



# Metodologie di Programmazione

Lezione 27: I tipi generici (parte 1)

# Lezione 27:

## Sommario

---



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- Introduzione ai tipi generici
- Classi generiche
- Generici e collezioni
- Metodi generici
- Problematiche

# Che cosa accomuna queste due classi?

```
public class CoppiaDiInteri
{
    private int a, b;

    public CoppiaDiInteri(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    public int getPrimo() { return a; }
    public int getSecondo() { return b; }
}
```

```
public class CoppiaDiDouble
{
    private double a, b;

    public CoppiaDiDouble(double a, double b)
    {
        this.a = a;
        this.b = b;
    }

    public double getPrimo() { return a; }
    public double getSecondo() { return b; }
}
```

Stesso codice,  
ma tipi dei campi  
diversi!

Ci vorrebbe  
qualcosa di  
**generico**...

# Che cos'è un tipo generico?

- I tipi generici, in Java, sono un **modello di programmazione** che permette di definire, con una sola dichiarazione, un intero **insieme di metodi o di classi**
- Un meccanismo **MOLTO** potente
- Da usare con **consapevolezza**

Cup<T>

# Avete già utilizzato ampiamente i generici!



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- In tutte le collezioni Java!
- Creare istanze di classi con generici:

```
new ArrayList<String>();
```

- Dichiarare e assegnare variabili di tipi generici

```
List<String> listaDiStringhe = new ArrayList<String>();
```

- Dichiarare metodi che prendono in input tipi generici

```
public void metodo(List<String> lista)
{
    // ...
}
```

# Definire una classe generica

```
public class Coppia<T>
{
    private T a, b;

    public Coppia(T a, T b)
    {
        this.a = a;
        this.b = b;
    }

    public T getPrimo() { return a; }
    public T getSecondo() { return b; }
}
```

Definisce un tipo generico della classe

Usa il tipo generico della classe

- Per definire un tipo generico della classe, si utilizza la sintassi a **parentesi angolari** dopo il nome della classe con il tipo generico da utilizzare
- Da quel punto, si utilizza il tipo generico come un qualsiasi altro tipo di classe

# Utilizzare una classe generica

- Semplicemente si istanzia la classe specificando il tipo desiderato:

```
Coppia<Integer> ci = new Coppia<Integer>(10, 20);  
Coppia<Double> cd = new Coppia<Double>(10.5, 20.5);  
Coppia<String> cs = new Coppia<String>("abc", "def");  
Coppia<Object> co = new Coppia<Object>("abc", 20);
```

- ...e chi più ne ha più ne metta!



# Specificare più tipi generici di classe



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- Specifichiamo i tipi generici separati da virgola:

```
public class Coppia<T, S>
{
    private T a;
    private S b;

    public Coppia(T a, S b)
    {
        this.a = a;
        this.b = b;
    }

    public T getPrimo() { return a; }
    public S getSecondo() { return b; }
}
```

- Per convenzione i tipi generici sono chiamati con le lettere **T**, **S**, ecc. (**E** nel caso in cui siano elementi di una collection)

# Estendere le classi generiche

- Ovviamente è possibile estendere le classi generiche per creare classi più specifiche
- Ad esempio, una classe **Orario** può estendere la classe **Coppia**:

```
public class Orario extends Coppia<Integer, Integer>
{
    public Orario(Integer a, Integer b)
    {
        super(a, b);
    }
}
```

- O una classe **Data**:

```
public class Data extends Coppia<Integer, Coppia<Integer, Integer>>
{
    public Data(Integer giorno, Integer mese, Integer anno)
    {
        super(giorno, new Coppia<Integer, Integer>(mese, anno));
    }
}
```

# Esercizio:

## Lista Linkata generica



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- Progettare una classe generica **ListaLinkata**

```
public class ListaLinkata<T>
{
    private Elemento first;

    private class Elemento
    {
        private T val;
        private Elemento next;
        public Elemento(T val, Elemento next) { this.val = val; this.next = next; }
    }

    public void addFirst(T e)
    {
        first = new Elemento(e, first);
    }

    public void removeFirst()
    {
        first = first.next;
    }

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        Elemento e = first;
        while(e != null) { sb.append(e.val); if (e.next != null) sb.append(","); e = e.next; }
        return sb.toString();
    }
}
```

- Progettare una classe generica **Pila** implementata mediante lista di elementi (provate anche con l'array)
- La classe è costruita con la dimensione iniziale dell'array ed implementa i seguenti metodi:
  - **push**: inserisce un elemento in cima alla pila
  - **pop**: elimina e restituisce l'elemento in cima alla pila
  - **peek**: restituisce l'elemento in cima alla pila
  - **isEmpty**: restituisce true se la pila è vuota

# Esercizio: la classe Pila

```
public class Pila<T>
{
    static final public int INITIAL_SIZE = 5;
    private T[] a;
    private int k = -1;

    @SuppressWarnings("unchecked")
    public Pila()
    {
        a = (T[])new Object[INITIAL_SIZE];
    }

    public void push(T o)
    {
        if (k == a.length-1) a = Arrays.copyOf(a, a.length*2);
        a[++k] = o;
    }

    public T peek()
    {
        if (k == -1) return null;
        return a[k];
    }

    public T pop()
    {
        if (k == -1) return null;
        return a[k--];
    }
}
```

