UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica etsinf

Unit 2. Primitive and reference variables

Introduction to Computer Science and Computer Programming
Introducción a la Informática y la Programación (IIP)

Year 2017/2018

Departamento de Sistemas Informáticos y Computación

Contents

1 Basic concepts: classes and programs ▷ 3

2 Edition, compilation, and execution in Java ▷ 8

3 Datatypes and variables ▷ 14

4 Primitive datatypes ▷ 27

5 Reference variables ▷ 47

6 Basic classes: `String` and `Scanner` ▷ 63

7 Utility classes: `Math` ▷ 73

8 Basic input/output ▷ 78

Contents

- 1 *Basic concepts: classes and programs* ▷ 3
- 2 Edition, compilation, and execution in Java ▷ 8
- 3 Datatypes and variables ▷ 14
- 4 Primitive datatypes ▷ 27
- 5 Reference variables ▷ 47
- 6 Basic classes: `String` and `Scanner` ▷ 63
- 7 Utility classes: `Math` ▷ 73
- 8 Basic input/output ▷ 78

Basic concepts: classes and programs

- In the Object Oriented Programming (OOP) paradigm, applications are organised in ***objects***
- ***Object***: group or collection of ***data*** and ***operations*** with a given structure and that model relevant aspects of a problem.
- Objects that share a behaviour can be grouped in different categories named ***classes***.
- ***Class***: describes the behaviour of each object (the object is an ***instance*** of the class).
- Java is an Object-Oriented Language (OOL): programming in Java is ***writing the classes and using them to create objects*** to properly solve the problem.

Basic concepts: classes and programs

- Predefined and programmer-defined classes.
- Three different types of classes:
  - ***Datatype classes***: objects (Unit 4)
  - ***Program classes***: executables
  - ***Utility classes***: operations (Unit 4)

  At this moment, we will only work with ***program classes***

Basic concepts: classes and programs

Program classes have a `main` method

Example: Hello World!

```
/* First class example:  the
   classic 'Hello world' message */

public class HelloWorld {
  public static void main (String [] args) {
    System.out.println("Hello world!"); // Shows on screen
  }
}
```

- /* and */ begin and end many-line comments
- // begins a comment till the end of the line

Basic concepts: classes and programs

Instruction blocks

- The Java language structure is ***block oriented***
- Instructions appear sequentially, separated by ;
- A ***block*** is situated between braces ({, })
- A block is composed of a sequence of one or more instructions

```
{
    System.out.println("Hello!");
    System.out.println("World!!!");
}
```

- Blocks can be declared and nested inside other blocks: ***external*** or ***internal*** blocks

Contents

1 Basic concepts: classes and programs ▷ 3

○ 2 *Edition, compilation, and execution in Java* ▷ 8

3 Datatypes and variables ▷ 14

4 Primitive datatypes ▷ 27

5 Reference variables ▷ 47

6 Basic classes: `String` and `Scanner` ▷ 63

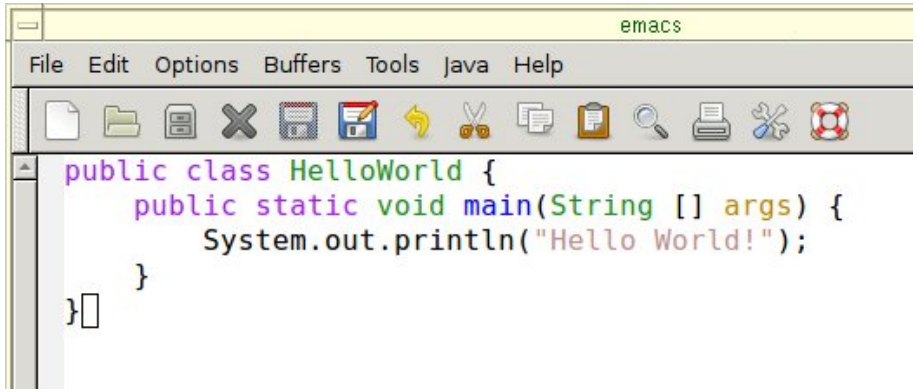
7 Utility classes: `Math` ▷ 73

8 Basic input/output ▷ 78

Edition, compilation, and execution in Java

Steps in program development:

1. ***Edition***: writing the code



2. ***Compilation***: convert into *bytecodes*

```
:/tmp$ javac HelloWorld.java
:/tmp$ ls HelloWorld.*
HelloWorld.class  HelloWorld.java
```

3. ***Execution***: make it work

```
:/tmp$ java HelloWorld
Hello World!
```

Edition, compilation, and execution in Java

Programming errors

Programming errors avoid program execution or cause incorrect program behaviour:

- Compilation errors: the program does not accomplish all the features of the definition of the language (usually easy to solve thanks to compiler errors)
- Execution errors: cause the program malfunction (usually more difficult to correct)
  - Runtime errors: stop the execution
  - Logical errors: cause results that are not correct

Edition, compilation, and execution in Java

Compilation errors

```
public class HelloWorld {
  plubic static void main (String [] args) {
    System.out.printl("Hello world!_);
  }
}
```

```
HelloWorld.java:2: <identifier> expected
    plubic static void main(String [] args) {
           ^
HelloWorld.java:3: unclosed string literal
        System.out.printl("Hello World!);
                          ^
HelloWorld.java:3: ':' expected
        System.out.printl("Hello World!);
                                        ^
HelloWorld.java:5: reached end of file while parsing
}
 ^
4 errors
```

Edition, compilation, and execution in Java

Runtime errors

```
public class Division {
  public static void main (String [] args) {
    System.out.println("Dividing 9 by 0:  " + 9/0);
  }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at Division.main(Division.java:3)
```

Edition, compilation, and execution in Java

Logical errors

```
public class CircleArea {
  public static void main (String [] args) {
    System.out.print("Area of circle of ");
    System.out.println("radius 3 = " + 2.14*3*3);
  }
}
```

```
Area of circle of radius 3 = 19.259999999999998
```

Contents

1 Basic concepts: classes and programs ▷ 3

2 Edition, compilation, and execution in Java ▷ 8

◦ 3 *Datatypes and variables* ▷ 14

4 Primitive datatypes ▷ 27

5 Reference variables ▷ 47

6 Basic classes: `String` and `Scanner` ▷ 63

7 Utility classes: `Math` ▷ 73

8 Basic input/output ▷ 78

Datatypes and variables

- ***Data***: information in a format usable by a computer
- A ***datatype*** defines:
  - A ***set of values***, and
  - The ***set of operations*** allowed on those values
- Datatypes in Java can be classified into:
  - ***Primitive datatypes*** (elemental or simple)
    `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`
  - ***Reference datatypes***: memory addresses that reference to an aggregation of data items
    * ***Predefined*** datatypes: `Scanner`, `String`, etc.
    * ***Programmer-defined*** datatypes: `Circle`, `BlackBoard`, etc.

