Unit 2

Use of objects and variables

Summary

- Objects and classes
- The class `String`
- Method invocation
- Variables
- Assignment
- References to objects
- Creation of objects: invocation of constructors
- Immutable and mutable objects
- The class `StringBuffer`
- Input from keyboard

2.1 A simple Java program

Consider the following Java program:

```
public class MyMessage {
  public static void main(String[] args) {
    //What does this program do?
    /* Do you know it?
       It's easy. */
    System.out.println("JA");
    System.out.println("VA");
  }
}
```

Form of the program

- Single words in a Java program are separated by blanks. E.g., `public class`.
- Continuing on a new line has also the effect of separating two words (as if we were inserting a blank).
- We can use an arbitrary number of blanks or new lines to separate words (at least one).

*The **indentation** has no effect at all on the execution of the program. However, for readability of the code, it is very important to use proper indentation.*

Comments

It is possible to annotate the text of the program with **comments**. In Java we can make use of two types of comments:

- `//` denotes the start of a comment that spans only till the end of the line;
- `/*` ... `*/` delimits a comment that can span several lines.

Comments have no effect on the execution of the program. Again, they are used to improve the readability of the program code.

Other properties of Java

- `public class MyMessage`. All Java programs are collections of Java classes.
- `public static void main(String[] args)`. This is the standard `main` method, which indicates where the execution of the program should start.
- `System` is a predefined Java *class*.

- `System.out` is an object that is an *instance* of the predefined class `PrintStream`. This object is defined in the class `System` and its purpose is to handle the *system output* of the program (e.g., monitor output).
- `println(...)` is a *method* (operation) that is supported for the objects of type `PrintStream` (it is a property of the class `PrintStream`).
- `System.out.println("...")` is a *call of the method* `println("...")` on the *invocation object* `System.out`, and it has `"..."` as *parameter*.

Note: typically, Java adopts the convention that class names start with a capital letter, and method names with a lowercase letter. *You better respect this convention, if you want your programs to be understood.*

2.2 Method calls

Method call

Syntax:

object`.`*methodName*`(`*parameters*`)`

- *object* is (a reference denoting) the invocation object
- *methodName*`(...)` is the called method
- *parameters* are the parameters passed to the method

Semantics:

Calls a method on an object, possibly passing additional parameters. The effect of a method call is the execution of the operation associated to the method, and sometimes the return of a value.

Example:

```
System.out.println("Ciao!");
```

- `System.out` is (a reference denoting) the invocation object
- `println(...)` is the called method
- `"Ciao!"` is the parameter passed to the method

The call of the `println` method causes the string passed as parameter, i.e., `"Ciao!"`, to be printed on the standard output channel (the monitor), which is denoted by `System.out`. Notice that this method does not return any result, but performs an operation on the object (denoted by) `System.out`.

2.3 Difference between the methods `print` and `println`

```
public class MyMessage2 {
  public static void main(String[] args) {
    System.out.print("JA");
    System.out.print("VA");
    System.out.println();
  }
}
```

The `println("...")` method prints the string `"..."` and moves the cursor to a new line. The `print("...")` method instead prints just the string `"..."`, but does not move the cursor to a new line. Hence, subsequent printing instructions will print on the same line. The `println()` method can also be used without parameters, to position the cursor on the next line.

2.4 Objects and classes in Java

- The **objects** are the entities that can be manipulated by programs (typically, by invoking their methods).
- Each object belongs to a **class** (we say that *it is an instance of the class*) .
- A class is constituted by a set of objects that have the same properties: its **istances**.
- The class to which an object belongs to determines which **methods are available** for (i.e., can be applied correctly to) the object. For example:

```
System.out.print(); //OK
System.out.foo();   //ERROR
```

2.5 The class `String`

Let us analyze the predefined class `String`. The objects that are instances of this class denote **strings**, i.e., sequences of characters.

Expressions delimited by double quotes, e.g., `"java"`, denote objects of the class `String` (to be precise, these are predefined references to objects of the class `String`), and are called `String` **literals**. In general, literals represent **constants**, hence `String` literals represent constants of type string.

