Object Oriented Programming

Java

Part 4: Environment classes

The java.lang package

- The **java.lang package** is automatically imported:
  - Interfaces:
    - **`Cloneable`**
    - **`Runnable`**
  - Classes:
    - **`Class`** e **`Object`**
    - **`Boolean`**, **`Number`** (and subclasses), **`Character`**, **`Void`**
    - **`Math`**
    - **`Process`**, **`Thread`**, **`System`** e **`Runtime`**
    - **`String`** and **`StringBuffer`**
    - **`Throwable`** and **`Exception`** (and subclasses)

The `Object` class (1)

- The `Object` class is the root of the class hierarchy in Java.
  - Every class directly or indirectly extends `Object`.
  - A variable of type `Object` can refer to any object.

The `Object` class (2)

- **Methods of the `Object` class**:
  - `public int hashCode()`
    Returns a hash code for this object.
  - `public String toString()`
    Returns a textual description for this object.
  - `public boolean equals(Object obj)`
    Returns equality between this object and **`obj`**.

    Recall that:
    - **Identity between object**: two references to the same object.
    - **Equality between objects**: two objects with the same state (with the same field's values).

The `Object` class (3)

- Both the `equals` and `hashCode` should be overridden if one wants to provide a notion of equality different from the default implementation provided by the `Object` class.
  - **By default, `equals` implements identity between objects** (with distinct objects it returns `false`).

    Note: The `==` and `!=` operators test always for identity between objects. That is, if the programmer overrides the `equals` method it continues to be able to test for identity through `==` and `!=`.

  - **By default, two distinct objects usually return a different `hashCode`**.

The `Object` class (4)

- **If the `equals` method is overridden to implement equality between objects, then the `hashCode` method should also be overridden accordingly, that is, it should be overridden in such a way that two objects evaluated as equal by `equals` must return the same value from `hashCode`.**
  - The mechanism used by hashed collections relies on `equals` returning `true` when it finds a key of the same value in the table (for instance, `Hashtable` and `HashMap`); and the key is computed from `hashCode`.

The `Object` class (5)

- Usually classes override the `toString` method.
- The method has several uses:
  - Debugging.
  - Providing a textual description of the object.

Classe `Object` (5)

- **Methods of the `Object` class (cont)**:
  - **`protected void finalize()`**
    Invoked by the garbage collector when the object is ceased to be referred.

    Note: In Java, an object exists while it is being referred. The garbage collector reclaims the memory occupied by objects that are no longer being used by the program.

  - **`protected Object clone()`**
    `throws CloneNotSupportedException`
    Builds and returns a copy of the calling object, with exactly the same fields. However, if the class of the calling `Object` does not implements the `Cloneable` interface, then the `CloneNotSupportedException` is thrown.

    Note: For any `obj`, we have that `obj.clone() != obj;`

Primitive types (1)

- **Primitive data types**:
  - **boolean** 1-bit (`true` or `false`)
  - **char** 16-bit Unicode UTF-16 (unsigned)
  - **byte** 8-bit signed integer
  - **short** 16-bit signed integer
  - **int** 32-bit signed integer
  - **long** 64-bit signed integer
  - **float** 32-bit IEEE 754 floating point
  - **double** 64-bit IEEE 754 floating point

Primitive types (2)

