Programming Java

STEP IT ACADEMY

Lesson 2

Variables, data types

Contents

1. Basics

1.1. The Concept of Strong Typing

When declaring a variable in Java, it is always necessary to specify its type. After declaring a variable, it cannot change its type throughout the scope of a variable in the code. Strong typing allows detecting errors in the code during compilation of the program.

1.2. Data Types

Java language has two main categories that divide variable types into primitive and reference.

Primitive data types are present in the syntax of the language. In order to use primitive types, there is no need to create custom classes or use any libraries.

Referenced type uses a reference to an object as a value.

Primitive data types are divided into:

- ***Integer*** primitive data types can store only integers (without a fractional part) in the range specified by the dimension of a type.

A table of integer types

Name	Capacity	Range of values
`byte`	8 bits (1 byte)	−128 to 127
`short`	16 bits (2 bytes)	−32,768 to 32,767
`int`	32 bits (4 bytes)	−2147483648 to 2147483647 or −231 to 231
`long`	64 bits (8 bytes)	−263 to 263

- ***Floating-point*** data types are used for storing fractional values. Floating-point data type can be represented as a decimal fraction. Floating point is used as a fractional part separator. For example, 1.5 is one and five decimal fractions. Scientific and exponential notation may also be used.

A table of floating-point types

Name	Capacity	Range of values
`float`	32 bits (4 bytes)	from −3. 4e − 038.. to 3. 4e + 038
`double`	64 bits (8 bytes)	from −1. 7e − 308.. to 1. 7e + 308

The features of special values with a floating point

Expressions	Outcome
Math.sqrt(-1.0)	NaN
0.0 / 0.0	NaN
1.0 / 0.0	Infinity
-1.0 / 0.0	−Infinity
NaN==NaN	false
Infinity==Infinity	false

Character type can only store integer numbers, which are interpreted as character codes from the table based on the Unicode. Unicode is a character encoding standard, in which each character has a specific code that corresponds to it. ***Unicode*** contains all the possible characters of various peoples of the world. Each character is encoded by two bytes. If it is impossible to type a character from the keyboard in the code, it can be replaced by a character record in the form of Unicode. For example: '\u0056'.

Name	Capacity	Range of values
`char`	16 bits (2 bytes)	0 to 65536

Boolean data type is used for storing the values obtained by calculating logical expressions, or it can be specified by logical literals **true** and **false**. In the standard implementation of the "Sun JVM", 32 bits are used for storing the boolean values, and in case of boolean array, it is optimized to 8 bits. It is recommended to use the BitSet class for storing a large number of boolean type values.

Name	Capacity	Range of values
`boolean`	-	**true** or **false**

Casting and Type Conversion

Java is a strongly (statically) typed language, and it always controls the operation with a certain data type, but in spite of this, Java gives the programmer the ability to copy values of one type to another.

If the dimension of the copied value type is less than the dimension of the type, to which the value is copied, then *implicit conversion* automatically occurs. Implicit conversion requires no special syntax.

For example:

```
short i = 3;
int j = i;
long n = 5; // number 5 is an int type literal
```

Explicit conversion (type casting) may also take place in expressions. All the operands are implicitly casted to the type with the greatest capacity.

For example:

```
byte a = 37;
short b = 12;
char c = 'a';
int sum = a + b + c; // all operands are implicitly
                     // casted to int
```

Table of implicit conversion

Source type	Implicitly converted types	Precision loss
byte	short, int, long	No
short	int, long	No
int	long	No
int	float, double	Yes
char	int	No
long	float, double	Yes
float	double	No

Example of implicit conversion with precision loss:

```
int big = 1234567890;
float f = bigNumber;
System.out.printf("%f", f);
```

Outcome: 1234567936.000000

If there is a need to copy a value of a type with larger capacity to a type with lower capacity, then it is necessary to use *explicit conversion*.

The syntax of explicit conversion:

```
<type> i = (<type>) source type value;
```

For example:

```
int i = (int) 7L;
float m = (float) 1.5; //1.5 is a double type literal

int j = 12;
byte b = (byte) j;
```

Explicit conversion error:

```
byte b = 55;
b = b * 2; // compilation error; 2 is the int type
           // literal and the result of expression
           // will also be int
```

Overflow or underflow may take place in implicit conversion or explicit conversion. Java does not control these processes and does not display any errors at compile time or runtime.

