A. Bechini
Course: Java

Introducing Java

- Java -
Introduction and Basic Syntax

Introducing Java

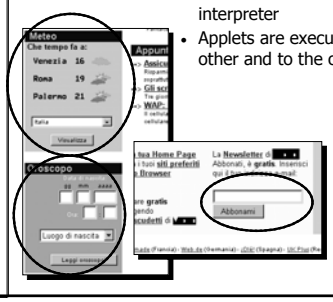
Java: A Brief History (I)

- In 1991 at SUN Microsystems it was developed a language named "OAK", within a project targeted to consumer electronics (TVs, VCRs, etc.)
- OAK didn't find any special interest neither in the industrial nor in the academic community. Things become to change as the project was turned towards Internet.
- In 1994-1995, starting from the experience matured with OAK, the Java language was created. It was presented a browser named HotJava, which was able to execute small applications within web pages: the Applets were born.
- Netscape decides to support the execution of applets in Navigator 2.0:
  this was the key event for the growth of the Java popularity.

Introducing Java

What are Applets?

- Simple Java programs within web pages
- The browser must contain a Java interpreter
- Applets are executed concurrently to each other and to the other applications



These examples are taken from pages of two Italian search engines

Introducing Java

Java: A Brief History (II)

- Nowadays, the released major versions are: 1.0, 1.1, 1.2 (a.k.a Java 2) and 1.3
  - Java 2 Platform, Standard Edition, v 1.2 (J2SE)
  - Java 2 SDK, Standard Edition, v1.2 (J2SDK)
  - Java 2 Runtime Environment, v1.2 (J2RE)
- Several new features have been added to each new version. For this reason, programs written with the old versions still run with the new ones: the opposite is not necessarily true.
- A number of IDE is currently present on the market, e.g. :
  - VisualAge - IBM
  - VisualCafé Webgain - Symantec
  - JBuilder - Borland
  - Visual J++ - Microsoft
  - Forte (netBeans, SunONE) - SUN

Introducing Java

Java: A Brief History (III)

- The origins of the Java popularity are rooted in the introduction of applets in web pages
- Anyway, Java is much more than a language for developing animations in web pages:
- It's an object-oriented, general-purpose language that can be proficiently used to build quite complex applications.
- In addition to the basic components of the language, the Java platform encompasses several libraries to support:
  - Distributed programming
  - GUI development
  - Database connectivity
  - …

Introducing Java

Java: A Brief History (IV)

Nowadays we can recognize a mature Java technology, and the Java language is a fundamental (but not unique) part of it.

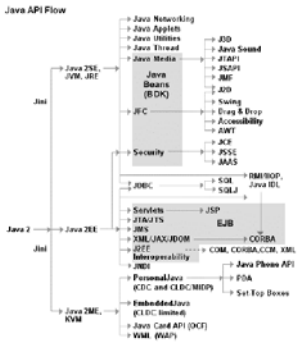
Some examples:

- JavaCards are smartcards that are able to execute Java code
- Jini is a technology that offers "plug-and-play" facilities for network connectivity to applications and appliances.
- EmbeddedJava allows embedded devices to execute Java code

… and much more

Special thanks to A. Vecchio for the previous version of these slides

Java Today



Introduction to the Java Language

Java is a language:

- Simple
- Object-oriented
- Interpreted
- Architecture independent
- Suitable to Internet
- Robust
- Secure
- Portable
- Multithreaded
- Dynamic

Java Is Simple

Java offers all the facilities of a powerful OO language, with a simple structure.

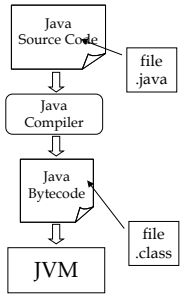
Some features of other OO languages have shown to lead to error-prone programs: such kind of features have not been included in Java.

The development of applications in Java reduces the number of the most common errors of 50%, respect to C++

⇩

The learning time is usually shorter, and the development time is decreased.

Java Is Interpreted… But How Much?



The compilation of Java source code gives origin to files that contain bytecode, a sort of set of instructions similar to machine language, but CPU-independent.

The execution of bytecode requires an interpreter: the Java Virtual Machine (JVM)

Bytecode: Example

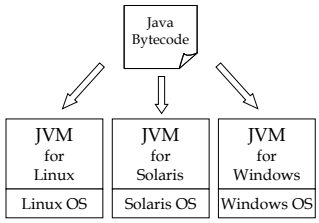
- A bytecode file is not practically comprehensible:
  - it is made of a sequence of characters in the UNICODE format
  - an usual editor would show bytecode in the following way (being unable to translate UNICODE to ASCII):

```
Êþº¾□□□-□□□□□□□□□□()V
□□□□□<init>□□□□Code□□□
ConstantValue□□□□□Edge□
□□Exceptions...
```

- The bytecode contains the complete information for the class descriptions, thus it's possible to build back the program source code starting from the bytecode.

