

# Generic Programming

**Prof. Michele Loreti**

**Programmazione Avanzata**

*Corso di Laurea in Informatica (L31)*

*Scuola di Scienze e Tecnologie*

# Generic class

A **generic class** is a class with one or more **type parameter**.

# Generic class

A **generic class** is a class with one or more **type parameter**.

As a simple example consider the class storing **key/value** pairs:

# Generic class

A **generic class** is a class with one or more **type parameter**.

As a simple example consider the class storing **key/value** pairs:

```
public class Entry<K,V> {  
    private K key;  
    private V value;  
    public Entry(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
    public K getKey() { return this.key; }  
    public V getValye() { return this.value; }  
}
```

## Generic class

When the class is **instantiated**, we have to provide the specific type we want to use:

```
Entry<String , Integer> entry = new Entry<String , Integer>("
    Harry" , 17);
```

## Generic class

When the class is **instantiated**, we have to provide the specific type we want to use:

```
Entry<String , Integer> entry = new Entry<String , Integer>("Harry" , 17);
```

The type parameters can be omitted in the constructor (exact values are inferred by the context):

```
Entry<String , Integer> entry = new Entry<>("Harry" , 17);
```

## Generic class

When the class is **instantiated**, we have to provide the specific type we want to use:

```
Entry<String , Integer> entry = new Entry<String , Integer>("Harry" , 17);
```

The type parameters can be omitted in the constructor (exact values are inferred by the context):

```
Entry<String , Integer> entry = new Entry<>("Harry" , 17);
```

The empty pair of angle brackets `<>` is mandatory! This is sometime named **diamond**

Example: `ArrayList<T>`

Generic classes are useful to design **data structures**.



Example: `ArrayList<T>`

Generic classes are useful to design **data structures**.

One of this is `ArrayList<T>` that implements a generic **list** containing elements of type `T`.

## Example: `ArrayList<T>`

Generic classes are useful to design **data structures**.

One of this is `ArrayList<T>` that implements a generic **list** containing elements of type `T`.

Some of the methods provided by this class `ArrayList<T>` are:

- **boolean** `add(T t)`: Appends the specified element to the end of this list.
- **void** `clear()`: Removes all of the elements from this list.
- **boolean** `contains(Object o)`: Returns **true** if this list contains the specified element.
- `T` `get(int i)`: Returns the element at the specified position in this list.
- **int** `size()`: Returns the number of elements in this list.
- `T` `remove(int i)`: Removes the element at the specified position and shifts any subsequent elements to the left.

## Generic Methods. . .

Just like a generic class, a **generic method** is a method with a type parameter.

## Generic Methods...

Just like a generic class, a **generic method** is a method with a type parameter.

A **generic method** can be defined either in a generic class or in a standard class:

```
public class Arrays {  
    public static <T> void swap( T[] array , int i , int j ) {  
        T temp = array[i];  
        array[i] = array[j];  
        array[j] = temp;  
    }  
}
```

## Generic Methods. . .

Just like a generic class, a **generic method** is a method with a type parameter.

A **generic method** can be defined either in a generic class or in a standard class:

```
public class Arrays {  
    public static <T> void swap( T[] array , int i , int j ) {  
        T temp = array[i];  
        array[i] = array[j];  
        array[j] = temp;  
    }  
}
```

The **type parameter** (<T>) is placed after the modifier and before the return type.

## Generic methods

When calling the generic method, the type parameter can be omitted:

```
String [] friends = new String [] { "Ron" , "Hermione" , "Nevil" };  
Arrays.swap( friends , 0 , 1 );
```

## Generic methods

When calling the generic method, the type parameter can be omitted:

```
String [] friends = new String [] { "Ron" , " Hermione" , " Nevil" };  
Arrays.swap( friends , 0 , 1 );
```

Sometime it is useful to explicitly pass the type parameter to the method:

```
Arrays.<String>swap( friends ,0 ,1 );
```

## Generic methods

When calling the generic method, the type parameter can be omitted:

```
String [] friends = new String [] { "Ron" , " Hermione" , " Nevil" };  
Arrays.swap( friends , 0 , 1 );
```

Sometime it is useful to explicitly pass the type parameter to the method:

```
Arrays.<String>swap( friends ,0 ,1 );
```

This is useful if we want to control the error message (and in general the type-inference).



