

# **Programmazione Orientata agli Oggetti**

---

Riflessione

# Sommario

- Introduzione
- `java.lang.Class<T>`
- Utilizzo della riflessione
- Unchecked Cast Warning
- Riflessione nello studio di caso
  - **FabbricaDiComandiRiflessiva**
- Equivalenza tra oggetti di una gerarchia
  - Esercizio

# Introduzione

- La riflessione (o *introspezione*) è una interessante caratteristica del linguaggio Java
- Permette di scrivere codice che introspettivamente analizza (e modifica) lo codice compilato (nello stesso linguaggio Java)
- Del resto i tool che utilizziamo, sono, in definitiva programmi che manipolano altro codice java
  - Spesso sono scritti essi stessi in Java
  - ✓ Esempi: Eclipse, JUnit

# La Classe `java.lang.Class<T>` (1)

- Per ogni tipo `T` definito da un programma Java esiste un oggetto che descrive il tipo stesso
  - ed in un certo senso, descrive anche il contenuto del corrispondente file `.class` contenente il codice oggetto
- Istanza della classe generica `java.lang.Class<T>`
  - Offre metodi che consentono di analizzare la classe
    - è possibile ottenere l'elenco dei metodi, l'elenco dei supertipi, ecc.
- ✓ Consultare la documentazione javadoc di `Class<T>`

# La Classe `java.lang.Class<T>` (2)

- La JVM fornisce forti garanzie su questi oggetti:
  - per ogni tipo `T` esiste un corrispondente ed *unico* esemplare istanza della classe `java.lang.Class<T>`
  - la creazione di tale oggetto avviene al momento del caricamento del file `.class`, ovvero quando «serve» per l'esecuzione, ad es. viene istanziata una istanza del tipo oppure viene acceduto un suo campo statico
  - ✓ in base alle specifiche del linguaggio, non tutti gli oggetti `Class<T>` devono essere necessariamente creati sin dall'inizio dell'esecuzione

# Riferimenti ad Oggetti `Class<T>`

- Tutti i tipi (comunque definiti: classi, interface, ...) possiedono una variabile statica pubblica chiamata **`class`** che memorizza un riferimento al corrispondente oggetto di tipo **`java.lang.Class`**
- Esempio:

```
Class<it.uniroma3.diadia.comandi.ComandoVai> classeDiVai =  
    it.uniroma3.diadia.comandi.ComandoVai.class;
```

- Tutte gli oggetti ereditano un metodo di istanza (polimorfo) di **`java.lang.Object`** che restituisce un riferimento all'oggetto associato alla classe del suo *tipo dinamico*
- Ad esempio:

```
Class<ComandoVai> classeDiVai = new ComandoVai().getClass();
```

# Interrogare Oggetti Class<T>

- Possiamo interrogare l'oggetto di tipo `Class<T>` per ottenere le proprietà del tipo `T`
- Es.: stampiamo l'elenco di tutti i metodi di un tipo

```
import java.lang.reflect.Method;
import java.util.Iterator;
public class EsempioRiflessione {
    public static void main(String[] args) {
        Class<Iterator> classeDiIterator = Iterator.class;
        for (Method m : classeDiIterator.getMethods())
            System.out.println(m);
    }
}
```

- Possibile esecuzione (con Java 7):

```
$ java EsempioRiflessione
public abstract boolean java.util.Iterator.hasNext()
public abstract java.lang.Object java.util.Iterator.next()
```

# Uso della Riflessione

- La riflessione viene applicata nello sviluppo di programmi con funzionalità complesse. Es.
  - per lo sviluppo di IDE
  - in alcuni framework, come JUnit
    - il suo *runner* riesce a trovare i metodi annotati con `@Test` utilizzando proprio l'introspezione
  - per la persistenza dei dati (ORM >>>)
- Tuttavia, la riflessione risulta particolarmente utile anche in contesti meno sofisticati:
  - offre la possibilità di creare oggetti a partire dal nome della classe memorizzata in un oggetto `String`
  - ✓ quindi in effetti decidendo il tipo dell'oggetto creato a tempo dinamico e *non* già a tempo statico!