Java Is Architecture-Independent

It's possible to execute bytecode over whatever system which is equipped with a JVM



Java Is Suitable to Internet

Java has been designed keeping in mind its use over Internet:

- Portability of bytecode is a fundamental feature in an heterogeneous environment (as Internet) made of computers with different HW and SW architectures
- The language provides libraries to let the programmer access networked resources in a straightforward way

Java Is Robust

Java gives:

- A severe type-checking mechanism
- Exception handling
- Control over null-pointers
- Control over array bounds

In particular, the exception handling policy makes it MANDATORY to the programmer to tackle problems that might happen at runtime.

Java Is Secure

- A Java program may require the use of objects from "remote" classes (e.g. an applet from another host).
- The *Class Loader* is in charge of "loading" the requested classes, previously checking their presence in the local filesystem.
- The *bytecode verifier* is in charge of checking that:
  - no code violates system security
  - no code forces pointers
  - no violations of access rights are performed
- Applets are permitted to connect only back to the host they have been downloaded from. The access to local resources is given only upon an explicit authorization of the user.

Java Is Portable

In addition to the portability of bytecode, there exists also portability at semantic level, e.g.:

- APIs (Application Programmer Interface) do not change on different systems
- The Unicode character set is able to express symbols of all languages in the world
- Integers are always represented on 32 bits, doubles on 64 bits, etc.
- It is guaranteed that operands are always evaluated from the right to the left

Java Is Multithreaded (I)

- Threads, or lightweight processes, let the simultaneous execution of different activities within the same program.
- Each thread is associated an execution flow.
- The context switch between threads yields a low *overhead* because threads share the same address space.

Java Is Multithreaded (II)

Possible uses for threads:

- Execution of intrinsic parallel algorithms
- To make the graphic interfaces more reactive
- Use of actual parallelism of underlying multiprocessors
- Responses to an unknown number of service requests
- Execution of time-consuming I/O operations and other program tasks at the same time.

Java Is Dynamic

The components of an application are linked together only at runtime.

This way, a component can be easily changed without a complete recompilation of the whole program

In particular, new modules can be dynamically added: as a program requests them, the *ClassLoader* works to find the new classes (first locally, then over the net) and downloads them.

Java: Really Slow?

- A Java program is usually slower than the corresponding one written in C/C++
- Performances are enhanced by means of a number of different techniques:
  - Just in Time Compilers
  - HotSpot technology (JDK 1.2)
  - Efficient Garbage Collectors
  - Native Compilation

Java Is Object-Oriented (OO)

Nella programmazione o-o un programma è costituito da varie componenti autonome (oggetti), ciascuna delle quali svolge un ruolo specifico e comunica con le altre secondo modi prestabiliti

Java supporta tutte le caratteristiche di un linguaggio Object Oriented

La programmazione orientata agli oggetti (1)

Con l'aumentare della dimensione dei programmi, si è andati incontro ad alcuni problemi riguardanti:

- modularizzazione del lavoro
- assemblaggio dei componenti
- testing
- mantenimento del software

Soluzione:

la programmazione orientata agli oggetti

La programmazione orientata agli oggetti (2)

Tutti i dati definiti dall'utente e quelli messi a disposizione dal linguaggio (ad esclusione dei tipi primitivi) sono oggetti.

Ciò consente:

- uno sviluppo delle applicazioni più modulare
- la riusabilità del codice
- una più semplice manutenzione del software

Gli Oggetti

- Un oggetto è un legame tra:
  - variabili (che descrivono lo stato di un oggetto)
  - operazioni che agiscono sulle variabili
- Normalmente un oggetto viene usato per descrivere un pezzetto di realtà: *componenti* e *capacità*
  - ad es: una bicicletta
    - possiede: ruote, sterzo, pedali, … *(stato)*
    - può: sterzare, forare, cambiare una ruota, … *(operazioni)*

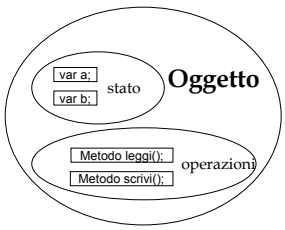
La programmazione orientata agli oggetti (3)

Un linguaggio orientato agli oggetti:

- Consente di definire oggetti
- Realizza le seguenti caratteristiche:
  - Incapsulamento La definizione di un oggetto lega le operazioni e lo stato di un oggetto, nascondendo l'implementazione
  - Ereditarietà Nuovi tipi di oggetti possono essere creati come estensioni di tipi già esistenti, consentendo il riutilizzo del codice
  - Polimorfismo Applicando la stessa funzione ad oggetti di diverso tipo è possibile ottenere gli stessi risultati semantici