Datatypes and variables

Identifiers

- Datatypes are applied on ***variables***, ***constants*** and ***methods***
- These elements and other elements (e.g., class names) are named by ***identifiers***
- ***Identifiers*** must begin with a letter and be followed by any combination of letters, numbers, and underscore (_) and dollar ($) symbols

Valid examples	radius, MAX_VALUE, data1
Invalid examples	5sphere, my value, a.var

- Java is ***case-sensitive***: capital letters are interpreted as different from lowercase letters (i.e., `data` is not the same identifier as `Data` or `DATA`)

Datatypes and variables

Identifiers

Reserved words have a specific meaning for the language and cannot be used as identifiers. The same occurs for `null`, `true`, and `false`

List of reserved words:

`abstract`	`continue`	`for`	`new`	`switch`
`assert`	`default`	`goto`	`package`	`synchronized`
`boolean`	`do`	`if`	`private`	`this`
`break`	`double`	`implements`	`protected`	`throw`
`byte`	`else`	`import`	`public`	`throws`
`case`	`enum`	`instanceof`	`return`	`transient`
`catch`	`extends`	`int`	`short`	`try`
`char`	`final`	`interface`	`static`	`void`
`class`	`finally`	`long`	`strictfp`	`volatile`
`const`	`float`	`native`	`super`	`while`

Grey reserved words must not be used

Datatypes and variables

Identifiers

- It is recommended to use identifiers that are good descriptors

```
name, toString, volume, initialAmount
```

- Variable's identifiers use to be lowercase; when it has several words, the initial word begins with lowercase and the rest by capital letters

```
sphereRadius, cubeVolume
```

- Constant's identifiers use to be in capital letters; when it has several words, they are separated by the underscore symbol

```
PI, MAX_STUDENTS
```

Datatypes and variables

Variables

- All data used by a program is represented by ***variables***
- Each ***variable*** is associated to a ***datatype***, which expresses:
  - The ***set of values*** that can be stored in the variable,
  - The ***set of operations*** that are allowed on the variable, and
  - The ***size*** of the memory zone that is using
- ***Variable declaration***: defines its ***identifier*** and ***datatype***
- Java is a ***strongly typed language***: any variable must be ***declared*** previously to its use

Datatypes and variables

Variables

- Declaration syntax:

```
type varname1, varname2, ...varnamen;
```

- Examples:

```
int var1, var2, sum;
char c;
double d1, d2;
```

- ***State of a variable***: its content in a given moment of the execution of the program
- ***State of a program***: state of all their variables in a given moment of the execution
- A program execution can be viewed as a sequence of changes of the state that transform an initial state (data) into a final state (solution)

Datatypes and variables

Assignment

- ***Assignment***: changes the state of (i.e., gives values to) a variable

```
varId = expression;
```

  `varId` and `expression` must be of ***compatible datatypes***

- ***Expression***: sequence of literals, constants, variables, operators and calls to methods that follows the syntax and is evaluated to a value of a datatype

- Examples of expressions: (suppose `int i; double x; char c;`)

```
c           i + 2
-3.05       d * 3.5
5 - 3       2 + c
```

Datatypes and variables

Assignment

- Assignment evaluates `expression` and stores the result into `varId`

```
int initialAmount;
initialAmount = 50;
```

- Assignment on its own is evaluated to a result that may be (or not) used

```
int currentAmount, initialAmount;
currentAmount = initialAmount = 50;
```

- The content of a variable ***gets lost*** when a new value is assigned to it

Datatypes and variables

Assignment

When declaring a variable, an initial value can be assigned (***initialization***)

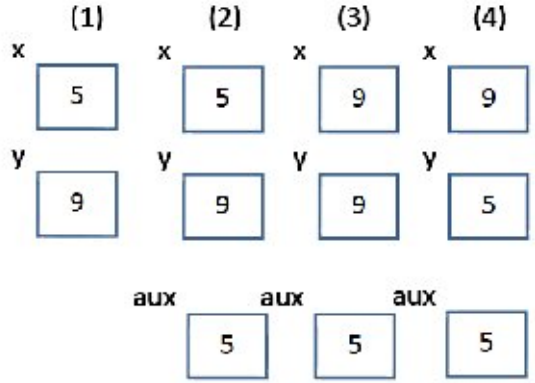
```
int var1, var2, sum = 5;
char ch1, ch2 = 'u';
double d1 = 2.0, d2 = 3.0 + d1;

var1 = 15;
sum = sum + 2;
```

Datatypes and variables

Assignment

Destructive character of the assignment makes necessary the use of auxiliar variables for some operations

```
int x = 5, y = 9;      (1)
int aux = x;           (2)
x = y;                 (3)
y = aux;               (4)
```



Datatypes and variables

Trace

Trace: representation of the state of the program

	x	y	z
`int x = 7, y = -1;`	7	-1	-
`y = 3;`	7	3	-
`int z = x + y;`	7	3	10
`x = z - x;`	3	3	10
`z = 2 * z;`	3	3	20
`y = z / y;`	3	6	20

Datatypes and variables

Constants

- ***Constant***: value that cannot be changed during the execution
- Constants in Java are declared with modifier `final`

```
final type constId = value;
```

- Constants are associated to a datatype
- Constants are declared as variables but they must be initialized

```
final int NUM_STUDENTS = 29;
```

Contents

1 Basic concepts: classes and programs ▷ 3

2 Edition, compilation, and execution in Java ▷ 8

3 Datatypes and variables ▷ 14

○ 4 *Primitive datatypes* ▷ 27

5 Reference variables ▷ 47

6 Basic classes: `String` and `Scanner` ▷ 63

7 Utility classes: `Math` ▷ 73

8 Basic input/output ▷ 78

Primitive datatypes

Integer numbers

Name	Size (N)	Minimum value $-(2^{N-1})$	Maximum value $+(2^{N-1}-1)$
`byte`	8 bits	-128	127
`short`	16 bits	-32768	32767
`int`	32 bits	-2147483648	2147483647
`long`	64 bits	-2^{63}	$2^{63}-1$

- Integer literals are considered as `int` (e.g., 35)
- To force to a `long` value, `L` or `l` must be added at the end (e.g., 35L)
- Integer literals can be expressed in several ***numerical bases***
  - Decimal (base 10): 193
  - Octal (base 8): 0301 ($3 \times 8^2 + 0 \times 8^1 + 1 \times 8^0$)
  - Hexadecimal (base 16): 0xC1 ($12 \times 16^1 + 1 \times 16^0$)

Primitive datatypes

Floating point numbers

Name	Size	Minimum value	Maximum value	Precision
`float`	32 bits	1.4×10^{-45}	3.4×10^{38}	7 digits
`double`	64 bits	4.9×10^{-324}	1.8×10^{308}	15 digits

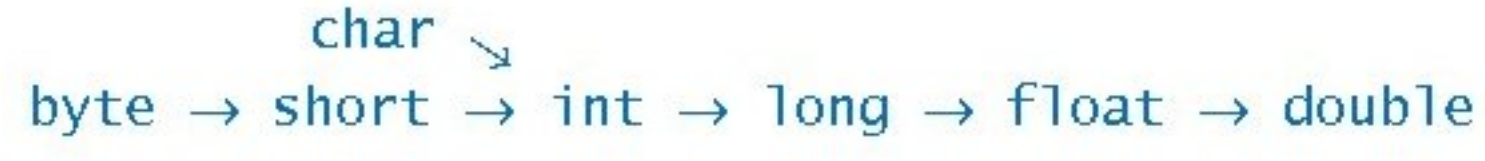
- Real number literals are considered as `double`
- To force to `float`, `F` or `f` must be added at the end (e.g., 35.5f)
- Floating point literals can be expressed in several notations:
  - Decimal: `-123.05, 0.2243, 0.00000000001`
  - Scientific: `23.4e2, -1.9E-18, 1e-11`
- ***Precision*** and ***range*** depend on the number of bits of the datatype

