Java come linguaggio object-oriented

Lezioni basate su un seminario tenuto da
D. Ancona ed E. Zucca

Dicembre 1997

Java come linguaggio object-oriented

Lezioni basate su un seminario tenuto da
D. Ancona ed E. Zucca

Dicembre 1997

L'obiettivo è di:

- ricapitolare i principi dell'approccio object oriented
- illustrare come questi principi sono realizzati in Java
- presentare altre caratteristiche del linguaggio Java

Approccio object oriented = oggetti + classi + inheritance

- estendibilità e riusabilità del software
- incapsulazione (interfaccia e implementazione)
- paradigma "simile al mondo reale"

Note storiche

- Simula67 (Dahl e Nygaard, Università di Oslo, fine anni '60): linguaggio per simulazioni, riconosciuto come il primo linguaggio o-o

- Smalltalk (Goldberg e Robson, Xerox PARC, primi anni '70): introduzione della terminologia o-o, prototipo dei linguaggi o-o. Sistema per lo sviluppo di applicazioni principalmente grafiche. È interpretato e non permette type checking statico.

- Linguaggi "ibridi" (metà anni '80): Object-Pascal, C++, Eiffel e molti altri.

- Java (Arnold e Gosling, SUN, 1996): o-o puro, con type-checking statico più altre caratteristiche avanzate.

Java è un linguaggio sviluppato dalla Sun Microsystems già nel 1991 (sotto il nome di Oak) con l'obiettivo di avere un linguaggio piccolo e facilmente portabile.

È diventato di interesse generale quando è stato accoppiato con HotJava (1994), un World Wide Web browser in grado di caricare da un server *applet* Java cioè mini-applicazioni che possono essere eseguite sulla macchina client)

HotJava ha fornito anche un esempio di applicazione complessa scritta in Java.

Tuttavia, è anche un linguaggio general-purpose

- object oriented "puro" - con un insieme di librerie di classi
- disegnato per aiutare a programmare correttamente (garbage collection automatica, controlli di tipo, eccezioni, . . .)
- semplice (uno dei goal iniziali di design)
- portabile, cioè indipendente dalla piattaforma. È sufficiente che sia implementata la macchina virtuale Java.Infatti il codice Java è tradotto dal compilatore Java in *bytecode* che sono eseguiti dall'interprete Java. Questo interprete è "built-in" nei browser abilitati Java. Lo svantaggio è nel tempo di esecuzione.
- "familiare", cioè simile a C, C++ e altri linguaggi (Eiffel, Smalltalk, . . .)

La versione corrente è Java Development Kit 1.1

In Java si possono scrivere applet o applicazioni stand-alone. Un'applicazione consiste di una o più classi di cui una con un metodo `main()`. Un esempio di applicazione:

```
class HelloWorld {
    public static void main (String args[]) {
       System.out.println("Hello World!");
    }
}
```

Il file sorgente deve chiamarsi HelloWorld.java

Il compilatore produce un file .class per ogni classe o interfaccia

Una classe dotata di un metodo main è un'applicazione, cioè è eseguibile

Classi

Esempio:

```
class Rectangle {
  int length,width;
  void setLength (int l) {length = l;}
  void setWidth (int w) {width = w;}
  int area () {return width * length;}
  void print ()
   {System.out.println("I am a rectangle:  = "
    + length + ", " + width);}
}
```

Campi: length, width

Metodi: setLength, setWidth, area, print

campi = dati, metodi = codice

Oggetti

Esempio:

```
class Rectangle {
. . .
   public static void main(String args[]){
     Rectangle r=new Rectangle();
     r.setLength(2);r.setWidth(3);r.print();}
```

Esecuzione:

```
I am a rectangle:  = 2, 3
```

La classe Rectangle introduce il *tipo-oggetto* Rectangle e fornisce uno schema per creare oggetti di tipo Rectangle

Rectangle r=new Rectangle();

L'oggetto r è dotato di un proprio *stato* (i campi length e width)

Regola: lo stato non dovrebbe essere visibile all'esterno (*encapsulation*)

Lo stato viene modificato ed ispezionato tramite i metodi della classe

Nozione di oggetto

Interfaccia: interazione dell'oggetto col mondo esterno (solo metodi)

```
interface RectangleInterface
{void setLength(int l);
 void setWidth(int w);
 int area();
 void print();}
```

Stato: ogni oggetto della classe ha una copia individuale dei campi

Invocazione di un metodo:

r.setLength(2)

La dot-notation suggerisce che il metodo invocato appartiene all'oggetto r: l'invocazione di `setLength` agisce sullo stato di r

Tutti gli oggetti della classe Rectangle hanno lo stesso comportamento

Aspetti sintattici

La sintassi segue lo stile di C e C++. Il punto e virgola è un terminatore. Ci sono tre tipi di variabili: instance variables, class variables, local variables. Non ci sono variabili globali. Le dichiarazioni di variabili locali possono apparire ovunque in una definizione di metodo. Le variabili locali vanno inizializzate, mentre quelle di istanza e di classe hanno valori di default dipendenti dal tipo (`null`, `0`, `'\0`, `false`). Queste ultime inoltre possono essere dichiarate `final` cioè costanti.