Incapsulamento

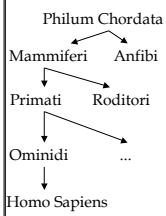
Dall'esterno l'oggetto è visto come:
un insieme di variabili
un insieme di operazioni che l'oggetto è in grado di compiere



Normalmente un oggetto:
possiede più variabili di quelle visibili dall'esterno
può possedere più metodi rispetto a quanti sono visibili dall'esterno

L'ereditarietà (2)

Le sottoclassi ereditano attributi e comportamento dalle loro superclassi



La philum chordata contiene tutte le creature che hanno una spina dorsale

Tutti i mammiferi hanno una spina dorsale: ereditano questa caratteristica dall'essere una sottoclasse della philum chordata

I mammiferi hanno anche delle caratteristiche specializzate: nutrono i piccoli con il latte, hanno un solo osso nella mascella inferiore,...

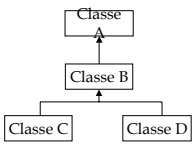
I primati ereditano tutte le caratteristiche dei mammiferi (anche la spina dorsale)

I primati si distinguono ulteriormente: grossa scatola cranica, ...

L'ereditarietà (1)

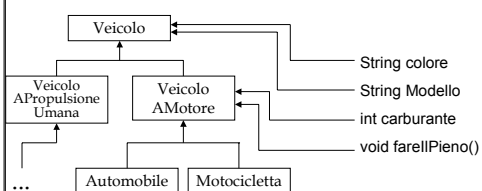
Nella programmazione orientata agli oggetti tutte le classi sono organizzate in una gerarchia

Ogni classe ha una superclasse (classe che la precede nella gerarchia) e può avere una o più sottoclassi (classi che la seguono nella gerarchia)



A è superclasse di B
B è sottoclasse di A
B è superclasse di C e D
C e D sono sottoclassi di B

L'ereditarietà (5)

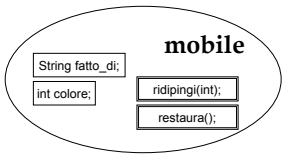


Le variabili colore e modello dovrebbero appartenere alla classe Veicolo, mentre la variabile carburante ed il metodo farelIPieno() dovrebbero appartenere alla classe VeicoloAMotore

La classe Motocicletta dovrebbe contenere solo le variabili e i metodi che sono peculiari di una motocicletta ad esempio
boolean conIlSidecar;

Ereditarietà: esempio (1)

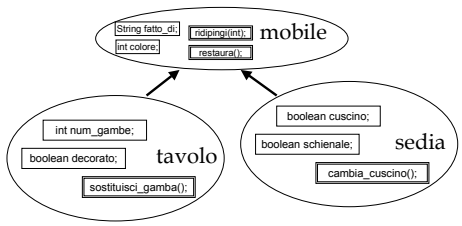
- si può definire la categoria *mobile*, che possiede uno stato (*fatto_di, colore*) e delle operazioni (*ridipingi, restaura*)



Introducing Java

Ereditarietà: esempio (2)

- definire delle sottocategorie di *mobile*
  - *tavolo* con uno stato (*num_gambe, decorato*) e un insieme di operazioni (*sostituisci_gamba*)
  - *sedia* con uno stato (*schienale, cuscino*) e le operazioni (*cambia_cuscino*)



Introducing Java

Ereditarietà: esempio (3)

- Su un oggetto di tipo *mobile,* si può soltanto fare: *ridipingi* e *restaura,* tale oggetto possiede soltanto le variabili *fatto_di* e *colore*

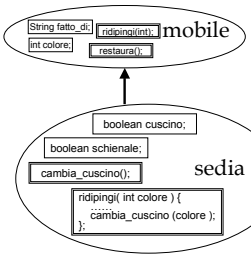
Gli oggetti *tavolo* e *sedia*, oltre alle loro operazioni, possono fare anche *ridipingi* e *restaura*
Operazioni *ereditate* da *mobile*

Gli oggetti *tavolo* e *sedia* possiedono nel loro stato le variabili *fatto_di* e *colore*
Variabili *ereditate* da *mobile*

Introducing Java

Ereditarietà: esempio (4)

- La classe sedia ridefinisce un metodo ereditato: *ridipingi*, per aggiungervi una funzionalità



