Il linguaggio Java

1

Java

- Linguaggio di programmazione
  - Object oriented
  - Sintassi simile al C
  - Multipiattaforma (windows, linux, Mac OS X, ...)
  - Ricco di API «standard»
  - Originariamente pensato per il web (sicurezza)
- Eseguito su Java Virtual Machine (JVM)
  - Macchina astratta (stack-based)
  - Sorgenti trasformati in linguaggio bytecode (.class)
  - Eseguiti da un processo «java»
    - Interpretati – compilati JIT (Just in Time)

2

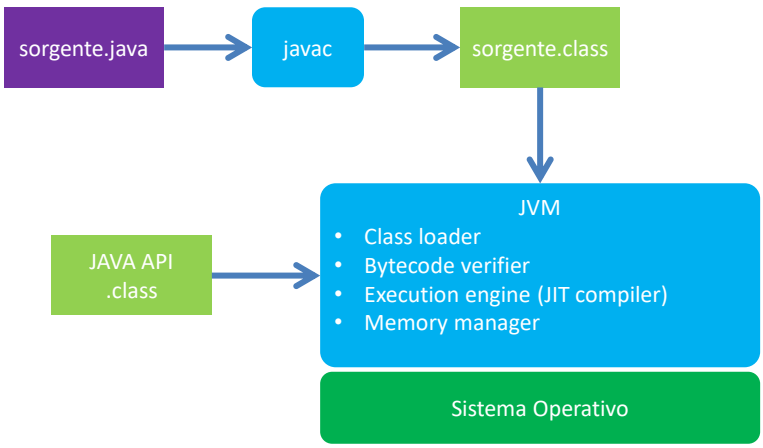
Hello world!

HelloWorld.java

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


3

JVM



4

JDK & JRE

- Java Developement Kit
  - **javac** → compilatore
    - Trasforma file **.java** in file **.class**

      $ javac HelloWorld.java
  - **javap** → disassemblatore
    - $ javap –c HelloWorld.class
  - …
- Java Runtime Environment
  - **java** → esecutore
    - Esegue i file .class

      $ java HelloWorld

      Hello world!
- http://www.oracle.com/technetwork/java/javase/downloads

5

JVM - bytecode

```
public static void main(String[] args) {
    int a=10;
    int b=(a*6)+2;
    System.out.println(b);
}
```

- istruzioni per macchina astratta che lavora con uno stack, operandi prelevati dallo stack e risultato inserito su stack
- accedono anche a un insieme di variabili locali (indicate con un numero)

```
public static void main(java.lang.String[]);
  Code:
     0: bipush        10
     2: istore_1
     3: iload_1
     4: bipush        6
     6: imul
     7: iconst_2
     8: iadd
     9: istore_2
    10: getstatic java/lang/System.out:Ljava/io/PrintStream;
    13: iload_2
    14: invokevirtual java/io/PrintStream.println
    17: return
```


6

Tipi di base

- **byte** (8 bit, -128 ÷ 127)
- **short** (16 bit, -32728 ÷ 32727)
- **int** (32 bit, $-2^{31} \div 2^{31}-1$)
- **long** (64 bit, $-2^{63} \div 2^{63}-1$)
- **float** (32 bit, floating point IEEE 754)
- **double** (64 bit, floating point IEEE 754)
- **char** (16 bit, codifica UNICODE)
- **boolean** (true - false)
- String

Variabili e Costanti

- Dichiarazione variabili simile a C
  - **<tipo> <variabile> [= <espressione>];**
  - *Esempi*

```
int a = 32;
int f = 0xff00;      /*esadecimale*/
int b = 0b11110000; // binario
float x = 0.1f;
double y = 2.34d;
char c = 'x';
char nl = '\n';
char alpha = '\u03b1';  //codice unicode
String name = "John";
```

- Contrariamente al C e C++ ogni variabile e array è sempre inizializzata a zero, false, null

Conversione da stringa

- Per convertire una stringa in numero
  - int x = Integer.parseInt("123");
  - double d = Double.parseDouble("3.14");
  - float f = Float.parseFloat("12.1");

Operatori

- Sono gli operatori del C
  - **Aritmetici**: +, -, /, *, %
  - **Bit-wise**: &, |, ^, ~
  - **Shift ops:** <<, >>, >>>
  - **Logici**: &&, ||, !
  - **Confronto**: <, <=, >=, >, ==
  - **Parentesi**: (,)
  - **If in linea:** ... ? ... : ...
  - **Assegnamento:** =, +=, -=, *=, etc.
  - *Esempi:*
    - x = (3*y)-12;
    - b = ((f & 0xff00) >> 8) | ((g & 0xff) << 8);

