

# Programmazione ad Oggetti

---

Generics: concetti base

# Obiettivi

- I *generics* sono uno strumento per scrivere classi (e metodi) ***parametriche rispetto ad un tipo*** introdotti nelle versioni java più recenti
- Ci concentriamo soprattutto su come usare classi generiche
  - Al termine lo studente dovrà essere in grado di usare classi generiche (in particolare quelle del package `java.util`)
- La progettazione di classi generiche va oltre gli obiettivi del corso
  - Però, da un punto di vista didattico è utile introdurre i *generics* progettando una semplice classe

# Introduzione

- Supponiamo di dover scrivere una classe **Coppia**, che consente di gestire coppie di oggetti dello stesso tipo
- Vogliamo una classe generica, che possa essere usata in contesti diversi. Ad esempio la classe dovrà gestire:
  - Coppie di stringhe (istanze della classe **String**)
  - Coppie di attrezzi (istanze della classe **Attrezzo**)
  - Coppie di URL (istanze della classe **URL**)
  - ...

# La classe generica `Coppia`

- La classe `Coppia` deve offrire:
  - Un metodo per ottenere il primo elemento della coppia
  - Un metodo per ottenere il secondo elemento della coppia
  - Un costruttore che prende come parametri due riferimenti ad oggetti dello stesso tipo

# Controllo sui tipi

- Considereremo di seguito del codice che fa riferimento alla seguente classe **Persona**

```
import java.util.*;

class Persona {
    private String nome;

    public Persona(String nome) {
        this.nome = nome;
    }

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

# Una possibile soluzione: il polimorfismo

- Una possibile soluzione consiste nello sfruttare il polimorfismo, ed in particolare il principio di sostituzione
- Definiamo una classe che gestisce una coppia di oggetti istanza di Object
  - per il principio di sostituzione (e per la gerarchia delle classi Java) la nostra classe può gestire coppie di oggetti istanza di qualsiasi classe (in quanto sottotipi di Object)

# La classe Coppia implementata con Object

```
public class Coppia {  
    private Object primo;  
    private Object secondo;  
  
    public Coppia() {}  
  
    public Coppia(Object primo, Object secondo) {  
        this.primo = primo;  
        this.secondo = secondo;  
    }  
  
    public Object getPrimo() {  
        return this.primo;  
    }  
  
    public Object getSecondo() {  
        return this.secondo;  
    }  
  
    public void setPrimo(Object primo) {  
        this.primo = primo;  
    }  
  
    public void setSecondo(Object secondo) {  
        this.secondo = secondo;  
    }  
  
}
```

# Controllo sui tipi

- Consideriamo il seguente codice: compila e gira correttamente

```
public class CollezioniTest {  
    @Test  
    public void testCheCompilaEgira() {  
        Coppia coppia = new Coppia();  
        Persona p1 = new Persona("Pippo");  
        coppia.setPrimo(p1);  
        Persona p2 = new Persona ("Pluto");  
        coppia.setSecondo(p2);  
        Persona persona = (Persona)coppia.getPrimo();  
        System.out.println(persona.getNome());  
    }  
}  
}
```



# Controllo sui tipi

- Consideriamo il seguente codice

```
public class ProblemiConCoppiaTest {  
    @Test  
    public void testCheCompilaEgira() {  
        Coppia coppia = new Coppia();  
        Persona p1 = new Persona("Pippo");  
        coppia.setPrimo(p1);  
        Persona p2 = new Persona("Pluto");  
        coppia.setSecondo(p2);  
        System.out.println(coppia.getPrimo().getNome());  
  
        System.out.println((Persona) coppia.getPrimo());  
    }  
}
```

Solleva un errore a  
tempo di  
**compilazione!**

# Controllo sui tipi

- Consideriamo il seguente codice: compila correttamente ma l'esecuzione fallisce