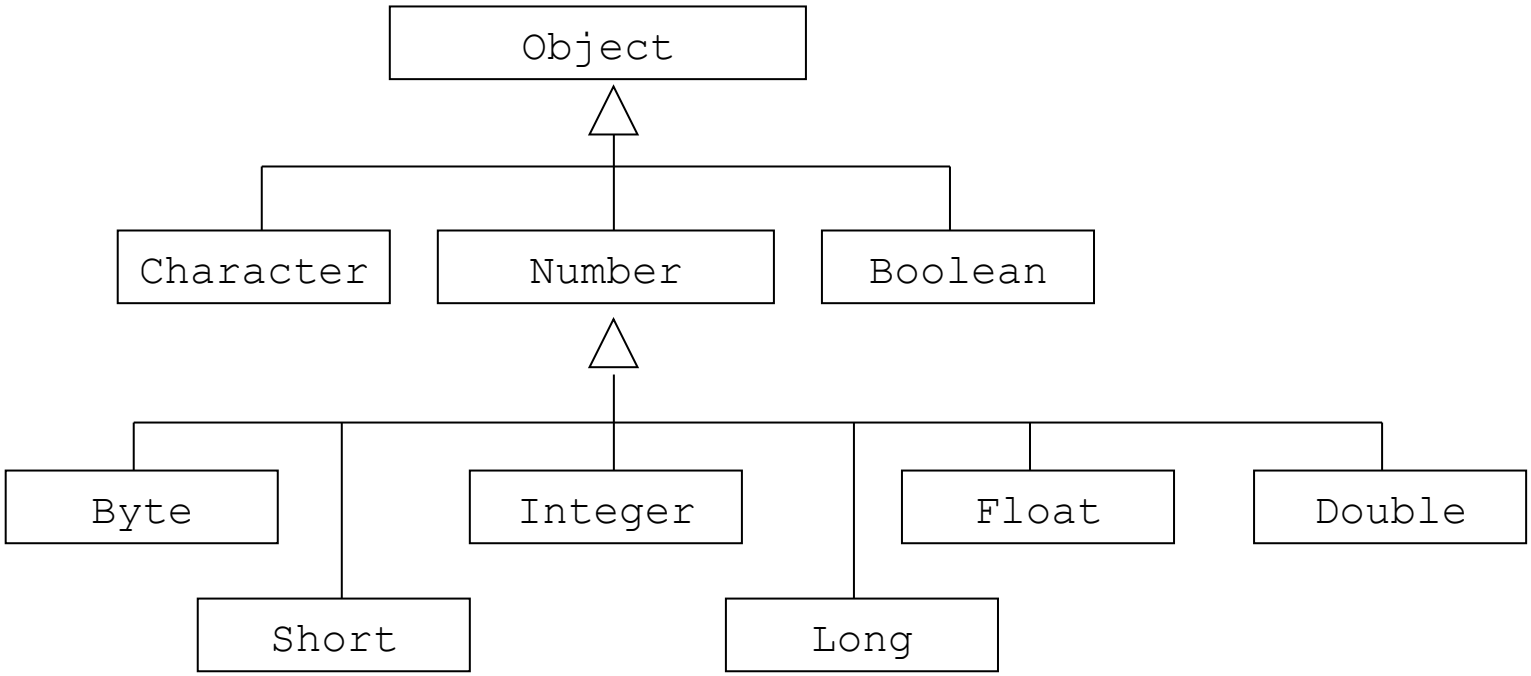
- In Java, for each primitive type there is in `java.lang` package a corresponding **wrapper class**.
- These wrapper classes, **`Boolean`**, **`Character`**, **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`** and **`Double`** provide a home for method and variables related to the type, e.g. string conversion and value range constants.
- Primitive data types:
  - Are fast and more convenient than reference types.
  - Occupy always the same space, independently from the machine where they are running on.

Primitive types (3)

- **Type casting**:
  - The java performs **implicit casting** of primitive types, in the order: `byte->short->int->long->float->double`.
  - An expression with distinct types results in a value of the superior type (for instance, `5+3.0` results in the value `8.0`).
  - When the implicit casting is not possible, usually an **explicit casting** is used (for instance, when casting a `float` into an `int` the fractional part is lost by rounding towards zero and `(int)-72.3` results in the value `-72`).

Wrapper classes (1)

- **Type hierarchy of wrapper classes**:



Wrapper classes (2)

- Instances of a given wrapper class contain a value of the corresponding primitive type.
- Each wrapper class defines an **immutable object** for the primitive value that is wrapping, that is, once the object is created the value represented by that object can never be created.
- The language provides **automatic conversion** between primitives and their wrappers:
  - **Boxing**: converts a primitive value to a wrapper object.
  - **Unboxing**: extracts a primitive type from a wrapper object.

```
Integer val = 3;
```

Only semantic needs to be known.

Wrapper classes (3)

- In the following, `Type` refers to the wrapper class, and `type` is the corresponding primitive type.
- **Each wrapper class, `Type`, has the following methods**:
  - **`public static Type valueOf(type t)`**
    returns an object of the specified `Type` with the value `t`.
  - **`public static Type valueOf(String str)`**
    returns an object of the specified `Type` with the value parsed from `str` (except for `Character` class).
  - **`public type typeValue()`**
    returns the primitive value corresponding to the current wrapper object.

```
Integer.valueOf(6).intValue();
```

Only semantic needs to be known.

