Types of data

In Java, as in other languages, we distinguish many (built-in) data types that can store numbers, characters, strings (texts), logical type, and many sophisticated, complex types that can perform operations in addition to storing values.

Data types can be divided into two types:

- simple (primitive)
- complex (object-oriented/reference)

> **Tip**
>
> The names of primitive types are written in lowercase, while object-oriented types are capitalized.

Variables

Variables can be compared to "containers" or "boxes" for data. The data that can be stored in a specific variable are defined by their type. Each variable has:

- a type
- a name
- a value

Pattern of a variable declaration

```
type variable-name;
```

While declaring the variable, you **must** specify its name. The value can be assigned later.

Pattern of declaration and initialization of a variable

```
type variable-name = value;
```

Before a variable can be used, a value must be assigned to it. The code below will result in an error!

Example of using a variable without initialization

```
int size;
System.out.println("Size = " + size);
```

> **Note**
>
> The entry `"Size = " + size` means concatenation of two values. Concatenation is performed by the + operator which in this case combines the Size = string with the value of the `size` variable. More information can be found in chapter Arithmetic and concatenation operators.

> **Tip**
>
> Names of variables should be descriptive and self-documenting.

The division of data types determines the types of variables:

```
- variables of simple types
they store values only

- variables of complex types
they store only references (an address of a memory
cell) to the data that they “point to”
```

To better understand the difference between the variables of simple and complex (object-oriented) types, I will use some examples.

Coins and banknotes can be compared to simple type variables, because the value indicated by the denomination is stored in them. However, a bank card or a credit card is in this case responsible for a variable of the complex type (references); the card only *indicates* the account where the money (value) is physically located.

A remote control that remotely executes operations on a receiver, e.g. a TV, an amplifier, etc., is another example of analogy to the reference (object-oriented) types. Unlike variables of simple types (that store the value), variables of complex types only store the reference to the object, therefore it is possible to have more than one variable that has the same reference to the object; it is like having more than one remote control for a TV or a stereo set.

> Important
>
> Java features strict type control. Each variable must be of a specific type that has been once declared—it cannot be changed anymore!

Primitive types

There are eight basic types in Java. These are types that have a certain size in memory; they can store numeric values (integers and floating point values), characters and logical values. All numerical types are **signed** , i.e., they have a sign.

> Tip
>
> Variables (values) of basic types are compared using sign ==.

> Important
>
> In a simple type variable, only the value is stored!

Integer types

Integer types are numbers with no fractional part. They can represent both positive and negative values.

```
- byte: from `- 128` to `127` (`2^8^` numbers)

- short: from `- 32 768` to `32 767` (`2^16^` numbers)

- int: from `- 2 147 483 648` to `2 147 483 647`
(`2^32^` numbers)

- long: from `- 9 223 372 036 854 775 808` to `9 223
372 036 854 775 807` (`2^64^` numbers); suffix `l` or
```

```
`L` is applied (`L` is recommended)
```

The most frequently used types are `byte`, `int` and `long`.

```
- `byte`
in programming, we often use bits, and *1 byte* is 8
bits; `byte` is often used when working with files,
networks and large arrays

- ` int`
is the default integer data type in Java. When you
enter a value, for instance ` 5`, Java treats this
value as ` int` by default.
```

Operations on numbers (integers) are also performed on `int` types by default. This range is sufficient for most operations

```
- ` long`
sometimes during operations on the ` int` type we can
exceed the range for this type and we need a larger
one, e.g. when adding very large numbers of the
```

`int` type we get a number out of range—then it is worth switching to the `long` type. The long values are marked with letter `l` or `L`, e.g. `5L`, `120000000L`, `-556677889900L`

Examples of variables of integer types

```
byte b = (byte) 56;
short s = (short) 89;
int i = 256;
long l = 5500L;
```

- `(byte)` indicates *casting* on the `byte` type
- `(short)` indicates *casting* on the `short` type

In this case, casting is not necessary (Java can perform it automatically), but this mechanism is worth knowing (more information about it can be found in chapter CONVERSION OF TYPES)

By default, integer operations in Java are performed using the int type. Please look at the code below:

```
long a = 24 * 60 * 60 * 1000 * 1000;
```

```
long b = 24 * 60 * 60 * 1000;
System.out.println("a / b = " + a / b);
```

