# Interfaces and Lambda Expressions

**Prof. Michele Loreti**

**Programmazione Avanzata**

*Corso di Laurea in Informatica (L31)*

*Scuola di Scienze e Tecnologie*

# Interfaces and Lambda expressions

Interfaces are a key feature of object-oriented programming...

# Interfaces and Lambda expressions

Interfaces are a key feature of object-oriented programming...

... they specify what should be done

# Interfaces and Lambda expressions

Interfaces are a key feature of object-oriented programming...

... they specify what should be done

... without having to provide an implementation.

# Interfaces and Lambda expressions

Interfaces are a key feature of object-oriented programming...

... they specify what should be done

... without having to provide an implementation.

Recently functional programming has risen in importance because it is well suited for concurrent and event-driven programming.

# Interfaces and Lambda expressions

Interfaces are a key feature of object-oriented programming...

... they specify what should be done

... without having to provide an implementation.

Recently functional programming has risen in importance because it is well suited for concurrent and event-driven programming.

`Java` integrates aspects of functional programming in the object-oriented approach.

# Interaces

An interface is a mechanism for spelling out a contract between two parties:

# Interaces

An interface is a mechanism for spelling out a contract between two parties:

- the supplier of a service;

# Interaces

An interface is a mechanism for spelling out a contract between two parties:

- the supplier of a service;
- the classes that want their objects to be usable with the service.

# Interaces

An interface is a mechanism for spelling out a contract between two parties:

- the supplier of a service;
- the classes that want their objects to be usable with the service.

**Example:** Consider a service that works on sequences of integers, reporting the average of the first *n* values:

```
public static double average(IntSequence seq, int n)
```

# Interaces

An interface is a mechanism for spelling out a contract between two parties:

- the supplier of a service;
- the classes that want their objects to be usable with the service.

**Example:** Consider a service that works on sequences of integers, reporting the average of the first $n$ values:

```
public static double average(IntSequence seq, int n)
```

Such sequence can take many forms!

# Interfaces at work

For a sequence of integers we have to consider, at least, two methods:

# Interfaces at work

For a sequence of integers we have to consider, at least, two methods:

- test if there is another element in the list;

# Interfaces at work

For a sequence of integers we have to consider, at least, two methods:

- test if there is another element in the list;
- get the next element.

# Interfaces at work

For a sequence of integers we have to consider, at least, two methods:

- test if there is another element in the list;
- get the next element.

These informal descriptions allow us to derive the following interface:

```
public interface IntSequence {
  boolean hasNext();
  int next();
}
```

# Interfaces at work

This interface allow us to implement method average:

# Interfaces at work

This interface allow us to implement method average:

```
public static double average(IntSequence seq, int n) {
  int count = 0;
  double sum = 0;
  while (seq.hasNext() && count<n) {
    count++;
    sum += seq.next();
  }
  return count == 0 ? 0 : sum/count;
}
```

# Interfaces at work

This interface allow us to implement method average:

```
public static double average(IntSequence seq, int n) {
  int count = 0;
  double sum = 0;
  while (seq.hasNext() && count<n) {
    count++;
    sum += seq.next();
  }
  return count == 0 ? 0 : sum/count;
}
```

**We don't know the exact implementation of IntSequence!**

# Implementing an Interface

The classes that want to be usable with the average method must implement the IntSequence interface

# Implementing an Interface

The classes that want to be usable with the average method must implement the IntSequence interface

```
public class SquareSequence implements IntSequence {

  private int i=0;

  public boolean hasNext() {
    return true;
  }

  public int next() {
     i++;
    return i*i;
  }

}
```

# Example: Fibonacci Sequence

```
public class FibonacciSequence implements IntSequence {

  private int a = 1;
  private int b = 1;

  public boolean hasNext() {
    return true;
  }

  public int next() {
    int res = a;
    a = b;
    b = res+a;
    return res;
  }

}
```

# Example: Digit Sequence

```
public class DigitSequence implements IntSequence {
  private int number;
  public DigitSequence( int number ) {
    this.number = number;
  }
  public boolean hasNext() {
    return this.number != 0;
  }
  public int next() {
    int result = this.number % 10;
    this.number /= 10;
    return result;
  }
  public int rest() {
    return this.number;
  }
}
```

# Interface type

Let us consider the following portion of code:

```
IntSequence seq = new DigitSequence(19876);
double avg = Util.average(seq, 100);
```

# Interface type

Let us consider the following portion of code:

```
IntSequence seq = new DigitSequence(19876);
double avg = Util.average(seq, 100);
```

The type of variable seq is IntSequence!

# Interface type

Let us consider the following portion of code:

```
IntSequence seq = new DigitSequence(19876);
double avg = Util.average(seq, 100);
```

The type of variable seq is IntSequence!

