Corso di Laurea in
Informatica e Comunicazione Digitale

a.a. 2006 - 2007

Interazione Uomo-Macchina

Introduzione a Java

Seconda Parte

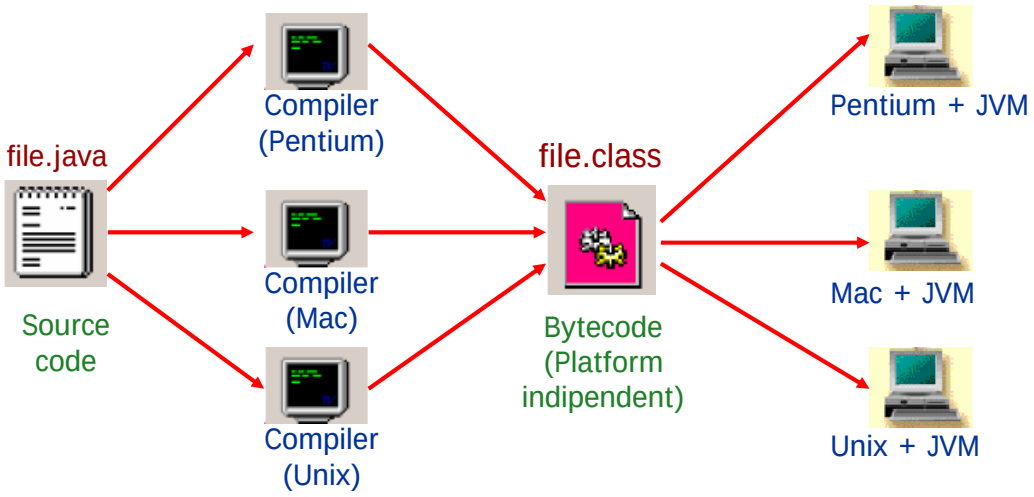
Irene Mazzotta
Giovanni Cozzolongo
Vincenzo Silvetri

Richiami: Java

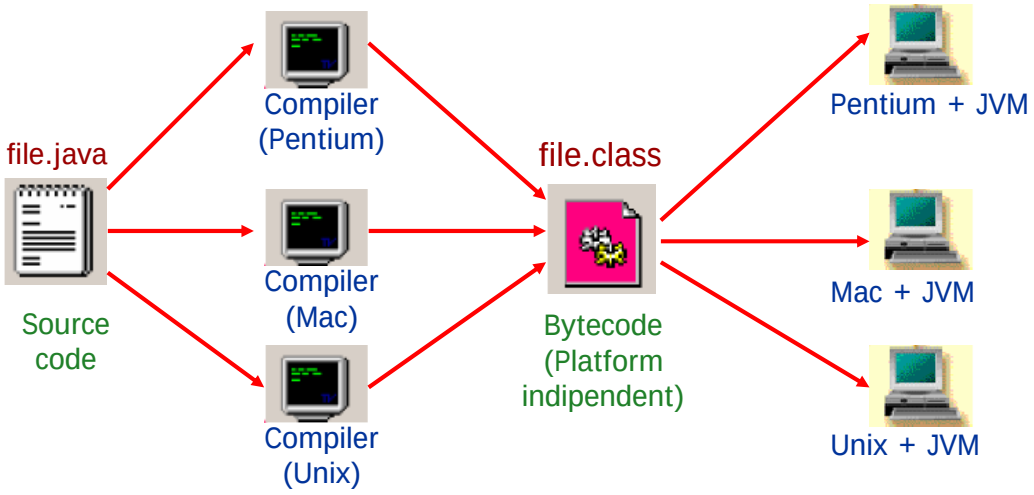
- Il linguaggio **Java** è un linguaggio di programmazione orientato agli oggetti, derivato dal C++ (e quindi indirettamente dal C).
- La piattaforma di programmazione Java è fondata sul Linguaggio stesso, sulla Java Virtual Machine (JVM) e sulleAPI.

Caratteristica principale: **Portabilità**.

WRITE ONCE, RUN EVERYWHERE!



WRITE ONCE, RUN EVERYWHERE!



Esecutori di Bytecode

Java può essere usato per creare due tipi di programmi:

- *stand-alone program* (applicazione)
  - eseguiti da interpreti java;
- *applet* eseguiti da
  - browser Web
  - applicativi ad hoc.

Altre caratteristiche di Java

- Semplice
- Sicuro
- Robusto
- Indipendente dall'architettura
- Distribuito
- Interpretato
- Dinamico
- Ad elevate prestazioni
- Concorrente (multithread)

Semplicità

- Sintassi simile a C e C++ ma privo di costrutti ridondanti/"pericolosi“.
- Ci sono solo tre gruppi di primitive (numerico, boolean e array) e delle classi specifiche per le stringhe.

Sicurezza

- Java prende le decisioni relative alle allocazioni di memoria a run-time (*Binding dinamico*).
- L’inesistenza di puntatori impedisce eventuali errori, anche involontari, ad altri programmi in memoria.