1 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1 = 0.5000000000000001
1 + (-0.1 - 0.1 - 0.1 - 0.1 - 0.1) = 0.5
1 - 0.9 = 0.09999999999999998

Primitive datatypes

Datatype conversion and casting

- In assignment, variable and expression must have compatible datatypes
- ***Implicit datatype conversions*** follow this order:




```
int e1 = 10;
long e2 = e1;     // int e1 is converted into a long
double e3 = e2;   // long e2 is converted into a double
```

- ***Explicit conversion*** (***casting***) forces the conversion between datatypes

```
(datatype) expression
```

```
double low = 10.0, upp = 20.0;
double sizeInt = (upp - low) / 2;
int numInt = (int) ((upp - low) / sizeInt);
```

Primitive datatypes

Arithmetic operators

Operator	Meaning	Operator	Meaning
+	Addition or sign	+=	Addition and assignment
-	Subtraction or sign	-=	Subtraction and assignment
*	Multiplication	*=	Multiplication and assignment
/	Division	/=	Division and assignment
%	Remainder	%=	Remainder and assignment
++	Increment by 1	--	Decrement by 1
a++	a = a + 1	a+=b	a = a + b

Increment/decrement operators can be used in ***prefixed*** or ***suffixed*** notation:

- ++a: a is incremented, and then it is used in the expression
- a++: a is used in the expression, and then it is incremented

Primitive datatypes

Arithmetic operators

	Expression	Result
Integers	3 + 5	8
	2 * 6	12
	7 / 2	3
	7 % 2	1
Reals	3.5 + 5.6	9.1
	3.1 * 2.0	6.2
	15.0 / 2.0	7.5
	7.0 % 2.0	1.0

Instruction	a	b
`int a = 0;`	0	
`a++;`	1	
`++a;`	2	
`a--;`	1	
`--a;`	0	
`int b = a++;`	1	0
`b = ++a;`	2	2
`b = a--;`	1	2
`b = --a;`	0	0

Primitive datatypes

Arithmetic operators

- When integers are divided, ***the result is an integer number (quotient) and the remainder gets lost***
- When an integer number is divided by zero, an ***exception*** is thrown

```
public class ejem1 {
    public static void main (String [] args){
        int den=0;
        int res=100/den;
    }
}
```

```
java ejem1
java.lang.ArithmeticException: / by zero
                at ejem1.main(ejem1.java:4)
```

- Floating point numbers divided by zero give as a result `Infinity` or `NaN`

Expression	Result
5.0 / 0.0	Infinity
-5.0 / 0.0	-Infinity
0.0 / 0.0	NaN

Primitive datatypes

Overflow

- Results of numerical expressions may exceed the datatype range: ***overflow***
- ***Integer arithmetic*** does not produce explicit overflows; incorrect results are obtained

`byte`	$127 + 1$	$=$	-128
`short`	$32767 + 1$	$=$	-32768
`int`	$2147483647 + 1$	$=$	-2147483648
`long`	$9223372036854775807 + 1$	$=$	-9223372036854775808

- To obtain correct results, datatypes with a proper range must be chosen

`int`	1000000 * 1000000	$=$	-727379968
`long`	1000000 * 1000000	$=$	1000000000000

Primitive datatypes

Overflow

- Real arithmetic reports overflows (larger than infinite) and underflows (smaller than precision)
- Results out of range produce `Infinity` of `-Infinity`

`float`	1e38 * 10 = `Infinity`
`double`	1e308 * 10 = `Infinity`

- Infinity results get propagated in the expression evaluation

(5.0/0.0)+166.386 = `Infinity`

Primitive datatypes

Numerical datatypes example

Composed arithmetic operators example:

```
long seconds = 765432; // amount of seconds
long days = seconds/(24*60*60);
seconds %= 24*60*60;
System.out.print("Days:  " + days);
System.out.println(" (Remaining seconds:  " + seconds + ")");
long hours = seconds/(60*60);
seconds %= 60*60;
System.out.print("Hours:  " + hours);
System.out.println(" (Remaining seconds:  " + seconds + ")");
long minutes = seconds/60;
seconds %= 60;
System.out.print("Minutes:  " + minutes);
System.out.println(" Remaining seconds:  " + seconds);
```

What is shown on the screen:

```
Days:  8 (Remaining seconds:  74232)
Hours:  20 (Remaining seconds:  2232)
Minutes:  37 Remaining seconds:  12
```

Primitive datatypes

Characters

Name	Size	Encoding
`char`	16 bits	Unicode

- `char` datatype represents values that are interpreted as a character (letters, numbers and special characters)
- Internally, a `char` is a positive integer number
- Any `char` is associated to a positive integer via its ***encoding***
- Java uses the ***Unicode*** encoding ***UTF-16*** `http://www.unicode.org`
- The 8 bits ***ASCII/ANSI*** encoding is a subset of Unicode (from 0 to 255)

Primitive datatypes

Characters

Any character encoding must include:

- Contiguous codes for the 10 digits, in numerical order
- Contiguous codes in lexicographical order for capital and lowercase letters
- A character for the space, end of the line, and to other special characters (tab, bell, etc.)
- The difference between the codes of any capital letter and the corresponding lowercase letter is always the same

Check that the following table accomplishes these features

Primitive datatypes

Characters

ASCII table (7 bits) 'A' row 4 column 1 → 41 hex → 65

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Primitive datatypes

Characters

- Character literals are represented between single quotation marks (')

```
char capitalA='A', lowercaseZ='z', questionMark='?',
     digit0='0', blankSpace=' ';
```

- Unicode code representation: `\u(4 digit code)` (hexadecimal)

```
char c='\u0021'; System.out.println(c);
```

Screen output:

```
!
```

0021 is the hex code for character !

- `char` literals and vars can be managed with integer arithmetics and casting

```
char a = 'a';
char c = (char) (a + 10);
System.out.println(c);
```

Screen output:

```
k
```

Primitive datatypes

Characters

- Some special characters are represented with ***escape sequences***

Escape sequence	Description
\t	Tab
\n	New line
\r	Carriage return
\b	Backspace
\’	Single quotation marks
\"	Doble quotation marks
\\	Backslash

```
char c1='\"', c2='\\', c3='\'';
System.out.println(c1);
System.out.print(c2);
System.out.println(c3);
```

Screen output:

```
"
\'
```

Primitive datatypes

Logical

Name	Size	Values
`boolean`	1 bit	`true` / `false`

- `boolean` datatype stores boolean (logical) values
- `boolean` literals: `true` and `false`
- ***Boolean expression***: any expression evaluated to `true` or `false`
- Boolean expressions are built from:
  - numerical expressions by using ***relational operators***
  - other boolean expressions by using ***boolean operators***

Primitive datatypes
Relational operators

Operator	Operation
==	Equal to
!=	Different to
>	Greater than
>=	Greater than or equal to
<	Lower than
<=	Lower than or equal to

