

Introducing Java

Java: Classes, Encapsulation, Visibility Modifiers, Packages



Introducing Java

A Java class is typically organized this way:

```
class ClassName {  
    <attributes – “variables”>;  
    ClassName(ParameterList){  
        <constructor body>;  
    }  
    returnType method1(ParameterList) {  
        <method body>;  
    }  
    ...  
    returnType methodN(ParameterList) {  
        <method body>;  
    }  
}
```

Java Classes (I)



Java Classes (II)

Una classe Java deve essere memorizzata in un file che si deve chiamare *nome_della_classe.java*

Per convenzione

- il nome di una classe ha la prima lettera maiuscola
- gli identificatori delle variabili e i nomi dei metodi cominciano con una lettera minuscola
- se gli identificatori sono composti da più parole quelle successive alla prima hanno tutte la prima lettera maiuscola

Es:

- NomeDiUnaClasse
- nomeDiVariabile
- nomeDiUnMetodo



Example: the Class "Motorbike"

```
class Motorbike {  
    String color;  
    String model;  
    int fuelLevel;  
}  
  
public Motorbike(String mod, String col) {  
    model=mod;  
    color=col;  
    fuel=100;  
}  
  
void refill () {  
    if(fuelLevel==100)  
        System.out.println("Already Full");  
    else {  
        fuelLevel=100;  
        System.out.println("Refilled!");  
    }  
    ...  
}
```

Member variables
Attributes and state of an object

Constructor
It initializes the object state at creation

Member functions, or Methods
They define the possible operations for the object



Variables and Methods

Variables and methods can be accessed using the dot notation:

- objName.varName
- objName.methodName(paramList)

If "myM" is an object of type Motorbike, its members can be accessed in the following way:

```
myM.model="Honda";  
myM.fuelLevel=50;  
myM.refill();
```

Methods always have (),
also in case
they do not take any parameter



Methods

Methods return a value:

```
returnedType methodName(paramList) {  
    <method body>  
}
```

Example:

```
int leftFuel () {  
    return fuelLevel;  
}
```



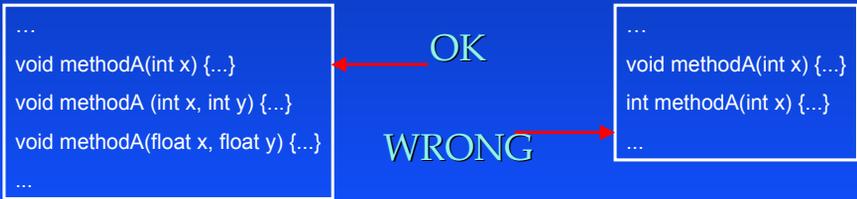
The "void" keyword must be used whenever no value is returned:

```
void refill() {  
    ...  
}
```



Method Overloading

Different methods may hold the same name
They can be distinguished because of their signature:
– *Parameters: different in number, type and/or order*
This mechanism is known as **overloading**
It's not possible to tell apart two methods by the returned type only



Object Creation (1)

Object are class instances.
Let's create an object of class Motorbike:

```
Motorbike m;  
m=new Motorbike("Gilera","red");
```

Here the variable m is declared as a reference to an object of type Motorbike

A new object of type Motorbike is created, and it will be referred through the variable m



Object Creation (2)

What happens in memory?

Instruction

```
Motorbike m;
```

```
m=new Motorbike("Gilera","red");
```

Within memory:

m

null

m

•

Ducati
blu
100

The "new" keyword allocates in memory the proper space for a new object



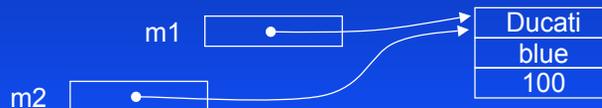
Objects and References

```
Motorbike m1=new Motorbike("Ducati","blue");
```

```
Motorbike m2;
```

```
m2=m1;
```

- The first code line declares a reference m1, creates the object "blue Ducati", and makes the variable refer to it
- The second code line declares a reference m2 to a "Motorbike" object, and it is automatically initialized to "null" by default.
- The third code line makes m2 refer the same object as m1



In the current Java programming jargon, references to objects are called "objects" too.



