# Programmazione Orientata agli Oggetti

Classi Astratte e
Costanti Enumerative

# Contenuti

- Classi astratte
  - Metodi astratti
  - Classi astratte o interface?
- Metodi e Classi finali
- Costanti enumerative
  - Pre-Java 5
  - Java 5 Enum
  - Quando usarli
  - Enum e Collezioni

# Contenuti

- Classi astratte
  - Metodi astratti
  - Classi astratte o interface?
- Metodi e Classi finali
- Costanti enumerative
  - Pre-Java 5
  - Java 5 Enum
  - Quando usarli
  - Enum e Collezioni

# Introduzione (1)

- Abbiamo visto come l'estensione può essere uno strumento utile per il riuso del codice
- Le classi estese ereditano anche l'implementazione della classe base
  - La classe estesa può avere variabili di istanza e metodi aggiuntivi a quelli ereditati dalla classe base
  - La classe estesa può ridefinire metodi della classe base
- La classe estesa è un sottotipo della classe base, quindi, vale il principio di sostituzione:
  - ✓ possiamo usare istanze di una classe estesa al posto delle istanze della classe base

# Introduzione (2)

- Abbiamo visto che anche le interface favoriscono il riuso del codice
- Grazie al principio di sostituzione possiamo scrivere codice con metodi polimorfi
  - accettano come parametro formale un riferimento ad un tipo (statico) astratto
  - il parametro attuale sarà un riferimento ad un sottotipo (dinamico) concreto di tale tipo
- Talvolta, classi diverse che implementano una medesima interface possono voler condividere anche una significativa porzione dell'implementazione

# Motivazioni

- Nella pratica, capita di voler definire classi base pensate **solo** per essere estese e non per essere direttamente istanziate
  - ✔ allo scopo di evitare duplicazioni nel codice
  - ✔ Per favorire la qualità *interna* (<<)
- Queste classi contengono una parziale implementazione da condividere con le sottoclassi
  - variabili di istanza
  - implementazione di alcuni metodi
    - cosidetti metodi «concreti»
  - segnatura di altri metodi
    - cosidetti metodi «astratti»
- I metodi che rimangono privi di implementazione nella classe base possono poi essere «completati» nelle classi derivate

# Classi Astratte

- Una classe astratta contiene una definizione parziale della implementazione
- Una classe astratta non può essere istanziata, ma possono essere istanziate le classi (concrete) che la estendono
- Ovviamente vale la relazione sottotipo-supertipo, e quindi il principio di sostituzione
  - ✓ i riferimenti alle istanze delle classi che estendono una classe astratta possono essere usate quando è atteso un riferimento ad una istanza della classe base
- ✔ *Analogie* con le interface:
  - le interface non possono essere istanziate, ma possono essere istanziati oggetti di classi che le implementano
  - vale il principio di sostituzione
- ✔ *Differenze* con le interface:
  - le interface NON contengono implementazione (non più vero da Java 8+!!!)

# Esempio (Caso di Studio)

- Supponiamo di voler introdurre nel nostro gioco dei personaggi (es. mostri, maghi, ecc.)
- I personaggi sono nelle stanze del gioco (per semplicità accontentiamoci di un singolo personaggio per stanza)

```
import …
public class Stanza {
    private String nome;
    private Map<String, Stanza> uscite;
    private Map<String, Attrezzo> nome2attrezzo;

    private AbstractPersonaggio personaggio;
    … … …
    public void setPersonaggio(AbstractPersonaggio personaggio) {
        this.personaggio = personaggio;
    }

    public AbstractPersonaggio getPersonaggio() {
        return this.personaggio;
    }
    …
}
```

# Esempio (Caso di Studio)

- I personaggi hanno un nome (una stringa) ed una descrizione (una stringa)
- I personaggi possono rispondere al saluto del giocatore
  - introduciamo il comando **`saluta`** per salutare il personaggio presente nella stanza
- I personaggi possono agire
  - introduciamo il comando **`interagisci`**: provoca l'interazione del giocatore con il personaggio presente