Robustezza

- Fortemente tipizzato
  - controllo statico dei tipi
  - il compilatore Java effettua severi controlli in fase di compilazione.
- Gestione **automatica** della memoria mediante il meccanismo del *garbage collection*;
- Gestione delle eccezioni (tutti gli errori a run-time dovrebbero essere gestiti dal programma).

Java vs C++

Sintassi simile al C++ ma…Java

- Non usa puntatori.
- Non supporta l'ereditarietà multipla (ma la ottiene mediante le interfacce).
- Ha il tipo stringa (le stringhe sono oggetti veri e propri).
- Ha una gestione migliore della memoria (garbage collection).

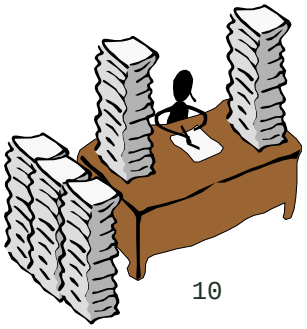
"Conclusione: Java è un 'C++' --"

Perché programmare a *oggetti*?

- Programmare a *oggetti*…
  - non velocizza l'esecuzione dei programmi…
  - non ottimizza l'uso della memoria...

E allora perché programmare a *oggetti*?

- Programmare a *oggetti* facilita la progettazione, lo sviluppo e il mantenimento di sistemi software molto complessi!



Convenzioni

I nomi delle Classi iniziano con la MAIUSCOLA

I NOMI DEGLI Oggetti iniziano con la minuscola

Point point = new Point();

Classe e Oggetti

- Classe: descrizione astratta di un oggetto
- Oggetto: istanza di una classe: un oggetto è una coppia (stato, funzioni)

ES.: Classe Automobile
Oggetto Berlina
Oggetto Monovolume

ATTENZIONE:

Una *Istanza* non può esistere senza la sua *Classe* di appartenenza!!!

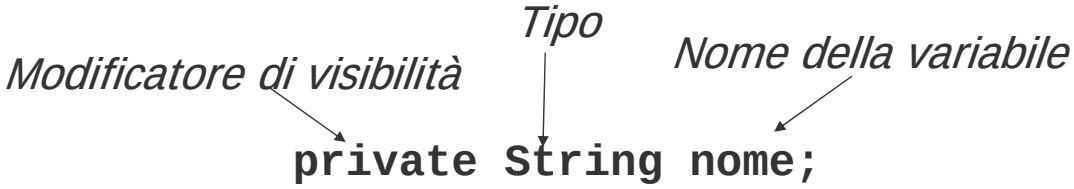
Classe e Oggetti

- Nella classe definiamo:
  - Le *variabili*: consentono di memorizzare le informazioni di ciascuna istanza (cioè lo stato dell'oggetto).
  - I *costruttori*: specificano come inizializzare lo stato di una nuova istanza.
  - I *metodi*: specificano le operazioni che determinano il comportamento delle istanze.

```
public class ClassName {
    Variabili di istanza
    Costruttori
    Metodi
}
```


Variabili di istanza

- Nella definizione di una variabile di istanza:
  - Modificatore di visibilità
  - Tipo
  - Nome della variabile



Costruttori

- I costruttori inizializzano lo stato di un oggetto
  - Memorizzano i valori iniziali negli attributi;
  - A tal fine spesso ricevono parametri dall'esterno.
- I costruttori hanno lo stesso nome della loro classe.
- Se non si specificano costruttori, viene definito automaticamente un "costruttore di default".

```
public class Persona {
    private String nome;
    private int civico;

    public Persona(String nome, int civico) {
      this.nome = nome;
      this.civico = civico;
    }
}
```


Costruttori: La parola chiave this

- **this** rappresenta un riferimento all'istanza che sto manipolando
- E' indispensabile farne uso quando il nome del campo a cui si vuole accedere è nascosto dal nome di una variabile locale o di un parametro

```
public class Persona {
    private String nome;
    private int civico;

    public Persona(String nome, int civico) {
       this.civico = civico;
      this.nome = nome;
    }
    …
}
```

Costruttori: La parola chiave this

- Per favorire la leggibilità del codice adottiamo comunque la convenzione di usare **sempre** **`this`** quando ci riferiamo a variabili di istanza

```
public class Persona {
   private String nome;
   private int civico;

   public Persona(String nome, int civico) {
      this.civico = civico;
     this.nome = nome;
   }
}
```


Costruttori: L'operatore new

- Crea un nuovo oggetto della classe specificata.
- Invoca l'esecuzione del **costruttore** (che inizializza lo stato dell'oggetto)
- Ritorna un riferimento all'oggetto creato

```
Persona persona = new Persona()
```

Costruttori: I parametri

- Costruttori e metodi ricevono dati attraverso parametri
- I parametri sono definiti nell'intestazione del costruttore o del metodo

```
public class Persona {
   private String nome;
   private int civico;

   public Persona(String nome, int civico) {
     this.nome = nome;
     this.civico = civico;
   }
}
```

