Introduzione al linguaggio Java

Ignazio Palmisano Oriana Licchelli Domenico Redavid
Dipartimento di Informatica

2005/2006

1 Introduzione

1.1 Overview

- I due paradigmi della programmazione
  - Procedurale vs Orientato agli oggetti
  - Oggetti: cosa sono mai?
  - Incapsulamento, Ereditarietà, Polimorfismo
- Il linguaggio Java
  - Portabilità
  - Semplicità

1.2 Cos'è Java?

- Linguaggio definito dalla Sun Microsystems
- Permette lo sviluppo di applicazioni su piattaforme multiple, in reti eterogenee e distribuite (Internet)
- Esistono più compilatori Java e Virtual Machines
  - IBM
  - Eclipse
  - GNU (non ancora completata...)
  - Blackdown
- Esistono test per il livello di aderenza allo standard
- Esiste un modo standard per estendere il linguaggio (non dipendente solo da Sun): JCP (Java Community Process)

1.3 I Due Paradigmi della Programmazione

1.3.1 Esistono due differenti modi per descrivere il mondo

- Come un sistema di processi (modello procedurale)
  - Tipicamente descritto con flow-chart
  - Usa procedure, funzioni, strutture
  - Cobol, Fortran, Basic, Pascal, C
- Come un sistema di cose (modello a oggetti)
  - Tipicamente descritto come gerarchie e dipendenze tra classi
  - Usa dichiarazioni di classi e di metodi
  - Simula, Smalltalk, Eiffel, C++, Java

1.4 Oggetti: cosa sono mai?

- Classe: descrizione astratta dei dati relativi a un concetto e dei comportamenti tipici relativi (funzioni)
- Oggetto: istanza di una classe
- Comunicazione attraverso messaggi (invocazione)
- Un oggetto è una coppia (stato, funzioni)

1.4.1 Esempio

- Automobile è una classe (rappresenta il concetto generico di automobile)
- Fiat Brava è una classe (rappresenta il concetto di un tipo di automobile)
- L'oggetto targato AX 266 WS è un'istanza di Fiat Brava
- Oggetto: istanza (esemplare) di una classe
- Creazione di un oggetto: ISTANZIAZIONE
- Due istanze della stessa classe NON sono lo stesso oggetto

- Due istanze diverse hanno la stessa INTERFACCIA

```
package slides;
public class Automobile {
    public void avviati() {
        /* implementazione*/
    }
    public void rifornisci() {
        ...
    }
    public static void main(String[] args) {
        Automobile a = new Automobile();
        Automobile b = new Automobile();
        a.avviati();
        b.avviati();
    }
}
```

2 Object Oriented

2.1 Fondamenti dell'Object Oriented (OO)

- Incapsulamento
- Ereditarietà
- Polimorfismo

2.1.1 Incapsulamento

- Interfaccia pubblica: ciò che è visibile all'esterno
  - Contratto di interfaccia: quello che un oggetto DEVE fare
  - Esempio: strutture dati e specifiche
- Interfaccia non pubblica: i fatti vostri
  - Information Hiding: non far vedere ai vicini ciò che non hanno bisogno di sapere
- Campi pubblici: da usare con prudenza
  - Un campo pubblico equivale a una variabile globale
  - La gestione delle variabili globali è DIFFICILE
  - MOLTO lavoro di debug

2.2 Interazioni Tra Oggetti

2.2.1 Visibilità

I modificatori di accesso determinano ciò che fa parte dell'interfaccia pubblica o privata

- `public`: una classe, un metodo o un campo pubblico può essere visto in qualunque parte del codice
- `protected`: un metodo o un campo protetto può essere visto solo nelle sottoclassi della classe che lo dichiara
- `private`: un metodo o un campo privato può essere visto solo nella classe che lo dichiara
- `default`: una classe, un metodo o un campo senza modificatori espliciti ha visibilità di default (visibile a livello di package)

2.2.2 Package

Un package riunisce più classi in un'unità logica

- I package possono essere annidati, ma i sottopackage non sono parte del package radice
- Un package corrisponde a una directory
- I file delle classi di un package devono stare nella directory di quel package
- La dichiarazione del package è la prima istruzione in una classe

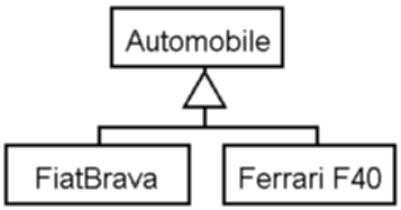
2.3 Ereditarietà

2.3.1 Estensione di una classe

- Aggiunta di campi e metodi a una classe esistente
- I campi e metodi esistenti non vengono modificati

2.3.2 Ridefinizione di una classe

- I metodi della superclasse vengono ridefiniti (override)
- I campi possono essere oscurati (definizione di un nuovo campo con lo stesso nome), ma non è consigliabile

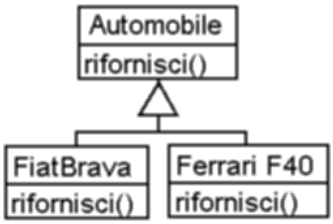


2.4 Polimorfismo (a tempo di compilazione)

2.4.1 Esempio

- L'invocazione di `rifornisci()` da `FerrariF40` causa l'esecuzione del relativo metodo
- Se `FerrariF40` non ridefinisce `rifornisci()`, si risale la gerarchia fino ad `Automobile.rifornisci()`

