Progettazione di classi

Scegliere una classe

- Una classe rappresenta un singolo concetto
  - Esempi:
    - Una classe puo` rappresentare un concetto matematico
      - Point
      - Rectangle
      - Ellipse
    - Una classe puo` rappresentare un'astrazione di un'entita` della vita reale
      - BankAccount
      - Borsellino
    - Una classe puo` svolgere un lavoro: classi di questo tipo vengono dette *Attori* e in genere hanno nomi che terminano con "er" o "or"
      - StringTokenizer
      - Random (la scelta del nome non e` molto appropriata!)
    - Classi "di utilita`" che non servono a creare oggetti ma forniscono una collezione di metodi statici e costanti
      - Math

Coesione

- I metodi e le costanti dell'interfaccia dovrebbero essere strettamente correlati al **singolo** concetto espresso dalla classe

  Es.: la classe Purse manca di coesione

```
public class Purse  {
      public Purse(){...}
      public void addNickels(int count){...}
      public void addDimes(int count){...}
      public void addQuarters(int count){...}
      public double getTotal(){...}
      public static final double NICKEL_VALUE =0.05;
      public static final double DIME_VALUE =0.1;
      public static final double QUARTER_VALUE =0.25; ...
   }
```

Coesione

- La classe Purse esprime due concetti:
  - *borsellino* che contiene monete e calcola il loro valore totale
  - *valore delle singole monete*
- Soluzione: Usa due classi:

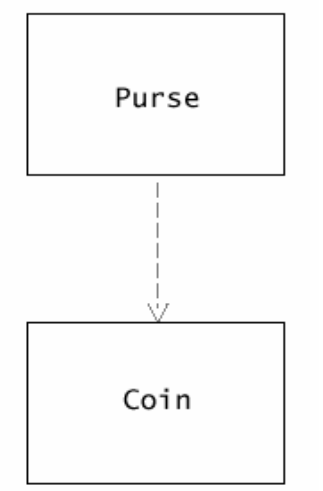
```
public class Coin
   {
      public Coin(double aValue,String aName){...}
      public double getValue(){...}
   }

public class Purse
   {
      public Purse(){...}
      public void add(Coin aCoin){...}
      public double getTotal(){...}
   }
```

Accoppiare

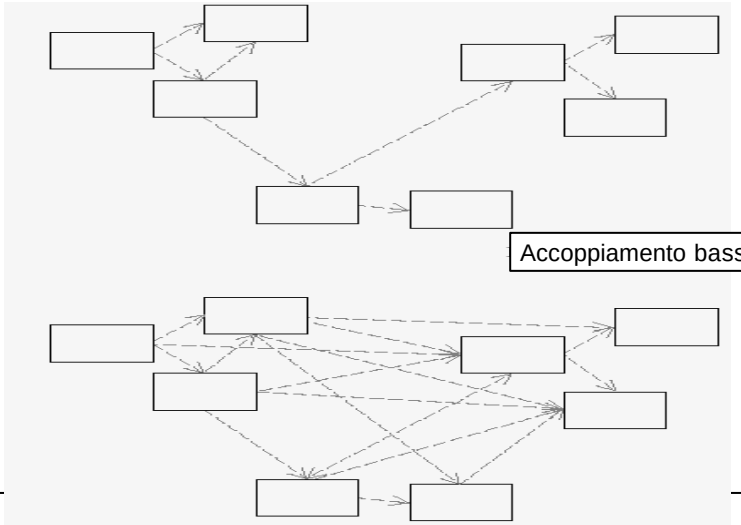
- Una classe A *dipende* da una classe B se usa un'istanza o invoca un metodo di B
  - Es. Purse dipende da Coin perchè usa un'istanza di Coin
  - Es. Coin non dipende da Purse
- E' possibile avere molte classi che dipendono tra di loro (accoppiamento elevato)
  - Problemi dell'accoppiamento elevato:
    - Se una classe viene modificata potrebbero dover essere modificate tutte le classi che dipendono da essa
    - Se si vuole usare una classe in un altro programma bisognerebbe usare anche tutte le classi da cui quella classe dipende

Dipendenza tra Purse e Coin

Notazione UML per rappresentare i diagrammi delle dipendenze tra classi o oggetti.