Wrapper classes (4)

- `public static type parseType(String str)`
  converts the string `str` to a value of the specified primitive `type`.
- `public static String toString(type val)`
  returns a string representation of the given primitive value.

- **All wrapper classes, with the exception of `Boolean`, define three constant fields**:
  - `public static final type MIN_VALUE`
    the minimum value the `type` can have.
  - `public static fine type MAX_VALUE`
    the maximum value the `type` can have.
  - `public static final int SIZE`
    the number of bits used to represent the `type`.

Only semantic needs to be known.

Wrapper classes (5)

Primitive type	Wrapper class	Getting the value
boolean	Boolean	booleanValue()
char	Character	charValue()
byte	Byte	byteValue()
short	Short	shortValue()
int	Integer	intValue()
long	Long	longValue()
float	Float	floatValue()
double	Double	doubleValue()

Note:
Not all names match!

```
Integer I = new Integer(3);
int i = I.intValue(); // i passa a ter 3
String srt_i = Integer.toString(i); // str_i has value "3"
```

Only semantic needs to be known.

`Character` class (1)

Static methods	Description
`boolean isLowerCase(char)`	Is the char a lowercase letter?
`boolean isUpperCase(char)`	Is the char an uppercase letter?
`boolean isDigit(char)`	Is the char a digit?
`boolean isSpace(char)`	Is it a `'\t'`,`'\n'`,`'\f'` or `' '`?
`char toLowerCase(char)`	Converts the char to lowercase letter.
`char toUpperCase(char)`	Converts the char to uppercase letter.
`int digit(char,int)`	Returns the numeric value of char digit in the specified radix.

```
char c = Character.toUpperCase('g');
```

Only semantic needs to be known.

`Character` class (2)

- **The `Character` class also has the following method**:
  - **`public static int getType(char)`**
    returns the `char`'s Unicode type - the return value is one of the following constants:
    - **`CURRENCY_SYMBOL`**
    - **`LOWERCASE_LETTER`**
    - **`UPPERCASE_LETTER`**
    - **`MATH_SYMBOL`**
    - **`SPACE_SEPARATOR`**
    - **`...`**

Only semantic needs to be known.

`Byte`, `Short`, `Integer` and `Float` classes

- **`Byte`, `Short`, `Integer` and `Float` classes also have the following method**:
  - `public static type parseType(String str, int radix)`
    converts `str` to a primitive type value of type `type` in the specified `radix` (decimal, by default).

Operators (1)

- **Binary Arithmetic operators**:
  - `+` addition
  - `-` Subtraction
  - `*` multiplication
  - `/` division
  - `%` remainder

- The arithmetic operators can operate on any primitive numerical type (including chars, where the Unicode code is used).

Operators (2)

- **Increment/decrement operators**:
  - ++ increment
  - −− decrement

- The increment/decrement operators can be applied to any primitive numerical type (including chars, next/previous Unicode code).

Operators (3)

```
int var = 5;
```

Statement	Result	Value after the statement
`System.out.println(var++);`	5	6
`System.out.println(++var);`	6	6
`System.out.println(var--);`	5	4
`System.out.println(--var);`	4	4
`System.out.println(var%3);`	2	5

Operators (4)

- **Relational operators**:
  - `>` greater than
  - `>=` greater than or equal to
  - `<` less than
  - `<=` less than or equal to
- **Equality operators**:
  - `==` equal to
  - `!=` not equal to

- The relation and equality operators return a Boolean value, and they can be applied to primitive numerical types and chars.

Operators (5)

- The equality operators can be applied to any Boolean types.
- The equality operators can also be used to test identity between references:
  - In this case it refers to identity between objects, not equivalence:
    - **Identity:** `ref1==ref2` is `true` iff the two references refer to the same object (or if both are `null`).
    - **Equality:** `ref1.equals(ref2)` tests if the two references refer to (possibly distinct) objects with the same state (the same field values).
      - By default `equals` implements `==`.
      - If equality is needed, `equals` need to be overridden.

Operators (6)

- **Logical operators**:
  - `!` logical negation
  - `&` logical AND
  - `|` logical inclusive OR
  - `^` logical exclusive OR (XOR)
  - `&&` conditional AND (with lazzy evaluation)
  - `||` conditional OR (with lazzy evaluation)

- The logical operators combine Boolean expressions to yield Boolean values.

Operators (7)

- The `instanceof` operator evaluates whether a reference refers to an object that is an instance of a particular class or interface:
  - **Ref `instanceof` Ident**
    verify if the reference Ref is of type Ident.