# Generici e collezioni (1)



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- Alcuni esempi prototipici:

```
public interface List<E>
{
    void add(E x);
    Iterator<E> iterator();
}
```

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
}
```

# Generici e collezioni (2)



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- L'utilizzo principe dei tipi generici è nelle **collezioni**
- Ad esempio, vediamo come è definita **ArrayList**:

```
public class ArrayList<E>  
    extends AbstractList<E>  
    implements List<E>, RandomAccess, Cloneable, Serializable
```

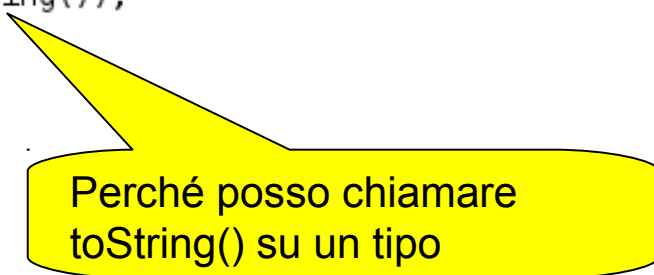
- Quando scriviamo:  

```
new ArrayList<String>();
```
- La classe viene trattata come: **public class**  
**ArrayList<String> extends AbstractList<String>**

# Definire un metodo generico

- Per definire un metodo generico **con proprio tipo generico** è necessario anteporre il **tipo generico** tra parentesi angolari al tipo di ritorno:

```
static public <T> void esamina(ArrayList<T> lista)
{
    for (T o : lista)
        System.out.println(o.toString());
}
```



Perché posso chiamare  
toString() su un tipo  
generico?



# Esercizio:

## Inverti lista e massimo



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- Implementare i seguenti due metodi generici statici:
  - **inverti**, che data in input una lista di elementi di tipo generico, restituisca un'altra lista con gli elementi in ordine invertito
  - **max**, che data in input una lista di elementi di tipo generico, ne restituisca il valore massimo
- **Nota:** per il secondo metodo è necessario utilizzare il costrutto `<T extends InterfacciaOClasse>`, che impone un vincolo sul supertipo di T

# Trova le differenze

```
static public void esamina(ArrayList<Frutto> frutti)
{
    // ...
}
```

Accetta un ArrayList<Arancia>?  
Perché?

```
static public <T extends Frutto> void esamina(ArrayList<T> frutti)
{
    // ...
}
```

Indica che **T** può essere di tipo  
Frutto **OPPURE** un suo sottotipo

# Per rendere “sicura” la collection a tempo di compilazione (1)



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- Se fosse permesso fare l'**upcasting** tra tipi generici si avrebbero situazioni imprevedibili:

```
public class ViolenzaSuUnArrayList
{
    public static void main(String[] args)
    {
        ArrayList<Mela> mele = new ArrayList<Mela>();

        prendiFrutta(mele);

        // qui avrei una lista di mele con una pera!!!
        System.out.println(mele.toString());
    }

    public static void prendiFrutta(ArrayList<Frutto> frutti)
    {
        frutti.add(new Pera());
    }
}
```

- Il **controllo di consistenza di tipo** viene effettuato solo a tempo di compilazione

# Per rendere “sicura” la collection a tempo di compilazione (2)



SAPIENZA  
UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA

- E' possibile permettere il passaggio di sottotipi di Frutto utilizzando la sintassi **T extends** Frutto:

```
public class ViolenzaSuUnArrayList
{
    public static void main(String[] args)
    {
        ArrayList<Mela> mele = new ArrayList<Mela>();

        prendiFrutta(mele);

        // qui avrei una lista di mele con una pera!!!
        System.out.println(mele.toString());
    }

    public static <T extends Frutto> void prendiFrutta(ArrayList<T> frutti)
    {
        frutti.add(new Pera());
    }
}
```

- Tuttavia non è possibile “aggiungere” una pera a un elenco di mele...

# Differenze tra array e collection

- L'upcasting è invece possibile **\*a tempo di compilazione\*** con gli array:

```
public class ViolenzaSuUnArray
{
    public static void main(String[] args)
    {
        Mela[] mele = new Mela[] { new Mela(), new Mela() };

        prendiFrutta(mele);

        // qui ho un array di mele con una pera!!!
        System.out.println(Arrays.toString(mele));
    }

    public static void prendiFrutta(Frutto[] frutti)
    {
        frutti[1] = new Pera();
    }
}
```

- A tempo di esecuzione otteniamo però

```
Exception in thread "main" java.lang.ArrayStoreException: Pera
    at ViolenzaSuUnArray.prendiFrutta(ViolenzaSuUnArray.java:17)
    at ViolenzaSuUnArray.main(ViolenzaSuUnArray.java:9)
```