Accoppiamento elevato e accoppiamento basso



Il parametro implicito

```
public void preleva(double amount)
   {
      double newBalance = balance - amount;
      balance = newBalance;
   }
```

- il metodo preleva fa riferimento alla variabile istanza **balance** dell'oggetto con cui è stato invocato
  - contoMaria.preleva(100);
    - fa riferimento a contoMaria.balance -100

Il parametro implicito

- L'invocazione di un metodo dipende
  - dai valori o dagli oggetti passati come parametri
  - dall'oggetto con cui è invocato il metodo
- quindi oltre ai parametri presenti nella lista dei parametri (*parametri espliciti*), ogni metodo ha un *parametro implicito* che rappresenta l'oggetto con cui viene chiamato il metodo

Il parametro implicito

- E` possibile fare riferimento al parametro implicito usando la parola chiave **this**

```
public void preleva(double amount)
{
  double newBalance = this.balance - amount;
  this.balance = newBalance;
}
```

Il parametro implicito

- Nell'invocazione **contoMaria.preleva(100);** il parametro implicito this ha valore contoMaria
- Nota: I metodi static non hanno il parametro implicito

Matodi accessori e metodi modificatori

- Accessore: non cambia lo stato del parametro implicito (es.: getBalance)
- Modificatore: cambia lo stato dell'oggetto del parametro implicito (es.: deposit)
- Regola empirica: Un modificatore dovrebbe restituire void
- Classi immutabili: contiene solo metodi accessori (es.: String)

Effetti collaterali

- **Effetti collaterali:** qualsiasi modifica che può essere osservata al di fuori dell'oggetto
- **Esempio:** un metodo che modifica un parametro esplicito di tipo oggetto

```
public void transfer(double amount, BankAccount other)
 {
    balance = balance - amount;
    other.balance = other.balance + amount;
 }
```

  - **Nota:** E` buona norma evitare di scrivere metodi che modificano i parametri espliciti

Effetti collaterali : altro esempio

- **Esempio:** un metodo che stampa dati di una classe

```
public void printBalance()
{
 System.out.println("Il valore del bilancio
                                    è:"+ balance);
   . . .
}
```

Effetti collaterali: Osservazioni

- Controindicazioni:
  - Si assume che chi usa la classe BankAccount conosca l'italiano
  - Il metodo println viene invocato con l'oggetto System.out che indica lo standard output: per alcuni sistemi non è possibile usare l'oggetto System.out (per esempio sistemi embedded)
  - La classe BankAccount diventa dipendente dalle classi System e PrintStream (PrintStream è il tipo di System.out)
- E` preferibile scrivere il metodo getBalance() che restituisce il valore di balance e stampare con

```
System.out.println("Il bilancio è:"+  getBalance());
```

Effetti collaterali: altro esempio

- **Esempio:** un metodo che stampa messaggi di errore

```
public  void deposit(double amount)
{
   if (amount < 0)
     System.out.println("Valore non consentito");
   . . .
}
```

  - **Nota:** I metodi non dovrebbero mai stampare messaggi di errore: per segnalare problemi si devono usare le *eccezioni*

Modifica parametri di tipo primitivo

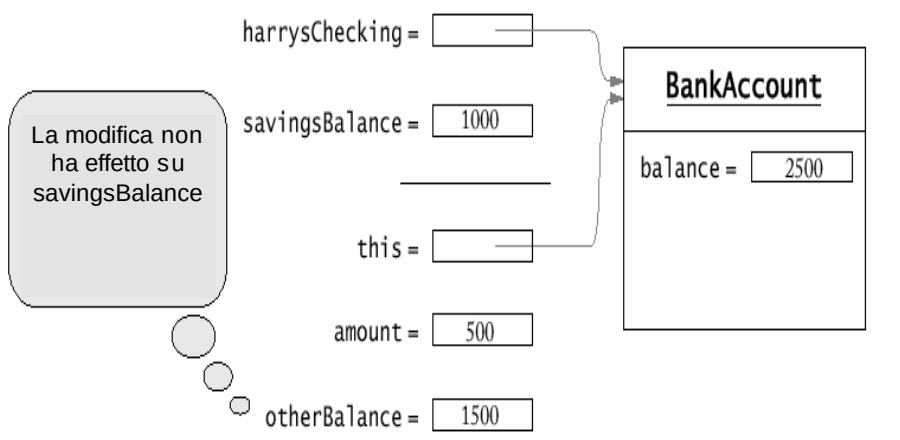
```
void transfer(double amount, double otherBalance)
 {
   balance = balance - amount;
   otherBalance = otherBalance + amount;
 } // non trasferisce amount su otherBalance
```