- Per fare riferimento manualmente all'implementazione della superclasse, si usa `super().rifornisci()`



2.5 Polimorfismo (a tempo di esecuzione)

2.5.1 Esempio

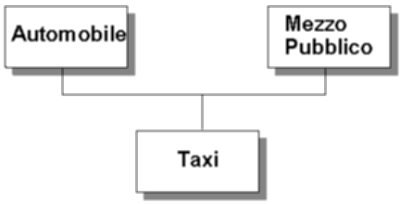
- Un oggetto di tipo `FerrariF40` è un oggetto di tipo `Automobile`
- Posso gestire questo oggetto attraverso una variabile di tipo `Automobile`
- Il viceversa non è sempre vero
- Upcast: da `FerrariF40` ad `Automobile`
- Downcast o cast: da `Automobile` a `FerrariF40`

```
Automobile a = new FerrariF40();
a.rifornisci();
FerrariF40 b = (FerrariF40) a;
b = (FerrariF40) new Brava();
```

2.6 Ereditarietà multipla

2.6.1 Si può estendere più di una classe?

- Java non supporta l'ereditarietà multipla
- È possibile usare le dichiarazioni di `interface` per emulare questo comportamento



3 Java in dettaglio

3.1 Alcune caratteristiche di Java

- Orientato agli oggetti
- Architetturalmente neutro e portabile
  - Non dipende dalla macchina fisica né dal sistema operativo
  - Il codice compilato (bytecode) può essere eseguito su qualunque Virtual Machine
- Robusto e sicuro
  - Gestione delle eccezioni
  - Non presenta le debolezze di C e C++ sulle stringhe
- Supporta programmazione distribuita e concorrente
- Supporta la reflection sulle classi
- Scalabile
- . . . Case sensitive . . .
- Non supporta alcune caratteristiche "pericolose" di C e C++
  - Niente puntatori espliciti (tutte le variabili sono puntatori)
  - Niente aritmetica dei puntatori
  - Niente deallocazione esplicita della memoria
  - Niente struct e typedef: tipi e strutture sono classi
  - Niente preprocessore (define)
  - Le stringhe non sono array di caratteri

3.2 Compilazione ed esecuzione

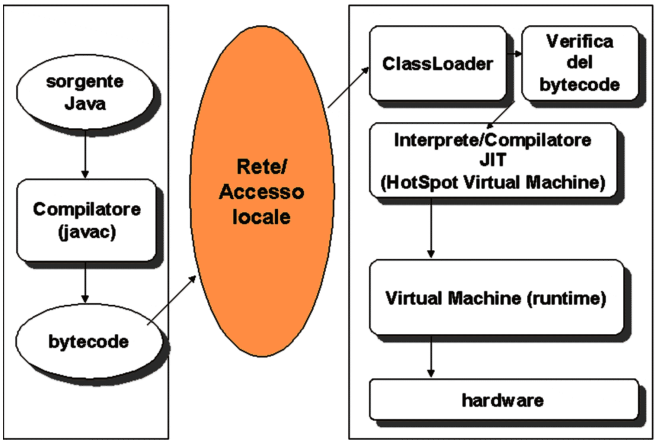
- Il compilatore Java (javac) produce bytecode per una macchina astratta (la Java Virtual Machine). . .
- `javac -classpath <classpath> Programma.java` → `Programma.class`
- . . . che viene eseguito da un programma nativo (l'implementazione della Virtual Machine per il sistema)
- La Virtual Machine è libera di ottimizzare il bytecode che viene eseguito (JIT, Just In Time compiling)
- Il bytecode stesso non viene mai modificato (resta compatibile con lo standard)
- `java -classpath <classpath> Programma` esegue Programma.class

3.2.1 Che succede sotto il cofano?

- Maggiori controlli a compile-time e a run-time
  - La VM controlla gli indici degli array
  - L'allocazione di oggetti è esplicita
  - La deallocazione viene fatta in automatico dalla VM (Garbage Collection)

- Verifica del bytecode a load-time
  - Check di versione: la versione della VM è più recente o uguale del compilatore? (1.1, 1.4 o 1.5?)
  - Check di compatibilità: il bytecode è compilato per questo profilo? (MIDP, PJava, J2SE, J2EE)

3.3 compile-load-run



3.4 Reflection

- Ogni file sorgente contiene una (e una sola) classe pubblica
- Il nome della classe coincide col nome del file
- La compilazione produce un file col nome della classe ed estensione .class contenente il bytecode relativo alla classe
- I file vengono caricati dal ClassLoader
- IMPORTANTE: Java è case sensitive anche sui nomi dei file

3.4.1 `java.lang.reflect`

- Il package java.lang.reflect contiene classi per descrivere i componenti di una classe (Method, Constructor, ecc)
- Esempio: ho bisogno di un oggetto, ma non conosco il nome della classe quando scrivo il codice

```
Class c = Class.forName("java.lang.String");
Object o = c.newInstance();
```

3.5 Sintassi

- Java è un linguaggio imperativo orientato agli oggetti
- Contiene espressioni all'interno di metodi
- I metodi appartengono a classi
- Le classi vengono raccolte in package e organizzate in gerarchie
- Un tipo particolare di classe: l'interfaccia
- La radice della gerarchia delle classi e delle interfacce è la classe Object
- In Java, i tipi di dati primitivi sono boolean, byte, char, float, double, int, short e long
- Tutti gli altri tipi sono classi