Java usa l'insieme di caratteri Unicode, che include non solo i caratteri ASCII ma migliaia di altri caratteri per rappresentare la maggior parte degli alfabeti

Un tipo è:

- uno degli otto tipi predefiniti;
- un nome di classe o interfaccia;
- un array

Gli otto tipi predefiniti sono built-in nel sistema quindi più efficienti. Si noti che sono però completamente indipendenti dalla macchina. Quattro tipi interi (`byte`, `short`, `int`, `long`), due tipi floating-point (`float` e `double`)), `char` (i caratteri Unicode, quindi 16 bit) e `boolean` (`true` e `false`).

Non esiste `typedef`.

Le stringhe sono oggetti di una classe predefinite `String`. Possono essere creati implicitamente scrivendo un literal. Un'oggetto può essere convertito in stringa se la sua classe fornisce un'implementazione del metodo `toString`.

Gli statement sono quelli di C e C++. Non c'è `goto` ma solo la possibilità di utilizzare `break` e `continue` relativi ad un loop più esterno etichettato

Visibilità 1

Assumiamo che tutte le classi siano definite nello stesso file

Campi e metodi possono essere *private*

Campi e metodi *private* sono visibili solamente all'interno della classe in cui sono stati dichiarati

Esempio:

```
class Rectangle {
  private int length,width;
  private void setLength (int l) {length = l;}
  private void setWidth (int w) {width = w;}
  int area () {return width * length;}
  void print ()
   {System.out.println("I am a rectangle:  = "
    + length + ", " + width);}
}
```

`length`, `width`, `setLength` e `setWidth` sono visibili solo dentro la classe Rectangle

Gli altri metodi sono visibili anche all'esterno della classe

Si noti che in Java (a differenza ad esempio di Smalltalk) la visibilità è *per classe* e non *per oggetto*.

```
   class Rectangle{...
     boolean equals (Rectangle r) {
       return length == r.length &&
         width == r.width;}
   ...
  }
```

Corretto in Java e C++, ma non in Smalltalk!

Terminologia

- *object-based*: supporta la nozione di *oggetto* (Modula-2, Ada)
- *class-based*: supporta le nozioni di *oggetto* e *classe* (CLU)
- *object-oriented*: supporta le nozioni di *oggetto*, *classe* e *inheritance* (Smalltalk, C++, Java ...)

Identità

Ogni oggetto ha una propria identità

Esempio:

```
Rectangle r1=new Rectangle(),
          r2=new Rectangle(), r3=r1;
```

r1 ed r2 denotano due oggetti diversi: hanno stesso stato, ma identità diverse (r1 != r2)

r1 ed r3 denotano lo stesso oggetto (r1 == r3)

In pratica, r1 ed r3 contengono lo stesso indirizzo: in Java non esiste una nozione esplicita di *puntatore*, ma viene recuperata in modo "pulito" tramite la nozione di identità degli oggetti

Uguaglianza tra identità: r1 == r3, predefinita

Uguaglianza tra stati: di solito va implementata dall'utente

Esempio:

```
class Rectangle{
. . .
boolean equals (Rectangle r)
   {return length == r.length && width == r.width;}
. . .}
```

Identità

Assegnazione tra oggetti:

```
Rectangle r3=r1;
```

r3 ed r1 sono lo stesso oggetto

```
Rectangle r3=(Rectangle)(r1.clone());
```

r3 ed r1 non sono lo stesso oggetto, comunque r1.equals(r3)==true.

Object clone() è definita nella classe Object ed effettua una *shallow copy* dell'oggetto su cui è invocato: i campi oggetto vengono condivisi. Per avere campi oggetto distinti (*deep copy*) bisogna ridefinire clone().

Creazione e distruzione

Creazione: new C() crea un nuovo oggetto della classe C

Costruttori: al momento della creazione posso utilizzare dei parametri per inizializzare lo stato del nuovo oggetto

Esempio:

```
class Rectangle{
. . .
Rectangle (int l, int w) {
     length = l;width = w;}. . .}

Rectangle r=new Rectangle(2,3);
```

Distruzione: affidata ad un *garbage collector*

Metodo speciale `finalize`: viene invocato al più una volta, subito prima che l'oggetto venga distrutto dal garbage collector

Esempio:

```
class ProcessFile{
   Stream File;
. . .
protected void finalize()
  {if (File != null) File.close();} }
```

Uso di this

Dentro ogni metodo *m* è possibile fare riferimento all'oggetto stesso sul quale *m* è stato invocato, tramite la parola chiave *this*

Esempio:

```
boolean equals (Rectangle r)
   {return length == r.length && width == r.width;}
```

è equivalente a

```
boolean equals (Rectangle r)
   {return this.length == r.length &&
           this.width == r.width;}
```

Utilizzo principale: se ci sono variabili locali che mascherano le instance variables;
passaggio di parametro o restituzione del valore di un metodo

Esempio:

```
Rectangle greaterArea(Rectangle r){
  if (area()>r.area()) return this;
  else return r; }
```

this e i costruttori

Esempio:

```
class Rectangle{
. . .
Rectangle (int l, int w) {
     length = l;width = w;}
Rectangle () {
     this(1,1);}
. . .}
```

Nota: se non vengono definiti costruttori, il linguaggio fornisce un costruttore costante di default (`Rectangle()`)

La definizione del costruttore `Rectangle(int l,int w)` cancella il costruttore di default