- Their result is always of `boolean` datatype
- With `boolean` operands only == and != can be employed

```
int x = 5;
boolean b1 = 6 == x,
        b2 = x <= 7,
        b3 = (4 + x) > 10,
        b4 = 'a' < 'b',
        b5 = true == false;
b2 = b3 = 5.5 != 6.3;
```

Primitive datatypes

Boolean operators

Operator	Operation	Meaning
!	NOT	Logical negation
&&	AND	Conjunction / logical 'and'
||	OR	Disjunction / logical 'or'
^	XOR	Exclusive 'or'

- Operate on `boolean` data and its result is `boolean`
- `&&` and `||` are ***shortcut operators***: they stop the evaluation when the result is clear (e.g., `5<3 && 5<x` is evaluated to `false` independently of the value of `x`)
- Be careful: `&` and `|` operators exist, but with a different meaning

Primitive datatypes

Boolean operators

Truth table

x	y	x && y	x || y	x ^ y	!x
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

```
(val>=15) && (val<=20)     // true when val in [15,20]

(val>15) || (val==15)     // true when val>=15

(x>=0 && x<5 || x>=10 && x<=20 ) && x%2 != 1
           // true when x is even and it is in the
           // range [0,5[ or in the range [10,20]
```

Primitive datatypes

Operator precedence

	Group	Class	Operators
+	0	Parenthesis	`( )`
↓	1	Unary postfix	`(parameters)  expr++  expr--`
↓	2	Unary prefix	`++expr  --expr  +expr  -expr  !`
↓	3	Creation and casting	`new  (type) expr`
↓	4	Multipliers	`*  /  %`
↓	5	Addition	`+  -`
↓	6	Relationals	`>  >=  <  <=`
↓	7	Equality	`==  !=`
↓	8	Bitwise conjunction	`&`
↓	9	Exclusive disjunction	`^`
↓	10	Bitwise disjunction	`|`
↓	11	Shortcut conjunction	`&&`
↓	12	Shortcut disjunction	`||`
↓	13	Ternary operator	`?:`
-	14	Assignments	`=  op= (op is +,-,*,/,%,&,|,^)`

- Associativity: left to right
- Precedence can be altered by using parenthesis

Contents

1 Basic concepts: classes and programs ▷ 3

2 Edition, compilation, and execution in Java ▷ 8

3 Datatypes and variables ▷ 14

4 Primitive datatypes ▷ 27

○ 5 *Reference variables* ▷ 47

6 Basic classes: `String` and `Scanner` ▷ 63

7 Utility classes: `Math` ▷ 73

8 Basic input/output ▷ 78

Reference variables

- Java applications are organised as a set of ***objects***
- ***Object***: collection of ***data*** and ***operations***, instance of a ***class***
- ***Class***: defines object structure:
  - ***Attributes***: data
  - ***Methods***: operations
- For example, `Point` class:

  `https://docs.oracle.com/javase/7/docs/api/java/awt/Point.html`

  - Attributes: `x`, `y`
  - Methods: `getLocation`, `move`, `translate`, `...`

Reference variables

- Objects are used via ***reference variables***
- Use of objects implies access to their attributes and methods
- Java allows to create and use objects of many predefined classes
- Java allows to define new classes and create objects of that classes (Unit 4)

Reference variables

Declaration and features

- Reference variables are declared like primitive variables

```
classType refname1, refname2, ...refnamen;
```

  Examples:

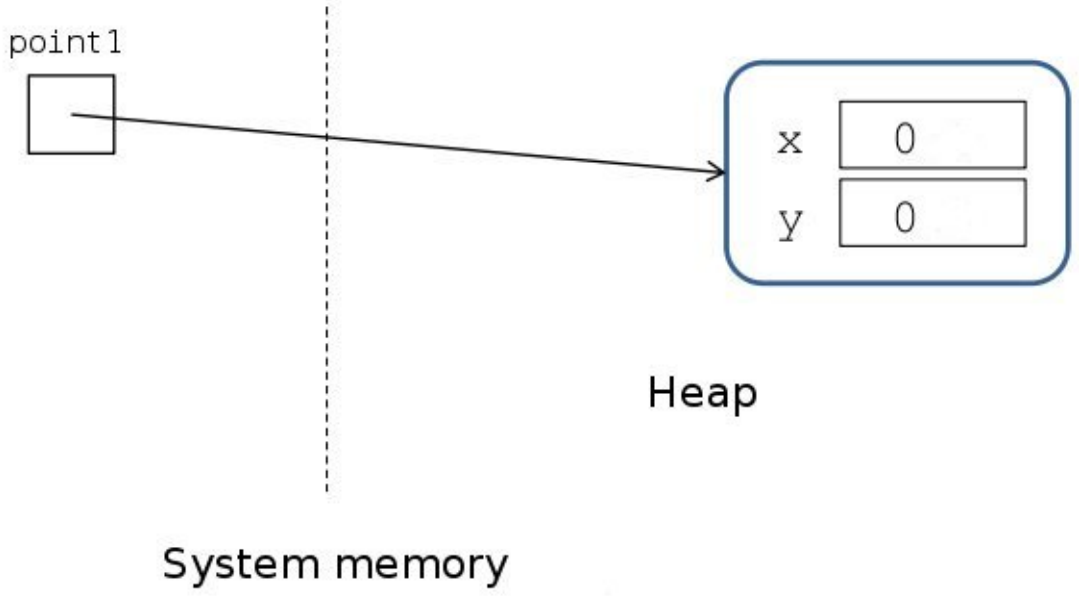
```
Scanner kbd;
String s1, s2;
Point p1, p2, p3;
```

- Objects are always manipulated via references
- `null` value: indicates the reference does not refer to an actual object
- Two reference variables with the same value refer to the same object
- Operands on references: =, ==, !=, .
- Primitive vars keep actual value, but reference vars keep the ***memory address*** of the actual value

Reference variables

Memory representation

Representation of a `Point` object, with coordinates (0,0), accessed via the reference var `point1`



Reference variables

Creation: `new`

- Reference declaration ***does not create*** the object: `new` must be used

```
// Declare p, that does not reference to any object
Point p;

// Create a Point whose reference is assigned to p
p = new Point();
```

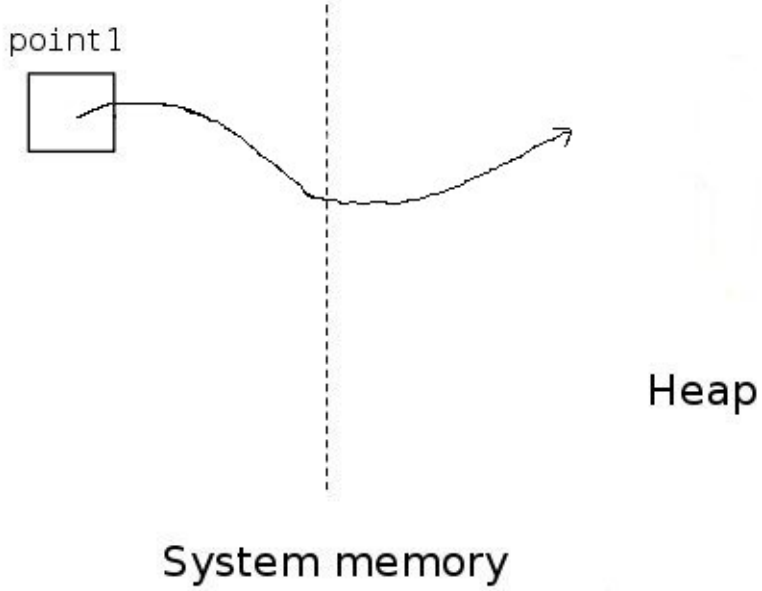
```
// Alternative:  declare p and assign to it the
// reference of the Point object that is created
Point p = new Point();
```

