

# Introduzione al linguaggio Java

Corso di Gestione della Conoscenza d'Impresa  
Prof. Giovanni Semeraro  
Dott. Pasquale Lops

Dipartimento di Informatica  
Università degli Studi di Bari

A.A. 2006/2007

# Credits

Slide realizzate e modificate da:

- Oriana Licchelli
- Ignazio Palmisano
- Domenico Redavid
- Pasquale Lops
- Pierpaolo Basile



- 1 Object Oriented
- 2 Java in dettaglio
- 3 Cicli
- 4 Input/output e collezioni

# Overview

## Parte uno

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

## Parte due

- Il linguaggio Java
  - Portabilità
  - Semplicità

## 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)



# I Due Paradigmi della Programmazione

## 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

## 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)

### 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

## Oggetti: cosa sono mai?

### Oggetti: esempio

- 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 awiati() { /* implementazione */ }  
    public static void main(String[] args) {  
        Automobile a = new Automobile();  
        a.awiati();  
    }  
}
```

# Fondamenti dell'Object Oriented (OO)

- Incapsulamento
- Ereditarietà
- Polimorfismo

## 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

# Interazioni Tra Oggetti (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)

## Interazioni Tra Oggetti (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

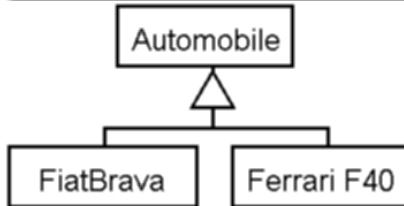
# Ereditarietà

## Estensione di una classe

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

## 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



# Polimorfismo (a tempo di compilazione)

## 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()`



## Polimorfismo (a tempo di esecuzione)

### Esempio

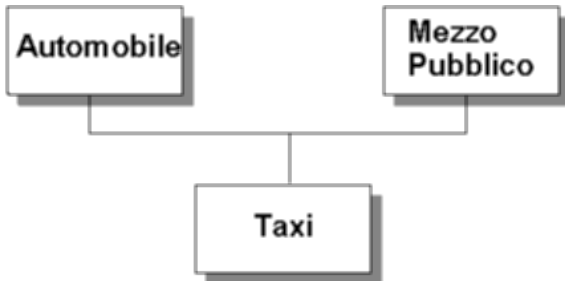
- Un oggetto di tipo `FerrariF40` è un oggetto di tipo `Automobile`
- Posso gestire questo oggetto attraverso una variabile di tipo `Automobile`
- Il viceversa non è sempre vero
- Upcast: da `FerrariF40` ad `Automobile`
- Downcast o cast: da `Automobile` a `FerrariF40`

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

## Ereditarietà multipla

Si può estendere più di una classe?

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



## Alcune caratteristiche di Java

- Orientato agli oggetti
- Architetaturalmente 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 ...

## Alcune caratteristiche

- 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

# Compilazione ed esecuzione (1)

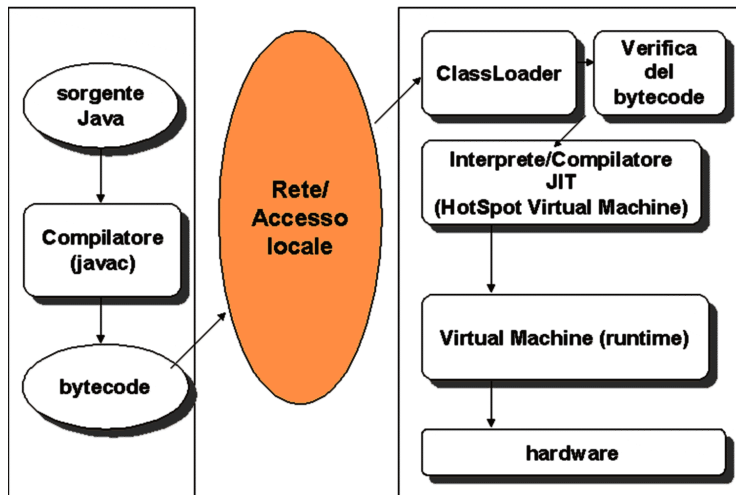
- Il compilatore Java (javac) produce bytecode per una macchina astratta (la Java Virtual Machine)...
- `javac -classpath <classpath> Programma.java → Programma.class`
- ... che viene eseguito da un programma nativo (l'implementazione della Virtual Machine per il sistema)
- La Virtual Machine è libera 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`

## Compilazione ed esecuzione (2)

### 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)

## compile-load-run



# 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

## 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();
```

# 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

## 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) {
        System.out.println("Ahi ahi ahi, quanto mi costi..."); // stampa sulla console
        for (int i = 0; i < quanto; i++) { // for classico...
            System.out.println("Glu glu glu");
            /*commento su
             * più linee
             */
        }
    }
}
```

## Reminders

### Tipi primitivi

Tipi interi	byte	8 bit	short	16 bit
	int	32 bit	long	64 bit
Tipo carattere (Unicode)	char	16 bit		
Tipo booleano	boolean	1 bit		
Tipi floating-point (IEEE)	float	32 bit	double	64 bit

### Sequenze di escape

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

# Operatori e Modificatori di flusso

## Operatori

Relazionali >, >=, <, <=, !=, ==

Aritmetici +, -, \*, /, %

## 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

# Cicli

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

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

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

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

```

## 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 automaticamente un "costruttore di default" ...
- ... che ovviamente non fa nulla ...

`package` slides;

```
public class FerrariF40 extends Automobile {  
    public FerrariF40() {}  
}
```

## 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 newOwner, int newPrice) {  
        this(newOwner);  
        this.price = newPrice;  
    }  
}
```

# 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; }
}
```

# 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

## static e final

### 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()`

### 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

## 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;  
        }  
    }  
}
```

# 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`

## Collisioni

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

# Classpath

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

File DOS	<code>java\util\Vector.class</code>
File UNIX	<code>java/util/Vector.class</code>

- Per indicare al class loader i percorsi da usare si specifica il `CLASSPATH`
- `CLASSPATH=c:/java; c:/java/lib` allora `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`

<code>java.lang.String</code>	<code>String.class</code> in <code>classes.zip</code> nella sottodirectory <code>java/lang</code>
-------------------------------	--

## Radice della gerarchia 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`

### 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

# 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

## 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()`

# Interfacce (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

## Interfacce (2)

- Un'interfaccia è una classe contenente solo dichiarazioni di metodi e costanti
- Le implementazioni sono demandate 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() {...}
}
```

## 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...");}  
    }  
}
```

## Esempio di gestione di eccezioni

```
package slides;
import java.io.FileInputStream;
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

## Raffinamento. . .

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

```
package slides;
import java.io.FileInputStream;
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");
        }
    }
}
```

## ... 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`

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

## ... Raffinamento...

- Una catch può catturare l'eccezione, esaminarla e risolverla
- 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);}
```

# 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( );}  
}
```

## 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>

### 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

## Trucchi per la codifica

---

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

# 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`

## 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

# 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)`

# 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, Hashtable, 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)

## 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 gli errori a runtime (ClassCastException individuate a compile time)
- È possibile inserire tipi primitivi (boxing - unboxing automatico)
- In realtà, il bytecode è lo stesso, ma cast e boxing sono gestiti dal compilatore → meno bug

## 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);  
    }  
}
```