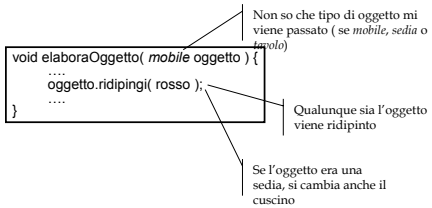
Se per un oggetto *sedia* si invoca:

restaura:	viene eseguito il metodo come definito in mobile.
ridipingi	viene eseguito il metodo ridefinito in sedia

Introducing Java

Polimorfismo

- Oggetti diversi possono definire operazioni con lo stesso nome
- Le azioni realmente svolte possono differire da oggetto ad oggetto (ad es. l'operazione *ridipingi,* per *sedia,* sono ridefinita per invocare anche *cambia_cuscino* oltre che per cambiare colore)



Introducing Java

La programmazione orientata agli oggetti (4)

- Una classe è la definizione di un modello di oggetto
- Una classe non è un oggetto
- Un oggetto è una istanza di una classe
- Una classe descrive lo stato di un oggetto *(dichiarando le variabili che lo compongono)* e le operazioni che tale oggetto può eseguire *(definendo i metodi dell'oggetto)*

Introducing Java

La programmazione orientata agli oggetti (5)

Class Members

La definizione di una classe *incapsula* le variabili che conterranno lo stato di un oggetto (*variabili membro*) e le funzioni che definiscono le operazioni che l'oggetto può eseguire (*metodi membro*)

Structure of Java Programs

A Java program is made of one or more class definitions, each of them contained in a separate file called *classname.class*. At least one of them must contain the *main()* method, which is the starting point for the execution.

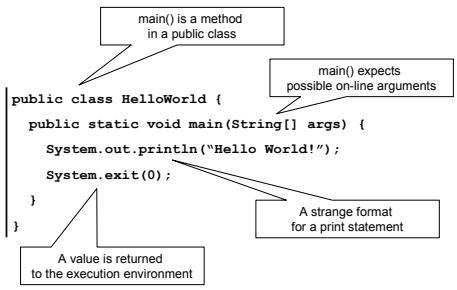
public static void main (String args[]);

In order to execute a Java program, we have to invoke the interpreter, *java*, whose unique parameter is the name of a class with a main() method.

java ClassName

The interpreter executes the *main()* method until it terminates (either naturally or forced) and all the threads it has possibly created are terminated.

A Classical Starting Point: Hello World !



A Simple Program

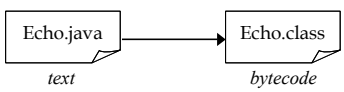
```
public class Echo {
   public static void main(String argv[] ) {
        for (int i=0; i<argv.length; i++)
                System.out.print(argv[i] + " ");
        System.out.print("\n");
        System.exit(0);
   }
}
```

This program prints the arguments passed on the command line.
The main() method must have as unique argument an array of strings, and the array length can be known through argv.length
The program defines the class Echo and the main() method.
System.exit() causes the program termination and returns the integer value 0.

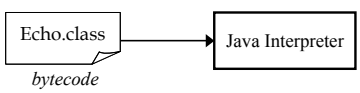
Compilation and Execution

The class *Echo* must be written in a file called:
Echo.java

Compilation: javac Echo.java



Execution: java Echo



Java: Syntax and Semantics

- Identifiers – Keywords
- Data Types - Expressions
- Control Structures
- Arrays

Comments

A Java compiler tells apart three different types of comments:

/* text */ Traditional comment: all the characters from /* to */ are ignored, as in plain C

// text One-line comment: all the characters from // tp the end of the line are ignored

/ text */** Documentation comment: the text from /** to */ can be processed by a separate tool (*javadoc)* that produces documentation on a Java program. It must be placed just before declarations (variables or methods)

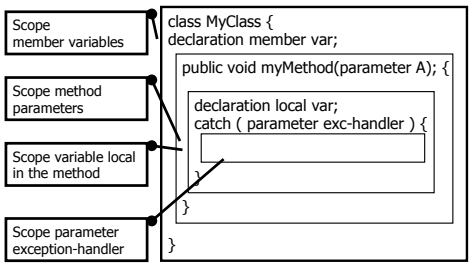
Special thanks to A. Vecchio for the previous version of these slides

Variables

- A variable is a memory location with an associated data type (it may be a **primitive data type** or a **reference**).
- A variable always contains a piece of data which is compatible with the associated type.
- The value changes because of assignments or by inc/dec operators (++/--).
- All the data types have a default value, used for variable initialization: 0, null=\u0000, false.

Variables and Scope

A variable is visible only inside the block it is declared in.



Data Types

- The data type determines the values a variable can take, and the corresponding operations as well.
- There exist two cathegories of data types:

  primitive: Integer, Real, Char and Boolean
  reference: *(pointers to objects)* classes

- The format and size of data types are guaranteed to be the same on every JVM, regardless of its internal implementation and regardless of the particular host machine.

