Programmazione 2

Quarta esercitazione

05/04/2024

Obiettivi dell’esercitazione

1. Il metodo `equals()` e il suo override
2. Variabili di classe (*o static*)
3. Metodi `static`
4. Eccezioni

1. Il metodo `equals()` e il suo override

Avevamo visto che la classe Object (la superclasse universale) ha dei metodi generici molto utili che sarebbe sempre opportuno sovrascrivere:

`String toString()`	Restituisce una stringa che restituisce la descrizione dell'oggetto
`Boolean equals(Object otherObject)`	Verifica se l'oggetto è uguale a un altro
`Object clone()`	Crea una copia completa dell'oggetto

`toString()` lo abbiamo già visto…

oggi vediamo come sovrascrivere `equals()`

1. Il metodo `equals()` e il suo override

La relazione di equivalenza deve rispettare ovvie proprietà:

- riflessività: `x.equals(x)`
- simmetria: `x.equals(y)` ⇒ `y.equals(x)`
- transitività: `x.equals(y) && y.equals(z)` ⇒ `x.equals(z)`
- `x.equals(null) == false`

L'implementazione di `equals()` nella classe `Object` confronta gli indirizzi di memoria degli oggetti. Per cui restituisce **true** solo se i due oggetti sono effettivamente la stessa istanza.

1. Il metodo `equals()` e il suo override

Cosa succede?

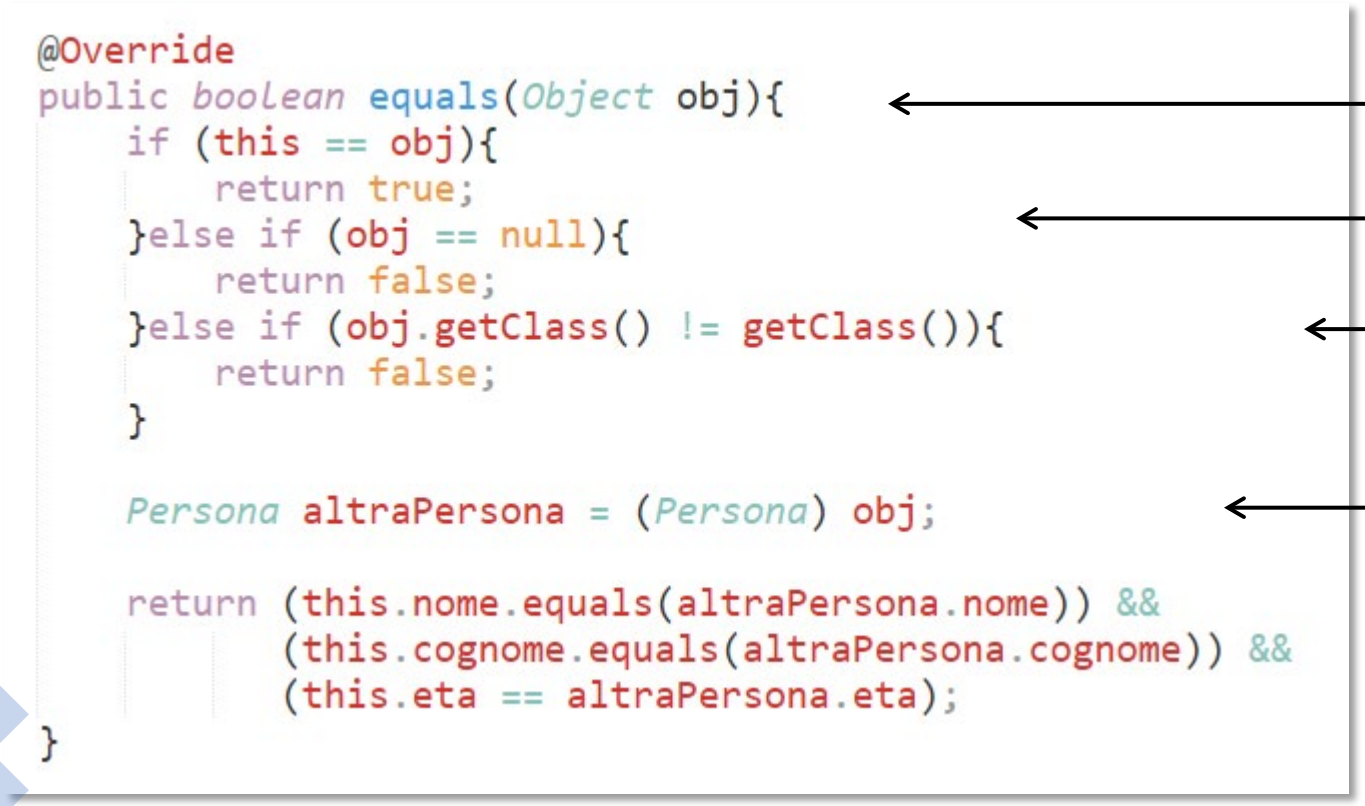
```
Persona p = new Persona();

assert p.equals(p);

assert !p.equals(new Persona());
```

Qui?

