Unit 4

Primitive data types in Java

Summary

- Representation of numbers in Java: the primitive numeric data types `int`, `long`, `short`, `byte`, `float`, `double`
- Set of values that can be represented, and operations on such primitive numeric data types
- Difference between the assignment for primitive data types and the assignment of references to objects
- Operators obtained by composition and expressions with side-effect
- Definition of constants
- Precision in the representation
- Conversion of strings in numbers and viceversa; input and output of numeric values
- Compatibility among primitive numeric data types; type conversions (cast)
- Other primitive data types in Java: `char` and `boolean`

4.1 Data types in mathematics

To effectively describe the nature of data that can be represented in a Java program and the operations to manipulate such data, we will use the concept of **data type**.

A data type is characterized in mathematical terms by:

- A *domain*, i.e., a set of possible values (e.g., integer numbers, real numbers, etc.);
- A set of *operations* on the elements of the domain (e.g., sum, multiplications, etc.);
- A set of *literals*, denoting mathematical constants (e.g., 23).

Such a characterization allows us to identify in a precise way the values that we intend to represent and the operations to manipulate them.

4.2 Primitive data types in Java

To deal with numerical information, Java uses six predefined data types, called **primitive numerical data types**. These are `int`, `long`, `short`, `byte`, `float`, and `double`, and they allow us to represent integer and real numbers.

Java offers two additional `non-numeric primitive data types`: `char` (to represent alphanumeric characters and special symbols) and `boolean` (to represent the truth values `true` and `false`).

We will describe these data types in Java by specifying for each of them:

- The *domain*: the set of possible values that can be represented in the memory of the computer by means of the primitive data type (note that this set will always be finite);
- The set of *operations*: operators of the programming language that allow us to perform elementary operations on values of the primitive data type (e.g., `+`, `-`, `/`, `*`, etc.)
- The set of *literals*: symbols of the language that define values of the primitive data type (e.g., `10`, `3.14`, `'A'`, `true`, etc.)

Moreover, we will specify the size of the memory occupied by a value of a certain data type, which will be significant for the numeric data types.

4.3 The data type `int`

Type	`int`	
Dimension	32 bit (4 byte)	
Domain	the set of integer numbers in the interval $[-2^{31}, +2^{31}-1]$ (more than 4 billion values)	
Operations	`+`	sum
	`-`	difference
	`*`	product
	`/`	integer division
	`%`	rest of the integer division
Literals	sequences of digits denoting values of the domain (e.g., 275930)	

Example::

```
int a, b, c;  // Declaration of variables of type int
a = 1;        // Use of literals
b = 2;
c = a + b;    // Arithmetic expression that involves operators of the language
```

4.4 Variables of primitive types and variables of type reference to an object

There is a substantial difference between variables of type reference to an object and variables of primitive types:

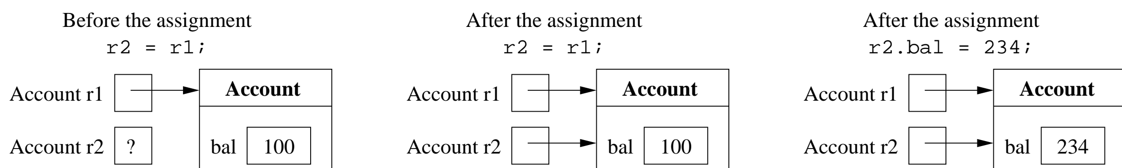
- the value of a variable of type reference to object is a **reference** to an object, and not the object itself;
- the value of a variable of a primitive type is a **value** of the primitive type itself (and not the reference to a value or an object).

Example:

- Use of variables of type reference to object:

```
public class Account {
  public int bal;
  public Account(int x) {
    bal = x;
  }
}
...
Account r1, r2;
r1 = new Account(100);
r2 = r1;
r2.bal = 234;
System.out.println(r1.bal); // prints 234
```

  The variables `r1` and `r2` contain **references to objects**, **not objects**. The assignment `r2 = r1` assigns to `r2` the reference to the object of type `Account`, not the object itself. Each successive modification to the object referenced from `r2` will also be visible through `r1`.



- Use of variables of primitive types:

```
int c1, c2;
c1 = 100;
c2 = c1;
c2 = 234;
System.out.println(c1); // prints 100
```

The assignment `c2 = c1` assigns the value `100` to `c2`. Successive modifications of `c2` do not influence `c1`.

Before the assignment `c2 = c1;`	After the assignment `c2 = c1;`	After the assignr `c2 = 234;`
int c1 100	int c1 100	int c1 100
int c2 ?	int c2 100	int c2 234

4.5 Methods that modify variables of primitive types

We have a similar difference in parameter passing, when we have a parameter of a primitive data type as opposed to a parameter of type reference to object. We will see next that passing parameters of type reference to object allows us to design methods that modify variables in the calling program unit, which is impossible if we pass directly primitive data types as parameters.