Primitive Data Types (I)

	Type	Description
integers	byte	integer 8 bits
	short	integer 16 bits
	int	integer 32 bits
	long	integer 64 bits
reals	float	32-bit floating point single precision
	double	64-bit floating point single precision
	char	16 bit UNICODE (character)
	boolean	true or false

Primitive Data Types: Default Values

Both variables and array elements are not allowed to hold an indefinite value; Java assigns, at declaration, a default value

Type	Default value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000' (null)
boolean	false
<reference>	null

Primitive Data Types (II)

In Java, the following types (typical of C) do not exist:

pointer: the reference data type may be regarded as a pointer, but it is not possible to create a pointer to an empty area of the memory.
struct and union: in order to build complex data types, classes and interfaces can be used instead of them.

The use of classes and objects, instead of pointers and low-level data structures, leads to less error-prone programs.
In fact, they can be handled in a more secure and robust way.

Special thanks to A. Vecchio for the previous version of these slides

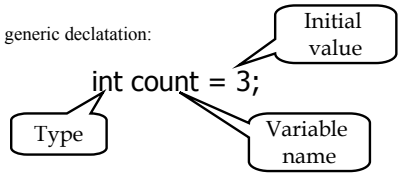
Variable Declaration

- In Java there's no difference between the declaration and the definition of a variable
- A variable can be defined everywhere in the code (it's a good programming practice grouping declarations in a C-like fashion)
- The definition of a reference variable and the creation of the "corresponding" object may be done at different moments in time.

```
MyType myVariable;  // initialized to null
myVariable = new MyType();  // obj creation
```


Examples of Variable Declarations

Format of a generic declatation:



Other definitions:

```
char c;
String txt = "a sentence";
```

Default Initialization (c = null)

Identifiers

- The *Identifiers* are the names used to identify: variables, methods, classes, etc.
- An *Identifier* is an unbounded sequence of characters (letters and numbers) that starts with a letter.
  - *Letters*: A-Z, a-z, _ (underscore), $.
  - Numbers: 0-9. \u0030-\u0039
- Two identifiers are identical if they exactly match (according to the Unicode chars)

Two identifiers, apparently similar, might be different.
E.g. in case the following characters are used:

Syntax: Keywords

The following char sequences cannot be used as identifyers:

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

Literals

It's the portion of code representing the value of a primitive type, of the "String" type or the null value

boolean literals: true, false

null literal: null

char literals: 'a' '%' '\t' '\\' '\"' '\u03a9' '\uFFFF'

String literals: "" "\"" "text" "string over" +
"2 rows"

Numeric Literals

Integer literals (32 bit):

Decimal: 0, every number starting with 1-9

Exadecimal: 0x0 0x1a1f 0xab3d 0x6D9F
(0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F)

Octal: 00 0567 0777
(0 1 2 3 4 5 6 7)

long int literals(64 bit): 0l, 0L, 0777L, 0x1000L

Introducing Java

The Integer Types

- The declaration int count;
  specifies that the variable *count*
  can only contain integer numbers,
  both positive and negative ones.
- The possible values for the different types of integers
  depend on the number of bits
  used for their representation:
  - byte (8 bits) -128 to 127
  - short (16 bits) -32768 to 32767
  - int (32 bits) -2147483648 to 2147483647
  - long (64 bits) -9223372036854775808
    to 9223372036854775807

55

Introducing Java

The char Type

- The primitive type *char* is represented on 16 bits,
  that express an unsigned integer (positive).
- Such a number is the UNICODE code
  that corresponds to the represented character
- The possible values for a *char* are:
  from 0 (= \u0000) to 65535 (= \uffff)
- The declaration: char c = 'A';
  is equivalent to:
  ```
  char c;   // c = null
  c = 'A';  // assignment
  ```

56

Introducing Java

The char Type and UNICODE

- Java programs are written using the character set
  Unicode 2.0
- All the elements of the program text (ASCII characters) are
  converted to the Unicode format:

  \uXXXX where XXXX is the exadecimal number
  of the ASCII code.
- The first 128 characters of Unicode correspond to the
  ASCII characters

57

Introducing Java

Special Characters

```
\b     /* \u0008: backspace BS          */
\t     /* \u0009: horizontal tab HT     */
\n     /* \u000a: linefeed LF           */
\f     /* \u000c: form feed FF          */
\r     /* \u000d: carriage return CR    */
\"     /* \u0022: double quote "        */
\'     /* \u0027: single quote '        */
\\     /* \u005c: backslash \           */
```

They are useful for the representation of "graphical"
characters in char or String literals:

"First :\n\tSecond." leads to: First :
Second.