# Esempio: i «Personaggi»

- Possiamo introdurre diverse tipologie di personaggi. Ad esempio potremmo avere:
  - *Mago:* possiede un attrezzo che può donare
  - *Strega:* se interagiamo con una strega questa ci trasferisce in una stanza tra quelle adiacenti. Siccome è permalosa:
    - se non l'abbiamo ancora salutata, ci «trasferisce» nella stanza adiacente che contiene meno attrezzi
    - altrimenti in quella che contiene più attrezzi
  - *Cane:* morde! Ogni morso diminuisce i CFU del protagonista

# Esempio: Caratteristiche Generali

- Tutte le tipologie di personaggi condividono una parte della implementazione
  - le variabili per memorizzare nome e descrizione
  - metodi accessori
  - costruttori
  - il codice che gestisce la risposta al saluto
- Tutte le tipologie di personaggi hanno un metodo (astratto) per modellare l'azione:

```
abstract public String agisci(Partita partita);
```

- Tuttavia non ha senso definire sino al dettaglio il comportamento di un generico personaggio
- ✓ il comportamento dipende infatti dalle specificità di ogni particolare personaggio

# Esempio (cont.)

- Ogni personaggio ha un comportamento specifico
  - il mago ci dona un attrezzo
  - la strega ci sposta in una stanza
  - il cane morde
- Il codice che implementa l'azione è specifico del personaggio: ogni tipologia di personaggio ha la propria, specifica, implementazione
- In altri termini, il personaggio è definibile prescindendo da alcuni suoi dettagli che lo differenziano da tutti gli altri
  - alcune sue proprietà possono essere «concrete» (fornite anche di una implementazione)
  - altre solo «astratte» (ovvero senza l'implementazione)

# Esempio: la Classe Astratta Personaggio (1)

```
package it.uniroma3.personaggi;
import …

public abstract class AbstractPersonaggio {
   private String nome;
   private String presentazione;
   private boolean haSalutato;

   public AbstractPersonaggio(String nome, String presentaz) {
       this.nome = nome;
       this.presentazione = presentaz;
       this.haSalutato = false;
   }

   public String getNome() {
       return this.nome;
   }

   public boolean haSalutato() {
       return this.haSalutato;
   }
```

# Esempio: la Classe Astratta Personaggio (2)

```
    public String saluta() {
        StringBuilder risposta =
                new StringBuilder("Ciao, io sono ");
        risposta.append(this.getNome()+".");
        if (!haSalutato)
                risposta.append(this.presentazione);
        else
                risposta.append("Ci siamo gia' presentati!");
        this.haSalutato = true;
        return risposta.toString();
    }

    abstract public String agisci(Partita partita);

    @Override
    public String toString() {
        return this.getNome();
    }

}
```

# La Classe ComandoInteragisci

```
package it.diadia.comandi;
import ...
public class ComandoInteragisci implements Comando {

    private static final String MESSAGGIO_CON_CHI =
                                        "Con chi dovrei interagire?...";
    private String messaggio;
    private IO io;

    @Override
    public void esegui(Partita partita) {
        AbstractPersonaggio personaggio;
        personaggio = partita.getStanzaCorrente().getPersonaggio();
        if (personaggio!=null) {
            this.messaggio = personaggio.agisci(partita);
            io.mostraMessaggio(this.messaggio);
        } else io.mostraMessaggio(MESSAGGIO_CON_CHI);
    }
    public String getMessaggio() {
        return this.messaggio;
    }

    @Override
    public void setParametro(String parametro) {}
}
```

# Esempio: la Classe Mago

```
public class Mago extends AbstractPersonaggio {

    private static final String MESSAGGIO_DONO = "Sei un vero simpaticone, " +
        "con una mia magica azione, troverai un nuovo oggetto " +
        "per il tuo borsone!";

    private static final String MESSAGGIO_SCUSE = "Mi spiace, ma non ho piu' nulla...";

    private Attrezzo attrezzo;

    public Mago(String nome, String presentazione, Attrezzo attrezzo) {
        super(nome, presentazione);
        this.attrezzo = attrezzo;
    }

    @Override
    public String agisci(Partita partita) {
        String msg;
        if (this.attrezzo!=null) {
            partita.getStanzaCorrente().addAttrezzo(this.attrezzo);
            this.attrezzo = null;
            msg = MESSAGGIO_DONO;
        }
        else {
            msg = MESSAGGIO_SCUSE;
        }
        return msg;
    }
}
```