In questo caso, se si vuole mantenere il costruttore costante bisogna definirlo esplicitamente (vedi esempio)

this(1,1) è la notazione per invocare esplicitamente un costruttore dentro la definizione di un altro costruttore

tale invocazione deve essere la prima istruzione del corpo del costruttore

Campi e metodi statici (di classe)

Esempio:

```
class Rectangle{
static rectNum=0;
static void printNofRects(){
      System.out.println("Alive rectangles: "+
                                    rectNum);}
Rectangle (int l, int w) {
     length = l;width = w;rectNum++;}
protected void finalize(){rectNum--;};
```

Nota: esiste una sola copia di `rectNum` condivisa da tutti gli oggetti della classe

L'inizializzazione dei campi statici può essere eseguita usando dei *blocchi di inizializzazione statici*

Restrizione sui metodi statici: nessun riferimento a this (sia esplicito, che implicito), dato che il metodo appartiene alla classe e non ai singoli oggetti. Accesso solo a campi e metodi statici

Invocazione:

```
Rectangle.printNofRects(); /* preferibile */
r.printNofRects(); /* corretta, ma sconveniente */
```

I metodi di classe possono anche essere usati per simulare tipi di dato usuali; ad esempio la classe `Math` nel package `java.lang`.

Arrays

Gli array in Java sono considerati come particolari oggetti

Esempio:

```
int[] a = new int[3];
for (int i = 0;i < a.length;i++) a[i]=i;
```

Nota: dimensione fissata al momento della creazione e accessibile tramite il campo *length*. Indici da 0 a *length*-1

Array multi-dimensionali:

```
float [][] mat = new float[4][4];
```

o equivalentemente

```
float[][] mat = new float[4][];
for (int y=0;y < mat.length; y++)
   mat[y] = new float[4];
```

Inizializzazione:

```
int[][] a = {{1},{1,1},{1,2,1},{1,3,3,1}};
```

Nota: ogni array annidato può avere una dimensione diversa

Esempio: 1

```
class RecStack{
   int top;
   RecStack rest;

boolean isEmpty() {
   return rest == null;
}

int pop (){
   if (isEmpty()) /* errore */;
   else if (rest.isEmpty())
      {rest = null; return top;}
   else {
         int oldTop = top;
         top = rest.pop();
         return oldTop;
        }
}

void push (int e){
   if (isEmpty()) rest = new RecStack();
   else rest.push(top);
   top=e;
}
```

Esempio: 1

```
void print (){
   System.out.print("(");
   auxPrint();
   System.out.println(")");
}

private void auxPrint(){
   if (!isEmpty()) {
      System.out.print(top);
      if (!rest.isEmpty()) {
         System.out.print(", ");
         rest.auxPrint();
        }
     }
}

}
```

Esempio: 2

```
class Sieve {
  int myPrime;
  Sieve next;
  static RecStack myStack = new RecStack();

  void filter (int i){
    if (myPrime == 0) {
      myPrime = i;
      next = new Sieve();
      myStack.push(i);
    }
    else if (i%myPrime!=0) next.filter(i);
  }
}

class PNG{
public static void main (String argv[]){
    Sieve first = new Sieve();
    for (int i=2; i < 100 ; i++)
      first.filter(i);
    Sieve.myStack.print();
  }
}
```

Inheritance

Esempio

```
class Cuboid  extends Rectangle {
  private int height;
  void setHeight (int h) { height = h;};
  int volume () { return area () * height;};
};
```

Cuboid = classe *heir*, Rectangle = classe *parent*

A cuboid *is-a* rectangle

Uso:

```
class RectCuboidTest {
  public static void main(String argv[]) {
    Rectangle r = new Rectangle();
    ...
    Cuboid c = new Cuboid();
    c.setLength(3); c.setWidth(3); c.setHeight(3);
    r = c; // polimorfismo
...
  }
};
```

La classe Cuboid riutilizza il codice di Rectangle e lo estende. Nota bene: anche in mancanza del codice sorgente di Rectangle; è sufficiente il codice “oggetto”. In Java la classe Rectangle non va ricompilata ogni volta che si modifica Cuboid

possibilità di estendere e modificare librerie software in modo molto flessibile

L'inheritance si accoppia a polimorfismo nel senso che:

Un'espressione di tipo heir può stare in un contesto di tipo parent

Vale cioè la regola di *subsumption*

$$\frac{e : T \quad T \leq T'}{e : T'}$$

Questo è sensato poichè un oggetto di classe heir sa rispondere (almeno) a tutti i messaggi della classe parent; l'assegnazione opposta

```
c = r;
```

potrebbe causare errore a run-time

Inheritance con overriding

```
class Rectangle { ...
void init () {length = 1; width = 1;}
  void print () {
    System.out.println("I am a rectangle:  = "
    + length + ", " + width);}}

class Cuboid  extends Rectangle { ...
  void init () {length = 1; width = 1;height = 1;}
void print () {
   System.out.println("I am a cuboid:  "
   + length + ", " + width + ", " + height);}}
```

Uso con *late binding* (*dynamic binding*)

```
Rectangle r = new Rectangle(); ...
r.print();//viene invocato print di Rectangle
Cuboid c = new Cuboid(); ...
r = c;
r.print();//viene invocato print di Cuboid
```

nell'ultima linea r ha *tipo statico* Rectangle, *tipo dinamico* Cuboid

Nota: overriding di un campo = hiding del campo della superclasse; per i campi il binding è **statico**

Un esempio di inheritance *non conservativa*

```
public class SetByStack  extends RecStack { ...
 public void push (int e) { ...
  //controlla se l'elemento c'e' gia'
//altrimenti lo inserisce
  };
};
```

A SetByStack *is-like* a RecStack

Si noti che con la ridefinizione cambiano anche gli altri metodi; ad esempio

```
void pushInterval (int low, int hi) { ... };
```

Implementazione del late binding: la versione del metodo da invocare viene cercata nella gerarchia delle classi (un albero poichè l'inheritance è *singola*), fino ad `Object`.