58

Introducing Java

Arrays

Array declaration and creation are two separate operations
- declaration: the variable name is specified
- creation: the required memory space is allocated

Array declaration, for every kind of data type:

```
char s[];
int [] array;
int xx[][];              // 2-dimensional matrix
int [] x, yy[];      // equivalent to the following:
int x[], yy[] [];    // equivalent to the previous one
```

59

Introducing Java

Array Creation (I)

The *creation (memory allocation)*
is carried out by means of the "*new* "operator :

```
byte mybBuffer[] = new byte [1024];
int myTable [] [] = new  int [10] [15];
```

The allocated memory spaces are initialized with the
default value for the specified type
int⌫0, char⌫null, boolean⌫false

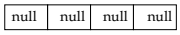
60

Array Creation (II)

The creation of an array of objects
does not imply the creation of the array elements.

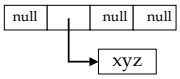
The type of the elements of an array of objects
(like String and user-defined objects) is "reference",
thus they are initialized to *null*.

```
String [] names = new String[4];
```



An array element will have a meaningful content at the creation of the corresponding object.

```
names[1] = "xyz";
```



Array Creation (III)

Examples of array creation and initialization:

```
int primes[] = {2, 3, 5, 7, 11, 13, 17, 19};
String [] friends = {"mario", "bob", "thierry", "shiva"};
```

Dynamically, the array is created and its elements are initialized. The elements can be generic expressions, and not only contant expressions.

It's not possible to create "fixed" arrays:

```
int vett [60];
```

An array cannot be used before its creation:

```
int vett [];
for (int i=0; i<vett.length; i++) {    vett[i] = i;    }
```

WRONG!

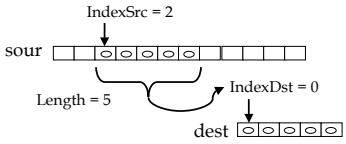
Copying an Array

```
char sour[] = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'o' );
char dest[] = new char[5];
System.arraycopy( sour, 2, dest, 0, 5 );
System.out.println( new String( dest ) + " " + new String( sour ) );
```

The above code prints: *caffe decaffeinato*

The parameters have the following meaning:

arraycopy(*sour*, *IndexSrc*, *dest*, *IndexDst*, *Length*);



Multidimensional Arrays (I)

The following statemets create arrays of arrays:

```
int pot [] [] = new int [32] [8];
int param [] [] = { {11, 12, 13}, {21, 22, 23}, {31, 32, 33} }
```

Some dimensions can be omitted (at the end)

```
double temp [] [] [] = new double [5] [3] [];
```

It's not legal to write:

```
double temp [] [] [] = new double [5] [] [3];
```

WRONG!

Multidimensional Arrays (II)

It's possible to define flag-shaped arrays:

```
int [] [] twoDim = { {1,2}, {3, 4, 5}, {6, 7, 8, 9} };
```

Example of a "triangular" array:

```
long triangle [] [] = new long [10] [] ;        // 10 elements
for (int i=0; i < triangle.length; i++) {       // for each element
        triangle[i] = new long [i+1]; // it allocates an array
        for (int j=0; j< i+1; j++)
                triangle[ i ] [ j ] = i+j;
}
```


Expressions

Most part of a Java computation is spent in the evaluation of expressions:

- For their side-effects
- To assign a value to a variable
- To calculate their value, as arguments or operands in more complex expressions.

Every expression has an associated type

```
byte = byte + byte;      int = short + int;      double = int + double;
```

Java operates an automatic "safe" cast
E.g. *short* to *int*, *int* to *double*.
double to *int* is not done, as well as *int* to *boolean*.

Expressions: evaluation

Operands in expressions are evaluated ***rightward***.

```
int i = 1;
int j = (i=2) * i;   //j takes always 4
```

"Partial" values can be used in evaluations, e.g. :

```
int t = 3;
t += (t=2);        //definitely t=5
```

this stands for *t = t + (t=2);*

the first operand is evaluated ($t \rightarrow 3$),
then the second ($t \rightarrow 2$),
then 3+2 is evaluated ($3+2 \rightarrow 5$)
and assigned to the variable ($t \leftarrow 5$).

Generally it is recommended to write down expressions as more readable as possible.

Expressions: parentheses

Pay attention: some expessions, apparently similar, could result in different oucomes. E.g.

```
class foo {
        public static void main(String args[]) {
                double d = 8e+307;
                System.out.println(4.0 * d * 0.5);
                System.out.println(2.0 * d);
        }
}
```

output: Infinity **// as the first evaluation determines an overflow**

1.6e+308 **// as in the case (4.0 * (d * 0.5))**

NB: " x + y / z " is equivalent to " x + (y/z) "

Operators

Operators carry out "calculations"
from one, two or three ***operands***, getting to a ***result***.

