Progettazione di classi

Scegliere una classe

- Una classe rappresenta un singolo concetto
- 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

Scegliere una classe

- Una classe può svolgere un lavoro: classi di questo tipo vengono dette Attori e in genere hanno nomi 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 tester)

Coesione

- l'interfaccia pubblica dovrebbe essere strettamente correlata 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:
  - *borsa* che contiene monete e calcola il loro valore totale
  - *valore delle singole monete*
- Soluzione: Si usano due classi

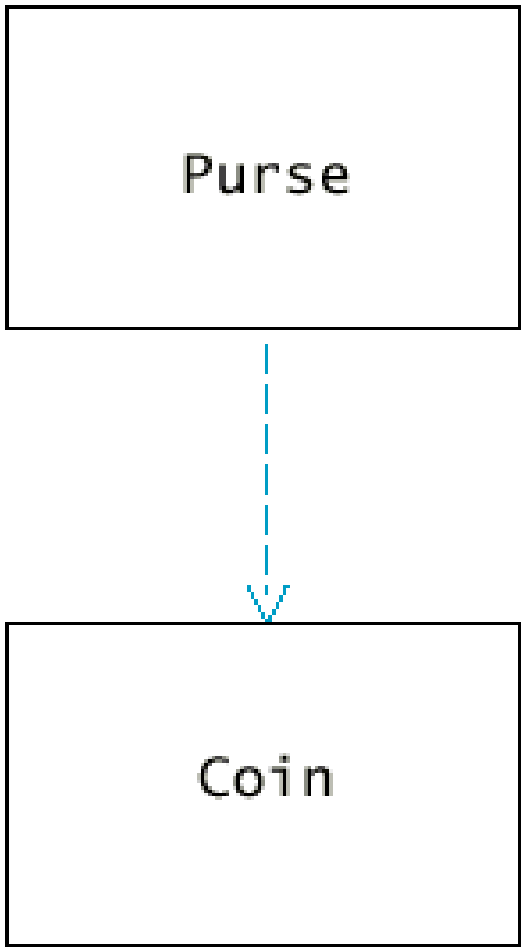
```
public class Coin
   {
      public Coin(double aValue,String aName){...}
      public double getValue(){...}
   }

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

Accoppiamento

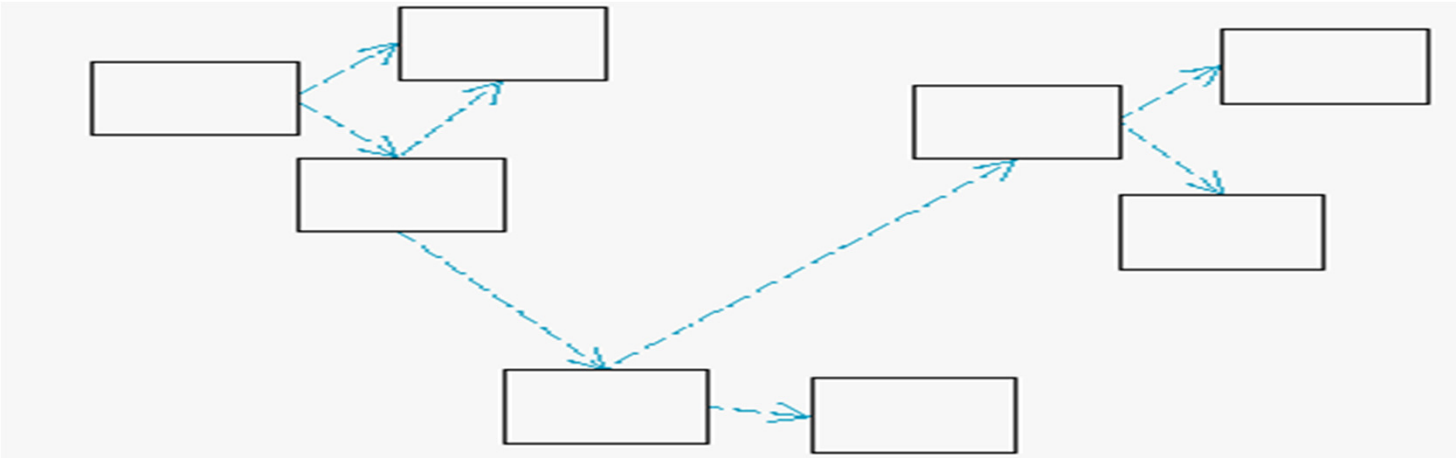
- Una classe A *dipende* da una classe B se usa esemplari di B (oggetti o metodi di B)
  - Es: Purse dipende da Coin perchè usa un'istanza di Coin
  - Es: Coin non dipende da Purse
- E' possibile avere molte classi che dipendono tra di loro (accoppiamento elevato)
  - Problemi dell'accoppiamento elevato:
    - Se una classe viene modificata tutte le classi che dipendono da essa potrebbero necessitare una modifica
    - Se si vuole usare una classe A in un programma, di fatto tutte le classi B da cui A dipende vengono usate nel programma

Dipendenza tra Purse e Coin

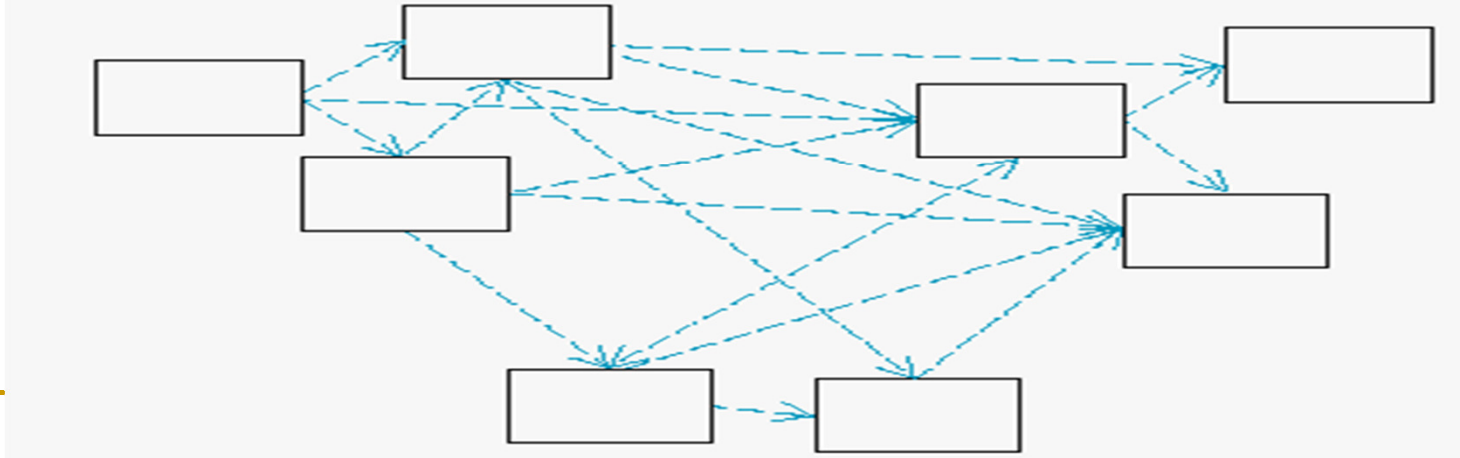
Notazione UML per rappresentare i diagrammi delle dipendenze tra classi o oggetti.



Accoppiamento elevato e accoppiamento basso



Accoppiamento basso



Accoppiamento elevato

Metodi modificatori

- Lo scopo di un metodo modificatore è cambiare lo stato del parametro implicito

- ogni altro uso dovrebbe essere evitato
  - non deve computare valori da restituire come ad es. nelle funzioni (tipo di restituzione void)
  - non deve modificare lo stato di altri oggetti (effetto collaterale)

Effetti collaterali

- ogni effetto osservabile al di fuori del parametro implicito
- introducono dipendenze e possono causare comportamenti inattesi
- è buona norma ridurre al minimo gli effetti collaterali

Esempi di effetti collaterali

- 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;
  }
```

- un metodo che stampa dati di una classe

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

Perchè non usare I/O da astrazioni?