Suppose we want to write a method that modifies a variable of a primitive type (i.e., a method with *side-effect* on variables). For example, we try to implement a method that increments a variable of type `int`:

```
public static void increment(int p) {
  p = p + 1;
}
```

if we now invoke the `increment()` method as follows:

```
public static void main(String[] args){
  int a = 10;
  increment(a);
  System.out.println(a);   // prints 10
}
```

we see that the program prints `10`, instead of `11`, as we could have expected. This is because, during the invocation of the `increment()` method, the value 10 stored in the local variable `a` is *copied* in the formal parameter `p`. The `increment()` method modifies the formal parameter `p`, but does not modify the content of the local variable `a`.

To obtain the desired effect, we can instead pass a variable that is a reference to an object that contains the integer:

```
public static void increment(MyInteger x) {
  x.a = x.a + 1;
}
```

where the class `MyInteger`, that acts as a *wrapper* of the integer, could simply be defined as follows:

```
class MyInteger {
  public int a;
}
```

This allows us to rewrite our program as follows:

```
public static void main(String[] args){
  MyInteger r = new MyInteger();
  r.a = 10;
  increment(r);
  System.out.println(r.a); // prints 11
}
```

Note that the value 10 is stored in the instance variable `a` of the object `MyInteger`, which is referenced by `r`. The reference stored in the variable `r` is copied in the formal parameter `x` of the method `increment()` when the method is invoked. Hence, `x` refers to the same object to which `r` refers, and the increment performed on the instance variable of such an object is visible also after the `increment()` method has terminated.

4.6 Wrapper classes for the primitive data types

In fact, Java already provides so-called **wrapper classes** for the primitive data types. Each primitive data type has an associated class, that typically has the same name as the data type, but starts with a capital letter

(except for `int` and `char`, where the name of the wrapper class is different):

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

- These classes define special static methods that allow us to perform operations (such as conversions from and to strings) on the corresponding primitive data types.
- Moreover, they allow us to "wrap" values of primitive types into objects (which is why they are called *wrapper* classes). We already discussed the usefulness of this aspect in parameter passing.

4.7 Boxing and unboxing

To make use of the wrapper classes, we need to have means to convert a value of a primitive data type (such as `int`) to an "equivalent" value of the corresponding wrapper class (called *boxing*), and vice-versa (called *unboxing*).

- Boxing is done by creating an object of the corresponding wrapper class passing the primitive type value as an argument to the wrapper class constructor.

  Example:

  ```
  Integer anIntegerObject = new Integer(25);
  Double aDoubleObject = new Double(3.6);
  Character aCharacterObject = new Character('A');
  ```

  The variable `anIntegerObject` now references an object of the class `Integer` that corresponds to the `int` value `25`. Similarly, for the variables `aDoubleObject` and `aCharacterObject`.
- Unboxing is done by invoking specific methods of the wrapper classes (called *accessor methods*), that return the primitive data type value corresponding to an object of a wrapper class. The names of an accessor method is always *`primitivetype`*`Value()`, and it returns a value of *`primitivetype`*.

  Example:

  ```
  int i = anIntegerObject.intValue();
  double d = aDoubleObject.doubleValue();
  char c = aCharacterObject.charValue();
  ```

  The variable `i` now contains the value `25`, the variable `d` the value `3.6`, and the variable `c` the value `'A'`.

4.8 Automatic boxing and unboxing

Starting with version 5.0, Java will automatically do the boxing and unboxing operations.

- Automatic boxing takes place when we directly assign a value of a primitive type to an object of a wrapper class, without explicitly invoking the constructor via `new`. In this case, the call to the constructor is automatically inserted by the Java compiler.

  Example: The three statements we have seen before performing the boxing operation can be equivalently rewritten in the following simpler way without making use of the constructors.

  ```
  Integer anIntegerObject = 25;
  Double aDoubleObject = 3.6;
  Character aCharacterObject = 'A';
  ```

- Similarly, automatic unboxing takes place when we use an object of a wrapper class instead of a value of a primitive data type. In this case, the Java compiler automatically inserts a call to the appropriate access method.

  Example: The three statements we have seen before performing the unboxing operation can be equivalently rewritten in the following simpler way without making use of the access methods.

```
int i = anIntegerObject;
double d = aDoubleObject;
char c = aCharacterObject;
```

4.9 Reading of numbers of type `int`

To read a number of type `int` from an input channel, we use:

1. a method to read a string from an input channel (e.g., `showInputDialog()` of the class `JOptionPane`)
2. the static method `parseInt()` of the class `Integer` to obtain the number corresponding to the string read as a value of type `int`.

Example:

```
String s = JOptionPane.showInputDialog("Insert an integer number");
int i = Integer.parseInt(s);
```

or

```
int i = Integer.parseInt(
          JOptionPane.showInputDialog("Insert an integer number"));
```

Note that, if `parseInt()` is called on a string containing characters different from digits, an error occurs when the program is run.

4.10 Writing of numbers of type `int`

To write a number of type `int`, we can directly use the `print()` or `println()` methods:

Example:

```
int i = 1;
System.out.println(4);
System.out.println(i);
System.out.println(i + 4);
```

Note: the symbol `+` can be used both for the sum of two numbers and to concatenate two strings: `"aaa" + "bbb"` corresponds to `"aaa".concat("bbb")`.