Unary operators take just one operand.

count**++**; increments count by one

Bynary operators exploit two operands.

a **+** b; //sum of two variables
a **=** b; //*assaign* (no comparison) value b to variable a

Ternary operators (just one in Java) act on three operands.

expr **?** op1 **:** op2 // if *expr* is true, returns *op1*, otherwise *op2*

Binary arithmetic operators

+ sum of two elements op1 + op2
(with int operands, it returns an int, with long ones, it returns a long, etc...).

- subtraction: op1 - op2
***** multiplication: op1 * op2
/ division: op1 / op2
% modulo: op1 % op2

NB: + is used also to expess string concatenation
String res = "There are " + count +" elements."

Unary arithmetic operators

++	increments by 1
--	decrements by 1

postfix notation	count++; if count was 5, it returns 5 and then increments count
prefix notation	++count; sif count was 5, it increments counts and then returns 6

+ as unary operator +op pit casts *op* to *int* if it was *byte, short* or *char*
- as unary operator -op arithmetic negation of *op*

Relational operators

They are aimed at comparing operators according to a specific relationship between them.
E.g. **!=** returns true if the operands are not equal

		Return *true* if:
>	op1>op2	op1 greater than op2
>=	op1 >= op2	op1 greater orequal to op2
<	op1 < op2	op1 less than op2
<=	op1 <= op2	op1 less or equal to op2
==	op1 == op2	op1 equal to op2
!=	op1 != op2	op1 not equal to op2

They are typically used coupled to conditional operators in order to build up complex decisional expressions

Special thanks to A. Vecchio for the previous version of these slides

Logic operators

They are aimed at performing logic operations on operands.

		Return *true* if:
&&	op1 && op2	op1 AND op2 ; op2 is evaluated iff op1 is *true* (*lazy* or *shortcut* evaluation).
||	op1 || op2	op1 OR op2 ; op2 is evaluated iff op1 is *false* (*lazy* or *shortcut* evaluation)
!	! op1	op1 is *false*
&	op1 & op2	op1AND op2 (op2 is *always* evaluated)
|	op1 | op2	op1 OR op2 (op2 is *always* evaluated)

Bitwise operators

>>	op1 >> op2	rightward shift of *op1* 's bits of *op2* positions, with sign estension (divide by 2).
<<	op1 << op2	leftward shift of *op1* 's bits of *op2* positions, with sign estension (times 2)
>>>	op1>>> op2	rightward shift of *op1* 's bits of *op2* positions, with zero estension.
&	op1 & op2	bitwise AND: ($12_{(1100)}$ & $13_{(1101)}$ =$12_{(1100)}$)
|	op1 | op2	bitwise OR: (12 | 13 = 13)
^	op1 ^ op2	bitwise XOR: (12 ^ 13 = $1_{(0001)}$)
~	~op1	bitwise negation

Bitwise operators: example

This kind of operators is commonly used to manage a set of boolean flags that "encode" the state of somewhat components.
E.g. to deal with the object *refrigerator*, that could be: open/closed, full/empty, on/off:

We define three costants (*final* makes the values not changeable) and the variable *flag*:

```
final int OPEN = 1;      // first bit:1, the others 0
final int FULL = 2;      // second bit:1, the others 0
final int ON = 4;        // third bit:1, the others 0
int flag = 0;            // all bits assigned to false
```

The refrigerator state is set and checked:

```
flag = flag | OPEN;      //  now it's open for sure
if ( flag & ON ) {...    //  in case it is ON ...
```


Assignment operators

`int count = 0;` *count* is initialized by an assignment

`i = i + 2;` can be expressed also as: `i += 2;`

+=	op1 += op2	same as op1 = op1 + op2;
-=	op1 -= op2	op1 = op1 - op2;
*=	op1 *= op2	op1 = op1 * op2;
/=	op1 /= op2	op1 = op1 / op2;
%=	op1 %= op2	op1 = op1 % op2;
&=	op1 &= op2	op1 = op1 & op2;
|=	op1 |= op2	op1 = op1 | op2;
^=	op1 ^= op2	op1 = op1 ^ op2;
>>=	op1>>= op2	op1 = op1 >> op2;
<<=	op1<<= op2	op1 = op1 << op2;
>>>=	op1>>>= op2	op1 = op1 >>> op2;

Cast operator *(type)*

The cast operator (type) can be applied to any data type, primitive types (int, char,…) and objects (class instances). It's aimed at converting values across different types, provided that compatibility is assured between them:

```
(type) variable;
```

