in the memory
- What *values* can be assigned to a *memory location* (variable)
- What *operations* are possible on a piece of data

¹not used anymore, but still reserved

3.1 The type

Properties of a value

- Each *piece of data* stored in a computation and information system is necessarily associated with *certain properties*:
 - Representation, precision, potential for manipulation, ...
- We *abstract* these properties by using *mathematical concepts*:
 - *Sets* to describe the range of values, *functions* to describe the possible operations

Datatype

The overall properties of a piece of data in a storage location are recorded as the *datatype* or simply *type*.

- To understand datatypes it makes sense to understand the *underlying representation*

Declaring the type

- To communicate the type of a variable, the variable has to be *declared*
- A declaration specifies
 - the *datatype*,
 - the variable's *name* (an identifier), and
 - optionally an *initial value*.

Example 1. ;3;[declaring an integer variable]

```
      type
      int studentSemesterCount ;
                variable name
```

Example 2. ;4;[declaring and initializing a floating-point variable]

```
      type                                     initial value
      float gradePointAverage = 0.7 ;
                variable name
```

Example 3. ;5-[declaring multiple floats]

```
      type
      float top, average, bottom ;
                multiple variable names
```

Identifiers as variable names

In variable declarations, observe the following *naming conventions*:

- start with a first word in *lower-case*
- for consecutive words, *capitalize* first letter
- Examples: `studentSemesterCount`, `gradePointAverage`, `total`, ...

Primitive types

Integer types

- **byte**: an *8-bit* signed two's complement integer.
(−128 to 127, why?)
- **short**: a *16-bit* signed two's complement integer.
(−32,768 to 32,767)
- **int**: a *32-bit* signed two's complement integer.
(−2,147,483,648 to 2,147,483,647)
- **long**: a *64-bit* signed two's complement integer.
(−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

Primitive types

Floating-point types

- **float**: a single-precision *32-bit* IEEE 754 floating point.
($\pm 3.4 \times 10^{38}$ with 7 significant bits)
This data type should *never* be used for precise values, such as currency! (Why?)
- **double**: a double-precision *64-bit* IEEE 754 floating point. Generally the default choice for decimal values.
($\pm 1.7 \times 10^{308}$ with 15 significant bits)
Never use for precise values, same reason.

Primitive types

Other types

- **boolean**: **true** or **false**. The “size” (representation) isn't something that's *precisely defined*...
- **char**: a single *16-bit Unicode* character.

Example

- **short** `number = -30637;`
Declares a 16-bit signed (two's complement) integer with the name “number” and the initial value −30637
- **char** `jutsu = '\u8853';`
Declares a single character named “jutsu” with the initial value 術
- **double** `average = 54.597;`
Declares a double-precision floating point number (52-bit mantissa, 11-bit exponent) with the initial value 54.597
- **boolean** `flag = true;`
Declares a boolean variable with an initial value of **true**

4 Special values and literals

4.1 Characters and booleans

Character values

Character literal values include:

- Single characters surrounded by quotation marks:

```
char letter = 'L';
```

- Unicode values in hexadecimal:

```
letter = '\u262D';
```

- Special characters

Escape Sequence	Unicode	Character
<code>\b</code>	<code>'\u0008'</code>	Backspace
<code>\n</code>	<code>'\u000a'</code>	Line feed
<code>\t</code>	<code>'\u0009'</code>	Horizontal Tabulation
<code>\'</code>	<code>'\u0027'</code>	Single quote
<code>\"</code>	<code>'\u0022'</code>	Double quote
<code>\\</code>	<code>'\u0055'</code>	Backslash

Boolean values

- The reserved words **true** and **false** are the *only* legal values for variables of type **boolean**!

```
boolean understood = true;
```

- A **boolean** variable stores *one bit* worth of information, however the internal representation is *not* defined.

4.2 Integers

Integer values

Integer values can be given as

- Simple (signed) decimal numerals

```
byte b = -128;
```

- Signed binary, octal, or hexadecimal numbers

```
/*A hexadecimal prefixed with 0x */ int i = -0x1FA29;
```

```
/*An octal prefixed with 0 */ short s = 0177;
```

```
/*A binary prefixed with 0b */ long l = 0b1001010010111101;
```

- An integer numeral is by default of type **int**. Literals of type **long** are suffixed with “L”

```
long l = 23L;
```

Integer values

If an integer literal is *small enough* to fit into a **byte** or a **short**, it will be automatically converted. The same is true for **long** literals and **int**, **byte**, and **short**.