# Creazione Introspettiva di Oggetti

- Esistono poche modalità per la creazione di oggetti e tutte comportano l'invocazione esplicita od implicita (cfr. letterali `String`, *Boxing*) di un costruttore con l'operatore `new`
- Ad esempio:

```
import it.uniroma3.diadia.comandi.*;
```

```
...
```

```
ComandoVai vai = new ComandoVai();
```

- Con la riflessione è possibile invece anche creare oggetti di tipo `T` invocando un metodo `newInstance()` offerto da un oggetto di tipo `Class<T>`, con questa segnatura:

```
T newInstance()
```

- Ad esempio:

```
Class<ComandoVai> classeDiVai = ComandoVai.class;
```

```
Comando vai = classeDiVai.newInstance();
```

# Caricamento a Tempo di Esecuzione di Oggetti Class

- Un'altra interessante possibilità è offerta dalla classe `Class<T>` tramite il metodo:

```
public static Class<?> forName(String className)
```

permette di caricare una classe fornendo il suo nome completamente qualificato (come `String`) e restituisce il corrispondente oggetto `Class<?>`

- Esempio:

```
warning: [unchecked] unchecked cast
Class<ComandoVai> classeDiVai = (Class<ComandoVai>)
Class.forName("it.uniroma3.diadia.comandi.ComandoVai");
```

```
Class<ComandoVai> classeDiVai = (Class<ComandoVai>)
```

```
Class.forName("it.uniroma3.diadia.comandi.ComandoVai");
```

- ✓ Il caricamento presuppone un vero accesso al contenuto di `ComandoVai.class`
  - possono sollevarsi `ClassNotFoundException`

# Creazione di Oggetti di Tipo Scelto a Tempo di Esecuzione

- Usando questi metodi possiamo creare oggetti a partire dal nome di una classe (deciso a tempo dinamico) contenuto in un oggetto **String**

- Esempio:

warning: unchecked cast

```
Class<ComandoVai> classeDiVai = (Class<ComandoVai>)
    Class.forName("it.uniroma3.diadia.comandi.ComandoVai");
Comando vai = (Comando)classeDiVai.newInstance();
```

- Il warning sul downcast serve ad avvertirci che il compilatore non è in grado di distinguerlo dal seguente codice che chiaramente solleva una **ClassCastException** durante l'esecuzione. Dove?...

```
Class<ComandoVai> classeDiVai = (Class<ComandoVai>)
    Class.forName("it.uniroma3.diadia.Gioco");
Comando vai = (Comando)classeDiVai.newInstance();
```

# Unchecked Cast Warning

- Quando si lavora con l'introspezione, dove la tipizzazione diventa più lasca, l'uso dei Java Generics mostra più frequentemente le limitazioni della loro tardiva (ma retrocompatibile) introduzione in Java (5)
- Il compilatore può emettere degli avvertimenti legati ad un uso troppo «permissivo» dei tipi e non è più in grado di fornire tutte le garanzie tipiche dei linguaggi fortemente e staticamente tipati (ovvero l'assenza di **ClassCastException** durante l'esecuzione)
  - ✓ «avvertimento» non «errore»: la compilazione rimane possibile
- Un *Unchecked Cast Warning* viene emesso dal compilatore per informare che l'uso dei tipi non è abbastanza stringente da permettergli di offrire garanzie a tempo statico sull'assenza di errori di tipo a tempo dinamico

# Unchecked Cast Warning (cont.)

- Il problema è esacerbato dalla decisione di lasciar convivere le versioni pre-generics delle classi con quelle propriamente generiche per la retrocompatibilità
  - ✓ `java.lang.Class` non generica esiste da sempre
  - L'implementazione dei Java Generics, in definitiva, si basa sull'ampliamento dei controlli sui tipi a tempo statico ed un meccanismo di trasformazione dei tipi generici verso quelli non generics, i soli disponibili a tempo dinamico (vedi *erasure* <<)

```
List lPreG = new ArrayList();
```

```
lPreG.add(new Integer(3));
```

```
ArrayList<String> listG = (ArrayList<String>) lPreG;
```

```
String s = listG.get(0);
```

**Tempo statico (compilazione):**  
*warning: unchecked cast*

**Tempo dinamico:** `ClassCastException`  
`java.lang.ClassCastException:`  
`Integer cannot be cast to String`

- Se il contesto non permette una tipizzazione forte (come purtroppo spesso accade con le API della riflessione) le garanzie offerte vengono rilassate (esattamente ai livelli considerati ordinari nell'uso del JCF pre-generics)