Se non si trova si ha un errore dinamico (messaggio non compreso). Ciò avviene ad esempio per linguaggi non tipati staticamente (Smalltalk) safe type system = se il programma è corretto rispetto ai tipi **non** ci sono errori “messaggio non compreso”

Java ha safe type system; il controllo di tipo è prevalentemente statico (+ ClassCastException per casting down)

Utilità di inheritance + polimorfismo

Figure = Rectangle | Circle | Triangle

Linguaggio tradizionale

```
function area (f: Figure) : real =
case f.kind of
Rectangle : area:= f.length * f.width;
Circle : area:= f.radius * Pi;
Triangle : area:= f.basis * f.height /2
```

Linguaggio object oriented

```
class Figure { ...
  double area () { ... } };
class Rectangle extends Figure { ...
  private double length; private double width;
  double area () { return length * width;};};
class Circle extends Figure { ...
  static final double Pi = 3.1416;
  private double radius;
  double area () { return radius * radius * Pi;};};
class Triangle extends Figure { ...
  private double basis; private double height;
  double area () { return basis * height /2;};
};
```

```
class FigureTest {
  public static void main(String argv[]) {
    Figure[] r = new Figure[10];
    ...
    double areaTot = 0.0;
    for (int i = 0;i<r.length;i++)
      areaTot =+ r[i].area();
    ...
  }
};
```

Un utilizzatore esterno non deve neppure sapere quali tipi di figure esistono

Per aggiungere un nuovo tipo di figura, ad es. Ellipsis, basta definire una nuova classe; non occorre modificare il codice precedente (principio *open-closed*)

Inheritance e SottoTipo

In certi casi sembra ragionevole quando si ridefinisce un metodo specializzare i tipi degli argomenti, tipicamente quando ci sono argomenti dello stesso tipo del ricevitore

```
class Rectangle { ...
  boolean equals (Rectangle r) {
    return length == r.length && width == r.width;};
};
class Cuboid  extends Rectangle { ...
  boolean equals (Cuboid c) {
    return length == r.length && width == r.width
      && height == c.height;
  };
};
```

Problema:

```
Rectangle r = new Rectangle();
Rectangle r1 = new Cuboid ()};
System.out.println(r1.equals(r));
  //staticamente corretto
```

Se venisse invocata la versione di equals in Cuboid, avrei errore dinamico

Java risolve (quasi) il problema permettendo:

Overriding: la lista dei tipi degli argomenti + il tipo di ritorno devono essere identici

Overloading: la lista dei tipi degli argomenti deve essere diversa

Nell'esempio, Cuboid ha quindi **due** metodi `equals`

```
public boolean equals (Rectangle r) //vecchio
public boolean equals (Cuboid c) //nuovo
```

e nell'esempio sopra viene correttamente invocata la versione di equals in Rectangle;

Tuttavia, la risoluzione dell'overloading viene fatta staticamente, quindi

```
  Cuboid c = new Cuboid();
  Rectangle r1 = new Cuboid ()};
  System.out.println(c.equals(r1));
//viene invocata la versione in Rectangle
```

Servirebbero i *multimetodi*

Risoluzione dell'overloading

Più precisamente, l'algoritmo di risoluzione è il seguente:

1. Si cercano tutti i metodi che potrebbero essere invocati = nome giusto e tipi (**statici!**) argomenti compatibili);

2. Se si trova più di un metodo, si elimina via via ogni metodo "meno specifico" di un altro;

3. Se si resta con un solo metodo ok, altrimenti errore

Interfacce e classi astratte

In Java una classe è un tipo. Si usa infatti in due modi diversi

```
Rectangle r; //tipo
r = new Rectangle() //schema per istanziare oggetti
```

Ciò corrisponde a due diverse "anime" dell'inheritance

nozione di sottotipo (gerarchie di specializzazione e classificazione, relazione *is-a*)

modo per riutilizzare software

In realtà, le due nozioni possono essere separate:

Tipo di un oggetto = interfaccia operazionale

Classe di un oggetto = implementazione

In particolare, più classi possono implementare lo stesso tipo

Java accoglie parzialmente questo punto di vista: le classi sono tipi, ma esistono anche tipi "puri" che si chiamano *interfacce*

```
interface Stack {
boolean isEmpty();
  int pop();
  void push (int e);
  void print ();
}

class StackByArray implements Stack { ... };

class RecStack implements Stack{ ... };
```

```
class stacksTest{
  public static void main () {
    Stack s = new RecStack ();
    s.print();
    for (int i=1;i<20;i++) s.push(i);
    s.print();
    for (int i=1;i<5;i++) s.pop();
    s.print();
    s = new StackByArray ();
    s.print();
    for (int i=1;i<20;i++) s.push(i);
    s.print();
    for (int i=1;i<5;i++) s.pop();
    s.print();
  };
};
```