- Dopo aver eseguito le seguenti istruzioni

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);
```

il valore di savingsBalance è 1000 e non 1500

Passaggio di parametri: per valore / indirizzo



Pre-condizioni

- **Pre-condizioni:** requisiti che devono essere soddisfatti perchè un metodo possa essere invocato
  - Se le precondizioni di un metodo non vengono soddisfatte, il metodo potrebbe avere un comportamento sbagliato
- Le pre-condizioni di un metodo devono essere pubblicate nella documentazione
  - Esempio:

```
/**
   Deposita denaro in questo conto.
   @param amount la somma di denaro da versare   (Precondition:
amount >= 0)
*/
```

Pre-condizioni

- Uso tipico:
  - Per restringere il campo dei parametri di un metodo
  - Per richiedere che un metodo venga chiamato solo quando l'oggetto si trova in uno stato appropriato

Pre-condizioni

- Nel caso in cui le pre-condizioni non siano soddisfatte un metodo può
  - lanciare un'eccezione

    Esempio:
    ```
    if (amount < 0)
        throw new IllegalArgumentException();
    balance = balance + amount;
    ```
  - non fare niente

    Esempio:
    ```
    if (amount < 0)
        return; // don't do this
    balance = balance + amount;
    ```
    - e` molto meglio lanciare un'eccezione: facilita il collaudo del programma

Pre-condizioni

- Verificare che le pre-condizioni sono soddisfatte potrebbe essere molto costoso e complicato per cui il metodo potrebbe non effettuare alcun controllo
  - Il cattivo comportamento del programma sarà da addebitarsi al chiamante

    Esempio
    ```
    // se amount è negativo e rende balance negativo
    // è colpa del metodo chiamante
    balance = balance + amount
    ```

Post-condizioni

- Post-condizioni: che devono essere soddifatte al termine dell'esecuzione del metodo
- Due tipi di post-condizioni:
  - Il valore di ritorno deve essere computato correttamente
  - Al termine dell'esecuzione del metodo, l'oggetto con cui il metodo è invocato si deve trovare in un determinato stato
- Contratto: Se il chiamante soddisfa le pre-condizioni, il metodo deve soddisfare le post-condizioni

I metodi statici

- I metodi statici non hanno il parametro implicito
  - Esempio: il metodo **sqrt** di **Math**
- I metodi statici vengono detti anche *metodi di classe* perchè non operano su una particolare istanza della classe
  - Esempio: Math.sqrt(m);
    - Math è il nome della classe non di un oggetto

I metodi statici

- Metodi che manipolano esclusivamente tipi primitivi

```
public static boolean
    approxEqual(double x, double y)
          { . . .}
```

- Non ha senso invocare approxEqual con un oggetto come parametro implicito
- Dove definire approxEqual?
  - Scelta 1. nella classe che contiene i metodi che invocano approxEqual
  - Scelta 2. creiamo una classe, simile a Math, per contenere questo metodo e possibilmente altri metodi che svolgono elaborazioni numeriche

```
public class Numeric{
  public static boolean
    approxEqual(double x,
                          double y){
    . . .
  }
    //altri metodi numerici
}
```

Il metodo main

- Il metodo **main** è statico
  - quando viene invocato non esiste ancora alcun oggetto

    public static void main (String [ ] args){…}

Programmazione O.O. e metodi statici

- Se si usano troppi metodi statici si utilizza poco la programmazione orientata agli oggetti
- Se si usano troppi metodi statici vuol dire che le classi che usiamo non modellano adeguatamente le entità su cui vogliamo operare

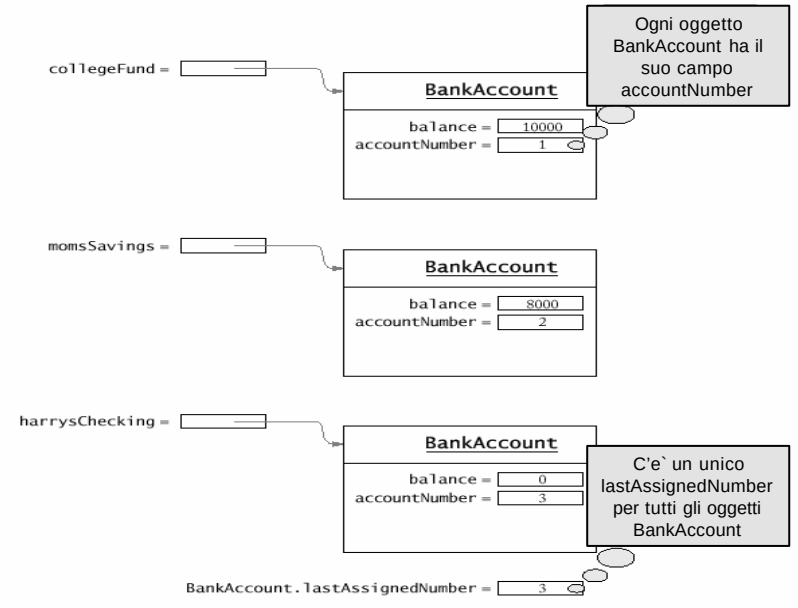
Variabili statiche

- Problema: vogliamo assegnare a ciascun conto un numero identificativo diverso
  - Il costruttore crea il primo conto con il numero 1, il secondo con il numero 2, ecc.