Constructor (1)

It's a special method used for the object initialization, done immediately after its creation

The constructor method is named exactly like the class, and its general format is this:

```
ClassName (paramList) {...}
```

The constructor does not return any value, and in this case also the void keyword must be omitted

A generic class may have:

- 1 No explicit constructor
(in this case, it's implicit, and it's inherited by the parent class: all the Java classes implicitly derive from java.lang.Object)
- 2 Just one constructor
- 3 Multiple constructors (overloading)



Constructor (2)

- 1 No constructor (implicit):

```
class Rectangle {  
    int x1;  
    int y1;  
    int x2;  
    int y2;  
}
```

Rectangle rect = new Rectangle();

The obj rect is initialized by default with
x1=0, y1=0, x2=0, y2=0

- 2 One constructor:

```
class Rectangle {  
    int x1;  
    int y1;  
    int x2;  
    int y2;  
  
    Rectangle(int a, int b, int c, int d) {  
        x1=a;  
        y1=b;  
        x2=c;  
        y2=d;  
    }  
}
```

Rectangle rect = new Rectangle(1,1,3,4);

The obj ret is initialized with
x1=1, y1=1, x2=3, y2=4



Constructor (3)

3 Multiple constructors:

```
class Rettangolo {  
    int x1;  
    int y1;  
    int x2;  
    int y2;  
  
    Rettangolo(int c, int d) {  
        x1=0;  
        y1=0,  
        x2=c;  
        y2=d;  
    }  
  
    Rettangolo(int a, int b, int c, int d) {  
        x1=a;  
        y1=b;  
        x2=c;  
        y2=d;  
    }  
}
```

```
Rettangolo ret=new Rettangolo(1,1);
```

```
Rettangolo ret2=new Rettangolo(1,2,3,4);
```

Constructor (4)

A constructor can invoke another constructor using the keyword *this* (it's used instead of the constructor name), with the proper parameters: `this(arg1, arg2, ...)`

```
class Rettangolo {  
    int x1;  
    int y1;  
    int x2;  
    int y2;  
  
    Rettangolo(int c, int d) {  
        x1=0;  
        y1=0,  
        x2=c;  
        y2=d;  
    }  
  
    Rettangolo(int a, int b, int c, int d) {  
        x1=a;  
        y1=b;  
        x2=c;  
        y2=d;  
    }  
}
```

These ones
are
equivalent

```
class Rettangolo {  
    int x1;  
    int y1;  
    int x2;  
    int y2;  
  
    Rettangolo(int c, int d) {  
        x1=0;  
        y1=0,  
        x2=c;  
        y2=d;  
    }  
  
    Rettangolo(int a, int b, int c, int d) {  
        this(c,d);  
        x1=a;  
        y1=b;  
    }  
}
```





Object Destruction (1)

In Java there exists no destructor method, as in C++
Thus, how to get the memory space occupied by objects that are no more used?

A system thread, called *Garbage Collector*, is in charge of this task.

But... When an obj can be removed from memory?
Exactly when is no more reachable by any reference.



Object Destruction(2)

Example:

```
Motorbike moto;  
moto=new Motorbike("Ducati","blue");  
moto=new Motorbike("Honda","green");
```

Now, the first object (the blue Ducati)
cannot be accessed anymore.

The Garbage Collector can free
the memory area occupied by the blue Ducati.



Object Destruction : the finalize() method

We must remember that:

1. *Objects might not get garbage collected*
2. *Garbage collection is not destruction*
3. *Garbage collection is only about memory*

Thus, the additional cleanup activities can be placed in a special final method called `finalize()` (not mandatory)



Inner Classes

A class can contain the definition of another class

As usual, the class `Extern` must be defined in a file named *Extern.java*

By compiling such a file, two bytecode files are generated: *Extern.class* and *Extern\$Intern.class*

```
class Extern {  
    class Intern {  
        int i=0;  
        void inc () {  
            i++;  
        }  
        Intern n;  
        ...  
    }  
}
```



Gli array di oggetti (1)

In Java gli array sono oggetti

Si possono dichiarare sia array di tipi primitivi che array di oggetti

```
int[] a;
```

Questa dichiarazione indica che **a** è un array di interi

In effetti **a** è un riferimento ad un oggetto di tipo "array di interi"
e come accade per gli altri oggetti esso va creato

```
int[] a=new int[10];
```