# Riflessione - Studio di Caso (1)

- Possiamo scrivere codice in cui vengono creati dinamicamente oggetti a partire dal nome della loro classe
- Un esempio nel nostro studio di caso: possiamo creare gli oggetti **Comando** derivando il nome della classe del sottotipo concreto da utilizzare a partire dall'istruzione digitata dall'utente
- Scriviamo una classe che implementi **FabbricaDiComandi** sfruttando la riflessione: riusciamo in maniera semplice ed elegante ad estirpare definitivamente una delle più “resistenti” «codice-fisarmonica» presenti sin dalla versione iniziale del codice

# Riflessione - Studio di Caso (2)

- L'idea è quella di costruire il nome della classe che processa un comando a partire dal nome del comando contenuto nell'istruzione digitata. Ad esempio:

`"vai" → "it.uniroma3.diadia.comandi.ComandoVai"`

- Quindi, si prevedono i seguenti passi:

(1) Lettura istruzione

`"vai sud"`

(2) Costruzione nome della classe

`"it.uniroma3.diadia.comandi.ComandoVai"`

(3) Caricamento dell'oggetto `Class` con `Class.forName()`

(4) Istanziamo un oggetto di tale classe con `newInstance()`

- Un piccolo refactoring preparatorio...

# FabbricaDiComandi

- Ridefiniamo l'interfaccia **FabbricaDiComandi** come segue (aggiungendo la clausola **throws** all'unico metodo):

```
public interface FabbricaDiComandi {  
    public Comando costruisciComando(String istruzione)  
        throws Exception; // (>> eccezioni)  
}
```

- Così non siamo costretti a gestire subito le diverse tipologie di eccezioni che possono venire sollevate
- ✓ Dopo la lezione sulle eccezioni (>>)
  - sarà possibile rivedere questa scelta che al momento operiamo per non allontanare troppo la discussione dall'introspezione
  - per le stesse motivazioni, trascuriamo anche la gestione dei comandi non validi



# FabbricaDiComandiRiflessiva

```
public class FabbricaDiComandiRiflessiva implements FabbricaDiComandi {  
    public Comando costruisciComando(String istruzione) throws Exception {  
        Scanner scannerDiParole = new Scanner(istruzione); // es. 'vai sud'  
        String nomeComando = null; // es. 'vai'  
        String parametro = null; // es. 'sud'  
        Comando comando = null;  
  
        if (scannerDiParole.hasNext())  
            nomeComando = scannerDiParole.next(); // prima parola: nome del comando  
        if (scannerDiParole.hasNext())  
            parametro = scannerDiParole.next(); // seconda parola: eventuale parametro  
  
        StringBuilder nomeClasse  
            = new StringBuilder("it.uniroma3.diadia.comandi.Comando");  
        nomeClasse.append( Character.toUpperCase( nomeComando.charAt(0) ) );  
        // es. nomeClasse: 'it.uniroma3.diadia.comandi.ComandoV'  
        nomeClasse.append( nomeComando.substring(1) );  
        // es. nomeClasse: 'it.uniroma3.diadia.comandi.ComandoVai'  
        comando = (Comando) Class.forName( nomeClasse.toString() ).newInstance();  
        comando.setParametro( parametro );  
  
        return comando;  
    }  
}
```

}

# Osservazioni

- Il metodo `newInstance()` invoca il costruttore *no-arg*, che *deve* essere visibile
  - (Finalmente!) il vero motivo per
    - il metodo `setParametro(String)` nella interface `Comando`
    - le classi dei comandi munite di un costruttore *no-args*
  - Al contrario, prevedendo costruttori diversi di comando in comando ci saremmo privati della possibilità di crearli tutti allo stesso modo mediante l'uso del metodo `newInstance()` che presuppone l'assenza di argomenti
- N.B. esistono (da Java 5) anche meccanismi per invocare costruttori con parametri
  - ✓ rispetto agli obiettivi formativi del corso, i tecnicismi in questione non aggiungono molto (vedere i javadoc... )

# Sommario

- Introduzione
- `java.lang.Class<T>`
- Utilizzo della riflessione
- Unchecked Cast Warning
- Riflessione nello studio di caso
  - **FabbricaDiComandiRiflessiva**
- **Equivalenza tra oggetti di una gerarchia**
  - Esercizio

