Introduzione a Java

4

Giovanni Pardini
pardinig@di.unipi.it

Dipartimento di Informatica
Università di Pisa

13 dicembre 2009

Sommario

Sommario

Iteratori

Un iteratore permette di iterare, *in modo astratto*, sugli elementi di una collezione

- non è necessario conoscere l'implementazione interna della struttura dati

L'interfaccia `Iterator` è utilizzata per rappresentare gli iteratori:

```
1  public interface Iterator {
2      // ritorna true se ci sono altri elementi da generare
3      boolean hasNext();
4
5      // restituisce l'elemento successivo
6      Object next();
7
8      // rimuove l'elemento attuale
9      void remove();    //opzionale
10 }
```

Iteratori: utilizzo

L'interfaccia `Collection`, supertipo di tutte le strutture dati standard di Java che implementano collezioni di oggetti , fornisce un metodo per ottenere un iteratore:

```
Iterator iterator()
```

Esempio

```
1  ArrayList l = new ArrayList();
2  l.add("rosso");
3  l.add("blu");
4  l.add("giallo");
5  l.remove(1);
6
7  Iterator it = l.iterator(); // ArrayList implementa Collection
8  while (it.hasNext()) {
9    String s = (String)it.next();
10   System.out.println(s);
11 }
```

- L'esempio stampa le stringhe *rosso* e *giallo*

Sommario

Eccezioni

La gestione delle eccezioni in Java permette di:

- gestire gli errori separatamente dal codice che può generarli
- gestire gli errori dove è più opportuno farlo
  - invece che direttamente dove l'errore può essere generato

- Una eccezione, che viene lanciata *(sollevata) (thrown)* in caso di errore, blocca la normale esecuzione e causa un trasferimento del flusso di controllo
  - dal punto in cui è avvenuta l'eccezione
  - ad un gestore dell'eccezione *(exception handler)*, in cui viene catturata *(caught)*

- Solo oggetti appartenenti a sottoclassi di `Throwable` possono essere utilizzati come eccezioni (lanciati/catturati)
  - si usano classi diverse per indicare diversi tipi di errore

Lancio e cattura delle eccezioni

Una eccezione può essere lanciata:

- dalla macchina virtuale, per segnalare un errore
- esplicitamente dal programmatore (comando `throw`)

All'interno dei metodi, è possibile definire dei blocchi `try/catch` per:

- catturare le eccezioni che possono essere sollevate dai comandi contenuti nel blocco
- definire i gestori per queste eccezioni

- Se una eccezione non viene catturata nel metodo, essa si propaga dal chiamato al chiamante *(stack unwinding)*, fino a che non viene trovato un opportuno gestore
  - la chiamata del metodo viene terminata, rimuovendo il record dallo stack dei record di attivazione
  - se l'eccezione risale oltre il metodo `main`, il programma viene terminato

Lancio delle eccezioni: sintassi

Per **lanciare** esplicitamente una eccezione si usa un comando

```
throw <oggetto eccezione>
```

dove `<oggetto eccezione>` è un oggetto di una sottoclasse di `Throwable`

Esempio

```
throw new NullPointerException();
```

Esempio

```
throw new NullPointerException("p is null");
```

- Viene creata una eccezione con un messaggio di descrizione

Cattura delle eccezioni: sintassi

Per **catturare** le eccezioni e **definirne** i gestori si usa un blocco:

```
try {
  // blocco controllato
  ...

} catch (<Eccezione1> <nomeVariabile1>) {
  // gestore Eccezione1
  ...
} catch (<Eccezione2> <nomeVariabile1>) {
  // gestore Eccezione2
  ...
}
```

dove `<Eccezione i>` è il nome di una eccezione (classe)

- Quando viene lanciata una eccezione nel blocco controllato:
  - i gestori delle eccezioni sono esaminati secondo l'ordine di definizione
  - un gestore dichiarato con una eccezione E può catturare tutte le eccezioni di tipo F, con F = E, o F sottotipo di E

Esempio: NullPointerException (1)

```
1  public class Test {
2    public static void main(String[] args) {
3
4      boolean ok = false;
5      try {
6          Object obj = null;
7          String s = obj.toString(); // NullPointerException
8          ok = true;
9      } catch (NullPointerException e) { // gestore eccezione
10         System.out.println("Eccezione!");
11     }
12     System.out.println(ok);  // false
13   }
14 }
```

- Il metodo d'istanza `toString` non può essere invocato, dato che la variabile `obj` è `null`
  - viene lanciata una `NullPointerException`, che descrive l'errore
  - il flusso di controllo viene trasferito al gestore dell'eccezione

Esempio: NullPointerException (2)

```
1  public class Shape {
2
3    private Point position;
4    private boolean visible;
5
6    public Shape(Point p, boolean vis) {
7      if (p == null)
8        throw new NullPointerException("p is null");
9      position = p;
10     visible = vis;
11   }
```

