Introduzione a Java

Flavio De Paoli

Concetti di OO

- **Astrazione**
  - **le classi sono astrazioni degli oggetti**
  - **le interfacce definiscono le proprietà**
- **Ereditarietà**
  - **una classe può essere la specializzazione di un'altra classe (es. poligono-rettangolo)**
- **Polimorfismo**
  - **gli oggetti possono assumere forme diverse (es. Un rettangolo è un poligono)**

Perché Java

- Il software tradizionale è inadeguato per i sistemi distribuiti multi piattaforma
  - gli eseguibili (in binario) dipendono dal Sistema Operativo e dall'hardware
- Java propone di usare interpreti di un linguaggio intermedio (Byte Code)
  - portare l'interprete = eseguire ogni programma su ogni piattaforma
- Programmi Java sono robusti, adattabili, scalabili, efficienti, sicuri ...

Caratteristiche del linguaggio

- Sintassi e costrutti elementari noti
  - deriva dal C e dal C++
- Semplice
  - mantiene le peculiarità utili del C++
  - rimuove quelle superflue o pericolose
- Portabile
  - non dipende dall'ambiente target

Caratteristiche del linguaggio

- **Object Oriented**
  - **ereditarietà**
  - **polimorfismo**
  - **binding dinamico**
  - **reference (alla Eiffel)**
  - **separazione tra interfaccia e implementazione**
- **Dinamico**
  - **il link avviene a tempo di esecuzione**
  - **è possibile realizzare sistemi aperti**

Caratteristiche del linguaggio

- **Robusto & affidabile**
  - **tipizzazione**
  - **mancanza di puntatori**
  - **gestione automatica della memoria**
  - **controlli run-time (tipo, bounds, ...)**
  - **gestione automatica delle eccezioni**

Caratteristiche del linguaggio

- **Programmazione concorrente**
  - **sia il linguaggio, sia la macchina virtuale definiscono i thread per realizzare più flussi di controllo**
  - **definizione classi e metodi** `synchronized` **per mutua esclusione,**
  - **sospensione e risvaglio dell'esecuzione dei thread**
    `wait()` `notify()` `notifyAll()`

Perché il bytecode

twiligth.dse.disco.unimib.it/depaoli/cs

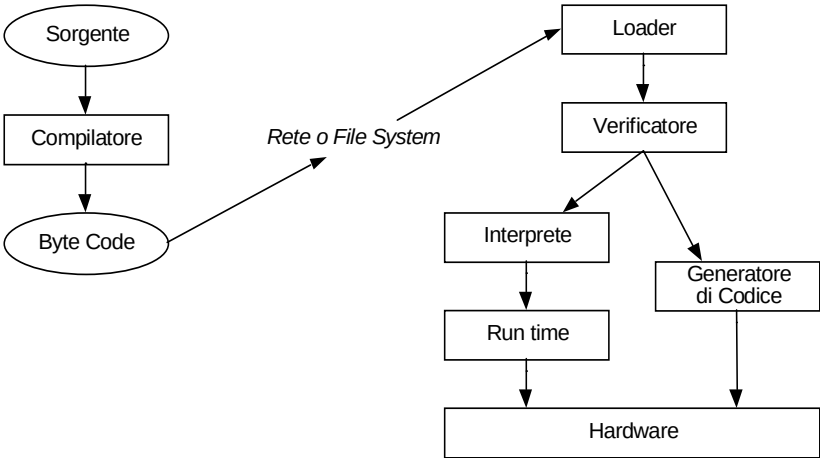
5	+	7	*	3	=	

5	7	3	*	+	=	

(5 + 7) * 3

5 7 + 3 *

L'ambiente Java

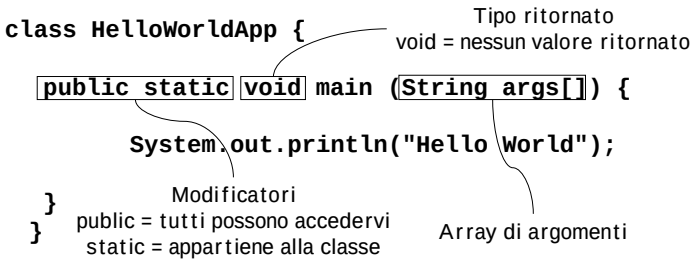


L'ambiente JSDK

- **Il programmatore ha a disposizione**
  - **il linguaggio**
  - **un API costituito da una ricca libreria**
- **Per la compilazione si usa**
  - **il compilatore *javac***
- **Per l'esecuzione**
  - **l'interprete *java* per i normali programmi**
  - **l'*appletviewer* per gli applet**
- **Per sviluppare**
  - **un debugger *jdb***

Un programma Java

- In Java esistono solo classi e oggetti (istanze di classi)
- Un programma è una classe che contiene un metodo denominato main



Dichiarazioni e tipi

- Java è un linguaggio *fortemente tipizzato*
  - ogni elemento deve essere dichiarato
  - tutti gli elementi hanno un tipo
- Esistono due tipi di dichiarazioni:
  - variabili di tipi primitivi
  - istanze di classi
- I tipi primitivi non sono oggetti
  - semplicità ed efficienza
  - (ma possono essere oggetti per compatibilità)

I tipi primitivi

- **Definizione precisa per i tipi primitivi**
  - `boolean` `true o false`
  - `char` `16 bit`
  - `byte` `8 bit con segno`
  - `short` `16 bit con segno`
  - `int` `32 bit con segno`
  - `float` `32 bit floating point`
  - `double` `64 bit floating point`
  - `long` `64 bit con segno`

Dichiarazioni

- **Una dichiarazione di variabile di un tipo primitivo funziona come in Pascal o C**
- **Una dichiarazione di istanza crea un *reference* ad un oggetto (non l'oggetto)**
- **Gli oggetti vengono creati dinamicamente**
- **Esempi**

```
int x; char ch; bool valido;

UnaClasse unRef;
unRef = new UnaClasse();

UnaClasse unRef = new UnaClasse();
```