# Riflessione, Criteri di Equivalenza & Polimorfismo

- La scelta di un opportuno criterio d'equivalenza tra due oggetti risulta significativamente più complicata se questi fanno parte di una gerarchia di tipi comune
- E' un problema frequente!
  - ✓ succede ogni qualvolta si crea un `Set<E>` con `E` polimorfo
- La riflessione permette però di codificare controlli sul tipo dinamico di un oggetto

```
@Override
public boolean equals(Object o) {
    if (o==null || o.getClass()!=this.getClass()) return false;
    .../* resto del metodo per confrontare due
        oggetti dello stesso tipo dinamico... */
}

@Override
public int hashCode() {
    return this.getClass().hashCode()+ ... ;
}
```

# Equivalenza e Polimorfismo

- Supponiamo ad esempio di considerare una popolazione di due sottoinsiemi disgiunti di persone, studenti e docenti, come ad es. accade in un dipartimento universitario
- Modelliamo questa situazione introducendo due classi minimali: **Persona** e **Studiante**:

```
public class Persona {
    private String nome;
    public Persona(String nome) { this.nome = nome; }
    public String getNome()      { return this.nome; }
}

public class Studiante extends Persona {
    private String matricola;
    public Studiante(String nome, String matricola) {
        super(nome);
        this.matricola = matricola;
    }
    public String getMatricola() { return this.matricola; }
}
```

# Un Criterio di Equivalenza

- I docenti sono ben pochi ed il loro ricambio è molto lento
  - assunzione: il nominativo sarà un identificatore per tutta la vita dell'applicativo
- Gli studenti sono molti di più ed il loro ricambio è più veloce
  - serve una matricola per identificarli perché presto vi saranno omonimie

```
public class Persona { ...
    @Override public boolean equals(Object o) {
        if (o==null) return false;
        return this.getNome().equals(((Persona)o).getNome());
    }
    @Override public int hashCode() {
        return this.getNome().hashCode();
    }
}

public class Studente extends Persona { ...
    @Override public boolean equals(Object o) {
        if (o==null) return false;
        return this.getMatricola().equals(((Studente)o).getMatricola());
    }
    @Override public int hashCode() {
        return this.getMatricola().hashCode();
    }
}
```

# Set di Oggetti Polimorfi

- L'attuale definizione dei due metodi, sembra essere molto naturale e si presta alla scrittura di alcuni metodi (nella classe **Dipartimento**)
- Ad esempio:
  - **Set<Persona> getAllDocentiAfferenti()**od anche
  - **Set<Studente> getAllStudentiIscrittiAdUnCDS()**
- Questi metodi non mischiano i due tipi di oggetti e risultano facilmente realizzabili
- Che cosa accadrebbe invece se scrivessimo il metodo
  - **Set<Persona> getAllPersoneDelDipartimento()**che restituisce l'insieme di tutte le persone, sia docenti (di tipo dinamico **Persona**) che studenti (**Studente**)???

# Problemi della Definizione «Naturale»

- L'attuale definizione dei due metodi, sebbene sembri naturale, risulta problematica non appena si creano collezioni che mischino oggetti di entrambi i tipi dinamici
  - ✓ Perché a ben vedere, assume confronti solo tra oggetti dello stesso tipo dinamico (o suo sottotipo per il p.d.s.)
- Come completare il seguente codice per rendere l'ass. vera?

```
Persona p = new Persona("Antonio");  
Studiante s = new Studiante("Antonio", "54321");  
Set<Persona> persone = new HashSet<>();  
persone.add(p);  
persone.add(s);  
assertEquals(???, persone.size());
```

- ✓ Non è una vera relazione di equivalenza in quanto asimmetrica:
  - `p.equals(s) → true`
  - `s.equals(p) → ClassCastException!`



# Il Tipo Dinamico nei Criteri di Equivalenza (1)

- Aggiungiamo il controllo sul tipo dinamico, decidendo che oggetti di un tipo non sono mai equivalenti a oggetti di altro tipo

```
public class Persona { ...
    @Override public boolean equals(Object o) {
        if (o==null || o.getClass()!=this.getClass()) return false;
        Persona that = (Persona)o;
        return this.getNome().equals(that.getNome());
    }
    @Override public int hashCode() {
        return this.getClass().hashCode()+this.getNome().hashCode();
    }
}

public class Studente extends Persona { ...
    @Override public boolean equals(Object o) {
        if (o==null || o.getClass()!=this.getClass()) return false;
        Studente that = (Studente)o;
        return this.getMatricola().equals(that.getMatricola());
    }
    @Override public int hashCode() {
        return this.getClass().hashCode()+this.getMatricola().hashCode();
    }
}
```