- Se il parametro p è null viene lanciata esplicitamente una eccezione NullPointerException, che descrive l'errore

Esempio: NullPointerException (3)

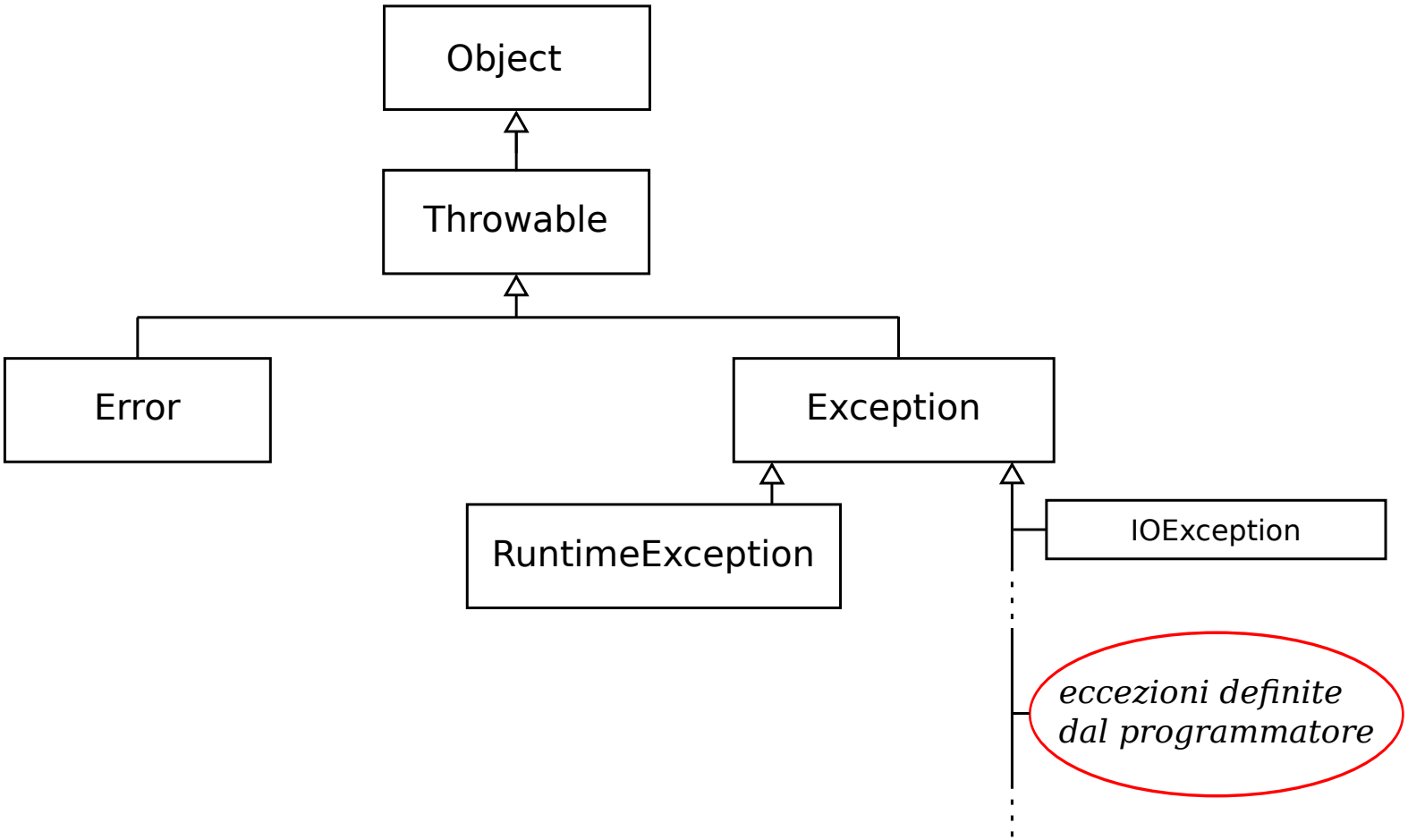
```
1 public class Shape {
2   ...
3   public Shape(Point p, boolean vis) {
4     if (p == null)
5       throw new NullPointerException("p is null");
6     position = p;
7     visible = vis;
8   }
```

```
1 public class TestException {
2   public static void main(String[] args) {
3     try {
4       Shape2 s = new Shape2(null, true);
5     } catch (NullPointerException ex) {
6       System.out.println("Eccezione: " + ex.getMessage());
7     }
8   }
9 }
```

- L'eccezione si propaga dal chiamato (costruttore) al chiamante (main)