Gestione della memoria

- **La gestione della memoria è automatica**
  - **creazione esplicita degli oggetti (**new**)**
  - **distruzione di un oggetto quando non esistono più riferimenti ad esso (*garbage collection*)**
- **Il meccanismo di *garbage collection* garantisce una esecuzione *sicura* dei programmi**
  - **Il programmatore non ha bisogno (né può) gestire direttamente la memoria**

Array

- **Sono oggetti a tutti gli effetti**
- **Dichiarati e creati dinamicamente**
- **Esempi (per tipi primitivi)**

```
int    numbers[] = new int[10];
int[]  numbers = new int[10];
int    matrix[] [] = new int[5] [10];
int    numbers[] = {0, 1, 2, 3, 4};
int    matrix[] [] = {{0, 1}, {0, 1, 2}};
```

Array

- La creazione definisce le dimensioni (che non possono essere cambiate)
- Ogni elemento va creato esplicitamente se si tratta di oggetti
- Esempio (per array di oggetti)

```
Point lista[];
lista = new Point[2];
lista[0] = new Point();
lista[1] = new Point();
```

Array

- L'attributo `length` consente di conoscere le dimensioni di un array a run-time
  - la lunghezza va da `0` a `(length-1)`
- A run-time vengono fatti controlli sul rispetto dei limiti
- Gli array possono essere estesi
  - viene creato un nuovo array con nuove dimensioni

Array dinamici

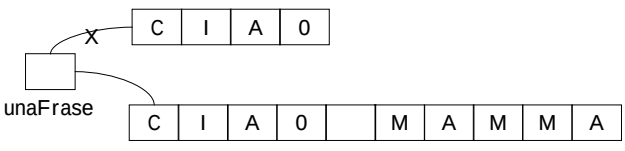
- **Consideriamo la classe**

```
class Esempio {
  static public void main(String args[])) {
    String unaFrase = "ciao";
    unaFrase = unaFrase + " mamma";
    System.out.println("La frase: " + unaFrase);
  }
```

  - **L'operatore + di concatenazione genera una nuova frase composta dai due operandi**
  - **Il run-time system elimina l'oggetto che conteneva la vecchia frase**

Array dinamici

```
class Esempio {
  static public void main(String args[])) {
    String unaFrase = "ciao";
    unaFrase = unaFrase + " mamma";
    System.out.println("La frase: " + unaFrase);
  }
```



La classe *String*

- **La classe** `String` **rappresenta sequenze di caratteri**
  - **un esempio di istanza:**
    `"una stringa di caratteri"`
  - **le stringhe sono costanti: non possono essere cambiate dopo la creazione**
- **Le operazioni permettono di**
  - **accedere a singoli caratteri e a sottostringhe**
  - **confrontare, copiare, cercare, …**

Controllo

- **Sintassi e semantica del C**
- **Istruzione condizionale**
  - **if-then-else**

```
if (espressione logica) {
        ...
} elsif (espressione logica) {
  ...
} else {
  ...
}
```

Controllo

- **Cicli a condizione iniziale e finale**
  - **while-do**

```
while (espressione logica) {
    ...
  }
```

  - **do-while**

```
do {
    ...
  } while (espressione logica)
```

Controllo

- **Ciclo a conteggio**
  - **for**

```
for (espr1; esprLogica; espr2) {
    ...
  }
```

Valutata una volta, prima dell'esecuzione del for

Valutata prima di ogni iterazione

Valutata dopo di ogni iterazione

  - **Uso tipico**

```
int numeri[] = {3,5,8,3,6,7};

for (int i=0; i<numeri.length; i++) {
    System.out.print(numeri[i] + " ");
  }

// output : 3 5 8 3 6 7
```

L'operatore ++

- L'operatore ++ postfisso incrementa di 1 il valore *dopo* averlo restituito

```
int i = 0;
int x = i++; // x vale 0 e i vale 1
```

- L'operatore ++ prefisso incrementa di 1 il valore *prima* di restituirlo

```
int i = 0;
int x = ++i; // x vale 1 e i vale 1
```

- Istruzione condizionale
  - switch

```
switch (espressione) {
   case costante1: ... break;
   ...
   default: ... break;
  }
```

Struttura di una applicazione

- Segue la struttura dei file systems
  - package
  - unità di compilazione
  - classi
  - interfacce

Struttura di una applicazione

- Package
  - costituito da una o più unità di compilazione
  - corrisponde alle directory
- Unità di compilazione
  - contiene definizioni di classi e interfacce
  - una sola può essere pubblica e costituisce l'interfaccia dell'unità di compilazione
  - corrisponde ai file

Struttura di una applicazione

- **Classe**
  - dichiara variabili e operazioni
  - contiene l'implementazione delle operazioni
  - può estendere una classe e implementare più interfacce
- **Interfaccia**
  - dichiara costanti e operazioni
  - una classe può implementare una o più interfacce per realizzare ereditarietà multipla
  - può estendere una classe e più interfacce

Information hiding

- **Le informazioni contenute in un modulo devono essere private, oppure esplicitamente dichiarate pubbliche**
  - ogni modulo deve essere conosciuto attraverso la propria interfaccia ufficiale
  - netta separazione tra funzione e implementazione
  - è fondamentale per la modificabilità:
    Se un modulo cambia solo nella parte privata, i moduli cliente non vengono influenzati

Cos'è una applicazione OO

- **Una applicazione OO**
  - **è una collezione strutturata di classi indipendenti**
  - **viene costruita assemblando classi esistenti con tecnica bottom-up.**
- **Il termine strutturato della definizione esprime l'esistenza di relazioni tra le classi**

Relazioni tra classi

- **Relazione *cliente (usa)***
  - **Una classe è cliente di un'altra quando ne usa i servizi e le risorse**
- **Relazione *discendente***
  - **Una classe è un discendente di una o più classi quando è stata progettata come estensione o specializzazione di queste classi**
  - **Si ottiene con i meccanismi di (multiple) inheritance.**

Le classi

- **Attraverso la definizione di classe è possibile descrivere una categoria di oggetti**

```
class Point { // una semplice classe
    public double x, y;
  }
```

- **Una classe definisce la struttura di un insieme di oggetti**
  - **In prima istanza una classe è qualcosa di analogo al record del Pascal o Ada**