Note the difference between the following two statements:

```
System.out.println(3 + 4);   // prints 7 (as int); + denotes sum
System.out.println("3" + 4); // prints 34 (as String), since the integer 4 is
                             // first converted to a String; + denotes concat
```

- In the first statement, "`+`" is applied to two integers, and hence denotes the addition operator. Hence, the argument `3+4` of `println()` is of type `int`.
- In the second statement, , "`+`" is applied to a string and an integer, and hence denotes string concatenation. More precisely, the integer `4` is first converted to the string `"4"`, and then concatenated to the string `"3"`. Hence, the argument `"3"+4` of `println()` is of type `String`.

Both statements are correct, since the method `println()` is overloaded: the Java library contains both a version that accepts an integer as parameter, and a version that accepts a string as parameter.

4.11 Integer expressions

The following fragment of a Java program shows expressions with operators on integers:

```
int a, b, c;
a = 10 / 3 + 10 % 3;
b = 2 * -3 + 4;
c = 2 * (a + b);
```

The following precedence rules between operators hold in Java (these are the same as those used in arithmetics):

1. unary `+`, unary `-` (e.g., `-x`)
2. `*`, `/`, `%`
3. `+`, `-`

Brackets can be used to change the way subexpressions have to be grouped. Note that in Java, only round brackets, `(` and `)`, and not square or curly brackets, can be used to group subexpressions

Example: In Java, the expression `a+b*-c` is equivalent to `a+(b*(-c))`

4.12 Numeric overflow

The set of values that can be represented with a primitive type is limited to a specific interval (e.g, [-2^{31},2^{31}-1] for the type `int`). By applying arithmetic operators to values of a given data type, one could obtain a result that falls outside of this interval, and hence cannot be represented with the same primitive data type. Such a situation is called **overflow**.

Example:

```
int x = 2147483647;     // maximum value that can be represented as an int
int y = x + 1;          // operation that causes overflow, since the result
                        // 2147483648 cannot be represented as an int
System.out.println(y); // prints -2147483648 instead of 2147483648
                        // (which is the number we would expect)
```

Note that, by adding `1` to the biggest number that can be represented as an `int`, we have an overflow, and the result is the smallest number that can be represented as an `int`. Informally, it is as if we were "cycling" around the representation.

4.13 Combined assignment operators

Consider the following fragment of a Java program:

```
int sum, a, salary, increase;
sum = sum + a;
salary = salary * increase;
```

It can be abbreviated as follows:

```
sum += a;
salary *= increase;
```

In general, the assignment:

```
x = x operator expression
```

can be abbreviated as:

```
x operator= expression
```

For each of the arithmetic operators `+`, `-`, `*`, `/`, `%`, there is the corresponding combined assignment operator `+=`, `-=`, `*=`, `/=`, `%=`.

4.14 Increment and decrement operators

To increment by 1 the value of an integer variable `x`, we can use any of the following three statements, which are all equivalent:

```
x = x + 1;
x += 1;
x++;
```

Note that the most compact form is the one using the *post-increment* operator `++`.

Similarly, to decrement by 1 an integer variable, we can use any of the following three statements, which are all equivalent:

```
x = x - 1;
x -= 1;
x--;
```

Also in this case, the most concise form is the one that uses the *post-decrement* operator `--`.

4.15 Expressions with side-effect and statements (optional)

Java uses the term *expression* to indicate two different notions:

- those expressions whose only effect is the calculation of a value, such as the expressions of type `int`, which can be composed according to the rules of arithmetics;
- those expressions that, besides calculating a value, correspond to an operation on memory, such as an assignment (simple or combined) or an increment. We call these *expressions-with-side-effect*. Recall that we use the term side-effect to indicate a modification of the state (i.e., the memory) of the program. Expressions of this type can be transformed into statements by ending them with "`;`", and this is exactly what we have done till now to assign a value to (or increment/decrement) variables. By transforming an expression-with-side-effect into a statement, we give up considering it an expression that has an associated value.

Example:

- `23*x+5` is a mathematical expression;
- `x = 7` is an expression-with-side-effect that is valid in Java and whose value is (the right hand side of the assignment). If we end them with "`;`" we obtain the statement `x = 7;`
- `y = x = 7` is also a valid Java expression, which has two side-effects: the first one assigns 7 to `x`, while the second one assigns the value of the expression `x = 7` (which, as said, is 7) to `y`.

While Java allows us to use both types of expressions without limitations, we will *use expressions-with-side-effect only to form statements*, and we will always avoid their use inside arithmetic expressions.

Example: The statement

```
x = 5 * (y = 7);
```

should be rewritten as follows:

```
y = 7;
x = 5 * y;
```

This distinction is motivated by the fact that expressions are an *abstraction for the mathematical concepts of function and of function application*, while expressions-with-side-effect (statements) are an *abstraction for the concept of assignment*, i.e., of the modification of a memory location of the program.

4.16 Definition of constants and magic numbers

A **magic number** is a numeric literal that is used in the code without any explanation of its meaning. The use of magic numbers makes programs less readable and hence more difficult to maintain and update.