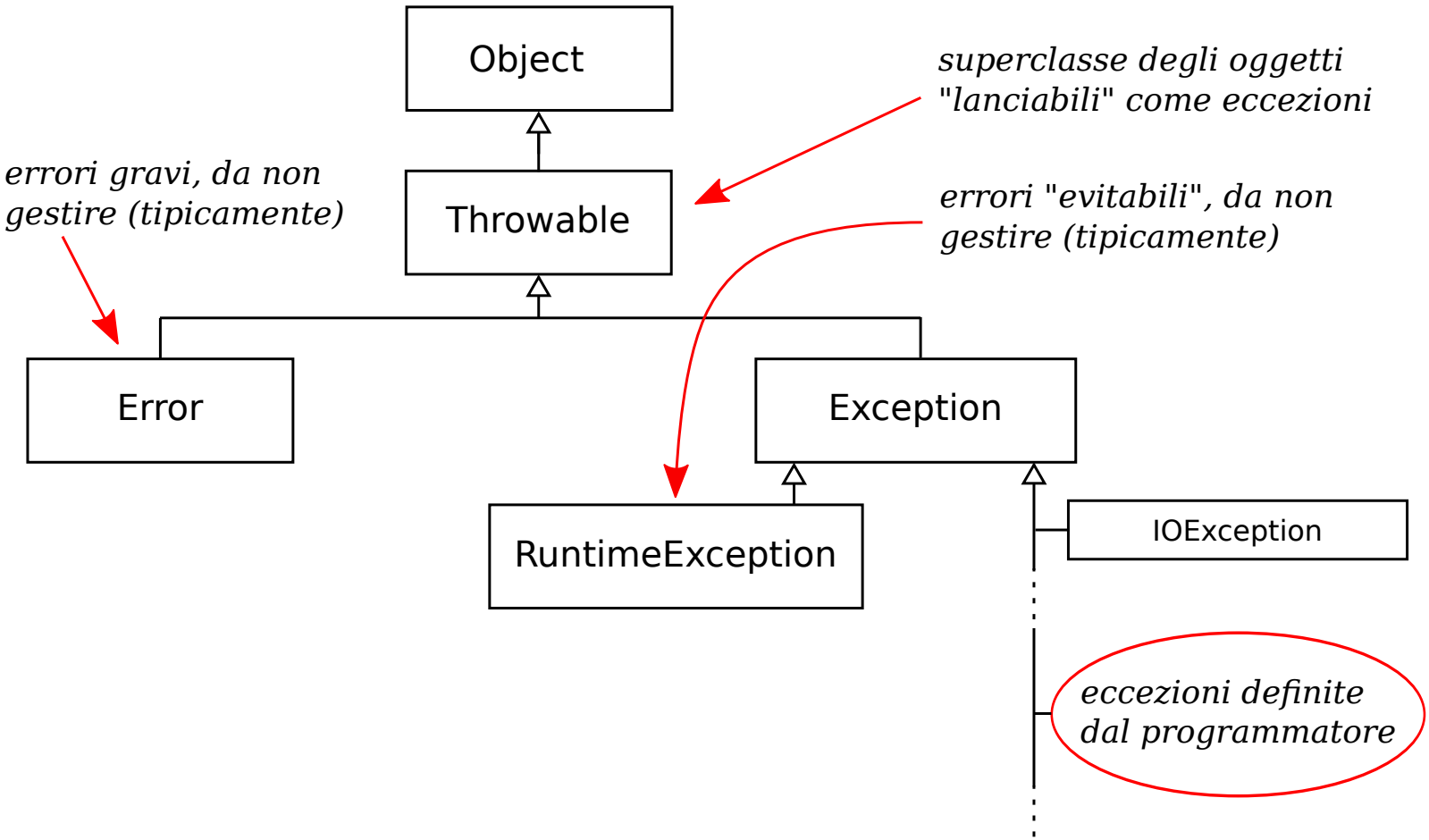
Gerarchia di eccezioni

- Si usano classi diverse per diversi tipi di errore



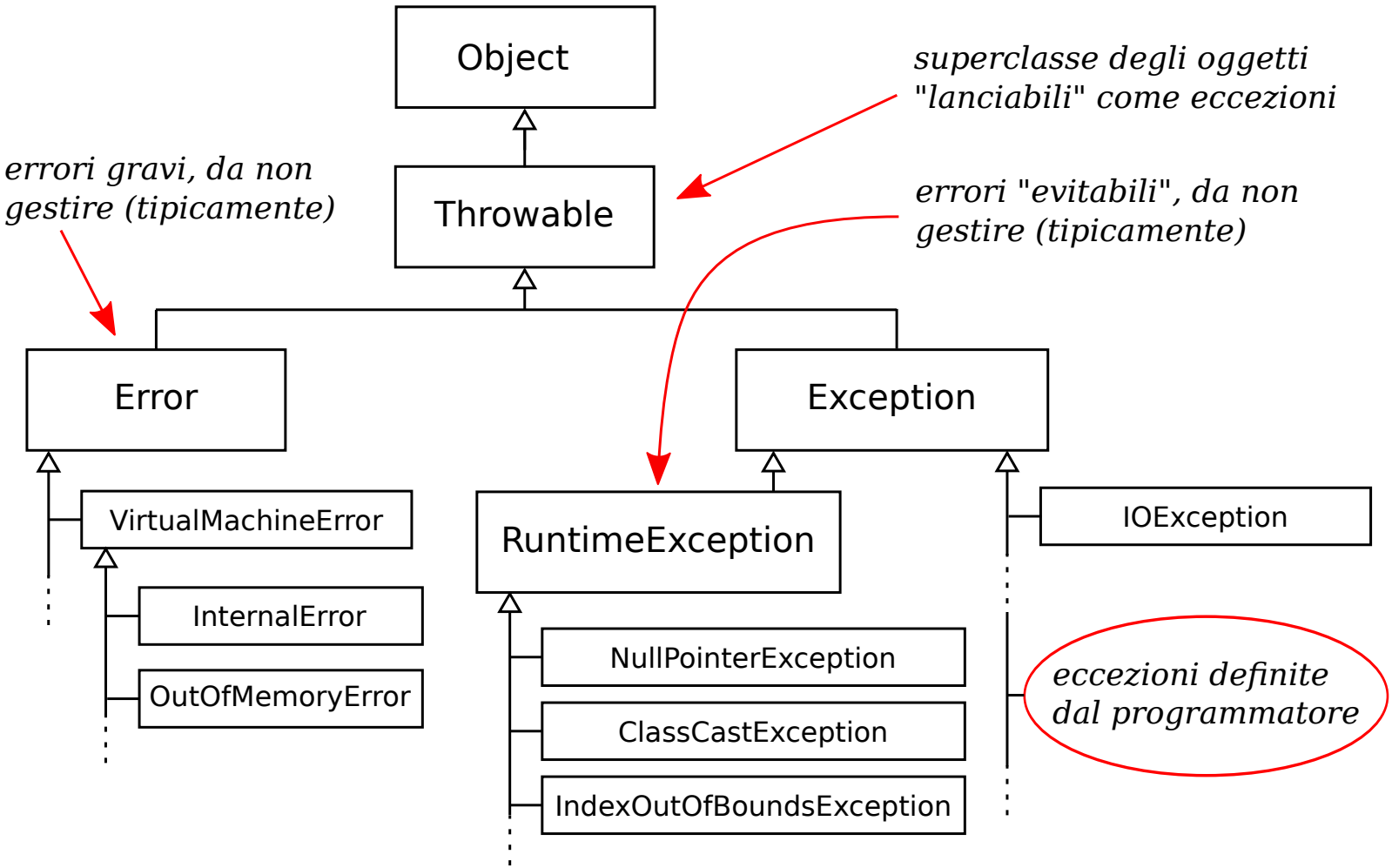
Gerarchia di eccezioni

- Si usano classi diverse per diversi tipi di errore



Gerarchia di eccezioni

- Si usano classi diverse per diversi tipi di errore



Tipi di eccezioni

Java ditingue tra i seguenti tipi di eccezioni:

- sottoclassi di **`Error`**: errori gravi che si suppone l'applicazione non debba mai catturare
  - es.: `NoClassDefFoundError`, `OutOfMemoryError`, `StackOverflowError`
- sottoclassi di **`RuntimeException`**: rappresentano tipicamente errori del programmatore, quindi evitabili con opportuni controlli
  - es.: `NullPointerException`, `ClassCastException`, `ArithmeticException`
- altre sottoclassi di **`Exception`**: rappresentano gli errori che il programma può incontrare durante il normale funzionamento
  - es.: `IOException`, `FileNotFoundException`
  - le eccezioni definite dal programmatore
  - a differenza delle altre, non sono **mai** lanciate autonomamente dalla virtual machine

Eccezioni *checked/unchecked*

In Java, i tipi eccezione sono distinti tra:

- eccezioni **unchecked**
  - classi `Error`, `RuntimeException` e loro sottoclassi
- eccezioni **checked**
  - tutte le altre classi eccezione
  - in particolare, le sottoclassi di `Exception` (eccetto `RuntimeException`)

- Le eccezioni checked devono essere gestite esplicitamente nel programma

Eccezioni checked (1)

Le eccezioni checked devono essere gestite esplicitamente nel programma

La clausola `throws` è utilizzata per dichiarare le eccezioni (checked) che un metodo/costruttore può lanciare

Dichiarazione di metodo

```
<tipo ritorno> <nomeMetodo> ( <parametri> )
                        throws <Eccezione1>, ..., <Eccezione N>
```

Costruttore

```
<NomeClasse> ( <parametri> )
                        throws <Eccezione1>, ..., <Eccezione N>
```

Eccezioni checked (2)

Se all'interno di un metodo/costruttore possono essere lanciate eccezioni checked, cioè:

- è presente un comando `throw`, oppure
- viene invocato un metodo/costruttore che dichiara eccezioni checked

allora tali eccezioni devono essere gestite esplicitamente:

- racchiudendo i comandi coinvolti in un blocco `try/catch`,
- oppure dichiarando che il metodo può lanciare quelle eccezioni (`throws`)

