3TC36: Object-Oriented Programming in Java

Programming interfaces

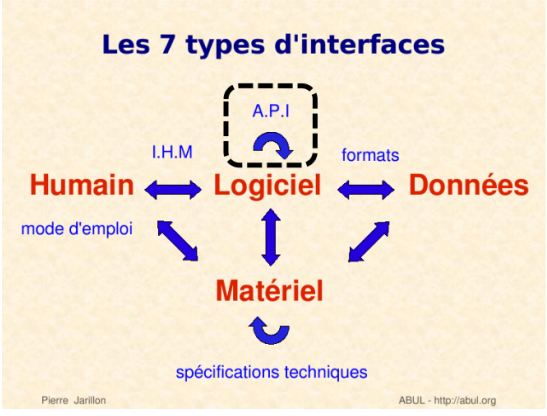
Dominique Blouin
dominique.blouin@telecom-paris.fr
Le March 10, 2026

Learning objectives

- Concept of programming interface
- Composition (inheritance) of interfaces
- Constant and marker interfaces
- Main interfaces of the JDK collections
- Programming to the interface

What is an interface?

- **"interface"** (Robert dictionary):
  - In general: Common boundary between two systems, two sets, or two devices.
  - In computing: A device enabling **communication** between two **elements** of a **computer system**.
    - Example: A **graphical interface** allows communication between a **user** and a **computer system**.
  - Figurative meaning: Relationship.
    - Example: Managing **the interface** with a client.



- For this course, we focus on **software-to-software** interfaces.
- Also called APIs:
  - Application Programming Interface or Interface for application programming

Recap

- Objects are **entities** that communicate by sending **messages**.
- Objects contain values called **attributes**.
- Among the attributes, some are **primitive** types (numbers, characters) and others are **reference** types to other objects.
- A **reference** to an object allows sending it a **message**.
- For each type of message the object can receive, the class declares a **method** associated with the message type.
- This method is a procedure that is **executed** by the object when it receives the associated message type.

The interface of an object

- The method signatures of a class describe **how** other objects can **communicate** with it.
- To interact with an object, one must have a **reference** to the object and know its **interface**.
- Knowing the interface of an object means knowing the **signatures of its methods** and, of course, the associated documentation.
- In Java, it is possible to materialize the interface of an object using **interface declarations**. How do we do it? We declare an `interface` that lists method signatures, then a `class` that provides the actual code.
- Do not confuse programming interfaces, the subject of this course, with graphical interfaces (GUIs).

Example: A canvas to draw figures

```
public interface Canvas {

    String getName();
    int getWidth();
    int getHeight();
    Collection<Figure> getFigures();
}
```

notice

NO method bodies. Just signatures followed by semicolons (;).
This interface says: “Anything that claims to be a Canvas must provide these four methods.”
It says **nothing** about how.

An interface is not a class

- A class can declare to **implement** one or **more interfaces** defined by its **visible** methods.
- An interface can be seen as a **commitment** to implement the **methods** declared in the interface.
- The interface **does not specify** how its methods are implemented.
- Two classes can implement the same interface very **differently**.

Example of implementing an interface

```java
public class BasicCanvas implements Canvas {

    private final int width;
    private final int height;
    private final String name;
    private final ArrayList<Figure> figures;

    public BasicCanvas( final int width,
                        final int height,
                        final String name ) {
        this.width = width;
        this.height = height;
        this.name = name;
        figures = new ArrayList<>();
    }
    @Override
    public int getWidth() {
        return width;
    }
    @Override
    public int getHeight() {
        return height;
    }
    @Override
    public Collection<Figure> getFigures() {
        return figures;
    }
    @Override
    public String getName() {
        return name;
    }
}
```

```java
public interface Canvas {

    String getName();
    int getWidth();
    int getHeight();
    Collection<Figure> getFigures();
}
```

- If a class declares that it **implements** an interface, it must provide an implementation for **all** the methods **declared** in the interface.
  - Unless the class is **abstract**.
- The class can also provide **other** methods that are not declared in the interface.
- The compiler is vigilant!