Istruzioni

- uguali a quelle del linguaggio C
  - **if (**<condizione>**)** <blocco> [**else** <blocco>]
  - **while(**<condizione>**)** <blocco>
  - **do** <blocco> **while(**<condizione>**);**
  - **for(**<init>**;**<condizione>**;**<incr>**)** <blocco>
  - **switch(**<espr>**) {**
    case <const>**:** <istruzioni>
    ...
    default: <istruzioni>
    }
  - **{** <istruzione>**;** … **}**

Array

- Vettori di un tipo:
  - <tipo>**[]** <variabile> [**= new** <tipo>**[**<espr>**]**];
  - <tipo>**[]** <variabile> [= **{** <espr$_1$>, <espr$_2$>, ... <espr$_n$> **}**]
  - *Esempi:*
    - int[] x = new int[100];
    - float[] ff = new float[n];
    - char[] s = { 'a', 'b', 'c', 'd' };
    - int[][] m = new int[3][2]; //matrice 3x2
    - float[][] v = { {1.1, 2.2, 3.3}, {4.4, 5.5, 6.6} }; //matrice 2x3

Array

- Si usano [...] per accedere agli elementi con indice da 0 a lunghezza -1;
  - Esempi:
    - x[1] = x[0]*2;
    - int l = x.length; //lunghezza del vettore
- Se si accede con indice non valido genera una Eccezione!
- Gli array sono degli oggetti allocati nello heap, mentre variabili con tipo di base sono nello stack

Array

- Array simile a puntatore
- Attenzione alla assegnazione, copia il riferimento non tutto l'array

```
int[] x = { 5, 3, 1};
int[] y = x; //copia riferimento
y[1] = 2;
System.out.println(x[1]);  //stampa 2 non 3!
```

Esercizio

- Sommare i numeri passati da riga di comando

  $java Somma 12 34 450

Somma.java

```
public class Somma {
        public static void main(String[] args) {
                int s = 0;
                for(int i=0; i<args.length; i++) {
                        System.out.println(args[i]);
                        s += Integer.parseInt(args[i]);
                }
                System.out.println("somma="+s);
        }
}
```


String

- Sequenze di caratteri immutabili
  - String a = "Hello ";
  - String name = "John";
  - String msg = a **+** name **+** "!"; **//concatenazione**
  - int l = msg**.length**(); **//lunghezza della stringa**
- Anche le stringhe sono oggetti memorizzati nello heap
- Non sono array!

String

- ATTENZIONE al confronto!

```
String a = "1234";
String x = "12";
String b = x + "34";
if(a == b) //confronta i due puntatori
    System.out.println("Uguali!");
```

Il confronto a == b difficilmente sarà vero.

Si usa:

```
if(a.equals(b))
    ...
if(a.compareTo(b)==0) //si usa per comparare due stringhe
    ...
```


Esercizio

- Cercare una stringa in un array di stringhe.

```
String[] mesi = {"gen", "feb", "mar", ...};
String m = "mar";
int mese = 0;
for(int i=0; i<mesi.length; i++){
        if(mesi[i].equals(m)){
                mese = i+1;
                break;
        }
}
if(mese==0)
        System.out.println("mese non valido");
else
        System.out.println("mese: "+mese);
```

Garbage Collector

- Gli oggetti creati con operatore **new** non devono essere distrutti esplicitamente (come in C++) il Garbage Collector si occupa di liberare lo spazio di memoria non più raggiungibile.
- E' una attività eseguita quando c'è bisogno di memoria.
- Si può usare il programma *jconsole* (fornito con il JDK) per vedere l'occupazione di memoria della JVM

19

Classi e Oggetti

- Una **classe** rappresenta un insieme di entità che condividono:
  - delle stesse caratteristiche (attributi)
  - delle stesse funzionalità che possono essere eseguite su queste entità.
  - Esempio: automobile
- Gli **oggetti** sono gli elementi che fanno parte di questi insiemi e sono detti istanze della classe

20

Classi in Java

visibilità della classe

Point.java

File con stesso nome della classe

```
public class Point {
        private float x = 0;
        private float y = 0;

        public Point(float xx, float yy) {
                x = xx; y = yy;
        }
        public float getX() {
                return x;
        }
        public float getY() {
                return y;
        }
        public Point add(Point p) {
                return new Point(x + p.x, y + p.y);
        }
}
```

ATTRIBUTI

visibilità dell'attributo

COSTRUTTORE

visibilità del metodo

METODI

Classi, uso

PointsProgram.java

```
class PointsProgram {
        public static void main(String[] args) {
                Point p = new Point(1,2);
                p.x = 3; //ERRORE, attributo x è privato
                p = p.add(new Point(1,1));

        System.out.println(p.getX()+","+p.getY());
                p = null;
        }
}
```