The class `String` provides various methods, including (the following table is taken from the Java documentation):

String	**concat**(String str) Concatenates the specified string to the end of this string.
int	**length**() Returns the length of this string.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.

Example:

```
public class MyMessage3 {
  public static void main(String[] args) {
    System.out.println("java".toUpperCase());
  }
}
```

The method `toUpperCase()` of the class `String` converts a string to upper case letters. In this case, `toUpperCase` is called on the object (denoted by the literal) `"java"`, and it returns a reference to a new object of the class `String` that denotes the string `"JAVA"`, and which is printed by the program.

2.6 Method signature and method header

The **signature** of a method consists of the name of the method and the description (i.e., type, number, and position) of its parameters.

Example:

- `toUpperCase()`
- `println(String s)`

Note: the names of the parameters have no relevance for the signature

The **header** of a method consists of the signature plus the description of (the *type* of) the result.

Example:

- `String toUpperCase()`
- `void println(String s)`

The keyword `void` denotes the fact that the method *does not return any result* (i.e., the method implements an action and not a function).

Two methods of the same class can have the same name, provided they have different signatures. Methods with the same name (but with different signatures) are said to be **overloaded**. For example, the `substring` method in the class `String`.

2.7 Parameters and results of a method

The **parameters** of a method represent the arguments that the calling block passes to the method, and that are necessary to realize the operation the method should perform.

Example: the method `println(String s)` can be called on the object `System.out` through the statement `System.out.println("ciao!")`. The parameter represented by the string `"ciao"` is used inside the method as the string to print on the output channel (the monitor), denoted by `System.out`.

If the method has to return a result (return type is different from `void`), such a result is computed by the method and returned to the calling block.

Example: consider the metodo `String concat(String s)`. If we call such a method on the string `"JA"` and pass it as parameter the string `"VA"`, it computes and returns the string `"JAVA"`.

2.8 Evaluation of a method call

To call a method we have to know the invocation object and its parameters.

Example: `"xxx".concat("yyy")`

- invocation object: `"xxx"`
- parameters: `"yyy"`

Once the invocation object and the parameters are known, we can execute the method and compute the result.

Example: `"xxx".concat("yyy")`

returns the string `"xxxyyy"`

In this case, the evaluation of the invocation object and of the parameters is immediate. In general, both the invocation object and the parameters passed as arguments may need to be obtained by first calling other methods.

2.9 Evaluation of the expression denoting the invocation object for a method

What does the following statement print?

```
System.out.println("xxx".concat("yyy").concat("zzz"));
```

To provide an answer, we have to understand how the expression `"xxx".concat("yyy").concat("zzz")` is evaluated.

1. It is possible to evaluate immediately the subexpression `"xxx".concat("yyy")`
   - `"xxx"` denotes the invocation object
   - `"yyy"` denotes the parameter
   - both are directly available, hence we can compute `"xxx".concat("yyy")`, which returns the string `"xxxyyy"`
2. After having evaluated `"xxx".concat("yyy")`, we can continue with `concat("zzz")`
   - `"xxxyyy"` denotes the invocation object
   - `"zzz"` denotes the parameter
   - both are now available, hence we can compute `"xxxyyy".concat("zzz")`, which returns the string `"xxxyyyzzz"`

Hence, the statement `System.out.println("xxx".concat("yyy").concat("zzz"));` prints `"xxxyyyzzz"`.

The evaluation of the expression that denotes the invocation object is done from left to right, by computing the invocation objects and calling on them the methods that still need to be evaluated.

2.10 Evaluation of the expressions denoting the parameters of a method

What does the following statement print?

```
System.out.println("xxx".concat("yyy".concat("zzz")));
```

To provide an answer, we have to understand how the expression `"xxx".concat("yyy".concat("zzz"))` is evaluated.