- Se una eccezione checked può propagarsi al chiamante del metodo deve essere dichiarata con `throws`
  - controllo del compilatore

Esempio: stack

Implementiamo uno stack di `Object`, con operazioni:

- `void push(Object obj)`
- `Object pop()`
- `boolean isEmpty()`

- Il tentativo di invocare il metodo `pop()` su uno stack vuoto deve lanciare l'eccezione `EmptyException`

Definizione di una eccezione

```
1  public class EmptyException extends Exception {
2
3    public EmptyException() {
4      super();
5    }
6
7    public EmptyException(String message) {
8      super(message);
9    }
10
11   public EmptyException(Throwable cause) {
12     super(cause);
13   }
14
15   public EmptyException(String message, Throwable cause) {
16     super(message, cause);
17   }
18 }
```

- L'eccezione è checked, in quando deriva direttamente da Exception

Stack: implementazione

```
1  public class MyStack {
2    private ArrayList list; // implementazione privata
3
4    public MyStack() {      // costruttore
5      list = new ArrayList();
6    }
7
8    public boolean isEmpty() {
9      return list.size() == 0;
10   }
11
12   public void push(Object obj) throws NullPointerException {
13     if (obj == null)
14       throw new NullPointerException("obj");
15     list.add(obj);
16   }
17   public Object pop() throws EmptyException {
18     if (isEmpty())
19       throw new EmptyException();
20     return list.remove(list.size() - 1);
21   }
22 }
```

Stack: eccezioni checked (1)

Nel metodo `pop()`, omettere di dichiarare l'eccezione `EmptyException` come lanciabile (`throws`) sarebbe stato un errore

```
1 public class MyStack {
2   ...
3
4   public Object pop() {
5     if (isEmpty())
6       throw new EmptyException()); // eccezione non gestita
7     return list.remove(list.size() - 1);
8   }
9 }
```

Stack: utilizzo (1)

Anche utilizzando la classe Stack è neccessario gestire le eccezioni

```
1  public class StackTest {
2
3    public static void main(String[] args) {
4      Stack st = new MyStack();
5      provaStack(st);
6    }
7
8    private static void provaStack(Stack st) {
9      st.push("Rosso");
10     st.push("Verde");
11     System.out.println(st.pop()); // eccezione non gestita
12   }
13 }
```

Stack: utilizzo (2)

Soluzione 1: racchiudiamo il comando `pop()` in un `try/catch`

```
1  public class StackTest {
2
3    public static void main(String[] args) {
4      Stack st = new MyStack();
5      provaStack(st);
6    }
7
8    private static void provaStack(Stack st) {
9      try {
10       st.push("Rosso");
11       st.push("Verde");
12       System.out.println(st.pop());
13     } catch (EmptyException e) {
14       System.out.println("Stack is empty!");
15     }
16   }
17 }
```

Stack: utilizzo (3)

Soluzione 2: dichiariamo l'eccezione `EmptyException` come lanciabile dal metodo

```
1  public class StackTest {
2
3    public static void main(String[] args) {
4      Stack st = new MyStack();
5
6      try {
7        provaStack(st);
8      } catch (EmptyException e) {
9        System.out.println("Stack is empty!");
10     }
11   }
12
13   private static void provaStack(Stack st)
14                                       throws EmptyException {
15     st.push("Rosso");
16     st.push("Verde");
17     System.out.println(st.pop());
18   }
19 }
```

Clausola `finally`

La clausola `finally` può essere aggiunta ad un blocco `try` per indicare un blocco di codice da eseguire sempre (anche in caso di eccezione) all'uscita dal blocco `try`

- anche se si esce dal blocco tramite `return`, `break`, o `continue`

```
try {
  // blocco controllato
  ...

} catch (<Eccezione1> <nomeVariabile1>) {
  // gestore Eccezione1
  ...
} finally {
  // blocco di chiusura
  ...
}
```

- Il blocco `finally` si usa per garantire che le risorse acquisite vengano rilasciate *sempre*, anche in caso di errore

Clausola `finally`: esempio

```
1  public class FinallyTest {
2    public static void main(String[] args) {
3      String s = "a";
4      try {
5        s += "b";
6        if (true) throw new EmptyException();
7        s += "c";
8
9      } catch (Exception e) {
10       System.out.println("Eccezione!");
11       s += "d";
12
13     } finally {
14       s += "e";
15       System.out.println(s);  // abde
16     }
17   }
18 }
```

Sommario

Reflection

La reflection indica l'abilità di un programma di accedere, a run-time, alle proprie strutture interne

La classe `Class` è usata per rappresentare le *classi*

- a run-time, ogni istanza di `Class` rappresenta una classe

È possibile ottenere un oggetto `Class` in due modi:

- invocando il metodo d'istanza `getClass()` su un oggetto
- accedendo alla costante statica `class` di una classe