Example:

```
int salary = 20000 * workedhours;
// what is the meaning of 20000?
```

It is better to define symbolic names, so called **constants**, and use these instead of numeric literals.

```
// definition of a constant SALARY_PER_HOUR
final int SALARY_PER_HOUR = 20000;
...
// now it is clear what SALARY_PER_HOUR means
int salary = SALARY_PER_HOUR * workedhours;
```

`SALARY_PER_HOUR` is a `constant`, which in Java is a variable whose content does not change during the execution of the program. We can declare a constant by using the modifier `final` in the variable declaration, which indicates that the value of the variable cannot be modified (i.e., it stays constant).

The main advantages of using constants are:

- *Readability of the program:* an identifier of a constant with a significant name is much more readable than a magic number (e.g., `SALARY_PER_HOUR` is self-explaining, `20000` is not);
- *Modifiability of the program:* to modify the value of a constant used in the program it is sufficient to change the definition of the constant (e.g., `final int SALARY_PER_HOUR = 35000`), while with magic numbers we would have to modify all occurrences of the value in the program (e.g., by replacing occurrences of `20000` con `35000`). Notice that it may be difficult to determine which occurrence of the specific magic number actually corresponds to the one we have to change.

Note: The declarations of constants (containing the `final` modifier) can be dealt with in the same way as variable declarations. Specifically, if the declaration is local to a method, the scope of the constant is the method itself. Instead, if we apply the `final` modifier to the declaration of an instance variable, then the constant will be associated to each object the moment it is created, and different objects may have different values for the constant.

4.17 Other primitive data types for integer numbers: `byte`

Type	`byte`	
Dimension	8 bit (1 byte)	
Domain	the set of integer numbers in the interval $[-2^7, +2^7-1] = [-128, +127]$	
Operations	`+`	sum
	`-`	difference
	`*`	product
	`/`	integer division
	`%`	rest of the integer division
Literals	sequences of digits denoting values of the domain (e.g., 47)	

Example:

```
byte a, b, c;               // Declaration of variables of type byte
a = 1;                      // Use of literals
b = Byte.parseByte("47");   // Conversion from String to byte
c = a - b;                  // Arithmetic expression
```

4.18 Other primitive data types for integer numbers: `short`

Type	`short`	
Dimension	16 bit (2 byte)	
Domain	the set of integer numbers in the interval $[-2^{15}, +2^{15}-1] = [-32768, +32767]$	
Operations	`+`	sum
	`-`	difference
	`*`	product
	`/`	integer division
	`%`	rest of the integer division
Literals	sequences of digits denoting values of the domain (e.g., 22700)	

Example:

```
short a, b, c;                  // Declaration of variables of type short
a = 11300;                      // Use of literals
b = Short.parseShort("22605");  // Conversion from String to short
c = b % a;                      // Arithmetic expression
```

4.19 Other primitive data types for integer numbers: `long`

Type	`long`	
Dimension	64 bit (8 byte)	
Domain	the set of integer numbers in the interval $[-2^{63}, +2^{63}-1]$	
Operations	`+`	sum
	`-`	difference
	`*`	product
	`/`	integer division
	`%`	rest of the integer division
Literals	sequences of digits ending with an `l` (or `L`) denoting values of the domain (e.g., 9000000000L)	

Example:

```
long a, b, c;                     // Declaration of variables of type long
a = 9000000000L;                  // Use of literals
b = Long.parseLong("9000000000l"); // Conversion from String to long
c = b / 300000L
```

4.20 Primitive data types for real numbers: `double`

Besides the types for representing integer numbers, in Java there are two primitive data types for representing real numbers. Due to the way in which real numbers are represented internally in memory, these numbers are also called **floating point numbers**.

The data type for floating point numbers that is used by default in the Java mathematical library is `double`.

Type	`double`		
Dimension	64 bit (8 byte)		
Domain	set of 2^{64} positive and negative real numbers	Minimum absolute value	$1.79769313486231570 \cdot 10^{-308}$
		Maximum absolute value	$2.250738585072014 \cdot 10^{+308}$
		Precision	$\sim$ 15 decimal digits
Operations	`+`	sum	
	`-`	difference	
	`*`	product	
	`/`	division	
Literals	sequences of digits with decimal dot optionally ending with a `d` (or `D`) denoting values of the domain (e.g., `3.14` or `3.14d`)		
	representation in scientific notation (e.g., `314E-2` or `314E-2d`)		

Example:

```
double pi, p2;  // Declaration of variables of type double
pi = 3.14;      // Use of literals
p2 = 628E-2d;   // Use of literals
p2 = pi * 2;    // Arithmetic expression
```

4.21 Primitive data types for real numbers: `float`

Type	`float`		
Dimension	32 bit (4 byte)		
Domain	set of 2^{32} positive and negative real numbers	Minimum absolute value	$1.4012985 \cdot 10^{-38}$
		Maximum absolute value	$3.4028235 \cdot 10^{+38}$
		Precision	$\sim$ 7 decimal digits
Operations	`+`	sum	
	`-`	difference	
	`*`	product	
	`/`	division	
Literals	sequences of digits with decimal dot ending with an `f` (or `F`) denoting values of the domain (e.g., `3.14f`)		
	representation in scientific notation (e.g., `314E-2f`)		

Example:

```
float pi, a, b;  // Declaration of variables of type float
pi = 3.14f;      // Use of literals
a = 314E-2F      // Use of literals
a++;             // Use of increment operator (equivalent to: a = a + 1.0d;)
```

4.22 Reading of numbers of type `double` or `float`

To read a number of type `double` (or `float`) from an input channel, we use:

1. a method to read a string from an input channel (e.g., `showInputDialog()` of the class `JOptionPane`)
2. the static method `parseDouble()` (respectively, `parseFloat()`) of the class `Double` (respectively, `Float`) to obtain the number corresponding to the string read as a value of type `double` (respectively, `float`).