Le classi

- **Una classe è caratterizzata delle operazioni *definite sulle sue variabili***

```
class Point {
  public double x, y;
  public void translate (double a, double b) {
    // sposta di a orizzontale e b verticale
    x = a + x;  y = b + y;
  }
  public double distance (Point other) {
    //distanza da other
    return Math.sqrt(Math.pow(x - other.x, 2)+
                     Math.pow(y - other.y, 2));
  }
}
```

Esempio

```
Point A = new Point();
Point B = new Point();
double dis;

A.x = 15;
A.y = 45;
A.translate(10, 5);

B.x = 35;
B.y = 15;
B.translate(10, 5);

dis = A.distance(B);
```

Costruttori e operazioni

- **I costruttori garantiscono la creazione di oggetti corretti**

Non sono più accessibili direttamente

```
class Point {
  private double x, y;
   Point(double a, double b) { // un costruttore
     x=a;
     y=b;
   }
   public void translate(double a,double b) {...}
   public double distance(Point other) {...}
}
```

- **Le operazioni effettuano modifiche che garantiscono la correttezza**

Esempio 2

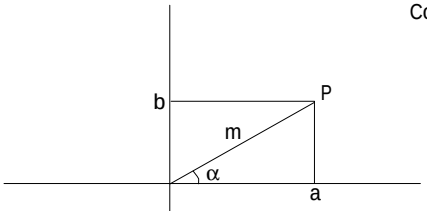
```
Point A = new Point(15, 45);
Point B = new Point(35, 15);
double dis;

A.translate(10, 5);

B.translate(10, 5);

dis = A.distance(B);
```


Coordinate cartesiane e coordinate polari



Coordinate cartesiane
P = (a,b)

Coordinate polari
P = (m, α)

Information hiding
L'uso di costruttori e operatori
"nasconde" il tipo di notazione utilizzata

Uso delle interfacce

```
interface DocumentoDiIdentita {
  public String dammiNome();
  public Date dammiDataNascita();
}
```

```
class CartaIdentita implements DocumentoDiIdentita {
  private String cognome;
  private String comune;
  private Date scadenza;
  CartaIdentita(...) { ... }
  public String dammiNome() {...}
  public Date dammiDataNascita() {...}
}
```

```
class PatenteGuida implements DocumentoDiIdentita {
  private String cognome;
  private String prefettura;
  private Date scadenza;
  private String categoria;
  PatenteGuida (...) { ... }
  public String dammiNome() {...}
  public Date dammiDataNascita() {...}
}
```

Classi e Interfacce

```
[public] class nome [extends nome_superclasse ]
    [implements interfaccia1, interfaccia2, ...]
  {
  [modificatore] dichiarazione di variabile
  ...
    [modificatore] dich. e impl. di operazione
  ...
  }
[public] interface nome
  [extends interfaccia1,interfaccia2, ...] {
    [modificatore] dichiarazione di variabile
  ...
    [modificatore] dich. di operazione
  ...
  }
```

Il controllo dell'accesso

- **Non tutti gli elementi di una classe devono essere accessibili liberamente**
- **Si usano dei modificatori per classi e metodi:**
  - *public*
  - *protected*
  - **nessun modificatore**
  - *private*

Il controllo dell'accesso

- *public* **chiunque può accedervi e (i metodi) vengono ereditati (è ammessa solo una classe public per file)**
- *protected* **accesso consentito alle classi dello stesso package (per i metodi) e a tutte le sottoclassi**
- vuoto **accesso ristretto al package**
- *private* **accesso ristretto alla classe o al file cui la classe è dichiarata**

Il controllo dell'accesso

- **I modificatori possono essere combinati**
  - *private protected* **accesso ristretto alle sottoclassi, qualunque sia il package di appartenenza (protected in C++)**

Altri modificatori

- **È possibile definire classi e metodi incompleti**
  - *abstract* **classi con metodi astratti e metodi senza implementazione**
  - **per usarle è necessario estenderle**
- **È possibile definire costanti**
  - *final* **rende variabili e metodi costanti**
  - **le variabili non sono modificabili**
  - **i metodi non sono ridefinibili**

Oggetti

- Gli oggetti devono essere
  - dichiarati con un tipo (classe)
  - creati con il costrutto new

```
class Point {
  public double x, y;
}


Point lowerLeft, upperRight;
lowerLeft = new Point();
upperRight = new Point();
lowerLeft.x = 0.0;
lowerLeft.y = 0.0;
```

Attributi e metodi di classe

- In Java è possibile definire metodi e attributi di classe utilizzando il modificatore `static`

```
public static Point origin = new Point();
```

- I metodi di `Math.pow` e `Math.sqrt` sono esempi di metodi statici (di classe)
- I metodi di classe non possono accedere ad attributi di istanza se non sono passati come parametro

Esempio

```
class Point {
  private double x, y;
  public static final double xOrigin = 0;
  public static final double yOrigin = 0;
  Point(double a, double b) { // un costruttore
    x=a;
    y=b;
  }
  public void translate(double a,double b) {...}
  public double distance(Point other) {...}
}
```

- **Tutti gli oggetti di classe Point hanno origine degli assi di riferimento comune**

L'ereditarietà

- **Riuso del sofware: progettare componenti per differenza**
- **La tipizzazione classica garantisce la coerenza dei tipi durante la compilazione, ma non permette la combinazione di tipi anche se voluta**
- **L'ereditarietà (*inheritance*) fornisce i meccanismi necessari per una combinazione controllata**

Esempio: poligoni

- Supponiamo di voler costruire una libreria di elementi grafici: punti, segmenti, vettori, cerchi, poligoni in generale, rettangoli, quadrati ...
- La classe dei punti è:

```
class Point { // una semplice classe
  private double x, y;
  Point (double x, double y) ...;
  public void translate(double a, double b)...;
  public double distance (Point p)...;
}
```

- Definiamo poligoni e rettangoli

Esempio: `class Polygon`

```
class Polygon {
  Point vertices[];

  public Polygon(...){ … } // il costruttore

  public void translate(double a, double b){
    // sposta di a orizzontale, b verticale
    for (int i =0; i < vertices.length; i++)
      vertices[i].translate(a, b);
  } // translate

  public void rotate(Point center,double angle){
    ... // ruota di angle intorno a center
  } // rotate
```