According to what we remember from our math lessons, we can reduce the `a / b` quotient by pulling value `24 * 60 * 60 * 1000` in front of the bracket and expect a result equal to **1000**. Unfortunately, in the current form, the value displayed on the screen will be as below:

```
5
```

What is more, if both the `a` and `b` values are multiplied by `1000` first, the result will be even more surprising:

```
long a = 24 * 60 * 60 * 1000 * 1000 * 1000;
long b = 24 * 60 * 60 * 1000 * 1000;
System.out.println("a / b = " + a / b);
```

Result:

```
-3
```

Why is this happening? Why when we multiply and divide positive numbers, we get a negative number as a result?

As I have mentioned earlier, Java performs calculations on the `int` type by default. When multiplying `int` type values, the range is exceeded and the result becomes incorrect (it is still the `int` type). Then it is assigned to the `long` type.

To avoid such a situation, it is enough to add letter `L` to one of the values, which will allow Java to perform the operation on the `long` type.

```
long a = 24 * 60 * 60L * 1000 * 1000;
long b = 24 * 60 * 60 * 1000;
System.out.println("a / b = " + a / b);
```

Floating point types

Floating point types are types with a fractional part, which can be both positive and negative values.

```
- `float`
```

```
a single precision format on 32 bits; suffix `f` or `F`
is applied (`F` is recommended)

- `double`
a double precision format on 64 bits; **the default
type for floating point values**; to increase
readability, you can use suffix `d` or `D` (`D` is
recommended)
```

Examples of variable of floating point types

```
float f = 3.141592F;
double d = 2.718281828D;
```

Types `float` and `double` should not be used to store financial values, because in some cases the string of these numbers leads to loss of information (precision). This is due to the representation of floating point numbers in a binary system in which there is no accurate representation of fraction `1/10`, just like in the decimal system there is no exact representation of fraction `1/3`.

Example of a problem with precision

```
System.out.println(2.00 - 1.10);
System.out.println(2.00 - 1.50);
System.out.println(2.00 - 1.80);
```

The result of the operation will be as below:

```
0.8999999999999999
0.5
0.19999999999999996
```

> **Tip**
>
> For storing financial values, use the `BigDecimal` class.

Character types

```
- `char`
is used to store one character. These characters are
entered between apostrophes, e.g. `'a'`, `'T'`, `'+'`,
```

```
They are
```

character codes (non-negative integers from `0` to `65 556`). Each code indicates a character in *Unicode standard*. We can express them in hexadecimal notation (in the position system based on number 16). Their values range from `'u0000'` to `'uFFFF'`. `u` is a substitute symbol for a character in the *Unicode*. `char` is the type **without a sign** (*unsigned*) – it stores non-negative values.

Examples of character type variables

```
char a = 'a';
char bigA = 'A';
char newLine = '
';
char plus = '+';
```

- `'A'` to `'u0041'`
- `'B'` to `'u0042'`
- `'C'` to `'u0043'`
- `'a'` to `'u0061'`
- `'b'` to `'u0062'`
- `'c'` to `'u0063'`
- `'0'` to `'u0030'`
- `'1'` to `'u0031'`
- `'9'` to `'u0039'`

In addition to the u symbol, there are several other important and useful substitute symbols.

Substitute symbols for special characters	substitute symbol name
b	Backspace
t	Horizontal tab
`	
`	New line
r	Carriage return
"	Double quote
'	Single quote
\	Backslash

Example of substitution symbols applied

```
System.out.print("Hello, World!
");
System.out.print("tHello, World!
");
System.out.print("tt"Hello, World!"
");
```

The result will be as below:

```
Hello, World!
        Hello, World!
                "Hello, World!"
```

Logical type

- `boolean` a logical type that can store one of two values: `false` or `true`. It is used to verify logic conditions

Examples of variables of a logical type

```
boolean isAdult = true;
boolean active = false;
```

> **Note**
>
> The table below is for reference only, you do not have to remember all ranges or values exactly.

type	size	description	default value
`byte`	1 byte	-128 ... 127	`(byte)` 0
`short`	2 bytes	−32 768 ... 32 767	`(short)` 0
`int`	4 bytes	−2 147 483 648 ... 2 147 483 647	0