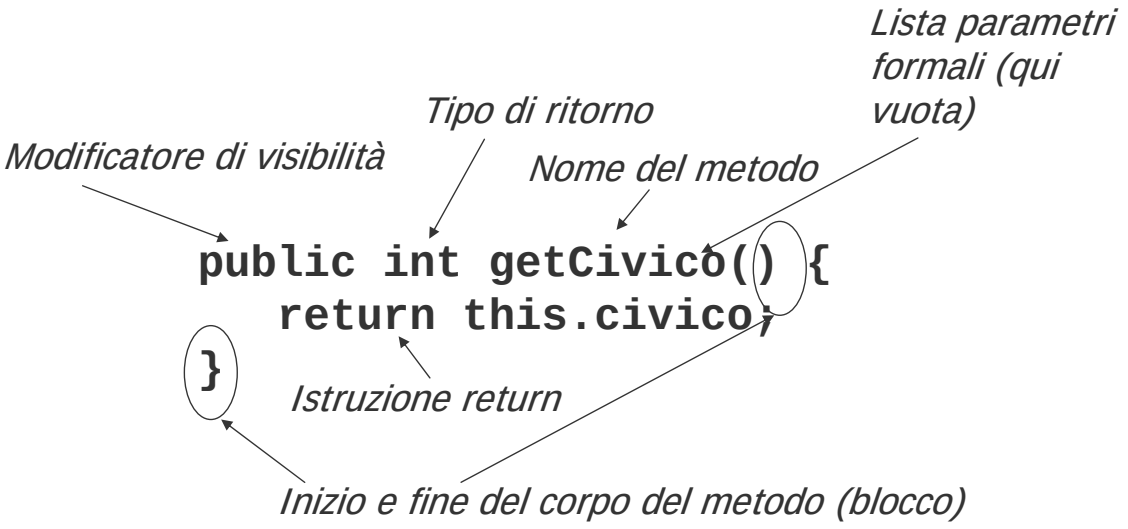

Costruttori: I parametri

```
public class Persona {
   private String nome;
   private int civico;

   public Persona(String nome, int civico) {
     this.nome = nome;
     this.civico = civico;
   }
}
```

- Questo costruttore ha due parametri:
  - **nome**, che è di tipo **String**
  - **civico**, che è di tipo **int**

Metodi

- Implementano il “comportamento” di un oggetto
- Hanno una *intestazione* ed un *corpo*



Metodi

```
public class Persona {
  …

  /**
   * Imposta il nome di una persona
   * @param il nome di una persona
   */
   public void setNome(String nome) {
      this.nome = nome;
   }
}
```

Metodi: Modificatori d'accesso

L'information hiding è ottenuto mediante I *modificatori di accesso* che limitano la visibilità delle variabili e dei metodi agli oggetti delle classi esterne e alle sottoclassi.

I modificatori sono:

public	visibile da tutti
friendly	visibile da tutti nello stesso package (sarebbe il default)
protected	visibile dalle sottoclassi
private	nascosto a tutti, tranne che all'interno della classe

Metodi: Modificatori d'accesso semplici linee guida

- Tutte le variabili di istanza (e di classe) devono essere private
  `private TipoAttributo nomeAttributo;`
- Se ci sono forti motivazioni per concedere la possibilità di aggiornare dall'esterno un attributo si definisce un metodo "modificatore" pubblico (o amico)
  - Convenzione nome metodo modificatore:
    `void setNomeAttributo(TipoAttributo nomeAttributo)`
- Se ci sono esigenze per accedere in lettura si definisce un metodo "accessore" pubblico (o friendly)
  - Convenzione nome metodo accessore:
    `TipoAttributo getNomeAttributo()`

Metodi: valore restituito

- Un metodo può restituire un valore
  - **`void`** se non restituisce nessun valore (come nel caso del metodo **`setNome(String)`**)

```
/**
 * Ritorna il nome della persona
 * @return String il nome della persona
 */
public String getNome() {
     return this.nome;
}
```


Chiamate a metodi esterni

- Forma generale: *dot-notation*

```
oggetto.nomeMetodo(lista-parametri-attuali)
```

- Diciamo che effettuiamo una *chiamata a un metodo esterno* quando chiamiamo un metodo di un altro oggetto

Chiamate a metodi esterni

- Forma generale

  `nomeMetodo(lista-parametri-attuali)`

- E' equivalente a:

  `this.nomeMetodo(lista-parametri-attuali)`

  per leggibilità preferiamo quest'ultima

Tipi Primitivi in Java

`boolean`	vero (**`true`**) o falso (**`false`**)
`char`	caratteri Unicode 2.1 (16-bit)
`byte`	interi a 8 bit (con segno e in C2)
`short`	interi a 16 bit (con segno in C2)
`int`	interi a 32 bit (con segno in C2)
`long`	interi a 64 bit (con segno in C2)
`float`	numeri in virgola mobile a 32-bit
`double`	numeri in virgola mobile a 64-bit