- Le istanze di `Class` sono create automaticamente dalla virtual machine
  - c'è sempre una sola istanza di `Class` per ogni classe caricata

Reflection: esempi

Esempio

```
1 Point p = new Point(3,4);
2 Class c = p.getClass();
3 System.out.println(c.getCanonicalName()); // "Point"
4 System.out.println(c.isPrimitive());      // false
5 System.out.println(c.isArray());          // false
```

Esempio

```
1 Class c1 = Stack.class;
2 System.out.println(c1.getCanonicalName()); // "Stack"
3 System.out.println(c1.isPrimitive());      // false
4 System.out.println(c1.isInterface());      // true
```

Esempio

```
1 Object o = new Integer(3);
2 boolean b = (o.getClass() == Integer.class); // true
```

Sommario

Tipi generici

I generics *(tipi generici)* di Java realizzano una forma di polimorfismo parametrico

- permettono di definire *tipi* e *metodi* generici *(parametrici)*

Esempio: albero binario in ML/Java

```
type 'a btree = Empty
              | Node of 'a * 'a btree * 'a btree
```

```
1 public class Node {
2   Node left;
3   Node right;
4   Object value;
5 }
```

- In Java, senza il polimorfismo parametrico, sarebbe necessario usare il tipo `Object`

Classe `ArrayList`

La classe `ArrayList` rappresenta una lista, di dimensione variabile, *di oggetti dello stesso tipo*

- in pratica, contiene elementi di tipo `Object`

I metodi di inserimento e rimozione degli elementi forniti dalla classe utilizzano la classe `Object` per realizzare il polimorfismo

```
void add(Object obj)

Object get(int index)

Object remove(int index)
```

- Sta al programmatore garantire che la lista contenga sempre elementi dello stesso tipo effettivo

Esempio (1)

Esempio

```
1 ArrayList list = new ArrayList();
2 list.add(new Point(1,1));
3 list.add(new Point(2,2));
4 list.add(new Point(3,3));
5 Point p = (Point)list.get(1);  // cast necessario
```

- È necessario un cast da `Object` a `Point`

Esempio (2)

Nella lista è possibile inserire elementi di diverso tipo

- nessun problema per il compilatore, fintanto che gli oggetti appartengono a sottotipi di `Object`

Esempio

```
1 ArrayList list = new ArrayList();
2 list.add(new Point(1,1));
3 list.add(new Integer(5)); // ok
4 list.add(new Point(3,3));
5 Point p = (Point)list.get(1);  // ClassCastException
```

- Il cast fallisce a run-time

Classe `ArrayList` *generica*

I tipi generici permettono di evitare il problema

- il compilatore controlla che gli elementi inseriti siano di tipo corretto (dichiarato)
- non serve più il cast per rimuovere gli elementi, dato che il compilatore garantisce che gli oggetti abbiano lo stesso tipo

parametrico

Esempio

Il tipo `ArrayList<T>` è la versione generica della classe `Arraylist`

- per istanziare un tipo generico si deve specificare il tipo concreto del parametro `T`

Esempio (1)

```
1 ArrayList<Point> list = new ArrayList<Point>();
2 list.add(new Point(1,1));
3 list.add(new Point(2,2));
4 list.add(new Point(3,3));
5 Point p = list.get(1);  // cast non necessario
```

- Non è più necessario il cast
  - gli elementi sono sempre `Point`

Esempio (2)

Esempio

```
1 ArrayList<Point> list = new ArrayList<Point>();
2 list.add(new Point(1,1));
3 list.add(new Integer(5)); // Integer non compatibile con Point
4 list.add(new Point(3,3));
5 Point p = list.get(1);
```

- Il compilatore non consente di invocare il metodo add con un oggetto di tipo `Integer`, dato che `Integer` non è sottotipo di `Point`
- Nella lista non è più possibile inserire elementi di diverso tipo

Tipi generici: sintassi

Un tipo (classe/interfaccia) generico è dichiarato con la seguente sintassi, dove i `Ti` sono i parametri formali del tipo

```
class NomeClasse<T1,...,Tn> {
  // definizione
}
```

All'interno della definizione del tipo, è possibile utilizzare i nomi `T1,...,Tn` al posto dei tipi reali, dove opportuno

- il tipo è parametrico rispetto al tipo dei parametri con cui verranno istanziati `T1,...,Tn`

Per istanziare un tipo generico `NomeClasse<T1,...,Tn>`, è necessario specificare i tipi concreti da utilizzare

```
NomeClasse<Tipo1,...,Tipo n>
```

- I tipi generici possono essere istanziati soltanto con tipi riferimento

Esempio: classe `Pair` (1)

La classe `Pair` rappresenta coppie di oggetti

- il tipo è polimorfo nei tipi dei due oggetti

```
1  public class Pair<T,U> {
2    private T first;
3    private U second;
4
5    public Pair(T first, U second) {
6      this.first = first;
7      this.second = second;
8    }
9
10   public T getFirst() {
11     return first;
12   }
13   public U getSecond() {
14     return second;
15   }
16 }
```