`long`	8 bytes	−9 223 372 036 854 775 808 … 9 223 372 036 854 775 807	0L
`float`	4 bytes	-3,40282347E+38 … </subscript> 3,40282347E+38	0.0F
`double`	8 bytes	-1,79769313486231570E+308 … </subscript> 1,79769313486231570E+308	0.0
`char`	2 bytes	0 … 65 556 (`'u0000'` … `'uFFFF'`)	`'u0000'`
`boolean`	1 bit*	`false`, `true`	

Primitive types—summary

*) `boolean` type stores *1 bit* of information, but according to the documentation its size is not precisely specified

Object-oriented types

In addition to the above-mentioned simple types (which only store the value), Java provides thousands of object-oriented types that can consist of both primitive and object-oriented types. In addition to storing values, they also provide operations (methods).

In addition to the types found in the Java distribution, there are hundreds of thousands types created and shared (in the form of libraries) by Java programmers around the world.

What is more, each of us—by programming and defining new classes—creates new types of data every day.

> **Tip**
>
> Variables of object-oriented types are compared using `the equals()` method.

> **Note**
>
> Instances of classes (objects) of object-oriented types are created using a key word: `new`.

> **Important**
>
> The default value for object-oriented variables is

> The default value for object-oriented variables is null.

String

Perhaps the most popular object-oriented type in Java is `String` – we were using it in the program displaying text `Hello, World!`. A `String` is called a *chain type* since it consists of a chain (string) of characters, that is, individual characters (chars) surrounded by quotes `" "`. For instance, `“Alice has a cat.”`, `"Hello, World!"`, `"New York 10019"` or `""` (indicating an empty string of characters). Character strings are often called *inscriptions* or *literals*.

Examples of variables of the `String` type

```
String helloWorld = "Hello, World!";
String title = "Thinking in Java";
```

The String class is unique in many respects. Primitive types have only values that we assign to variables using the **=** sign. To create an object-oriented type (create an object of a given class; more information on this topic can be found in chapter “Object-oriented programming”), you should use the keyword new before invoking the constructor.

Examples of creating `String` type objects

```
String helloWorld = new String("Hello, World!");
String title = new String("Thinking in Java");
```

- entry `new String("text")` indicates invoking the constructor of the `String` class and transferring the `"text"` argument to it; it creates a new object of the `String` class

The `String type` is so popular that we do not need to use the keyword `new` and the constructor in order to create a new instance; you simply need to assign a string of characters to the variable.

Examples of creating objects of the `String type` – short version

```
String helloWorld = "Hello, World!";
String title = "Thinking in Java";
```

> **Important**
>
> There is a subtle difference between these two

> entries!

In addition to storing entries, you can perform operations on the `String type`, that is, you can invoke methods.

An example of invoking the `length()` method that returns the length of a string

```
System.out.println("Alice has a cat, and a cat has
Alice!".length());
```

```
– to invoke the method, we use a dot character (`.`)
followed by the method name, in this case `length()`;
in round
```

brackets we pass arguments to the method – in this case we invoked the argumentless (non-parametric) method.

The following text will be displayed on the screen:

```
26
```

An example of invoking `substring()` method that returns the string length

```
String text = "Alice has a cat, and a cat has Alice!";
System.out.println(text.substring(13, 25));
```

```
– the `substring(int beginIndex, int endIndex)` method
returns the substring (fragment of the text) according
to the
```

passed argument values. `beginIndex` indicates the beginning of the fragment (including this character), while `endIndex` indicates the end of the fragment (without this character); the values are provided starting **from zero**

```
Alice has a cat, and a cat has Alice!
012345678911111111111222222
          0123456789012345
             ^           ^
             |           |
        (inclusive) (exclusive)
```

The screen will display:

```
and a cat has Alice
```

If we try to invoke an operation, i.e., refer after the dot (.) on an uninitialized variable, a runtime error will occur (an exception `NullPointerException` will be returned).