**Terminology:** S is a supertype of T (the subtype) when any value of T can be assigned to a variable of type S (without cast).

# Interface type

Let us consider the following portion of code:

```
IntSequence seq = new DigitSequence(19876);
double avg = Util.average(seq, 100);
```

The type of variable seq is IntSequence!

**Terminology:** S is a supertype of T (the subtype) when any value of T can be assigned to a variable of type S (without cast).

Occasionally, you may need to convert a variable of supertype in a specific subtype with the cast operation!

# Interface type

Let us consider the following portion of code:

```
IntSequence seq = new DigitSequence(19876);
double avg = Util.average(seq, 100);
```

The type of variable seq is IntSequence!

**Terminology:** S is a supertype of T (the subtype) when any value of T can be assigned to a variable of type S (without cast).

Occasionally, you may need to convert a variable of supertype in a specific subtype with the cast operation! **It is dangerous and in general Wrong!**

# Interface type

Let us consider the following portion of code:

```
IntSequence seq = new DigitSequence(19876);
double avg = Util.average(seq, 100);
```

The type of variable seq is IntSequence!

**Terminology:** S is a supertype of T (the subtype) when any value of T can be assigned to a variable of type S (without cast).

Occasionally, you may need to convert a variable of supertype in a specific subtype with the cast operation! **It is dangerous and in general Wrong!**

If **really needed** use instanceof to checlk the correctness of the operation.

# Extending Interfaces

An interface can extend another, requiring or providing additional methods on top of the original ones.

# Extending Interfaces

An interface can extend another, requiring or providing additional methods on top of the original ones.

**Example:**

```
public interface Closeable {
  void close();
}
```

# Extending Interfaces

An interface can extend another, requiring or providing additional methods on top of the original ones.

**Example:**

```
public interface Closeable {
  void close();
}
```

```
public interface Channel extends Closeable {
  boolean isOpen();
}
```

# Implementing Multiple Interfaces

A class can implement any number of interfaces.

# Implementing Multiple Interfaces

A class can implement any number of interfaces.

All the methods defined in all the implemented interfaces must be provided.

# Implementing Multiple Interfaces

A class can implement any number of interfaces.

All the methods defined in all the implemented interfaces must be provided.

**Warning:** handle possible clash of names!

# Constants

An interface can contain definitions of constants.

# Constants

An interface can contain definitions of constants.

Each field defined in the interface is automatically considered as public static final .

# Constants

An interface can contain definitions of constants.

Each field defined in the interface is automatically considered as `public static final`.

These fields can be accessed via the standard `.` notation.

# Constants

An interface can contain definitions of constants.

Each field defined in the interface is automatically considered as public static final .

These fields can be accessed via the standard . notation.

**Example:**

```
SwingConstants.NORTH
```

# Methods in the Interface

In the previous versions of `Java` all methods in an interface was abstract (that is without body).

# Methods in the Interface

In the previous versions of `Java` all methods in an interface was abstract (that is without body).

In the latest version of `Java` an interface can contain three kinds of methods with a concrete implementation:

- static;

# Methods in the Interface

In the previous versions of `Java` all methods in an interface was abstract (that is without body).

In the latest version of `Java` an interface can contain three kinds of methods with a concrete implementation:

- static;
- default;

# Methods in the Interface

In the previous versions of `Java` all methods in an interface was abstract (that is without body).

In the latest version of `Java` an interface can contain three kinds of methods with a concrete implementation:

- static;
- default;
- private methods.

# Static Methods

It may be convenient to equip interfaces with static methods (like the factory methods) that provide generic functionalities for a given type.

```
public interface IntSequence {

   ...

   static IntSequence digitsOf(int n) {
     return new DigitSequence(n);
   }

}
```

# Default Methods

Starting from Java 1.9, we can provide a default implementation for any interface method:

# Default Methods

Starting from Java 1.9, we can provide a default implementation for any interface method:

```
public interface IntSequence {
  default boolean hasNext() { return true; }
  int next();
  ...
}
```

# Default Methods

Starting from Java 1.9, we can provide a default implementation for any interface method:

```
public interface IntSequence {
  default boolean hasNext() { return true; }
  int next();
  ...
}
```

A class implementing an interface can choose to override or not the default implementation.

# Default Methods

Starting from Java 1.9, we can provide a default implementation for any interface method:

```
public interface IntSequence {
  default boolean hasNext() { return true; }
  int next();
  ...
}
```

A class implementing an interface can choose to override or not the default implementation.

The use of default methods is particularly useful for interface evolutions!