```
public class BankAccount {
    public BankAccount() {
        lastAssignedNumber++;  //numero da assegnare al nuovo c/c
        accountNumber = lastAssignedNumber;
    }
    . . .
    private double balance;
    private int accountNumber;
    private static int lastAccountNumber;
}
```

- Se **lastAccountNumber** non fosse dichiarata **static**, ogni istanza di BankAccount avrebbe il proprio valore di lastAccountNumber

Variabili statiche e variabili istanza



Inizializzazione di variabili statiche

- Le variabili statiche **non** devono essere inizializzate dal costruttore

```
public BancAccount{
  lastAssignedNumber = 0;
 …
} /*errore: lastAssignedNumber viene azzerata ogni volta
    che viene costruito un nuovo conto*/
```

- Si può usare un'inizializzazione esplicita

  Es.: `private static int lastAssignedNumber = 0;`

- Non assegnando nessun valore, la variabile assume il valore di default del tipo corrispondente: valori 0, false o null

Uso delle variabili statiche

- Le variabili statiche vengono usate raramente
  - I metodi che modificano variabili statiche hanno effetti collaterali
  - I metodi che leggono variabili statiche potrebbero avere comportamenti diversi a seconda del valore delle variabili statiche
    - Se chiamiamo due volte uno stesso metodo fornendo gli stessi argomenti, esso potrebbe avere comportamenti diversi

Uso delle variabili statiche

- E` bene non usare le variabili statiche per memorizzare temporaneamente dei valori
  - **Esempio:**
    - Un metodo che memorizza i risultati di una computazione in variabili statiche in modo che possano essere disponibili alla fine della sua esecuzione:
      - → se non si reperiscono immediatamente i valori delle variabili statiche, questi potrebbero essere modificati da un altro metodo

La costanti statiche

- Una *costante statica* e dichiarata usando le parole chiave **static** e **final**
  - **Es.:** public static final COSTO_COMMISS=1.5;
- E` ragionevole dichiarare statica una costante
  - Sarebbe inutile che ciascun oggetto della classe BankAccount avesse una propria variabile COSTO_COMMISS con valore costante 1.5
  - E` molto meglio che tutti gli oggetti della classe BankAccount facciano riferimento ad un'unica variabile COSTO_COMMISS
- Le costanti statiche si possono usare liberamente

Gli Identificatori in Java

- Tutti gli identificatori (variabili, metodi, classi, package, ….) in java seguono le stesse convenzioni del C :
  - Possono essere costituiti da
    - Lettere
    - Numeri
    - Carattere di underscore (_)
  - Non possono iniziare con un numero
  - Non possono essere parole chiave di java

Dichiarazione di variabili

- In Java le variabili possono essere dichiarate ovunque nel codice e si possono fare anche cose del tipo

```
int a=20;
int n=a*10;
```

Visibilità delle variabili

- Campo di visibilità di una variabile (scope): parte del programma in cui si può fare riferimento alla variabile mediante il suo nome
- Campo di visibilità di una variabile locale: dalla sua dichiarazione alla fine del blocco
  - Nell'ambito di visibilità di una variabile locale non è possibile definirne un'altra avente lo stesso nome
  - Esempio: for(int i=0 ;i<10 ; i++){

```
…
float i = 3.5; / *errore: qui non si puo`dichiarare
                  un'altra variabile di nome i */
}
```

Visibilità sovrapposte

- I campi di visibilità di una variabile locale e di una variabile istanza possono sovrapporsi
  - La variabile locale oscura la variabile di istanza con lo stesso nome