Adesso **a** riferisce un array di 10 interi

La lunghezza di un array è contenuta in una variabile membro

```
a.length
```

mi dice quale è la dimensione dell'array (in questo caso 10)



Gli array di oggetti (2)

Con una sintassi analoga è possibile creare array di oggetti

```
Motocicletta[] motoArr=new Motocicletta[10]
```

In questo modo abbiamo creato un array di 10 riferimenti ad
oggetti di tipo Motocicletta (*NB: non un array di oggetti
Motocicletta*)

Tali riferimenti hanno al momento della creazione tutti il valore null

A questo punto è possibile

assegnare oggetti di tipo Motocicletta a tali riferimenti

```
motoArr[3]=new Motocicletta("Ducati", "blu");
```

```
motoArr[4]= un oggetto già esistente
```

*Un array di tipi primitivi contiene direttamente il loro valore
un array di oggetti contiene i riferimenti agli oggetti*



Gli array di oggetti (3)

In fase di dichiarazione è possibile utilizzare le parentesi graffe per specificare gli elementi dell'array

```
int[] vet={1, 2, 3 ,4}
```

Crea un array di 4 interi inizializzati con tali valori

```
Motocicletta[] motoArr={new Motocicletta("Ducati", "blu"),  
                          new Motocicletta("Honda", "verde")}
```

Crea un array di due riferimenti ad oggetti di tipo Motocicletta e li fa puntare ai due oggetti creati

Modifiers



Java provides a number of "modifiers," that are used to change the behavior and the visibility of class members (both variables and methods)

Access (visibility) modifiers:

- **public**
- **protected**
- **private**

The modifier **static** applies to class methods and variables

The modifier **abstract** is used to define abstract methods and classes

The modifier **final** applies to class, methods and variables

The modifiers **synchronized** and **volatile** will be discussed later (related to threads and I/O)

The modifier **native** is used for native methods



Access Control (1)

I modificatori **public**, **protected** e **private** consentono di controllare il livello di accesso a variabili e metodi di una classe

- Variabili e metodi dichiarati **public** possono essere acceduti da una qualunque altra classe
- Variabili e metodi dichiarati **protected** possono essere acceduti dai membri della classe (variabili e metodi della stessa classe), dalle sottoclassi e dalle classi del package (vedremo come più classi possano essere raggruppate in package)
- Variabili e metodi dichiarati **private** possono essere acceduti solo all'interno della classe nella quale sono stati dichiarati

Le sottoclassi non possono accedere a dati o metodi dichiarati **private** in una superclasse



Access Control (2)

In case no access modifier is used, the access level for variables and methods is the following:

package level, a.k.a. **friendly** level

Variables and methods in the **package** level can be accessed from all the other classes in the same package

The **package** protection level is the default one



Access Control (3)

Quando un metodo o una variabile vengono dichiarati **public** divengono visibili da una qualunque altra classe Java

The class `Counter` belongs to package `x` and it defines a variable `x` whose protection level is the *friendly* one, and a **public** method `inc()`

```
package x;
public class Counter{
    int c;
    public Counter(){
    }
    public void inc(){
        c++;
    }
}
```



Access Control(4)

The classes `Alien` and `Counter` don't belong to the same **package**

`Alien` può comunque invocare il metodo `inc()` su un oggetto di tipo `Contatore` in quanto tale metodo era stato dichiarato **public**

`Alien` non può accedere alla variabile `C` della classe `Contatore` in quanto per default essa ha un livello di accesso *friendly*

```
package y;
public class Alien{
    Counter i=new Counter();
    void action(){
        i.inc(); // Ok
        i.c++; // Error
    }
}
```



Access Control(5)

Il livello *protected* è una via di mezzo tra i livelli *private* e *package*

Le variabili ed i metodi *protected* sono accessibili a tutte le classi dello stesso package e alle sottoclassi esterne

Questo livello risulta particolarmente utile nei metodi e nelle variabili specifici dell'implementazione, ma che potrebbero essere utili anche nelle sottoclassi



Access Control(6)