Example of overflow in implicit conversion:

```
byte bMax = 127;
bMax++;
System.out.println(bMax);
```

Outcome: -128

Example of overflow in explicit conversion:

```
short maxValue = 256;
byte bb = (byte) maxValue;
System.out.println(bb);
```

Outcome: 0

Example of underflow:

```
double d1 = 0.3333333333333333;
// the loss of digits following the point
// starting from the eighth digit
float f1 = (float) d1;
System.out.println(f1);
```

Outcome: 0.33333334

Example of underflow:

```
float f3 = 3.64f;
int i3 = (int) f3; // fractional part is dropped
System.out.println(i3);
```

Outcome: 3

Note!!! *In explicit conversion, overflow or underflow should be controlled by the programmer.*

Wrapper Classes

For all primitive types in Java, there are wrapper classes used for operating with values of primitive types as objects (reference types).

Since Java is an object-oriented language, all the data following the paradigm of OOP should be encapsulated in objects. But the use of objects for mathematical calculations significantly reduces performance, complicates the code, and occupies a large amount of memory. Therefore, in addition to reference types, Java also has primitive types, which are exceptions from the object model of the language. Wrapper classes return the work with primitive types to the object model.

Example of addition with the use of objects:

```
Integer sum = new Integer(2) + new Integer(2);
```

Example of addition with primitive types:

```
int sum = 2 + 2;
```

All wrapper classes are immutable (the value stored within the wrapper class object cannot be modified), it is also impossible to inherit from wrapper classes. These rules are related to a logical purpose of wrapper classes, which just wrap numeric values.

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

All integer wrapper types are inherited from the Number abstract class.

The values are passed to the wrapper class using constructor as a literal, a primitive type variable, or a string.

For example:

```
Integer i = new Integer(3);
Integer i1 = new Integer("123");
Number n = new Short(4);
```

Wrapper classes have static and non-static methods.

Static methods. The methods such as parseXXX cast (convert explicitly) a string to a primitive type value.

Examples:

```
int i1 = Integer.parseInt("123");
boolean b1 = Boolean.parseBoolean("TRUE");
```

The valueOf method casts a string or number to a wrapper class.

Examples:

```
Integer i2 = Integer.valueOf("5");
Integer i3 = Integer.valueOf(7);
Float f = Float.valueOf(1.3f);
```

The toBinaryString method is in wrappers classes of integer types only. This method casts a number to a string in binary form.

For example:

```
String binaryString = Integer.toBinaryString(21);
System.out.println(binaryString);
```

Outcome: 10101

The toHexString method casts a number to a string in a hexadecimal form.

For example:

```
String hexString = Float.toHexString(1.5f);
System.out.println(hexString);
```

Outcome: 0x1.8p0

For example:

```
String hexString = Integer.toHexString(255);
System.out.println(hexString);
```

Outcome: ff

The Character wrapper class has a lot of different methods unique to it. Most of them are simple and their names make it clear what they do.

For example:

```
// checks whether a character is a digit

boolean isDigit1 = Character.isDigit('3');
boolean isUpper = Character.isUpperCase('b');
// true if the digit is in uppercase
```

Tip: *Always try to use primitive types, while the wrapper classes should only be used where it is impossible to use primitive types.*

Autoboxing and Unboxing

Autoboxing is a mechanism of implicit creation (without the use of the **new** operator) of the wrapper class object of the corresponding primitive type.

For example:

```
Integer i = 5;
Boolean b = true;
```

Unboxing is a mechanism of implicit conversion of the wrapper class object to a corresponding primitive type.

For example:

```
int i = new Integer(5);
```

Autoboxing and unboxing may take place when assigning or passing to a method.

Example of autoboxing:

```
public static void main(String[] args)
{
    test(3); // literal 3 is wrapped
             // in the wrapper class object
}

public static void test(Integer value)
{
    System.out.println(value);
}
```

Autoboxing and unboxing simplify the code, but they can cause a number of errors that can be detected only at runtime.

For example:

```
Integer i = null;
int j = i; // NullPointerException will occur
```

For example:

```
Number n = 555;
byte b = (Byte) n; // ClassCastException will occur
```

For example:

```
static void inc(List<Integer> list, int v) {
  for (Integer i:list) {
  // there is no change in variable value in the list
    i +=v;
  }
}
```

Note: *Autoboxing and unboxing appeared in java since version JDK 1.5.*

1.3. Comments

Java language allows putting various comments, observations, and explanations in the source code, which are not subjected to compilation and are not a part of bytecode.

- `//` — ***inline comment***; the compiler ignores everything from double backslash to the end of the line.

For example:

```
int i = 5; // int i = 3;
```

Multi-line comment allows excluding a part of the source code from the compilation.