Reference

- La variabile **p** è una reference ad oggetto di tipo *Point*.
- Gli oggetti associati alle reference sono allocati esclusivamente nello Heap
- Una reference può avere valore **null**
  - **p = null;**
  - Se una reference è null e si chiama un metodo usando la reference viene generata una eccezione «null pointer exception» che blocca il programma
  - p.getX()
- Attenzione al confronto! è un confronto tra reference

```
Point p1 = new Point(1,1);
Point p2 = new Point(1,1);
…
if(p1==p2) … //questo è falso, p1 e p2 puntano a due oggetti diversi
```


Passaggio parametri

- I parametri di un metodo sono passati:
  - **per valore** i tipi di base
  - **per riferimento** oggetti, stringhe ed array

```
void calcola(int a, float[] b, String c) {
  a = a * 2;
  b[0] = 1;
  c = c + "d";
}
...
int v = 10;
float[] x = new float[5];
String s = "abc";
calcola(v, x, s);
```

Visibilità

- **private**: visibile solo dai metodi della classe
- **protected**: visibile dalla classe, dalle classi derivate *e dalle classi dello stesso package*
- **public**: visibile da tutti
- se omesso è visibile dalle classi all'interno dello stesso package
- Una classe può essere solo public o visibile all'interno del package

File .java

- Il nome del file java deve corrispondere al nome della unica **classe pubblica** presente al suo interno
- Possono essere definite anche altre classi nel file ma devono omettere la visibilità *public*, quindi saranno accessibili a livello di package

Esempio: Lista interi

```
public class IntList {
        private int value;
        private IntList next = null;
        public IntList(int value, IntList next) {
                this.value = value; this.next = next;
        }
        public IntList(int value) {
                this(value,null);
        }
        public IntList add(int v) {
                if(next == null) next = new IntList(v);
                else next.add(v);
                return this;
        }
        public String toString() {
                if(next==null)
                        return "" + value;
                return value + ", " + next.toString();
        }
```

chiama altro costruttore

per *method chaining*, **this** rappresenta l'oggetto stesso sul quale il metodo è stato chiamato

Lista, uso

```
IntList l = new IntList(3).add(2).add(1);
System.out.println(l);
l = new IntList(4, l);
System.out.println(l);
```

method chaining

chiama l.toString()

Overloading

- Posso definire metodi con lo stesso nome ma con parametri in numero e di tipo diverso
- Es.
  - Point add(Point p) …
  - Point add(float x, float y) …
- Posso scrivere:
  - p = p.add(1,2);

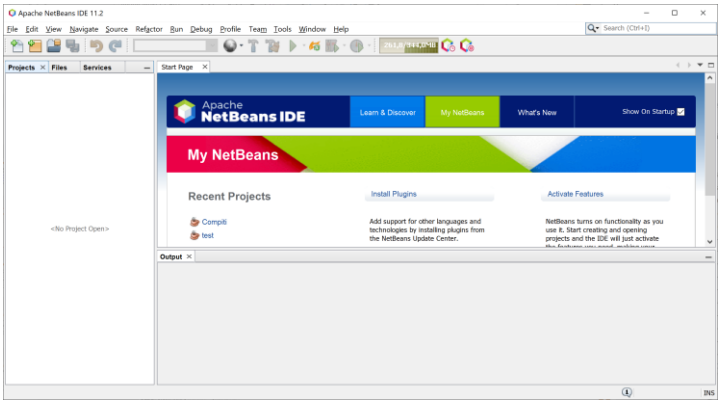
  Invece di
  - p = p.add(new Point(1,2));
- Overloading dei costruttori

IDE per Java

- **Integrated Development Environment**
- Ci sono tante possibilità, tra cui:
  - eclipse https://www.eclipse.org/eclipseide/
  - intelliJ IDEA https://www.jetbrains.com/idea/
  - **Apache NetBeans** https://netbeans.apache.org/
- In questo corso usiamo NetBeans

Apache NetBeans v12

- Progetto open source ora gestito da apache.org, fino alla versione 8 era gestito da Oracle



31

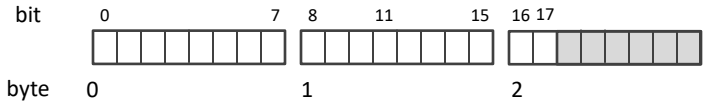
Esercizio: BitArray

- Realizzare un array che contenga *n* bit, in modo che si possa impostare ed accedere ad ogni singolo bit e che usi la minore quantità di memoria possibile.
- Usare **bit-wise e shift operators** per modificare il singolo bit di un byte
- Testarlo definendo un array di 1001 bit e impostare a 1 i bit in posizione pari e 0 i bit dispari