# Classi e Metodi Astratti

- Le classi astratte:
  - utilizzano il modificatore **`abstract`** nella definizione
  - non possono essere istanziate *direttamente* (sebbene devono avere, anche solo implicitamente, almeno un costruttore!)
  - definiscono implementazioni poi ereditate dalle sottoclassi
- I metodi astratti:
  - utilizzano **`abstract`** nella segnatura
  - non possiedono corpo
- La presenza anche di un solo metodo astratto rende necessaria la dichiarazione della classe come astratta
  - ma una classe può essere dichiarata astratta anche se non possiede alcun metodo astratto (e non potrà essere istanziata)
- Le sottoclassi, per essere concrete ed istanziabili, devono completare l'implementazione sovrascrivendo *tutti* i metodi astratti (oppure, a loro volta, devono dichiararsi astratte…)

# Classi Astratte

- Servono a definire implementazioni parziali che verranno completate nelle classi concrete che le estendono
- Rispetto alle interface ?
  - come le interface non possono essere istanziate
  - diversamente dalle interface riportano una implementazione parziale e vengono estese
- Molto usate nei framework, ma la progettazione di framework va (ben!) oltre gli obiettivi formativi di questo corso...

# Classi Astratte vs Interface

- Classe astratta
  - *pro:* permette di riutilizzare l'implementazione
  - *contro:* limita fortemente le possibilità di estensione
    - ✓ la gerarchia delle implementazioni Java è *lineare*
- Interface
  - *pro:* nessun limite di estensione
    - ✓ ereditarietà multipla delle interfacce
  - *contro:* nessun meccanismo di riutilizzo del codice
- Come scegliere?
  - preferire le interface, meno vincolanti
  - Valutare l'alternativa solo in presenza di codice e soprattutto logica da riutilizzare: per es. con i framework

# Testing di Classi Astratte (1)

- Qual è l'obiettivo del testing di classi astratte?
  - ✔ testare i metodi implementati direttamente nella classe astratta
- Conviene definire una sua estensione concreta solo ai fini del testing
  - ✔ fornisce una implementazione concreta (e *minimale*) dei metodi astratti per permettere la creazione dell'oggetto da testare
  - ✔ ad es. implementa i metodi astratti restituendo costanti
- Si testano i metodi concreti della superclasse

# Testing di Classi Astratte (2)

```
public class FakePersonaggio extends AbstractPersonaggio {

    public FakePersonaggio(String nome, String presentazione) {
        super(nome, presentazione);
    }

    @Override
    public String agisci(Partita partita) {
        return "done";
    }

}
```

- Scriviamo la classe di test **`AbstractPersonaggioTest`** che verifichi il comportamento dei metodi concreti ereditati da **`AbstractPersonaggio`**

# Esercizi (1)

- Definire la classe **`Cane`**, che estende la classe **`AbstractPersonaggio`**: quando interagiamo con un cane, questi morde, togliendoci CFU!
- Definire il comando *interagisci*
- Introdurre **`AbstractComando`** per eliminare i metodi replicati nelle implementazioni dell'interface **`Comando`** e relativi alla gestione del nome e del parametro del comando
  - Utilizzarla anche per ospitare l'oggetto **`IO`** per la gestione dell'input/output. Come offrire l'accesso alle sottoclassi?

✔ N.B. quest'ultimo utilizzo delle classi astratte risulta utile per esercitarsi con l'uso dello strumento ma è in *nitido* contrasto con la raccomandazione di limitare il loro utilizzo al caso di sufficiente logica duplicata

# Esercizi (2)

- Definire la classe **`Strega`** come riportato nella descrizione precedente
  - ✓ è anche un buon esercizio sulle collezioni...