## Example...

Consider the following variant:

```
public class Arrays {  
    public static <T> T[] swap( int i , int j , T ... values) {  
        T temp = values[i];  
        values[i] = values[j];  
        values[j] = temp;  
        return values;  
    }  
}
```

## Example...

Consider the following variant:

```
public class Arrays {  
    public static <T> T[] swap( int i , int j , T ... values ) {  
        T temp = values[i];  
        values[i] = values[j];  
        values[j] = temp;  
        return values;  
    }  
}
```

Let us consider the following code:

```
Integer[] result = Arrays.swap( 0 , 1 , 1 , 2 , 3 );
```

## Example...

Consider the following variant:

```
public class Arrays {  
    public static <T> T[] swap( int i , int j , T ... values ) {  
        T temp = values[i];  
        values[i] = values[j];  
        values[j] = temp;  
        return values;  
    }  
}
```

Let us consider the following code:

```
Integer[] result = Arrays.swap( 0 , 1 , 1 , 2 , 3 );
```

**Is this correct?**

## Example...

Consider the following variant:

```
public class Arrays {  
    public static <T> T[] swap( int i , int j , T ... values ) {  
        T temp = values[i];  
        values[i] = values[j];  
        values[j] = temp;  
        return values;  
    }  
}
```

Let us consider the following variants:

```
Double [] result = Arrays.swap( 0 , 1 , 1.5 , 2 , 3 );
```

```
Double [] result = Arrays.<Double>swap( 0 , 1 , 1.5 , 2 , 3 );
```

## Example...

Consider the following variant:

```
public class Arrays {  
    public static <T> T[] swap( int i , int j , T ... values ) {  
        T temp = values[i];  
        values[i] = values[j];  
        values[j] = temp;  
        return values;  
    }  
}
```

Let us consider the following variants:

```
Double [] result = Arrays.swap( 0 , 1 , 1.5 , 2 , 3 );
```

```
Double [] result = Arrays.<Double>swap( 0 , 1 , 1.5 , 2 , 3 );
```

**How can we solve this problem?**

## Type Bounds. . .

Sometime, the type parameters of a generic class or method must need to fulfil certain requirements.

## Type Bounds. . .

Sometime, the type parameters of a generic class or method must need to fulfil certain requirements.

In this case we can specify a **type bound** to require that the actual type extends certain classes or implements certain interfaces.

## Type Bounds. . .

Sometime, the type parameters of a generic class or method must need to fulfil certain requirements.

In this case we can specify a **type bound** to require that the actual type extends certain classes or implements certain interfaces.

```
public static <T extends AutoCloseable>
    void closeAll( ArrayList<T> elems ) throws Exception {
        for (T elem: elems) elem.close();
    }
```



## Type Bounds. . .

Sometime, the type parameters of a generic class or method must need to fulfil certain requirements.

In this case we can specify a **type bound** to require that the actual type extends certain classes or implements certain interfaces.

```
public static <T extends AutoCloseable>
    void closeAll( ArrayList<T> elems ) throws Exception {
        for (T elem: elems) elem.close();
    }
```

A type parameter can have multiple bounds:

```
T extends Runnable & AutoCloseable
```

## Array and subtyping. . .

Let us consider the class `Employee` and its subclass `Manager`.

## Array and subtyping. . .

Let us consider the class `Employee` and its subclass `Manager`.

Is a `Manager[]` a subtype of `Employee[]`?

## Array and subtyping. . .

Let us consider the class `Employee` and its subclass `Manager`.

Is a `Manager[]` a subtype of `Employee[]`?

Is **sound** to use a `Manager[]` where a `Employee[]` is expected?

## Array and subtyping. . .

Let us consider the class `Employee` and its subclass `Manager`.

Is a `Manager[]` a subtype of `Employee[]`?

Is **sound** to use a `Manager[]` where a `Employee[]` is expected?

The answer is **NO!**

## Array and subtyping...

Let us consider the class `Employee` and its subclass `Manager`.

Is a `Manager[]` a subtype of `Employee[]`?

Is **sound** to use a `Manager[]` where a `Employee[]` is expected?

The answer is **NO!**

Let us consider the following portion of (pseudo)-code:

```
public void set( Employee[] staff , int i , Employee e ) {  
    staff[i] = e;  
}  
...  
Manager[] mngrs = ...  
set( mngrs , 0 , new Employee() );  
mngrs[0].callManagerSpecificMethod();
```

## Generic classes and Subtyping. . .

Similarly, in general we cannot use an `ArrayList<Manager>` where an `ArrayList<Employee>` is expected.