Esempio: class Polygon

```
public double perimeter() {  // perimetro
  Point first;
  Point second;
  double result = 0;
  for (int i = 0; i < vertices.length-1; i++) {
   first = vertices[i];
   second = vertices[i+1];
   result += first.distance(second);
  }
  return result += vertices
    [vertices.length-1].distance(vertices[0]);
} // perimeter
…

} // class Polygon
```

Costruttore

- **Dato:**
  double punti[]={$x_1, y_1, x_2, y_2, x_3, y_3, \ldots, x_n, y_n$}
- **Un costruttore:**

```
public Polygon (double punti[]) {
  int x = 0;
  int y = 1;
  vertices = new Point[punti.length/2];
  for (int i=0; i<vertices.length; i++) {
    vertices[i]=new Point(punti[x], punti[y]);
    x=x+2;
    y=y+2;
  }
}
```

- **Un altro costruttore**

```
public Polygon (Point[] punti) {
  vertices = punti;
}
```

Esempio: class Rectangle

- Un rettangolo è un poligono con alcune proprietà aggiuntive (ha 4 lati, 4 angoli retti, una diagonale, ...).
- La definizione della classe dei rettangoli può essere costruita partendo dalla classe dei poligoni.

Esempio: class Rectangle

```
class Rectangle extends Polygon {
  double side1, side2; // i due lati
  double diagonal; // la diagonale
  Rectangle(Point center, double s1,
            double s2, double angle)
    // crea rett. centrato in center, con lati
    // di lunghezza s1 e s2 e ruotato di angle
  { super(…) // invoco il costruttore del padre
    ...
  } // Rectangle

  public double perimeter() { // ridefinizione
          return 2 * (side1 + side2)
  } // perimeter

 } // class Rectangle
```

Esempio: `class Rectangle`

- `Rectangle` **è un erede di** `Polygon`
  - **tutte le variabili e le operazioni della classe genitrice sono applicabili alla nuova classe**
- **Bisogna garantire che l'oggetto della classe padre sia costruito correttamente**
  - **Si realizza con la chiamata**

```
super(...); // con i giusti parametri
```

Coerenza tra tipi

- **L'ereditarietà soddisfa le regole di tipo**
- **Date le dichiarazioni**

```
Polygon p = new Polygon(…);
Rectangle r = new Rectangle(…);
```

- **sono legali:**

```
p.perimeter()       // definito per i poligoni
p.translate(...)    // con giusti parametri
r.diagonal, r.side1, r.side2
r.translate(...)    // ereditata
r.perimeter()       // definito per i rettangoli
```

Polimorfismo

- Polimorfismo indica l'abilità di assumere forme (*identità*) differenti
- Per la programmazione OO significa che una entità ha la capacità di riferirsi a run-time a istanze di classi diverse
- Per i linguaggi tipizzati questa capacità viene vincolata dall'ereditarietà
  - Il polimorfismo è la chiave per rendere più flessibile il type system.

Polimorfismo

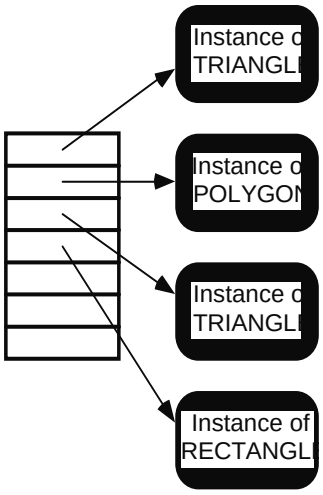
- Principio di sostituibilità:

  Una entità di tipo `Polygon` può riferirsi ad un oggetto di tipo `Rectangle`, ma non viceversa
- Per esempio si può definire un array polimorfo:

```
Polygon[] a;
```

NB: In Java le variabili sono considerate references agli oggetti, non oggetti

Array polimorfi



Dynamic Binding

- **La regola conosciuta come dynamic binding stabilisce che**
  - **l'identità dell'oggetto al momento della chiamata determina l'operazione da applicare**
  - **la selezione dell'operazione da applicare è fatta dinamicamente**

Esempio

```
double x;
Polygon p = new Polygon(...);
Rectangle r = new Rectangle(...);

x = p.perimeter();
```

- viene selezionata la procedura di `Polygon`

```
x = r.perimeter();
```

- viene selezionata la procedura di `Rectangle`

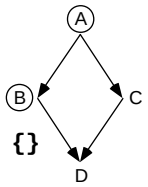
```
p = r;
x = p.perimeter();
```

- viene selezionata la procedura di `Rectangle`

Ereditarietà multipla

- Java prevede
  - ereditarietà singola per le classi
  - ereditarietà multipla per le interfacce
- Ciò permette di non avere conflitti
- Esempio

```
interface A {}
interface B extends A {}
class C implemets A {}
class D extends C implements B {}
```



Eccezioni

- Java ha un meccanismo di rilevamento e gestione delle eccezioni indipendente dal codice "normale"
- Esistono due tipi di eccezioni (indistinguibili per chi le riceve)
  - eccezioni predefinite (*unchecked*)
  - eccezioni definite dal programmatore (*checked*)
- È obbligatorio trattare quelle *checked*

Eccezioni

- I metodi che possono generare eccezioni lo segnalano con clausole esplicite `throws`
- Esempio

```
void foo()
       throws E1 {
  ...
}
```

- Le eccezioni
  - devono essere “gestite”
  - oppure devono essere “rilanciate” esplicitamente

Gestione delle eccezioni

- **Si può "gestire" (`catch`) una eccezione con il costrutto try-catch**
- **Esempio**

```
void esempioEccezioni1() {
      ...
  try {
      ... foo() ...
      ... // azioni che possono provocare E1 e E2
  }
  catch (E1 e) {
      ... // gestione dell'eccezione E1
  }
  catch (E2 e) {
      ... // gestione dell'eccezione E2
  }
   ...
}
```


Gestione delle eccezioni

- **Si puo' "rilanciare" (`throws`) una eccezione**
- **Esempio**

```
void esempioEccezioni2()
       throws E1 {
        ...
  try {
      ...  foo() ...
      ... // azioni che possono provocare E1 e E2
  }
  catch (E2 e) {
      ... // gestione dell'eccezione E2
  }
}
```