1. It is possible to evaluate immediately the subexpression `"yyy".concat("zzz")`
   - `"yyy"` denotes the invocation object
   - `"zzz"` denotes the parameter
   - both are directly available, hence we can compute `"yyy".concat("zzz")`, which returns the string `"yyyzzz"`
2. After having evaluated `"yyy".concat("zzz")`, we can continue with `"xxx".concat(...)`
   - `"xxx"` denotes the invocation object
   - `"yyyzzz"` denotes the parameter computed at item 1

- both are now available, hence we can compute `"xxx".concat("yyyzzz")`, which returns the string `"xxxyyyzzz"`

Hence, the statement `System.out.println("xxx".concat("yyy".concat("zzz")));` prints `"xxxyyyzzz"`.

The evaluation of the expressions denoting the parameters is done from the inside to the outside, computing each time the parameters of each call before the call is evaluated.

2.11 Static methods

Static methods are methods that do not require an invocation object. The syntax of the call of a static method is the following:

ClassName`.`*methodName*`(`*parameters*`)`

where:

- *ClassName* is the class to which the method belongs (i.e., the class where the method is defined)
- *methodName*`(...)` is the called method
- *parameters* are the parameters passed to the method

Calling a static method is similar to calling a non-static method, except that we do not have to specify the invocation object but just the parameters. Note that the name of the method is preceded by the name of the class to which the method belongs. This is because in Java methods have a name that is local to the class and hence, to be able to identify the method, it is necessary to specify the class of the method.

Example:

```
JOptionPane.showInputDialog("Insert a string")
```

is the call of the method `showInputDialog` defined in the predefined class `JOptionPane`. This method is passed the parameter `"Insert a string"` of type `String`. This method opens a dialog window which requires user input. Such a window displays the message "Insert a string" and an input fields where we can type in a string that the method will return. We will see this method in more detail later on.

Note: the `main` method of a class is a static method that has `void` as return type (i.e., it does not return anything) and has as parameter an array of `String` objects (see later).

2.12 Variables

Example:

```
public class Java1 {
  public static void main(String[] args) {
    System.out.println("java".toUpperCase());
    System.out.println("java".toUpperCase());
  }
}
```

Note that the expression `"java".toUpperCase()` is evaluated twice. To avoid this, we could store the result of the evaluation of this expression in a **variable** and reuse it for printing.

```
public class Java2 {
  public static void main(String[] args) {
    String line;
    line = "java".toUpperCase();
    System.out.println(line);
    System.out.println(line);
  }
}
```

In the `Java2` program, `line` is a variable of type `String` to which we assign the value of `"java".toUpperCase()`, which is then printed twice.

*A **variable** represents a memory location that can be used to store the reference to an object.*

2.13 Variables: main properties

Variables are used to denote the data inside a program.

A variable is characterized by the following **properties**:

1. **Name**: it is necessary in order to identify the variable; for example: `line`. Such a name must be a **Java identifier**, i.e.:
   - a sequence of letters, digits, or the `'_'` character, starting with a letter or with `'_'` (but not with a digit)
   - it can be of any length
   - lowercase and uppercase letters are considered to be different
   - some identifiers, called **keywords**, are reserved: `class`, `public`, `if`, `while`, etc.
2. **Type**: specifies the type of the data that the variable can store; for example: a variable of type `String` can store a reference to a string.
3. **Address** of the memory location containing the stored data:
   - each variable has an associated memory location
   - the size of the memory location depends on the type of the variable
   - in Java there is no way to know the address of a memory location!!! This solves several problems related to security, such as attacks by viruses, etc.
4. **Value**: the data denoted by the variable at a certain point during the execution of the program; for example: the reference to the object `"JAVA"`.

During program execution, the name, type, and address of a variable cannot change, while the value can.

Note: even if a variable contains the reference to an object, often we abuse terminology and say that the value of the variable "is the object" which the variable refers to.

2.14 Variables and shoe-boxes

We can intuitively understand the meaning of variables by comparing them with labeled shoe-boxes in a cupboard:

1. name – label
2. type – form of the box (determines which type of shoes we can put in the box)
3. address – position in the cupboard (the fact that the location does not change means that the shoe-box is nailed to the cupboard)
4. value – shoe inside the box

2.15 Variable declarations

Variables are introduced in a program through **variable declarations**.