- Definire ed organizzare i test-case per tutte le classi appena introdotte; rifattorizzare i test-case già prodotti per le classi modificate

# Contenuti

- Classi astratte
  - Metodi astratti
  - Classi astratte o interface?
- Metodi e Classi finali
- Costanti enumerative
  - Pre-Java 5
  - Java 5 Enum
  - Quando usarli
  - Enum e Collezioni

# Metodi e Classi `final` (1)

- Per evitare che un metodo possa essere ridefinito si usa il modificatore **`final`**

```
public class ClasseConMetodoFinale {
   public final int nonMiPoteteSovrascrivere() {…}
}
```

- In questo modo il metodo non può essere ridefinito nelle classi che estendono **`ClasseConMetodoFinale`**
  - viene sollevato un errore a tempo di compilazione

```
public class EstendeClasseConMetodoFinale
                extends ClasseConMetodoFinale {
  @Override
  public int nonMiPoteteSovrascrivere(){…}//ERRORE
}
```

# Metodi e Classi `final` (2)

- E' possibile dichiarare **`final`** intere classi

```
public final class NonMiPoteteEstendere {
   …
}
```

- La classe **`NonMiPoteteEstendere`** non può essere estesa
  - ✔ anche in questo caso errore a tempo di compilazione

```
public class EstendeClasseFinale
        extends NonMiPoteteEstendere { // ERRORE
   …
}
```

- Molte classi della libreria standard sono dichiarate **`final`**, un esempio per tutti: **`java.lang.String`**

# Contenuti

- Classi astratte
  - Metodi astratti
  - Classi astratte o interface?
- Metodi e Classi finali
- Costanti enumerative
  - Pre-Java 5
  - Java 5 Enum
  - Quando usarli
  - Enum e Collezioni

# Java Enum e Costanti Enumerative

- I metodi e le classi **`final`** sono stati inizialmente motivati da questioni legate alle sicurezza
- Hanno conosciuto un altro diffuso utilizzo, ma forse meno noto perché nascosto da un consistente strato di zucchero sintattico
- *Costanti Enumerative:* valori costanti raccolti in insiemi finiti e stabili. Es.:
  - i 12 mesi dell'anno ed i 7 giorni della settimana
  - i 7 *tipi* di tetramino nel gioco Tetris!
  - i 4 punti cardinali (>>)
- I Java Enum (da Java 5) consentono di modellare efficacemente insiemi di «costanti enumerative» usando opportunamente
  - Classi astratte
  - Classi **`final`**
  - Costruttori privati

# Enumerazioni (pre Java 5)

- Per capire il loro funzionamento, ed apprezzarne i vantaggi, conviene descrivere come venivano realizzati prima di Java 5, (versione in cui ne fu introdotto il supporto)
  - ✔In effetti è la pratica tuttora utilizzata in C
- In assenza di un meccanismo dedicato, la modellazione di costanti enumerative ripiegava spesso nell'uso di generiche costanti di un tipo «preso in prestito»
- Ad esempio, mediante semplici numeri interi:

```
public interface Direzione {
   // Da NORD in senso orario
   static final public int NORD  = 0;
   static final public int OVEST = 1;
   static final public int SUD   = 2;
   static final public int EST   = 3;
}
```

# Enumerazioni (pre Java 5)

- Principale vantaggio: semplicità ed immediatezza
- Importanti conseguenze negative, fra tutte una tipizzazione piuttosto lasca:

```
int direzione = 5; // COMPILA! Ma non dovrebbe…
```

- Ne derivano molti svantaggi, ad es. Metodi con segnature di non immediata interpretazione:

```
public int direzioneOpposta(int arg) { … }
```

# Enumerazioni Tipate (pre Java 5)

- Soluzioni più articolate prevedono infatti almeno la creazione di un tipo dedicato, per rendere i controlli a tempo statico e la tipizzazione più stringenti
- Una soluzione diffusamente utilizzata:
  - classi **`final`**
  - costruttori privati

```
public final class Direzione {
  private int ordinal;
  private Direzione(int index) { this.ordinal = index; }
  public int getOrdinal()      { return this.ordinal;  }
  static public final Direzione NORD  = new Direzione(0);
  static public final Direzione EST   = new Direzione(1);
  static public final Direzione SUD   = new Direzione(2);
  static public final Direzione OVEST = new Direzione(3);
  static private final Direzione[] cardinali =
                                    { NORD, EST, SUD, OVEST };
  static public Direzione[] values() { return cardinali; }
}
```