name of the method	operation
`charAt(int index)`	returns the character located at the `index` position
`endsWith(String suffix)`	checks whether the text ends with a `suffix` string
`equalsIgnoreCase(String otherString)`	compares the text and ignores the case of the letters
`indexOf(String str)`	returns the position of the beginning of the `str` string in the text
`isEmpty()`	checks whether the text is empty (`""`)
`lastIndexOf(String str)`	returns the last position of the beginning of the `str` string in the text
`replace(char oldChar, char newChar)`	returns the text, replacing `oldChar` with `newChar`
`startsWith(String prefix)`	checks whether the text begins with the `prefix` string
`toLowerCase()`	returns the text, changing uppercase letters to lowercase ones
`toUpperCase()`	returns the text, replacing lowercase letters with uppercase ones
`trim()`	returns text, removing white characters from the beginning and end

Some useful methods of the `String class`

The `String class` is immutable (unmodifiable)! Please note that the methods of this class do not modify the original value of the string and only return the modified copy.

```
String hello = "Hello, World!"
hello.toUpperCase();
System.out.println(hello);
```

- The `toUpperCase` was invoked, which does not modify the original text but only returns the modified string; in this case, we did not assign the result to the new variable, so we have lost the effect of its operation.

```
Hello, World!
```

I would like to point to the difference in the invoking of the methods that we have used so far. Please look at the code below:

```
String.valueOf(23);
String text = new String("Alice has a cat");
text.endsWith("cat");
```

```
– It is the invoking of the `valueOf() method` on the
`**String class**`. We did not invoke this method on an
```

(object-oriented) *variable*, i.e. we did not invoke this method on a particular value of this variable, but only (in a *context-free* manner) on the `String class`. The `valueOf()` method is a *static method*, that is, it can be run without having (creating) a variable of the `String type`.

```
> **Note**
> Classes, methods, blocks and static variables are not
within the scope of this Handbook.
```

- this is invoking the `endsWith()` method on a **specific** variable. Depending on the value stored in the `text` variable, the action (result of the invocation) of this method will vary.
  The `endsWith()` method is an *instance method*, that is, we invoke it on a `String` *class object*. You will learn more about it in chapter Object-oriented programming.

Classes responsible for date and time

Now I will show you several classes that store the date and time and offer the opportunity to work on this data.

- `LocalDate`
- `LocalTime`
- `LocalDateTime`

Example of receiving current date and time

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
```

> ## Tip
>
> To use these classes, we need to import them (indicate to Java in which *package* (directory) they are located). At the beginning of the file, before the declaration of the class, add the following text:
>
> ```
> import java.time.*;
> ``
> ```

The result will vary depending on the start time, but it will have the format as below.

```
2019-04-08
20:21:16.726
2019-04-08T20:21:16.726
```

Based on the output, we see that:

- `LocalDate` represents date
- `LocalTime` represents time
- `LocalDateTime` represents date and time

> ## Note
>
> You will often encounter

You will often encounter also `Date` and `Calendar` classes where date and/or time are processed.

As an exercise, test the methods of these classes.

Class for simple mathematical operations

For simple mathematical operations, we can use the `Math` class.

Example of some basic mathematical operations

```
System.out.println(Math.max(5, 10));
System.out.println(Math.min(55, 100));
System.out.println(Math.abs(-77));
System.out.println(Math.ceil(3.55D));
System.out.println(Math.floor(3.55D));
System.out.println(Math.pow(2, 10));
System.out.println(Math.random());
System.out.println(Math.round(9.99D));
System.out.println(Math.sqrt(81));
System.out.println(Math.PI);
System.out.println(Math.E);
```

Note

`Math.PI` indicates a reference to a constant. A constant is similar to a variable, but it has a value assigned that cannot be changed. In addition, the names of constants are written in capital letters.

The result of running the above code:

```
10
55
77
4.0
3.0
1024.0
0.20464094804059307
10
9.0
3.141592653589793
```

```
2.718281828459045
```

- this value differ for each run; the `random()` function returns a pseudo-random value within the range of `<0, 1)`.

Your exercise now it to test the above for other values and using other methods.

Wrappers of simple types

Apart from the simple types mentioned before, everything in Java is an object. For each simple type, its equivalent, that is, a wrapper class, was created in Java.

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

Wrappers of simple types

Example of creation and application

```
byte byteValue = 56;
Byte byteObject = new Byte(byteValue);
short shortValue = 89;
Short shortObject = new Short(shortValue);
int intValue = 256;
Integer integerObject = new Integer(intValue);
long longValue = 5500L;
Long longObject = new Long(longValue);
float floatValue = 3.141592F;
Float floatObject = new Float(floatValue);
double doubleValue = 2.718281828D;
Double doubleObject = new Double(doubleValue);
char charValue = 'a';
Character characterObject = new Character(charValue);
```