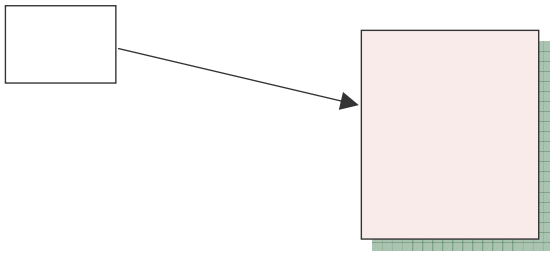
Tipi Primitivi e Oggetti

```
int i;
```

Tipo primitivo

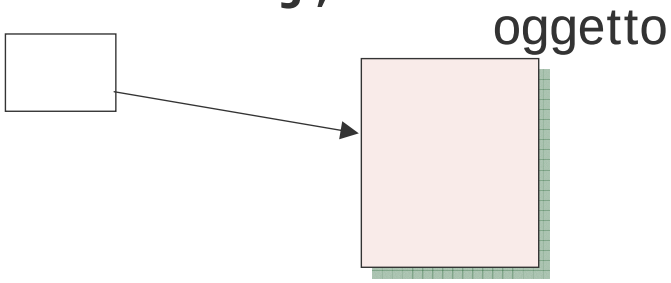
```
MiaClasse obj;
```

Oggetto



Tipi Primitivi e Oggetti

```
MiaClasse obj;
```



- La variabile **`obj`** non memorizza direttamente l'oggetto, ma un *riferimento* all'oggetto

Tipi Primitivi e Oggetti

```
int i;
```

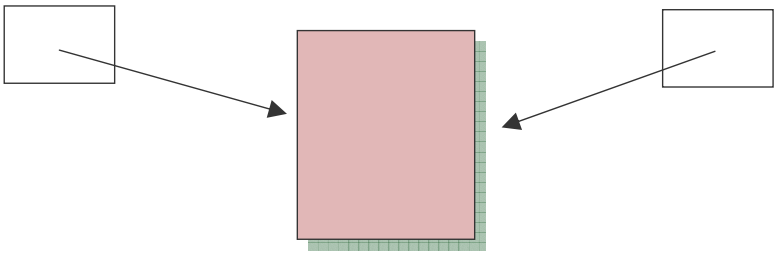
Tipo primitivo

- Nel caso dei tipi primitivi, il valore è memorizzato direttamente nella variabile

Tipi Primitivi e Oggetti

Oggetti

```
MiaClasse a;        MiaClasse b;
```



```
a = b;
```

Tipi primitivi

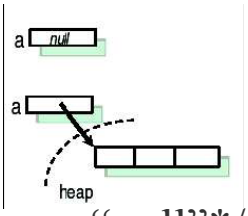
```
int a;              int b;
```

Array

- In Java gli array sono oggetti
- I componenti di un array:
  - sono tutti dello stesso tipo
  - possono essere di tipo primitivo o reference (inclusi altri array)
  - sono indicizzati con int (indice primo elemento: 0), con controllo di validità degli indici a runtime

Esempi:

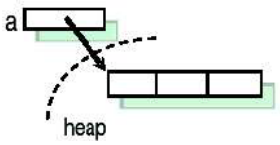
```
int[ ] a;                /* dichiarazione */
/* anche int a[ ]; */
a = new int[3];          /* creazione e inizializzazione a "null"*/
a[0] = 0;                /* accesso all'array */
```



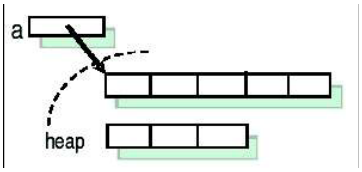
Array

- La lunghezza di un array è fissata al momento della sua creazione, e non può essere cambiata
- ... ma si può assegnare un nuovo array di diversa lunghezza all array reference:

```
int[ ] a = new int[3];
```

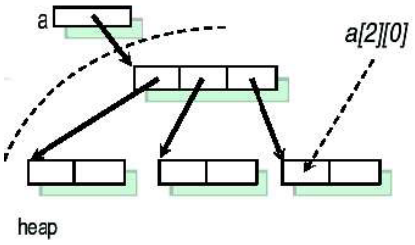


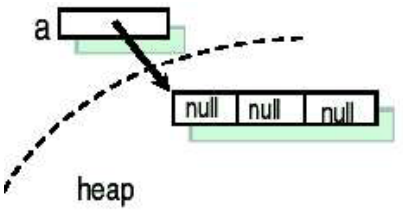

```
a = new int[5];
```



Array di array

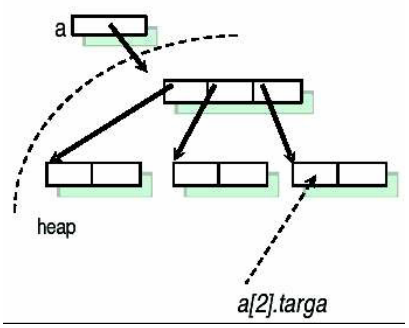
- short [ ] [ ] a; /* array di array di short */
- short a[ ] [ ]; /* equivalente */
- a = new short [3][2];
- a = new short [3][ ];





Array di oggetti

```
class Automobile {
public int targa;
public int velocità;
}
Automobile[ ] a = new Automobile[3];
```



Variabili in Java

In Java esistono tre tipi di variabili:

- *variabili di istanza*: definiscono gli attributi e lo stato di un oggetto; persistono per tutto il tempo di vita dell'oggetto
- *variabili di classe*: definiscono gli attributi e lo stato di una classe (si dichiarano ***static)***
- *variabili locali*: utilizzate all'interno dei metodi per memorizzare informazioni utili alla esecuzione del metodo stesso; sono create nel momento in cui viene creato il metodo e sono distrutte quando il metodo termina.