Questo permette di avere anche una forma di *multiple inheritance*; infatti una classe può avere un'unica classe parent, ma può implementare più interfacce (non sorgono infatti problemi di conflitto).

```
interface rectangle{
 void Move(double dx, double dy);
 void Draw();
 void UnDraw();}

interface coloredObject{
 void ChangeColor(color c);}

class coloredRectangle implements
      rectangle, coloredObject {...}

interface stack {
 boolean isEmpty();
 int pop();
 void push (int e);
 void print ();}
```

Classi astratte

Classe astratta = “via di mezzo” tra classe normale ed interfaccia. Tipicamente utile quando si vogliono avere operazioni derivate da un nucleo di primitive.

```
abstract class  List {
  abstract boolean isEmpty ();
  abstract int head();
  abstract  List tail();
  abstract List cons (int elem);
  static List emptyList() { return new EmptyList();
//non si puo' fare astratto};
};
```

```
boolean equals (List list) {
    if (isEmpty()) return list.isEmpty();
    else return head().equals(list.head())
      && tail().equals(list.tail());
  };

  List append (List list)  {
    if (isEmpty()) return list;
    else return tail().append(list).cons(head());
  };

  List reverse () {
    if (isEmpty()) return emptyList();
    else return tail().reverse().
  append(emptyList().cons(head()));
  };
};
```

$\text{List} = \Lambda \mid \text{int} \times \text{List}$

List parent di EmptyList e NonEmptyList

```
class EmptyList extends List{
  boolean isEmpty () { return true;};
  int head () { ... errore };
  List tail () {... errore};
  List cons (int elem)
    { return new NonEmptyList(elem);};
};
```

```
class NonEmptyList extends List {
  private int head;
  private List tail;
  private NonEmptyList copy (){
    NonEmptyList copy = new NonEmptyList(head);
    if (!tail.isEmpty())
      copy.tail = ((NonEmptyList)tail).copy();
    return copy;
  };
  NonEmptyList(int elem) {
    head = elem;
    tail = emptyList();
  }

  isEmpty () {return false;};
  int head () {return head;};
  List tail () {return tail;};
  List cons (int elem) {
    NonEmptyList cons = new NonEmptyList(elem);
    cons.tail = copy();
    return cons;
  };
};
```

Uso di super

super si usa al posto di this per invocare metodi della class parent

```
public class Cuboid  extends Rectangle { ...
  boolean equals (Cuboid c) {
    return super.equals(c)
      && height == c.height;
 }

  void init(){
    super.init();
    height = 1;
}
}
```

cioè in questo caso la ricerca del metodo da invocare inizia dalla classe parent

Inheritance e costruttori

```
class Cuboid  extends Rectangle { ...
  Cuboid () {
    super (); //superfluo
    height = 1;
  };
  Cuboid (int l, int w, int h) {
    super(l,w); height = h;
  };
  Cuboid (Rectangle r){
    super(r); height = 1;
  };
  Cuboid (Cuboid c) {
   this(c.length,c.width,c.height);};
```

I costruttori **non** sono ereditati

Il primo (e solo il primo!) statement di un costruttore può essere:

una chiamata ad un altro costruttore della stessa classe: `this(...);`

una chiamata ad un costruttore della superclasse: `super(...);`

Se non c'è una chiamata di costruttore esplicita si assume una chiamata implicita `super();`

Fasi dell'esecuzione di un costruttore

1. esecuzione della chiamata iniziale (esplicita o implicita) ad un altro costruttore;
2. se la precedente era una chiamata `super(...);`, esecuzione degli inizializzatori dei campi;
3. esecuzione degli altri statement

Quindi quando si crea un'istanza di una classe si eseguono i costruttori di tutte le superclassi (a partire da `Object`) e si inizializzano tutti i campi una volta sola

Casting

In Java esiste il *casting* (conversione di tipo esplicita), ma è controllato. Le cose che si possono fare sono:

- casting tra tipi aritmetici
- casting da un tipo oggetto (classe o interfaccia) ad un suo supertipo `T`, detto *casting up*, *safe casting* o *widening*: sempre corretto, stesso effetto con assegnazione ad una variabile di tipo `T`; serve per accedere ai campi hidden di una superclasse o per risolvere l'overloading
- casting da un tipo oggetto ad un sottotipo, detto *casting down*, *unsafe casting* o *narrowing*: valido solo se si sa che effettivamente l'oggetto ha quel sottotipo (dinamico);

```
Rectangle r = new Cuboid();
Cuboid c = (Cuboid) r;
```

In caso contrario viene sollevata l'eccezione `ClassCastException`

È possibile controllare la validità di un casting con l'operatore `instanceof` che ritorna `true` sse l'oggetto è del tipo (dinamico) indicato

anche: `String name = obj.getClass().getName()`

Regola: ogni volta che si deve usare un casting "pericoloso" ispezionare il tipo dinamico tramite *instanceof*

Utilità del casting: permette di recuperare informazioni sul tipo statico, perse a causa della regola di subsumption

Ad esempio in Cuboid potrei ridefinire

```
public boolean equals (Rectangle r) {
  if (r instanceof Cuboid)
    { return equals((Cuboid)r);}
  else return super.equals(r);
};
```

Allora nel caso visto prima avrei

```
Cuboid c = new Cuboid();
Rectangle r1 = new Cuboid ()};
System.out.println(c.equals(r1));
//viene invocata la versione in Cuboid
```

