Java: Classes and Interfaces

The main concepts in simple terms

Fields, methods, constructors, access modifiers, interfaces, abstract classes, records and enums

1. What does a class contain?

A class can contain:

- fields — the data of an object;
- constructors — used to create objects;
- methods — the actions an object can perform.

A simple example:

```
public class Student {

    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void introduce() {
        System.out.println(
            "My name is " + name
        );
    }
}
```

Here:

- `name` and `age` are fields;
- `Student(...)` is a constructor;
- `introduce()` is a method.

Main idea

A class describes what data an object stores and what actions it can perform.

2. Creating an object

An object is created using the `new` keyword:

```
Student student =
    new Student("Anna", 14);
```

Here:

```
Student
```

is the variable type,

```
student
```

is the variable name,
and

```
new Student("Anna", 14)
```

creates a new object.
After that, we can call one of its methods:

```
student.introduce();
```

3. Constructor

A **constructor** is called when a new object is created.
Its name is the same as the class name:

```
public class Book {

    private String title;

    public Book(String title) {
        this.title = title;
    }
}
```

A constructor does not have a return type.
Therefore, we do not write:

```
// Incorrect
public void Book(String title) {
}
```

This code declares an ordinary method, not a constructor.
The keyword `this` refers to the **current object**:

```
this.title = title;
```

On the left is the object's field, and on the right is the constructor parameter.

4. Multiple constructors

A class can have several constructors with different parameters.

```
public class User {

    private String name;
    private boolean active;

    public User(String name) {
        this(name, true);
    }

    public User(
            String name,
            boolean active) {

        this.name = name;
        this.active = active;
    }
}
```

The call

```
this(name, true);
```

calls another constructor of the same class.
This call must be the first statement in the constructor.

5. Access modifiers

Access modifiers determine where fields and methods can be accessed from.

	Same class	Package	Subclass	Everywhere
`private`	yes	no	no	no
`no modifier`	yes	yes	only in the same package	no
`protected`	yes	yes	yes	no
`public`	yes	yes	yes	yes

At first, it is useful to remember two main options:
private
is commonly used for the internal state of a class.
public
is used for operations that should be available to other code.
For example:

```
public class Player {

    private int health;

    public int getHealth() {
        return health;
    }
}
```

6. static

The keyword **static** means that a field or method belongs to the class itself, not to one particular object.
For example:

```
public class MathHelper {

    public static int square(int x) {
        return x * x;
    }
}
```

The method can be called without creating a `MathHelper` object:

```
int result = MathHelper.square(5);
```

The result is:

```
25
```

Important

An ordinary method is called on an object.

```
student.introduce();
```

A static method is usually called through the class name.

```
MathHelper.square(5);
```

7. final

The keyword **final** has different meanings depending on where it is used.

final field

After it has been assigned, the field cannot be assigned a different value or reference:

```
private final String name;
```

final method

It cannot be overridden in a subclass:

```
public final void print() {
    System.out.println("Hello");
}
```

final class

It cannot be extended:

```
public final class Calculator {
}
```

Constants often use both `static` and `final`:

```
public static final int MAX_SIZE = 100;
```

8. What is an interface?

An **interface** defines a contract:

"what an object must be able to do".

For example:

```
public interface Printable {

    void print();
}
```

The interface says that an object must provide a `print()` method.
A class can now implement this interface:

```
public class Document
        implements Printable {

    @Override
    public void print() {
        System.out.println(
            "Printing document"
```

```
        );
    }
}
```

The keyword

```
implements
```

means that the class implements an interface.

9. Why do we need interfaces?

An interface allows us to work with different objects in the same way.
For example:

```
public class Photo
        implements Printable {

    @Override
    public void print() {
        System.out.println(
            "Printing photo"
        );
    }
}
```

Now both `Document` and `Photo` can be used through the common type:

```
Printable item = new Photo();

item.print();
```

We can also create an array containing different objects:

```
Printable[] items = {
    new Document(),
    new Photo()
};

for (Printable item : items) {
    item.print();
}
```

This is why interfaces work so well with polymorphism.

10. A class can implement multiple interfaces

Java does not allow a class to extend several classes at the same time.
However, one class can implement multiple interfaces.
For example:

```
public interface Printable {
    void print();
}

public interface Savable {
    void save();
}
```

One class can implement both:

```
public class Document
        implements Printable, Savable {

    @Override
    public void print() {
        System.out.println("Print");
    }

    @Override
    public void save() {
        System.out.println("Save");
    }
}
```

A `Document` object now has two roles:

- it can be printed;
- it can be saved.

11. Abstract class

An **abstract class** is a class that cannot be instantiated directly.
It can contain:

- fields;
- constructors;
- ordinary methods;
- abstract methods.

For example:

```
public abstract class Animal {

    private final String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void sound();
}
```

The method

```
public abstract void sound();
```

does not have an implementation.
A subclass must provide one:

```
public class Dog extends Animal {

    public Dog(String name) {
        super(name);
    }
```

```
    @Override
    public void sound() {
        System.out.println("Woof!");
    }
}
```

The call

```
super(name);
```

calls the constructor of the parent class.