- I nomi `T`, `U` denotano i parametri formali del tipo

Esempio: classe Pair (2)

Esempio

```
1 Integer in = new Integer(4);
2
3 Pair<Integer,String> p;
4 p = new Pair<Integer,String>(in, "abc");
5 Integer i1 = p.getFirst();
6 String s = p.getSecond();
```

- Possiamo cambiare la definizione della classe `Point` in modo da usare la classe `Pair`

Esempio: classe Pair (3)

```
1  public class Point extends Pair<Double,Double> {
2
3    public Point(double x, double y) {
4      super(new Double(x), new Double(y));
5    }
6
7    public double getX() {
8      return getFirst().doubleValue();
9    }
10   public double getY() {
11     return getSecond().doubleValue();
12   }
13
14   public Point add(Point p1) {
15     return new Point(getX() + p1.getX(), getY() + p1.getY());
16   }
17
18   ...
```

- Le altri classi non vengono modificate, dato che il comportamento di Point è preservato

Esempio: Stack (1)

- Introduciamo una interfaccia generica `Stack`
- modifichiamo l'implementazione di `MyStack` per implementare l'interfaccia `Stack`

Esempio

```
1 public interface Stack<T> {
2
3   void push(T obj) throws NullPointerException;
4
5   T pop() throws EmptyException;
6
7   boolean isEmpty();
8 }
```

Esempio: Stack (2)

```
1  public class MyStack<T> implements Stack<T> {
2    private ArrayList<T> list; // implementazione privata
3
4    public MyStack() {  // costruttore
5      list = new ArrayList<T>();
6    }
7
8    public void push(T obj) throws NullPointerException {
9      if (obj == null)
10       throw new NullPointerException("obj");
11     list.add(obj);
12   }
13   public T pop() throws EmptyException {
14     if (isEmpty())
15       throw new EmptyException();
16     return list.remove(list.size() - 1);
17   }
18
19   public boolean isEmpty() {
20     return list.size() == 0;
21   }
22 }
```

Esempio: Stack (3)

Esempio

```
1 Stack<String> stack = new MyStack<String>();
2 stack.push("Rosso");
3 stack.push("Bianco");
4 String t = stack.pop();
```

Esempio

```
1 Stack<String> stack = new MyStack<String>();
2 stack.push("Rosso");
3 stack.push("Bianco");
4 stack.push(new Integer(3)); // errore di tipo
```

Tipi generici e sottotipi

Siano

- `B` un sottotipo di `A`
- `G<T>` un tipo papametrico

allora: `G<B>` **non** è un sottotipo di `G<A>`

- Altrimenti si potrebbero avere errori di tipo a run-time, *anche per operazioni che non dovrebbero causarli*

Esempio

```
1 Stack<String> s1 = new MyStack<String>();
2 Stack<Object> s2 = (Stack<Object>)s1; // errore di tipo, non
3                                       // permesso dal compilatore
4 s2.push(new Integer(8));
5 String x = s1.pop(); // si avrebbe errore a run-time
```

Wildcards

Gli wildcard types sono utilizzati per denotare tipi parametrici *in cui il tipo del parametro non è fissato*

Nella forma più semplice, un wildcard type è definito usando il simbolo '?' al posto del tipo concreto

```
TipoParametrico<?>
```

Esempio

```
Stack<?> s;
Pair<Integer,?> p;
List<?> l;
```

Wildcard: esempio

```
1 Rectangle r1 = new Rectangle(new Point(0,0), new Point(4,3));
2 Rectangle r2 = new Square(new Point(5,0), 2);
3
4 List<Rectangle> list = new ArrayList<Rectangle>();
5 list.add(r1);
6 List<?> l = list;
7
8 Object o = l.get(0);
9 Rectangle rec = (Rectangle)l.get(0); // serve un cast
```

- Il tipo degli elementi non è conosciuto dal compilatore

Bounded wildcards

In un wildcard type, è possibile indicare dei vincoli (sui tipi) per il tipo parametrico che non è specificato

```
TipoParametrico<? extends AltroTipo>  // upper bound
TipoParametrico<? super AltroTipo>    // lower bound
```

- Una qualsiasi istanza del tipo parametrico `TipoParametrico`, il cui parametro è sottotipo/supertipo di `AltroTipo`

Esempio

```
List<? extends Shape> l;
```

- Un qualsiasi tipo in cui il parametro estende `Shape`:
  - `List<Shape>`, `List<Circle>`, `List<Rectangle>`, `List<Square>`

Bounded wildcards: esempio (1)


```
1 Rectangle r1 = new Rectangle(new Point(0,0), new Point(4,3));
2 Rectangle r2 = new Square(new Point(5,0), 2);
3
4 List<Rectangle> list = new ArrayList<Rectangle>();
5 list.add(r1);
6 List<? extends Shape> l = list;
7
8 Shape o = l.get(0);  // nessun cast
```

