Paradigm : A way of structuring and thinking about code : **-imperative -declarative -object-oriented**
Types
Scalar : int init to 0 - **Ref to an object** `Point myPoint = new Point();`

Class
Overload constructor : multiple declaration with diff param. - `this` **Keyword** : `this.` to call attribut or method of the class, `this()` superclass' constructor.
Example code :

```
public class Robot {
    int energy;
    // Constructor: initializes a Robot with an energy
        value
    public Robot(int initialEnergy) {
        energy = initialEnergy;
    }
    // Method: recharge the robot
    public void recharge() {
        energy = 100;
        System.out.println("Robot␣is␣fully␣recharged!");
    }
}
```

To call a constructor in another use `this`

```
Point(double rho, double theta) {
    this((int)(rho * Math.cos(theta)), (int)(rho * Math.sin
        (theta)));
}
```

Object invoke each other methods via references. obj.method(arg) ¡- send a message to the object referenced by obj.
Reference allows an object to send data. Exchnaged data are described by a methode of the class.

Static
Class : It can be called without creating an instance of the class but can not access to instance var. - **Attribute** : shared by all instances

Running a Java Program

```
public static void main(String[] args) {
    Point myPoint = new Point(10, 10);
    myPoint.writeCoordinates();
}
```

Lists
ArrayList : - Access : veryfast o(1) -Add/Remove : At the end/début : Fast, at the middle : Slow

```
ArrayList<Point> points = new ArrayList<>();
points.add(new Point(1,2));
//Loop (2 possibilites)
for(int = 0; i < points.size(); i++){...}
for(Point p : points){...}
```

LinkedList : - Access : Slow -Add/Remove: Fast

```
LinkedList<Point> points = new LinkedList<>();
```

Encapsulation: protect internal state, control access.
Principles: use **private** attributes and **public** getters/setters. Hide implementation details to allow safe changes.
Example:

```
public class Account {
  private double balance;
  public Account(double initial) { balance = initial; }
  public double getBalance() { return balance; }
  public void deposit(double amount) { if (amount > 0) {
  balance += amount;} }
  public void withdraw(double amount) { if (amount > 0 &&
      amount <= balance) {balance -= amount;} }
}
```

Benefits: prevents invalid states, improves modularity, facilitates maintenance. **Inheritance**
Subclass (child) ⟵ **Superclass** (parent)
Principles: **Enrichment** (add attributes/methods), **Redefinition** (override methods using `@Override` to ensure correct method overriding and enable compile-time checking).
Examples:

```
class LuxuryItem extends Item {
  @Override
  double getVAT() { return 0.33 * getNetPrice(); }
}
class Television extends LuxuryItem {
  private int voltage, screenSize;
  int getVoltage() { return voltage; }
}
```

If no parent specified, inherits from `Object` class (`java.lang` package). Java uses **single inheritance** (one parent only). Child constructor must call `super()` as first instruction.
Polymorphism: an object instantiated from a subclass can be used as its own type and as any of its ancestor types. Allows writing flexible and reusable code: `Item item = new Television();` is valid. Method calls are dynamically resolved based on the real object type at runtime.
Binding: **Static** (compile-time, variable type), **Dynamic** (runtime, real object type).
Useful methods: `getClass()`, `toString()`.
Final keyword: **Method** (no override), **Class** (no extend), **Attribute** (immutable). **Abstract Class**: cannot instantiate directly. **Concrete Classes**: can instantiate (e.g., `TV`, `Computer`).
Example:

```
public abstract class LuxuryItem extends Item { }
//But this is possible:
LuxuryItem li = new Television();
```

Notes:
- Can declare variables of abstract type.
- Only subclass methods usable.