Remarks on interface naming

- We introduced an interface named `Canvas`. In Java, there will be **a name collision** if the class and the interface have the **same name** and are declared in the **same package**.
- To avoid this, we named our implementing class **differently**:
  - `BasicCanvas`.
- We could also have named the class `Canvas` and the interface `CanvasInterface` (or `ICanvas` for short).
- However, a good practice is to use a more **generic name** for the interface, which has a **higher level of abstraction** than the class.
- More **specific** names can then be given to the different **implementations** based on their **characteristics**:

```
/* This is a default basic implementation...*/
public class BasicCanvas implements Canvas { // ...
/* This implementation optimizes resources consumption...*/
public class OptimizedCanvas implements Canvas { // ...
```

Interfaces as a *design tool*

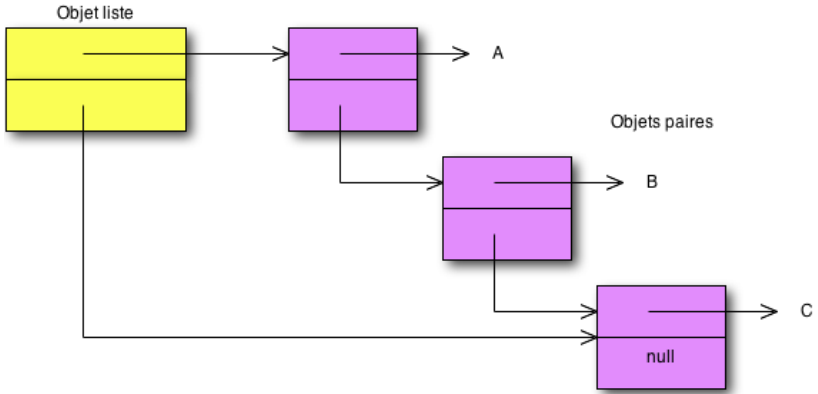
- Interfaces are a design tool:
  - They should be used during the **software design phase**.
- During this phase, we identify the different **types** of objects in the problem.
  - Then each object is characterized by its **documented interface**.
- We think about the interface before thinking about the class:
  - If well documented, the interface serves as a **specification for the methods** that objects must implement.

Example: the `List` interface

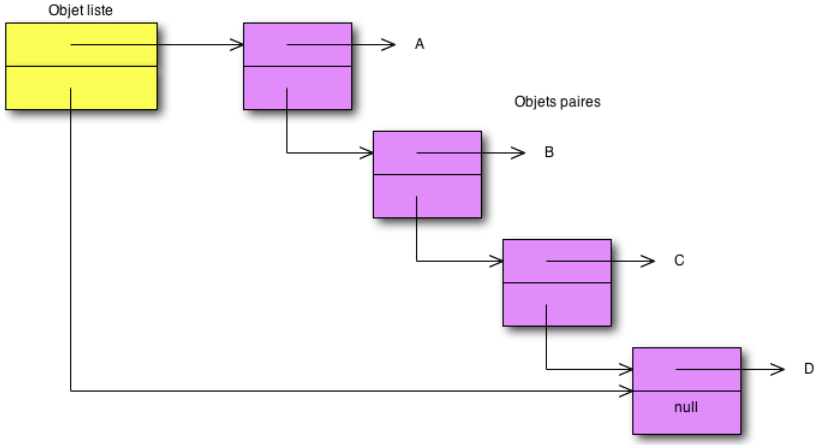
- Search: JAVA SE List
- Examples for two of its implementations:
  - ArrayList
  - LinkedList
- `ArrayList`: Implementation storing elements in an **array**.
- `LinkedList`: Implementation storing elements in a **linked list**.

Structure of a *linked list*

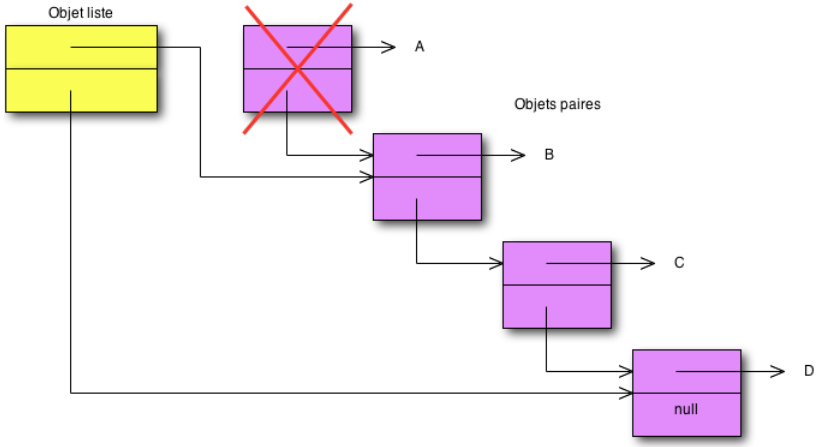
- It is implemented using **auxiliary objects** (pairs) linked by **references**.
- A pair (node) has a **reference** to the next pair (node).
- The **last object** in the list has a **null** reference for the next pair.