# Il Tipo Dinamico nei Criteri di Equivalenza (2)

- Asimmetria risolta. Dati:

```
Persona p = new Persona("Antonio");  
Studente s = new Studente("Antonio", "54321");
```

- Risulta:

- `p.equals(s)` → `false`
- `s.equals(p)` → `false!`

- ✓ Una soluzione percorribile *solo* nelle seguenti ipotesi
  - i docenti sono separati dagli studenti
  - non interessa creare oggetti **Persona** per modellare studenti (per i quali usiamo *sempre* oggetti della classe **Studente**)
- Se ora però, allarghiamo la popolazione, sino a coprire l'intero Ateneo, risulta presto necessario risolvere le omonimie tra docenti, ad es. mediante il C.F. (necessario comunque per gli aspetti fiscali)
- Per non costringere la segreteria a raccogliere anche i C.F. degli studenti (assumiamo non strettamente necessario) preferiamo creare una nuova sottoclasse di **Persona**, chiamata **Docente**, piuttosto che aggiungere il campo direttamente nella superclasse

# Gerarchia Totale e Disgiuntiva

```
public class Docente extends Persona {
    private String cf;
    public Docente(String nome, String cf) {
        super(nome);
        this.cf = cf;
    }
    public String getCF() { return this.cf; }
    @Override
    public boolean equals(Object o) {
        if (o==null || o.getClass()!=this.getClass()) return false;
        Docente that = (Docente)o;
        return this.getCF().equals(that.getCF());
    }
    @Override
    public int hashCode() {
        return this.getClass().hashCode()+this.getCF().hashCode();
    }
}
```

- Gerarchia totale: in questo modo docenti e studenti avranno un tipo esplicitamente dedicato e non vengono mai istanziati oggetti che non siano un sottotipo di **Persona** (si potrebbe marcare la classe **Persona** come **abstract** solo per esserne sicuri già a tempo di compilazione)
- Una parte del codice può agevolmente essere messa a fattor comune

# Refactoring di equals() & hashCode()

```
public abstract class Persona { ...
    @Override public boolean equals(Object o) {
        if (o==null || o.getClass()!=this.getClass()) return false;
        Persona that = (Persona)o;
        return this.getNome().equals(that.getNome());
    }
    @Override public int hashCode() {
        return this.getClass().hashCode() + this.getNome().hashCode();
    }
}

public class Docente extends Persona {...
    @Override
    public boolean equals(Object o) {
        return super.equals(o) && this.getCF().equals(((Docente)o).getCF());
    }
    @Override
    public int hashCode() {
        return super.hashCode() + this.getCF().hashCode();
    }
}

public class Studente extends Persona {...
    @Override
    public boolean equals(Object o) {
        return super.equals(o) && this.getMatricola().equals(((Studente)o).getMatricola());
    }
    @Override
    public int hashCode() {
        return super.hashCode() + this.getMatricola().hashCode();
    }
}
```

# Gerarchia Parziale

- E se invece avessimo perseguito la strada senza la classe **Docente**, ovvero aggiungendo il campo per il C.F. a **Persona**, lasciandolo comunque nullo per tutti gli studenti?
- Si tratta di una gerarchia parziale: esistono oggetti che sono istanza della superclasse ma NON della sottoclasse

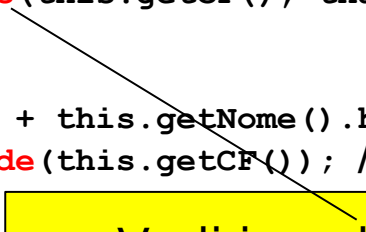
```
public class Persona {  
    private String nome;  
    private String cf;  
    public Persona(String nome, String cf) {  
        this.nome = nome  
        this.cf = cf;  
    }  
    public String getNome() { return this.nome; }  
    public String getCF() { return this.cf; }  
}  
  
public class Studente extends Persona {  
    private matricola;  
    public Studente(String nome, String matricola) {  
        super(nome, null); // Studenti senza C.F.  
        this.matricola = matricola;  
    }  
    public String getMatricola() { return this.matricola; }  
}
```