Example:

```
String s = JOptionPane.showInputDialog("Insert a number (e.g., 3.14)");
double i = Double.parseDouble(s);
```

or

```
double i = Double.parseDouble(
            JOptionPane.showInputDialog("Insert a number (e.g., 3.14)"));
```

4.23 Writing of numbers of type `double` or `float`

To write a number of type `double` or `float`, we can directly use the `print()` or `println()` methods:

Example: The following code fragment

```
double d = 98d;
System.out.println("d = " + d);
float x = 0.0032f;
System.out.println("x = " + x);
```

prints on the screen

```
d = 9.8E1
x = 3.2E-3
```

4.24 Reading of numbers using the `Scanner` class

Starting with Java version 5.0, we have an alternative way to read numbers from an input channel, namely by making use of the `Scanner` class. We first create a `Scanner` object that reads from the input channel. Then we can directly read a value of a primitive numeric data type by invoking on the `Scanner` object one of the methods `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, and `nextDouble()`.

Example:

```
Scanner scanner = new Scanner(System.in);
...
System.out.println("Insert a byte");
byte b = scanner.nextByte();
System.out.println("Insert a short");
short s = scanner.nextShort();
System.out.println("Insert an int");
int i = scanner.nextInt();
System.out.println("Insert a long");
long l = scanner.nextLong();
System.out.println("Insert a float");
float f = scanner.nextFloat();
System.out.println("Insert a double");
double d = scanner.nextDouble();
```

4.25 Exercise: the class `BankAccount`

Specification: Write a class for handling bank accounts that are characterized by the name and surname of the owner and by the balance of the account in Euro. Implement the methods `deposit()` and `withdraw()` and the method `toString()` to obtain a string containing the information on a bank account.

Example of usage:

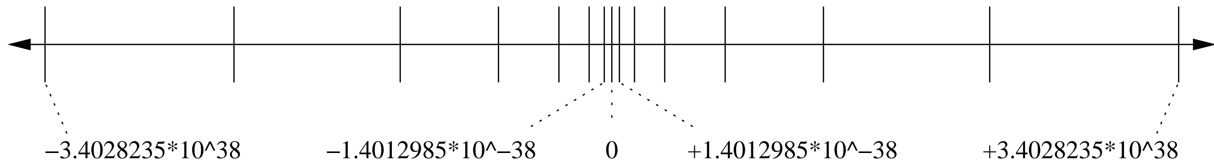
```
public class TestBankAccount {
  public static void main (String[] args) {
    BankAccount ba = new BankAccount("Mario", "Rossi");
    System.out.println("Before the operations: " + ba);
```

```
    ba.deposit(1000);
    ba.withdraw(100);
    System.out.println("After the operations:  " + ba);
  }
}
```

4.26 Precision in the representation: rounding errors

Not all numbers between $-3.4028235 \cdot 10^{+38}$ and $+3.4028235 \cdot 10^{+38}$ can be represented as a `float` (similar considerations hold for `double`).

This aspect is shown in the picture below: the closer we are to zero, the closer to each other are the numbers that can be represented (depicted by the vertical lines); the more we move away from zero the wider from each other are the numbers that can be represented.



Example: The number that is nearest to $+3.4028235 \cdot 10^{+38}$ and that can be represented as a `float` is $+3.4028234 \cdot 10^{+38}$.

This leads to approximations due to rounding errors in computing the values of expressions.

Example:

```
float x = 1222333444.0f;
System.out.println("x   = " + x);
x += 1.0;
System.out.println("x+1 = " + x);
```

prints

```
x   = 1.222333444E9;
x+1 = 1.222333444E9;
```

while

```
int j = 1222333444;
System.out.println("j   = " + j);
j += 1;
System.out.println("j+1 = " + j);
```

prints

```
j   = 1222333444;
j+1 = 1222333445;
```

4.27 Precision in measures

The precision of the result of an operation depends from the precision with which we know the data.

Example: Suppose we know the dimension of a rectangle with a precision of only one decimal digit after the point. Then, the area of the rectangle cannot have a higher precision, and hence it makes no sense to consider the second decimal digit as significant:

$9.2 * 5.3 = 48.76$ (the second decimal digit is not significant)

$9.25 * 5.35 = 49.48$ (here it is)

This is not caused by the representation of numbers in a programming language, but by the limitations on our knowledge about the input values of a problem.

4.28 Predefined static methods for mathematical operations

To calculate mathematical functions on values of a numeric type, Java defines some classes containing static methods that can be used to calculate such functions. For example, the predefined class `Math` contains several

such methods, e.g., to calculate the square root (`sqrt()`), the absolute value (`abs()`), trigonometric functions (`sin()`, `cos()`, `tan()`), etc.