## Generic classes and Subtyping. . .

Similarly, in general we cannot use an `ArrayList<Manager>` where an `ArrayList<Employee>` is expected.

However, it could be convenient to use a mechanism where we can define a method where **a list of elements extending a base class is expected**.



## Array Covariance...

In Java, for reason related to **usability**, we can use an array of a subclass where an array of a superclass is expected:

```
public static void process( Employee[] staff ) { ... }
```

## Array Covariance...

In Java, for reason related to **usability**, we can use an array of a subclass where an array of a superclass is expected:

```
public static void process( Employee[] staff ) { ... }
```

We can pass a `Manager[]` array since `Manager[]`!

## Array Covariance...

In Java, for reason related to **usability**, we can use an array of a subclass where an array of a superclass is expected:

```
public static void process( Employee[] staff ) { ... }
```

We can pass a `Manager[]` array since `Manager[]`!

This behaviour is called **covariance**.

## Array Covariance...

In Java, for reason related to **usability**, we can use an array of a subclass where an array of a superclass is expected:

```
public static void process( Employee[] staff ) { ... }
```

We can pass a `Manager[]` array since `Manager[]`!

This behaviour is called **covariance**.

**Warning:** this is **unsound** and an `ArrayStoreException` is thrown if a **wrong assignment** is performed.

## Array Covariance...

In Java, for reason related to **usability**, we can use an array of a subclass where an array of a superclass is expected:

```
public static void process( Employee[] staff ) { ... }
```

We can pass a `Manager[]` array since `Manager[]`!

This behaviour is called **covariance**.

**Warning:** this is **unsound** and an `ArrayStoreException` is thrown if a **wrong assignment** is performed.

This mechanism cannot be used for generic classes!

# Subtype wildcards

To partially solve this problem we can use **wildcards**:

# Subtype wildcards

To partially solve this problem we can use **wildcards**:

```
public void printNames(  
    ArrayList<? extends Employee> staff  
    ) {  
    for ( int i=0 ; i<staff.size() ; i++ ) {  
        System.out.println(staff.get(i).getName());  
    }  
}
```

## Subtype wildcards

To partially solve this problem we can use **wildcards**:

```
public void printNames(  
    ArrayList<? extends Employee> staff  
    ) {  
    for ( int i=0 ; i<staff.size() ; i++ ) {  
        System.out.println( staff.get(i).getName() );  
    }  
}
```

The wild card ? **extends** Employee indicates any type extending Employee.



## Subtype wildcards

To partially solve this problem we can use **wildcards**:

```
public void printNames(  
    ArrayList<? extends Employee> staff  
) {  
    for ( int i=0 ; i<staff.size() ; i++ ) {  
        System.out.println( staff.get(i).getName() );  
    }  
}
```

The wild card ? **extends** Employee indicates any type extending Employee.

We can pass to this method either an `ArrayList<Employee>` or an `ArrayList<Manager>`.

# Subtype wildcards

How can we use an instance of `ArrayList<? extends Employee>`? (say it staff)

## Subtype wildcards

How can we use an instance of `ArrayList<? extends Employee>`? (say it `staff`)

**Question:** Is the following code valid?

```
Employee e = staff.get(i);
```

## Subtype wildcards

How can we use an instance of `ArrayList<? extends Employee>`? (say it `staff`)

**Question:** Is the following code valid?

```
Employee e = staff.get(i);
```

**Answer:** Yes!

## Subtype wildcards

How can we use an instance of `ArrayList<? extends Employee>`? (say it `staff`)

**Question:** Is the following code valid?

```
Employee e = staff.get(i);
```

**Answer:** Yes!

**Question:** Is the following code valid?

```
staff.add(new Employee(...));
```

## Subtype wildcards

How can we use an instance of `ArrayList<? extends Employee>`? (say it `staff`)

**Question:** Is the following code valid?

```
Employee e = staff.get(i);
```

**Answer:** Yes!

**Question:** Is the following code valid?

```
staff.add(new Employee (...));
```

**Answer:** No!

# Supertype Wildcards

We can use wildcards to ask that a given parameter is a supertype of another type.

# Supertype Wildcards

We can use wildcards to ask that a given parameter is a supertype of another type.