```
boolean booleanValue = true;
Boolean booleanObject = new Boolean(booleanValue);

System.out.println("byteValue = " + byteValue);
System.out.println("byteObject = " + byteObject);
System.out.println("doubleValue = " + doubleValue);
System.out.println("doubleObject = " + doubleObject);
System.out.println("charValue = " + charValue);
System.out.println("characterObject = " +
characterObject);
System.out.println("booleanValue = " + booleanValue);
System.out.println("booleanObject = " + booleanObject);
```

The result is:

```
byteValue = 56
byteObject = 56
doubleValue = 2.718281828
doubleObject = 2.718281828
charValue = a
characterObject = a
booleanValue = true
booleanObject = true
```

Variables `xxxObject` are object-oriented, we can execute methods on them, while we cannot do it on `xxxValue` variables because they are variables of primitive types, that is, they only have a value.

Example of invocation of a few methods

```
double doubleValueFromByte = byteObject.doubleValue();
float floatValueFromInteger =
integerObject.floatValue();
char charValueFromCharacter =
characterObject.charValue();
boolean booleanValueFromBoolean =
booleanObject.booleanValue();
```

Autoboxing and unboxing

Thanks to the built-in *autoboxing* and *unboxing* mechanisms, we can shorten the record and conversion between primitive types and the corresponding object-oriented types (wrappers).

- autoboxing automatic conversion of simple types into complex ones. Where, for example, an `Integer` type is expected, we can

provide e.g. value 5 or a variable of the `int` type, and it will be automatically converted into the `Integer` type. The same applies to other types.**Example of *autoboxing*.**`Byte byteObject = 56; Short shortObject = 89; Integer integerObject = 256; Long longObject = 5500L; Float floatObject = 3.141592F; Double doubleObject = 2.718281828D; Character characterObject = 'a'; Boolean booleanObject = true;`

- unboxing automatic conversion of object-oriented types to primitive ones. Where, for example, a `double` type is expected, we can provide a variable of the `Double` type. It will be automatically converted into the `double` type. The same applies to other types.**Example of *unboxing*.**`byte byteValue = new Byte((byte) 56); short shortValue = new Short((short) 89); int intValue = new Integer(256); long longValue = new Long(5500L); float floatValue = new Float(3.141592F); double doubleValue = new Double(2.718281828D); char charValue = new Character('a'); oolean booleanValue = new Boolean(true);`
    - constructor `Byte(byte value)` requires passing a `byte` type value – it is required to cast the `int` type values

to `byte`: `(byte)`

```
– constructor `Short(short value)` requires passing the `short` type value – it is also required to cast the `int` type
```

value to `short`: `(short)`

The ranges of values for basic types and their wrappers are the same. The difference lies in the default values: for object-oriented types it is `null`, which has no equivalent (there is no such value) in primitive types.

When converting object-oriented types to primitive ones (*unboxing*), pay attention to whether the variable value is `null`. If this is the case, an error will appear.

An example of incorrect `unboxing`

```
Integer integerObject = null;
int intValue = integerObject;
```

Result:

```
Exception in thread "main"
java.lang.NullPointerException
```

Important

```
Integer a = 0;
Integer b = null;
```

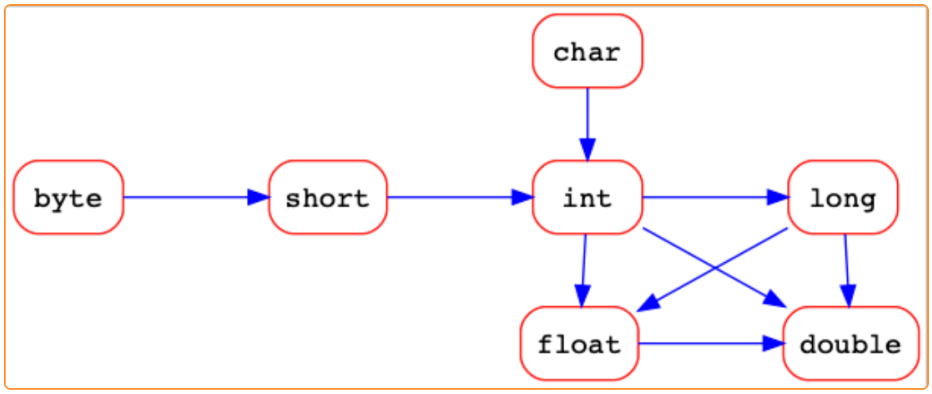
Variables `a` and `b` are not the same, `0` and `null` are completely different values. Variable `a` indicates the object of the `Integer` class that stores value `0`. Variable `b` does not indicate any object!

Tip You can compare a variable `b` to a remote control without a receiver so it has nothing to control.

Conversion of types

Java copes quite well with adjusting the types. Only in some cases we must use additional mechanisms to help it.

Diagram of conversion of types



Let us look at types: `byte`, `short`, `int` and `long`. Their ranges increase: byte is included in short, short in int, and int in long. Any value from the byte range will fit in the short range, etc. However, the result for the other direction is not so obvious. For example, if you store a small value in an int type variable (that fits in short or byte), you can *cast* an int type variable to a variable with a smaller

range.

Conversions from a smaller range to a larger one happen automatically (this is indicated by the direction of the arrows). Conversions from a larger range to a smaller one must be done consciously using the projection operator: `(<typ>)`.

Example of automatic conversions to extend the type

```
byte b = (byte) 56;
short s = (short) 89;
int i = b;
long l = s;
float f = i;
```

- `byte` type is extended automatically to the `int` type
- `short` type is extended automatically to the `long` type
- `int` type is extended automatically to the `float` type

Example of an explicit narrowing conversion

```
byte a = 127;
byte b = (byte) 130;
float c = 3.14F;

int d = 5;
byte e = (byte) d;
long f = (long) c;

System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("f = " + f);
```

- there is no need to cast, because value (literal) `127` is in the range of the `byte` type
- casting is necessary because value `130` is out of range; pay attention to the result!
- casting is necessary, because Java can only "see" the `int` type of variable `c` (it "cannot see" the value) and requires explicit casting
- casting is necessary, because the range is exceeded; besides, the fractional part is lost

The result is as below:

```
a = 127
b = -126
f = 3
```

Conversion from complex types to simple types

Objects of simple-type wrapper classes provide a number of methods for converting from complex types to simple types.

- `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`
  - `byteValue()`
  - `shortValue()`
  - `intValue()`
  - `floatValue()`
  - `doubleValue()`
- `Character`
  - `charValue()`
- `Boolean`
  - `booleanValue()`

Wrapper classes offer methods that enable creating objects based on values or simple-type variables.

- `Byte.valueOf(byte b)` creates a Byte class object based on the b value
- `Short.valueOf(short s)` creates a Short class object based on the s value
- `Integer.valueOf(int i)` creates an Integer class object based on the i value
- `Long.valueOf(long l)` creates a Long class object based on the l value
- `Float.valueOf(float f)` creates a Float class object based on the f value
- `Double.valueOf(double d)` creates a Double class object based on the d value
- `Boolean.valueOf(boolean b)` creates a Boolean class object based on the b value

Wrapper classes (except `Character`) provide additional methods for converting values stored in a `String` type to a primitive type. These are methods in the `parseXXX(String s)` form.

Example of converting values stored in `String`

```
System.out.println(Boolean.parseBoolean("true"));
System.out.println(Boolean.parseBoolean("TRUE"));
System.out.println(Boolean.parseBoolean("1"));
System.out.println(Boolean.parseBoolean("0"));
System.out.println(Integer.parseInt("123"));
System.out.println(Double.parseDouble("3.1415"));
```

Result:

Wynik

