



Progettazione di classi



Outline

- Imparare ad identificare le classi più appropriate per risolvere un problema
- Concetti di coesione e accoppiamento
- Metodi statici e variabili statiche
- Package



Scelta delle classi

- Per scrivere una buona applicazione usando un linguaggio ad oggetti come Java è bene fare un'adeguata progettazione iniziale
- Il punto focale su cui concentrarsi sono le **classi**
- Nella programmazione funzionale classica ci si concentra sulle funzioni: sul flusso che il codice dovrebbe seguire
- Nella programmazione ad oggetti invece l'accento è sulle entità, cioè gli oggetti appartenenti alle varie classi individuate
- I metodi, cioè la parte funzionale, devono essere pensati come associati alle entità

Scegliere una classe

- Una classe rappresenta un singolo concetto del dominio dell'applicazione
- Nome della classe = nome che esprime il concetto
- Una classe può rappresentare un concetto matematico

Point

Rectangle

Ellipse

- Una classe può rappresentare un'astrazione di un'entità della vita reale

BankAccount

Borsa

CashRegister

Moneta

Scegliere una classe

- Una classe può svolgere un lavoro: classi di questo tipo vengono dette Attori e in genere hanno nomi in inglese che terminano con “er” o “or”

Scanner

Random (meglio se RandomNumberGenerator)

- Classi “di utilità” che non servono a creare oggetti ma forniscono una collezione di metodi statici e costanti

Math

- Classi starter: in genere contengono il solo metodo **main** e hanno il solo scopo di avviare la computazione (classi test)

Scelta delle classi

- Quale potrebbe essere una classe poco valida?
- In generale sono sintomi di errori di progettazione:
 - Se dal nome di una classe non si capisce cosa dovrebbero fare gli oggetti della classe stessa
 - Se il nome di una classe non rappresenta un gruppo di entità, ma una specifica funzione
- Es: classi come **CalcolaBustaPaga** oppure **ProgrammaPerIlPagamento**

Domande

- Qual'è la regola da usare per trovare le classi?
 - **Risposta:** Cercare i sostantivi nelle descrizioni dei problemi.
- Supponiamo di dover scrivere un programma per giocare a scacchi. La classe **Scacchiera** è appropriata? E la classe **MuoviPezzo**?
 - **Risposta:** Sì (**Scacchiera**) e no (**MuoviPezzo**).



Coesione e accoppiamento

- Vediamo due criteri utili per analizzare la qualità di una interfaccia pubblica di una classe:
 - *Coesione*
 - *Accoppiamento*