Let us consider the following code:

```
public void printAll(
    ArrayList<? extends Employee> staff ,
    Predicate<Employee> filter
) {
    for ( int i=0 ; i<staff.size() ; i++ ) {
        if ( filter.test(e) ) {
            System.out.println( staff.get(i).getName() );
        }
    }
}
```



# Supertype Wildcards

A typical use of method `printAll` is:

```
printAll( employees , e -> e.getSalary() > 10000);
```

# Supertype Wildcards

A typical use of method `printAll` is:

```
printAll( employees , e -> e.getSalary() > 10000);
```

Another example is:

```
printAll( employees , e -> e.toString().length() % 2 == 0);
```

# Supertype Wildcards

A typical use of method `printAll` is:

```
printAll( employees , e -> e.getSalary() > 10000);
```

Another example is:

```
printAll( employees , e -> e.toString().length() % 2 == 0);
```

**Warning:** the last one does not work!

## Supertype Wildcards

A typical use of method `printAll` is:

```
printAll( employees , e -> e.getSalary() > 10000);
```

Another example is:

```
printAll( employees , e -> e.toString().length() % 2 == 0);
```

**Warning:** the last one does not work!

The problem is that

```
e -> e.toString().length() % 2 == 0
```

has type `Predicate<Object>`.

# Supertype Wildcards

Supertype wildcards can be used:

```
public void printAll(  
    ArrayList<? extends Employee> staff ,  
    Predicate<? super Employee> filter  
    ) {  
    for ( int i=0 ; i<staff.size() ; i++ ) {  
        if (filter.test(e)) {  
            System.out.println(staff.get(i).getName());  
        }  
    }  
}
```

# Supertype Wildcards

Supertype wildcards can be used:

```
public void printAll(  
    ArrayList<? extends Employee> staff ,  
    Predicate<? super Employee> filter  
    ) {  
    for ( int i=0 ; i<staff.size() ; i++ ) {  
        if (filter.test(e)) {  
            System.out.println(staff.get(i).getName());  
        }  
    }  
}
```

In that case we can use any predicate defined on any superclass of Employee.

# Wildcards and Type Variables

We can generalise method `printAll` to work with any type `T`:

# Wildcards and Type Variables

We can generalise method `printAll` to work with any type `T`:

```
public <T> void printAll(  
    ArrayList<T> staff ,  
    Predicate<? super T> filter  
) {  
    for ( int i=0 ; i<staff.size() ; i++ ) {  
        if (filter.test(e)) {  
            System.out.println(staff.get(i).getName());  
        }  
    }  
}
```



# Unbounded Wildcards

When we have limited information about the exact type we are receiving, unbounded wildcards can be used:

# Unbounded Wildcards

When we have limited information about the exact type we are receiving, unbounded wildcards can be used:

```
public static boolean hasNull( ArrayList<?> elements ) {  
    for( int i=0 ; i<elements.size() ; i++ ) {  
        if (elements.get(i) == null) {  
            return true;  
        }  
    }  
    return false;  
}
```

# Type Erasure

When a generic class is **compiled** all the generic informations are lost and the class is translated into a **raw** type.

# Type Erasure

When a generic class is **compiled** all the generic informations are lost and the class is translated into a **raw** type.

For instance, the class `Entry` considered before is handled as:

# Type Erasure

When a generic class is **compiled** all the generic informations are lost and the class is translated into a **raw** type.

For instance, the class `Entry` considered before is handled as:

```
public class Entry {  
    private Object key;  
    private Object value;  
    public Entry(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
    public Object getKey() { return this.key; }  
    public Object getValye() { return this.value; }  
}
```

- No primitive type arguments;

# Restrictions on Generics

- No primitive type arguments;
- At Runtime, all type are raws...

```
if (a instanceof ArrayList<String>) ... //WRONG!
```

# Restrictions on Generics

- No primitive type arguments;
- At Runtime, all type are raws...

```
if (a instanceof ArrayList<String>) ... //WRONG!
```

- Type variables cannot be instantiated;



# Restrictions on Generics

- No primitive type arguments;
- At Runtime, all type are raws...

```
if (a instanceof ArrayList<String>) ... //WRONG!
```

- Type variables cannot be instantiated;
- Arrays of parametrised type cannot be constructed;

# Restrictions on Generics

- No primitive type arguments;
- At Runtime, all type are raws...

```
if (a instanceof ArrayList<String>) ... //WRONG!
```

- Type variables cannot be instantiated;
- Arrays of parametrised type cannot be constructed;
- Class type variables cannot be used in static context.

To be continued...