- Il livello di accesso *private* è quello più restrittivo
- Una variabile istanza o un metodo di tipo *private* sono visibili solo nella classe all'interno della quale sono stati definiti

```
class Employee {  
    private float salary;  
    private String address;  
    private void moreSalary(float x) {  
        salary=salary*(1+x);  
    }  
    public void inflation(float i) {  
        this.moreSalary(i);  
    }  
    ...  
}
```

Subclasses are not allowed to access *private* members of their superclasses



Access Control(7)

Una classe che dichiara un oggetto di tipo Dipendente non può accedere alle variabili salario e indirizzo, ne può invocare il metodo aumentaSalario()

```
class Employee {  
    private float salary;  
    private String address;  
    private void moreSalary(float x) {  
        salary=salary*(1+x);  
    }  
    public void inflation(float i) {  
        this.moreSalary(i);  
    }  
    ...  
}
```

```
Employee d=new Employee();  
d.salary++; // Error  
d.moreSalary((float)10); //Error  
d.inflation((float)0.01); // Ok
```

Access Control(8)

I dati ed i metodi che devono rimanere interni alla classe e non devono essere visibili dall'esterno devono essere dichiarati *private*

```
class Dipendente {  
    private float salario=100;  
    private String indirizzo;  
    private void aumentaSalario(float x) {  
        salario=salario*(1+x);  
    }  
    public void inflazione(float i) {  
        this.aumentaSalario(i);  
    }  
    ...  
}
```

Ma in questo caso come è possibile accedere ai dati privati?

Ad esempio come è possibile accedere alla variabile indirizzo della classe Dipendente?

Si devono fornire dei metodi per regolamentare l'accesso ai dati





Access Control(9)

In questo caso è possibile leggere e modificare la variabile indirizzo attraverso i due metodi `getIndirizzo()` e `setIndirizzo()`

```
Dipendente d=new Dipendente();  
d.indirizzo="v.Verdi 32"; //Errore  
d.setIndirizzo("v.Verdi 32"); //OK
```

```
class Dipendente {  
    private float salario=100;  
    private String indirizzo;  
    private void aumentaSalario(float x) {  
        salario=salario*(1+x);  
    }  
    public void inflazione(float i) {  
        this.aumentaSalario(i);  
    }  
    ...  
    public void setIndirizzo(String ind){  
        indirizzo=ind;  
    }  
    public String getIndirizzo(){  
        return indirizzo;  
    }  
}
```



Access Control(10)

I metodi di accesso non fanno nessun controllo, anche se non è possibile accedere direttamente alla variabile privata, è comunque possibile leggerne il valore (`getIndirizzo()`) e modificarla (`setIndirizzo()`)

Il vantaggio deriva dal fatto che l'implementazione rimane nascosta (in questo caso è banale) mentre i metodi di accesso costituiscono l'interfaccia dell'oggetto con il mondo esterno (**incapsulamento dei dati**)

Nel caso che debba essere cambiata l'implementazione interna, se l'interfaccia non viene modificata, non è necessario apportare modifiche ad altre classi che usano quella modificata



Access Control(11)

Summary of protection levels

Visibilità	Public	Protected	Package	Private
Dalla stessa classe	si	si	si	si
Da classi dello stesso package	si	si	si	no
Da classi al di fuori del package	si	no	no	no
Da sottoclassi nello stesso package	si	si	si	no
Da sottoclassi al di fuori del package	si	si	no	no

The *static* modifier (1)

Variabili static

Le variabili dichiarate *static* sono anche dette *variabili di classe*

Una variabile *static* è comune a tutti gli oggetti istanza della stessa classe

A differenza delle altre non ne esiste una copia per ogni oggetto, ne esiste una sola copia comune a tutte le istanze

Ciò comporta che se un oggetto (istanza della classe) modifica il valore di una variabile *static*, gli altri oggetti vedranno il valore modificato

Such static variables can be used to:

- Make different instances of the same class communicate among each other
- Keep a global state, shared by all the instances





The *static* modifier(2)

```
class Foo {  
    static int genCount=0;  
    int localCount=0;  
    Pippo() {  
        genCount ++;  
        localCount ++;  
    }  
}
```

Supponiamo di avere il seguente codice:

```
...  
Foo p1=new Foo();  
Foo p2=new Foo();
```