Coesione

- Una classe deve rappresentare un singolo concetto
- Una classe è **coesa** se l'interfaccia contiene solo operazioni tipiche del concetto che la classe realizza
- 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:
 - *portamonete* che contiene monete e calcola il loro valore totale
 - *valore* delle singole monete
- Soluzione: Si usano 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() {...}
}
```



Coesione

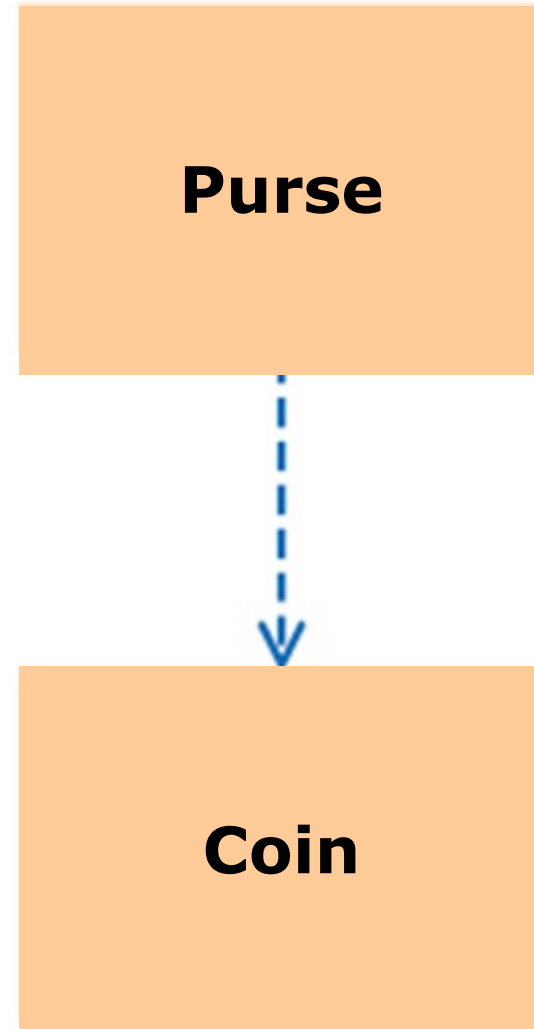
- Una classe con bassa coesione fa tante cose insieme, svolgendo molto lavoro "sparso" e non correlato (ha troppe responsabilità). Questo tipo di situazione sarebbe da evitare in quanto queste classi risultano:
 - complesse da riutilizzare (*bassa riusabilità*);
 - complicate da mantenere (*scarsa manutenibilità*);
 - delicate e critiche in quanto soggette a continui cambiamenti (*bassa flessibilità*)
- Una forma comune di bassa coesione si ha in quelle classi che presentano un grandissimo numero di **metodi** (pubblici o privati).

Accoppiamento

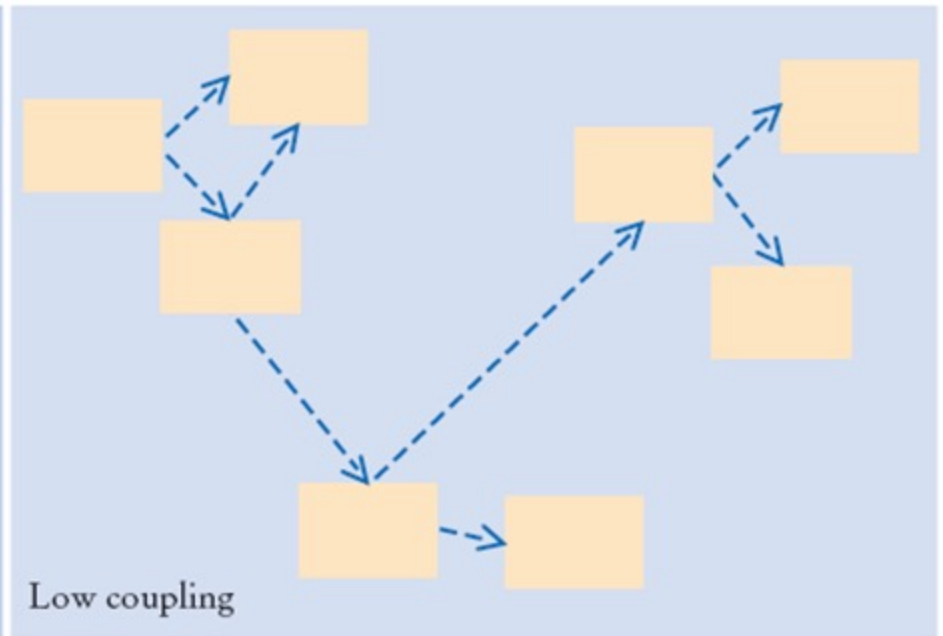
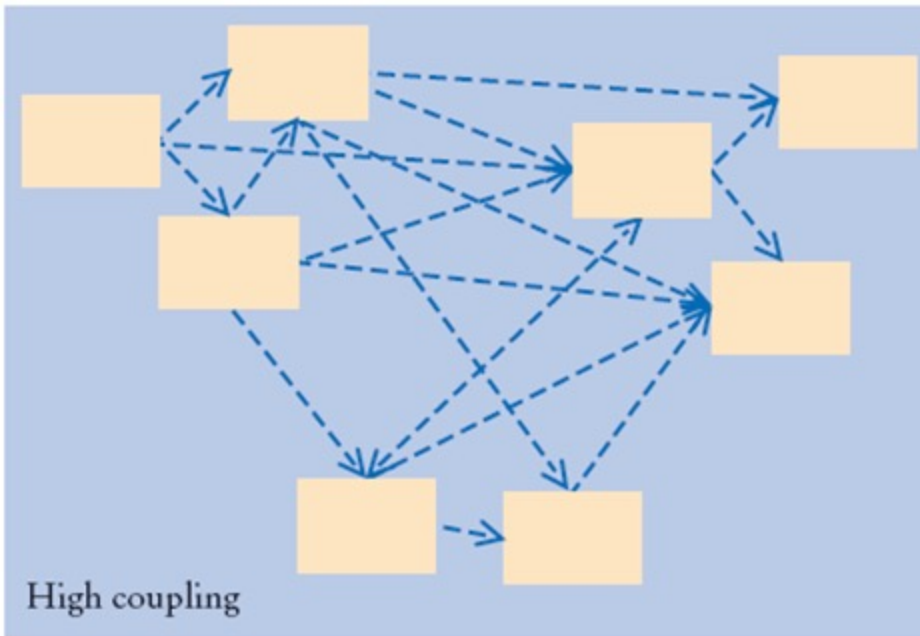
- Una classe *A* *dipende* da una classe *B* se usa esemplari di *B* (oggetti o metodi di *B*)
 - Es: *Purse* dipende da *Coin* perché usa un'istanza di *Coin*
 - Es: *Coin* non dipende da *Purse*
- E' possibile avere molte classi che dipendono tra di loro (**accoppiamento elevato**)
 - Problemi dell'accoppiamento elevato:
 - Se una classe viene modificata tutte le classi che dipendono da essa potrebbero necessitare di una modifica
 - 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.



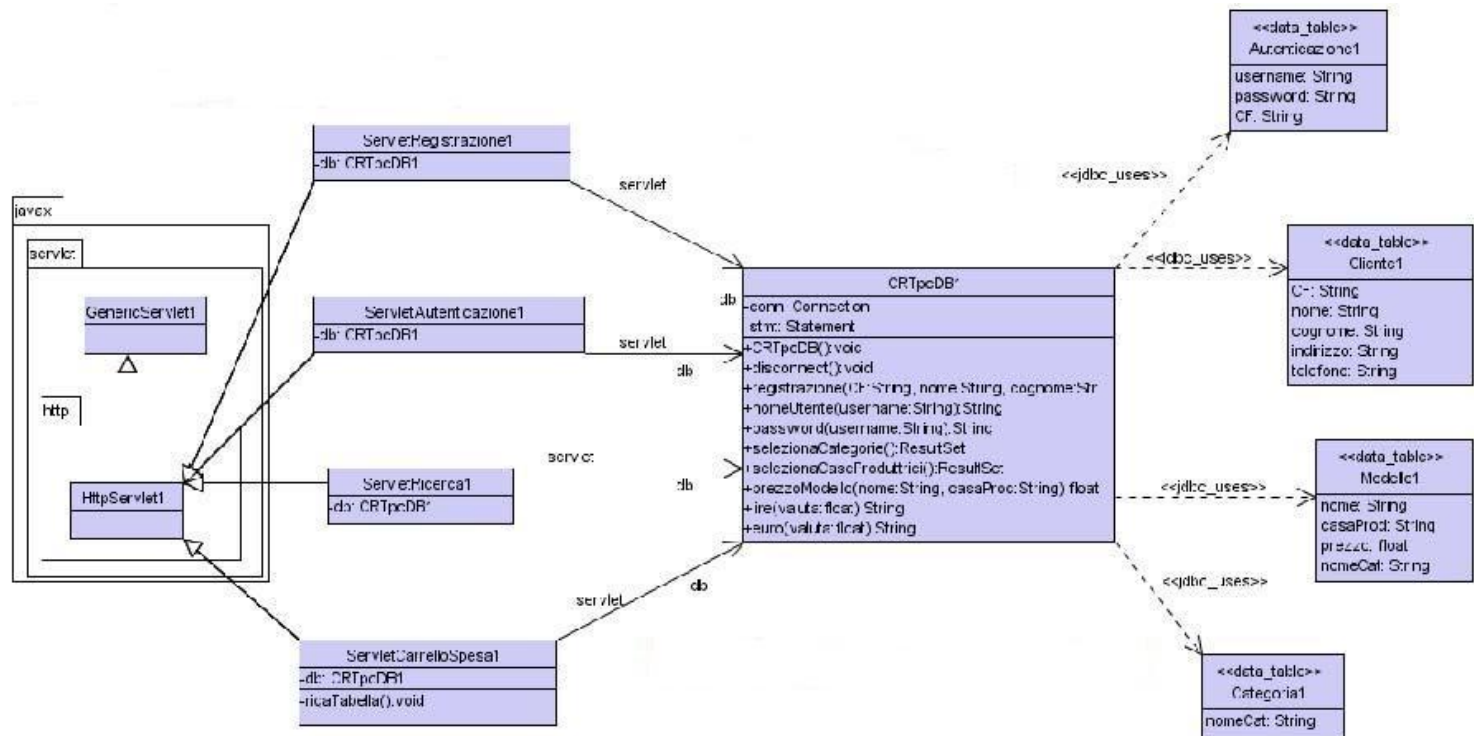
Accoppiamento elevato e accoppiamento basso



Esempi di astrazione funzionale

Un unico “oggettone”
per incapsulare tutte le
funzioni di interfaccia alla
base dati (pessima scelta)

Riuso: no
Efficienza: no
Accoppiamento: alto
Coesione: bassa