# Enumerazioni Tipate (pre Java 5)

- Risolti i problemi legati alla tipizzazione lasca:

```
Direzione dir = new Direzione(5); // NON COMPILA

public Direzione direzioneOpposta(Direzione arg) {…}
```

- Nuove possibilità, ad es. cicli for-each:

```
for(Direzione dir : Direzione.values()) { … }
```

- Si può anche pensare di affiancare metodi (polimorfi) alle costanti. Ad es. per il calcolo della direzione **`opposta()`**:

```
import static Direzione.*;

assertEquals(SUD, NORD.opposta());
```

- Si rende però necessario creare un sottotipo per ogni costante della stessa enumerazione>>

# Enumerazioni Polimorfe (pre Java 5)

```
public abstract class Direzione {
  private int ordinal;
  private Direzione(int index) { this.ordinal = index; }
  public int getOrdinal()      { return this.ordinal;  }
  abstract public Direzione opposta();
  static public final Direzione NORD  = new NORD();
  static public final Direzione EST   = new EST();
  static public final Direzione SUD   = new SUD();
  static public final Direzione OVEST = new OVEST();
  static final class NORD extends Direzione  {
    private NORD()  { super(0); }
    @Override public Direzione opposta() { return SUD; }
  }
  // …similarmente per EST, SUD…
  …
  static final class OVEST extends Direzione  {…}
}
```

Permette di dichiarare una nuova classe *interna* (>>) e di invocare il costruttore privato

# Limitazioni dell'Enumerazioni pre Java 5: Perché i Java 5+ Enum

- Efficace: ma per N costanti è necessario creare N+1 classi...
  - ✔ Decisamente macchinoso, anche con l'aiuto di un IDE evoluto
- I Java Enum introdotti in Java 5 sostanzialmente adottano questa stessa soluzione, ma in aggiunta offrono:
  - Una sintassi meno verbosa; ad es. per enum senza metodi aggiuntivi basta scrivere:

```
public enum Direzione {
    NORD, EST, SUD, OVEST;
}
```

  - Una superclasse astratta **`java.lang.Enum`** condivisa da tutti i tipi enumerati
    - con metodi condivisibili da tutti i tipi enumerati (es. **`ordinal()`**)
  - Un generoso aiuto da parte del compilatore
    - definisce alcuni metodi statici aggiuntivi (>>)

# Java Enum: Sintassi

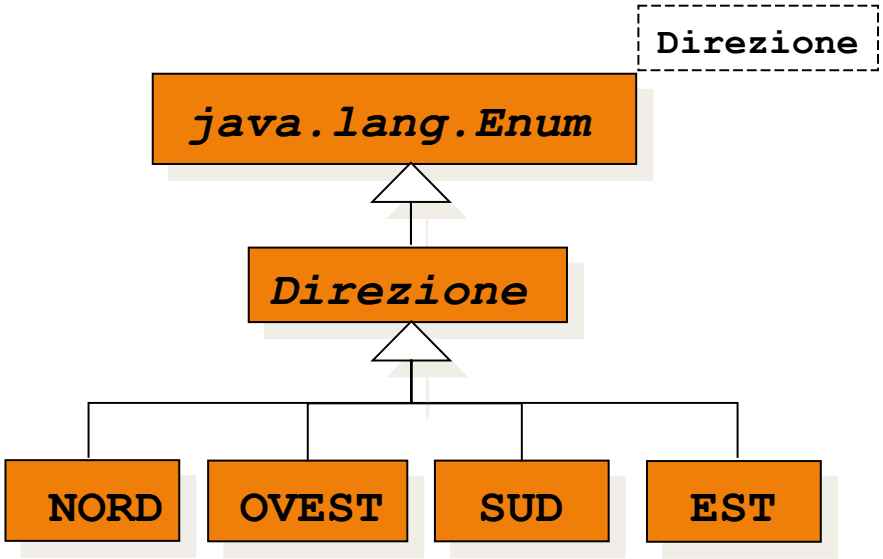
- In presenza di metodi aggiuntivi, la sintassi dei Java Enum in caso di enumerazioni polimorfe è più prolissa, ma comunque più compatta della soluzione “manuale”:

```
public enum Direction {
   NORD() {
      @Override public Direction opposta() {
         return SUD;
      }
   },
   …
   OVEST() {
      @Override public Direction opposta() {
         return EST;
      }
   };
   public abstract Direction opposta();
}
```