3.6 Esempio

```
package slides; //dichiarazione del package

/** <b>Commento Javadoc</b>
 * Descrizione della classe */
public class BravaAX266WS extends Automobile {
    public static final String owner =
      "Ignazio";

    /** Rifornisce l'oggetto di carburante
     * @param quanto quantitativo di
     carburante da rifornire */
    public void rifornisci(int quanto) {
      //stampa sulla console
      System.out.println(
      "Ahi ahi ahi, quanto mi costi...");
      for (int i = 0; i < quanto; i++) {
          // for classico...
          System.out.println("Glu glu glu");
          /*commento su
           * pi\'u linee
           * */
      }
    }
}
```

3.7 Reminders

3.7.1 Tipi primitivi

Interi	byte	8 bit	short	16 bit
	int	32 bit	long	64 bit
Carattere Unicode	char	16 bit		
Booleano	boolean	1 bit		
Floating point	float	32 bit	double	64 bit

3.7.2 Sequenze di escape

Backslash	$\backslash\backslash$	Tabulazione	$\backslash t$
Nuova linea	$\backslash n$	Doppia virgoletta	$\backslash"$
Spazio indietro	$\backslash b$	Virgoletta semplice	$\backslash'$
Ritorno del carrello	$\backslash r$	Carattere Unicode	$\backslash udddd$
Salto Pagina	$\backslash f$	Carattere ottale	$\backslash ddd$

3.8 Operatori e Modificatori di flusso

3.8.1 Operatori

Relazionali	$>, >=, <, <=, !=, ==$
Aritmetici	+, -, *, /, %

3.8.2 Modificatori di flusso ...

- condizionali: if - else, switch
- di ciclo: while, do - while, for
- di interruzione di ciclo: break, continue (BEWARE!!!)
- ritorno di valori: return
- gestione eccezioni: try - catch - finally
- Le condizioni per if, for, while sono espressioni booleane; non possono essere espressioni intere come in C

4 Cicli

4.1 Cicli

```
for(int i=0; i¡100; i++){
...
}
```

```
boolean b=true;
while (b){
...
b=false;
...
}
```

- For: inizializzazione, invariante, incremento
- While: invariante

4.2 Array

- Un array è un oggetto: va inizializzato e ha un campo che ne indica la lunghezza (length)
- Gli array possono contenere tipi primitivi o oggetti
- Vengono dichiarati mettendo [ ] dopo il nome dell'oggetto, o dopo il nome della variabile, oppure dopo entrambi
- Si accede con [ ]: ax[0], ax[15]

```
int[ ] ax;                          // interi
Object[ ][ ] ay;                    // Object
int[ ][ ] aay;                      // array di array
Object[ ] els;
ax = new int[5];
ax = new int[]{3, 4, 5, 6, 7};
ay = new Object[5][4];              // elementi null
els = new Object[ ]
    {"a", "b", "c"};
```

4.3 Costruttori (1)

- I costruttori (e i metodi) vengono invocati in un determinato ambiente, rappresentato dall'oggetto corrente
- L'oggetto corrente è rappresentato da `this`
- Se non si specificano costruttori, viene definito un "costruttore di default" ...
- ... che ovviamente non fa nulla ...

```
package slides;
public class FerrariF40 extends Automobile {
    public FerrariF40() { }
}
```

4.4 Costruttori (2)

- I costruttori si possono concatenare

```
package slides;
public class FerrariF40 extends Automobile {
    private String owner;
    private int price;
    public FerrariF40() { }
    public FerrariF40(String newOwner) {
        this();
        this.owner = newOwner;
    }
    public FerrariF40(String newOwn, int newPrice) {
        this(newOwn);
        this.price = newPrice;
    }
}
```

4.5 Metodi

- I metodi non statici possono essere richiamati solo su un oggetto
- I metodi statici non hanno bisogno di oggetti per essere invocati
- Due o più metodi possono avere lo stesso nome, ma devono avere argomenti diversi (overloading)
- I metodi non si distinguono per il valore ritornato

```
package slides;
public class Esempio {
  private static boolean esempio = false;
  public boolean esempioVariabile = false;
  public static boolean getEsempio() { return esempio; }
  public boolean getEsempioVariabile() { return esempioVariabile; }
  public int getEsempioVariabile() { return 1; }
}
```

4.6 Costruttori vs Metodi

- I metodi ritornano sempre un valore, anche se void
- Il tipo ritornato deve essere scritto esplicitamente
- I costruttori non hanno un tipo ritornato (ritornano implicitamente la classe a cui appartengono)
- I costruttori devono avere lo stesso nome della classe a cui appartengono
- I metodi non statici possono usare campi e metodi statici e non statici della propria classe
- I metodi statici possono usare solo campi e metodi statici della propria classe
- I metodi statici non hanno un oggetto corrente, e quindi non hanno un this

4.7 static e final

4.7.1 static

- Un campo static è unico per la classe: tutte le istanze della classe fanno riferimento allo stesso campo
- Esempio: `System.out`
- Un metodo static può essere invocato senza aver istanziato un oggetto della classe
- Esempio: `System.currentTimeMillis()`

4.7.2 final

- Le classi possono avere campi e metodi final
- Un campo final è una costante non modificabile
- Un metodo final non è ridefinibile
- Una classe final non è estendibile