ATTENZIONE: variabili di istanza ≠ variabili locali!!!

static e final

static

- Un campo static è unico per la classe: tutte le istanze della classe fanno riferimento allo stesso campo (variabile di classe: condivisa da tutti gli oggetti della stessa classe).

```
private static int perTutti;
```

- Un metodo static può essere invocato senza aver istanziato un oggetto della classe.

final

- Un campo final è una costante non modificabile

```
final double COSTANTE_GRAVITA = 9.8;
```

- Un metodo final non è ridefinibile
- Una classe final non è estendibile

Ancora su final

```
final int a = 10;
a = 3; // ERRORE DI COMPILAZIONE
```

```
final int a = 10;
int b = 4;
a = b; // ERRORE DI COMPILAZIONE
```

ATTENZIONE: quando **`final`** si usa su

- primitivi: rende costante la variabile
- riferimenti: rende costante il riferimento, ma non il contenuto dell'oggetto referenziato

Costanti e variabili di classe

- È ragionevole che una costante sia anche un variabile di classe. Perché?
- Per questo le dichiarazioni delle costanti di solito sono espresse come segue:

```
final static double COSTANTE_GRAVITA = 9.8;
```

Un particolare metodo *static*: il metodo main()

- E' il punto d'accesso di un'applicazione Java.
- Almeno una classe di un applicazione Java deve contenere il metodo main().

```
public static void main(String args[])
```

public: modificatore d'accesso. Il metodo è visibile a tutti!

static: parola riservata. Il metodo è invocato senza dover creare un'istanza della classe che contiene il metodo main().

void: il metodo non produce risultato

String arg[]: una sequenza da 0 a inf di parametri separati da uno spazio. **Qual è il significato di questo vettore?**

Metodo main(): String args[]

Usato per passare argomenti ad un'applicazione java.

E' sufficiente far seguire, al nome dell' applicazione, i valori desiderati separti da uno spazio: tale sequenza viene interpretata e memorizzata come un vettore di stringhe, args[] che è definito in main.

Metodo main(): String args[]

```
//:Args.java
// Determina se passata una stinga
// sulla riga di comando

public class Args {
  public static void main(String args[]) {
    if (args.length < 1) {
       System.out.println("Nessun argomento");
    }
    else {
      System.out.println("Stringa digitata: " +
                                        args[0]);
    }
  }
}
```

Input: java Args ciao_a_tutti

Output: Stringa digitata: ciao_a_tutti

La sintassi per l'avvio nel caso specifico è: java Args.class ciao_a_tuttii

Operatori e Strutture di controllo

Operatori

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

Strutture di controllo

- *selezione*: if - else, switch
- *iterazione*: while, do - while, for
- *interruzione di ciclo*: break, continue, return
- *gestione eccezioni*: try - catch - finally

ATTENZIONE: Le condizioni per if, for, while sono espressioni booleane; non possono essere espressioni intere come in C

Gestione Eccezioni

- Non tutti gli errori presenti in un programma possono essere rilevati a tempo di compilazione.
- Quando si verifica una condizione anomala in una sequenza di codice al momento dell'esecuzione si dice che si è verificata una ***eccezione*.**
- Una condizione eccezionale è un problema che impedisce la regolare esecuzione del programma.
- Per gestire una eccezione si potrebbe uscire dal contesto corrente relegando il tutto ad un contesto più alto.
- L'uscita dal contesto corrente corrisponde al ***sollevamento di una eccezione*** *(Throwing exception)*

Lanciare un'eccezione

- Se immaginiamo che qualcosa possa andare storto, dobbiamo gestire l'eccezione
- Per lanciare un'eccezione:
  - Viene costruito l'oggetto "eccezione":

    ```
    new ExceptionType("... ");
    ```

  - L'oggetto eccezione viene *lanciato* con l'istruzione:

    ```
    throw eccezione
    ```

Sollevare eccezioni: l'istruzione throw

- Lancia una eccezione in modo esplicito.
- Sintassi: throw ThrowableInstance;

  Dove ThrowableInstance è una istanza della classe Throwable o di una sua sottoclasse.

```
If (t == null) throw new
                        NullPointerException()
```

Dove NullPointerException() è una classe predefinita di eccezioni.