E qui?

1. Il metodo `equals()` e il suo override

Sovrascriviamo il metodo `equals()` ovvero facciamo il suo override:



```java
@Override
public boolean equals(Object obj){
    if (this == obj){
        return true;
    }else if (obj == null){
        return false;
    }else if (obj.getClass() != getClass()){
        return false;
    }

    Persona altraPersona = (Persona) obj;

    return (this.nome.equals(altraPersona.nome)) &&
           (this.cognome.equals(altraPersona.cognome)) &&
           (this.eta == altraPersona.eta);
}
```

Controllo subito che non si tratti della stessa istanza.

Controllo che non sia **null**.

Controllo che facciano parte della stessa classe (*).

Faccio un casting in modo da aver accesso alle variabili di esemplare.

Confronto i valori di tutti i parametri.

(*) Potremo usate l'operatore `instanceof`, ma tale verifica non è abbastanza specifica, perché darebbe esito positivo anche se `obj` appartenesse a una sottoclasse di `Persona`.

1. Il metodo `equals()` e il suo override

Quando sovrascriviamo il metodo `equals` in una sottoclasse, ricordarsi sempre di invocare il metodo `equals` della superclasse!

```
@Override
public boolean equals(Object obj){
    if (
        ...

    }else if(!super.equals(obj)){
        return false;
    }

    ...
```

Lo si può mettere in coda alla prima catena di `if else`.

In questo modo vengono verificate tutte le variabili di esemplare relative alla superclasse e ci resta solo da verificare quelle della sottoclasse che stiamo implementando.

2. Le variabili di classe (*o statiche*)

A volte occorre memorizzare valori che appartengono più correttamente a una classe che a un suo oggetto, in questi casi possiamo usare le *variabili statiche* o *variabili di classe*.

Sono variabili che non appartengono all'oggetto istanziato, bensì all'intera classe.

```
private static int contatore = 1000;
```

3. Metodi `static`

Talvolta servono metodi che non vengono invocati con un oggetto come parametro implicito: metodi di questo tipo vengono detti *metodi di classe* o *metodi statici*.

Un esempio tipico:

```
Math.sqrt(x)
```

Math è il nome di una classe, non di un oggetto!

Quando è utile implementare un metodo statico?

Quando ci servono metodi che non operano su uno specifico oggetto ma possono essere di supporto nel contesto in cui stiamo programmando.

3. Metodi `static`

Ad esempio:

Ipotizziamo di avere la classe BankAccount che rappresenta un conto bancario e ci serve un generico metodo che, dato un importo in *dollari*, esegua la conversione in *euro*.

È un metodo che opera in qualche modo su un oggetto di tipo BankAccount?

No.

In questo caso ha senso implementare il metodo come **statico**.

```
public static double converti(double dollari){
  return dollari * this.tasso;
}
```

3. Metodi `static`

Altre considerazioni:

- Perché il metodo `main` è un metodo statico?

Nel momento in cui il programma viene eseguito, non esiste ancora alcun oggetto, pertanto il primo metodo del programma deve essere necessariamente un metodo statico.

- Se vi accorgete che state implementando molti metodi statici che usano molte variabili statiche, è probabile che non abbiate identificato le classi migliori per la soluzione del vostro problema secondo un approccio orientato ad oggetti.

3. Metodi `static`

Curiosità:

Perché le variabili di classe e i metodi di classe si chiamano «statici» ***visto che il significato della parola statico è*** «che sta fermo in un posto» ***ed ha poco a che vedere con il concetto di*** `static` ***in Java?***

La parola riservata `static` *è solo un'eredità del C++ che lo utilizza nello stesso contesto.*

I progettisti del C++ non vollero inventarsi una nuova parola riservata per indicare i ***metodi di classe****, quindi decisero di riciclare* `static`*, usata molto raramente per indicare delle variabili che rimangono in una posizione fissa durante più invocazioni successive di metodi (caratteristica non presente in Java).*

Fu possibile quindi riutilizzare questa parola riservata senza confondere il compilatore, mentre il fatto che potesse confondere gli esseri umani non sembrò un grosso problema.