E.g.:

```
int x=3; double d=4.2, dd;
dd = (double) x;      // value x is converted into double,
                      // dd=3.0
x = (int) d;     // this is a "lossy" transformation, as
                 // an approximation is introduced:   x = 4
```


Operator *instanceof*

instanceof is a relational opreator:
it checks the leftside object to belong to the specified class.
It returns *true* if it belongs to, *false* either if it does not belong to or if th object is *null*.

if (*myObject.instanceof(* **type** *)*);

it cannot be applied to primitive types: just to objects (class instances)

Special thanks to A. Vecchio for the previous version of these slides

Syntax: Keywords and literals

The keywords const and goto are of no use in Java

The words true, false, null are not keywords, but literals, and cannot be used as itentifiers.
NB *true*, *false* and *null* are written all in lowercase

Control Structures: *if*, *else*

If (expr) *Statement*
else *Statement*

Statement can be either a single intruction or a block { }.

```
int count;
count = getCount();
if (count < 0) {                    // boolean expression
        System.out.println("error");
}
else {
        System.out.println("There are " + count + " elements.");
}
```

The type boolean is not converted into any other types;
0 and *null* are not equivalent to *false*,
and values *not-zero* or *not-null* are not equivalent to *true*.

Control Structures : *switch*

Syntax:

```
Switch (expr1) {
        case expr2:
                statements;
                break;
        case expr3:
                statements;
                break;
        default:
                statements;
                break;
}
```

Example:

```
Switch (mese) {
        case 4:
        case 6:
        case 9:
        case 11:        numDays = 30;
                        break;
        case 2:         numDays = 28;
                        break;
        default:        numDays = 31;
}
```

- In *case* experssions, the following types are allowed:
  byte, *char*, *short*, *int* and *long*

Control Structures : The *for* Loop

Syntax:

```
for (init_expr; test_expr; incr_expr) {
        statements;
}
```

Example (*i* from 0 to 9):

```
for (int i=0; i<10; i++) {
        System.out.println("i="+i);
}
```

Multiple expressions are allowed
in the initialization and increment sections:

```
int i; String s;
for (  i=0, s="text";                   // variable initialization
       (i<10) && (s.length() >= 1 );    // test
       i++, s=s.substring(1)) {         // variable increment
                System.out.println(s);  // loop body
}
```


Control Structures : *while*, *do-while*

while:

```
while (test_expr) {
        statements;
}
```

do - while:

```
do {
        statements;
} while (test_expr);
```

example:

```
int i=10;
while (i-->0) {
        boolean b = getFlag();
        if (b) {
                do { .... } while ( j != 0 );
        }
}
```


Flux Control

label:*statement*; It labels a statement *for*, *while* or *do*

return *expr*; exits from the method, returning the value of *expr*

break [label]; it terminates the loop that contains it

continue [label]; pass to the next loop iteration:
first *incr_expr* and than *test_expr* are executed.

for (init_expr; test_expr; incr_expr);

Special thanks to A. Vecchio for the previous version of these slides

Flux control: *break*, *continue*

```
extl: while (!done) {
        if ( test(a,b) == 0) continue;          // goes to point 3
        if ( test2(c) ) break;                  // goes to point 4
        for (int i=0; i<10; i++) {
                if (a>0)    continue;           // goes to point 1
                if (b<10)  continue esterno;    // goes to point 3
                if (c>1000)   break;            // goes to point 2
                if (c<10)   break esterno;      // goes to point 4
                ...
                // point 1. Executes Incr_expr and goes on with the
                                loop
        }
        // point 2. Out of the for cycle
        ...
        // point 3. Goes on with the while cycle
}
// punto 4. Out of the while loop.
```


Example of Java Program

```
public class Examp {
   public static void main (String args[]) {
        int tot = 0;
        int len[] = new int[args.length];
        System.out.println("There are " + args.length + " parameters:");
        for ( int i=0; i<args.length; i++) {
             len[i] = args[i].length();
             tot += len[i];
             System.out.println("-> " + args[i] + " is long " + len[i]);
        }
        System.out.println("Totally, " + tot + " characters are present.");
   }
}
```

length is a var in arrays

length() is a method in class String

javac Examp.java
java Examp one two three four

Exercises

Creare una classe di nome PrintArgs che prende gli argomenti passati all'applicazione e li memorizza in un array. Stampare quindi gli elementi dell'array in ordine inverso, uno per linea, oppure un apposito messaggio se non ci sono argomenti.
Creare una classe che faccia la somma dei primi 'n' numeri interi, dove 'n' è un parametro passato da linea di comando (usare il metodo *String.intValue()*)

Special thanks to A. Vecchio for the previous version of these slides