Operators (8)

- **Bit manipulation operators**:
  - `&` bitwise AND
  - `|` bitwise inclusive OR
  - `^` bitwise exclusive OR (XOR)
  - `~` complement (toggles each bit in its operand)
  - `<<` shift bits left, filling with 0 bits on the right-hand side
  - `>>` shift bits right, preserving the sign
  - `>>>` shift bits right, filling with 0 bits the left-hand side
- The bitwise operators apply only to integer types (including char).

Operators (9)

- The **conditional operator ? :** provides a single expression that yields one of the two values based on a Boolean expression:
  - **Expr-Bool ? Expr1 : Expr2**
    if Expr-Bool is true then returns Expr1 else returns Expr2.

Operators (10)

- **Assignment operators**:

  = simple assignment

  op= composed assignment

- The left-operand must be a variable. The righ-operand is an expression.
- The op operator might be any arithmetic, logic or bitwise operator.

Operators (11)

- **String concatenation operator**: **+**

```
String s1 = "boo";
String s2 = s1+"hoo";
s2 += "!";
System.out.println(s2);
```

- The **`new`** **operator** creates an instance of a class or array.

Operators (12)

- **Operator precedence**:

1. Unary operators	`++ -- + - ~ !`
2. New or cast	`new (type)`
3. Multiplications	`* / %`
4. Additions	`+ -`
5. Shift	`<< >> >>>`
6. Relational	`< > >= <= instanceof`
7. Equality	`== !=`
8. AND	`&`
9. Exclusive OR	`^`
10. OR	`|`
11. Conditional AND	`&&`
12. Conditional OR	`||`
13. Conditional	`?:`
14. Assignment	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

`++x>3&&!b`
is equivalent to
`((++x)>3)&&(!b)`

Operators (13)

- When two operators have the same precedence, the **operator associativity** determines the order of the operator evaluation.
  - **Left associative**: `expr1 op expr2 op expr3` is equivalent to `(expr1 op expr2) op expr3`.
  - **Right associative**: `expr1 op expr2 op expr3` is equivalent to `expr1 op (expr2 op expr3)`.
- Assignment operator is right associative. All other binary operators are left associative.
- The conditional operator is right associative.

Arrays (1)

- An **array** is an object containing a fixed number of elements, all from the same base type.
  - Arrays are objects themselves so they extend `Object`.
  - The base type can be a primitive or a reference type (including a references to other arrays).
  - J2SE also makes available unlimited collections, for instance, `Vector`, `Stack`, ...

Arrays (2)

<u>Syntax</u>

Base_type[] Ident = new Base_type [length]

- Array dimension is omitted in type declaration, the number of components in an array is determined when it is created using `new`.
- An array length is fixed at its creation and cannot be changed.
- The square brackets in type declaration may appear after the variable name Ident instead or after the type Base_type:
  - Base_type [] Ident or Base_type Ident []

Arrays (3)

- **Multidimensional arrays**:
  - Declared with several [ ].
  - The first (left-most) dimension of an array must be specified when the array is created.
  - Specifying more than the first dimension is a shorthand for a nested set of `new` statements.

```
float[][] mat = new float[4][4];
```

```
float[][] mat = new float[4][];
for (int i=0; i < mat.length; i++)
   mat[i] = new float[4];
```

Arrays (4)

- In a multidimensional array, each nested array can have a different size, and so we can have:
  - Triangular arrays
  - Rectangular arrays
  - …

Arrays (5)

- The length of an array is available from its `length` field
  `public final int length`
- The access to an element is done by using the name of the array and the index enclosed between [and]: Ident[pos].
- The first element of the array has index 0, and the last element of the array has index `length-1`.
- The access to indexes outside the proper range throws an exception `ArrayIndexOutOfBoundException`.

```
int[] ia = new int[3];
... //inicialização de ia
for (int i=0; i < ia.length; i++)
   System.out.println(i + ": " + ia[i]);
```

Arrays (6)

- An array might have dimension 0, and so it is said to be an **empty array**.
  - There is a big difference between a `null` array reference and a reference to an empty array.
  - Useful in methods' return.
- The usual modifiers can be applied to array variables and fields.
  - The modifiers apply only to the array reference and not to its elements.
  - An array variable that is declared **final** means that the array reference cannot be changed after initialization; it does not mean that array elements cannot be changed!