Variable declaration

Syntax:

type variableName`;`

- *`type`* is the *type* of the variable
  - for variables of type reference to an object it is the name of the class of which the object is an instance;
  - otherwise, it is a primitive predefined type – see Unit 4
- *`variableName`* is the name of the variable being declared

Semantics:

The declaration of a variable reserves a memory location for the variable and makes the variable available in the part of the program (block) where the declaration appears (more details in Unit 3). Notice that we have to declare a variable before we can use it.

Example:

```
String line;
```

- `String` is the type of the variable
- `line` is the name of the variable

After such a declaration, the variable `line` can be used in the program block in which the declaration appears (e.g., in the `main` method, if the declaration appears there).

We can also declare several variables of the same *`type`* with a single declaration.

```
type variableName-1,...,variableName-n;
```

Such a declaration is equivalent to:

```
type variableName-1;
 ...
type variableName-n;
```

2.16 Assignment

The assignment statement is used to store a value in a variable.

Assignment

Syntax:

```
variableName = expression;
```

- *`variableName`* is the name of a variable
- *`expression`* is an expression that, when evaluated, must return a value of the type of the variable

Semantics:

The variable *`variableName`* is assigned the value of the *`expression`* on the right hand side of the `=` symbol. Such a value can be (a reference to) an object, or some data of a different type (see later). After the assignment, the value of a variable stays unchanged till the next assignment.

Example:

```
line = "java";
```

- `line` is a variable of type `String`
- `"java"` is a (very simple) expression that returns a value of type `String`, specifically `"java"` itself

The result of the execution of the assignment is that afterward `line` denotes (the object denoted by) `"java"`.

Example: what is the meaning of the following statement?

```
s = s.concat("yyy");
```

The execution of the statement amounts to:

1. evaluate the expression on the right hand side: i.e., concatenate to the current value of `s` (e.g., `"xxx"`) the string `"yyy"`, obtaining as result `"xxxyyy"`
2. replace the current value of `s` (i.e., `"xxx"`) with the value that has just been computed (i.e., `"xxxyyy"`)

If the value of `s` before the execution of the assignment was `"xxx"`, after the assignment the value of `s` is `"xxxyyy"`.

Note: an assignment is different from an equality test (which we will see in Unit 4).

2.17 Initializing a variable

Initializing *a variable means specifying an initial value to assign to it (i.e., before it is used at all).*

Notice that a variable that is not initialized does not have a defined value, hence it cannot be used until it is assigned such a value. If the variable has been declared but not initialized, we can use an assignment statement to assign it a value.

Example: The following program contains a semantic error:

```
public class Java3 {
  public static void main(String[] args) {
    String line;
    System.out.println(line);
    System.out.println(line);
  }
}
```

The variable `line` is not initialized before we request to print it (the error is detected at compile time).

A variable can be initialized at the moment it is declared, through the statement:

type variableName = expression;

In Java, the above statement is equivalent to:

type variableName;
variableName = expression;

Example:

```
String line = "java".toUpperCase();
```

is equivalent to

```
String line;
line = "java".toUpperCase();
```

2.18 Object references

In Java, variables cannot contain objects, but only references to objects.

The objects are constructed and allocated in memory independently from the declarations of variables. Specifically:

- objects denoted by a literal (such as literals of type `String`, e.g., `"foo"`, `"ciao"`, etc.) are allocated in memory at compilation time;
- all other objects must be constructed and allocated through an explicit statement (see later).

A variable whose type is a class contains a reference to an object of the class (i.e., the address of the memory location where the object is allocated).

Example:

```
String s;
s = "xxx";
```

The first statement declares a variable `s` of type `String`. Such a variable is not initialized yet. The second statement assign to such a variable the reference to the object denoted by `"xxx"`.

Notice that two variables may contain a reference to the same object.