32

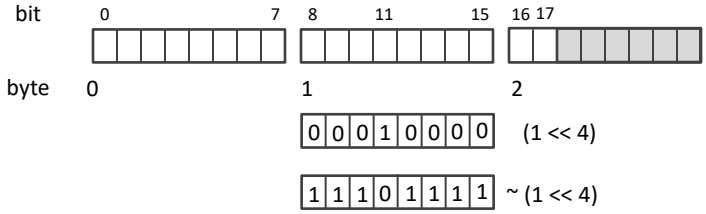
Esercizio BitArray

- N = 18bit ➔ servono 3 byte (N+7)/8



- **p/8** indica il byte dove si trova il bit da modificare
- **7-p%8** indica posizione bit nel byte (da destra)
- Es: p=11
  11/8 = 1
  7 - 11%8 = 7 - 3 = 4

Esercizio BitArray



- Leggere stato di bit in posizione *x* in byte *b*
  b & (1 << x)
- Impostare a 1 un bit in posizione *x* in byte *b*
  b = b | (1 << x)
- Impostare a 0 bit in posizione *x* nel byte *b*
  b = b & ~ (1 << x)

Soluzione: BitArray

```
public class BitArray {
        private byte[] bits;
        public BitArray(int nbits) {
                bits = new byte[(nbits+7)/8];
        }
        public boolean get(int bit) {
           return (bits[bit/8] & (1 << (7-bit%8))) >0;
        }
        public BitArray set(int bit, boolean v) {
                if(v)
                        bits[bit/8] |= 1<<(7-bit%8);
                else
                         bits[bit/8] &= ~(1<<(7-bit%8));
                return this;
        }
}
```


BitArrayTest

```
public class BitArrayTest {
        public static void main(String[] args) {
                BitArray bits = new BitArray(1001);
                for(int i =0; i<1001; i++)
                        bits.set(i, i%2==0)
                for(int i=0; i<1001;i++)
                        System.out.print(bits.get(i)? "1" : "0");
                System.out.println();
        }
}
```

Metodi & attributi statici

- I metodi o attributi statici sono associati all'intera classe e non al singolo oggetto
- Gli attributi statici sono delle variabili globali e i metodi statici sono delle funzioni

```
class Global {
        private static int time = 100;
        public static void setTime(int t) {
                time = t;
        }
        public static int getTime() {
                return time;
        }
}
...
Global.setTime(Global.getTime()+1);
```


37

Gerarchia delle classi

- Una classe può essere derivata da un'altra classe estentendone le caratteristiche (attributi o metodi)
- Java usa ereditarietà singola, quindi si può estendere da una sola classe. La sintassi è:

```
class <nome_classe> [extends <nome_classe>] {
    ...
}
```

- Se omesso *extends* la classe viene derivata dalla classe predefinita *Object*

38

Ereditarietà

- La classe derivata possiede tutte le caratteristiche della classe padre.
- A questa sono aggiunte le caratteristiche specifiche
- Es.

```
class Persona {
    public String nome, cognome;
    ...
}
class Studente extends Persona {
    public String nmatricola;
    ...
}
```


Erediterietà, Uso

```
...
Studente s = new Studente();
s.nome = "Paolo";
s.cognome = "Rossi";
s.nmatricola = "12345678";
...
Persona p = s;
System.out.println(p.nome+" "+p.cognome);
```

Visibilità protected

- Attributi e metodi dichiarati con visibilità **protected** sono accessibili dalle classi derivate **e dalle classi dello stesso package**.

Ereditarietà & costruttore

```
public class Persona {
    protected String nome, cognome;

    public Persona(String nome, String cognome) {
            this.nome = nome;
            this.cognome=cognome;
    }
    ...
}
```

Ereditarietà & costruttore

```
class Studente extends Persona {
    private String nmatricola;
    public Studente(String nm, String cg, String mt) {
            super(nm,cg);
            nmatricola = mt;
    }
    ...
}
```

Chiama il costruttore della classe Persona

Polimorfismo

- I metodi sono tutti implicitamente polimorfici se vengono ridefiniti nelle classi derivate

```
class Persona {
        ...
        public void print() {
                System.out.println(nome+" "+cognome);
        }
}
class Studente extends Persona {
        ...
        public void print() {
                super.print();
                System.out.println(nmatricola);
        }
}
```

Chiama metodo print della classe padre

Polimorfismo

```
Persona p = new Studente("mario", "rossi", "12345");

p.print();
//chiama Studente.print() e non Persona.print()
```


@Override

- Quando si fa override di un metodo in una classe derivata si può aggiungere l'annotazione opzionale *@Override*

```
class Studente extends Persona {
 ...
 @Override
 public void print() {
 ...
 }
}
```

- Rendendo esplicita l'intenzione di ridefinire un metodo il compilatore controlla se esiste il metodo nella classe padre ed è compatibile (stessi tipi di parametri) e in caso non ci sia fallisce (senza @Override, ce ne saremmo accorti solo all'esecuzione)