# java.lang.Enum: Gerarchia

- Per ogni Enum dichiarato il compilatore genera
  - Una sottoclasse (ad es. **Direzione**) della classe astratta generica **java.lang.Enum** che modella il tipo enumerato
  - Una sua sottoclasse per ciascun valore costante (ad es. **NORD**)

```
public abstract class Enum<E extends Enum<E>>
                          implements Comparable<E>
```



# `java.lang.Enum`: Costruttore

- La classe **`java.lang.Enum`** accentra funzionalità per tutti i tipi enumerati
  - Ad es. metodi per gestire i numeri ordinali (**`ordinal()`**) ed il nome del tipo enumerato (**`name()`**)
- Unico costruttore:

**`Enum protected Enum(String name, int ordinal)`**

*Sole constructor. Programmers cannot invoke this constructor. It is for use by code emitted by the compiler in response to enum type declarations.*

**Parameters**:

**`name`** - *The name of this enum constant, which is the identifier used to declare it.*

**`ordinal`** - *The ordinal of this enumeration constant (its position in the enum declaration, where the initial constant is assigned ordinal 0).*

# Java Enum: Costruttori delle «Costanti Enumerative»

- I valori costanti possono avere uno o più costruttori ed uno stato
  - Si dichiara un costruttore *privato* solo nel tipo enumerato (es. **`Direzione`**)
  - Lo stato *deve* essere immutabile (>>)
    - ✔ è preferibile dichiarare tutte le variabili di istanza **`final`**

```
public enum Direzione {
    NORD(0) {      @Override
                   public Direzione opposta() { return SUD; }
    },…,
    OVEST(270) {   @Override
                   public Direzione opposta() { return EST; }
    };
    private final int gradi;
    private Direzione(int gradi) {
       this.gradi = gradi;
    }
    public int getGradi() { return this.gradi; }
    public abstract Direzione opposta();
}
```

# `java.lang.Enum`: Metodi (1)

- Documentiamo la semantica degli altri metodi di **`java.lang.Enum`**
  - tramite qualche test di unità, per es. sul tipo **`Direzione`**
  - metodo **`ordinal()`**

```
import static org.junit.Assert.*;
import org.junit.Test;
import static Direzione.*;
public class EnumTest {
    @Test
    public void testOrdinal() {
        assertEquals(0, NORD.ordinal());
        assertEquals(1, EST.ordinal());
        assertEquals(2, SUD.ordinal());
        assertEquals(3, OVEST.ordinal());
    }
… // altri test-case a seguire
}
```

# java.lang.Enum: Metodi (2)

- Ogni tipo enumerativo è associato ad una classe per l'intero tipo più un oggetto-singleton/classe per ciascuno dei valori costanti del tipo
- Le costanti ricordano la classe del loro supertipo (ed associato a tutta la collezione di costanti enumerative) mediante il metodo

**`Class<E>  getDeclaringClass()`**

- *returns the Class object corresponding to this enum constant's enum type.*

```
public class EnumTest {…
    @Test
    public void testGetDeclaringClass() {
        assertSame(Direzione.class, NORD.getDeclaringClass());
        assertNotSame(Direzione.class, NORD.getClass());
        assertNotSame(EST.getClass(), NORD.getClass());
        … // similarmente per le altre direzioni
    }
…}
```

# `java.lang.Enum`: Metodi (3)

- Ad ogni valore costante è associato un singolo oggetto (singleton)
- Evidente con un test sul metodo **`valuesOf()`**, che consente di creare gli oggetti associati alle costanti di un tipo enumerato anche senza scomodare la più "pesante" API sull'introspezione (>>)

```
public class EnumTest {…
    @Test
    public void testTuttiSingleton() {
        assertSame(NORD, Direzione.valueOf("NORD"));
        final Direzione singleton = Direzione.valueOf("NORD");
        assertSame(singleton, NORD);
        …
        assertNotSame(EST, NORD);
        … // similarmente per le altre direzioni
    }
}
```