Gestione eccezioni

Immaginiamo che la classe A invochi al suo interno il metodo della classe B. Chiamiamo A ClasseClient e B ClasseServer.

```
class ClasseClient {
    ClasseServer server;
    public void metodoClient() {
      int i;
     i = server.metodoServer();
    }
}

class ClasseServer {
    public int metodoServer() throws NullPointerException(){
      …
    if (qualcosa è andato storto)
      throw new NullPointerException();
    }
}
```

Se qualcosa va storto questo metodo lancia una eccezione di puntatore a *"null"*

L'effetto di una eccezione

- Il metodo che lancia una eccezione finisce prematuramente
- Non viene ritornato nessun valore
- Il controllo non ritorna al punto di chiamata del client, ma il client può catturare e gestire l'eccezione

Catturare eccezioni: l'istruzione *try - catch*

- Le chiamate a metodi che potrebbero incorrere in situazioni di eccezione devono essere incapsulate in un blocco *try - catch.*
- Le chiamate al metodo che lancia una eccezione devono essere effettuate all'interno di un blocco *try* {}.
- L'eventuale eccezione viene catturata e gestita nel blocco *catch*{}.

Gestione eccezioni

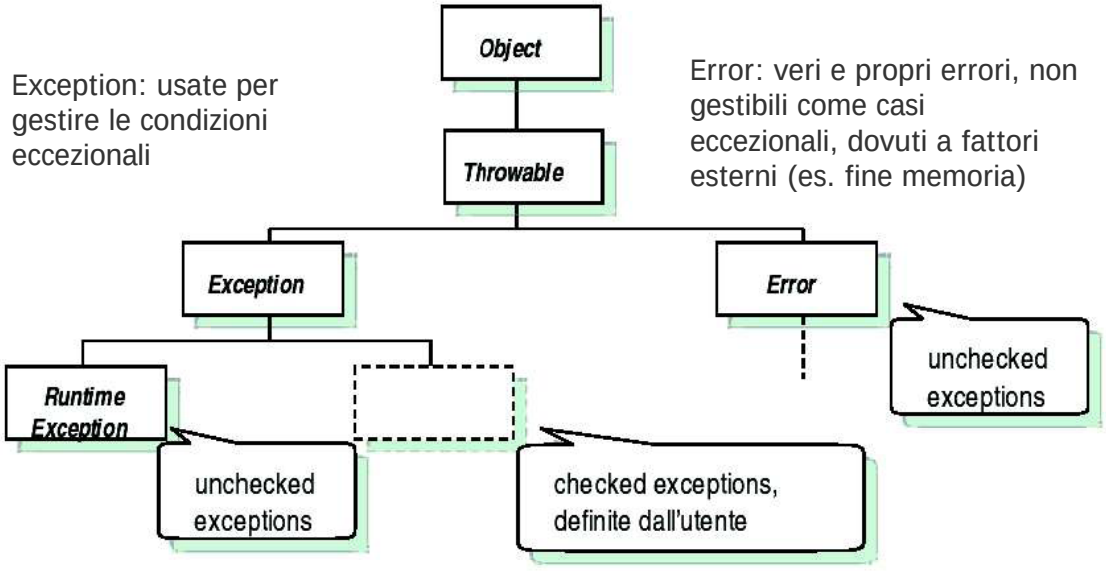
```
class ClasseClient {
    ClasseServer server;
    public void metodoClient() {
      int i;
      try {
              i = server.metodoServer();
      }
      catch (NullPointerException e) {
       … codice gestione situazione anomala
      }
      …
    }
}

class ClasseServer {
    public int metodoServer() throws NullPointerException(){
      …
    if (qualcosa è andato storto)
      throw new NullPointerException();
    }
}
```

Provo a chiamare un metodo che dichiara di lanciare una eccezione se qualcosa va storto

Se il metodo ha sollevato una eccezione **NullPointerException** gestisco la situazione come segue

Eccezioni standard in Java

Exception: usate per gestire le condizioni eccezionali



Error: veri e propri errori, non gestibili come casi eccezionali, dovuti a fattori esterni (es. fine memoria)

Le eccezioni derivate da run-time error non devono essere inserite nell'elenco throws del metodo!!!

Gestione della eccezione

- Nel caso di **checked exception** il client è **obbligato** a gestire l'eccezione
  - Ovvero il programmatore deve scrivere codice che specifichi come comportarsi in caso di eccezione
  - Il compilatore verifica che il programmatore abbia scritto il codice di gestione della eccezione
- Nel caso di una **unchecked exception**
  - Il compilatore non fa nessuna verifica
  - Se non vengono gestite causano la terminazione del programma. Un esempio che già conosciamo: **`NullPointerException`**

Gestione dell'eccezione

- Il client che chiama un metodo che dichiara di lanciare una eccezione deve "gestire" l'eccezione
- Per la gestione di una eccezione il client ha due possibilità
  - non se ne preoccupa direttamente, ma delega a sua volta la gestione dell'eccezione al metodo chiamante
  - cattura e gestisce direttamente l'eccezione

Catturare eccezioni multiple

- Un metodo di un oggetto server potrebbe lanciare diversi tipi di eccezione (corrispondenti a diversi tipi di anomalie)
- Il client può gestire diversamente queste situazioni