Obiettivi dell’esercitazione

1. Il metodo `equals()` e il suo override
2. Variabili di classe (*o static*)
3. Metodi `static`
4. Eccezioni

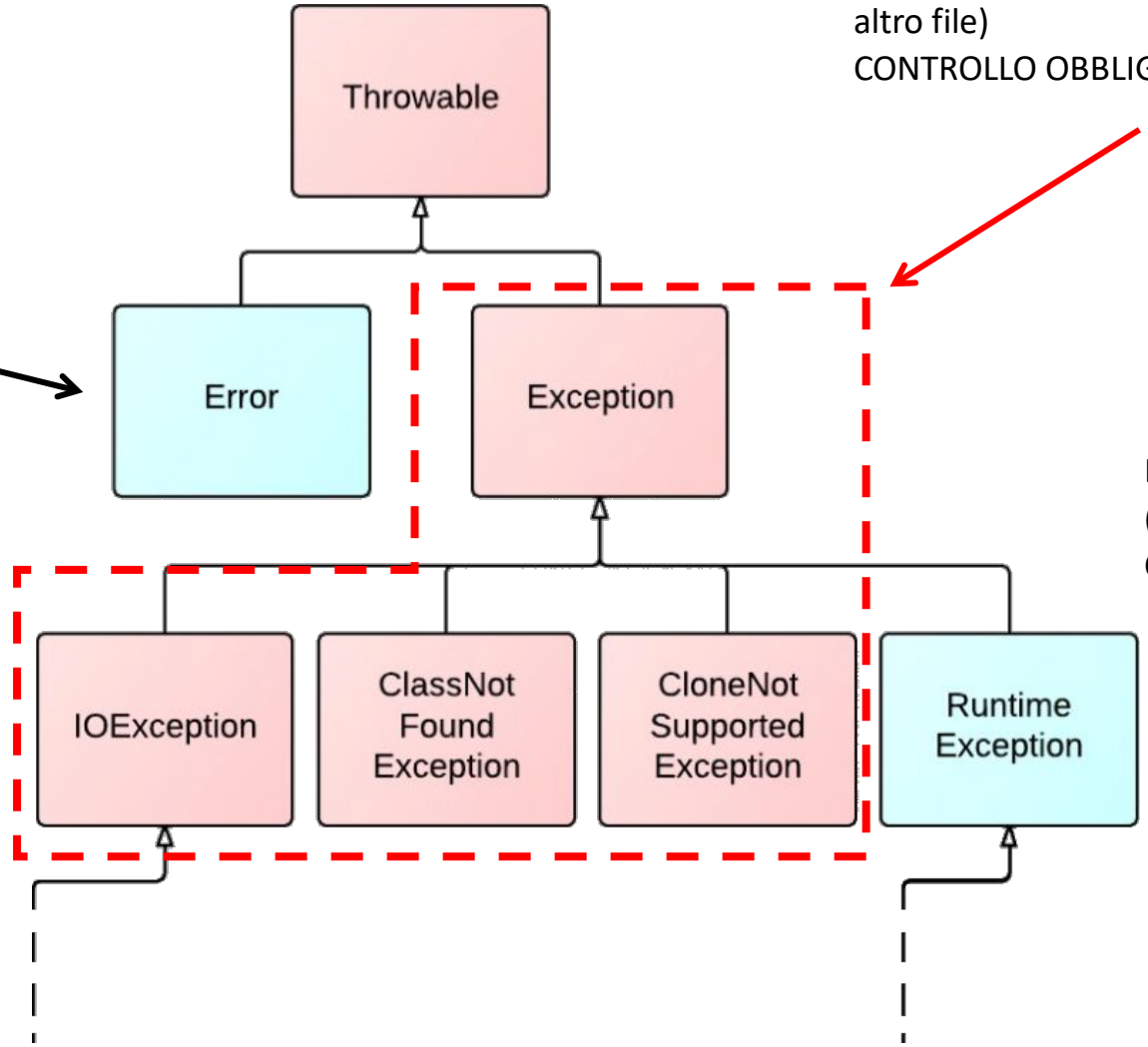
4. Eccezioni

- Un'eccezione è un evento che occorre durante l'esecuzione di un programma.
- Le eccezioni possono essere lanciate (**throw**) dai metodi e si propagano nei metodi chiamanti.
- È possibile catturare (**catch**) un'eccezione e fare qualcosa in risposta a questo evento (**exception handling**).

