# Corso di
# “Sviluppo di applicazioni Web”

Docente: Lorenzo Gallucci

Lezione #1:

- Applicazioni Web in Java
- equals() e hashCode()
- reflection

# Applicazioni Web in Java

- La produzione di applicazioni Web in Java può avvalersi di un gran numero di framework
- Numerosi campi:
  - Interfacciamento con database relazionali (“object/relation mapping”)
  - Strutturazione dell’applicazione per mezzo di “container IoC”
  - Framework per lo sviluppo dell’interfaccia Web (es. Spring MVC, Struts, Tapestry, ecc.)
- In alcuni casi è possibile usare Java per costruire l’interfaccia, sebbene il linguaggio in cui essa viene eseguita sia differente.
  - Es. GWT (Google Web Toolkit), in cui l’applicazione Java viene convertita in JavaScript

# Applicazioni Web in Java

- Ogni framework ha le proprie particolarità, ma per avvalersene è sempre necessario:
  - usare in maniera adeguata i costrutti del linguaggio
  - saper modellare correttamente ad oggetti
- È anche utile conoscere i meccanismi interni del linguaggio, per immaginare le possibilità del framework

# equals() e hashCode()

- Nella classe Object sono definiti due metodi di importanza notevole per tutta la piattaforma:
  - equals(): confronta *this* con l'Object passato, restituisce *true* se sono "uguali" (non necessariamente lo stesso oggetto)
    - L'implementazione di default considera un oggetto uguale solamente a sé stesso
  - hashCode(): restituisce un intero atto a fungere da codice hash per *this*, sfruttando i valori
    - L'implementazione di default ignora i campi dell'oggetto e dà un codice hash legato unicamente all'identità dell'oggetto
- *Sono sufficienti le implementazioni di default?*

# equals() e hashCode()

- Generalmente, quando si definisce un oggetto, il criterio di uguaglianza non può limitarsi all'identità
  - Es. oggetto User con campo *name*: due oggetti User sono "uguali" (rappresentano lo stesso utente) se hanno il campo *name* uguale
- Si rende quindi necessario fornire un'implementazione differente del metodo equals() (*override*)

# equals() e hashCode()

```
import java.util.*;

public class Name {
    private String first, last;

    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }

    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name)o;
        return n.first.equals(first) && n.last.equals(last);
    }
}
```

# equals() e hashCode()

- Tuttavia, aver reimplementato *il solo* equals() non è sufficiente a rendere un oggetto un “buon cittadino”
- Infatti, come vedremo più avanti, equals() e hashCode() hanno semantiche strettamente collegate
- Aver definito l’uno e non l’altro genera un’asimmetria che può portare a comportamenti inattesi (anche se facili da diagnosticare)

# equals() e hashCode()

```
import java.util.*;

public class NameTest {
    public static void main(String[] args) {
        Set<Name> s = new HashSet<Name>();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
            s.contains(new Name("Mickey", "Mouse")));
    }
}
```

Sebbene i due oggetti siano equivalenti, il risultato è **false!!!!!**

# equals() e hashCode()

- HashSet, come altre classi delle librerie di sistema, si basa sull'assunzione che due oggetti uguali fra loro debbano avere il medesimo codice hash

- La struttura dati sottostante cerca un oggetto equivalente a quello passato al metodo contains() *tra quelli con medesimo codice hash*
  - Il metodo hashCode() di default, però, dà codici diversi per oggetti diversi, indipendentemente dal loro contenuto
  - Il primo Name creato diventa così un oggetto "fantasma", sebbene risponda correttamente al metodo equals()

# equals() e hashCode()

```
import java.util.*;

public class Name {
    private String first, last;

    public Name(String first, String last) {
        this.first = first; this.last = last;
    }

    public boolean equals(Name n) {
        return n.first.equals(first) && n.last.equals(last);
    }

    public int hashCode() {
        return 31 * first.hashCode() + last.hashCode();
    }
}
```

# equals() e hashCode()

- Anche la precedente definizione risulta scorretta, perché:
  - public int hashCode()  … è ok
  - ma: public boolean equals(Name n)  … NO!
- Cambiare il tipo del parametro quando si vuole ridefinire un metodo *fa sì che non lo si ridefinisca affatto!*
  - Il metodo originale, nella classe Object, è:
    - public boolean equals(Object o)
  - I due metodi equals(Object) e equals(Name) sono di fatto del tutto scollegati