Manutenibilità: bassa



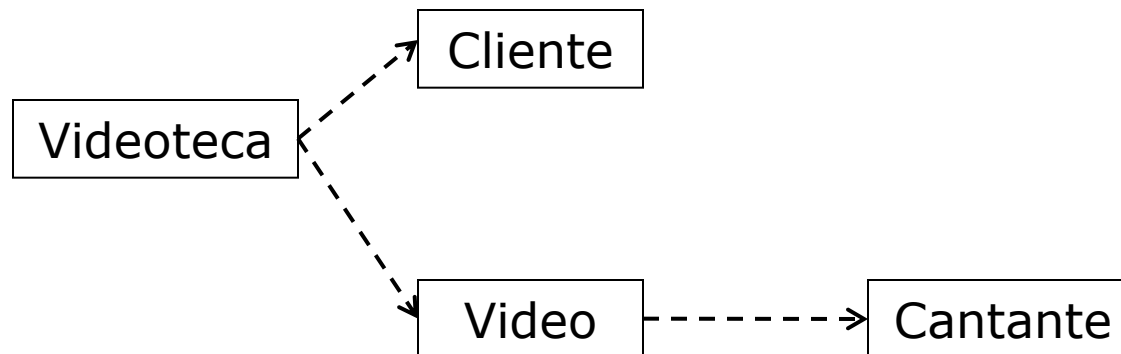
Esempio

- Supponete di avere una classe **Videoteca** per gestire il noleggio di video musicali da parte dei clienti di una videoteca, che abbia variabili istanza
 - `codiceCliente`
 - `nomeCantante`
 - `creditoCliente`
 - `listaVideoNoleggiati`
 - `titoloVideo`
 - `n_canzoniCantante`
 - `n_copieDisponibili`

e che abbia metodi che consentono di visualizzare ciascuno dei dati, e metodi per modificare, rispettivamente, il numero di copie disponibili, il numero di canzoni di ciascun cantante e il credito di ciascun cliente.
- Cosa c'è di sbagliato nella progettazione della classe Videoteca?
- Quali classi sono più adatte alla gestione della Videoteca?

Esempio

- La classe Videoteca, così come pensata, ha una bassissima coesione, poiché preposta a modellare entità distinte (il cliente, il video noleggiato e il relativo cantante).
- Per migliorare la progettazione, dobbiamo separare le diverse entità che entrano in gioco, definendo per esse classi separate e opportunamente accoppiate. Una possibile soluzione è offerta dal seguente schema: *Videoteca, Cliente, Video, Cantante*





Esempio

- La classe **Videoteca** gestisce il noleggio di un video da parte di un cliente per un dato numero di giorni
- La classe **Cliente** crea un cliente con un credito iniziale e ha metodi per aggiornare il credito e la lista dei video noleggiati, aggiungendo il titolo di un nuovo video a tale lista
- La classe **Video** crea un video con un certo numero di copie iniziali e ha metodi per aggiornare tale numero
- La classe **Cantante** crea un cantante con nome e numero di canzoni inizialmente inserite nel database e ha un metodo per incrementare tale numero.
- Tutte le classi (tranne Videoteca, che è la principale) hanno metodi per reperire i dati privati.

La classe Videoteca

- L'interfaccia pubblica delle classi individuate:

```
public class Videoteca {  
    private ArrayList<Cliente> listaClienti;  
    private ArrayList<Video> listaVideo;  
    // Il costruttore crea un oggetto videoteca  
    // inizializzando il cliente, il video e il  
    // numero di giorni di noleggio.  
    public Videoteca() {  
        //inizializzo gli arraylist di Cliente e di  
        //Video rispettivamente  
    }  
}
```

La classe Videoteca

- Il metodo noleggia, dopo aver fatto gli opportuni controlli sul numero di copie disponibili e sul credito del cliente, effettua il noleggio decrementando il numero di copie disponibili e il credito del cliente e aggiungendo il titolo del video noleggiato alla lista dei video noleggiati dal cliente.

```
public void noleggia(Cliente c, Video v){  
    }  
public inserisciCliente(Cliente c){  
    // inserisce l'oggetto cliente nella lista dei  
    // clienti  
}  
public inserisciVideo(Video v){  
    // inserisce l'oggetto video nella lista dei video  
}  
}
```

La classe *Cliente*

```
public class Cliente {  
    private String nomeCliente;  
    private int codiceCliente;  
    private double creditoCliente;  
    String[] videoNoleggiati;
```

- Il **costruttore** crea un oggetto `Cliente` assegnando un nome, un codice univocamente determinato (si utilizzerà per questo una variabile statica), un credito iniziale e inizializzando alla stringa vuota la lista dei video noleggiati.

```
    public Cliente(String s, double d) {
```

La classe *Cliente*

- Il metodo **aggiornaCredito** decrementerà il credito corrente dell'importo per il noleggio di un video.

```
public void aggiornaCredito() {  
}
```

- Il metodo **aggiornaListaVideo** aggiunge il titolo del video noleggiato alla lista dei video noleggiati dal cliente.

```
public void aggiornaListaVideo(Video v) {  
}
```

La classe *Cliente*

- I metodi per reperire i dati:

```
    public String getNome() {  
    }  
    public int getCodice() {  
    }  
    public double getCredito() {  
    }  
    public String getList() {  
    }  
}
```

La classe Video

```
public class Video {  
    private String titolo;  
    private Cantante cant;  
    private int n_copieDisponibili;
```

- Il **costruttore** crea un oggetto video assegnando un titolo, un cantante e il numero iniziale di copie disponibili.

```
    public Video(String s, Cantante c, int x){  
    }
```

- Il metodo **aggiornaCopie** decrementa il numero di copie disponibili.

```
    public void aggiornaCopie(){  
    }
```




La classe *Video*

- I metodi per reperire i dati.

```
public String getTitolo() {  
    }  
public Cantante getCantante() {  
    }  
public int getCopie() {  
    }  
}
```

La classe Cantante

```
public class Cantante {  
    private String nomeCantante;  
    private int numeroCanzoniCantante;
```

- Il **costruttore** crea un oggetto cantante inizializzando il nome e il numero di canzoni.

```
    public Cantante(String s, int i) {  
    }
```

La classe *Cantante*

- Il metodo **aggiungiCanzone** aumenta il numero di canzoni di quel cantante.

```
public void aggiungiCanzone() {  
    }  
}
```

- I metodi per reperire i dati.

```
public String getNome() {  
    }  
public int getNumero() {  
    }  
}
```

Metodi di accesso e metodi modificatori

- **Metodo di Accesso**: non cambia lo stato del parametro implicito

```
double balance = account.getBalance();
```

- **Modificatore**: cambia lo stato del parametro implicito (Es.: **deposit()**)

```
account.deposit(1000);
```

- **Regola empirica**: Un modificatore dovrebbe restituire **void**

- **Classi immutabili**:

- contengono solo metodi di accesso (es.: **String**)

```
String name = "John Q. Public";  
String uppercased = name.toUpperCase();  
// name is not changed
```

Effetti collaterali

- **Effetto collaterale**: qualsiasi modifica che può essere osservata al di fuori del metodo
- Es.: 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**: Se possibile, è buona norma evitare di scrivere metodi che modificano i parametri espliciti

Effetti collaterali: altro esempio

- Gli effetti collaterali possono introdurre dipendenze e possono causare comportamenti inattesi
- **Buona regola**: ridurre al minimo gli effetti collaterali
- **Es.**: 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 nei 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*

Domande

- Se **a** si riferisce ad un BankAccount, la chiamata **a.deposit(100)** modifica l'oggetto BankAccount **a**. E' un effetto collaterale?
 - **Risposta:** Si

Domande

- Consideriamo la classe **DataSet**. Supponiamo di aggiungere il metodo

```
public void read(Scanner in)
{
    while (in.hasNextDouble())
        add(in.nextDouble());
}
```

- Questo metodo ha un effetto collaterale oltre a cambiare il data set?
 - **Risposta:** Sì— il metodo impatta sullo stato del parametro **Scanner**.

Modifica parametri di tipo primitivo

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

○ Dopo aver eseguito le seguenti istruzioni

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

il valore di `savingsBalance` è 1000 e non 1500

Scambio per valore