Adding an element to the end of the list



Removing the first element of the list



Interfaces as a *contractual tool*

- A linked list structure can be used to implement a list:
  - Adding an object to the list means adding this object at the end of the list.
  - Taking the first object from the list means taking the first object of the list and removing it from the chain.
- The designers of the `LinkedList` class declared that this class fulfills the contract defined by the `List` interface.
  - It is worth noting that this does not prevent the class from offering additional methods specific to queues, such as those defined by the `Queue` interface.
- An array (`Array`) can also be used to implement a list:
  - Adding an object to the list means adding this object at the end of the underlying array, and increasing its size if necessary.
  - Taking the first object from the list means taking the first object of the array and removing it from the array.
- The designers of the `ArrayList` class declared that this class fulfills the contract defined by the `List` interface.
- The `List` interface thus acts as a **contract**.

Interfaces as an *encapsulation tool*

- By using the interface as a **type** for a variable, access to the object's methods is **restricted**.
- **Question:** Do these compile?

```
Queue<E> myQueue = new LinkedList<E>();
E myElement = myQueue.get(0);
myQueue.add(myElement);
```

← Does this compile?

← Does this compile?

- Check the documentation...

Interfaces as an *encapsulation tool*

```
Queue<E> myQueue = new LinkedList<E>();
E myElement = myQueue.get(0);    // No.
myQueue.add(myElement);          // Yes.
```

← Does this compile?

← Does this compile?

- **Answer:** Only methods **declared** in the `Queue` interface are accessible. Other methods of the `LinkedList` class are hidden.
- Although `myQueue` is an instance of `LinkedList`, operations specific to this class are **inaccessible**.
- `myQueue` becomes a queue of type `Queue`, and the programmer is **forced** to use it as such.
  - It can no longer be used as a `LinkedList`.

Interfaces for presenting a *specific view*

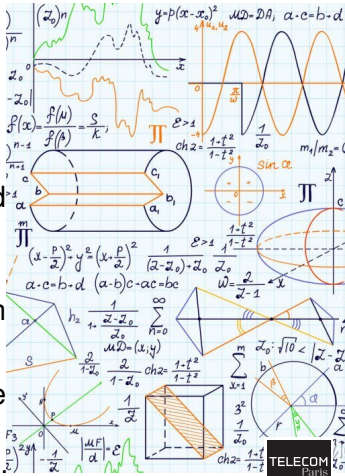
- Example:

```
Queue<E> myQueue = new LinkedList<E>();
```

- The `Queue` interface allows the linked list to be **viewed** as a **queue**.
- It therefore presents a **particular view** of the linked list.
- The provided object is the same, but the accessible methods differ:
  - Only **those of the view** (that is, of `Queue`) are available.

Shared properties of objects with *different identities*

- Classes **without any relation** can implement the **same interface**.
- In such cases, the interface describes a **property** that these classes **share** but may **implement differently**.
- For example: A graphic editor allowing drawing using predefined shapes:
  - Square, Rectangle, Circle, etc.
- There will be different classes for different types of shapes, such as `Square`, `Rectangle`, `Circle`, etc.
- The editor includes a grid-based drawing area (`Canvas`) where shapes are drawn.
  - This area will be represented by an object of the `Canvas` class.