Eccezioni

```
class StackByArray implements Stack {
  static int maxLength = 15;
  int[] elems = new int[maxLength];
  int length = 0;
  public int pop () throws EmptyStackException {
    if (length==0) throw new EmptyStackException();
    return elems[--length];
  };
  public void push (int e)
    throws FullStackException {
    if (length == maxLength)
      throw new FullStackException();
    elems[length++]=e;
  }; ...
};
```

Le eccezioni in Java sono oggetti, istanze di una classe heir di *Exception*

```
class EmptyStackException extends Exception{}
class FullStackException extends Exception{}
```

Un'eccezione è *sollevata* (*thrown*) quando si verifica una situazione di errore; può essere *catturata* (*caught*) con un *try* block

```
class stacksTest{
    public static void main ()
    throws EmptyStackException{
    s = new StackByArray ();
    for (int i=1;i<20;i++)
      try { s.push(i);}
      catch (FullStackException excp) {
        System.out.println("Full Stack"); break;
      };
    for (int i=1;i<5;i++) { s.pop();}
  };
};
```

Ogni metodo deve dichiarare quali eccezioni possono essere sollevate dalla sua esecuzione (altrimenti errore statico) Ogni metodo che invoca un metodo che dichiara un'eccezione nella sua `throws` clause deve fare una delle seguenti cose (altrimenti errore statico)

- catturare l'eccezione e gestirla
- catturare l'eccezione e "convertirla" in una propria eccezione (cioè dichiarata nella propria `throws` clause)
- dichiarare l'eccezione nella propria `throws` clause e propagarla

Packages

Java permette di raccogliere classi, interfacce e subpackages in un *package*

Un package può essere composto da diversi files (unità di compilazione)

Una dichiarazione di package `P` all'inizio del file indica che le classi e le interfacce nel file appartengono al package `P`

Esempio:

```
package stacks;

public interface Stack {
  boolean isEmpty();
  int pop();
  void push (int e);
  void print ();
}
```

L'interfaccia `Stack` appartiene al package `stacks`

Le classi/interfacce dichiarate `public` sono esportate all'esterno del package

Packages

Regola (ad es. in Unix): ogni file contiene al più una componente pubblica e, in caso affermativo, il nome del file (senza l'estensione .java) deve coincidere col nome di tale componente

Regola (ad es. in Unix): tutti i file di un package si trovano nella stessa directory che deve avere lo stesso nome del package. I subpackage corrispondono con le subdirectory ed il loro nome deve corrispondere col path-name

Esempio: stacks.integer-stacks è il subpackage di stacks contenuto in stacks/integer-stacks.

Un'altra regola è che un package da distribuire in rete inizi con qualcosa che identifichi univocamente il distributore: la convenzione raccomandata è l'indirizzo Internet rovesciato.

Due modi per utilizzare una componente `C` (public) di un package `P`:

- qualified notation: `P.C`
- import esplicito: `import P.C` (`import P.*` per importare tutte le componenti pubbliche - *non* i subpackage)

In caso di conflitto di nomi occorre usare il primo modo.

Le librerie sono organizzate in una gerarchia di package (radice `java`). Le componenti del package `java.lang` sono importate automaticamente.

Se un file sorgente non contiene alcuna dichiarazione di package, tutto il suo contenuto fa parte di un unico *unnamed package*

Si noti che il comando di `import` di Java è molto diverso da `#include`; quest'ultimo ha l'effetto di un'espansione del codice, mentre `import` si comporta come un linker, cioè indica al compilatore ed all'interprete in quali file si trovano le componenti necessarie. Più precisamente, Java usa due cose per trovare una classe: il nome del package e le directory elencate nella variabile `CLASSPATH`.

Visibilità 2

Campi e metodi di una classe possono avere i seguenti *access control modifier* con le seguenti regole di visibilità:

- `private`: visibili solo nella classe che li contiene (quindi non ereditati dalle sottoclassi)

- `package` (ossia nessun access modifier): visibili solo nel package che li contiene (sostituisce la nozione di visibilità dentro lo stesso file di testo del C).

- `protected`: ereditati da ogni sottoclasse; visibili nel package che li contiene e in ogni sottoclasse C (solo attraverso oggetti almeno di tipo C)

- `public`: ereditati da ogni sottoclasse e visibili ovunque la classe che li contiene sia visibile

Una tecnica abbastanza standard nell'object-oriented è quella di dichiarare `private` le instance variables e di definire dei metodi appositi per leggerle/aggiornarle (ad esempio `getLength`, `setLength`). Questo permette di avere variabili read-only. Conviene usare una convenzione uniforme (un'altra possibile è di usare lo stesso nome della variabile).

Visibilità ed inheritance

In generale, non si può ridurre la visibilità di un metodo, ma solo estenderla. Più in dettaglio:

- i metodi dichiarati `public` in una superclasse devono essere `public` nelle sottoclassi

- i metodi dichiarati `protected` in una superclasse devono essere `protected` o `public` nelle sottoclassi

- i metodi dichiarati `private` non sono ereditati

- i metodi dichiarati senza protezione (quindi `package`) non possono essere `private` nelle sottoclassi

Riassunto modificatori

- per controllare l'accesso ad una classe, metodo o variabile: `public`, `protected`, `private`

- `static` per variabili e metodi di classe

- `abstract` per classi e metodi astratti. Le classi astratte non possono avere istanze.