# equals() e hashCode()

- Di primo acchitto, può sembrare che il problema abbia rilevanza solo se si usano le classi della libreria di sistema (HashMap, ecc.)
- *Non è così!*
- Durante lo sviluppo di un software, può capitare infatti di avvalersi di varie librerie di terze parti, a cui si debbono/vogliono passare propri oggetti
- È plausibile che tali librerie, nello svolgere il proprio compito, facciano uso delle librerie di sistema
- Il problema rischia dunque di ripresentarsi, ma in strati meno “visibili” del nostro metodo di test

# equals() e hashCode()

- Morale:
- È *sempre* necessario implementare *sia* equals() *che* hashCode() per:
  - oggetti che rappresentano valori, da impiegare in propri algoritmi
  - oggetti di qualsiasi tipo, passati a librerie di terze parti
- Tuttavia:
  - Non è sempre *chiaro* quale debba essere il criterio soggiacente
    - Il campo “codiceFiscale” è cruciale? O lo è la coppia “nome” e “cognome”?
  - È dunque utile riflettere sull’uso che si vuole fare dello specifico oggetto nel proprio software, *prima* di implementare una coppia equals()/hashCode()

# equals()

- La documentazione JavaDoc per Object.equals() recita:

- Indicates whether some other object is "equal to" this one.
- The equals method implements an equivalence relation:
  - It is **reflexive**: for any reference value x, x.equals(x) should return true.
  - It is **symmetric**: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
  - It is **transitive**: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is **consistent**: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.

# equals()

- (cont.)

- For any non-null reference value x, x.equals(null) should return false.
- The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any reference values x and y, this method returns true if and only if x and y refer to the same object (x==y has the value true).

  Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

# hashCode()

- La documentazione JavaDoc per Object.hashCode() recita:
- Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.
- The general contract of hashCode is:
  - Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
  - If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

# hashCode()

- (cont.)
  - It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.
- As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

# Reflection

- Diversi linguaggi hanno capacità di “introspezione”
  - sono in grado di rispondere, da programma, ad interrogazioni sulla struttura del programma stesso
  - es. RTTI di C++
- Tali capacità variano fortemente, a seconda del carattere del linguaggio stesso
  - minime quando il typing è deciso a tempo di compilazione (es. C++)
  - massime per linguaggi “dinamici” (es. Javascript)

# Reflection

- E … in Java?
  - I tipi sono largamente decisi a tempo di compilazione, ma ….
  - La compilazione non è un processo che si esaurisce con la chiamata di *javac* !
- In effetti, in Java il *codice* delle classi viene gestito al pari dei *dati*
  - Ad ogni classe è associato un oggetto *Class*
  - Il codice facente capo alla classe (metodi, inizializzatori, costruttori, ecc.) è visibile alla Java Virtual Machine, ma *anche* allo sviluppatore

# Reflection

- Un utente può esaminare la struttura di una classe e recuperare:
  - Nome, superclasse, interfacce implementate
  - Informazioni sui metodi e sui campi
- Le informazioni possono essere sfruttate per accedere ai valori dei campi o per chiamare *dinamicamente* metodi sugli oggetti
- Agganciandosi all'API *ClassLoader* è anche possibile recuperare, a runtime, il codice macchina ("bytecode") di ogni classe eseguita

# Reflection

- Esempio: Java Bean

- Fin dai primordi Java include delle librerie per:
  - manipolare oggetti non noti a tempo di compilazione
  - esaminarne le “proprietà” (concetto più generale del “campo”, generalmente implementato con metodi get/set)

  → JavaBean “Introspectors”

# Reflection

- È però possibile anche *inserire* nuove informazioni
  - Nuove classi possono essere create a runtime, esaminando quelle esistenti o partendo da zero
- Codice *automodificante*!
- Varie librerie consentono di sintetizzare classi in tal modo:
  - Proxy (fornito all'interno della JDK)
  - CGLIB (su SourceForge)
  - BCEL (progetto Apache)
  - ecc.

# Reflection

- È bene notare che il codice così generato è sottoposto allo stesso processo di compilazione dinamica del codice compilato con javac
- Nelle moderne JVM, infatti, un meccanismo chiamato “HotSpot” identifica i metodi o gruppi di metodi più chiamati dell’applicazione e li sottopone ad un processo di compilazione ottimizzata
  - Il sistema HotSpot non fa distinzione tra classi provenienti da un file .jar o sintetizzate