```
double savingsBalance = 1000;  
harrysChecking.transfer(500, savingsBalance); ❶  
System.out.println(savingsBalance);  
...  
void transfer(double amount, double otherBalance)  
{  
    balance = balance - amount;  
    otherBalance = otherBalance + amount;  
}
```

❶ Before method call

harrysChecking =

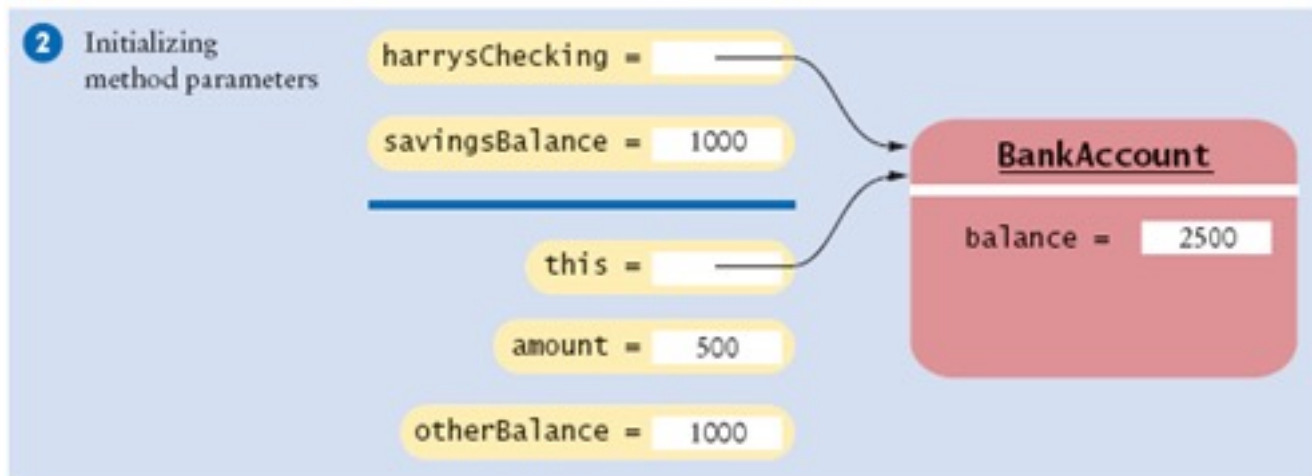
savingsBalance =

BankAccount

balance =

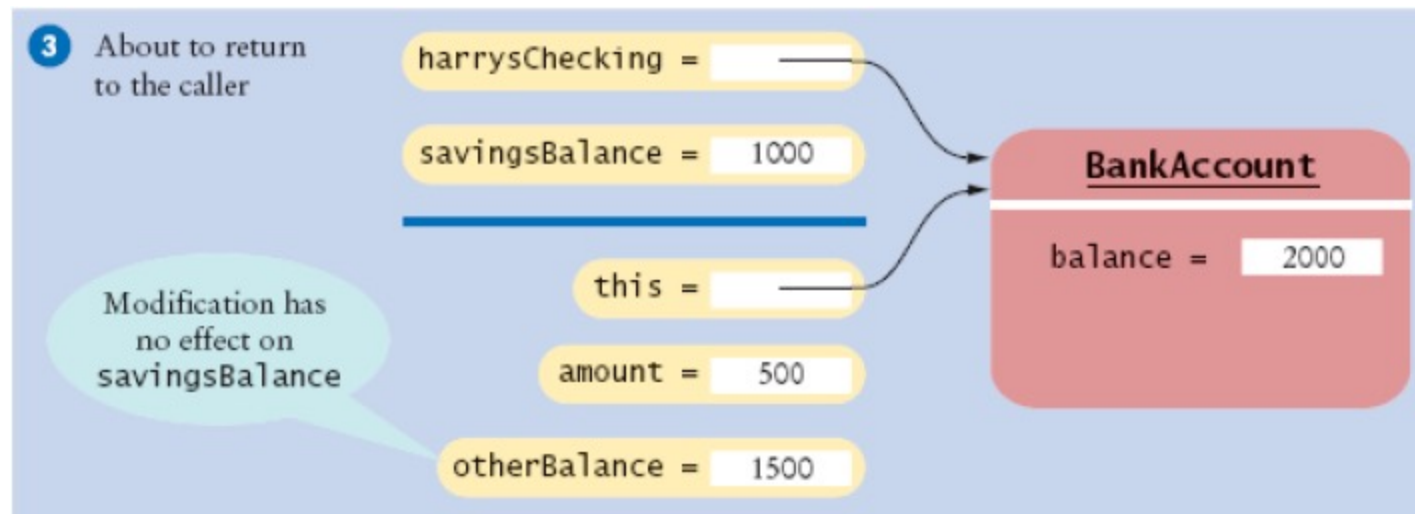
Scambio per valore

```
double savingsBalance = 1000;  
harrysChecking.transfer(500, savingsBalance); ❶  
System.out.println(savingsBalance);  
...  
void transfer(double amount, double otherBalance) ❷  
{  
    balance = balance - amount;  
    otherBalance = otherBalance + amount;  
}
```



Scambio per valore

```
double savingsBalance = 1000;  
harrysChecking.transfer(500, savingsBalance); ❶  
System.out.println(savingsBalance);  
...  
void transfer(double amount, double otherBalance) ❷  
{  
    balance = balance - amount;  
    otherBalance = otherBalance + amount;  
} ❸
```



Scambio per valore