Catturare eccezioni multiple

```
try {
    ...
    ref.process();
    ...
}
catch(ExType1 e) {
    // Take action on a pb1 exception.
    ...
}
catch(ExType2  e) {
    // Take action on a pb2 exception.
    ...
}
catch(ExType3  e) {
    // Take action on a pb3 exception.
    ...
}
```

Attenzione all'ereditarietà

```
try {
    ...
    ref.process();
    ...
}
catch(Exception e) {
    // Take action on an end-of-file exception.
    ...
}
catch(ExType2  e) {
    // Take action on a file-not-found exception.
    ...
}
catch(ExType2  e) {
    // Take action on a file-not-found exception.
    ...
}
```

qualsiasi eccezione verrebbe catturata dal primo **`catch`** **!**

La clausola `finally`

- Il meccanismo di gestione delle eccezioni è completato dal blocco **`finally`**
- Il codice nel blocco **`finally`** viene eseguito sempre, anche se l'eccezione non è stato rilevata
  - anche se nel blocco try o catch c'è una istruzione **`return`** !
- Tipicamente serve a chiudere file o connessioni (ad esempio ad un DBMS)

Sintassi try-catch-finally

```
try {
    …
}
catch(Exception e) {
    …
}
finally {
    azioni che verranno eseguite in ogni caso
}
```


Definire nuove eccezioni

- Ogni eccezione estende **`Exception`** o **`RuntimeException`**
- Definire nuovi tipi di eccezioni permette di fornire al chiamante informazioni diagnostiche
  - Include informazioni utili per decidere come gestire l'eccezione

Definire nuove eccezioni

```
public class MiaExeption extends Exception {

      public MiaException(String message) {
            super(message);
      }

}
```


Ereditarietà semplice

- **Obiettivo**: Permette di
  - derivare nuove classi a partire da classi già definite
  - aggiungere membri ad una classe, e modificare il comportamento dei metodi.
- L'eredità in java è realizzata mediante il comando **`extends`** applicato dopo la dichiarazione di una classe

  `public class Macchina extends MezzoDiTrasporto`
- Una classe estende sempre una e una sola altra classe, ma può a sua volta essere estesa da un numero arbitrario di classi.
- Se non specificato, la classe estende di norma la classe ***Object*** (fortemente consigliato l'uso della documentazione API)

Ereditarietà Semplice

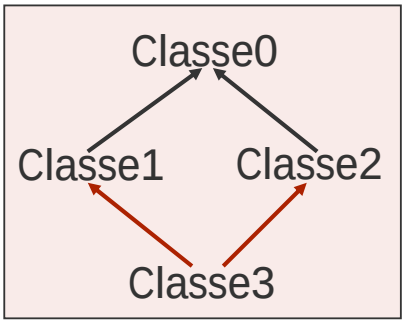
- Questo sistema rende disponibile un modo per creare gerarchie di classi ben definite

  Es.: vedi gerarchia classi vista nela lezione precedente (Berlina estende Macchina che, a sua volta, estende MezzoDiTrasporto…).

- L'*overriding* si attua semplicemente scrivendo all'interno della classe che estende (la sottoclasse) il metodo che vogliamo riscrivere, utilizzando la *stessa firma* del relativo metodo della superclasse (cioé, stesso nome e stessi attributi, in numero e tipo).

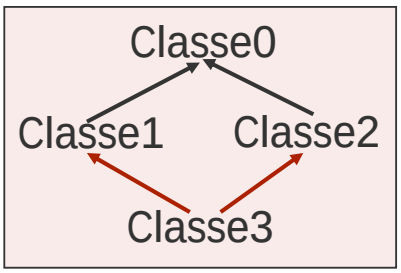
- È bene puntualizzare, quindi, che la tecnica dell'ereditarietà serve per **specificare** e **specializzare** un determinato comportamento, non viceversa.

Ereditarietà Multipla: problema




```
class Classe0 {
   void talk() {
        System.out.println("…");
}

class Classe1 extends Classe0 {
    void talk() {
            System.out.println(“ciao, ciao.");
   }
}

class Classe2 extends Classe0 {
       void talk() {
           System.out.println("I'm OK! ");
       }
}
```

Ereditarietà Multipla: problema




```
// (non compila.)
  class Classe3 extends Classe1, Classe2 {}
```

Cosa dovrebbe fare la seguente invocazione del metodo **`talk()`**?

```
Class0 class0 = new Class3();
class0.talk();
```


Ereditarietà Multipla: problema

- Se un membro (metodo o campo) è definito in entrambe le classi base, da quale delle due la classe estesa "eredita"?
- E se le due classi base a loro volta estendono una superclasse comune?
- Il problema è legato alle implementazioni (che vengono ereditate).