Arrays (7)

- **Array initialization**:
  - When an array is created, each element is set to the default initial value of the type (zero for numeric types, false for `boolean`, `null` for references, etc).
  - Arrays can be initialized in two different ways:
    1. With comma separated values inside braces:
       - There is no need to explicitly create the array using `new`.
       - The length of the array is determined by the number of initialization values given.

```
String[] animals = {"Lion", "Tiger", "Bear"};
```

Arrays (8)

- The `new` operator can be used explicitly, but in that case the dimension must be omitted (because, again, the length of the array is determined by the number of initialization values given).

```
String[] animals = new String[]{"Lion", "Tiger", "Bear"};
```

- The last value of the initialization list is allowed to have a comma after it.
- Multidimensional arrays can be initialized by nesting array initializers.

Arrays (8)

2. By direct assignment of its values:

```
String[] animals = new String[3];
animals[0] = “Lion”;
animals[1] = “Tiger”;
animals[2] = “Bear”;
```

Only semantic needs to be known.

Arrays (9)

- **The `System` class offers a method that copies the values of an array into another**:
  - `public static void arraycopy (Object src, int srcPos, Object dst, int dstPos, int count)`

    copies the content of `src` array, starting at `src[srcPos]`, to the destination array `dst`, starting at `dst[dstPos]`; exactly `count` elements will be copied.

Arrays (10)

- **Arrays as extension of `Object` class**:
  - Arrays do not introduce new methods, they only inherit methods from `Object`.
    - The **`equals`** method is always based on identity and not in equality.
  - The **`deepEquals`** method from the utility class **`java.util.Arrays`** allows to compare fro equality between arrays.
    - Checks for equivalence between two `Object[]` recursively, taking into account the equivalence of nested arrays.

Arrays (11)

```
String[] animals = {"Lion", "Tiger", ”Bear", };
String[] aux = new String[animals.length];
System.arraycopy(animals, 0, aux, 0, animals.length);

for (int i=0; i<aux.length; i++)
   System.out.println(i + ": " + aux[i]);

System.out.println(aux.equals(animals));
System.out.println(java.util.Arrays.deepEquals(aux, animals));
```

In the terminal is printed

```
0: Lion
1: Tiger
2: Bear
false
true
```

Arrays (12)

```
int[][] pascalTriangle1 = {
   { 1 },{ 1, 1 },{ 1, 2, 1 },{ 1, 3, 3, 1},{ 1, 4, 6, 4, 1}};

int[][] pascalTriangle2 = new int[5][];
pascalTriangle2[0] = new int[]{ 1 };
pascalTriangle2[1] = new int[]{ 1, 1 };
pascalTriangle2[2] = new int[]{ 1, 2, 1 };
pascalTriangle2[3] = new int[]{ 1, 3, 3, 1 };
pascalTriangle2[4] = new int[]{ 1, 4, 6, 4, 1 };

System.out.println(pascalTriangle1.equals(pascalTriangle2));
System.out.println(java.util.Arrays.deepEquals(
   pascalTriangle1, pascalTriangle2));
```

In the terminal is printed

```
false
true
```

`String` class (1)

- A **string** is an object that represents character strings.
  - Character strings are instances of the **`String`** class.
  - Strings are constant, their values cannot be changed after they are created.
    - If mutable string are required, use **`StringBuffer`** class.
  - A string is delimited by quotation marks (" and ").
  - The + operator concatenates two strings.

`String` class (2)

- **Building strings**:
  - Implicitly, by the use of a literal, or by operators such as `+` and `+=` on two objects of type `String`.
  - Explicitly, by the `new` operator (only some constructors, see Java documentation):
    - `public String()`
      Allocates a newly creates String that represents the empty char sequence ("").
    - `public String(String valor)`
      Copy constructor.

`String` class (3)

- **`public String (char[] value)`**
  Allocates a new `String` that represents the specified char array.
- **`public String(char[] valor, int offset, int count)`**
  Allocates a new `String` that represents a subarray of the specified char array. The offset is the index of the first char of the subarray. The count specifies the length of the subarray.

`String` class (4)

```
String s1 = "Bom";
String s2 = s1 + " dia";
String vazia = "";
```

```
String vazia = new String();
String s1 = new String("Bom dia");
char valor[] = {'B','o','m',' ','d','i','a'};
String s2 = new String(valor);
String s3 = new String(valor,4,3);
```

Only semantic needs to be known.