```
double savingsBalance = 1000;  
harrysChecking.transfer(500, savingsBalance); ❶  
System.out.println(savingsBalance); ❷  
...  
void transfer(double amount, double otherBalance) ❸  
{  
    balance = balance - amount;  
    otherBalance = otherBalance + amount;  
} ❹
```

❹ After method call

harrysChecking =

savingsBalance =

BankAccount

balance =



Scambio per valore / riferimento

- Scambio per valore: il parametro è copiato all'atto dell'invocazione
 - Variabili indipendenti
- Scambio per riferimento: Parametro attuale e parametro formale coincidono
 - E' possibile la modifica da parte del metodo
- Java ha solo lo **scambio per valore**

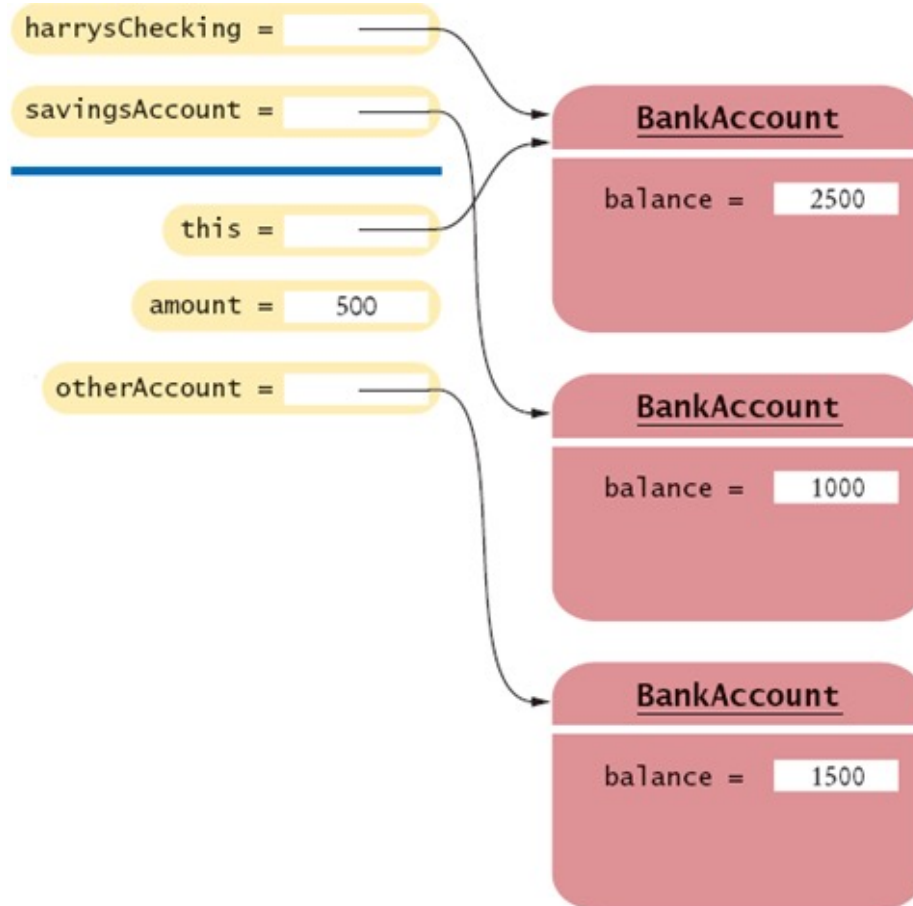
Scambio per valore

- **Attenzione:** un metodo **può** cambiare lo stato dell'oggetto ricevuto per valore
- **non può** cambiare l'oggetto da referenziare

```
public class BankAccount
{
    public void transfer(double amount, BankAccount otherAccount)
    {
        balance = balance - amount;
        double newBalance = otherAccount.balance + amount;
        otherAccount = new BankAccount(newBalance); // Non funziona
    }
    // dopo l'esecuzione del metodo l'oggetto referenziato da
    // otherAccount non muta stato
}
```

Passaggio di parametri: per valore/indirizzo

```
harrysChecking.transfer(500, savingsAccount);
```



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

- Controllare 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;`

- trasferisce il controllo ad un gestore delle eccezioni
 - può essere oneroso
-
- Si può assumere che quando si invoca il metodo le precondizioni siano sempre verificate
 - il controllo è a carico di chi invoca il metodo
 - approccio pericoloso: possibili valori errati

Pre-condizioni

- Altra possibilità non far fare niente al programma

- **Esempio:** `if (amount < 0) return;`
`balance = balance + amount;`
- sconsigliato: non aiuta il collaudo del programma

- Oppure usare **asserzioni**

```
assert amount >= 0;  
balance = balance + amount;
```

(il programma si interrompe con segnalazione di un **AssertionError** se l'asserzione non è verificata)

Assertzioni

- Per abilitarle da linea di comando

`java -enableassertions MyProg`

oppure `-ea`

- Per abilitarle in Eclipse, NetBeans, etc.

mettere opzione `-enableassertions` in VM parameters
(proprietà del progetto)

- Una volta testato il programma basta non abilitarle per far eseguire il programma senza valutare le asserzioni
- Buon compromesso tra
 - non fare nulla (nessun aiuto in fase di collaudo)
 - lanciare un'eccezione (appesantire il programma con gestione delle eccezioni)



Post-condizioni

- **Post-condizioni**: devono essere soddisfatte al termine dell'esecuzione del metodo
- Vanno riportate nella documentazione come per le pre-condizioni
- **Contratto**: Se il chiamante soddisfa le precondizioni, il metodo deve soddisfare le postcondizioni

Post-condizioni

- Due tipi di post-condizioni:
 - Il valore di ritorno deve essere computato correttamente
Es. metodo `getBalance()` di `BankAccount`
(Post-condizione: il valore restituito è il saldo del conto)
 - Al termine dell'esecuzione del metodo, l'oggetto con cui il metodo è invocato si deve trovare in un determinato stato
Es. metodo `deposit()` di `BankAccount`
(Post-condizione: `getBalance() >= 0`)

I metodi statici

- I metodi statici non hanno il parametro implicito
 - Esempio: il metodo `sqrt` di `Math`
- I metodi statici non possono fare riferimento a variabili di istanza
- I metodi statici vengono detti anche *metodi di classe* perché 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`

I metodi statici

- **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  
    ...  
}
```

I metodi statici

- Se un metodo manipola una classe che non si possiede, quel metodo non può essere aggiunto a quella classe
- Un metodo statico risolve questo problema:

```
public class Geometry
{
    public static double area(Rectangle rect)
    {
        return rect.getWidth() * rect.getHeight();
    }
    // Possono essere aggiunti altri metodi geometrici.
}
```

Programmazione O.O. e metodi statici

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

```
public static void main (String [ ] args){...}
```
- 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

Domande

- Supponiamo che Java non abbia metodi statici. Come si potrebbe utilizzare il metodo `Math.sqrt` per calcolare la radice di un numero `x`?

```
Math m = new Math();
```

```
y=m.sqrt(x);
```