Example:

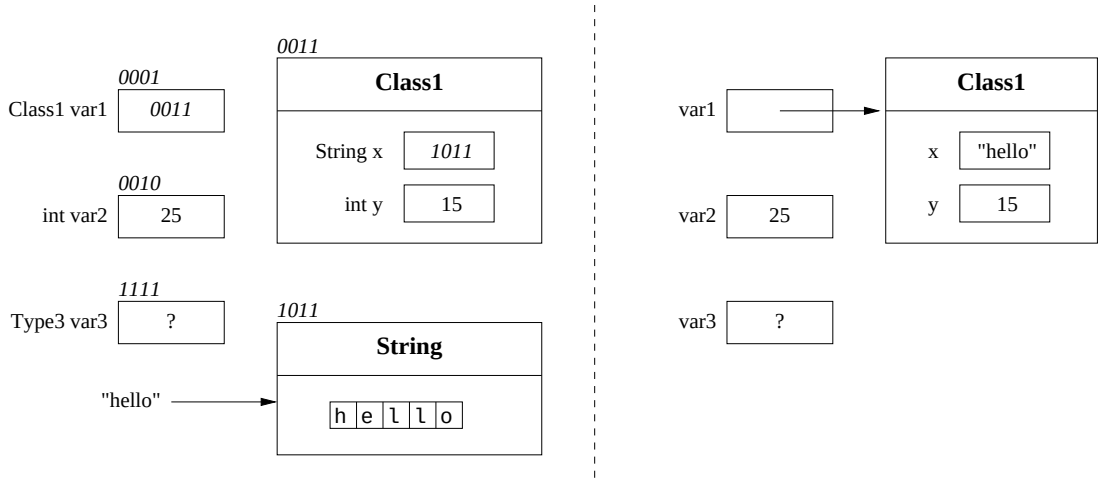
```
String s, t;
s = "xxx";
t = s;
```

After these two statement, both `t` and `s` contain a reference to the object denoted by `"xxx"`.

Variables of type object reference may have also a special value, namely `null`. Such a value means that the variable does not denote any object. Do not confuse variables whose value is `null` with variables that are not initialized. A variable that is not initialized does not have any value, not even `null`.

2.19 Graphical notation for representing variables

A variable is a reference to a memory location in which an object is stored. To represent variables and their values we use the following graphical notation.



The diagram represents the name, type, address, and value of a variables. The value is typically represented by an arrow pointing to the referenced object. For an object it is represented the class of which the object is an instance, and the state, i.e., the values of its properties. The diagram on the left shows the actual addresses of memory locations, to clarify the notion of object reference. In fact, in Java the addresses of objects are never represented explicitly, i.e., one is not interested which is the memory location that is actually used to store an object (see diagram on the right). Often, we will also omit the type of a variable, since it typically coincides with the type of the referenced object (note that this is not always the case, due to inheritance – see Unit 4).

Example: the situation in memory after the execution of the statements

```
String s,t,u;
s = "xxx";
t = "yyy";
u = t;
```

is shown in the following diagram.

2.20 Example: program on strings

```
public class Hamburger {
  public static void main(String[] args) {
    String s,t,u,v,z;
    s = "ham";
    t = "burger";
    u = s.concat(t);
    v = u.substring(0,3);
    z = u.substring(3);
    System.out.println("s = ".concat(s));
    System.out.println("t = ".concat(t));
    System.out.println("u = ".concat(u));
    System.out.println("v = ".concat(v));
    System.out.println("z = ".concat(z));
  }
}
```

2.21 Example: string concatenation

Construction of the string `"xxxyyyzzz"`, using variables and assignments:

```
String x = "xxx", y = "yyy", z = "zzz";
String temp = x.concat(y);
String result = temp.concat(z);
System.out.println(result);
```

We use one variable for each object and for each intermediate result.