- Any attempt to use an uninited reference will cause an error
- `new` use calls to a special type of methods: ***constructors*** (Unit 4)

Reference variables

Creation: `new`

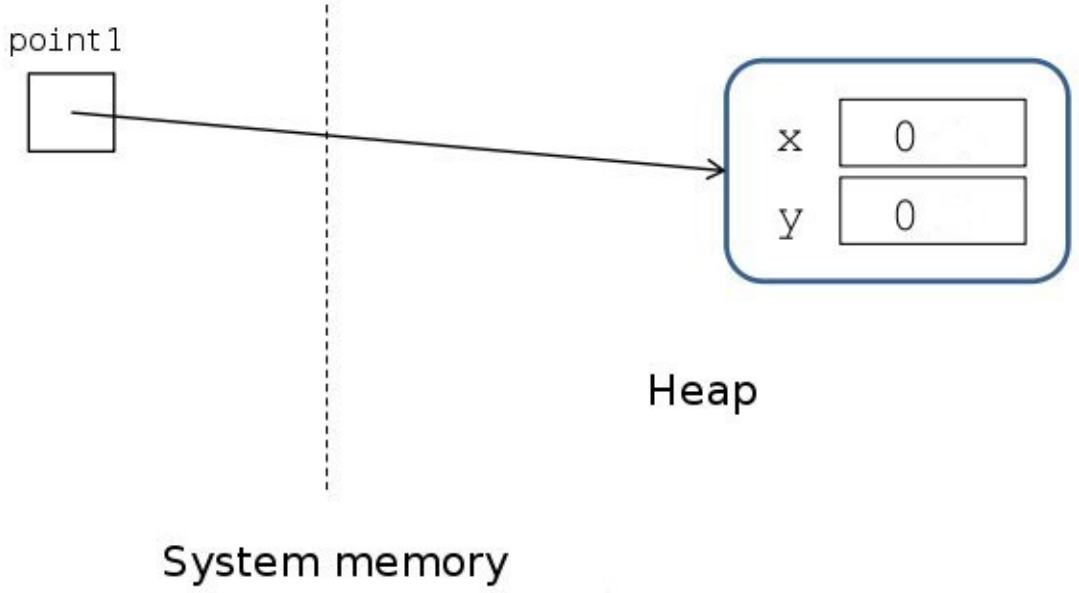
```
Point point1; // point1 does not reference to any object
```



Reference variables

Creation: `new`

```
point1 = new Point(); // point1 now references to an object
```



Reference variables

Use: the dot (.) operator

- ***Dot operator (.)***: used to access objects attributes or use a method
- Examples:

```
Point p = new Point();
String s = new String("Hello");
Scanner kbd = new Scanner(System.in);

int sum = p.x + p.y;            // Sum of coords of p is assigned to sum
p.x = 5;                        // Coordinate x of p is assigned value 5

p.move(10,10);                  // p coordinates are changed to (10,10)
int l = s.length();             // Length of s (5) is assigned to l
double x = kbd.nextDouble();    // Numeric keyboard input assigned to x
```

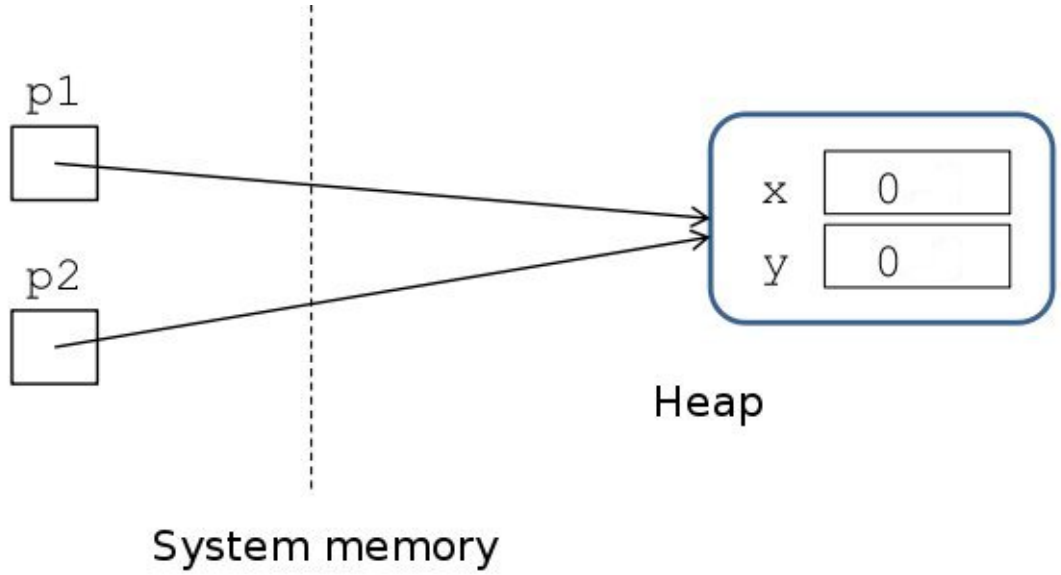
- Usually attributes cannot be directly accessed and special methods are used (Unit 4)

Reference variables

Differences with primitive variables: assignment

Reference assignment changes the references, but not the contents

```
Point p1 = new Point(); // p1 references to an object
                        // of the class Point
Point p2 = p1;          // p1 and p2 reference to the
                        // same object
```

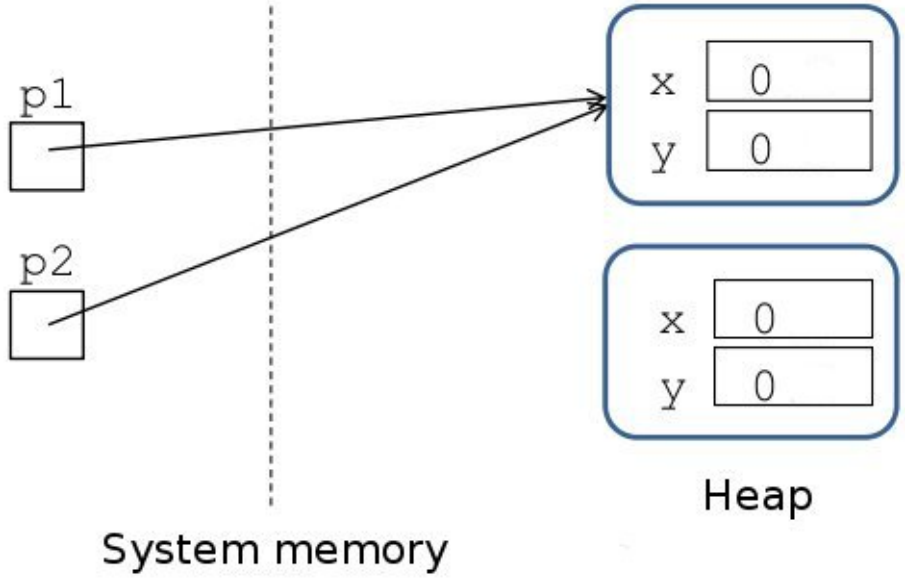


Reference variables

Differences with primitive variables: assignment

In this example an object loses its reference: it gets ***derreferenced***

```
Point p1 = new Point(); // p1 references to a Point object
Point p2 = new Point(); // p2 references to other Point object
p2 = p1;                // p1 and p2 reference the first object,
                        // the second object is derreferenced
```