Now:

p1.genCount e **p2.genCount** hanno lo stesso valore (uguale a 2),
mentre le due variabili istanza **localCount** hanno valore 1

Nel nostro caso si può accedere alla variabile **contGenerale** sia con **p1.contGenerale** e **p2.contGenerale**, che con **FooFoo.contGenerale** (direttamente con il nome della classe)

Questo perché è possibile accedere alle variabili **static** senza dover utilizzare una specifica istanza, ma utilizzando direttamente il nome della classe



The *static* modifier(3)

Metodi static

I metodi dichiarati **static** sono anche detti metodi di classe

I metodi dichiarati **static** possono essere richiamati indipendentemente dall'esistenza di un istanza della classe

I metodi **static** non possono accedere alle variabili della classe che non sono dichiarate anch'esse **static**

Regola:

I metodi che operano su uno specifico oggetto non devono essere dichiarati **static**, mentre quelli che sono di utilità generale e che non agiscono sulle singole istanze, devono essere dichiarati **static**



```
class Contatore {  
  
    static int cont1=0;  
    int cont2=0;  
  
    static void inc1() {  
        cont1++;  
    }  
    static void inc2(){  
        cont2++;  
    }  
}
```

Error

The *static* modifier (4)

La variabile `cont1` è **static** quindi ne esiste una sola, comune a tutte le istanze della classe `Contatore`

Il metodo `inc2()` contiene un errore in quanto è dichiarato **static** e cerca di modificare una variabile **non-static**



Il modificatore *final* (1)

L'utilizzo del modificatore **final** su classi, variabili e metodi ha i seguenti effetti:

- Se una classe è dichiarata **final** allora da questa non possono essere create delle sottoclassi
- Il valore di una variabile dichiarata **final** non può essere cambiato (è una costante)
- Un metodo dichiarato **final** non può essere ridefinito dalle sottoclassi

In pratica il modificatore **final** blocca l'implementazione o il valore di classi, metodi o variabili



Il modificatore *final* (2)

Classi finalizzate

Ci sono due motivi per dichiarare una classe *final*

- Vietare ad altri di creare sottoclassi
- Aumentare l'efficienza (in quanto il compilatore sa che non possono esserci delle sottoclassi che ridefiniscono i metodi di una classe *final*)

Molte classi di sistema sono dichiarate *final* per motivi di sicurezza ed efficienza

```
final class classeA {  
    ...  
}  
class classeB extends classeA {  
    ...  
}
```

Error during
compilation



Il modificatore *final* (3)

Variabili finalizzate

Una variabile dichiarata *final* ha un valore che non cambia con il tempo ed quindi è una costante

Nel caso di costanti il cui valore sia noto in fase di compilazione, il compilatore sostituisce la costante con il suo valore

In Java possono appartenere a questa categoria di costanti solo i tipi primitivi

Per convenzione le costanti note a tempo di compilazione hanno un identificatore composto da lettere tutte maiuscole



Il modificatore *final* (4)

Variabili finalizzate

Quando il modificatore **final** è utilizzato per un riferimento ad un oggetto, allora:

- al momento della dichiarazione al riferimento deve essere assegnato un oggetto;
- il riferimento non potrà in seguito puntare un altro oggetto

È possibile però modificare l'oggetto riferito

Non esiste un modo per creare oggetti costanti

(quindi non possono essere creati array costanti in quanto gli array sono oggetti)



Il modificatore *final* (5)

Variabili finalizzate

```
class Dati {  
  
    final int C1=1;  
    static final int C2=2;  
    public static final C3=3;  
  
    final int r1=(int) Math.random()*100;  
  
    Valore v1=new Valore();  
    final Valore v2=new Valore();  
    static final Valore v3=new Valore();  
  
    final int[] arr={7,8,9};  
}
```

Valori noti a tempo di compilazione

```
Dati d=new Dati();  
d.C1++; // Errore  
d.C2++; // Errore  
d.C3++; // Errore
```

Errori

Valore noto solo a run-time

```
Dati d=new Dati();  
d.r1++; // Errore
```

Errori

Oggetti

```
Dati d=new Dati();  
d.v2=new Valore(); // Errore  
d.v3=new Valore(); // Errore  
d.v2.variabilediValore++; // Ok  
d.arr[1]=39; // Ok  
d.arr=new int[3];  
// Errore
```