```
public class Coin
{
  public void draw(Graphics2D g2)
  {
    String name = "SansSerif"; // variabile locale
    ...
  }
  private String name; //variabile di istanza
  private double value ;
}
```

Visibilità sovrapposte

- Se in un metodo si vuole fare riferimento ad una variabile istanza che ha lo stesso nome di una variabile locale allora occorre usare il riferimento **this**

```
public class Coin
  {
    public void draw(Graphics2D g2)
     {
      String name = "SansSerif"; // variabile locale
      g2.setFont(new Font(name, . . .));   /* qui name si riferisce
                                              alla  variabile locale */
      g2.drawString(this.name, . . .); /* qui name si riferisce alla
                                          variabile  di istanza */
    }
  private String name; // variabile di istanza
  . . .
  }
```

Visibilità delle variabili locali

- Le variabili dichiarate all'interno di un metodo
  - Esistono finché il metodo è in esecuzione
  - Oscurano il nome dei campi statici e non statici
- Le variabili dichiarate in un blocco di istruzioni
  - Esistono finché il blocco è in esecuzione
  - Oscurano il nome dei campi statici e non statici
  - Non oscurano le variabili definite (e ancora "in vita") prima dell'inizio del blocco (**ERRORE IN COMPILAZIONE**)
    - Campi di visibilità non si possono sovrapporre

Visibilità di membri di classe

- Nome qualificato = *prefisso.nome membro*
- Prefisso
  - Nome classe per metodi e campi statici
    - Es.: Math.sqrt, Math.PI;
  - Riferimento a oggetto per variabili e metodi di istanza
    - Es.: contoMaria.getBalance();
- Un metodo può accedere ad un campo (non private) o invocare un metodo (non private) usando il nome qualificato

Visibilità di membri di classe

- All'interno di una classe si può accedere alle variabili istanza e ai metodi della classe specificandone semplicemente il nome (si sottintende il parametro implicito o il nome della classe stessa come prefisso)
  - Esempio:

```
public void trasferisci(double somma, BankAccount altro )
  {
   preleva(somma); // equivale a this.preleva(somma)
   altro.deposita(somma) ;
  }
```

Pacchetti

- Insieme di classi correlate
- Libreria Java costituita da numerosi package
- Possibile dichiarare appartenza di una classe ad un package mettendo sulla prima riga del file che contiene la classe:

  package *packagename*;

  - **Esempio :**