# Gerarchia Parziale con Campi Null

- Bisogna cambiare anche i metodi `equals()` e `hashCode()` di **Persona**, mostrando il sostanziale forte accoppiamento tra tutte le sue implementazioni nella gerarchia

```
public class Persona {...
    @Override public boolean equals(Object o) {
        if (o==null || o.getClass()!=this.getClass()) return false;
        Persona that = (Persona)o;
        return      this.getNome().equals(that.getNome())
            && java.util.Objects.equals(this.getCF(), that.getCF()); // N.B. gestione null
    }
    @Override public int hashCode() {
        return this.getClass().hashCode() + this.getNome().hashCode()
            + java.util.Objects.hashCode(this.getCF()); // N.B. gestione null
    }
}

public class Studente extends Persona {...
    @Override
    public boolean equals(Object o) {
        return super.equals(o) && this.getMatricola().equals(((Studente)o).getMatricola());
    }
    @Override
    public int hashCode() {
        return super.hashCode() + this.getMatricola().hashCode();
    }
}
```



Vedi javadoc! Consultare anche:  
**java.util.Objects.hash(Object... values)**

# I Problemi Non Finiscono Mai...

- Sinora tutte le varianti ipotizzavano che la diversità del tipo dinamico fosse sempre un fattore decisivo per concludere la NON equivalenza di due oggetti
- ✓ In alcune situazioni potrebbe non essere desiderabile
- Alcuni docenti sono ex studenti dello stesso ateneo...
  - ✓ Come rilevare l'equivalenza dei corrispondenti oggetti di tipo distinto? Ad es. per contare il numero di persone mai transitate, a qualsiasi titolo, in un ateneo?
  - al momento non è affatto possibile perché il nome non è identificatore e gli studenti non possiedono C.F.
- Ma se invece raccogliessimo il C.F. anche degli studenti...
  - ✓ Come converrebbe cambiare la gerarchia delle persone?

# I Problemi ... (cont.)

- Come converrebbe cambiare la gerarchia delle persone per poter contare il numero esatto di persone mai transitate in ateneo?
  - ✓ rimuoviamo tutti metodi `equals()` ed `hashCode()` nella gerarchia tranne che nella superclasse `Persona`, ove insiste SOLO sul C.F.!
- Ma se poi vogliamo fare un metodo  
`static public Set<Studente> iscrittiAppello()`  
senza tediare gli studenti nella modulo di iscrizione con la raccolta del C.F. ma solamente della matricola?
- Quindi per questi oggetti `Studente` vorremmo sovrascrivere il criterio di equivalenza basato sul C.F. di `Persona`
- In sostanza non saremmo più liberi di farlo senza contraddire l'ipotesi che tutte le persone abbiano un C.F.; ipotesi che a quel punto è *cablata* e non sovrascrivibile nella definizione del criterio di equivalenza in `Persona`



# Conclusioni: Equivalenza & Polimorfismo (1)

- Partire dal presupposto che non esiste un automatismo per operare la definizione dei criteri di equivalenza per tipi polimorfi, che rimane una scelta di progettazione
- La definizione di un criterio di equivalenza deve tenere conto di **tutta** la gerarchia
- Rappresenta un intrinseco motivo di forte accoppiamento tra porzioni correlate ma distinte di codice
- Preferire soluzioni semplici: spesso la necessità di soluzioni complicate e/o la definizione di molteplici criteri di equivalenza sulle stesse classi nasconde sottostanti problemi di modellazione e di scelta dei tipi
  - Ad es. una nuova e distinta classe **Prenotazione** che ospiti i riferimenti agli oggetti **Studente** ed un criterio di equivalenza (definito localmente alla classe **Prenotazione**) e basato sulla matricola risolverebbe facilmente i problemi appena menzionati