Gestione delle eccezioni

- Quando si verifica una eccezione
  - il blocco `try` in cui si è verificata l'eccezione `X` termina l'esecuzione
  - il controllo viene passato al blocco `catch`
  - se esiste un gestore per l'eccezione `X` allora viene eseguito [*il primo compatibile*]
  - altrimenti l'eccezione viene fatta propagare
- Se l'eccezione viene gestita, allora il blocco `try-catch` termina correttamente e il programma continua come se non si fosse verificata l'eccezione

Propagazione delle eccezioni

- Le eccezioni sono dei segnali trasmessi lungo la catena delle chiamate
  - Se una eccezione viene gestita, allora la normale esecuzione del programma può proseguire
  - Se una eccezione non viene gestita, allora questa viene trasmessa all'entità chiamante
  - Se al termine delle trasmissioni non viene fornito alcun gestore, allora viene eseguito un gestore di default
    (di solito genera la stampa di un messaggio)

La clausola `finally`

- **La clausola `finally` permette di ripristinare uno stato consistente**

```
try {
    ... // azioni che possono provocare E1 e E2 }
catch (E1 e) {
    ... // gestione dell'eccezione E1 }
catch (E2 e) {
    ... // gestione dell'eccezione E2 }
finally { // opzionale
    ... }
```

- **La clausola finally viene eseguita comunque (dopo la `try` o dopo una `catch`) prima di cedere il controllo**

Definizione di eccezioni

- **Le eccezioni sono regolari oggetti che vanno dichiarati come istanze di classe**
- **Le classi devono essere estensioni della classe `Exception` o della classe `Throwable`**

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
```

Dichiarazione di eccezioni

- **Esempio:**

```
public class MyException extends Exception {
  public Object v;

  MyException (Object value) {
    super("errore numero " + value);
    v = value;
  }
}
```

La classe `Throwable`

- **I principali elementi della classe** `Throwable` **sono**

```
public class java.lang.Throwable
  extends java.lang.Object
  implements serializable
{
  public Throwable();
  public Throwable(String msg);
  . . .
  public String getMessage();
}
```

Attivazione esplicita

- **Si può lanciare una eccezione con** `throw`

```
public void esempio (int valore)
  throws MyException {
  if (valore != 0)
    throw new MyException(new Integer(valore));
}
```

- **Solo le eccezioni dichiarate con** `throws` **possono essere lanciate con** `throw`
- **Naturalmente potrebbe comunque verificarsi una eccezione *unchecked***

Esempio

- **Scriviamo un programma che gestisce una semplice coda utilizzando le eccezioni**

1. definire la classe QueueException

```
class QueueException extends Exception {
  public QueueException(String msg) {
    super(msg) ;
  }
}
```

Esempio

2. definire la classe Coda

```
class Coda { // con array circolare

  // la parte privata
  private final static int dim = 10 ;
  private int out = 0 ;
  private int in = 0 ;
  private int conta = 0 ;
  private Object[] items = new Object[dim];
```

Esempio

```
// la parte pubblica
  public void accoda(Object x)
              throws QueueException {
    if (conta >= dim)
      throw new QueueException(”Coda piena") ;
    conta++ ;
    items[in] = x ;
    in = (in + 1) % dim ;
  }
  public Object rimuovi()
              throws QueueException {
    if (conta <= 0)
      throw new QueueException(”Coda vuota");
    conta--;
    Object temp = items[out] ;
    out = (out + 1) % dim ;
    return temp ;
  }
}
```

Esempio

3. il programma che usa Coda

```
public class Esempio {

  public static void main(String args[]) {
    System.out.println(”Inizio programma") ;
    Coda q = new Coda() ;
    try { // aggiunge elementi
      for (int i = 0 ; i < 5 ; ++i) {
        q.accoda(new Integer(i)) ;
      }
    }
    catch (QueueException e) {
      System.out.println(”Accodamento fallito");
    }
```

Esempio

```
    try {
      for (int i = 0 ; i < 5 ; ++i) {
        Integer n = (Integer)q.rimuovi() ;
        System.out.println(”Rimosso: " + i) ;
      }
    }
    catch (QueueException e) {
      System.out.println(e.getMessage()) ;
    }
    System.out.println(”Fine programma") ;
  }
}
```

Programmazione Concorrente

Programmazione concorrente

- è conveniente poter eseguire più programmi “contemporaneamente”
  - sfrutta i tempi morti della computazione interattiva
  - migliora le prestazioni
  - semplifica il progetto del software
- Java fornisce strumenti per programmi
  - concorrenti
  - cooperanti

Thread

- **Un thread è anche definito *lightweight process* o *execution context***
  - **È un singolo flusso di controllo sequenziale**
  - **Consente esecuzione parallela**
- **In Java è realizzato dalla classe** `Thread`
  - **Bisogna estendere** *Thread* **per ereditare e creare un nuovo flusso di controllo**
  - **È possibile creare oggetti runnable implementando l'interfaccia** *Runnable*

Esempio

```
class SimpleThread extends Thread {

  public SimpleThread(String str) { // str è l'id
    super(str); // chiama il costruttore di Thread
  }

  public void run() { // è il body del thread
    for (int i = 0; i < 10; i++) {
      System.out.println(i + " " + getName());
      try {
        sleep((long)(Math.random() * 1000));
      } catch (InterruptedException e){
        System.out.println(”ERROR: interrupted sleep”)
      }
    }
    System.out.println("DONE! " + getName());
  }
}
```

L'esecuzione di un Thread

- **Per creare un nuovo thread**

```
Thread myThread = new MyThreadClass();
```

- **Per renderlo "runnable "**

```
myThread.start();
```

- **Esempio**

```
class Esempio {
  public static void main (String args[]) {
    new SimpleThread("Primo thread").start();
    new SimpleThread("Secondo thread").start();
  }
}
```

L'esecuzione di un Thread

- **Per sospendere un thread (not runnable)**