2.22 Example: initials of a name

```
public class JFK {
  public static void main(String[] args) {
    String  first = "John";
    String  middle = "Fitzgerald";
    String  last = "Kennedy";
    String  initials;
    String  firstInit, middleInit, lastInit;
    firstInit = first.substring(0,1);
    middleInit = middle.substring(0,1);
    lastInit = last.substring(0,1);
    initials = firstInit.concat(middleInit);
    initials = initials.concat(lastInit);
    System.out.println(initials);
  }
}

// or simply
public class JFK2 {
  public static void main(String[] args) {
    String  first = "John";
    String  middle = "Fitzgerald";
    String  last = "Kennedy";
    System.out.println(first.substring(0,1).
                            concat(middle.substring(0,1)).
                            concat(last.substring(0,1)));
  }
}
```

2.23 Use of "+" for string concanenation

String concatenation is so common that Java provides a specific abbreviation for using the method `concat`. Specifically, the expression:

```
"xxx".concat("yyy")
```

can be rewritten as:

```
"xxx" + "yyy"
```

Example:

```
public class JFK {
  public static void main(String[] args) {
    String  first = "John";
    String  middle = "Fitzgerald";
    String  last = "Kennedy";
    String  initials;
    String  firstInit, middleInit, lastInit;
    firstInit = first.substring(0,1);
    middleInit = middle.substring(0,1);
    lastInit = last.substring(0,1);
    initials = firstInit + middleInit + lastInit;
    System.out.println(initials);
  }
```

```
}

// or simply
public class JFK2 {
  public static void main(String[] args) {
    String  first = "John";
    String  middle = "Fitzgerald";
    String  last = "Kennedy";
    System.out.println(first.substring(0,1) +
                       middle.substring(0,1) +
                       last.substring(0,1));
  }
}
```

2.24 Invocation of a constructor

The creation of new object is done by calling special methods called **constructors**.

Invocation of a constructor

Syntax:

```
new className(parameters)
```

- `new` is a predefined operator
- *`className`*`(`*`parameters`*`)` is the signature of a special method called constructor. The name of a constructor coincides with the name of the class to which it belongs.

A certain class can have many constructors, which differ in the number and/or types of their parameters (overloading of constructors).

Semantics:

The invocation of a constructor constructs a new object of the class to which the constructor belong and returns a reference to the created object. The object is constructed by making use of the parameters passed to the constructor.

Example:

```
new String("test")
```

- `new` is the predefined operator
- `String(String s)` is a constructor of the class `String`

The expression constructs a new object of the class `String` that is equal to the string denoted by `"test"`. The reference to such an object is returned by the expression.

Example:

```
public class Hello {
  public static void main(String[] args) {
    String s = new String("hello world");
    System.out.println(s);
  }
}
```

Note that:

- the invocation of the constructor `new String("hello world")` creates a new object that is an instance of `String` and that represents the string `"hello world"`
- the reference to such an object is assigned to the variable `s`
- the value of the object denoted by `s` (i.e., "`hello world`") is printed.

2.25 Empty string

The empty string represents the sequence of characters of length 0, and can be denoted by the literal `""`.

The class `String` has a constructor without parameters that creates an empty string:

```
String emptystring = new String();
```

The other constructor that we have seen for strings takes a string as parameter. hence, the two constructors have different signatures. This is a case of *overloading*.

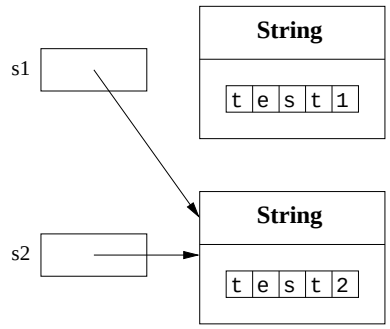
Note: do not confuse the empty string with value `null`.

2.26 Accessibility of objects

Consider the following statements:

```
String s1 = new String ("test1");
String s2 = new String ("test2");
s1 = s2;
```

The references to `s1` and `s2` are initially two references to two newly created objects. The assignment statement sets the reference of `s1` equal to the reference of `s2` (two references to the same object `"test2"`), while the reference to the object `"test1"` created by the first statement is lost.



The operation of recovering the memory used by objects that have been "lost" by the program is called **garbage collection**. In Java, such an operation is performed automatically by the runtime system (i.e., the Java Virtual Machine).