12. Abstract class or interface?

A simple rule:
Abstract class
is useful when related classes share common data and implementation.
For example:

Dog is an Animal.

Interface
is useful when we want to describe a capability or role.
For example:

Document is Printable.

	Abstract class	Interface
Object fields	yes	usually no
Constructor	yes	no
Implemented methods	yes	yes, using `default`
How many can be used	one class can be extended	multiple interfaces can be implemented
Main idea	common base	common contract

13. Default methods in interfaces

An interface can contain a method with an implementation.
The keyword `default` is used for this:

```
public interface Greeting {

    void sayHello();

    default void sayBye() {
        System.out.println("Bye!");
    }
}
```

A class must implement:

```
sayHello()
```

but it already receives the implemented method:

```
sayBye()
```

Important

For a first introduction, remember this:
an ordinary interface method defines a required contract, while a `default` method already contains an implementation.

14. record

A **record** is useful for small objects whose main purpose is to store data.
For example:

```
public record Student(
    long id,
    String name
) {
}
```

Creating an object:

```
Student student =
    new Student(1, "Anna");
```

Reading its data:

```
long id = student.id();
String name = student.name();
```

Java automatically provides a record with:

- a constructor;
- accessor methods;
- `equals()`;
- `hashCode()`;
- `toString()`.

Important

A regular class is useful for objects with behaviour and, when needed, mutable state.
A `record` is especially convenient for simple value objects.

15. enum

An **enum** is used when there is a known and limited set of possible values.
For example, days of the week:

```
public enum Day {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

Or an order status:

```
public enum OrderStatus {
    NEW,
    PAID,
    SHIPPED,
    CANCELLED
}
```

Usage:

```
OrderStatus status =
    OrderStatus.PAID;
```

The advantage of an `enum` is that a value can only be one of the allowed options.

16. Generics: a type in angle brackets

Generics allow a class or interface to work with different data types.
We have already seen them in collections:

```
List<String>
List<Integer>
```

We can also create our own generic class:

```
public class Box<T> {

    private final T value;

    public Box(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

Now the same class can be used with different types:

```
Box<String> textBox =
    new Box<>("Hello");

Box<Integer> numberBox =
    new Box<>(42);
```

The letter `T` represents a type that will be specified later.

Important

Generics allow us to write one reusable class while keeping compile-time type checking.

17. sealed types

This is a more modern Java feature.
The keyword **sealed** allows us to explicitly limit which classes can implement an

interface or extend a class.
For example:

```
public sealed interface Result
        permits Success, Failure {
}
```

Only the listed types are allowed:

```
public final class Success
        implements Result {
}

public final class Failure
        implements Result {
}
```

This is useful when the possible set of types is known and limited.
For example:

$$\text{Result} \longrightarrow \begin{cases} \text{Success}, \\ \text{Failure}. \end{cases}$$

For your first Java programs, `sealed` is not an essential topic, but it is useful to know that this feature exists in modern Java.

18. equals(), hashCode() and toString()

All Java classes inherit basic methods from the `Object` class.
These include:
equals()
compares objects.
hashCode()
returns the hash code of an object.
toString()
creates a text representation of an object.
For example:

```
System.out.println(student);
```

For a `record`, these methods are automatically generated based on its components.
For regular classes, they sometimes need to be overridden manually.

19. Common mistakes

Mistake 1.
Making every field `public`:

```
public int health;
```

Usually, an object's state is better protected using `private`.

Mistake 2.
Confusing a constructor with a method.
Constructor:

```
public Student(String name) {
}
```

Method:

```
public void print() {
}
```

Mistake 3.
Trying to create an object of an abstract class:

```
// Not allowed
Animal animal = new Animal("Bob");
```

Mistake 4.
Declaring that a class implements an interface but forgetting to implement its required method.

Mistake 5.
Using inheritance only because we want to reuse existing code.
Inheritance should represent a real relationship between types.

20. Quick reference

class
Describes the state and behaviour of objects.
constructor
Creates and prepares an object.
private
Hides part of a class from outside code.
public
Makes a class, field or method accessible to other code.
static
Belongs to the class rather than to one particular object.
final
Restricts further assignment, overriding or inheritance.
interface
Defines a common contract or role.
abstract class
Provides a common base for related classes.
record
Conveniently stores a group of related values.
enum
Describes a fixed set of possible values.
generics
Allow the same code to work safely with different types.

Main principle

Use a **class** when an object needs to store state and perform actions.
Use an **interface** when you need to describe a common capability or role:

“what an object can do”.

Use an **abstract class** when related classes share a common base.
Use a **record** for simple value objects.
Use an **enum** when the set of allowed values is known in advance.
The main idea is not to use a language feature simply because it is more complicated

or more modern.
Choose the feature that describes the task most clearly.