# Resolving default methods conflict

Let us consider the following interfaces:

```
public interface Person {
  String getName();
  default int getId() { return 0; }
}

public interface Identified {
  default int getId() { return Math.abs(hashCode()); }
}
```

# Resolving default methods conflict

Let us consider the following interfaces:

```
public interface Person {
  String getName();
  default int getId() { return 0; }
}

public interface Identified {
  default int getId() { return Math.abs(hashCode()); }
}
```

Consider now the class Employee defined as follows:

```
public class Employee implements Person, Identified {
  ...
}
```

# Resolving default methods conflict

Let us consider the following interfaces:

```
public interface Person {
  String getName();
  default int getId() { return 0; }
}

public interface Identified {
  default int getId() { return Math.abs(hashCode()); }
}
```

Consider now the class Employee defined as follows:

```
public class Employee implements Person, Identified {
  ...
}
```

**There is a conflict that we have to resolve by providing an implementation of getId.**

# Private methods

An interface can also contain private methods.

# Private methods

An interface can also contain private methods.

These can be static or not, and can be used only by other methods defined in the interface.

# Private methods

An interface can also contain private methods.

These can be static or not, and can be used only by other methods defined in the interface.

These private methods typically implement utility features and their use should be limited.

# Examples...

Many interfaces are provided within `Java` API:

- Comparable<T>;

# Examples...

Many interfaces are provided within `Java` API:

- Comparable<T>;
- Comparator<T>;

# Examples...

Many interfaces are provided within `Java` API:

- Comparable<T>;
- Comparator<T>;
- Runnable;

# Examples...

Many interfaces are provided within `Java` API:

- Comparable<T>;
- Comparator<T>;
- Runnable;
- EventHandler<T>.

# Lambda Expressions

A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple time.

# Lambda Expressions

A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple time. It can be defined as a sequence of parameters followed by a an expression

```
(String first, String second)->first.length()-second.length()
```

# Lambda Expressions

A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple time. It can be defined as a sequence of parameters followed by a an expression

```
(String first,String second)->first.length()-second.length()
```

or a block:

```
(String first,String second) -> {
  int difference = first.length()-second.length();
  if (difference<0) return -1;
  else if (difference>0) return 1;
  else return 0;
}
```

# Functional Interfaces

Lambda expressions are compatible with Functional Interfaces.

# Functional Interfaces

Lambda expressions are compatible with Functional Interfaces.

These are interfaces that contains a single abstract method.

# Functional Interfaces

Lambda expressions are compatible with Functional Interfaces.

These are interfaces that contains a single abstract method.

We can use a lambda expression (with the appropriate type) when a functional interface is expected:

```
Arrays.sort(anArray, (x,y) -> x.length()-y.length());
```

# Functional Interfaces

Lambda expressions are compatible with Functional Interfaces.

These are interfaces that contains a single abstract method.

We can use a lambda expression (with the appropriate type) when a functional interface is expected:

```
Arrays.sort(anArray, (x,y) -> x.length()-y.length());
```

The type of parameters can be inferred!

# Method References

Suppose that we want to sort strings regardless of letter case. We could call:

```
Arrays.sort( strings , (x,y) -> x.compareToIgnoreCase(y) );
```

# Method References

Suppose that we want to sort strings regardless of letter case. We could call:

```
Arrays.sort( strings , (x,y) -> x.compareToIgnoreCase(y) );
```

Alternatively, we can pass directly the method reference:

```
Arrays.sort( strings , String::compareToIgnoreCase );
```

# Method References

Suppose that we want to sort strings regardless of letter case. We could call:

```
Arrays.sort( strings , (x,y) -> x.compareToIgnoreCase(y) );
```

Alternatively, we can pass directly the method reference:

```
Arrays.sort( strings , String::compareToIgnoreCase );
```

There are many examples of use:

- list .remove(Objects:: isNull )

# Method References

Suppose that we want to sort strings regardless of letter case. We could call:

```
Arrays.sort( strings , (x,y) -> x.compareToIgnoreCase(y) );
```

Alternatively, we can pass directly the method reference:

```
Arrays.sort( strings , String::compareToIgnoreCase );
```

There are many examples of use:

- list .remove(Objects:: isNull )
- list .forEach(System.out:: println )

# Method References

There are three variations for method references:

- Class :: instanceMethod
- Class :: staticMethod
- object :: instanceMethod
- Class :: new

# Scope of a Lambda Expression

A lambda expression has the same scope as a nested block

# Scope of a Lambda Expression

A lambda expression has the same scope as a nested block

This means that a lambda expression can access to all the names defined in the enclosing scope.

# Scope of a Lambda Expression

A lambda expression has the same scope as a nested block

This means that a lambda expression can access to all the names defined in the enclosing scope.

**Example:**

```
public class AClass {
  private int value = 0;

  public void setValue( int value ) {
    this.value = value;
  }

  public Function<Integer, Integer> getLambda() {
    return (x) -> this.value+x;
  }
}
```

**To be continued...**