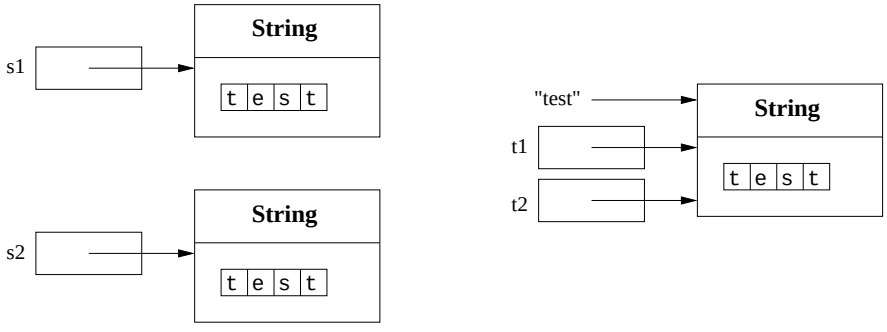
2.27 References to objects

The `new` operator creates new instances of objects.

Example:

```
String s1 = new String ("test");
String s2 = new String ("test");
String t1 = "test";
String t2 = "test";
```

The references `s1` and `s2` are references to different objects, while `t1` and `t2` are references to the same object.



2.28 Immutable objects

Objects of type `String` are *immutable* objects because they have no way (no methods) to change their state, i.e., the string they represent.

Objects that cannot change their state are called ***immutable****. They represent exactly the same information for all their lifetime.*

Example:

```
public class UpperLowerCase {
  public static void main(String[] args) {
    String s, upper, lower;
    s = new String("Hello");
    upper = s.toUpperCase();
    lower = s.toLowerCase();
    System.out.println(s);
    System.out.print("upper = ");
    System.out.println(upper);
    System.out.print("lower = ");
    System.out.println(lower);
  }
}
```

Notice that this program constructs 3 different strings (that are not modified anymore):

- the string `"Hello"`, by calling the constructor,
- the string `"HELLO"`, denoted by the variable `upper`, and
- the string `"hello"`, denoted by the variable `lower`.

2.29 Mutable objects: the class `StringBuffer`

Java has also a class that is very similar to `String` but whose instances are *mutable* objects: the class `StringBuffer`.

Specifically, the class `StringBuffer` has methods to modify the string represented by an object .

StringBuffer() Constructs a string buffer with no characters in it and an initial capacity of 16 characters.
StringBuffer(String str) Constructs a string buffer so that it represents the same sequence of characters as the string argument; in other words, the initial contents of the string buffer is a copy of the argument string.

StringBuffer	**append**(String str) Appends the string to this string buffer.
StringBuffer	**insert**(int offset, String str) Inserts the string into this string buffer.
StringBuffer	**replace**(int start, int end, String str) Replaces the characters in a substring of this StringBuffer with characters in the specified String.
String	**toString**() Converts to a string representing the data in this string buffer.

2.30 Mutable objects: methods with side-effect

Mutable objects must such that it is possible to modify their state. Such modifications are called **side-effects**. Methods that perform such modifications are called methods with side-effect.

Example:

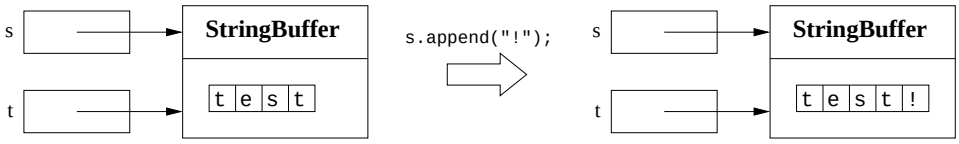
```
public class SideEffect1 {
  public static void main (String[] args) {
    StringBuffer s = new StringBuffer("test");
    StringBuffer t;
    t = s;
    s.append("!");
    System.out.println(s.toString());
    System.out.println(t.toString());
```

```
  }
}
```

Considerations:

- The execution of this program prints `"test!"` twice.
- Notice how `append` has a side-effect on the invocation object. The object stored in (more precisely, referenced by) `s` is modified by the execution of the `append` method.
- Since after the execution of the statement `t = s;`, we have that `s` and `t` reference the same object, also `t` denotes an object that represents `"test!"`.

Note: in general, the `append` methods returns a reference to the modified invocation object, however, in the `s.append("!");` statement, such a reference is not used.