4.8 Inizializzazione statica

- I campi static vengono inizializzati al primo riferimento alla classe
- Esistono i blocchi static, eseguiti al primo riferimento alla classe
- Sono utili per inizializzare i campi static

```
package slides;
class Quadrati {
    static int[] a = new int[10];
    static {
        for (int i = 0; i < 10; i++) {
            a[i] = i * i;
        }
    }
}
```

4.9 Package

- Un package è identificato dal nome
- `java.lang` è il package predefinito del linguaggio
- Nome lungo (nome completo, full qualified name): nome package + '.' + nome classe
- I punti nei nomi di package indicano i sottopackage
- Per utilizzare una classe bisogna usare il nome lungo: java.lang.String s = new java.lang.String
- Oppure si usa la direttiva `import`:
  import java.lang.* importa tutte le classi del package java.lang

4.9.1 Collisioni

- Usando l'istruzione import nome package.* sono possibili collisioni
- Si rimedia rimuovendo * e nominando le classi una a una

4.10 Classpath

- Un package corrisponde ad una directory
- Una classe corrisponde ad un file
- Package java.util.Vector equivale a:

File DOS	java*util**Vector.class*
File UNIX	java/util/Vector.class

- Per indicare al class loader i percorsi da usare si specifica il CLASSPATH
- CLASSPATH=c:/java; c:/java/lib, java.util.Vector viene cercata come:
  1. c:/java/lib/java/util/Vector.class
  2. c:/java/util/Vector.class
- CLASSPATH=c:/java/lib/classes.zip

java.lang.String	String.class in classes.zip nella sottodirectory java/lang

4.11 Radice della gerachia di classi: java.lang.Object

- Tutte le classi estendono un'altra classe
- Se non si estende esplicitamente un'altra classe, si estende (implicitamente) la classe Object
- Tutte le classi ereditano (direttamente o indirettamente) da Object

4.11.1 Alcuni metodi definiti da Object

- `toString()`: rappresentazione a stringa dell'oggetto
- `hashCode()`: intero che identifica (quasi) univocamente un oggetto
- `equals(Object o)`: restituisce true se l'oggetto è uguale all'argomento

4.12 Polimorfismo

- Un oggetto O istanza di una classe C o di un'interfaccia I è istanza delle superclassi di C e delle superinterfacce di I
- Se si richiama un metodo su O, l'implementazione utilizzata è quella più in basso nella gerarchia

4.12.1 Costruttori

- I costruttori non si ereditano: vanno dichiarati tutti
- Ogni costruttore per prima cosa costruisce la classe base, chiamando uno dei costruttori della classe base
- Il primo comando di un costruttore deve essere la chiamata a un costruttore della superclasse (`super(...)`)
- Oppure a un altro costruttore della classe (`this(...)`)
- Se non viene fatto esplicitamente, il compilatore inserisce la chiamata a `super()`

4.13 Interfacce (1)

4.13.1 Problema

- Pesce superclasse di Squalo e di PesceRosso
- Crostaceo superclasse di Granchio
- Acquario contenitore di Pesci
- → Come metto un Granchio in un Acquario?
- Java non consente che Granchio estenda Pesce e Crostaceo
- La soluzione è definire un'interfaccia Nuotatore
- Nuotatore astrae i metodi caratteristici di Pesce e Granchio
- Pesce implementa Nuotatore
- Granchio implementa Nuotatore
- Acquario accetterà Nuotatori, non Pesci

4.14 Interfacce (2)

- Un'interfaccia è una classe contenente solo dichiarazioni di metodi e costanti
- La realizzazione è demandata alle classi che implementano l'interfaccia

```
package slides;
public interface Nuotatore {
    public void nuota();
    public void mangia();
    public void abbocca();
}

package slides;
public class Pesce implements Nuotatore {
    public void nuota() {...}
    public void mangia() {...}
    public void abbocca() {...}
}

package slides;
public class Crostaceo implements Nuotatore {
    public void nuota() {...}
    public void mangia() {...}
    public void abbocca() {...}
}
```

4.15 The Bad Side: Errori ed Eccezioni

- Quando un comando può non andare a buon fine...
- Viene sollevato (o lanciato) un Throwable (Error o Exception)
- Un metodo deve dichiarare le eccezioni che può lanciare
- Le eccezioni devono essere gestite o propagate
- Mostrare che c'è stata un'eccezione stampando lo stack trace è utilissimo per scoprire errori inaspettati!

```
package slides;
public class ExceptionExample {
    public void lanciaEccezione() throws Exception {
        throw new Exception("Disastro!!!");
    }
    public void gestisciEccezione() {
        try {
            this.lanciaEccezione();
        } catch (Exception e) {
            e.printStackTrace();
        }finally {System.out.println("fine...");}
    }
}
```

4.16 Esempio di gestione di eccezioni

```
package slides;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
public class ExceptionExample2 {
    public static void main(String[] args) {
         try {
             InputStream in = new
                 FileInputStream(args[0]);
             OutputStream out = new
                 FileOutputStream(args[1]);
             int c;
             while( (c=in.read()) != -1) {
                 out.write(c);
             }
             in.close();
             out.close();
         } catch (Exception ex) {
           ex.printStackTrace();
         }
    }
}
```

- Gestione piuttosto grossolana
- Non distingue i tipi di eccezione
- Non tenta il recupero dell'esecuzione

4.17 Raffinamento...

- Si può raffinare intercettando le eccezioni di vario tipo
- catch() con un parametro del tipo dell'eccezione

```
package slides;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;

public class ExceptionExample3 {
  public static void main(String[] args) {
    try {
        InputStream in = new
            FileInputStream(args[0]);
```

```
    } catch (ArrayIndexOutOfBoundsException ex) {
        System.out.println("bad args");
    } catch (FileNotFoundException ex) {
        System.out.println("file not found");
    }
  }
}
```

4.18 . . . Raffinamento. . .