Classi astratte

- Sono classi che non possono avere istanze dirette, si possono solo derivare altre classi.
- Esempio: Shape

```
public abstract class Shape {
    protected int x;
    protected int y;
    public abstract void draw();
}
```

metodo astratto senza implementazione, andrà implementato nelle classi derivate

Classi astratte

```
public class Rectangle extends Shape {
        private int width;
        private int height;
        public Rectangle(int x, int y, int w, int h) {
                this.x=x; this.y=y;
                this.width = w; this.height = h;
        }
        public void draw() {
                ...
        }
}
```

Classi astratte

```
public class Circle extends Shape {
        private int radius;
        public Circle(int x, int y, int r){
                this.x=x; this.y=y;
                this.radius = w;
        }
        public void draw(){
                ...
        }
}
```


49

Esempio: CompositeShape

```
public class CompositeShape extends Shape {
        private Shape[] shapes;
        private int n;
        public CompositeShape(int nshape){
                this.n = 0;
                this.shapes = new Shape[nshape];
        }
        public CompositeShape add(Shape s){
                this.shapes[n++]=s;
                return this;
        }
        public void draw(){
                for(int i=0; i<n; i++)
                        this.shapes[i].draw();
        }
}
```

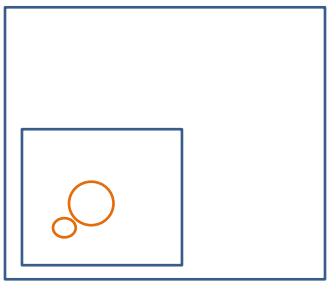

50

Uso CompositeShape

```
CompositeShape s1 = new CompositeShape(10);
s1.add(new Rectangle(10,10, 100,100));
s1.add(new Rectangle(20,20, 50, 50));
CompositeShape s2 = new CompositeShape(3);
s2.add(new Circle(50,50,5));
s2.add(new Circle(60,60,15));
s1.add(s2);
s1.draw();
```


51

CompositeShape



52

Interfacce

- Java non permette ereditarietà multipla per problemi dovuti a gestione attributi nell'ereditarietà a diamante
- Ma permette di definire delle «interfacce»
  - classi che definiscono solo funzionalità.
  - le interfacce danno la possibilità di definire l'insieme delle funzionalità che devono essere implementate per poter interagire con un insieme di classi
  - i metodi sono implicitamente astratti e pubblici
  - possono avere metodi e attributi statici
  - possono essere derivate da altre interfacce
- **Una classe può essere derivata da più interfacce**

Interfacce

- Sintassi:

[**public**] **interface** <Nome> [**extends** <Nome>[,<Nome>,...<Nome>]] {
    <metodi astratti pubblici>
    <attributi statici>
    <metodi statici>
}

[**public**] **class** <Nome> [**extends** <Nome>] [**implements** <Nome>[,<Nome>...]] {
    ...
}

- Una classe che implementa una o più interfacce deve definire tutti i metodi astratti delle interfacce e di tutte le interfacce da cui eventualmente derivano

Esempio

```
public interface MovableObject {
        void move(int dx, int dy);
}

class Rectangle extends Shape
        implements MovableObject, Cloneable {
...
        public void move(int x, int y) {
                ...
        }
}

MovableObject mo = new Rectangle(...)
mo.move(10,20);
```

Metodo astratto e pubblico

Cast

- L'operatore di cast serve a trasformare una istanza di un tipo in altro tipo analogo ma meno preciso. Per i tipi base per esempio:

  long l = 10;

  int a = (int) l;

- E' necessario un cast nella traformazione da int a byte, da int a short, da double a float etc.

Cast & instanceof

- Una reference ad una classe padre può essere trasformata in una reference a classe figlia tramite il cast.

```
Persona p = new Studente();
...
Studente s = (Studente) p;
```

- Ok se p effettivamente punta a una istanza che è Studente o sua derivata, ma se ciò non è vero viene generata eccezione *ClassCastException*
- Questo è necessario quando si vuole usare un metodo della classe derivata

Cast & instanceof

- E' possibile anche controllare se una reference punta a una istanza di una classe (o sua derivata) tramite operatore **instanceof**
- x **instanceof** Y è vero se la reference *x* punta a un oggetto della classe Y o a una sua derivata.

```
Studente s = null;
if(p instanceof Studente)
        s = (Studente) p;
```

final

- Il modificatore **final** viene usato per indicare una variabile, un attributo o anche un metodo non più modificabile.

```
final int A = 1000;
final float PI;
PI = 3.14; //da questo punto in poi PI non è modificabile
```

- un attributo final, static e public viene usato in una classe per indicare una costante.
- Un metodo final indica un metodo che non può essere ridefinito in una classe figlia.
- Si può avere anche una classe final da cui non è possibile derivare altre classi.