# Enumerazioni: Criterio di Equivalenza (1)

- Quindi riassumendo:
  - ogni tipo enumerato di N valori costanti genera N+1 classi
  - di queste N sono singleton (classi a singola istanza) che modellano i valori
  - ad ogni valore corrisponde quindi una sola classe che possiede una sola istanza
- Tutto questo si riflette sul criterio di equivalenza dei tipi enumerati che risulta essere quello più naturale
- Ciascuno dei valori costanti è un oggetto che
  - è equivalente solo a se stesso
  - non è equivalente a nessuno degli altri

# Enumerazioni: Criterio di Equivalenza (2)

- **`equals()`** e l'operatore **`==`** finiscono per modellare lo stesso criterio di equivalenza
  - ATTENZIONE! NON è affatto vero in generale: vale solo per i tipi enumerativi!
- **`hashCode()`** ritorna il valore restituito dal metodo **`ordinal()`**
  - **`HashSet`** ed **`HashMap`** su tipi enumerativi possiedono prestazioni ottimali (non esistono conflitti!)

```
public class EnumTest {…
   @Test
   public void testCriterioDiEquivalenza() {
        assertEquals(NORD, NORD);
        assertNotEquals(NORD, EST);
        assertNotEquals(NORD, SUD);//…
         … // similarmente per le altre direzioni
   }…
}
```

# Enumerazioni: Ordinamento

- E’ direttamente quello basato sul valore ordinale
  - N.B. Induce anche un criterio di equivalenza basato sull’identicità degli oggetti
    - Ovvero come l’operatore ==

```
public class EnumTest {…
  @Test
  public void testCompareTo() {
    assertTrue(NORD.compareTo(EST)<0);
    assertTrue(EST.compareTo(SUD)<0);
    assertTrue(SUD.compareTo(OVEST)<0);
    assertTrue(OVEST.compareTo(NORD)>0);
  }
…}
```

# Enum.toString() / name()

- Non è necessario ridefinire un metodo **`toString()`** in quanto è naturalmente definito sulla base del nome stesso della costante

**`String name()`**

- *Returns the name of this enum constant, as declared in its enum declaration.*

**`String toString()`**

- *Returns the name of this enum constant, as contained in the declaration.*

```
public class EnumTest {…
  @Test
  public void testToStringAndName() {
    assertEquals("NORD", NORD.toString());
    assertEquals("NORD", NORD.name());
    … // similarmente per le altre direzioni
  }…
}
```

# Java Enum e Metodi «Fantasma» (1)

- I Java Enum introducono «zucchero sintattico»: la controparte nei sorgenti di codice oggetto generato non risulta osservabile
  - si pensi a boxing-unboxing (<<)
- Sorprendentemente, anche *alcuni* metodi appaiono, in particolare il metodo statico **`values()`** ed anche **`valueOf()`**
  - Metodi di cui non esiste il codice sorgente!
  - Eclipse non fornisce meta-informazioni! (e javadoc)
- Il compilatore genera automaticamente questi metodi adottando un comportamento particolare per tutte le classi che estendano **`java.lang.Enum`**:

```
public class EnumTest {…
  @Test
  public void testValues() {
     final Direzione[] expected = { NORD, EST, SUD, OVEST } ;
     assertArrayEquals(expected , Direzione.values());
  }…
}
```

# Java Enum e Metodi «Fantasma» (2)

- Per avere documentazione è necessario riferirsi direttamente alla Java Language Specification

```
/**
* Returns an array containing the constants of this enum
* type, in the order they're declared.  This method may be
* used to iterate over the constants as follows:
*    for(E c : E.values())
*        System.out.println(c);
* @return an array containing the constants of
* this enum type, in the order they're declared */
public static E[] values();
/**
* Returns the enum constant of this type with the specified name.
* The string must match exactly an identifier used to declare
* an enum constant in this type.  (Extraneous whitespace
* characters are not permitted.)
* @return the enum constant with the specified name
* @throws IllegalArgumentException if this enum type has no
* constant with the specified name */
public static E valueOf(String name);
```