Shared properties of objects with different identities

- **All shape classes** must implement the `Drawable` interface:

```
public interface Drawable {
    void paint(Graphics graphics);
}
```

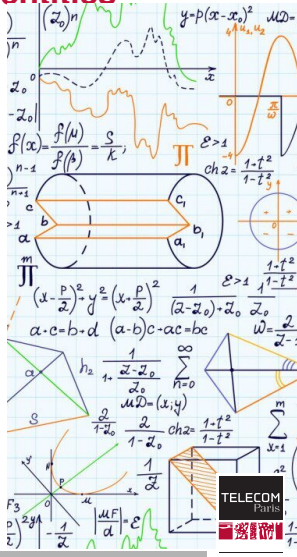
- For each shape to be drawn, the graphic editor will ask the shape to draw itself by sending the `paint(graphics)` message and passing an object of type `Graphics`.
- To display itself, each figure will know how to use this **graphics** object.

Shared properties of objects with different identities

- What about the canvas grid?
- We will define a `Grid` class that implements `Drawable`:

```
public class Grid implements Drawable {
    @Override
    public void paint(Graphics graphics) {  // ...
    }          // ...
}
```

- The editor program will treat the `Grid` type grid like any other shape during rendering, even though the grid and other shapes are **not of the same nature (identity)**.
- The `Drawable` interface represents the shared **ability to draw oneself**, common to these two classes of different identities.
- This is polymorphism through **interfaces**, not inheritance.



Composition (via inheritance) of interfaces

- Interfaces can be **composed** to define a new interface.
- This is achieved through **inheritance** from one or **more** interfaces.
- This means that the new interface contains its own method declarations but also, implicitly, the **method declarations of the interfaces it inherits from**.
- Of course, we can do without this possibility of composition, since a class can implement multiple interfaces.
- However, composing these interfaces into a single interface can simplify programming and make the used OO model more **explicit**.

Example of interface composition via inheritance

```java
public interface Whistling {
    void whistle();
}
```

```java
public interface Bird extends
    Whistling, Flying {
}
```

```java
public class BasicBird implements Bird {
    // ...
}
```

```java
public interface Walking {
    void walk();
}
```

```java
public interface Human extends
    Whistling, Walking {
}
```

```java
public class BasicHuman implements Human {
    // ...
}
```

```java
public interface Flying {
    void fly();
}
```

Name conflicts

- When an interface inherits from multiple other interfaces, declarations in these different interfaces may have methods with the **same name**.
  1. If two method declarations have identical headers, there is no issue:
     - The implementing class only needs to implement this method **once**.
  2. If two method declarations have the **same method name** but **different parameters** (either in type or number), there is no issue:
     - The implementing class must implement **both methods**.
  3. A problem arises when two method declarations have the same method name, the same **parameter signatures**, but a **different return type**.
     - In this case, the compiler will reject the program due to a **name conflict**.
- This type of name conflict also occurs with class inheritance:
  - Two methods with the same name and identical parameter signatures but different return types can only be declared if the return type of the subclass method **inherits** the return type of the superclass method.

Example (of no conflict)

```
public class RectangularShape extends Shape {
    private final int width;
    private final int height;     // ...
}
```

```
public abstract class Component {     // ...
    public Shape getShape() {
        return shape;
    }    // ...
}
```

```
public class Area extends Component {
   // ...
    @Override
    public RectangularShape getShape() {
        return (RectangularShape) super.getShape();
}
```

← No conflict because RectangularShape inherits from Shape

Constants in interfaces

- An interface contains method declarations.
  - Normally, it does not contain implementations of these methods...
    - Except for so-called **default** methods (since Java 8, not covered here).
- You can also declare **constants** in an interface.

```
public interface MathConstants {
    double PI = 3.1416;
}
```

- If multiple classes need to share the same constants, it is best to declare them in an interface.
- In any class, you can access this constant by `MathConstants.PI`.
- In any class that implements the `MathConstants` interface, you can directly access this constant (by simply writing `PI`).

```
public class Circle implements MathConstants { // ...
    public double getArea() {
        return PI * getRadius() * getRadius();
    } // ...
}
```

Interfaces as markers

- Java provides various data structures for **object collections**, such as `ArrayList`, `LinkedList`, etc.
- An `ArrayList` object allows access to its elements by a numeric index. This access is done in **constant time**.
  - The time to access an element at index `n` is bounded by a **constant** that does **not depend** on `n`.
- A `LinkedList` object also allows access to its elements by a numeric index. But, access to elements **is no longer** in constant time.
  - To access an element, one must start from the first pair of the list and follow the chain of pairs until the desired element is reached.
  - The access time to the element at index `n` is **bounded** by an affine function of `n`
    - `f(x) = (ax + b)`.