- Inserendo una catch(Exception ex) si intercettano tutte le eccezioni. Gli Error non derivano da Exception e non vengono intercettati
- Le eccezioni vengono provate in ordine
- Scambiando 1 e 2, la 1 non viene più raggiunta perché FileNotFoundException deriva da IOException

```
public void test() {
    try {
        InputStream in =
            new FileInputStream("fileName");
        in.read();
    } catch (FileNotFoundException ex) { // 1
        ex.printStackTrace();
    } catch (IOException ex) { // 2
        ex.printStackTrace();
    }
}
```

4.19 . . . Raffinamento. . .

- Una catch può catturare l'eccezione, esaminarla e risollevarla
- Una catch può trasformare l'eccezione
- Il blocco finally viene eseguito in qualunque caso
  - La try termina normalmente
  - Un'eccezione è sollevata e intercettata da una catch
  - Viene eseguito un return

```
try {...
} catch (SpecialException ex) {
    throw ex;
}

try {...
} catch (SpecialException ex) {
    throw new OtherException(ex);
}
```

4.20 Finally

- Il blocco finally viene utilizzato per effettuare pulizie finali

```
try{
    in= new FileInputStream (fin)
    out=new FileOutputStream (fout)
} catch (...) { ...
}finally {
    if(in!=null) {in.close();}
    if(out!=null) {out.close( );}
}
```

4.21 Intermezzo: Convenzioni di scrittura del codice

- Esistono delle convenzioni per la formattazione del codice Java; l'intento è rendere i programmi più semplici da leggere
  `http://java.sun.com/docs/codeconv/`
  `http://developer.java.sun.com/developer/onlineTraining/Downloads`

4.21.1 Alcune convenzioni...

- I nomi di classi iniziano con la maiuscola
- I nomi di metodi e attributi iniziano con la minuscola
- I nomi dei package sono tutti in minuscolo
- La prima parentesi graffa si trova sulla riga che precede l'inizio del blocco; l'ultima si trova su una riga separata
- Si scrivono le parentesi per i blocchi costituiti da una linea

4.22 Trucchi per la codifica

```
while (...) {                    // crea una variabile
UnaClasse unaClasse =            // per ogni iterazione
new UnaClasse(); }               // del ciclo
```

```
UnaClasse unaClasse;             // genera una sola
while (...) {                    // variabile per
unaClasse = new UnaClasse();     // l'intero ciclo
}
```

- `String string = new String();`
- String produce istanze immutabili. . .
- Posso riscrivere così: `String string = null;`
- L'uso di string prima dell'inizializzazione genera un'eccezione
- Questo vale per tutti gli oggetti immutabili (Character, Boolean, Double, Integer. . .)

5 Input/output e collezioni

5.1 Input/Output

- Il package per l'I/O è `java.io`
- `import java.io.*`
- Il concetto principale è lo Stream, un oggetto in cui si scrive (OutputStream) o da cui si legge (InputStream)
- Caso particolare: RandomAccessFile (non è uno Stream)
- Tutte le eccezioni di I/O sono derivate da `IOException`

5.1.1 File

- Classe wrapper per operazioni legate ai file
- Un oggetto File rappresenta il nome di un file, non il file
- Gestisce anche varie operazioni relative alle directory.
- Permette di creare un file o una directory
- Accetta anche URL, permettendo di aprire un file su un sistema remoto

5.2 Stringhe

- In generale, non si può fare un confronto fra stringhe del tipo s==t ma si usa la seguente sintassi:
  `boolean s.equals(t)`
- Il motivo è che l'operatore == verifica che le variabili siano identiche o meno, non verifica il contenuto
- Vale per tutti i tipi non primitivi
- Se si vuole ignorare il maiuscolo-minuscolo, si deve usare:
  `s.equalsIgnoreCase(t)`

5.3 Contenitori

List, Map, Set e Iterator sono interfacce estensioni di Collection (fanno parte del Collection Framework); ne viene consigliato l'uso rispetto agli analoghi del JDK 1.1 (Vector, Hastable, Enumeration)

- `java.util.Map`: Tabella chiave/valore
- `java.util.List`: Implementazione della struttura dati Lista
- `java.util.Iterator`: Enumerazione di elementi contenuti in un contenitore
- `java.util.Set`: Implementazione della struttura Insieme

Ognuna ha molte implementazioni con funzionalità ed efficienza differenti (insiemi ordinati, liste concatenate o con implementazione basata su array)

5.4 Contenitori in Java 1.5

- Java 1.5 (o 5.0) introduce il tipo in Collection e Iterator
- In Java 1.4, l'unico tipo contenuto in un contenitore è Object
  - Molto generale, a volte troppo
  - Non si possono inserire tipi primitivi
- in Java 1.5, è possibile specificare il tipo dei dati contenuti in una Collection
  - Semplifica la scrittura del codice
  - Diminuisce gi errori a runtime (ClassCastException individuate a compile time)
- É possibile inserire tipi primitivi (boxing - unboxing automatico)
- In realtà, il bytecode è lo stesso, ma cast e boxing sono gestiti dal compilatore → meno bug