Methods Example:

```
public abstract class Item {
  private final double netPrice;
  public Item(double price) { this.netPrice = price; }
  public final double getNetPrice() { return netPrice; }
  public abstract double getVAT();
  public final double getATIPrice() { return netPrice +
      getVAT(); }
}
```

Interfaces Use to only define not methode code

```
    public interface Canvas {
        String getName ();
        int getWidth ();
        Collection <Figure > getFigures ();
    }

    public class BasicCanvas implements Canvas {
        private final int width ;
        private final ArrayList <Figure > figures ;

        public BasicCanvas ( final int width, final int
            height, final String name ) {
            ...
        }

        @Override
        public int getWidth() {
            return width;
        }
        ...
    }
```

Name collision : Interfaces and implementation can not have the same name in the same package. **The purpose :** - During the software design phase - Specification for the methods - Acts as a contract - Type for a variable - Shared properties - Classes need to share the same constants.
Inheritance :

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

Name conflict : If two method inherited have the same name : - If the signatures are the same : No problem - If parameters are diffrent : Implement both methods - If return types are different : Not allowed
Marker : U can use cst to use it everywhere with `nameInterface.cst` or with an implementation.

Collections in Java Java provides several implementations for object collections, such as `ArrayList` and `LinkedList`. An `ArrayList` allows constant-time access to an element at index n, but insertion and removal operations are expensive. In contrast, a `LinkedList` has access time bounded by an affine function $f(x) = ax + b$, but insertion and removal operations are inexpensive.

Sorting and Collection Definition To inform sorting algorithms about the type of access (random or sequential), Java introduces the empty interface `RandomAccess`. This allows algorithms to check the type using `if (collection instanceof RandomAccess)`. A collection is a data structure that groups a variable number of objects into a single object. In Java, collections are represented by classes or interfaces and are accompanied by algorithms to manipulate them.

Programming to the interface : Use *interfaces* (e.g., `List`) instead of concrete classes (e.g., `ArrayList`) to allow easy implementation changes. **Example**:

```
List<Component> components = new ArrayList<>();
```

Benefit: change the implementation without modifying the rest of the code. **Errors**

Architecture MVC La **View** est l'interface utilisateur (fenêtres). Le **Model** contient les données et la logique métier, déterminant l'état de l'UI. Le **Controller** connecte la View et le Model en réponse aux actions de l'utilisateur.

Patron Observer-Observable Toute modification du Model notifie automatiquement les Views. Deux interfaces principales : `Observer` avec `modelChanged()` et `Observable` avec `addObserver()` et `removeObserver()`. Ce mécanisme permet plusieurs vues synchronisées sur un même modèle. The exception propagates up through the method call stack to the main() method, three types exist : - **Severe errors** Handled by objects of the Error - **Programmer-handled errors** Objects of the Exception class - **Language-related errors**: Objects of the RuntimeException class (a subclass of Exception) - A11 are **subclasses** of the Throwable class.

```
    try {
        // Open file
        // Read and process file
    }
    catch (IOException ex) {
        // Handle I/O errors
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
    finally {
        // Close file (always executed)
    }
```

The catch is chosen by its type and the order

```
    public String computeFullName(String firstName, String
         lastName)
    throws InvalidFirstNameException{
        if (firstName == null || firstName.isEmpty()) {
            throw new InvalidFirstNameException("First␣name
                ␣is␣invalid");
}
```

Create own exception

```
public class InvalidLastNameException extends Exception {
    private final String name ; // Data embedded into the
         exception
    public InvalidLastNameException ( String name ) {
        super ("␣Invalid␣last␣name␣:␣" + name );
        this . name = name ;
    }
    public final String getName() {
        return name ;
    }
}
```

Logging
It use to **store messages**, composed of three parts : **Logger**(send messsages) - **Formatter**(Set the format of messages) - **Appender/hander**(Send messages to the file or console) They are diffrent levels (`Severe-Warning-Info-Config..`)

```
import java.util.logging.Logger;
import java.util.logging.Level;
import java.util.Arrays;

public class Main {
    private static final Logger LOGGER = Logger.getLogger(
         Main.class.getName());

    public static void main(String[] args) {
        LOGGER.info("Demarrage␣de␣la␣simulation");
    }
}
```