`String` class (5)

- **Public methods from `String`:**
  1. String properties:
  - **`int length()`**
    Length of this string.
  - **`int compareTo(String str)`**
    Compares the this string with the specified string lexicographically. The comparison is based on the Unicode value of each char in the strings. Returns a negative integer if this string lexicographically precedes the specified string; a positive integer if this string lexicographically follows the specified string; and the result is 0 if the strings are equal.

Only semantic needs to be known.

`String` class (6)

2. Examining individual chars:

- **`char charAt(int)`**
  Char at the specified index.
- **`char[] toCharArray()`**
  Returns this string as a char array.
- **`int indexOf(char)`**
  First index where the specified char occurs in this string.
- **`int lastIndexOf(char)`**
  Last index where the specified char occurs in this string.
- **`String substring(int,int)`**
  Substring of this string between specified positions.
- **`String substring(int)`**
  Substring from specified position until the end of this string.

The first char of the string is in index 0.

`String` class (7)

3. Usual string operations:

- **`String replace(char oldChar,char newChar)`**
  Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
- **`String toLowerCase()`**
  Converts all the chars in this string to lower case.
- **`String toUpperCase()`**
  Converts all the chars in this string to upper case.
- **`String trim()`**
  Returns a copy of the string, with leading and trailing whitespaces omitted.
- **`String concat(String)`**
  Concatenates the specified string to the end of this string.

`String` class (8)

```
String s = "/home/asmc/oop-lecture.ppt";
...
int inicio, fim;
inicio = s.lastIndexOf('/');
fim = s.lastIndexOf('.');
System.out.println(s.substring(inicio+1,fim));
```

In the terminal is printed `oop-lecture`

Only semantic needs to be known.

`String` class (9)

- **Conversion between primitive type and `String`:**

Type	To String	From String
boolean	`String.valueOf(boolean)`	`Boolean.parseBoolean(String)`
int	`String.valueOf(int)`	`Integer.parseInt(String,int)`
long	`String.valueOf(long)`	`Long.parseLong(String,int)`
float	`String.valueOf(float)`	`Float.parseFloat(String)`
double	`String.valueOf(double)`	`Double.parseDouble(String)`

Only semantic needs to be known.

`String` class (10)

- **Conversion between char array and String**:
  - In the `String` class:
    - **`public char[] toCharArray()`**
  - In the `System` class:
    - **`public static void arraycopy(Object src, int srcPos, Object dst, int dstPos, int count)`**
      copies the content of `src` array, starting at `src[srcPos]`, to the destination array `dst`, starting at `dst[dstPos]`; exactly `count` elements will be copied.

`String` class (11)

```
public static String squeezeOut(String from, char toss){
   char chars[]=from.toCharArray(); // char array from String
   int len=chars.length;            // length of char array

   for (int i=0; i<len; i++) {
      if (chars[i]==toss) {
         --len; // final string has one less char
         System.arraycopy(
            chars, i+1, chars, i, len-i); // shift right end
          --i; // to continue searching in the same position
      }
   }
   return new String(chars, 0, len);
}
```

`System.out.println(squeezeOut("Programação por Objetos", 'o'));`

prints in the terminal `Prgramaçã pr Objets`

`String` class (12)

- **The `String` class overrides the `equals` method from `Object` to return `true` iff two strings have the same content.**
- **It also overrides `hashCode` to return a hash based on the contents of the `String` so that two strings with the same content have the same `hashCode`.**

```
String s1 = new String("abc"), s2 = "abc";
```

Expression	Result	Justification
`s1==s2`	`false`	Distinct objects
`s1.equals(s2)`	`true`	Same content

Only semantic needs to be known.

`Math` class (1)

- The Math class contains methods for performing basic numeric operations and mathematical constants.

Constant	Meaning
PI	π
E	e

```
System.out.println("Pi=" + Math.PI);
```

Only semantic needs to be known.

`Math` class (2)

- All methods are static (`num` is used for `int`, `long`, `float` or `double`)

Method	Description
`double sin(double)`	Sine
`double pow(double,double)`	1st arg raised to the power of the 2nd arg
`num abs(num)`	Absolute value
`num max(num,num)`	Maximum
`num min(num,num)`	Minimum
`int round(float)` `long round(double)`	Round to the nearest number
`double sqrt(double)`	Square root