Errore



Il modificatore *final* (6)

Metodi finalizzati

Un metodo può essere dichiarato *final* in modo tale che le sottoclassi non possano ridefinirlo, *in questo modo il metodo mantiene lo stesso significato*

Un metodo *final* può far aumentare l'efficienza del codice - il compilatore, se lo ritiene opportuno, può rimpiazzare la chiamata al metodo con il codice del metodo stesso



I metodi nativi

Il corpo di un metodo può essere implementato in C/C++

- Al fine di riutilizzare codice già esistente
- Per eseguire operazioni che in Java non sono possibili (accesso a dispositivi fisici)

Quando un metodo è dichiarato nativo significa che il suo corpo è implementato da una libreria dinamica (file .dll per Windows e .so per Unix)

```
...  
native void methodX(int param);  
static {  
    System.loadLibrary(lib);  
}  
...
```

Si mette solo la segnatura

Carica la libreria lib che deve fornire l'implementazione del metodo metodoX()

L'uso di metodi nativi rende il codice non portabile ed è quindi sconsigliato



Introducing Java

I package, le classi e i membri

- I **package** costituiscono uno strumento per raggruppare classi
- Un package può contenere un numero qualunque di classi correlate per scopo o ereditarietà
- Un package può contenere altri package, quelli interni sono detti sottopackage
- L'impiego dei package consente di
 - strutturare le classi all'interno di unità logiche
 - evitare il conflitto tra i nomi di classi
 - regolamentare con maggior finezza l'accesso a classi, metodi e variabili



Introducing Java

I package, le classi e i membri

- Java è stato progettato per supportare il caricamento dinamico dei moduli, prelevandoli anche dalla rete
In queste situazioni si deve evitare conflitti nello spazio dei nomi (ad esempio Java non supporta l'uso di variabili globali)
- Tutte le variabili e i metodi sono dichiarati all'interno di classi e sono parte delle classi stesse
 - E tutte le classi fanno parte di un package*
- In questo modo tutti i campi e i metodi possono essere individuati in maniera univoca, attraverso il loro nome completato dai nomi del package e delle classi che li contengono
- Anche le classi di sistema di Java sono organizzate in package
Per default sono visibili quelle del package `java.lang`



packages, classes and members

Il nome completo di una variabile o un metodo è composto dal **nome del package**, dal **nome della classe** e dal **nome del membro** separati da **punti**

Esempio:

Il nome completo del metodo `resize()` della classe `ImageEditor` appartenente al package `graphics`, contenuto a sua volta nel package `application` è :

Identifica il package Identifica la classe
`application.graphics.ImageEditor.resize()`

I file .class contenenti il bytecode devono essere contenuti in una directory il cui nome è lo stesso del package

Nel nostro caso:

`application/graphics/ImageEditor.class`



packages, classes and members

L'opzione `-classpath` dei comandi `java` e `javac` indica alla JVM ed al compilatore dove sono poste le classi necessarie (le classi di sistema sono automaticamente visibili)

```
java -classpath /home/rossi <classe che usa ImageEditor>
```

Fa sì che il file `ImageEditor.class` venga cercato con il path `/home/rossi/application/graphics/ImageEditor.class`

E' possibile fornire più path separati da `:`` (unix) o da `;` (windows)



packages, classes and members

Esiste una regola basata sui nomi di dominio di Internet che evita che organizzazioni distinte producano package con lo stesso nome

Ad esempio:

`it.unipi.iet.rossi.applications.graphics.ImageEditor.resize()`

Identifica il package

identifica in maniera univoca il metodo `resize()` della classe `ImageEditor`, contenuta nel package `graphics`, contenuto a sua volta nel package `applications`

Dato che i nomi di dominio di Internet sono unici, i package ed i sottopackage possono essere identificati in maniera univoca

In questo modo se due organizzazioni producono classi con lo stesso nome, le classi saranno comunque distinguibili in quanto appartenenti a due package diversi