2.31 Example: initials of a name using the `StringBuffer` class

```
public class SideEffect2 {
  public static void main (String[] args) {
    String s = "name surname";
    StringBuffer sbuf = new StringBuffer(s);
    sbuf.replace(0,1,s.substring(0,1).toUpperCase());
    sbuf.replace(5,6,s.substring(5,6).toUpperCase());
    System.out.println(sbuf.toString());
  }
}
```

2.32 Keyboard input

In Java there are many ways to read strings from input. The simplest one is to use the predefined method `showInputDialog`, which is defined in the class `JOptionPane`, which in turn is part of the `swing` library. Using such a method, we can read input from the keyboard according to the following schema:

```
import javax.swing.JOptionPane;

public class KeyboardInput {
  public static void main (String[] args) {
    ...
    String inputString = JOptionPane.showInputDialog("Insert a string");
    ...
    System.out.println(inputString);
    ...
    System.exit(0);
  }
}
```

- `import javax.swing.JOptionPane;` – imports the class `JOptionPane` from the library `javax.swing`
- `String inputString = JOptionPane.showInputDialog("Insert a string");`
  1. creates a dialog window (see figure) showing the message "`Insert a string`",
  2. reads a string from the keyboard,
  3. returns (a reference to) such a string, and
  4. assigns the reference to the variable `inputString`.
- `System.exit(0);` must be added to the `main` method when the predefined library class `JOptionPane` is used – this is necessary, since dialog windows are not handled directly by `main`, and hence we have to provide an explicit command to terminate them.

2.33 Example: initials of a name read from input

```
import javax.swing.JOptionPane;

public class Initials {
  public static void main (String[] args) {
    String fn = JOptionPane.showInputDialog("Insert first name");
    String ln = JOptionPane.showInputDialog("Insert surname");
    String ifn = fn.substring(0,1).toUpperCase();
    String iln = ln.substring(0,1).toUpperCase();
    System.out.println("Name: " + fn + " " + ln);
    System.out.println("Initials: " + ifn + iln);
    System.exit(0);
  }
}
```

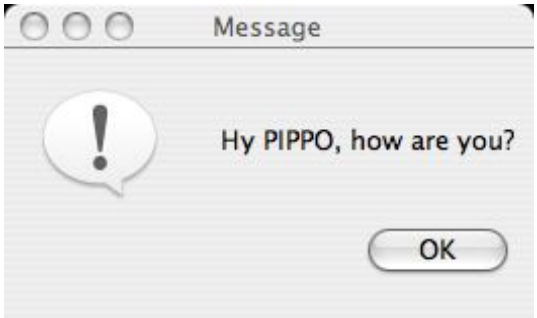
2.34 Output on a window

Using the class `JOptionPane` it is also possible to send output to a dialog window. Specifically, the method `showMessageDialog` can be used. The following program illustrates its use:

```
import javax.swing.JOptionPane;

public class OutputWindow {
  public static void main(String[] args) {
    String name = JOptionPane.showInputDialog("What is your name?");
    name = name.toUpperCase();
    String stringToShow = "Hy " + name + ", how are you?";
    JOptionPane.showMessageDialog(null, stringToShow);
    System.exit(0);
  }
}
```

- `JOptionPane.showMessageDialog(null,stringToShow);` creates a dialog window (see Figure) that shows the string denoted by the `stringToShow` variable; the value `null` for the first parameter indicates that the window that has to be created is not a sub-window of an existing window.



Exercises

Exercise 2.1. Write a Java program that creates an object of type `String` representing your name, and prints the first and the last characters of the string.

Exercise 2.2. Illustrate by means of a diagram what happens in memory when the program of Exercise 2.1 is executed.

Exercise 2.3. Modify the Java program in Exercise 2.1 using as few variables as possible.

Exercise 2.4. Write a Java program that reads from input a non-empty string and prints the string obtained from it by inverting the first and the last characters. Illustrate by means of a diagram what happens in memory when the program is executed.

Exercise 2.5. Solve Exercise 2.4 using the class `StringBuffer`. Illustrate by means of a diagram what happens in memory when the program is executed.