Reference variables

Differences with primitive variables: object copy

It must be done attribute by attribute

E.g.:

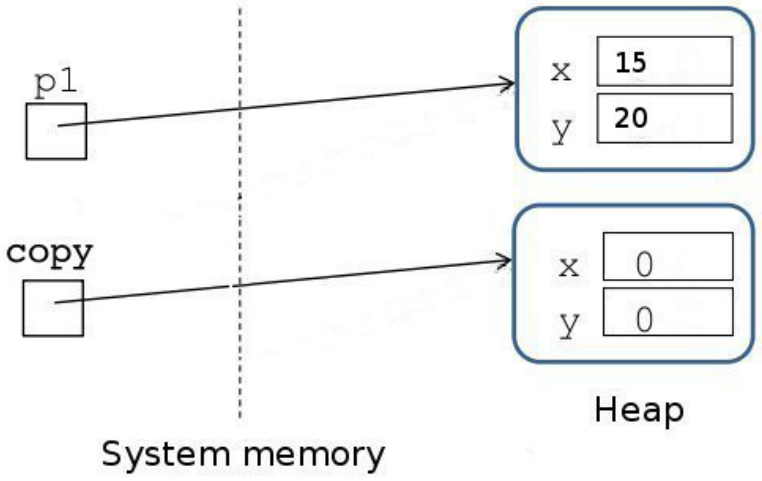
```
// Create a Point in (15,20)
Point p1 = new Point(15,20);

// create a copy
Point copy = new Point();
copy.x = p1.x;
copy.y = p1.y;
```

In general, this cannot be done directly and requires special methods (Unit 4)

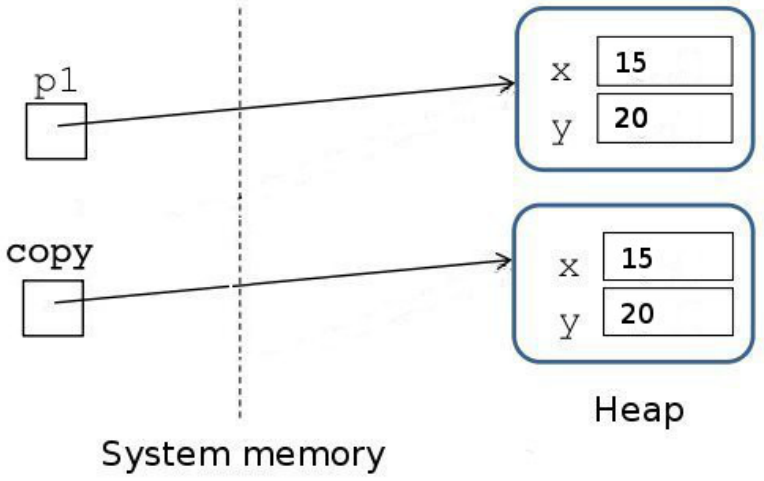
Reference variables

Differences with primitive variables: object copy

After `Point copy = new Point();`



After `copy.y = p1.y;`



Reference variables

Differences with primitive variables: object comparison

- In primitive types, == and != compare variables values
- However, in objects they compare ***the values of the references***
- Two different object references (memory addresses) are different even when their attributes have the same value
- E.g.:

```
Point p1 = new Point();
p1.x = 3; p1.y = -2;
Point p2 = p1;
Point p3 = new Point();
p3.x = 3; p3.y = -2;
System.out.println(p1 == p1);
System.out.println(p1 == p2);
System.out.println(p1 == p3);
```

→

```
true
true
false
```

Reference variables

Differences with primitive variables: object comparison

Internal equality of the objects requires compare attribute by attribute:

```
System.out.println(p1.x == p1.x && p1.y == p1.y);
System.out.println(p1.x == p2.x && p1.y == p2.y);
System.out.println(p1.x == p3.x && p1.y == p3.y);
```

→

```
true
true
true
```

The method `equals()` is the usual way of comparing object equality (Unit 4)

Reference variables
Garbage Collector

- Subsystem of the JVM to recover memory of derreferenced objects
- Usual in languages based on virtual machines (e.g., Java, C#, Python)
- In other languages (e.g., C++, Ada) the programmer must explicitly free the memory
- In Java, it starts automatically
- Its execution could be suggested by the use of `System.gc()`

Contents

1 Basic concepts: classes and programs ▷ 3

2 Edition, compilation, and execution in Java ▷ 8

3 Datatypes and variables ▷ 14

4 Primitive datatypes ▷ 27

5 Reference variables ▷ 47

○ 6 *Basic classes: `String` and `Scanner`* ▷ 63

7 Utility classes: `Math` ▷ 73

8 Basic input/output ▷ 78

Basic classes: `String` and `Scanner`

`String` class

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

- `String` is a predefined class (in the `java.lang` package)
- Allows to manage strings (sequences of characters)
- Literal `String` references: between double quotes (")
- `String` objects can be constructed in different ways

  E.g.:

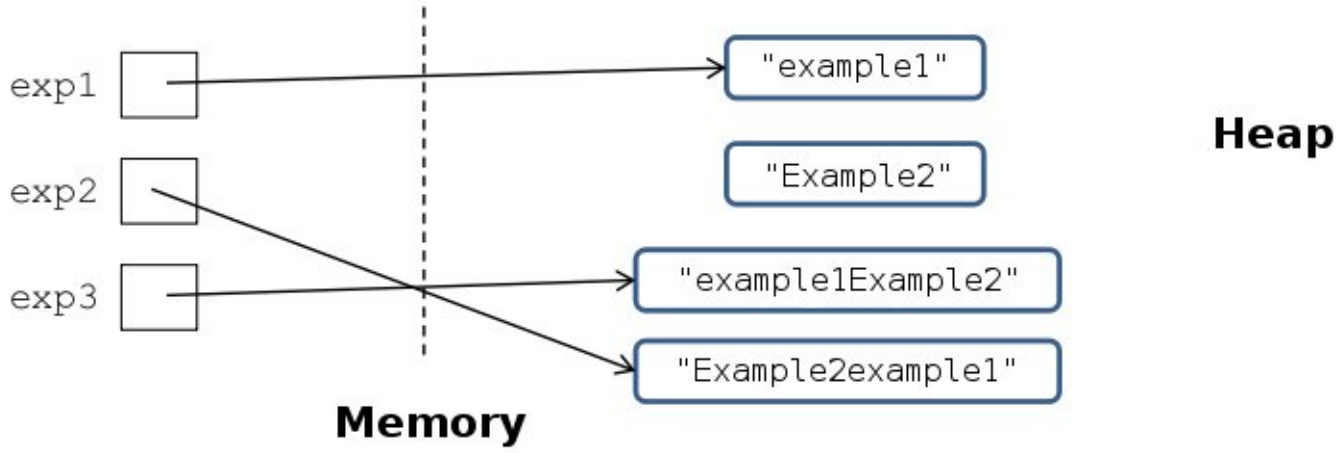
```
String st1 = "This is a String example";
String st2 = new String("This is a String example");
String st3 = "";
String st4 = new String();
```