Interfaces as markers

- The `Collections` class in the JDK provides various sorting algorithms to reorder a collection of objects according to a specified order.
- The optimal sorting algorithm depends on whether elements can be accessed in constant time or not.
- There are specialized algorithms for arrays (constant-time index access) and others specialized for linked lists (linear-time index access).
- How can the sorting algorithm provided by the `Collections` class know whether index access to the elements is constant or linear?
- Java allows the programmer to indicate this explicitly. But how?

Interfaces as markers

- Java introduces an **empty** interface (declaring no methods) named `RandomAccess`:

```
public interface RandomAccess { }
```

- A programmer who wants to indicate that their collection class has constant-time index access simply declares that their class implements the `RandomAccess` interface.
- The generic sorting algorithm in the `Collections` class can determine if the submitted collection allows constant-time index access by testing if the object **implements** the `RandomAccess` interface.
- To do this, it uses the following expression:

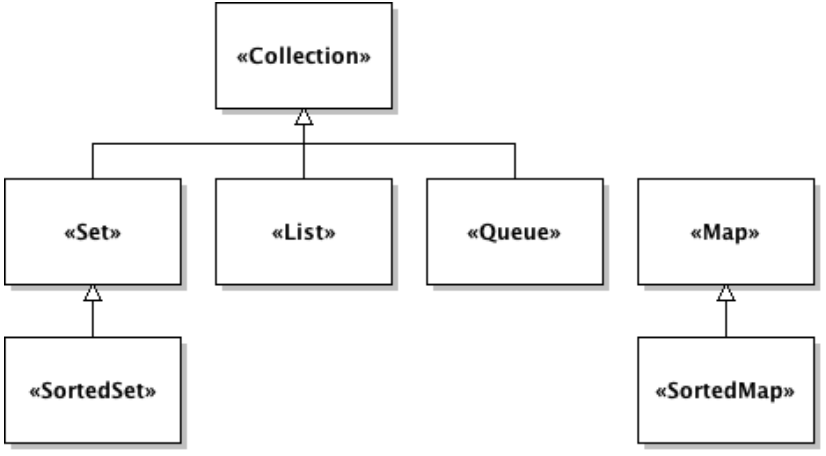
```
if (collection instanceof RandomAccess) {    /* ... ArrayList ... */   }
else {    /* ... LinkedList ... */    }
```

- This explains the existence of many **empty interfaces** in the JDK.

Interfaces of JDK collections

- A collection is a data structure that groups a variable number of objects into a single object.
- Collections are represented in Java by implementation classes; there are different ways to group objects depending on the **intended use** of the collection.
- Collections are also represented by **interfaces** that define the **views** one can have on the objects that implement them.
- Collections are also represented by a **set of algorithms** that allow us to **manipulate** them.

Interfaces of JDK collections



The `Collection` interface

- The `Collection` interface is the root of the tree.
  - It contains what is common to **all collections**.
- Search: JAVA SE Collection

The `Set` interface

- The `Set` interface represents a finite set of objects.
- It cannot contain the same element twice:
  - A call to the `add(element)` method will return `false`, and the element will **not** be added.
- The order of objects is not guaranteed:
  - Objects will not necessarily be retrieved in the same order as they were inserted.
- Search: JAVA SE Set

The `SortedSet` interface

- The `SortedSet` interface represents a finite set of **ordered** objects.
- It cannot contain the same element twice:
  - A call to the `add(element)` method will return `false`, and the element will **not** be added.
- The order of objects is guaranteed:
  - An **ordering function** can be provided.
- Search: JAVA SE SortedSet

The `Map` interface

- The `Map` interface represents a **key – value** association table.
  - Also known as a **hash table**.
- The order of key-value pairs is not guaranteed.
- Search: JAVA SE Map

The `SortedMap` interface

- The `SortedMap` interface represents a **key – value** association table with an **order on the keys**.
  - A function for ordering the keys must be provided.
  - Elements are maintained in ascending order of the keys.
- Can be used to model a directory or a phonebook.
- Search: JAVA SE SortedMap

How to choose an adequate collection interface implementation?