# Conclusioni: Equivalenza & Polimorfismo (2)

- Il meccanismo di controllo del tipo dinamico va utilizzato o meno a seconda che sia richiesto dal criterio ritenuto più adatto nell'applicazione
- Possono esistere situazioni in cui conviene
  - confrontare il tipo dinamico
  - non controllare il tipo dinamico
  - controllarlo o meno in funzione di alcune informazioni sugli oggetti coinvolti!
- La definizione del più opportuno criterio di equivalenza tra oggetti è uno tra i vari problemi di modellazione
- E' pertanto impossibile definire dei criteri che risultino generali, ancor meno in presenza di gerarchie di tipi

# Conclusioni: Equivalenza & Polimorfismo (3)

- Indicazioni per evitare i più comuni errori:
  - laddove si utilizza di **Set<E>**, **Map<K,V>** chiedersi *sempre* se per tutti i tipi *attuali* di **E** e **K** sia stato già definito l'opportuno criterio di equivalenza, con i meccanismi previsti dal JCF
    - Direttamente mediante la coppia di metodi **equals()** / **hashCode()**
    - Indirettamente definendo un criterio di ordinamento totale (naturale e/o esterno, metodi: **compareTo()** e/o **compare()**) che induce un criterio di equivalenza
  - Se alla luce di questo ragionamento, lo stesso tipo dovrebbe supportare diversi criteri di equivalenza incompatibili, rivedere la modellazione e valutare se non sia il caso di introdurre anche altre classi per alcuni concetti di dominio al momento sono sfuggiti alla «tipizzazione»

# Esercizio

- Si riconsideri l'esercitazione  
**POO-polimorfismo-esercitazione (<<)**
- L'interface **Forma** è successivamente implementata da **Quadrato**, **Rettangolo**, **GruppoDiForme**
- Esercizio: munire tutte le forme geometriche di un opportuno criterio di equivalenza che tenga conto della posizione, del colore, oltre che della tipologia di ciascuna forma

# Esercizio (Suggerimento)

- Alcuni quesiti da porsi immediatamente:
  - i. Quando due oggetti **Rettangolo** sono equivalenti?
  - ii. Quando due oggetti **Cerchio** sono equivalenti?
  - iii. Ha senso confrontare un **Rettangolo** vs un **Cerchio**? **Rettangolo** vs **GruppoDiForme**? **Cerchio** vs **GruppoDiForme**?
  - iv. Quando due oggetti **GruppoDiForme** sono equivalenti?

# Esercizio (cont.)

1. Scrivere una batteria di test di unità per ciascuna classe concreta  
(**Cerchio/Rettangolo/GruppoDiForme**) per codificare quali sono i comportamenti attesi dal criterio di equivalenza tra forme omogenee
2. L'interfaccia **Forma** va modificata?
3. I metodi **equals()** / **hashCode()** in quali classi vanno introdotti?
4. Controllare che i test abbiano successo
5. Aggiungere altri test di unità coinvolgendo come fixture insiemi di una sola tipologia di forma:
  - **Set<Cerchio>**
  - **Set<Rettangolo>**
  - **Set<GruppoDiForme>**

# Esercizio (cont.)

6. Aggiungere dei test per verificare il corretto funzionamento dei metodi `equals()` / `hashCode()` quando sollecitati su forme di tipo dinamico diverso
7. Modificare il codice per soddisfare i test di prima
8. Aggiungere i test per verificare il comportamento di una collezione di diversi tipi di forme utilizzando come fixture un insieme
  - `Set<Forma>`
9. Come modificare il codice per soddisfare i nuovi test?

# Esercizio (cont.)

10. Introdurre una nuova classe astratta **AbstractForma**
11. Rifattorizzare quando più codice possibile facendo uso della nuova classe
12. Confermare che i test di unità già sviluppati continuino ad aver successo dopo il refactoring
13. Cambiare i requisiti e modificare di conseguenza il codice secondo un nuovo criterio di equivalenza che includa anche la classe **Punto** come caso particolare di **Forma** e che consideri le altre forme, come ad es. i cerchi (e i rettangoli, rispett.), equivalenti ai punti nel caso degenere (rispett. raggio e base/altezza pari a zero)



# Approfondimenti

- Per evitare gli errori più comuni nella definizione di un criterio di equivalenza, e per un approfondimento sul legame tra il principio di sostituzione ed i criteri di equivalenza tra tipi polimorfi:

Martin Odersky, Lex Spoon, and Bill Venners

*How to Write an Equality Method in Java*

<http://www.artima.com/lejava/articles/equality.html>