- limita riutilizzo delle astrazioni
  - 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)
- introduce dipendenze non necessarie
  - la classe BankAccount diventa dipendente dalle classi System e PrintStream
- per coerenza: un'astrazione solitamente non contempla uno specifico formato di I/O

I/O meglio se gestito in classi dedicate

- il metodo getBalance() restituisce il valore di balance e I/O gestito attraverso un'interfaccia utente
  - ad esempio, in classe starter/tester con

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

oppure

  - con un'interfaccia utente grafica (prossime lezioni)

Effetti collaterali: altro esempio

- un metodo che stampa messaggi di errore

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

  - i metodi non dovrebbero mai stampare messaggi di errore: per segnalare problemi si devono usare le *eccezioni* (prossime lezioni)

Modifica parametri di tipo primitivo

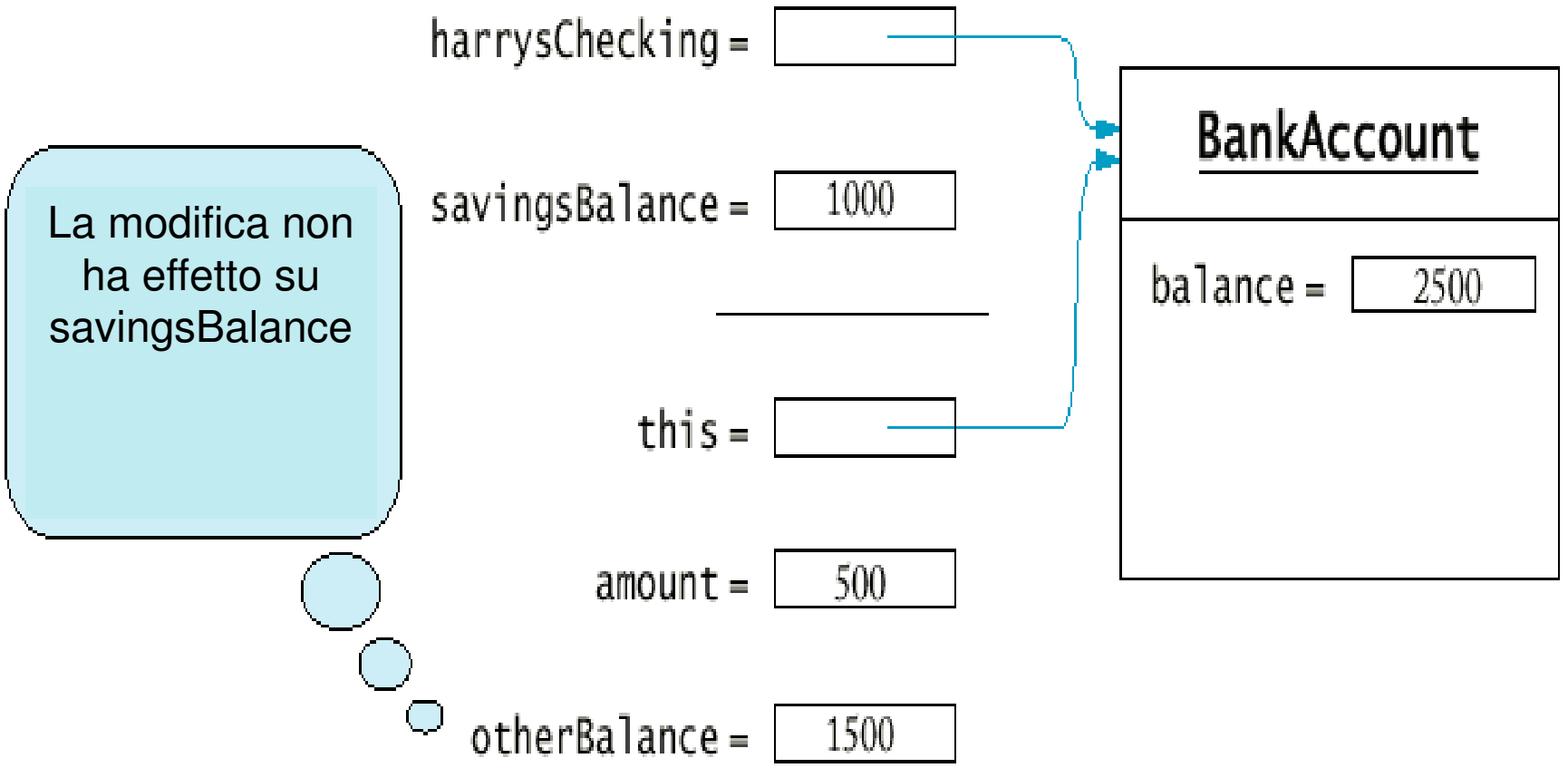
```
void transfer(double amount, double otherBalance)
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
} // non trasferisce amount su otherBalance
```