Per questo motivo:

- una classe può implementare tante interfacce
- ma può estendere una sola classe

Ereditarietà Multipla: interface

- Java supporta l'ereditarietà semplice (**extends**).
- Java realizza l'ereditarietà multipla grazie al meccanismo delle *interfacce* (**implements**).
- L'oggetto *interface* in java è una semplice dichiarazione di metodi che tutte le classi che implementano l'interfaccia devono realizzare al loro interno.

```
public class Macchina implements MazzoDiTrasporto
```

dove l'interfaccia MezzoDiTRasporto, è strutturata in tal modo:

```
public interface MezzoDiTrasporto{
   public int velocità();
   public int costoTratta();
   public String destinazione();
   public int CaricoUtile();
}
```


Ereditarietà Multipla: interface

- Con l'implementazione dell'interfaccia la classe *Macchina* deve contenere al suo interno i metodi *velocità(), costoTratta()*, … propri.
- L'utilizzo delle interfacce viene usato quando si hanno delle gerarchie di oggetti, o oggetti semplici che possiedono delle operazioni comuni (metodi), ma l'implementazione di queste sono differenti una dall'altra.
- Possono esistere gerarchie di interfacce!!!

Ereditairetà e riusabilita: *abstract*

- Abbiamo visto come l'ereditarietà può essere uno strumento utile per il riuso del codice (ma ricordate che va usata con cautela).
- In alcuni casi può essere utile definire classi base che sono pensate solo per essere estese
- Queste classi contengono una implementazione parziale
  - Si definiscono variabili di istanza
  - Si definisce l'implementazione solo di alcuni metodi; di altri si definisce solo la segnatura
- I metodi incompleti di implementazione vengono definiti completamente nelle classi estese

Ereditairetà e riusabilita: *abstract*

- Una *classe astratta* serve a questo scopo
  - Una classe astratta contiene una definizione parziale della implementazione.
  - Una classe astratta non può essere istanziata, ma possono essere istanziate le classi che la estendono.
  - Ovviamente vale la relazione sottotipo-supertipo, quindi le istanze delle classi che estendono una classe astratta possono essere usate in ogni caso in cui abbiamo un riferimento alla classe base.

Ereditairetà e riusabilità: *abstract*

```
public abstract class Personaggio {
  private String nome;
  private String descrizione;

  public Personaggio(String nome, String descrizione){
    this.nome = nome;
    this.descrizione = descrizione;
  }

  public String getNome() {
    return this.nome;
  }

  public String getDescrizione() {
    return this.descrizione;
  }

  abstract public void agisci(Giocatore giocatore);
}
```


Flussi di I/O in Java

La gestione dell'input e dell'output avviene mediante la classe **System**, usando le sue sottoclassi "**in**" ed "**out**"

Es.: *System.out.println(" stampa un messaggio);*
System.in.currentTimeMillis(); //ora attuale

La classe System gestisce l'input e output in maniera standard. E' possibile gestire i flussi di I/O in modo diverso?

E qui ci aiutano le due classi "**InputStream**" e "**OutputStream**" facenti parte del package **System.io**, in combinazione con la *bufferizzazione* dei dati. Queste tecniche vanno gestite con il controllo delle eccezioni.

Un esempio chiarificatore

```
import java.io.*;
public class TestInput {
public static void main(String args[]) {
try {
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  System.out.println("Come ti chiami ?");
  System.out.println("Ciao "+in.readLine());
 } catch (IOException e) { }
 }
}
```

Importazione package per la gestione del io, creazione standard della classe e del main

Creazione di un **buffer di input** ovvero tutti i tasti premuti vengono registrati in una locazione di memoria mediante **la lettura delle informazioni dalla tastiera**

ci provo try: {
……
 }
se non riesco: catch (IOException) {
……
}

Ma anche i File sono Flussi

La classe predisposta alla gestione dei flussi *da e per* i File è:

java.io.File

il comportamento di questa classe deve essere gestito mediante le eccezioni come visto nell'esempio precedente

Attenzione File è uno strumento molto potente e deve essere usato bene, basti pensare che con File possiamo leggere non solo un singolo file ma anche una Intera directory

Inoltre è possibile impostare dei filtri sui nomi dei file da leggere

Facciamo un esempio ma a voi la scoperta di tutti i metodi

Un esempio chiarificatore sul File

```
import java.io.*;
 public class TestFile {
 public static void main(String args[]) {
        String[] lista;
        File path;
        try {
        path = new File(".");
        lista = path.list();
                for(int i=0 ; i<lista.length ; i++) {
                System.out.println(lista[i]);
                  }
          } catch(Exception e) { }
       }
}
class Filtro implements FilenameFilter {
     String select;
     Filtro(String select) {
     this.select=“.java;"
     }
public boolean accept(File dir, String name) {
     String f=new File(name).getName();
     return f.indexOf(select) !=-1;
    }
}
```

definizione di un oggetto di tipo file
sua istanza

Lettura del contenuto della direcotry corrente filtrato mediante *Filtro* e stampa del suo contenuto

implementazione della classe FilenameFilter predisposta al Filtraggio dei nomi

Scrittura del metodo accept definito in java “abstract” per dare un senso al filtro. Viene invocato dal metodo File.list()