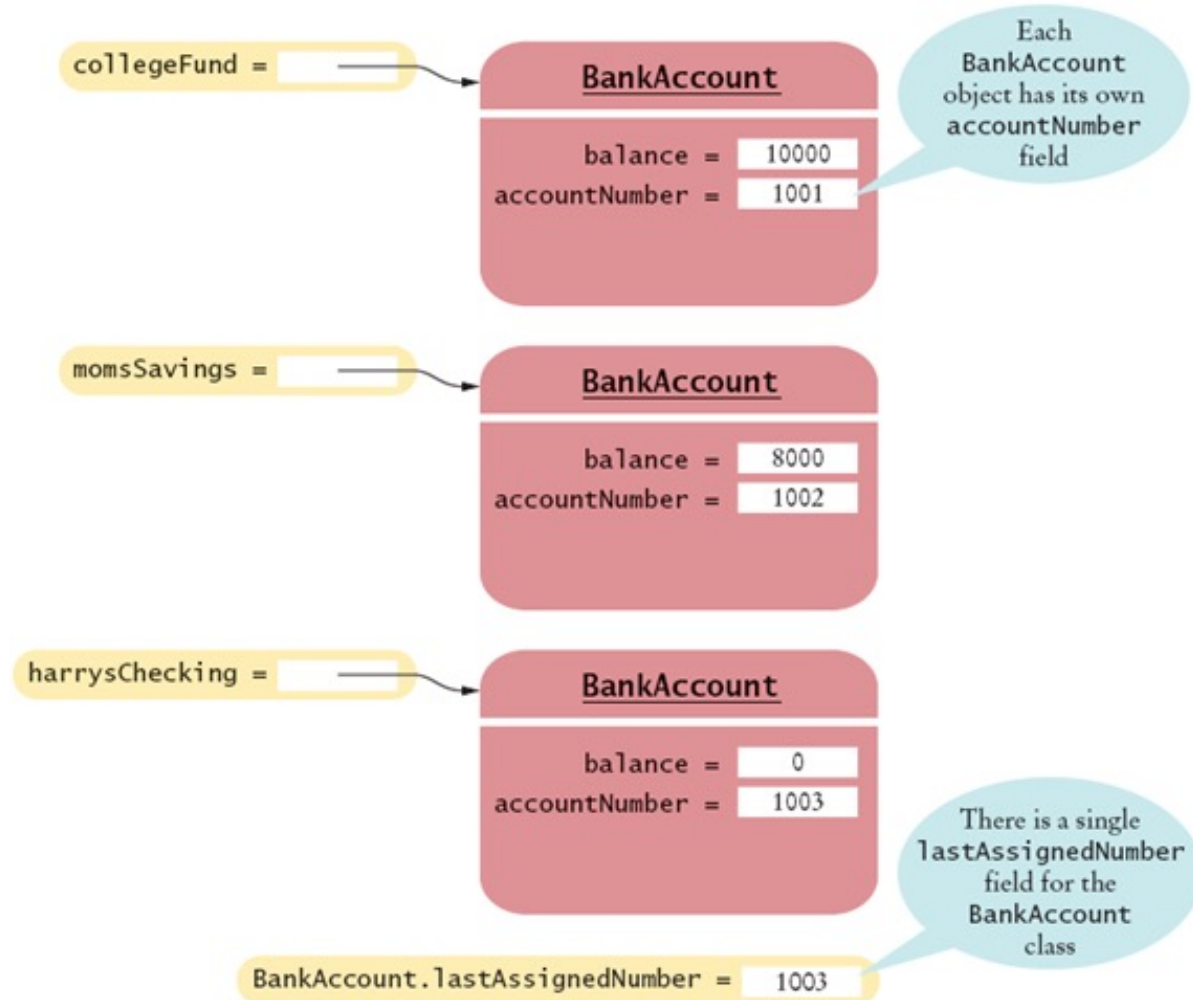
Variabili statiche

- 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 lastAssignedNumber;  
}
```

- Se `lastAssignedNumber` non fosse dichiarata `static`, ogni istanza di `BankAccount` avrebbe il proprio valore di `lastAssignedNumber`

Variabili statiche e variabili d'istanza



Inizializzazione di variabili statiche

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

```
public BankAccount{  
    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: 0, false o null

Le costanti statiche

- Una *costante statica* è 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

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 (non si possono ridefinire in blocchi annidati)
 - **Esempio:** `for (int i=0; i<10; i++){`

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

Visibilità sovrapposte

- I campi di visibilità di una variabile locale e di una variabile di 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 di 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 sovrapposte

- Errore tipico nei costruttori

```
public class Coin{
    public Coin(double inBalance, String aName)
    {
        String name = aName; // variabile locale, non di istanza
        balance = inBalance;
    }
    ...
    private String name; // variabile di istanza
    private double balance; // variabile di istanza
}
```

Visibilità di membri di classe

- All'interno di una classe si può accedere alle variabili di 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 appartenenza 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

- E' necessario un meccanismo che garantisca l'unicità dei nomi delle classi e dei package.
- Difficile pensare di usare nomi di classi differenti
- Basta assicurarsi che i nomi dei package siano differenti
- Per convenzione i nomi dei **package** sono scritti in lettere **minuscole**

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 :**
`rossi@di.unisa.it` diventa `it.unisa.di.rossi`