- `/*` — start of a multi-line comment,
- `*/` — end of a multi-line comment

For example:

```
/*
    Example of a multi-line comment
    int i =5;
*/
```

```
i = 3; // compilation error, since the i variable
       // is not declared in the code, but is located
       // within the comment block.
```

Javadoc is a special kind of comments used for describing the purpose of classes, methods, and class fields. JDK includes a utility that gathers all the javadoc comments of all the files of the project source code in a document (html), and creates an API (application programming interface) HTML document on their basis.

For example:

```
/**
 * Description of class purpose
 * @see Config (see the Config class with this)
 * @author VUnguryan (class author)
 * @ since 1.2.1 (class version)
 */
public class Controller {
}
```

Example of API for Java classes.

1.4. Variables

Variable is a named area of memory, to which a value of particular type can be written or overwritten, and from which this value can be read.

Variable type and its name (identifier) are specified when declaring variable in a program.

Syntax of variable declaration:

```
type identifier;
```

For example:

```
int a;
byte b1;
boolean $_$;
```

The language syntax allows declaring several variables of the same type in a single line at a time, but this is not recommended according to the Java Code Conventions.

For example:

```
float x, y, z;
```

Variables declared within a method are called *local variables*, and they require explicit initialization before they can actually be used.

Variable scope in the code is limited to the location of its declaration and block separator (curly braces). Variable declared within the class body is called class field, and if there is no initialization, instance variable takes the default value (depending on the type).

For example:

```
class A
{
static int x; // class field
    public static void main(String[] args) {
       int y; // local variable
       System.out.println(y);   // compilation error
                                // with a message
       System.out.println(x);   // outcome is 0
    }
}
```

Variable name:

- can contain all the characters of the Latin alphabet, digits, $ character, and underscore;
- cannot begin with a digit;
- cannot match Java keywords.

Examples of correct naming of variables:

```
long startTime;
int x;
boolean flag;
```

Examples of incorrect naming of variables:

```
int 1side;
char goto;
long st@rt;
```

The use of $ character and underscore in the names of variables is considered a bad practice.

Variables should conventionally be name according to the recommendations of Java Code Conventions.

1.5. Constants

Constant is a variable that should be initialized where it was declared (or in the class constructor), and cannot change its value in the scope of this variable in the code. The **final** keyword is used for specifying a constant.

For example:

```
final int const = 5;
const = 3; // compilation error
final int const1; // compilation error
```

The concept of literal fits the concept of constant.

Literal is a value of particular type explicitly specified in the code. A special syntax is used for specifying literals in the program code (see the table).

Literal type	Data type	Special characters	Example
Integer	int	x	3, 03, 0x3
	long	l, L	3l, 3L
Floating-point	float	f, F, e	1.5f, 1.5F, 1.5e-1f
	double	d, D, e	1.5, 1.5d, 1.5D, 1.5e-1
Boolean	boolean	true, false	true, false
Character	char	'	'a', '\u0041'
Reference	All *reference*	null	null

Note: *Starting with JDK 7.0, Java allows separating digits of numeric literals with an underscore. For example: 100_000_000.*
It is not recommended to use lower case letter l for specifying a long type literal since it is possible to misinterpret it (it looks like number 1).

1.6. Input-Output in a Console Program

Information output in the console:

The System class and its *out* static variable should be used for data output in the console.

Examples:

```
System.out.println("text"); // text output with
                            // transition
```

```
// to the next line
System.out.println(5); // literal output with
                       // transition to the next line

int i = 6;
// variable value output
System.out.println(i);
// text output without transition to the next line
System.out.print("text");
// formatted variable output
System.out.printf("x = %f", 0.5f);
```

The java.util.Scanner class should be used for data input from the console. The Scanner class has a number of methods that allow working with the input stream.

The Scanner class is a stream of data received from the source.

When the class is used from a library or from another package, it is necessary to specify where to download it from. For this purpose, it is necessary to add

```
import java.util.Scanner;
```

at the beginning of the file before the class body.

For example:

```
public static void main(String[] args)
{
    // creating the Scanner class object and passing
    // the standard output stream to a constructor
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter a number");
    // returns true if the entered number is integer
    boolean isInt = scanner.hasNextInt();
```

```
    System.out.println(isInt);
    // check whether the entered number is integer
    if (isInt)
    {
        // copying the integer value from the console
        // to the x variable
        int x = scanner.nextInt();
        System.out.println("You have entered " + x);
    }
    else
    {
        System.out.println("This is not an integer");
    }
}
```

System.in is a standard stream keyboard input.

Lesson 2
Variables, data types