Package

- I package servono a raggruppare classi semanticamente collegate, questo per gestire la complessità quando il numero di classi è elevato
- I package sono usati nelle API Java per organizzare la quantità di classi presenti
  - in Java 8 ci sono 217 package con 4240 classi
- https://docs.oracle.com/javase/8/docs/api/

Package di Java API

- **java.io** Provides for system input and output through data streams, serialization and the file system.
- **java.lang** Provides classes that are fundamental to the design of the Java programming language.
- **java.lang.annotation** Provides library support for the Java programming language annotation facility.
- **java.lang.instrument** Provides services that allow Java programming language agents to instrument programs running on the JVM.
- **java.lang.invoke** The java.lang.invoke package contains dynamic language support provided directly by the Java core class libraries and virtual machine.
- **java.lang.management** Provides the management interfaces for monitoring and management of the Java virtual machine and other components in the Java runtime.
- **java.lang.ref** Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
- **java.lang.reflect** Provides classes and interfaces for obtaining reflective information about classes and objects.
- **java.math** Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
- **java.net** Provides the classes for implementing networking applications.

Altri package di Java API

- **java.awt** Contains all of the classes for creating user interfaces and for painting graphics and images.
- **java.awt.event** Provides interfaces and classes for dealing with different types of events fired by AWT components.
- **java.awt.font** Provides classes and interface relating to fonts.
- **java.awt.geom** Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
- **java.awt.im** Provides classes and interfaces for the input method framework.
- **java.awt.im.spi** Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
- **java.awt.image** Provides classes for creating and modifying images.
- **java.awt.image.renderable** Provides classes and interfaces for producing rendering-independent images.
- **java.awt.print** Provides classes and interfaces for a general printing API.
- …

Usare package

- Per usare una classe di un package, o ci si riferisce al nome completo, es:
  - **java.net.InetAddress** ip = **java.net.InetAddress**.getLocalHost();
- Oppure si usa costrutto import all'inizio del file:

```
import java.net.InetAddress;
...
    InetAddress ip = InetAddress.getLocalHost();
```


Usare package

- Si possono anche importare tutte le classi di un package, es:

```
import java.net.*;
...
InetAddress ip = InetAddress.getLocalHost();
```

- Se un package ha dei subpackages con .* NON si importano anche le loro classi ma ogni subpackage va importato separatamente, es:

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
```

Definire un package

- Per il nome del package di solito si usa:
  - il dominio DNS rovesciato della ditta/istituzione,
  - seguito dal nome della applicazione/libreria
  - e quindi dal nome della sotto parte in cui è organizzata l'applicazione.
- esempi:
  - org.apache.commons.collections4.multimap
  - it.unifi.myproject.db (per le classi di gestione database)
  - it.unifi.myproject.ui (per le classi di gestione interfaccia utente)

65

Definire un package

- I file .java devono essere organizzati in una struttura delle directory seguendo il nome del package. Per esempio la classe *it.unifi.myproject.db.ImportData* deve trovarsi in:
  - *it/unifi/myproject/db/ImportData.java*
- inoltre nelle prime righe del file deve essere presente la dichiarazione di appartenenza al package:

```
package it.unifi.myproject.db;
...
public class ImportData ... {
        ...
```

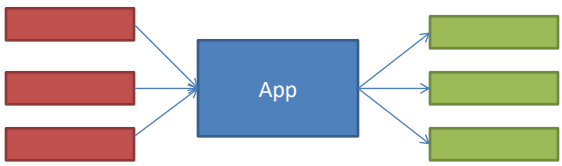

66

visibilità e package

- Quando il modificatore di visibilità (public/protected/private) di una classe/attributo/metodo è omesso la visibilità è *package-private* cioè è visibile solo alle classi che fanno parte dello stesso package.

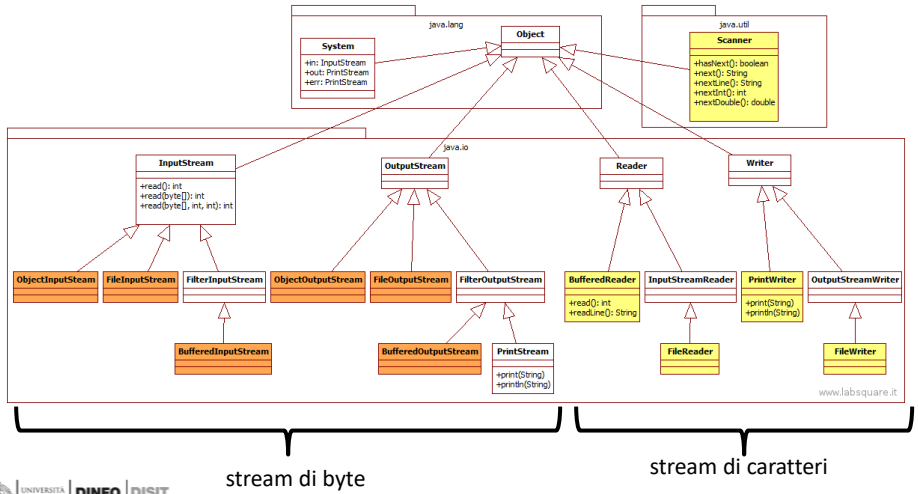
Streams

- Gli stream sono oggetti che permettono ad una applicazione di acquisire dati *(stream di input)* o di inviare dati *(stream di output)*



- I dati vengono **prelevati** (pull) da input stream e **inviati** (push) a output stream
- Java fornisce un insieme di classi per la gestione degli stream

Streams



69

FileInputStream

FileInputStream f = new **FileInputStream**("file.bin");
int b = **f.read**(); // legge un byte dal file
// ritorna -1 se raggiunto fine del file
byte[] bb = new byte[100];
int n = **f.read**(bb); // legge in bb al più 100 byte
//n indica quanti letti (-1 = fine file)
int n = **f.read**(bb, offset, len);
f.close(); //chiude file

70

FileOutputStream

- Simmetrico di FileInputStream

FileOutputStream f = new **FileOutputStream**("file.bin");
byte b = 13;
f.write(b); // scrive un byte sul file
byte[] bb = new byte[100];
...
f.write(bb); // scrive su file i 100 byte in bb
f.write(bb, offset, len);
f.close(); //chiude file

PrintStream

OutputStream os = new **FileOutputStream**("file.out");
PrintStream p = new **PrintStream**(os);

p.print("a");
p.println("hello!");

FilterInput/OutputStream

- Le classi **FilterInputStream** e **FilterOutputStream** sono usate per filtrare uno stream di input o output
- la classe PrintStream è derivata da FilterOutputStream
- come le classi **BufferedInputStream** e **BufferedOutputStream**

InputStreamReader

```
InputStream is = new FileInputStream("file.txt");
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
int c = br.read() //legge un carattere (16 bit)
char[] s=new char[100]
int n = br.read(s); //legge al più 100 caratteri e
                    // ritorna quanti caratteri letti
String line = br.readLine(); //legge una riga (fino a \n)
                    //ritorna riga senza "a capo" o
                    //null se file finito
```

Copy file

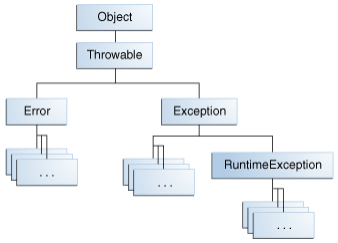
```
static void copy(InputStream is,OutputStream os) throws IOException {
        int c;
        while((c=is.read()) != -1) {
                os.write(c);
        }
}

FileInputStream inf = new FileInputStream("c:/tmp/file.txt");
FileOutputStream outf = new FileOutputStream("c:/tmp/file-copy.txt");
copy(inf,outf);
inf.close();
outf.close();

FileInputStream inf2 = new FileInputStream("c:/tmp/file-copy.txt");
copy(inf2,System.out);
inf2.close();
```


Eccezioni

- Le eccezioni vengono lanciate (throw) durante l'esecuzione del programma per indicare una condizione errata che non può essere risolta.
- Le eccezioni possono essere gestite per risolvere il problema o per dare indicazione di errore e continuare con operazione successiva.
- Le eccezioni se non gestite (catch) bloccano l'esecuzione dell'intero programma

Eccezioni

- Le eccezioni sono rappresentate da degli oggetti derivati dalla classe di sistema Throwable o meglio dalle classi Exception, Error e RuntimeException.
- Exception
  - Eccezioni conosciute, prevedibili e documentate (es. FileNotFoundException)
- Error
  - Sono errori imprevedibili generati nell'uso della classe (es. IOError)
- RuntimeException
  - Sono errori imprevedibili dovuti ad un errore logico nella applicazione (es. NullPointerException)



Try...catch

```
try {
        ... istruzioni...
}
catch(Classe exc) {
        ... istruzioni...
}
[finally {
        ... istruzioni ...
}]
```

in queste istruzioni o nei metodi richiamati può essere lanciata una eccezione

queste istruzioni gestiscono una eccezione di tipo Classe o una sua derivata, si possono avere più sezioni catch su eccezioni diverse

queste istruzioni eseguite comunque sia che venga venga lanciata eccezione gestita che non gestita. Serve a rilasciare risorse eventualmete acquisite di cui si deve garantire il rilascio.

Esempio

```
try {
        int[] v = new int[10];
        ...
        v[10] = 10;
        ...
} catch(IOException e) {
        e.printStackTrace();
} catch(Exception e) {
        e.printStackTrace();
        System.out.println("msg:"+e.getMessage());
} finally {
        System.out.println("finally!");
}
```

Genera eccezione ArrayIndexOutOfBoundsException

Gestisce l'eccezione

throws

- Le eccezioni che non sono *Error* o *RuntimeException* (dette *checked exceptions*) devono essere gestite o dichiarate nella firma del metodo/costruttore che possono essere lanciate (throws), il chiamante dovrà gestirle o dichiararle nel suo throws