4. Eccezioni

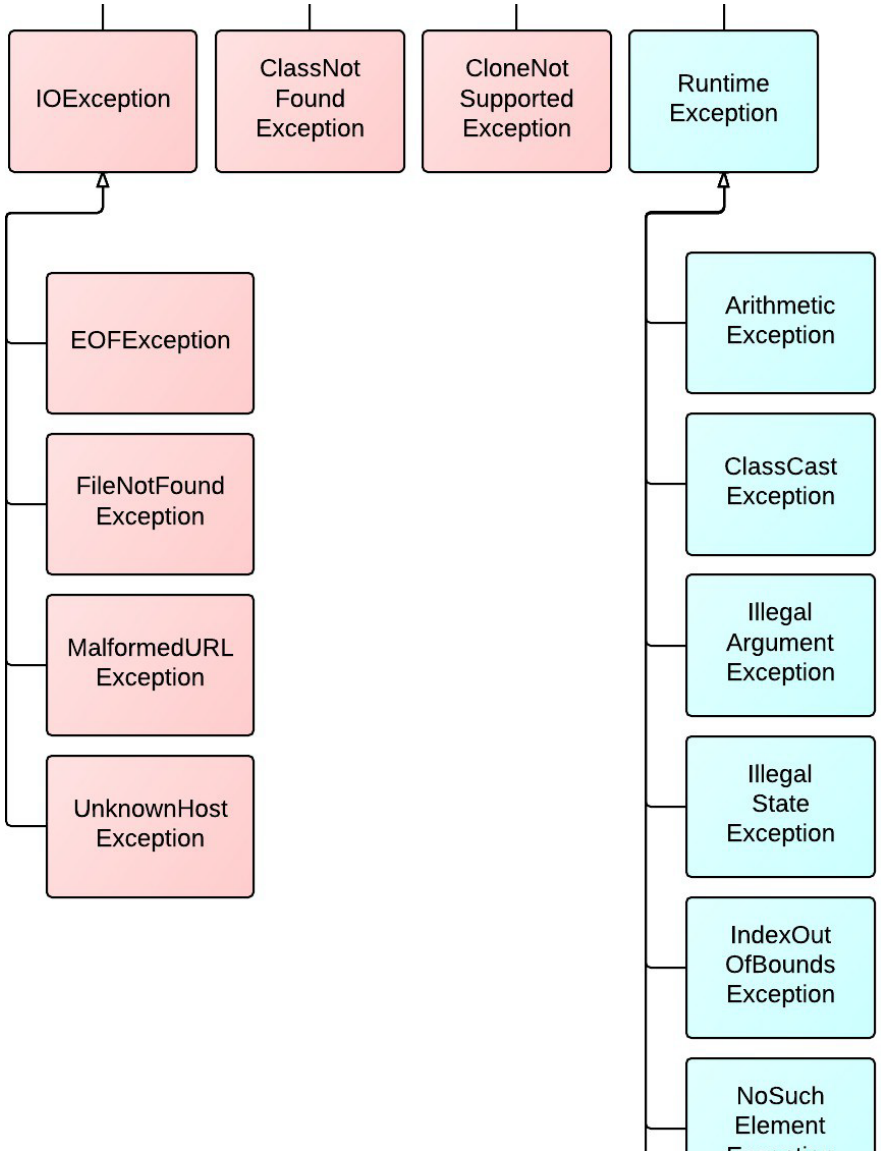
Situazioni più o meno prevedibili che il chiamante può risolvere (es. l'utente vuola aprire un file che non esiste, allora l'utente può scegliere di aprire un altro file)
CONTROLLO OBBLIGATORIO (**checked**)

Errori fatali che accadono di rado e che non ricadono nel controllo del programmatore (es. è finita la RAM)
CONTROLLO NON OBBLIGATORIO



Problemi che dipendono dal programmatore (es. il metodo ha ricevuto un oggetto null)
CONTROLLO NON OBBLIGATORIO (**unchecked**)

4. Eccezioni


IOException
ClassNot Found Exception
CloneNot Supported Exception
Runtime Exception
EOFException
FileNotFound Exception
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
Illegal Argument Exception
Illegal State Exception
IndexOut OfBounds Exception
NoSuch Element

4. Eccezioni

La clausola throw

```
public void preleva(double importo)
{
    if (importo > saldo)
    {
        throw new IllegalArgumentException("L'importo eccede il saldo");
    }
    saldo = saldo - importo;
}
```

← Questa istruzione non viene eseguita se viene lanciata l'eccezione.