- **byte** b = 0x7F; /*7 bits, OK */
- **short** s = 0x7FFF; /*15 bits, OK */
- **long** i = 0x12345678L; /*29 bits, OK */
- **byte** b2 = 0xFF; /*Error: 255 > 127 */
- **int** b2 = 0xFFFFFFFFFFFFFFFF; /*number too large*/

Integer values

Note:

- If a literal is too big for its target variable, you must *explicitly convert* it using a *type cast*. The number is converted by truncating the extra bits, which is probably not what you want.

```
/*0x100 = 256 */byte b = (byte) 0x100; /*b now equals 0! */
```
- An **int** literal can *always* be assigned to a **long** variable—its value will be the same as if it was assigned to **int** variable.

4.3 Floating points

Floating point values

- The *type* of a floating point value is by default **double**

```
double d = 3.141592654;
```
- To type a literal as **float**, it must be suffixed with “f”

```
float f = 3.141592654f;
```
- floating point values can be given in base-10 scientific notation

```
double d = 1.234e2; /*equals 123.4 */float f = 1.234e-3f; /*equals 0.001234 */
```
- Again: these types are *not* meant for precise arithmetics!

Floating point values

- You can assign a **float** to a **double**, but *not* vice versa!

```
double d = 3.141592654f; /*OK */float f = 3.141592654; /*type mismatch! */
```
- When an integer literal is assigned to a floating-point type, it is automatically “*promoted*” to floating-point, even if that means a *loss of precision*.

```
float f = 2; /*OK, f = 2.0 */float f2 = 1234512345L; /*OK, f2 = 1.23451238E9 */
```

5 Other aspects

5.1 Default values

Initialization

Always initialize your variables!

Default values for uninitialized variables

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

Note that *not all* variables are automatically initialized!

Constants

You may want to use constants to structure your code.

- A variable that is declared as **final** *cannot* be changed during runtime

```
final double PI = 3.141592654;
```

- By convention, names of constants are *all uppercase* using *underscore* to separate words.

```
final boolean ALL_UNDER_CONTROL = true;
```

6 Strings

6.1 Strings: a non-primitive datatype

Strings

- `String` is *not* a primitive data type: It is an *Object*.
- Predefined class `String` has *special support* in Java.
- A string literal is surrounded by double quotes.

```
String hamlet = "to_be_or_not_to_be";  
(ignore the "_" for now)
```

- Once a string has been created, we can use the *dot operator* to invoke its methods:

```
l = hamlet.length ();
```

Strings

- The `String` class has several *methods* to manipulate strings
 - `char charAt (int index)`: returns the character at the specified index
 - `String toLowerCase ()`: Converts all of the characters in this `String` to lower case.
 - `String replace(char oldChar, char newChar)`: Returns a new string resulting from replacing all occurrences of `oldChar` in this string by `newChar`.

7 Composite expressions

7.1 Using primitive data

Arithmetic operators

- Expressions may be composed through *operators*.
- Java provides *five* basic arithmetic operators:
 - `+` — Addition
 - `-` — Subtraction
 - `*` — Multiplication
 - `/` — Division
 - `%` — Modulus (remainder)
- There are also *unary* `+` and `-` operators (*i. e.*, with just one operand)
- The operators can be applied to *any* of the integer or floating-point types.

Precedence

Expressions in Java observe a standard precedence on operators

- Unary `+` and `-` have the *highest* precedence
- Multiplication, division, and modulus come *next*
- Addition and subtraction come *next*
- Assignments have the *lowest* precedence
- Operators with equal precedence are evaluated *left-to-right*
- Parentheses `(...)` *override* precedence

Comparisons

- Comparisons are applied to two expressions of *compatible type* and always yield a **boolean** result.

```
a < b a <= b a == b /*equals */ a > b a >= b a != b /*not equal to */
```

- Assignment operator `a = b;`

Do not confuse `a = b` with `a == b`!

Examples

```
double a;  
int i;  
boolean b;
```

```
a = 3.1415 + 42;    // a = 45.1415  
i = 4 - 9;         // i = -5  
i = i + 1;        // i = -4  
a = i * 2 + 3;    // a = -5  
a = i * (2+3);    // a = -20  
b = i > 0;        // b = false
```

Logical expressions

- Logical operators enable the composition of single Boolean values
- They are known from *last semester*:
 - Logical AND (A AND B) yields true only if both A and B evaluate to true. In Java: `A && B` or `A & B`.
 - Logical OR (A OR B) yields true if either A or B, or both yield true. In Java: `A || B` or `A | B`
 - Logical XOR (A XOR B) yields true if and only if exactly one of its operands is true. In Java `A ^ B`
 - Logical Negation inverts its operand. In Java `!A`
- `A && B` and `A || B`: evaluate the second operand *only if required*.
- `A & B` and `A | B`: *Both* operands have to be evaluated.