- `final` per classi, metodi e variabili "congelati" cioè "non modificabili"; in particolare, una classe `final` non può avere sottoclassi; un metodo `final` non può essere ridefinito nelle sottoclassi; una variabile `final` è una costante. Dichiarare una classe o un metodo `final` aumenta l'efficienza; molte classi e metodi delle librerie lo sono. Infatti se un metodo è `final` il sistema non deve effettuare la ricerca dinamica. I metodi che leggono/aggiornano le instance variables possono spesso essere `final`. I metodi privati e quelli dichiarati in classi `final` sono di fatto `final`; lo stesso vale per i metodi di classe.

L'ordine usuale dei modificatori è il seguente:

⟨ access ⟩ `static abstract final`

Dichiarare un metodo `private abstract` o `final abstract` produce un errore di compilazione.

I modificatori non si applicano alle variabili locali.

Caratteristiche generali

- object-oriented puro con single inheritance e garbage collector
- fortemente tipato
- interpretato
- indipendente dalla piattaforma: byte code indipendente dall' architettura, specifiche fissate per i tipi base
- multiple threads
- distribuito
- compilazione separata altamente dinamica

Caratteristiche non trattate

- threads
- input/output
- type classes (rappresentazione runtime delle classi)
- utility classes/interfaces
- Java e Internet

Cenni alle librerie

- `java.lang`: `Object`, `String`, `System`, le classi corrispondenti ai tipi primitivi
- `java.util`: `Date`, classi per collezioni (`Vector`, `Hashtable`, . . .)
- `java.io`
- `java.net` (`Socket`, `URL`, . . .)
- `java.awt` Abstract Windowing ToolKit (`Window`, `Menu`, `Button`, . . .)
- `java.applet`

Confronto con C e C++

I progettisti di Java hanno seguito due principi opposti: *simple* e *familiar*

- rimuovere sistematicamente tutte le caratteristiche "supeflue" o "dubbie" di C e C++
- salvare "l'apparenza" simile a C e C++ (e ad altri linguaggi noti come Eiffel, Smalltalk, Ada, . . .)

Differenze

- Java ha il tipo `boolean` che **non** può essere convertito ad un altro tipo
- gli array sono oggetti come gli altri e non puntatori
- le stringhe sono oggetti (classe predefinita `String` e `StringBuffer`) e non array di caratteri
- Java non ha `goto` ma ha `break` etichettati
- Java ha garbage collection automatica (come ogni linguaggio object oriented "puro")
- Java **non ha puntatori!!!** (come ogni linguaggio object oriented "puro")
- Java non ha `typedef` (l'effetto si ottiene con una classe) e `#define` (l'effetto si ottiene con una costante)
- Java non ha struct e union (l'effetto si ottiene con classi e inheritance)
- Java non ha coercion automatiche, non ammette overloading degli operatori e in generale ha un severo controllo di tipo

Somiglianze

- I tipi di dato primitivi sono essenzialmente gli stessi (`byte`, `short`, `int`, `long`, `float`, `double`, `char`)

- gli statement sono essenzialmente gli stessi

- la tecnica per inizializzare oggetti è sostanzialmente quella del C++ (costruttori)

Riassumendo potremmo dire che Java è una versione "ripulita" in senso object oriented puro di C++ (che è un linguaggio a paradigma misto), più altre features (eccezioni, threads, . . .); si può anche vederlo come un linguaggio object oriented che usa C come linguaggio in cui scrivere i body dei metodi

Riferimenti

- "The Java Programming Language" di K. Arnold e J. Gosling, ed. Addison Wesley

- "The Java Language Specification" di J. Gosling, B. Joy, G. Steele ed. Addison Wesley

- gli altri di "The Java Series"

- "Java in 21 Days", di L. Lemay, C. Perkins, M. Morrison.

- http://sunsite.dsi.unimi.it/java/java.html

Applicazioni tipiche

- tipi di dato astratti
- componenti generiche
- strutture dati eterogenee
- comportamento mutabile
- componenti adattabili
- prodotti semi-finiti

Tipi di dato astratti

Le classi sono degli strumenti molto convenienti per definire tipi di dato astratti

Infatti, rispetto ai linguaggi non o-o:

- permettono di evitare di programmare "esplicitamente" con puntatori. In alcuni casi l'utente è esonerato da incarichi spesso "ingrati" (ad es. deallocazione della memoria dinamica)
- forniscono degli strumenti ad hoc per nascondere certi dettagli implementativi e rendere il codice più affidabile

<u>Nota bene</u>: per l'astrazione dei dati non sono necessarie le nozioni di inheritance, subtyping e binding dinamico.