- Il compilatore sa che il tipo degli elementi è sottotipo di `Shape`
- È necessario il bounded wildcard in quando `List<Rectangle>` **non** è sottotipo di `List<Shape>`

```
List<Shape> l = (List<Shape>)list;  // errore
```

Bounded wildcards: esempio (2)

```
1 Rectangle r1 = new Rectangle(new Point(0,0), new Point(4,3));
2 Rectangle r2 = new Square(new Point(5,0), 2);
3
4 List<Rectangle> list = new ArrayList<Rectangle>();
5 list.add(r1);
6 List<? extends Shape> l = list;
7
8 l.add(r2); // errore
```

- Il tipo reale potrebbe non essere un supertipo di `Rectangle`

Metodi generici

Un metodo generico è definito rispetto ad uno o più parametri di tipo

```
<T1,...,Tn> tipoRitorno nomeMetodo (parametri) {
    // corpo
    ...
}
```

- I parametri di tipo (`T1,...,Tn`) possono essere usati
  - nel tipo di ritorno
  - nei parametri
  - nel corpo del metodo

Esempio

Metodo che ritorna il primo elemento di una lista

```
1 private static <T> T first(List<T> list) {
2   return list.get(0);
3 }
```

- Il tipo di ritorno dipende dal tipo del parametro `list`

Metodi generici: esempio (1)


```
1  public class GenericMethodsTest1 {
2    public static void main(String[] args) {
3      Integer i1 = 3;
4      Integer i2 = 5;
5      List<Integer> list = new ArrayList<Integer>();
6      list.add(i1);
7      list.add(i2);
8
9      Integer primo = first(list); // ritorna un Integer
10   }
11
12   private static <T> T first(List<T> list) {
13     return list.get(0);
14   }
```

- La chiamata del metodo `first`, con un parametro di tipo `List<Integer>`, ritorna un oggetto di tipo `Integer`

Metodi generici: esempio (2)

- È possibile aggiungere vincoli al tipo parametrico del metodo

```
1  public class GenericMethodsTest2 {
2    public static void main(String[] args) {
3      Rectangle r1=new Rectangle(new Point(0,0),new Point(4,3));
4      Rectangle r2 = new Square(new Point(5,0), 2);
5      List<Rectangle> list = new ArrayList<Rectangle>();
6
7      Rectangle rec = first(list); // ritorna un Rectangle
8    }
9
10   private static <T extends Shape> T first(List<T> list) {
11     return list.get(0);
12   }
13 }
```

- In questo caso, il metodo è invocabile solo con un parametro in cui T è sottotipo di Shape

Type erasure

L'informazione sui tipi parametrici è conosciuta soltanto a tempo di compilazione

- ogni parametro di ogni tipo generico diventa di tipo `Object`
- il compilatore inserisce gli opportuni cast dove necessario
  - che ha run-time non possono *mai* fallire

- Se non tutte le regole sono rispettate dal programmatore, possono verificarsi errori di tipo a run-time

Type erasure: esempio (1)

Classe `Pair<T,U>`

```
1  public class Pair<T,U> {
2    private T first;
3    private U second;
4
5    public Pair(T first, U second) {
6      this.first = first;
7      this.second = second;
8    }
9
10   public T getFirst() {
11     return first;
12   }
13   public U getSecond() {
14     return second;
15   }
16 }
```

Type erasure: esempio (2)

La classe `Pair<T,U>` viene compilata come la seguente:

```
1  public class Pair {
2    private Object first;
3    private Object second;
4
5    public Pair(Object first, Object second) {
6      this.first = first;
7      this.second = second;
8    }
9
10   public Object getFirst() {
11     return first;
12   }
13   public Object getSecond() {
14     return second;
15   }
16 }
```

- Tutti i parametri di tipo sono diventati `Object`

Type erasure: esempio (3)


```
1 Integer in = new Integer(4);
2
3 Pair<Integer,String> p;
4 p = new Pair<Integer,String>(in, "abc");
5 Integer i1 = p.getFirst();
6 String s = p.getSecond();
```

Il frammento di codice precedente, viene compilato come:

```
1 Integer in = new Integer(4);
2
3 Pair p;
4 p = new Pair(in, "abc");
5 Integer i1 = (Integer)p.getFirst(); // cast
6 String s = (String)p.getSecond();   // cast
```

- Il compilatore ha inserito i cast dove necessario

Type erasure: esempio (4)

```
1 ArrayList<Integer> list1 = new ArrayList<Integer>();
2 ArrayList<String> list2 = new ArrayList<String>();
3
4 boolean b = (list1.getClass() == list2.getClass());  // true
5 String nc = list1.getClass().getCanonicalName());
6                                                  // java.util.ArrayList
```

- Il tipo effettivo a run-time è uno solo: `ArrayList`

Sommario I