```
public class ProblemiConCoppiaTest {  
    @Test  
    public void testCheCompilaMaNonGira() {  
        Coppia coppia = new Coppia();  
        Persona p1 = new Persona("Pippo");  
        coppia.setPrimo(p1);  
        String p2 = new String("Pluto");  
        coppia.setSecondo(p2);  
        Persona persona = (Persona)coppia.getSecondo();  
        System.out.println(persona.getNome());  
    }  
}
```

**Solleva un errore (ClassCast Exception) a tempo di esecuzione!**

# Introduzione

- Un controllo lasco dei tipi a tempo di compilazione ha queste conseguenze
  - ci costringe a fare un cast ogni volta che accediamo ad un elemento della coppia (ma questo non è l'aspetto peggiore)
  - rimanda a tempo di esecuzione alcuni errori che tutto sommato erano rilevabili anche a tempo di compilazione
- Vediamo come questi problemi sono stati affrontati e risolti con i *generics*

# Introduzione

- I generics sono uno strumento per scrivere classi (e metodi) *parametriche rispetto ad uno o più tipi*
- Nella definizione il codice viene scritto in maniera parametrica rispetto ad un *tipo generico*
- Nell'uso il tipo viene istanziato

# La classe generica Coppia

- La definizione di una classe generica prevede la dichiarazione del parametro di tipo racchiuso tra parentesi acute

```
public class Coppia<T> {  
    ...  
}
```

- In questo modo abbiamo detto che all'interno della classe **Coppia** ogni volta che usiamo il simbolo T stiamo indicato il tipo secondo il quale la classe è parametrica

# La classe generica Coppia

- Nella definizione di campi e metodi all'interno della classe T viene usato come una dichiarazione di tipo

```
public class Coppia<T> {  
    private T primo;  
    private T secondo;  
  
    public Coppia(T primo, T secondo) {  
        this.primo = primo;  
        this.secondo = secondo;  
    }  
    public T getPrimo() {  
        return this.primo;  
    }  
    ...  
}
```

# La classe generica Coppia

```
public class Coppia<T> {  
    private T primo;  
    private T secondo;  
  
    public Coppia(T primo, T secondo) {  
        this.primo = primo;  
        this.secondo = secondo;  
    }  
  
    public T getPrimo() {  
        return this.primo;  
    }  
  
    public T getSecondo() {  
        return this.secondo;  
    }  
  
    public void setPrimo(T primo) {  
        this.primo = primo;  
    }  
  
    public void setSecondo(T secondo) {  
        this.secondo = secondo;  
    }  
}
```

# Usare una classe generica

- Quando usiamo una classe generica, dobbiamo *istanziare il tipo*
- Ad esempio, usiamo la nostra classe generica **Coppia**, per gestire coppie di oggetti **Persona**

```
import omessi
```

```
public class CoppiaTest {  
    @Test  
    public void testDiCoppiaDiPersone() {  
        Coppia<Persona> coppia;  
        Persona p1 = new Persona("Stanlio");  
        Persona p2 = new Persona("Olio");  
        coppia = new Coppia<Persona>(p1, p2);  
        assertSame(p1, coppia.getPrimo());  
        assertSame(p2, coppia.getSecondo());  
    }  
}
```



# Usare una classe generica

- Vediamo la classe parametrica `Coppia<T>` istanziata su un altro tipo (`java.awt.Color`)

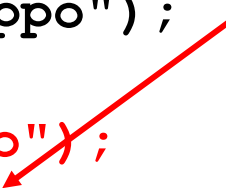
```
import java.awt.Color;
altri import omessi
public class CoppiaTest {
    @Test
    public void testDiCoppiaDiColori() {
        Coppia<Color> coppia;
        Color rosso = new Color(255,0,0);
        Color blue = new Color(0,0,255);
        coppia = new Coppia<Color>(rosso, blue);
        assertEquals(rosso, coppia.getPrimo());
        assertEquals(blue, coppia.getSecondo());
    }
}
```

# Controllo sui tipi

- Riconsideriamo il codice con cui abbiamo iniziato:

```
public class ProblemiConCoppiaTest {  
    @Test  
    public void testCheCompilaMaNonGira() {  
        Coppia<Persona> coppia = new Coppia<Persona>();  
        Persona p1 = new Persona("Pippo");  
        coppia.setPrimo(p1);  
        String p2 = new String("Pluto");  
        coppia.setSecondo(p2);  
        Persona persona = (Persona)coppia.getPrimo();  
        System.out.println(persona.getNome());  
    }  
}
```

Questa volta abbiamo un errore a tempo di compilazione!



# Tipo Formale - Tipo Attuale

- Non è difficile trovare una similitudine tra
  - il concetto di parametro formale/attuale inerente l'invocazione dei metodi
  - il concetto di tipo formale/attuale inerente la tipizzazione di classi generiche
- Attenzione a non dimenticare la prima delle differenze
  - il legame tra parametri formali/attuali è operato dalla JVM a tempo di esecuzione
  - il legame tra tipi formali/attuale è operato dal compilatore a tempo di compilazione

# Generics con più parametri

- È possibile definire classi, interfacce e metodi generici con più parametri di tipo
  - Sintatticamente, si separano i vari parametri con una virgola

```
public class Esempio<T, S> {...}
```

# Generics e tipi primitivi

- Non è possibile istanziare i tipi di una classe, interface o metodo generico con un tipo primitivo
- Se è necessario istanziare un tipo primitivo si usano le *classi wrapper*

```
public class TestCoppia {  
    @Test  
    public void testCheNonCompila() {  
        Coppia<int> coppia;           // ERRORE: non compila  
        int i1 = 100;  
        int i2 = 200;  
        coppia = new Coppia<int>(i1, i2); // ERRORE  
    }  
}
```

# Classi *Wrapper*

- Per ogni tipo primitivo esiste una classe *wrapper* che consente di "oggettificare" i dati memorizzati nei tipi primitivi
  - `int`                `-> Integer`
  - `double`           `-> Double`
  - `float`             `-> Float`
  - `char`              `-> Character`
  - `boolean`          `-> Boolean`

# Classi Wrapper

```
int i;  
i = 18;  
Integer iwrap = new Integer(i);  
...
```

*“incarto” il valore della  
variabile int in un  
oggetto Integer*



```
int value = iwrap.intValue();  
...
```

*"scarto" il valore*



# Classi *Wrapper*

- Le classi wrapper sono definite nel package `java.lang` (quindi non è necessario importarle)
- Per approfondimenti è sufficiente vedere la documentazione
- I metodi usati più frequentemente sono:
  - metodi `xxxValue()`
  - metodi `valueOf()` e `parseXxx()`
  - metodo `equals()`



# Generics e tipi primitivi

- Esempio:

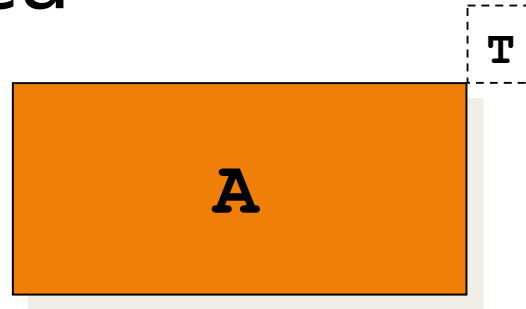
```
public class TestCoppia {  
    @Test  
    public void testCheCompila() {  
        Coppia<Integer> coppia;          // OK  
        Integer i1 = new Integer(100);  
        Integer i2 = new Integer(200);  
        coppia = new Coppia<Integer>(i1, i2); // OK  
    }  
}
```

# Generics: convenzioni sui nomi

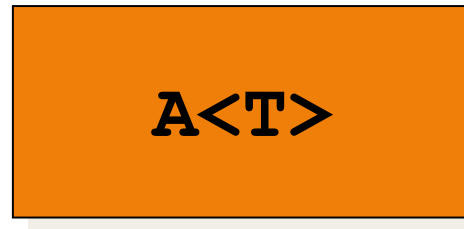
- `<T>`: Type (classe o interface)
- `<S>`: usato quando T è già in uso
- `<E>`: Elemento (molto usato nelle collezioni Java)
- `<K>`: chiave
- `<V>`: valore
- `<N>`: numero (una classe wrapper tra quelle usate per "oggettificare" i valori numerici)

# Generics: Rappresentazione diagrammatica

- Rappresentazione diagrammatica di una classe generica



- oppure



# Metodi generici

- È possibile definire anche metodi generici (cioè parametrici rispetto ad un tipo)
- Un metodo generico definisce i parametri (formali) di tipo nella segnatura del metodo, prima del tipo di ritorno