5.5 Esempi

Esempio di uso di liste in Java 1.5 e 1.4

```
public void test1_5(String[] args) {
  List<String> l=new ArrayList<String>();
  for(int i=0; i<10;i++) {
    l.add(String.valueOf(i));
  }
  String s=l.get(5);
  Collections c=(Collections)l.get(5);
}
```

```
public void test1_4 () {
  List l=new ArrayList();
  for(int i=0; i<10;i++) {
    l.add(String.valueOf(i));
  }
  String s=(String)l.get(5);
  Collections c=(Collections)l.get(5);
}
```

Esempio di cicli for semplificati, autoboxing e unboxing

```
public void simpleFor() {
  List<Integer> l=new ArrayList<Integer>();
  for(int i=0; i<100; i++) {
    l.add(i);
  }
  for(int k:l) {
    System.out.println(k);
  }
}
```

```
public void traditionalFor() {
  List l=new ArrayList();
  for(int i=0; i<100; i++) {
    l.add(Integer.valueOf(i));
  }
  int k=0;
  for(int j=0; j<l.size();j++) {
    k=((Integer)l.get(j)).intValue();
    System.out.println(k);
  }
}
```

6 JDBC e Sockets

6.1 JDBC

- JDBC: Java DataBase Connectivity
- É uno strato di astrazione sui database di qualunque genere; package java.sql
- Richiede librerie specifiche per i database da collegare (MySQL, SQL Server...)
- É incluso il driver per database ODBC

```
package slides;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBC {
  public void test() {
    try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection conn =
        DriverManager.getConnection(
          "jdbc:mysql://host/model? "+
          "user=user&password=password");
      Statement st = conn.createStatement();
      ResultSet set = st.executeQuery(
          "SELECT * FROM Tabella");
      while (set.next()) {...
      }
      set.close();
      st.close();
      } catch (ClassNotFoundException e) {
        e.printStackTrace();
      } catch (SQLException e) {
        e.printStackTrace();
      }
```

```
  }
}
```

6.2 Connessioni HTTP: Socket

- `java.net.Socket`, `java.net.ServerSocket`
- Consente di creare stream per la comunicazione tra processi
- Si basa su un protocollo client/server

```
package slides;
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.net.Socket;
public class Sockets {
    public void test()
      throws UnknownHostException, IOException {
        Socket s = new Socket(
          InetAddress.getByName("www.yahoo.it"),
          80);
        int i = s.getInputStream().read();
        s.getOutputStream().write(i);
        s.close();
    }
}
```

7 Applets

7.1 Applets

Le applets sono

- Piccole applicazioni a cui si accede su un server Internet
- Automaticamente installate
- Un'applet ha un accesso limitato alle risorse (sandbox)

```
package slides;
import java.awt.*;
import javax.swing.JApplet;
public class Applets extends JApplet {
    public void paint (Graphics g) {
     g.drawString("A Simple Applet", 20,20);
    }
}
```


```
<HTML>
  <HEAD><TITLE>Sample Applet </TITLE></HEAD>
  </BODY>
    <APPLET CODE="slides.Applets" WIDTH= 200 HEIGHT=100>
       Il tuo browser non supporta le applet
    </APPLET>
  </BODY> </HTML>
```

8 Threads

8.1 Threads

- Un thread è un flusso di esecuzione del programma
- Ci possono essere più flussi di esecuzione in contemporanea
- Stesso spazio di indirizzamento
- Problemi di sincronizzazione

Creazione

- Implementare Runnable
- Ridefinire il metodo run()
- Avviare il thread con start()

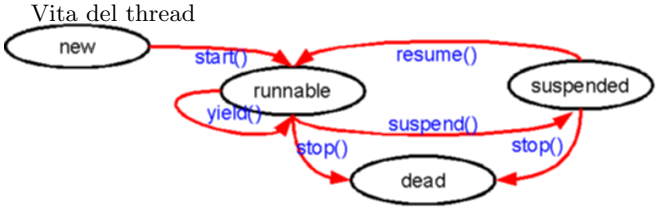
8.2 Esempio

```
package slides;

class Contatore extends Thread implements Runnable {
    public void run() {
        int n = 0;
        while (true) {
            System.out.println(n);
            ++n;
        }
    }

    public static void main(String[] args) {
        new Contatore().start();
    }
}
```

Vita del thread



8.3 Thread: ciclo di vita

- Un thread appena creato è nello stato new; non è attivo
- Per attivare un thread, occorre chiamare start(); il thread è runnable
- Un thread runnable ottiene ogni tanto il processore
- Un thread runnable può cedere il passo agli altri con yield() rimanendo attivo
- Un thread attivo può andare in stato di suspended (non ottiene mai il processore) con suspend()
- Un thread sospeso ritorna in esecuzione con resume()
- Un thread può morire (e non ritornare in esecuzione) con stop()

8.3.1 Perché un'interfaccia Runnable?

- Distinzione tra interfaccia e implementazione
- Si vuole gestire come un thread qualcosa che deriva da altre classi