```
Thread.sleep(1000); // millisecondi 1000=1 sec
wait (); // sospeso
```

- **Per riattivarlo**

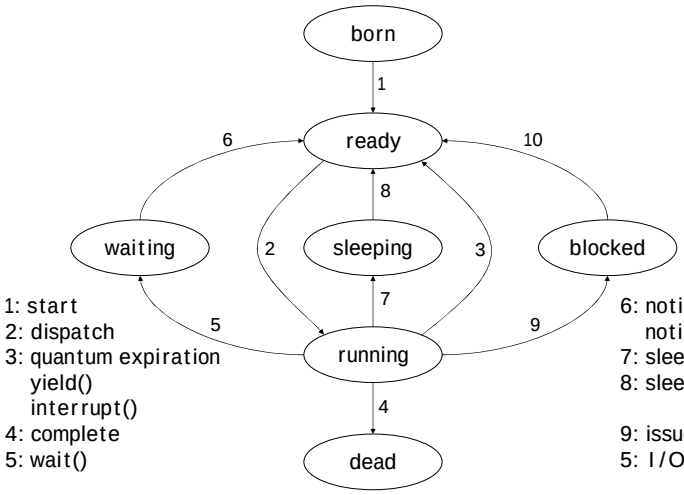
```
notify(); // risveglio un thread dall'insieme
  dei thread precedentemente sospesi
notifyAll(); // risveglio tutti i thread
  dell'insieme dei thread precedentemente sospesi
```

- **Per interromperlo**

```
myThread.interrupt(); // risveglia da una
  sospensione
```

- **Al termine dell'esecuzione del metodo** `run` **il** `Thread` **muore**

Gli stati di un Thread



1: start
2: dispatch
3: quantum expiration
yield()
interrupt()
4: complete
5: wait()

6: notify()
notifyAll()
7: sleep()
8: sleep interval
expires
9: issue I/O
5: I/O completion

Esempio Runnable

```
public class RunnableThread implements Runnable {
  private String nome;
  public RunnableThread (String q) {
       nome = new String (q);
  }
  public void run () {
    for (int i = 0; i < 20; i++) {
      long sleepTime = (long) (Math.random() * 1000);
      try {
        Thread.sleep (sleepTime);
      } catch (InterruptedException e){
        System.out.println(”ERROR: interrupted sleep”)
      }
      System.out.println(i+" "+nome+" "+sleepTime);
    }
    System.out.println ("Finito: " + nome );
  }
}
```

Esempio runnable

- **Creazione d esecuzione dei thread**

```
public class TwoRunnableThread {

  public static void main (String [] args) {

    RunnableThread f1 = new RunnableThread
  ("Alpha");
    RunnableThread f2 = new RunnableThread
  ("Delta");

     new Thread (f1).start();
     new Thread (f2).start();

  }
}
```

Monitor

- **Java usa monitor per la sincronizzazione**
- **mutua esclusione:**
  - **ogni metodo viene eseguito da un solo thread per volta**
  - **gli altri thread non possono interromperlo**
- **sospensione:**
  - **un thread può**
    sospendere la propria esecuzione
    riattivare l'esecuzione di altri thread

Mutua esclusione

- **I metodi di una classe possono essere definiti** `synchronized`
  - **i metodi synchronized sono eseguiti in mutua esclusione**
  - **se un thread effettua una chiamata di un metodo durante l'esecuzione di un metodo sinchronized, la chiamata viene posticipata (esiste una coda associata alla classe)**

Sospensione

- **Utilizzando i metodi** `wait` **e** `notify` **della classe** `Object` **è possibile controllare l'esecuzione dei thread**
- **La** `wait` **sospende il thread che la esegue**

Sospensione

```
public final void wait()
            throws InterruptedException
```

- **aspetta per un tempo indeterminato**

```
public final void wait(long timeout)
            throws InterruptedException
```

- **aspetta per timeout millisecondi o finché non viene risvegliato**
- **se timeout = 0 aspetta finché non viene risvegliato**

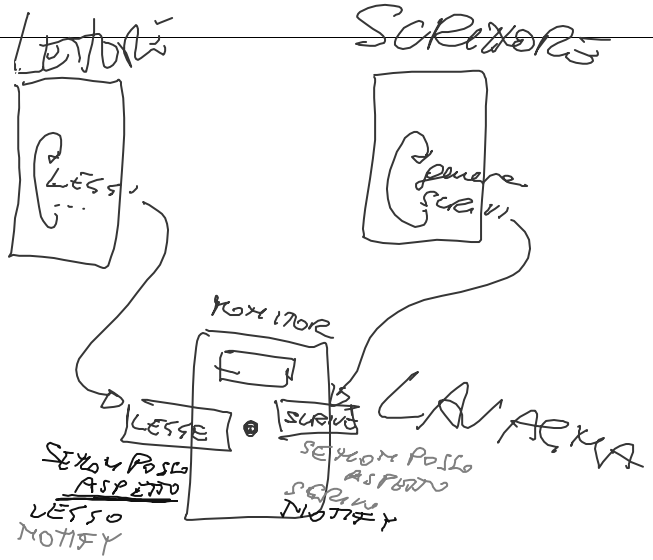
Sospensione

- **La** `notify` **risveglia un processo sospeso sulla coda associata alla classe in esecuzione**
  - **la notifica non può essere nominale**
  - **si può svegliare uno (non il primo!) o tutti**

```
public final void notify()
public final void notifyAll()
```

Esempio

- **Produttore - consumatore**
  - **Il produttore genera numeri da 0 a 9 e li scrive in un buffer**
  - **Il consumatore legge dal buffer i numeri prodotti dal produttore**
  - **Non si devono perdere dati**



Produttore

```
class Producer extends Thread {
  private Buffer buf;
  private int num;

  public Producer(Buffer b, int num) {
    buf = b;
    this.num = num;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      buf.put(i);
      System.out.println("Produttore #" +
       this.num + "ha inserito il valore: " + i);
    }
  }
}
```


Consumatore

```
class Consumer extends Thread {
  private Buffer buf;
  private int num;

  public Consumer(Buffer b, int num) {
    buf = b;
    this.num = num;
  }

  public void run() {
    int valore = 0;
    for (int i = 0; i < 10; i++) {
      valore = buf.get();
      System.out.println("Consumatore # " +
         this.num + " ha letto il valore: " + valore);
    }
  }
}
```

La classe Buffer