The following table is taken from the official documentation of the Java APIs:

Method Summary	
static double	**abs**(double a) Returns the absolute value of a double value.
static double	**ceil**(double a) Returns the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical integer.
static double	**cos**(double a) Returns the trigonometric cosine of an angle.
static double	**exp**(double a) Returns the exponential number *e* (i.e., 2.718...) raised to the power of a double value.
static double	**floor**(double a) Returns the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer.
static double	**log**(double a) Returns the natural logarithm (base *e*) of a double value.
static double	**max**(double a, double b) Returns the greater of two double values.
static double	**min**(double a, double b) Returns the smaller of two double values.
static double	**pow**(double a, double b) Returns of value of the first argument raised to the power of the second argument.
static double	**random**() Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
static long	**round**(double a) Returns the closest long to the argument.
static double	**sin**(double a) Returns the trigonometric sine of an angle.
static double	**sqrt**(double a) Returns the correctly rounded positive square root of a double value.
static double	**tan**(double a) Returns the trigonometric tangent of an angle.

Example:

```
int j = -2;
System.out.println(Math.abs(j));    // prints 2
```

4.29 Expressions that involve different primitive numeric types

When we have an expression that involves values of different data types, the type of the result is determined according to the following table. The table shows the type of the result of an expression of the form `a+b`, for each possible pair of types for `a` and `b`:

a+b	byte	short	int	long	float	double
byte	int	int	int	long	float	double
short	int	int	int	long	float	double
int	int	int	int	long	float	double
long	long	long	long	long	float	double
float	float	float	float	float	float	double
double	double	double	double	double	double	double

Example:

`int a; short b;` implies that `(a+b)` is an expression of type `int`.
`int a; float b;` implies that `(a+b)` is an expression of type `float`.
`float a; double b;` implies that `(a+b)` is an expression of type `double`.

The table reflects the following rules for the type of an arithmetic expression constituted by an arithmetic operator applied to two operands of different types:

- If the type of one operand denotes a subset of values denoted by the type of the other operand, then the type of the expression is the one with the larger set of values.

- If one operand is of an integer type (`byte`, `short`, `int`, `long`) and the other is of a floating point type (`float`, `double`), then the result of floating point type.
- Each time an arithmetic operation is performed on primitive types that are smaller than `int` (i.e., `byte` or `short`), then the compiler inserts a conversion of the types into `int` before executing the operation. Hence, the result will be at least of type `int`.

Note: Each time we want to assign the result of an operation to a variable that is of a type smaller than `int`, we have to insert an explicit type conversion (see below).

4.30 Assignments between different primitive numeric types

A value of a given type *cannot* be assigned to a variable of a data type with a smaller size, because we would risk to lose information. Moreover, a floating point value cannot be assigned to an integer variable.

The following table describes whether the assignment `a=b` is legal, for each possible type for `a` (on the rows) and `b` (on the columns).

a=b	byte	short	int	long	float	double
byte	OK	ERROR	ERROR	ERROR	ERROR	ERROR
short	OK	OK	ERROR	ERROR	ERROR	ERROR
int	OK	OK	OK	ERROR	ERROR	ERROR
long	OK	OK	OK	OK	ERROR	ERROR
float	OK	OK	OK	OK	OK	ERROR
double	OK	OK	OK	OK	OK	OK

Example:

- `int a; long b; a = b;`
  Error: a value of type `long` cannot be assigned to a variable of type `int`.
- `int a; float b; a = a + b;`
  Error: the expression `a+b` is of type float, and a value of type `float` cannot be assigned to a variable of type `int`.

4.31 Explicit type conversion (casting)

If we want to compile and execute the wrong assignment statements in the previous table, we have to insert an **explicit type conversion** (also called type **cast**).

Cast

Syntax:

```
(type) expression
```

- *`type`* is the name of a type
- *`expression`* is an expression whose type will be forced to *`type`*

Semantics:

Converts the type of an expression to another type in such a way that operations involving incompatible types become possible.

Example:

```
int a = (int) 3.75;      // cast to int of the expression 3.75 (of type double)
System.out.println(a);   // prints 3
```

When we perform a cast, the result could be affected by a **loss of precision**: in the previous example, the value `3.75` is **truncated** to `3`.

Example:

```
double d; float f; long l; int i; short s; byte b;

// The following assignments are correct
d = f;   f = l;   l = i;   i = s;   s = b;
```

```
// The following assignments are NOT correct
f = d;   l = f;   i = l;   s = i;   b = s;

// The following assignments are correct,
// but the result could be affected by a loss of precision
f = (float)d;   l = (long)f;   i = (int)l;   s = (short)i;   b = (byte)s;
```

4.32 The primitive data type `char`

Strings are constituted by single characters, which are values of type `char`. A variable of type `char` can contain just a single character. The domain of the type `char` is constituted by the more than 64000 characters of the Unicode standard. The Unicode standard establishes a correspondence between numbers and symbols that can be either alphabetical, numerical, or special symbols in various languages (including the Asiatic ones). For example, the character `'A'` corresponds to the numeric code 65., the character `'B'` to the numeric code 66, etc. For more details on the Unicode standard, you can consult the website `http://www.unicode.org/`.