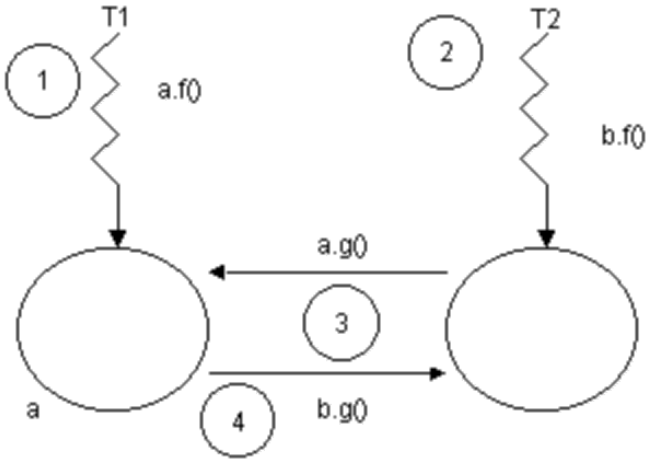
8.4 Sincronizzazione

- Garantire la completa esecuzione dei metodi che eseguono funzionalità critiche
- Quando un thread accede ad un oggetto sincronizzato, lo blocca
- Ogni nuovo thread viene sospeso

- Quando un thread libera l'oggetto, si attiva un thread sospeso
- La sincronizzazione rallenta un programma
- Se si ridefinisce un metodo sincronizzato, il metodo ridefinito non è automaticamente sincronizzato
- wait() e notify(): metodi della classe Object
- wait() provoca la seguente situazione:
  - sospende il thread corrente e lo pone in attesa
  - un altro thread va in esecuzione
  - il thread viene riattivato non appena si esegue notify() sull'oggetto
- Un thread riattivato accede all'oggetto e riprende l'esecuzione dal punto in cui era stato interrotto
- notify() riattiva un solo thread, per riattivare tutti quelli in attesa si usa il metodo notifyAll()

8.5 Esempio

```
package slides;
public class Stack {
    private int top;
    private int[] stack;
    synchronized void push(int x)
      throws InterruptedException {
        while (top > stack.length) {
          wait();
        }
    }
    synchronized int pop() {
        int r = stack[--top];
        notify();
        return r;
    }
}
```

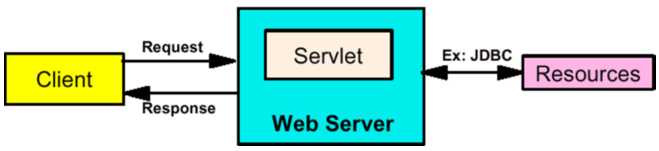


- T1 esegue a.f() e blocca a
- T2 esegue b.f() e blocca b
- T1 chiama b.g() ma b è bloccato, e si sospende
- T2 chiama a.g() ma a è bloccato, e si sospende
- Stallo: T1 e T2 sono sospesi in eterno

9 Servlet e JSP

9.1 Servlet

- Classi Java, eseguite lato server, independenti dalla piattaforma e dal protocollo
- Eseguite in un Application Server (IBM WebSphere Application Server, Apache Tomcat, ecc)
- Caricate nella Java Virtual Machine dell'Application Server
- Analoghe alle CGI, ma scritte in Java



9.2 Servlet e ciclo di vita

- HTTPServlet è la classe astratta che le servlet che lavorano con HTTP devono estendere
- void service(HttpServletRequest, HttpServletResponse) è il principale metodo da ridefinire
- Il package è javax.servlet.http
- Inizializzazione di una Servlet: il metodo init() con parametro la configurazione del Servlet (ServletConfig)
- Soddisfazione delle richieste: la richiesta è un oggetto HttpServletRequest; la risposta è un HttpServletResponse (pagina Html inviata al client)
- Distruzione della Servlet: il metodo destroy() viene richiamato quando l'Application Server viene chiuso o quando la servlet va modificata

9.3 Esempio

```
package slides;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleHttpServlet
        extends HttpServlet {
    protected void service(
        HttpServletRequest req,
        HttpServletResponse res)
          throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println(
          "<HTML><TITLE>Simple</TITLE><BODY>");
        out.println(
          "<H2>SimpleHttpServlet </H2><HR>");
        out.println(
          "<H4>pi\'u semplice di cos\'i...</H4>");
        out.println("</BODY><HTML>");
        out.close();
    }
}
```

9.4 HTTPSession

- HttpSession è una sessione tra client e server (stato) che persiste finché non viene esplicitamente chiusa oppure scade.
- Metodi utili per gestire la sessione:
  - getSession: ritorna la session corrente. Ha un parametro:
  - true: crea una nuova session se non esiste
  - false: ritorna null se la session non esiste
- setAttribute(nome, valore): permette di inserire un oggetto nella session corrente
- getAttribute(nome): permette di ritrovare un oggetto nella session corrente

9.5 Serializzazione

- Gli oggetti in sessione devono implementare l'interfaccia Serializable
- Un oggetto serializzabile può essere salvato in uno stream (ad esempio in un file) e poi ricreato a partire dallo stream
- La serializzazione nativa Java è binaria, ma si può costruirne una propria (ad esempio, in XML)
- Serializable non dichiara nessun metodo
- Sono serializzabili i tipi primitivi, le stringhe e tutti i tipi che comprendono solo questi tipi

9.6 Java Server Pages

- Java Server Pages: pagine dinamiche in cui il linguaggio di scripting è Java
- Una JSP ha bisogno di un Application Server
- Alla prima invocazione, la JSP viene compilata e viene generata una Servlet