Il metodo statico **`values()`** rende possibile usare cicli «for-each» sopra tipi enumerati

# Java Enum e «Switch-Statement»:

- E’ possibile usare tipi enumerati direttamente negli «switch statement»;  in precedenza era possibile farlo solo per le costanti letterali

```
import static Direzione.*;
public class DirezioneUtils {
  public static Direzione opposta(Direzione direzione) {
    switch (direzione) {
      case NORD:  return SUD;
      case EST:   return OVEST;
      case SUD:   return NORD;
      case OVEST: return EST;
      default:    return null;
    }
  }
}
```

# Conclusioni: Costanti Enumerative - Quando Usarle? (1)

- Quando è opportuno utilizzare costanti enumerative?
- Quando l'insieme di elementi da modellare:
  - ✓ è finito
  - ✓ tutti i suoi elementi siano già noti
  - ✓ è «stabile»: molto difficilmente cambierà nel tempo
- Quando per gli elementi ospitati:
  - ✓ ne conosciamo i nomi... «propri»
    - ad es. "NORD", "GENNAIO"
  - ✓ possono anche avere uno stato, purché *immutabile*
    - inutile/impossibile crearne diverse «istanze»

# Conclusioni:
# Costanti Enumerative - Quando Usarle? (2)

- Ed inoltre:
  - Il dominio suggerisce
    - una tipizzazione forte (concetto di “primo ordine”)
    - un tipo dedicato alla collezione di costanti
  - Anche per  l’esigenza di aggiungere metodi (polimorfi) alle costanti
    - Ad es. **`Direzione.opposta()`**

# Esercizio (Studio di Caso)

- Introdurre l'enumerazione **`Direzione`** nello studio di caso per modellare con un tipo dedicato le direzioni degli spostamenti e delle adiacenze tra oggetti **`Stanza`**
  - ✔ Attualmente si «prende in prestito» il tipo **`String`**
  - ✔ Ma attenzione, è un sintomo di cattiva «modellazione»:

    *http://c2.com/cgi/wiki?StringlyTyped*

# Esercizio (Tetris)

- Studiare il codice dell'esercitazione «Tetris», in particolare nel package **`tetris.tetramino`**
- Perché si è deciso di separare la modellazione del tetramino (classe **`tetris.tetramino.Tetramino`**) da quella del suo tipo (**`enum tetris.tetramino.Tipo`**)?
- *Suggerimento:*
  - Da quale campi è composto lo stato degli oggetti istanza dei due tipi?
  - Quali sono o possono essere dichiarati **`final`**?
  - Quale dei due tipi possiede uno stato mutabile?
- Cosa accadrebbe se fondessimo le due classi addossando le responsabilità di **`Tipo`** alla classe **`Tetramino`**?

# Enum e Collezioni

- Le costanti enumerative finiscono per costituire un tipo  di collezione di caratteristiche particolari
- Sono possibili rappresentazioni ancora più compatte ed efficienti rispetto alle generiche collezioni (che già sono particolarmente efficienti sui tipi enumerativi)
- Dai javadoc di **`java.util.EnumSet`**
  - *Implementation note: All basic operations execute in constant time. They are likely (though not guaranteed) to be much faster than their* **`HashSet`** *counterparts. Even bulk operations execute in constant time if their argument is also an enum set.*
- Vedere anche **`java.util.EnumMap`**
  - *Specializzazione di* ***`java.util.Map`*** *che utilizza come chiavi un tipo enumerativo*

# Conclusioni

- L’insieme dei meccanismi offerti da un linguaggio per la modellazione dei tipi è uno degli aspetti più caratterizzanti lo stesso
- Con le classi astratte abbiamo coperto l’insieme dei principali meccanismi per la definizione dei tipi:
  - Interface
  - Classi
  - Classi astratte
  - Generics
- Esistono ancora altri meccanismi:
  - Tipi enumerativi
  - Classi nidificate (>>)

  meno importanti dei precedenti, ma che contribuiscono alla richezza del sistema dei tipi in Java