`it.unipi` identifica l'Università degli Studi di Pisa

`iet` identifica il dipartimento di Ingegneria dell'Informazione

`rossi` identifica lo specifico sviluppatore

I nomi che cominciano con `java` e `sun` sono riservati



I package

- La dichiarazione `package` consente al programmatore di specificare a quale package appartiene il codice contenuto in un file sorgente
- La dichiarazione `package` deve necessariamente essere la prima istruzione del file sorgente
- Le classi che fanno parte di un package devono essere poste in una directory il cui path corrisponde al nome del package secondo le regole viste

Tutte le classi che non specificano in maniera esplicita la loro appartenenza ad un package fanno parte di un **package di default senza nome**

```
// Qui possono esserci solo commenti
// o righe vuote
package it.unipi.iet.rossi.applications.graphics;
class ImageEditor {
    ...
}
```



I package: il comando import (1)

- Per far riferimento ad una classe di un altro package è necessario indicare il suo nome completo
Ad esempio:
`java.util.Vector vet = new java.util.Vector();`
- La dichiarazione import consente di importare classi da un package
Si può importare una singola classe:
`import java.util.Vector;`
oppure si possono importare tutte le classi di un package usando *
al posto dei nomi di classe:
`import java.util.*;`
- In questo modo vengono importate solo le classi dichiarate public e non vengono importati i sottopackage del package java.util



I package: il comando import (2)

- Una volta importato un package è possibile far riferimento alle sue classi senza dover specificare il loro nome completo
`import java.util.Vector;`
...
`Vector vet=new Vector();`
- Se si importano due package che contengono classi con lo stesso nome non si può evitare di riferire le classi con il loro nome completo
- Ad esempio se si importano i package A e B, ed entrambi contengono la classe Pippo, è necessario usare a seconda dei casi i nomi A.Pippo o B.Pippo



I package e la protezione di classe (1)

- È possibile associare un livello di protezione alle classi, oltre che ai membri (variabili e metodi) delle classi stesse come visto in precedenza
Per una classe sono possibili solo due livelli di protezione:
 - *package* (*nessun modificatore*)
 - *public*
- Il livello di protezione *package* è quello di default, quindi una classe che non è dichiarata *public* rientra in tale ambito
- Il livello *package* rende una classe visibile solo dalle altre classi dello stesso package:
le classi all'esterno del package e quelle all'interno dei sottopackage non possono ne importarla ne riferirla
- Poiché esiste un package di default, che raggruppa tutte le classi che non dichiarano in maniera esplicita la loro appartenenza ad un altro package, tali classi sono visibili tra di loro



I package e la protezione di classe (2)

- Le classi dichiarate *public* possono invece essere importate da classi che appartengono ad un altro package
- **Le classi devono avere un livello package quando implementano funzionalità "interne"**, in modo da limitare l'effetto di eventuali modifiche e precludere ad altri di accedere a codice legato all'implementazione interna (una sorta di incapsulamento ma a più alto livello)
- Vediamo ad esempio due classi che implementano una lista, una pubblica e l'altra nascosta
- La definizione delle due classi è contenuta in un unico file, l'importante in questi casi è che solo una sia dichiarata *public*, ed il file *.java* abbia il nome di quest'ultima



I package e la protezione di classe (3)

```
package utilità;  
public class Lista {  
    private Elemento radice;  
    public void aggiungi(Object o) {  
        radice=new Elemento(o, radice);  
    }  
    ...  
}  
  
class Elemento {  
    private Object contenuto;  
    private Elemento successivo;  
    Elemento(Object o, Elemento e) {  
        contenuto=o;  
        successivo=e;  
    }  
}
```

- La classe Lista è dichiarata public quindi è possibile utilizzarla al di fuori del package utilità
- La classe Elemento invece non è visibile dall'esterno del package



Esercizi

- Creare la classe Persona con alcuni attributi e metodi
 - fisici(statura, ...), numeroCartaId, indirizzo, ...
 - getNumCartaId(), ...
- Derivare due sottoclassi: Studente e Professore ognuna con altri attributi e metodi
 - anno_di_corso, materia_insegnata, scuola,
 - getScuola(), ...
- Fornire tutte le classi di costruttori e di un metodo che ne stampi lo stato (il valore di tutte le variabili)
- Creare un array di Persona e inserire al suo interno qualche oggetto Studente e qualche oggetto Professore e qualche oggetto Persona
- Stampare il contenuto dell'array