9.7 Esempio Servlet

```
package slides;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.*;
import javax.servlet.http.*;
public class Saluto extends HttpServlet {
   protected void service(HttpServletRequest req,
                          HttpServletResponse res)
             throws ServletException, IOException {
    String nameLoc = req.getParameter("NAME");
    String output = "NESSUNO";
    if (nameLoc != null)
            output = nameLoc;
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println(
      "<HTML><TITLE>Saluto</TITLE><BODY>");
    out.println("<H1>Hello " + output + "</H1>");
    out.println("</BODY><HTML>");
    out.close();
   }
}
```

9.8 Esempio JSP

Input

```
<html><head>
<title>Input per Saluto</title>
</head>
<body>
 <form name=input method=Post
       action = "Saluto.jsp">
  <h2> Inserire il proprio nome: </h2>
  <input type="text" name="name">
  <input type="submit" value="OK">
 </form>
</body></html>
```

Saluto

```
<html>
<head><title>Saluto</title></head>
<body>
<% String chi =
   request.getParameter("name");
   if (chi==null) {chi="NESSUNO";} %>
<h1> Hello <%= chi %> </h1>
</body></html>
```

9.9 JSP e JavaBean

- Le JSP da sole non permettono una buona divisione tra codice java e codice html.
- Deleghiamo il saluto ad un Bean chiamato `salutoBean`
- Creazione di un Bean:
  - <jsp:useBean id="nome" class="classe"/>
- Inizializzazione di una proprietà del Bean:
  - <jsp:setProperty name="nome" property="prop" param="valore_parametro" />
- Utilizzo di una proprietà del Bean:
  - <jsp:getProperty name="nome" property="prop" />
  - <%= nomeBean.metodo() %>

9.10 Esempio JavaBean

Esempio di bean:

```
package slides;
public class SalutoBean {
      private String chi;
      public SalutoBean() {
        super();
      }
```

```
      public void setChi(String chi) {
        this.chi = chi;
      }
      public String getChi() {
        return chi;
      }
      public String getSaluto() {
        return "Hello " +
        ((chi==null) ? "NESSUNO" : chi);
      }
}
```

JSP corrispondente:

```
<html><head>
<title>Prova di una pagina JSP</title>
</head>
<body>
 <jsp:useBean id="saluta"
   class="slides.SalutoBean" />
 <jsp:setProperty name="saluta"
   property="chi" param="Pippo" />
 <%= saluta.getSaluto() %>
</body></html>
```

9.11 Direttive JSP

```
<html><head><title> Direttive </title></head>
<%@page language="java" %>
<%@page isErrorPage="true" %>
<%@page isThreadSafe="true" %>
<%@page import="saluto.jsp"%>
<%@page errorPage="exception.jsp"%>
<body>
 <% //questo \‘e uno scriptlet %>
 <%="ciccio"%><!--questa \‘e un’espressione
     che produce l’output di "ciccio"
     nella pagina-->
  <h2> Inserire il proprio nome: </h2>
  <input type="text" name="name">
  <input type="submit" value="OK">
 </form>
</body></html>
```

9.12 Oggetti impliciti

- Oggetti impliciti in JSP e Servlet:
- request: istanza di HttpServletRequest
  - getParameter(nomeAttributo)
  - Ciclo di vita = richiesta HTTP
- response: istanza di HttpServletResponse
  - Ciclo di vita = composizione risposta html
- session: istanza di HttpSession
  - disponibile se è stato settato a true l’attributo session della direttiva page
  - setAttribute(nome, valore)
  - getAttribute(nome)
  - Ciclo di vita = sessione

9.13 Gestione delle eccezioni nelle JSP

- Con l’attributo error-Page della direttiva page si può specificare la pagina di gestione dell’errore
- Una pagina di gestione delle eccezioni ha il valore true dell’attributo isErrorPage della direttiva page
- Quando viene sollevata un’eccezione in una JSP, il controllo passa alla relativa pagina di errore, che vede l’oggetto implicito exception che rappresenta l’eccezione sollevata

```
<%@ page language="java"
    isThreadSafe="false" %>
<%@ page isErrorPage="true" %>
<%@ page info = "Form per
    inserire il nome di un file" %>
<html><head>
<title>Error Page</title>
</head>
<body>
 <h2>SI E’ VERIFICATO UN ERRORE </h2>
 <p><%= exception.getMessage() %>
</body></html>
```

9.14 Esempio JSP: non thread safe

```
<%@ page language="java" isThreadSafe="false" %>
<%@ page info = "Contatore di accessi" %>
<%! int count=0;
    private static int FREQUENZA = 10;
    public void more() {count++;}%>
<html>
<head><title>Counter</title></head><body>
 <% more();
  /* Se il numero \‘e divisibile
   * per FREQUENZA, visualizziamo
   * un messaggio */
  if ((count % FREQUENZA)==0) { %>
  <%@ include file = "haivinto.jsp"%>
  <% } else { %>
  <%@ include file = "nonhaivinto.jsp"%>
  <% } %>
</body></html>
```

haivinto.jsp:

```
<h1> BRAVO HAI VINTO! </h1>
  visitatore numero: <%= count %>
```

nonhaivinto.jsp:

```
<H1> NON HAI VINTO!</h1>
  visitatore numero: <%= count %>
```