Basic classes: `String` and `Scanner`

`String` class

- `String` objects are ***immutable***, i.e., after initialised they cannot be changed
- ***Concatenation*** is the usual way for generating new `String` objects
- Concatenation operator: `+` (+= can be used as well)

```
String exp1 = "example1";
String exp2 = "Example2";
String exp3 = exp1 + exp2;  // exp3 references to "example1Example2"
exp2 += exp1;               // exp2 references to "Example2example1"
```



Basic classes: `String` and `Scanner`

`String` comparison

- Operators == and != ***compare the references***, no the objects
- Relational operators (>, <, >=, <=) cannot compare `String`
- ***Equality comparisons*** with `equals` method (Unit 4):

```
boolean b = st1.equals(st2);  // b true only if st1 and st2
// share the same characters in the same positions
```

- ***Relational comparisons*** with `compareTo` method (Unit 4):

```
int i = st1.compareTo(st2); // i gets assigned:
// <0 when st1 is previous to st2
// >0 when st1 is posterior to st2.
// 0 when st1 is equal (char by char) to st2.
```

- Comparison between `String` objects based on ***lexicographical order*** (depends on encoding)

Basic classes: `String` and `Scanner`

`String` comparison

Examples:

```
String s1 = "Hola", s2 = "Hello", s3;
boolean eq;
eq = s1 == s2;                    // eq == false
eq = s1.equals(s2);               // eq == false
s3 = s1;
eq = s3 == s1;                    // eq == true
eq = s3.equals(s1);               // eq == true
int comp1 = s3.compareTo(s1);     // comp1 == 0
int comp2 = s3.compareTo(s2);     // comp2 > 0
int comp3 = s2.compareTo(s3);     // comp3 < 0
```

Basic classes: `String` and `Scanner`

`String` methods

`length()`	Return length (number of characters) of the string
`trim()`	Return string without initial and final spaces
`charAt(n)`	Return char at position `n`
`substring(b,e)`	Return substring between positions `b` and `e-1`
`substring(b)`	As previous, but to the end of the string (`e` missing)
`toUpperCase()`	Return string with lowercase letters transformed into capital case
`toLowerCase()`	Return string with capital letters transformed into lowercase case
`indexOf(str)`	Return first pos of occurrence of `str` in the string, -1 if not present
`lastIndexOf(str)`	Return last pos of occurrence of `str` in the string, -1 if not present
`startsWith(pref)`	Return `true` when the string starts with `pref`
`endsWith(suf)`	Return `true` when the string ends with `suf`

See a complete list in the Java reference:

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

Methods that return `String` objects, return ***new objects*** (not the originals)

First position in a `String` is ***position 0*** (e.g., `s.charAt(0)` returns the first character in `s`)

Basic classes: `String` and `Scanner`

`String` methods

Examples:

```
String st1 = "Example 1";
String cap = st1.toUpperCase();      // cap is "EXAMPLE 1"
String low = st1.toLowerCase();      // low is "example 1"
int l = st1.length();                // l == 9
char c = st1.charAt(1);              // c == 'x'
String sub = st1.substring(3,5);     // sub is "mp"
String st = st1.concat(" and 2");    // st is "Example 1 and 2"
boolean b = st1.startsWith("Exa");   // b == true
boolean c = st1.endsWith("Exa");     // c == false
int init = st1.indexOf("mpl");       // init == 3
int fromTo = st1.indexOf("mpl",2);   // fromTo == 3
String st2 = "  Example 2    ";
String noWS = st2.trim();            // noWS is "Example 2"
int last = st2.lastIndexOf("  ");    // last == 11
```

Basic classes: `String` and `Scanner`

`Scanner` class

https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

- The `Scanner` class allows to read from the keyboard in an easy form
- Must add at the beginning of the code: `import java.util.*;`
- `Scanner` object declaration: `Scanner id = new Scanner(System.in);`
- Method `useLocale`: establishes the local configuration (`public Locale useLocale(Locale l)`)
- Usual declaration for getting input from the keyboard:

```
Scanner kbd = new Scanner(System.in).useLocale(Locale.US);
```

Basic classes: `String` and `Scanner`

`Scanner` methods

Some methods of the `Scanner` class:

`next()`	Returns `String` with next token/word
`nextLine()`	Returns `String` with all characters till newline
`nextByte()`	Returns next number interpreted as `byte`
`nextShort()`	Returns next number interpreted as `short`
`nextInt()`	Returns next number interpreted as `int`
`nextLong()`	Returns next number interpreted as `long`
`nextFloat()`	Returns next number interpreted as `float`
`nextDouble()`	Returns next number interpreted as `double`
`nextBoolean()`	Returns next logic value

To read a `char`: `char c = kbd.next(".").charAt(0);`

Basic classes: `String` and `Scanner`

`Scanner` methods

Example:

```
import java.util.*;
public class TestScannerLine {
  public static void main (String[] args) {
    Scanner kbd = new Scanner(System.in).useLocale(Locale.US);
    System.out.print("Input integer: ");
    int n = kbd.nextInt();
    kbd.nextLine();
    System.out.print("Input one line: ");
    String s1 = kbd.nextLine();
    System.out.print("Input the other line: ");
    String s2 = kbd.nextLine();
    System.out.println("\nInt: " + n);
    System.out.println("Line 1: " + s1);
    System.out.println("Line 2: " + s2);
  }
}
```

Contents

1 Basic concepts: classes and programs ▷ 3

2 Edition, compilation, and execution in Java ▷ 8

3 Datatypes and variables ▷ 14

4 Primitive datatypes ▷ 27

5 Reference variables ▷ 47

6 Basic classes: `String` and `Scanner` ▷ 63

◦ 7 *Utility classes: `Math`* ▷ 73

8 Basic input/output ▷ 78

Utility classes: `Math`

Utility classes and class methods

- ***Utility classes***: no objects are created, no executable
- Mission: provide useful methods and attributes (constants)
- Used by other classes to accomplish their features
- Methods pertain ***to the class, not to the objects***: ***class methods***
- The same happens with attributes
- Methods and attributes are accessed via ***class name***
- Usual example: `Math` class

Utility classes: `Math`

`Math` class

https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

Main features:

- ***Utility class***, with class ***constants and methods*** to perform advanced mathematical operations
- For using its public methods and attributes, the corresponding identifier must be preceded by `Math.`

`Math` provides:

- Two constant values: `Math.PI`, `Math.E`
- Several operations (class methods):
  - Trigonometric functions: `sin`, `cos`, ...
  - Logarithms - exponentials: `pow`, `log`, ...
  - Random number generation: `random`
  - Basic maths: `abs`, `max`, `min`, ...