Pacchetti e posizione nel file system

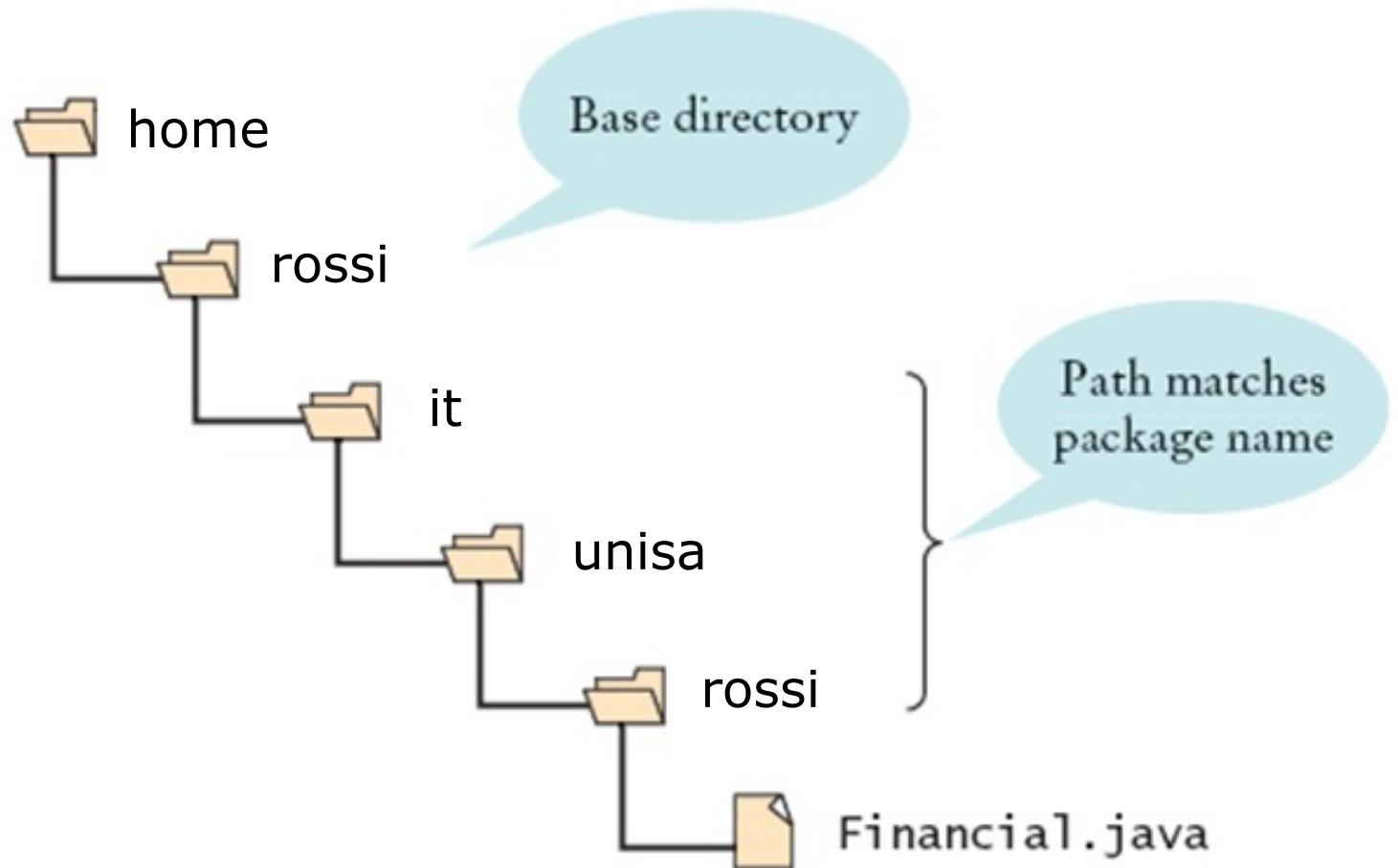
- Il nome del pacchetto deve coincidere con il percorso della sottocartella dove è ubicato il pacchetto
 - **Esempio:** il pacchetto **com.horstmann.bigjava** deve essere ubicato nella sottocartella:
com/horstmann/bigjava
 - Il percorso della sottocartella è specificato a partire da una directory prefissata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

Localizzazione dei pacchetti

- Supponiamo che la directory corrente sia `/home/rossi` e che in un file (con estensione java) vogliamo importare il package `it.unisa.rossi`
- I file che compongono il package devono stare nella sottodirectory `it/unisa/rossi` della directory corrente, cioè in

`/home/rossi/it/unisa/rossi`

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/rossi/esercizi: (UNIX)
 - Tutte le volte che importo classi non standard la ricerca parte da **/home/rossi/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/rossi/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 runtime
- 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

- Una libreria ha deciso di informatizzare il proprio archivio delle giacenze. Si è creato un file in cui, per ogni libro disponibile in magazzino, sono stati inseriti i seguenti elementi:
 - titolo del libro;
 - autore;
 - editore;
 - numero di copie disponibili.
- La libreria ha la necessità di effettuare le seguenti operazioni:
 - cercare tutti i libri di un particolare autore;
 - cercare tutti i libri che contengono una determinata stringa nel titolo;
 - cercare i libri che hanno il numero di copie massimo;
 - listare tutti i libri per cui il numero di copie disponibili è al di sotto di una determinata soglia.
 - aggiornare il numero di copie di un libro. Il libro da aggiornare è identificato dal titolo. L'operazione di aggiornamento deve essere propagata al file.
- Creare inoltre una classe main per testare le classi ed i metodi definiti. In particolare occorre
 - leggere da un file contenente i dati dei 5 libri mostrati di seguito;
 - visualizzare i libri scritti da Camilleri;
 - visualizzare i libri che hanno la parola sogni nel titolo;
 - visualizzare il libro con il numero massimo di copie;
 - visualizzare i libri che hanno meno di 15 copie disponibili;
 - aggiungere 30 copie al libro sogni rossi.

- **Una voce di notte**
- **Camilleri Andrea**
- **Sellerio Editore Palermo**
- **44**
- **Cinquanta sfumature di grigio**
- **James E. L.**
- **Mondadori**
- **21**
- **Il corpo umano**
- **Giordano Paolo**
- **Mondadori**
- **33**
- **Sogni rossi**
- **Mo Yan**
- **Einaudi**
- **11**
- **Fai bei sogni**
- **Gramellini Massimo**
- **Longanesi**
- **14**