Literals of type `char` can be denoted in various ways; the simplest one is through single quotes.

Example:

```
char c = 'A';  char c = '0';
```

4.33 Operations that involve values of type `char`

- Conversion from `char` to `int`, which corresponds to calculating the Unicode code of a character:

```
char c = 'A';
int i = c;             // i contains the Unicode code of the character 'A'
System.out.println(i); // prints 65
```

- Conversion from `int` to `char`, which corresponds to obtain a character from its Unicode code:

```
int i = 65;            // Unicode code of the character 'A'
char c = (char) i;
System.out.println(c); // prints 'A'
```

- Conversion from `char` to `String`:

```
char c = 'A';
String s = String.valueOf(c);
String s1 = Character.toString(c);  // is equivalent to the previous statement
```

- Extraction of a `char` from an object of type `String`:

```
String s = "hello";
char c = s.charAt(0); // extracts the character in position 0 from "hello",
                      // i.e., 'h', and assigns it to the variable c
```

- Reading a `char`:

```
String s = JOptionPane.showInputDialog("Insert a character");
char c = s.charAt(0);
```

- Writing a `char`:

```
char c = 'a';
System.out.println(c);
```

4.34 Boolean algebra: domain and operators

Java is equipped with the primitive data type `boolean` that allows us to deal with expressions that denote truth values (i.e., expressions whose value can be either `true` or `false`).

We recall first the basic notions of **boolean algebra**.

Domain	truth values **true** and **false**	
Operators	**and**	conjunction: a **and** b is **true** $\Leftrightarrow$ both a and b are **true**
	or	disjunction: a **or** b is **true** $\Leftrightarrow$ a is **true** or b is **true** (or both are)
	not	negation: **not** a is **true** $\Leftrightarrow$ a is **false**

Example:

- **true and false** is **false**
- **true or false** is **true**
- **not true** is **false**
- **false or** (**true and** (**not false**)) is equivalent to
  false or (**true and true**), which is equivalent to
  false or true, which is equivalent to
  true

4.35 Boolean expressions with variables: truth tables

We can obtain the value of a boolean expression with variables by substituting each variable with its truth value, and simplifying according to the meaning of the operators. To characterize the meaning of a boolean expression, we can construct a table in which, for each possible combination of truth values for the variables, we specify the truth value of the whole expression. Such a table is called a **truth table**.

Truth tables for the boolean operators

a	b	a **and** b
true	**true**	**true**
false	**true**	**false**
true	**false**	**false**
false	**false**	**false**

a	b	a **or** b
true	**true**	**true**
false	**true**	**true**
true	**false**	**true**
false	**false**	**false**

a	**not** a
true	**false**
false	**true**

Example:

a	b	c	(a **and** (**not** b)) **or** c
true	**true**	**true**	**true**
false	**true**	**true**	**true**
true	**false**	**true**	**true**
false	**false**	**true**	**true**
true	**true**	**false**	**false**
false	**true**	**false**	**false**
true	**false**	**false**	**true**
false	**false**	**false**	**false**

4.36 The primitive data type `boolean`

Type	`boolean`		
Dimension	1 bit		
Domain	the two truth values **true** and **false**		
Operations	`&&`	**and**	Note: in a `&&` b, the value of b is computed only if a is **true**
	`||`	**or**	Note: in a `||` b, the value of b is computed only if a is **false**
	`!`	**not**	
Literals	`true` and `false`		

Example:

```
boolean a,b,c,d,e;
a = true;
b = false;
c = a && b;              // c = a and b
d = a || b;              // d = a or b
e = !a;                  // e = not a
System.out.println(e);   // prints a string representing a boolean value,
                         // in this case the string "false" is printed
```

4.37 Expressions of type `boolean`

The following are simple expressions of type `boolean`:

- The *constants* `true`, `false`;

- *Variables* declared of type `boolean`;
- *Comparison operators* applied to primitive data types: `==` , `!=` , `>` , `<` , `>=` , `<=`
- *Calls to predicates* (i.e., methods that return a value of type `boolean`).

Complex expressions can be constructed from simple boolean expressions using the boolean operators `!`, `&&`, and `||`. For such operators, the following precedences hold:

1. `!`
2. `&&`
3. `||`

Example: The following is a correct expression of type `boolean`:

```
a || ! b && c
```

It is equivalent to:

```
a || ((! b) && c)
```

4.38 Comparison operators

When applied to primitive data types, the comparison operators return a value of type `boolean`.

`==`	equal to (**Attention**: `=` is different from `==`)
`!=`	different from
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Example:

`10 < 20`	is a boolean expression equivalent to	`true`
`10 == 20`	is a boolean expression equivalent to	`false`
`10 != 20`	is a boolean expression equivalent to	`true`

Example:

```
boolean x = 10 < 20;     // assignment of the value of a boolean expression
System.out.println(x);   // prints "true"
```

4.39 Comparing floating point numbers

A consequence of the rounding errors when representing or performing operations on floating point numbers is that the comparison between such numbers could produce unexpected results.