- Example for the `Map` interface:
- `Hashtable`:
  - Access to elements is **synchronized**, which allows handling **concurrent** data access.
  - However, synchronization introduces an execution time **overhead**.
- `HashMap`:
  - Access to elements is not **synchronized**, which does not allow parallelism.
  - However, there is no execution time overhead.

How to choose an adequate collection interface implementation?

- Example for the `List` interface:
- `ArrayList`:
  - Insertion or removal of elements is **expensive**.
  - However, access to an element is done in **constant time**.
- `LinkedList`:
  - Insertion or removal of elements is **inexpensive**.
  - However, access to an element is done in **linear** time.
- In summary, it is important to understand the application's **requirements** and carefully **study** the **characteristics** of the classes to choose the **right implementation**.

Programming to the interface

- Until now, when we needed a variable of type list of objects, we always declared it as an `ArrayList`:

```
public class Factory extends Component {

    private final ArrayList<Component> components;

    public Factory() {
        components = new ArrayList<Component>();
    }

    public ArrayList<Component> getComponents() {
        return components;
    }
    // ...
}
```

Programming to the interface

- However, what happens if we discover that in certain usage contexts of the program, the `ArrayList` implementation is not adequate for the need?
  - For example, not performing well enough...
- It will then be necessary to find in the code **all the declarations** where the `ArrayList` class has been used (attributes, local variables, method parameters, etc.) and **change** all these declarations in order to use the correct implementation.
- This could be very tedious, especially for large applications containing **thousands of classes** that need to be maintained for **many years**...
- Therefore, a good practice to avoid this problem is to **program to the interface**.

Programming to the interface

- To this end, we will instead use the `List` interface to declare our attributes, local variables, method parameters, etc.

```
public class Factory extends Component {

    private final List<Component> components;

    public Factory() {
        components = new ArrayList<Component>();
    }

    public List<Component> getComponents() {
        return components;
    }
    // ...
}
```

Programming to the interface

- Thus, if we need to change the implementation of the list, we will only need to modify the **code where the list is instantiated**.
  - The rest of the code using the list will compile regardless of the chosen implementation.

```
public class Factory extends Component {

    private final List<Component> components;

    public Factory() {
        components = new LinkedList<Component>();
    }

    public List<Component> getComponents() {
        return components;
    }
    // ...
}
```

Programming to the interface

- Programming to the interface (that is, declaring the **interface** rather than the class) significantly improves code **maintainability** by **abstracting** the implementation that will be used at runtime.
- Additionally, the interface only exposes the **common** methods of the implementing classes, preventing the programmer from using **implementation-specific** methods that would make the code harder to maintain when the implementation changes.
- Some applications may even choose to change the implementation **at runtime**, depending on changes in data characteristics or the environment, for example,
  - *Self-adaptive software...*

Iterable collections

- All collections support the enhanced `for` loop.
- If `myCollection` is a collection of `Data` objects, and the collection's class implements the `Iterable` interface, you can iterate over this collection using the enhanced `for` loop:

```
Collection<Data> myCollection = new ArrayList<>();

for (Data myData : myCollection) {
    myData.doSomething();
    // ...
}
```

- **All** JDK collections also support the creation of **iterators** that can be used to traverse them.
  - See the Iterator design pattern.

Collection algorithms

- The `Collections` class (with an ‘s‘) contains several class methods (`static`) implementing algorithms on collections.
- Search: JAVA SE Collections

Conclusion

- We have seen different uses of Java interfaces:
  - **Design tool**: We describe the interfaces of each object in a problem before implementing them with classes. Think about the interface **before** the class.
  - **Contract**: The interface is seen as a **contract** that the class **must adhere to**.
  - **Encapsulation tool**: The interface allows showing **only a certain aspect** of an object while hiding its other characteristics.
  - **Shared properties**: The interface allows describing a property **shared** by **different classes** that have no inherent relationship (different identities).
- **Composition**: Interfaces can be **composed** via inheritance.
- **Best practice**: **Programming to the interface**, not the implementation
  - It promotes code **maintainability** by facilitating the **change** of implementations, even during program execution...
- In a good program, **declarations** are made **at the interface**, and **instantiation is centralized** in a method (or class) designed for that purpose. For more, see the Factory Method design pattern.