```
class ... {
        ...
        public void method(...) throws ...lista eccezioni... {
                ...
        }
}
```

- Se non viene fatto il compilatore genera errore

Esempio throws (1)

```
class EsempioInputStream {
    int chkFile() throws IOException {
        InputStream is=new FileInputStream("input.bin");
        int c=0, chk=0;
        try {
            while((c=is.read()) != -1) {
                chk ^=c; //xor bit-wise
            }
        } finally {
            is.close();
        }
        System.out.println("chk: "+chk);
        return chk;
    }
```

throws FileNotFoundException derivata da IOException

throws IOException

viene chiuso lo stream anche se viene generata una eccezione durante la lettura

81

Esempio throws (2)

```
class EsempioInputStream {
    int chkFile() throws IOException {
        InputStream is;
        try {
            is = new FileInputStream("input.bin");
        } catch(FileNotFoundException ex) {
             is = new FileInputStream("input2.bin");
        }
        int c, chk=0;
        try {
            while((c=is.read()) != -1)   chk ^=c;
        } finally {
            is.close();
        }
        System.out.println("chk: "+chk);
        return chk;
    }
```


82

Lanciare eccezioni

- l'istruzione **throw** serve a lanciare una eccezione, es:

  throw new IllegalArgumentException("parametro errato");
- L'oggetto lanciato deve essere di una sottoclasse di Throwable

Definire nuove eccezioni

- Si possono definire nuove eccezioni, definendo una sottoclasse di Exception o RuntimeException.
- Per comprensibilità il nome della classe dovrebbe terminare in Exception
- Il costruttore può prendere come parametro il messaggio associato (se necessario)

Esempio

```
public class MyException extends Exception {
        public MyException() {
                super();
        }
        public MyException(String msg) {
                super(msg);
        }
}

… throw new MyException("errore…");
```


Input da console

- Per chiedere in input da console si può usare classe **Scanner**

```
import java.util.Scanner;
...
Scanner s = new Scanner(System.in);
int x = s.nextInt(); //legge un intero
float f = s.nextFloat(); //legge un float
String str = s.nextLine(); //legge una linea
```

Esempio

```
Scanner s=new Scanner(System.in);
int x;
do {
   try {
      System.out.print("x [1-5]: ");
      x=s.nextInt();
   }
   catch(InputMismatchException e) {
      x=0;
      s.nextLine();   //estrae l'input errato
      System.out.println("input non valido");
   }
} while (x<1 || x>5);
```


foreach

- Aggiunto in Java 5
- Permette di iterare su elementi di un array o di una collection
- esempio:

```
float[] v = new float[n];
...
for(float x : v) {
    System.out.println(x);
}
```

```
for(int i=0; i<v.length; i++) {
      float x = v[i];
      System.out.println(x);
}
```

Tipi base e Object

- Per ogni tipo di base è presente una classe derivata da **Object** (detta wrapper class) che permette di utilizzare classi generiche per memorizzare sia oggetti che tipi di base
  - **int → Integer**, **byte → Byte**, **short→ Short**
  - **float → Float**, **double → Double**
  - **char → Character**
  - **boolean → Boolean**

89

Boxing, Unboxing

- Il compilatore automaticamente trasforma un tipo di base in una istanza della wrapper class corrispondente (Boxing) e viceversa (Unboxing)

```
Integer x = 3;   //boxing
                 Integer x = new Integer(3);
int y = x*2;     //unboxing
                 int y = x.intValue() * 2;
```


90

Java Collection Framework (cenni)

- Insiemi di interfacce e classi per la gestione di insiemi di oggetti (liste, mappe, code, insiemi,...)
- estesi dai *generics* introdotti in Java 5 (simili a template C++)
- List<*Type*> x = new ArrayList<>();
- List<*Type*> x = new LinkedList<>();
- *Type* può essere solo una classe derivata da Object (non un tipo di base)
- Si accede agli elementi tramite iteratori oppure usando costrutto *foreach*

Esempio

```
import java.util.LinkedList; //classe concreta
import java.util.List; //interfaccia
...
List<Integer> lst = new LinkedList<>();
lst.add(1);
lst.add(2);
lst.add(3);
System.out.println(lst);  //[1, 2, 3]
lst.remove(1); //rimuove elemento in posizione 1
for(int y: lst){
        System.out.println(y);
}
```

Esempio iteratore

```
//forma equivalente al foreach
Iterator<Integer> i = lst.iterator();
while(i.hasNext()) {
        int y = i.next();
        System.out.println(y);
}
```