Example: Consider the following code fragment:

```
double r = Math.sqrt(2);   // computes the square root of 2
double d = (r * r) - 2;
System.out.println(d);     // prints 4.440892098500626E-16 instead of 0
boolean b = (d == 0);
System.out.println(b);     // prints false (we would expect true)
```

Hence, when *comparing floating point numbers*, we have to take into account the following:

- We cannot use simply `==` to perform the comparison.
- Two values x and y are close to each other if $|x - y| \leq eps$, where eps is a very small number (e.g., 10^{-14}).
- If x and y are very big or very small, it is better to use $\frac{|x-y|}{\max(|x|,|y|)} \leq eps$ instead. If one of the two numbers is zero, we must not divide by $\max(|x|, |y|)$.

4.40 Predicates

Methods that return a value of type `boolean` are called **predicates**.

Example: The following is the definition of a static method that is a predicate:

```
public static boolean implies(boolean a, boolean b) {
    return (!a || b); // not a or b
}
```

It can be used as shown in the following code fragment:

```
boolean x, y, z;
...
z = implies(x, y);
```

4.41 Precedence between operators of different types

When we have to evaluate complex expressions in which appear operators of different types, we have to take into account the various precedences:

1. logical negation (`!`) – high precedence
2. arithmetic operators
3. relational operators
4. logical conjunction (`&&`) and logical disjunction (`||`) – low precedence

Example:

```
 a+2  ==  3*b   ||   !trovato  &&  c <  a/3
```
is equivalent to
```
((a+2) == (3*b)) || ((!trovato) && (c < (a/3)))
```

Exercises

Exercise 4.1. Write a method:

```
public static double convertLireEuro(int x)
```

that, given an amount `x` in Lire, returns the corresponding amount in Euro.

Exercise 4.2. Write a predicate:

```
public static boolean sumOverflow(byte x, byte y)
```

that returns `true` if `x+y` causes overflow, and `false` otherwise.

[Hint: Assign first `x` and `y` to two variables of type `short`. Then compute the sum using the two new variables, and return the result of an expression of type `boolean` that verifies whether the result of the sum can fit into a byte, i.e., is greater than or equal to -128 and smaller than or equal to 127.

Exercise 4.3. Write a program that reads from the keyboard two integer number and prints on the screen:

- their arithmetic mean (i.e., their sum divided by 2)
- their geometric mean (the square root of their product)
- the greater and the smaller of the two numbers.

[Hint: use the methods of the class `Math`. For example, to calculate the square root, use the method `Math.sqrt`.]

Exercise 4.4. Write a predicate with header

```
public static boolean even(long x)
```

that returns `true` if the number x is even, `false` otherwise.

[Hint: use the operator `%` for the rest of the integer division and the equality operator `==` to construct an expression of type `boolean`, and return the result of such an expression with `return`.]

Exercise 4.5. Correct the following fragment of a Java program in such a way that it compiles without errors. The correction should be made without changing the types of the variables.

```
short x = 22;
byte y = x;
System.out.println(y);
```

Exercise 4.6. Consider the following variable declarations:

```
byte b;
short s;
int i;
long l;
float f;
double d;
char c;
boolean b1, b2;
```

For each of the following expressions, say what is its type:

1. `b+10L`
2. `(b+i)*l`
3. `(b+i)*l+f`
4. `s/f + Math.sin(f)`
5. `c == 'b'`
6. `l+1.5f`
7. `i<10`
8. `b1 == (f >= 5.0)`
9. `b1 && !b2`

Exercise 4.7. Given the variable declarations of Exercise 4.6, say which of the following statements causes a compiler error:

1. `s = 65L;`
2. `f = i+100;`
3. `i = 2*b + l;`
4. `b1 = s;`
5. `b2 = s >= 57;`
6. `c = b;`
7. `c = 'b';`

Exercise 4.8. What does the following program print when it is executed?

```
public class Account {
  public int bal;
  public Account(int x) {
    bal = x;
  }
}

public class Exercise_4_8 {
  public static void method(int a, Account b) {
    a *= 2;
    b.bal *= 2;
  }
  public static void main(String[] args) {
    int c = 100;
    Account r = new Account(100);
    method(c,r);
    System.out.println(c + " " + r.bal);
  }
}
```

Exercise 4.9. Write a class `Product` to maintain information about goods of a certain product that are stored in a warehouse. Each object of type `Product` is characterized by the name of the product (fixed the moment the product is created) and by the number of pieces of the product that are stored in the warehouse (initially 0). Implement the methods `downLoad()` (to increment the number of stored pieces), `upLoad()` (to reduce the number of stored pieces), and `toString()` to return the information about a product (e.g., `"Okinawa lamp, 25 pieces"`).

Example of usage:

```
public class TestProduct {
  public static void main (String[] args) {
    Product lamp = new Product("Lamp 60 Watt");
    System.out.println("Before the loading: " + lamp);
    lamp.downLoad(1000);
    lamp.upLoad(100);
    System.out.println("After the loading: " + lamp);
  }
}
```

Exercise 4.10. Write a method with header

```
public static char lastCharacter(String s)
```

that returns the last character of the string `s` passed as a parameter.