```
class Buffer {
  private int seq;     // la variabile buffer
  private boolean available = false;

  public synchronized int get() {
    while (!available) {
      try {
        wait();
      } catch (InterruptedException e) { }
    }
    available = false;
    notify();
    return seq;
  }
```

La classe Buffer

```
  public synchronized void put(int value) {
    while (available) {
      try {
        wait();
      } catch (InterruptedException e) { }
    }
    seq = value;
    available = true;
    notify();
  }
}
```

Un esempio di esecuzione

```
class Test {
  public static void main () {
    Buffer b = new Buffer();
    Producer P = new Producer(b, 1);
    Consumer C = new Consumer(b, 1);
    P.start();
    C.start();
  }
}
```

Pattern di riferimento

- Esistono due principali situazioni
  - Il monitor contiene la risorsa condivisa
    es. Il buffer di scambio dati
  - Il monitor coordina l'accesso ad una risorsa condivisa
    es. Accesso ad una stampante
- Se la risorsa è esterna il monitor si struttura in acquisizione/rilascio della risorsa stessa

Acquisisci/rilascia

- **Schema base Thread**

```
class Agente extends Thread {
  Coordinatore coordina;

  void run () {
    coordina.acquisisci();
    //  uso della risorsa
    coordina.rilascia();
  }
}
```

- **NOTA: si suppone che i clienti/utenti siano onesti, che rispettino le regole**

Acquisisci/rilascia

- **Schema base del monitor**

```
class Coordinatore {

  boolean occupato;

  synchronized void acquisisci() {
    while (occupato)
      try { wait(); } catch …
    occupato = true;
  }

  synchronized void rilascia() {
    occupato = false;
    notify();
  }
}
```

Esame del 6/96

- **Si vuole modellare, con thread e monitor, un sistema per la gestione delle code d'accesso a 3 stampanti.**
  Si progetti un sistema che definisce 10 clienti.
- **Ogni cliente effettua in ciclo le seguenti operazioni:**
  1. **guarda lo stato delle file alle stampanti**
  2. **sceglie una stampante e si accoda alla fila di quella stampante**
  3. **al risveglio utilizza la stampante per le proprie stampe**
  4. **rilascia la stampante**

Soluzione

```
public class Esame {

  public static void main (String args[]) {
    int n_clienti = 10;
    int n_stampanti = 3;
    Stampante stampanti[];
    stampanti = new Stampante[n_stampanti];
    for (int i = 0; i < n_stampanti; i++)
      stampanti[i] = new Stampante();
    for (int i = 1; i < n_clienti; i++)
      new Cliente(i, stampanti).start();
    System.out.println("il main termina");
  }

}
```

Classe Cliente

```
class Cliente extends Thread {

  int nome;
  Stampante stampanti[];
  int N = 2; // numero di stampe di un cliente

  Cliente(int id, Stampante stampanti[]) {
    nome = id;
    this.stampanti = stampanti;
  }
```

Classe Cliente

```
  public void run() {
    Stampante scelta_stampante = stampanti[0];
    int lung_fila = stampanti[0].fila();
    for (int i = 0; i < N; i++) {
      for (int j = 1; j < stampanti.length; j++)
        if (stampanti[j].fila() < lung_fila)
          scelta_stampante = stampanti[j];
      scelta_stampante.vorrei_stampare(nome);
      yield(); // uso della stampante
      scelta_stampante.ho_stampato(nome);
    }
    System.out.println("il cliente " + nome
                       + " termina");
  }
}
```

Classe Stampante

```
class Stampante {

  int n_clienti_in_coda = 0;
  boolean stampante_occupata = false;

  synchronized int fila() {
    return n_clienti_in_coda;
  }
```

Classe Stampante

```
 synchronized void vorrei_stampare(int cliente) {
    while (stampante_occupata) {
      n_clienti_in_coda += 1;
      System.out.println("il cliente " + cliente
                         + " si sospende");
      try { wait(); }
      catch (InterruptedException e) {  };
      n_clienti_in_coda -= 1;
    }
    System.out.println("il cliente " + cliente
                    + " si aggiudica la stampante");
    stampante_occupata = true;
  }
  synchronized void ho_stampato(int cliente) {
    System.out.println("il cliente " + cliente
                       + " libera la stampante");
    stampante_occupata = false; notify();
  }
}
```

Esame del 6/97

- Si vuole progettare un sistema in cui il programma principale genera N agenti ed affida loro altrettanti numeri casuali.
- Ogni agente ha il compito di stampare il numero che gli è stato affidato in modo da produrre una lista ordinata.

Struttura

Esempio (esame del 6/97)

- Lo schema del programma principale è il seguente:

```
class Esame {
  static final int N=20;
  // ...
  public static void main(String[] args) {
    int yourName, yourNumber;
    // ...
    for (int i=0; i < N; i++) {
      yourName = i;
      yourNumber = (int)(Math.Random() * 100);
      new Agente(yourName, yourNumber).start();
    }
    // ...
  }
}
```

Esempio (esame del 6/97)

- Ogni agente deve stampare il numero affidatogli, e solo quello, con l'istruzione:

```
System.out.println(myName + ": " + myNumber);
```

  dove `myName` e `myNumber` sono i valori passati al costruttore dal main.
- Si chiede di completare e modificare opportunamente la classe `Esame`, realizzare la classe `Agente` ed eventualmente altre classi ritenute utili.

Soluzione

```
public class Esame {
  static final int N=20;
  public static void main(String[] args) {
    int yourName, yourNumber;
    Monitor gestore = new Monitor(N);
    for (int i=0; i < N; i++) {
      yourName = i;
      yourNumber = (int)(Math.random() * 100);
      System.out.print(yourNumber + " ");
      new Agente
            (yourName, yourNumber, gestore).start();
    }
    System.out.println();
  }
}
```

Soluzione: classe Agente

```
class Agente extends Thread {
  int myName, myNumber;
  Monitor gestore;

  Agente(int name, int number, Monitor gestore) {
    myName = name;
    myNumber = number;
    this.gestore = gestore;
  }

  public void run () {
    gestore.mayI(myName, myNumber); //posso stampare?
    System.out.println(myName + ": " + myNumber);
    gestore.done(); // ho stampato!
  }
}
```

Soluzione: classe Monitor