In quale condizione mi trovo?
Non posso fare il prelievo perché l'importo inserito è troppo alto. Quindi è un problema di argomento e possiamo lanciare un eccezione di tipo `IllegalArgumentException`.

4. Eccezioni

Quando si invoca un metodo che può lanciare un eccezione controllata (checked) il compilatore verifica che non venga ignorata ovvero esige che sia dichiarato cosa fare in caso di lancio dell'eccezione.

Principalmente questo tipo di eccezioni riguarda la gestione di dati in ingresso e in uscita (file, flussi, ecc.).

Quando si invoca un metodo che può lanciare un eccezione non controllata (unchecked) il compilatore lascia la libertà di dichiarare o meno cosa fare in caso di lancio dell'eccezione.

4. Eccezioni

Ipotizziamo di voler leggere dati da un file. Usiamo quindi la classe Scanner.
Il suo costruttore può lanciare **`FileNotFoundException`**, che è un eccezione controllata, per cui siamo obbligati a gestirla in qualche modo.

```
String nomeFile = ...;
File inFile = new File(nomeFile);
Scanner in = new Scanner(inFile);
```

Ci troviamo davanti a **due** possibilità:

1) Terminare l'esecuzione del metodo che contiene l'istruzione che lancia l'eccezione e segnalare al metodo invocante che potrà ritrovarsi con tale eccezione che, a sua volta, dovrà gestire.

2) Gestire l'eccezione.

4. Eccezioni

Possibilità 1: non gestire l'eccezione e propagarla al chiamante.

```
public void read(String nomeFile) throws FileNotFoundException{
    File inFile = new File(nomeFile);
    Scanner in = new Scanner(inFile);
    ...
}
```

Dopo la clausola **`throws`** si specifica l'elenco dei tipi di eccezione che possono essere lanciate dal nostro metodo, separate da virgola.

Il chiamante del nostro metodo avrà le nostre medesime scelte.

4. Eccezioni

Possibilità 2: catturare e quindi gestire l'eccezione.

```
public void read(String nomeFile)
{
     try
     {
          File inFile = new File(nomeFile);
          Scanner in = new Scanner(inFile);
          String input = in.next();
          int value = Integer.parseInt(input);
          ...
     }
     catch(IOException exception)
     {
          exception.printStackTrace();
     }
     catch(NumberFormatException exception)
     {
          System.out.println("L'input non è un numero");
     }
}
```

Può lanciare `FileNotFoundException` (checked)

Può lanciare `NoSuchElementException` (unchecked)

Può lanciare `NumberFormatException` (unchecked)

Se qualcuna di queste eccezioni viene effettivamente lanciata, le rimanenti istruzioni del blocco `try` non vengono eseguite.

4. Eccezioni

La clausola `finally`

```
PrintWriter out = new PrintWriter(filename)
try
{
        writeData(out)
}
finally
{
        out.close();
}
```

Per essere visibile al blocco `finally`, `out` deve essere dichiarata fuori dal `try`.

Codice che può lanciare eccezioni.

Questo blocco di istruzioni viene sempre eseguito, anche in caso di eccezione.

4. Eccezioni

Consiglio: non usare `catch` e `finally` nel medesimo blocco `try`.

```
try
{
    PrintWriter out = new PrintWriter(filename)
    try
    {
        writeData(out)
    }
    finally
    {
        out.close();
    }
}
catch (IOException exception)
{
    Gestione dell'eccezione
}
```

Chiude le risorse

Gestisce l'eventuale eccezione

È di più semplice comprensione.

4. Eccezioni

E se le classi di eccezioni esistenti non mi bastano? Posso creare nuove classi estendendo quelle esistenti.

```
public class InsufficientFundsException extends RuntimeException
{
    public InsufficientFundsExceoption() {}

    public InsufficientFundsExceoption(String message)
    {
        super(message);
    }
}
```

Il messaggio `message` può essere recuperato col metodo `getMessage()` della classe `Throwable`

Fine quarta esercitazione