- 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



Chiamata per valore/per riferimento

- Un metodo può modificare lo stato di un oggetto passato come parametro esplicito ma non può rimpiazzarlo con un altro oggetto

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

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 le 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 pre-condizioni 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 le asserzioni

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

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

Asserzioni

- Per abilitarle da linea di comando

```
java -enableassertions MyProg
```

- Per abilitarle in eclipse

  inserire `-enableassertions` in VM arguments (menu Run, voce Run Configurations..)

- 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 soddifatte al termine dell'esecuzione del metodo

- Due tipi di post-condizioni:
  - Il valore restituito dal metodo 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: balance >= 0)

Invariante di una classe

- Una condizione che rimarrà vera
  - riguarda membri di una classe nelle varie istanze
- Es. /**

```
     *  Valore corrente di count:
     *  numero di elementi contati
     *  garantisci:  count >= 0   */
  public int count() {…….}
```

Riepilogo specifiche di proprietà

- post-condizione: una condizione che un implementatore garantisce quando il metodo termina
- invariante: una condizione che vale sempre
- invariante di classe: una condizione che vale sempre per tutte le istanze di una classe

Programmazione per contratti

- Le post-condizioni e le invarianti vanno riportate nella documentazione come per le pre-condizioni
- Il programmatore garantisce il soddisfacimento delle invarianti
- Contratto con l'utilizzatore: Se all'invocazione del metodo le pre-condizioni sono soddisfatte, il programmatore garantisce il soddisfacimento delle invarianti e delle post-condizioni

I metodi statici

- I metodi statici non hanno il parametro implicito
  - Esempio: il metodo **sqrt** di **Math**
- I metodi statici non possono fare riferimento alle variabili di istanza
- 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

- solitamente metodi funzionali che manipolano tipi primitivi, ad es.

```
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
   …
}
```

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
  - probabilmente le classi definite non modellano adeguatamente le entità su cui vogliamo operare