Utility classes: `Math`

`Math` class

Example:

```
double x = 2.0, y = 5.0;
// Exponentials - logarithms
double p = Math.pow(x, y);          // p == 32.0
double a = Math.sqrt(x);            // a == 1.414213562373095 1
double l = Math.log(y);             // l == 1.6094379124341003
// Trigonometric
double sin = Math.sin(Math.PI/2);   // sin is 1.0
double alf = Math.arcsin(sin);      // alf is 1.5707963267948966
double tan = Math.tan(Math.PI/2);   // tan is 1.6331239353195370E16
// Basic maths
double abs = Math.abs(-x);          // abs is 2.0
double max = Math.max(x,y);         // max is 5.0
double ceil = Math.ceil(3.76);      // ceil is 4.0
double flr = Math.floor(3.76);      // flr is 3.0
long round1 = Math.round(3.76);     // round1 is 4L
long round2 = Math.round(3.45);     // round2 is 3L
```

Utility classes: `Math`

`Math` class

Example:

```
/* Show a random number in the range [a,b] */
public class RandomAB {
  public static void main (String [] args) {
    Scanner kbd = new Scanner(System.in).useLocale(Locale.US);

    int a = kbd.nextInt(); int b = kbd.nextInt();
    int aux1 = a; int aux2 = b;
    a = Math.min(aux1,aux2); b = Math.max(aux1,aux2);

    double x0 = Math.random();   // x0 is double in [0,1[
    int rnk = b - a + 1;
    double x1 = x0 * rnk;        // x1 is double in [0, rnk[
    double x2 = a + x1;          // x2 is double in [a,a+rnk[ == [a,a+(b-a+1)[
                                 // == [a,b+1[
    int val = (int)x2;           // val is int in [a,b+1[ == in [a,b]
    System.out.println(val);
  }
}
```

Contents

1 Basic concepts: classes and programs ▷ 3

2 Edition, compilation, and execution in Java ▷ 8

3 Datatypes and variables ▷ 14

4 Primitive datatypes ▷ 27

5 Reference variables ▷ 47

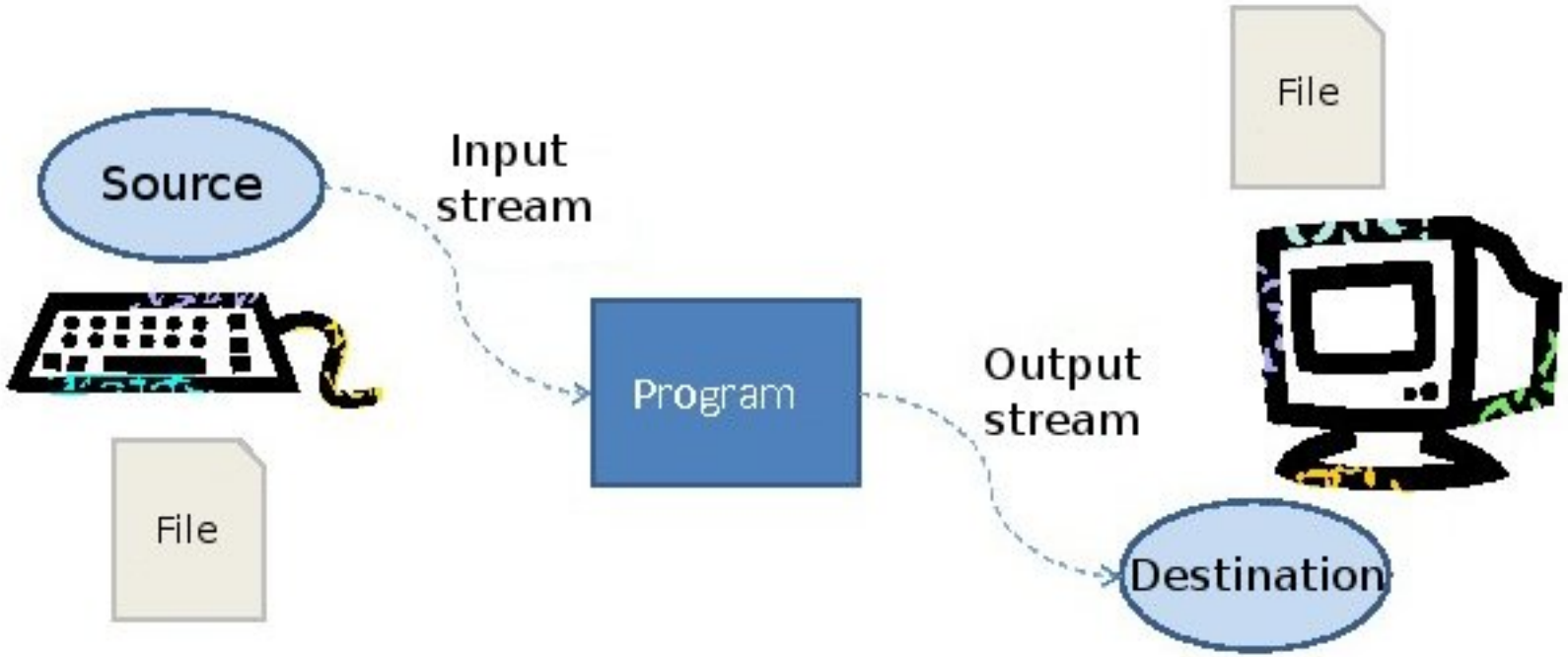
6 Basic classes: `String` and `Scanner` ▷ 63

7 Utility classes: `Math` ▷ 73

○ 8 *Basic input/output* ▷ 78

Basic input/output

Java input/output is done by using ***streams***, which are data sequences with a source (***input sources***) or a destination (***output sources***)



Basic input/output

Input

- By using the `Scanner` class methods

Output

- The syntax of the instructions that show a line on the screen are:

$$\texttt{System.out.println(}E_1 \texttt{ + } E_2 \texttt{ + ...+ } E_n\texttt{);}$$
$$\texttt{System.out.print(}E_1 \texttt{ + } E_2 \texttt{ + ...+ } E_n\texttt{);}$$

- `println` produces a new line character at the end
- Without parameters show a blank line

```
double r = 5.5, cx = 6, cy = 3;
System.out.println("Circle with radius " + r + ", unknown color");
System.out.println();
System.out.println(" and center (" + cx + "," + cy + ").");
System.out.print("Circle with radius " + r + ", unknown color");
System.out.println(" and center (" + cx + "," + cy + ").");
```

Basic input/output

```
import java.util.*;
public class TestScanner {
  public static void main (String [] args) {
    int y1, y2;
    final int ADULT_AGE = 18;
    Scanner kbd = new Scanner(System.in).useLocale(Locale.US);
    System.out.println("Input your birth year and the current year: ");
    y1 = kbd.nextInt();
    y2 = kbd.nextInt();
    System.out.print("By the end of this year ");
    System.out.println("you will be " + (y2 - y1) + " years old");
    System.out.println("Will you be an adult? " + ((y2 - y1) >= ADULT_AGE) );
  }
}
```

```
Input your birth year and the current year:
1967
2015
By the end of this year you will be 48 years old
Will you be an adult? true
```