```
true
true
false
false
123
3.1415
```

Summary

What have you learned in this chapter?

How do we divide data types in Java? We divide them into simple (primitive) and complex (object-oriented, reference) types.

What are the differences between simple and complex types? Simple types have only a value. Complex types, in addition to storing value, provide methods for operations on them.

What is a variable? A variable is like a "place", or a "container", for data. A variable always has its own type and only data of this type can be stored in it.

How do we compare variables (values) of primitive types? We use the == sign, e.g., `a == 5` or `a == c`.

How do we compare variables of object-oriented types? We use the `equals()` method.

How can I check whether the object variable is `null`? By comparing the variable to the `null` value, e.g. `a == null`.

What types of integers are offered by Java? There are four integer types in Java: `byte`, `short`, `int` and `long`. All of them are *signed* (they have a sign). The `long` type is marked by adding letter `L` to the value.

Is any of the integer types default? Yes, it is the `int` type.

What floating point types are offered by Java? There are two floating point types: `float` and `double`; float is of single precision, while double is of double precision. The float type is marked by adding letter `F` to the value, while the double type is marked by adding letter `D`.

What is the character type? The character (`char`) type enables storing exactly one character in the variable. These are characters in the Unicode standard, e.g. `'a'`, `'Z'` or `' '`. The char type is a type without a sign (*unsigned*), which means that the cores are non-negative numbers in the range from `0` to `65556`.

What is the logical type? The logical type (`boolean`) can have only two values: `false` or `true`. It is used to check logic conditions.

What are object-oriented types? These are types that in addition to storing values also provide operations (methods). Java provides several thousand (over 4,000) types. Developers around the world provide more than ten hundred thousand types. When creating a class, you also create a new type.

What is the String class for? The `String` class is used for storing strings and provides methods for operations on them. It features a short form of declaration and initialization:

```
String text = "I love Java!";
```

What does it mean that the String **is** *immutable*?** This means that the class does not change its status, but only creates a new instance with the changed value. If you perform an operation on the `String` class that modifies data (changes the string), then the original value is not actually changed; the method returns only the changed string.

What classes can be used to support the date and time? These can be `LocalDate`, `LocalTime` or `LocalDateTime` classes. Also *older* classes can be often encountered: `Date` and `Calendar`.

What class can you use for simple mathematical operations? It is the `Math` class.

What happens when I try to invoke a method on a variable that is `null`? When attempting to invoke a method on a `null` reference (variable), a program runtime error will occur. In detail, the `NullPointerException` will appear.

What are simple type wrappers? These are curtain classes for all simple types available in Java. They expand their capabilities by providing methods; they are a bridge between simple and object types.

What is *autoboxing* and *unboxing*? *Autoboxing* is a mechanism for automatic conversion of simple types into complex ones. *Unboxing* is a mechanism that works in the opposite direction.

```
**Note**
The mechanism may return an error if you try to convert
a variable that stores the null value to a simple type
variable. In simple types there is no equivalent of the
null value!
```

What is conversion of types? Conversion of types is the assignment of one type to another. If you convert a *narrower* type into a *wider* one (e.g. short to int), then such conversion is safe. It is called *casting up*. If, on the other hand, you convert a *wider* type into a *narrower* one, then such conversion can be dangerous. In such cases you should use the casting operator: `(<type on which you`

such cases you should use the casting operator: `(<type on which you cast>).`

Exercises

- Create (declare and initialize) several variables of primitive types based on things that surround you or can be found in your industry. Think over which type to use. Display the values.
- Try to assign a value out of range for the variable. See the result displayed. Try using casing operator `( )`.
- Use wrapper classes and try to create object-oriented equivalents for all variables (of primitive types) from the previous exercise.
- Test the chosen methods (`xxValue()`, `valueOf()` and `parseXXX()`) from wrappers of simple types.
- Write a program (create a new class with the `main` method) that displays `Hello, <Your name>!` on the screen. Make your name stored in a variable of the `String` type.
- Write a program that displays your business card like the one below:`############################ # # # John Smith # # Sesame Street 345/7b # # New York 10019 # # # ############################`
    - use different signs, for instance:
        - – for horizontal lines
        - | for vertical lines
        - / i (lub +) for corners
    - make the horizontal line be stored in a variable; you will be able to declare its value only once , but use it twice
    - make the characters for vertical lines and corners also be stored in variables
    - create further variables for the name, surname, street, building number, flat number, zip code
    - add a variable (and its display) that stores the phone number
    - suggest further variables, e.g. area code, address (concatenation of street , building number and flat number)
    - add boolean type variable specifying whether to display the frame; if it has a false value, then do not display it
    - create any variable and try to use it without initialization; observe the error
- Create a few `String` variables and test the presented methods that operate on strings.
- Create a few variables of the `LocalDate`, `LocalTime` and `LocalDateTime` types and test invoking a few methods on them. Display the results.
- Test the methods of the Math class.

Complete Lesson