L'astrazione dei dati viene usata praticamente in ogni programma

Inheritance e binding dinamico sono applicabili in casi più limitati

```java
abstract class  List {
  abstract boolean isEmpty ();
  abstract int head() throws EmptyListException;
  abstract List tail() throws EmptyListException;
  abstract List cons (int elem);
  static List emptyList(){return new EmptyList();};

  void print () throws EmptyListException  {
    if (!isEmpty()) {
      System.out.print(head()+" ");
      tail().print();
    }
    else System.out.println(".");
  };

  boolean equals (List list) throws EmptyListException {
    if (isEmpty()) return list.isEmpty();
    else return head()==list.head()
      && tail().equals(list.tail());
  };

  List append (List list) throws EmptyListException {
    if (isEmpty()) return list;
    else return tail().append(list).cons(head());
  };

  List reverse () throws EmptyListException {
    if (isEmpty()) return emptyList();
    else return tail().reverse().
      append(emptyList().cons(head()));
  };
};

class EmptyListException extends Exception {};
```

```java
class EmptyList extends List{
  boolean isEmpty () { return true;};
  int head () throws EmptyListException {
    throw new EmptyListException();};
  List tail () throws EmptyListException {
    throw new EmptyListException();};
  List cons (int elem) {
    return new NonEmptyList(elem,this);
  };
};

class NonEmptyList extends List {
  private int head;
  private List tail;
  NonEmptyList(int elem, List list) {
   head = elem;
tail = list;
  };
  boolean isEmpty () { return false;};
  int head () { return head;};
  List tail () { return tail;};
  List cons (int elem) {
    return new NonEmptyList(elem,this);
  };
};
```

```
class IntervalList extends List {

  private int top,bot;

  IntervalList(int b, int t){ top=t;bot=b;}
  public boolean isEmpty() { return false;}
  public int head() { return top;}
  public List tail() {
    if (top>bot) return new IntervalList(bot,top-1);
    else return new EmptyList();}
  public List cons(int e){
    if (e==top+1) return new IntervalList(bot,e);
    else if (e==bot-1) return new IntervalList(e,top);
else return new NonEmptyList(e,this);
  }

  public void print () {
    System.out.print("("+top);
    for (int i=top-1;i>=bot;i--)
       System.out.print(", "+i);
       System.out.println(")");
}
} // end IntervalList

public interface Stack {
  int pop()  throws EmptyStackException;
  void push (int e);
  void print () throws EmptyStackException;
  boolean isEmpty ();
}
```

```
public class MigrationStack implements Stack{
  private List intRep;
  public MigrationStack(){ intRep=new EmptyList();}
  public MigrationStack(int b, int t){
    if (t<b) intRep=new EmptyList();
    else intRep=new IntervalList(b,t);
  }
  public final boolean isEmpty(){
    return intRep.isEmpty();
  }
  public final int pop() throws EmptyStackException {
    try {
      int i=intRep.head(); intRep=intRep.tail();
       return i;
    }
    catch (EmptyListException excp) {
      throw new EmptyStackException();
    };
  }
  public final void push(int e) {
    intRep=intRep.cons(e);
  }
  public final void print() throws EmptyStackException {
    try { intRep.print();}
    catch (EmptyListException excp) {
  throw new EmptyStackException();
    };
  }
  public void append (Stack s) throws EmptyStackException {
    while (!s.isEmpty()) { push(s.pop());};
  }
 } // end MigrationStack
```

Componenti generiche

Una componente è detta *generica* quando può essere applicata a tipi diversi

Alcuni linguaggi o-o forniscono la possibilità di definire esplicitamente classi generiche (ad es. Eiffel, C++)

In tutti gli altri è possibile definire classi generiche sfruttando inheritance, subtyping, binding e typing dinamico

Esempio: il tipo di dato *albero di ricerca binario*

Tipicamente vorremmo poter scrivere un codice che sia indipendente dal tipo di oggetti contenuti negli alberi

In questo modo si può utilizzare sempre lo stesso codice per implementare alberi di interi, di stringhe e così via

La tecnica:

1. individuare quali operazioni ci si aspetta dalle componenti (in questo caso i nodi dell'albero)

2. definire una o più classe astratte (od interfacce) con tali operazioni

3. realizzare la classe generica usando le classi astratte definite al punto 2 per manipolare le componenti

```
abtract class node {
  node left,right;
  void setLeft(node n){left=n;}
  void setRight(node n){right=n;}
  node left(){return left;}
  node right(){return right;}
  abstract boolean equals(node n);
  abstract boolean lessThan(node n);}

interface tree {
  void insert(node n);
  void delete(node n);
  node search(node n);}

class genericTree implements tree {

node root;

void insert(node n){
 node current=root;
 node parent;
 while (current != null)
   {parent=current;
    if (n.equals(current)) return; // do nothing
    if (n.lessThan(current)) current=current.left();
    else current=current.right();
   }
 if (root == null) {root=n;return;}
 if (n.LessThan(parent)) parent.setLeft(n);
 else parent.setRight(n);} // insert
 ...
} // genericTree
```

```
class intNode extends node{
int i;

boolean equals(node n){
if (n instanceof intNode) return(i == ((intNode) n).i);
// else error!
}

boolean lessThan(node n){
if (n instanceof intNode) return(i < ((intNode) n).i);
// else error!
}

intNode(int i){this.i=i;}
} // intTree

node n1 = new intTree(3);
node n2 = new intTree(5);
node n3 = new intTree(1);
tree t = new genericTree();
t.insert(n1);
t.insert(n2);
t.insert(n3);
```

In Eiffel:

```
class genericTree[T -> node]
 ...
end

intTree:genericTree[intNode];
```

T corrisponde al tipo delle componenti (è un parametro della classe) e deve essere un sotto-tipo di *node*

Vantaggi e svantaggi dei due approcci:

classi generiche simulate: componenti eterogenee, typing dinamico

classi generiche come costrutto: componenti omogenee, typing statico