```
package com.horstmann.bigjava;
public class Numeric
{
  . . .
}
```

  - Se la dichiarazione è omessa, le classi create fanno parte di un package di default (senza nome)

Alcuni pacchetti della libreria Java

Package	Scopo	Classi campione
java.lang	Supporto al linguaggio	Math
java.util	Utility	Random
java.io	Input/output	PrintStream
java.awt	Abstract Windowing Toolkit (Interfacce grafiche)	Color
java.applet	Applet	Applet
java.net	Connessione di rete	Socket
java.sql	Accesso a Database	ResultSet
javax.swing	Interfaccia utente Swing	JButton
org.omg.CORBA	Common Object Request Broker Architecture	ORB

Nomi dei pacchetti

- Java può usare classi caricate dinamicamente via Internet
  - Necessita avere un meccanismo che garantisca l'unicità dei nomi delle classi e dei package.
- Difficile pensare di usare nomi classi differenti
- Basta assicurarsi che i nomi dei package siano differenti

Nomi dei pacchetti

- Per rendere unici i nomi dei pacchetti si possono usare i nomi dei domini Internet alla rovescia
  - **Esempi:** **it.unisa.mypackage**
    com.horstmann.bigjava
- In generale una persona non è l'unico utente di un dominio Internet, quindi meglio usare l'intero indirizzo di e-mail.
  - **Esempio** : marco@dia.unisa.it diventa

Pacchetti e posizione nel file system

- Il nome del pacchetto deve coincidere con il percorso della sottocartella dove è ubicato il pacchetto
  - **Esempio:** il pacchetto **com.horstmann**.bigjava deve essere ubicato nella sottocartella: com/horstmann/bigjava
    - Il percorso della sottocartella è specificato a partire da una directory prefissata o dalla directory corrente

Localizzazione dei pacchetti

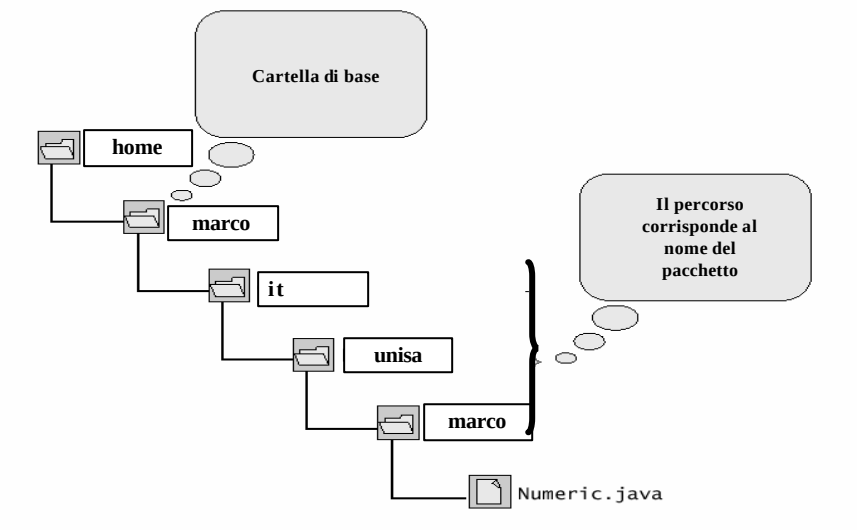
- Supponiamo che la directory corrente sia /home/marco e che in un file .java vogliamo importare il package

  it.unisa.marco

- I file che compongono il package devono stare nella sottodirectory it/unisa/marco della directory corrente, cioè in

  /home/marco/it/unisa/marco

Cartella di base e sottocartelle per i pacchetti



Localizzazione dei pacchetti

- Se vogliamo che Java cerchi i file componenti un package a partire da una particolare directory, possiamo
  - assegnare il suo path assoluto alla variabile di ambiente CLASSPATH
    - Es. export CLASSPATH=/home/marco/esercizi: (UNIX)
      - Tutte le volte che importo classi non standard la ricerca parte da /home/marco/esercizi
      - Comodo ma **non garantito** su tutti i sistemi e/o tutte le installazioni del JDK
  - Usare l'opzione -classpath del compilatore javac (garantito)
    - javac -classpath /home/marco/esercizi *Numeric.java*

Importare pacchetti

- Si può sempre usare una classe senza importarla
  - **Esempio:**
    java.awt.Rectangle r
    = new **java.awt.Rectangle**(6,13,20,32);
- Per evitare di usare nomi qualificati possiamo usare la parola chiave import
  - **Esempio:**
    import java.awt.Rectangle;
    . . .
    Rectangle r = new Rectangle(6,13,20,32);

Importare pacchetti

- Si possono importare tutte le classi di un pacchetto
  - **Esempio:**
    import java.awt.*;
- Nota: non c'è bisogno di importare java.lang per usare le sue classi

Il Problema della Collisione

- Se importiamo due package che contengono entrambi una certa classe *Myclass*, un riferimento a *Myclass* nel codice genera una collisione sul nome *Myclass*.
- In questo caso il compilatore chiede di usare i nomi completi per evitare ambiguità.
- Dati i package *pack1* e *pack2*, ci riferiremo alle classi *Myclass* come

```
pack1.Myclass e pack2.Myclass
```

Il Significato di import

- L'istruzione **import** dice soltanto al compilatore dove si trova un certo package o una certa classe.
- Per ogni riferimento ad una classe *Myclass*, che non faccia parte dello stesso package del file che stiamo compilando, il compilatore controlla **solo** l'esistenza del file *Myclass.class* nella locazione specificata da **import**.

Caricamento di Classi Importate

- Le classi importate, tramite l'istruzione **import** o specificando il loro nome completo, vengono caricate dal **Class Loader** a run-time
- Finché il codice non fa un riferimento esplicito ad una classe che è stata importata, la classe non viene caricata

Differenze tra import e #include

- **#include** del C e del C++
  - è una direttiva al preprocessore per inserire all'interno del sorgente un file contenente
    - prototipi delle funzioni di libreria e costanti predefinite

    oppure
    - prototipi di funzioni e costanti definite dal programmatore
  - Bisogna utilizzarla per forza
- **import** di java
  - È una semplificazione per specificare il nome di una classe
    - Non include niente nel file sorgente, dice solo dove si trova la classe
  - È possibile non usarla mai

Convenzioni sui nomi

- Per convenzione i nomi delle **classi** iniziano con lettera **maiuscola**, i nomi dei **package** sono in lettera **minuscola**.
- I nomi dei package in genere sono formati da identificatori separati da punti