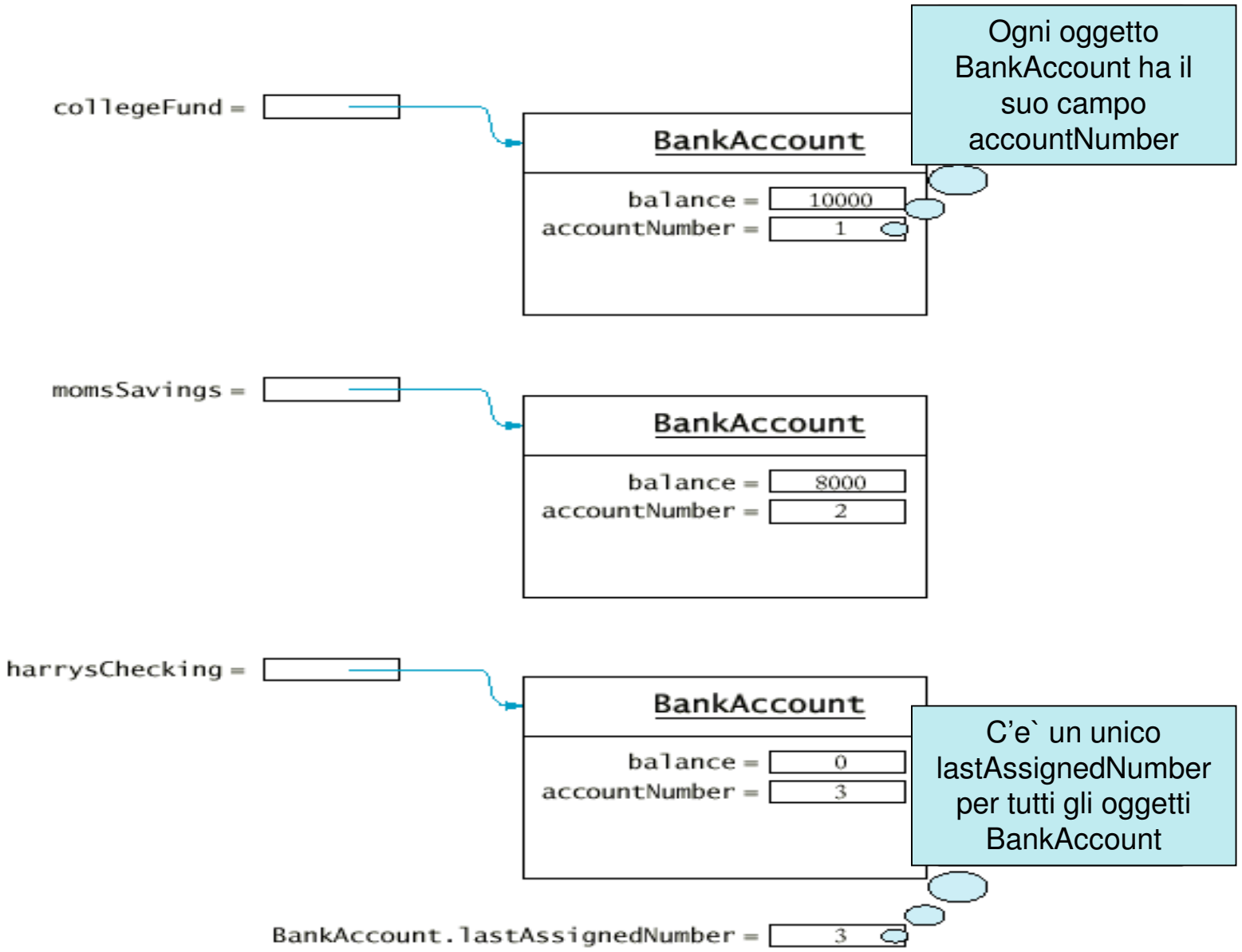
Variabili statiche

- 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 lastAssignedNumber;
}
```

- Se **lastAssignedNumber** non fosse dichiarata **static**, ogni istanza di BankAccount avrebbe la propria variabile lastAssignedNumber

Variabili statiche e variabili d'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: 0, false o null

Uso delle variabili statiche

- Le variabili statiche vengono usate raramente
  - i metodi che modificano variabili statiche hanno effetti collaterali (lo stato di tutti gli oggetti della classe viene modificato)
  - 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, questo potrebbe avere comportamenti diversi

Uso errato delle variabili statiche

- 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

La 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

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 (nomi non si possono ridefinire in blocchi annidati)
  - Esempio:

```
int i=0;
while(i<10){
    …
    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

- 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 o invocare un metodo usando il nome qualificato (con le restrizioni imposte dagli specificatori di accesso)