```
class Monitor {

  int numeri[]; // i numeri random
  int agenti[]; // gli agenti in ordine di stampa
  int conta = 0;
  int indice = -1;

  Monitor(int N) {
    numeri = new int[N];
    agenti = new int[N];
  }
```

Soluzione: classe Monitor

```
synchronized void mayI(int name, int number) {
  numeri[name] = number;
  if (++conta == numeri.length) {
    // se si sono registrati tutti
    sort(); // ordina i valori
    indice = 0; // il primo da stampare in agenti
    notifyAll(); // sveglia tutti i thread
  }
  while((indice==-1) || (agenti[indice] != name)){
    // finche' indice non valido o non tocca a me
    try { wait(); }
    catch (InterruptedException e) {
      System.out.println("Eccezione in MayI");
    }
  }
}
```

Soluzione: classe Monitor

```
synchronized void done() {
  indice++; // avanti il prossimo
  notifyAll();
}
void sort() {
  // inserisce in agenti gli indici corrispondenti
  // all'ordinamento di numeri
  int min;
  for(int i=0; i < numeri.length; i++) {
    min = 100;
    for(int j=0; j < numeri.length; j++){
      if(numeri[j] < min) {
        min=numeri[j];
        indice=j;
      }
    agenti[i] = indice;
    numeri[indice] = 100;
 } } }
```

Applets

- **Gli applet**
  - **permettono di eseguire veri programmi in ambito World Wide Web**
  - **vengono caricati con pagine html**

```
<HTML>
<HEAD> <TITLE> Una pagina HTML </TITLE> </HEAD>
<BODY>
Un programma Java:
<APPLET CODE="Name.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

Applets

- Grazie ai multithread e al caricamento dinamico è possibile attivare più programmi nella stessa pagina
- I programmi possono essere indipendenti o comunicare tramite monitor
- Java include una libreria di interfaccia socket che permette di scrivere applicazioni client/server

La struttura di un applet

```
import java.awt.Graphics;

public class Simple extends java.applet.Applet {
  StringBuffer buffer = new StringBuffer();
  public void init() {
    resize(500, 20);
    addItem("initializing... ");
  }
  public void start() {
    addItem("starting... ");
  }
  public void stop() {
    addItem("stopping... ");
  }
```

La struttura di un applet

```
public void destroy() {
    addItem("preparing for unloading...");
  }
  public void addItem(String newWord) {
    System.out.println(newWord);
    buffer.append(newWord);
    repaint();
  }
  public void paint(Graphics g) {
    g.drawRect(0,0,size().width - 1,
  size().height - 1);
    g.drawString(buffer.toString(), 5, 15);
  }
}
```

L'esecuzione di un applet

- **Viene creato un thread quando viene caricata la pagina**
- **L'interprete chiama i metodi** `init()` **e** `start()` **per far partire l'esecuzione**
- **Ogni volta che viene abbandonata/ripresa la pagina, vengono invocati i metodi** `stop()` **e** `start()`
- **Prima di terminare l'esecuzione viene invocato il metodo** `destroy()`

La grafica degli applet

- **È possibile controllare l'aspetto grafico di un applet e gestire eventi con la libreria AWT (abstract window toolkit)**
- **Alcuni metodi sono**
  - *paint(), update()*
  - *mouseEnter(), mouseExit(), mouseMove(), mouseUp(), mouseDown(), mouseDrag(), keyDown()*

Metodi per trattare tag HTML

- **La classe** `java.net.URL` **contiene metodi per trattare e leggere URL**
  - **È possibile specificare una URL così:**

```
...
URL documentURL = new
URL("http://www.dsi.unimi.it");
URL imageURL= new URL(documentURL,
"images/figura.gif");
...
System.out.println(imageURL);
```

  - **L'esecuzione produce la scritta**

```
http://www.dsi.unimi.it/images/figura.gif
```

Metodi per trattare tag HTML

- **All'interno di un applet sono utili i metodi:**
  `URL getDocumentBase()`
  - **restituisce la URL del documento che contiene l'applet**
    `URL getCodeBase()`
  - **restituisce la URL dell'applet stesso**
    `String getParameter(String name)`
  - **restituisce il valore del parametro specificato**

Grafica

- **Si possono caricare immagini con i metodi**
  `Image getImage(URL url)`
  - **carica l'immagine individuata dalla URL assoluta**
  - **l'immagine viene caricata solo la prima volta**
    `Image getImage(URL url, String name)`
  - **carica l'immagine individuata dalla URL assoluta e dal nome relativo**

Audio

- **Java definisce entità audio dette clip**
  ```
  AudioClip getAudioClip(URL url)
    AudioClip getAudioClip(URL url String name)
  ```
  - **carica l'audio clip individuato dalla URL**
  - **viene caricato solo la prima volta**
    ```
    void play(URL url)
      void play(URL url,String name)
    ```
  - **suona l'audio clip individuato dalla URL**
  - **se non esiste non succede nulla**

Audio

- **La classe AudioClip ha tre metodi che permettono di controllarne l'esecuzione**
  ```
  void play()
  ```
  - **inizia l'esecuzione**
    ```
    void loop()
    ```
  - **esegue in ciclo**
    ```
    void stop()
    ```
  - **sospende l'esecuzione**
- **Possono essere attivi più audio clip contemporaneamente**

Manipolare l'ambiente

- **Con alcune restrizioni è possibile modificare l'ambiente in cui un applet viene eseguito**
  - **Per esempio può caricare una nuova pagina**
- **L'interfaccia** `java.applet.AppletContext` **definisce i metodi opportuni**
  - **Dalla classe applet:**
    `AppletContext getAppletContext()`
  - **restituisce il contesto dell'applet (dipende dall'implementazione)**

`java.applet.AppletContext`

- **Si possono controllare alcuni aspetti dell'ambiente:**
  `abstract Applet getApplet(String name)`
  - **restituisce l'applet con il nome specificato o null se non esiste nella pagina corrente**
    `abstract Enumeration getApplets()`
  - **restituisce gli applets accessibili**
    `abstract void showDocument(URL url)`
  - **sostituisce la pagina corrente con la nuova pagina**