```
public <T> int mioMetodo(Coppia<T> c, T obj)
```

# Generics: wildcard e caratteri jolly

- Approfondiamo alcuni concetti avanzati relativi ai generics per mezzo di esempi (ricordiamo che in questo corso ci concentriamo sull'uso di classi generiche e non sulla loro progettazione)
  - Introduciamo altri metodi della classe **Coppia**
  - Definiamo una nuova classe, **Coppie**, che offre un insieme di metodi di utilità per lavorare su oggetti **Coppia**

# Generics: wildcard e caratteri jolly

- Aggiungiamo alla classe **Coppia** il metodo **addAll()**
  - aggiunge alla coppia corrente tutti gli elementi di un'altra coppia che viene passata come parametro
- La coppia che passiamo come parametro deve essere istanziata su un qualunque sottotipo degli oggetti della coppia corrente
- Questa particolarità si esprime con il carattere jolly **?** e con la parola chiave **extends**

# Generics: wildcard e caratteri jolly

- Questa particolarità si esprime con il carattere jolly ? e con la parola chiave **extends**
- Vediamone la segnatura:  

```
public void addAll(Coppia<? extends E> c)
```
- Cosa significa? Di che tipo deve essere il parametro?

# Generics: wildcard e caratteri jolly

`Coppia<? extends E>`

- Significa: un oggetto `Coppia` istanziato su `E` o su un qualsiasi sottotipo di `E`

- Esempio:

```
Coppia<Strumento> strumenti;
```

```
Coppia<Chitarra> chitarre;
```

```
...
```

```
strumenti.addAll(chitarre); // OK
```



# Generics: wildcard e caratteri jolly

```
public void addAll(Coppia<? extends T> coppia) {  
    this.setPrimo(coppia.getPrimo());  
    this.setSecondo(coppia.getSecondo());  
}
```

# Generics: wildcard e caratteri jolly

- Definiamo ora la classe **Coppie** (al plurale), che contiene metodi (generici) di utilità per manipolare oggetti **Coppia**.
- In particolare la classe **Coppie** offre i metodi:
  - **reverse()**  
prende come parametro una coppia e ne inverte gli elementi (il primo elemento diventa il secondo e viceversa)
  - **fill()**  
prende due parametri: una coppia e un elemento; riempie la coppia con l'elemento

# Metodi generici: esempio

```
public class Coppie {  
  
    public static <T> void reverse(Coppia<T> c) {  
        T tmp;  
  
        tmp = c.getPrimo();  
        c.setPrimo(c.getSecondo());  
        c.setSecondo(tmp);  
    }  
  
    ...  
  
}
```

# Generics: wildcard e caratteri jolly

- Il metodo `fill(Coppia , T)`
  - imposta entrambi gli elementi della coppia che viene passata come primo parametro, con un oggetto passato come secondo parametro
- E' un metodo parametrico. Il tipo del secondo parametro deve poter essere un qualunque sottotipo del tipo istanziato dalla coppia
- Si esprime così:

```
static <T> void fill(Coppia<? super T> coppia,  
                    T elemento)
```

# Generics: wildcard e caratteri jolly

`Coppia<? super T>`

- Significa: un oggetto `Coppia` istanziato su `T` o su un qualsiasi supertipo di `T`
- Esempio:

```
Coppia<Strumento> strumenti;
```

```
Chitarra fender;
```

```
...
```

```
Coppie.fill(stumenti, fender); // OK
```

# Generics: wildcard e caratteri jolly

```
public static <T> void fill(Coppia<? super T> coppia,  
                             T elemento) {  
    coppia.setPrimo(elemento);  
    coppia.setSecondo(elemento);  
}
```

# Riferimenti

- Un articolo che spiega i dettagli dei *generics*:

*<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>*

- Per sapere (quasi) tutto sui generics:

*<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>*

- Per scrivere test come in queste slides:

*<http://www.junit.org>*

# Esercizi

- Scrivere il codice della classe generica **Coppia<T>**
- Scrivere il codice della classe **Coppie**
- Scrivere classi di test per le classi **Coppia<T>** e **Coppie**