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) ;
}
```

Pacchetto (package)

- Insieme di classi correlate
- Le classi standard di Java sono raggruppate in 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 standard di 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
- 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** : marco@di.unisa.it diventa it.unisa.di.marco

Pacchetti e posizione nel file system

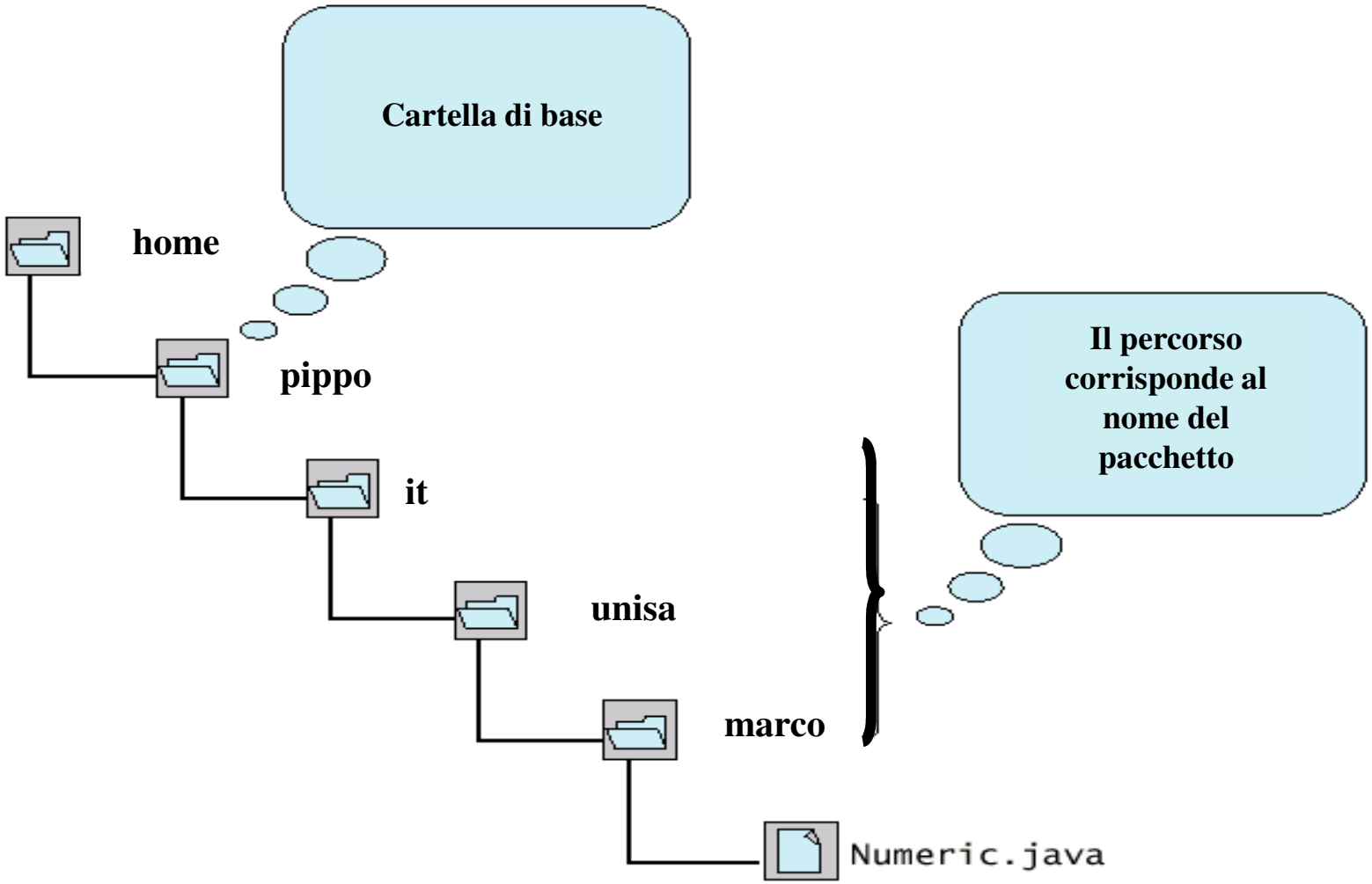
- Il nome del pacchetto coincide con il percorso della sottocartella dove è ubicato il pacchetto
  - **Esempio:** il pacchetto **com.horstmann**.bigjava deve essere ubicato nella sottocartella: com/horstmann/bigjava
    - Il percorso della sottocartella è specificato a partire da una directory prefissata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

Localizzazione dei pacchetti

- Supponiamo che la directory corrente sia /home/pippo e che in un file (con estensione 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/pippo/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/pippo/esercizi: (UNIX)
      - Tutte le volte che importo classi non standard la ricerca parte da /home/pippo/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/pippo/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

Importazione statica (Java 5.0)

- All'interno di una classe si può accedere ai nomi statici (variabili e metodi) di una classe specificandone semplicemente il nome se viene usata l'importazione statica
  - Esempio:

```
import static java.lang.System.*;
import static java.lang.Math.*;

public class Test(){
   public static void main(String[ ] args) {
                double r = sqrt(PI);
                out.println(r);
   }
}
```

Quando usare l'importazione statica?

- Il minimo indispensabile
  - quando è richiesto un accesso frequente ai membri statici di una o più classi
  - è preferibile (per la leggibilità) importare singolarmente i membri statici di cui si necessita invece di importare tutti i membri statici di una classe
- Se si esagera, il programma può diventare poco leggibile e difficile da mantenere
  - lo spazio dei nomi del programma viene "affollato" con tutti i nomi dei membri statici importati
  - potrebbe essere difficile risalire alla classe da cui importiamo un membro statico
- Se si usa in maniera appropriata migliora la leggibilità (omettendo il riferimento ripetuto ai nomi delle classi)