Annotations in Java

1. What are Annotations?

- **Annotations** are **metadata** (data about data) in Java.
- They provide **information to the compiler or JVM** but do not change program logic directly.
- Introduced in **Java 5**.
- Used for:
    - **Compiler instructions** (e.g., `@Override`)
    - **Runtime processing** (e.g., reflection, frameworks like Spring, Hibernate)
    - **Code generation / Documentation**

2. Syntax of Annotations

```
@AnnotationName
```

Example:

```
@Override
public String toString() {
    return "Hello";
}
```

3. Built-in Annotations in Java

A. Common Annotations

@Override

Ensures a method is overriding from parent class.
Compiler error if not correctly overridden.

```
class Parent {
    void show() {}
}
class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child show()");
    }
}
```

@Deprecated

Marks method/class as obsolete.
Compiler shows warning when used.

```
@Deprecated
void oldMethod() {
    System.out.println("This is deprecated.");
}
```

@SuppressWarnings

Tells compiler to ignore specific warnings.

```
@SuppressWarnings("unchecked")
List list = new ArrayList();
```

B. Meta-Annotations (Annotations for Annotations)

These define how annotations behave.

@Retention

Defines how long the annotation is retained:

- `SOURCE` → Removed by compiler
- `CLASS` → Stored in class file but not loaded in JVM
- `RUNTIME` → Available at runtime (via reflection)

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {}
```

@Target

Specifies where annotation can be applied:

- `TYPE` → class, interface, enum
- `METHOD` → methods
- `FIELD` → fields
- `PARAMETER` → parameters

```
@Target(ElementType.METHOD)
public @interface MyMethodAnnotation {}
```

@Inherited

Child classes inherit parent class annotations.

@Documented

Includes annotation in Javadoc.

4. Creating Custom Annotations

Example:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation {
    String author();
    String date();
}

public class Test {
    @MyAnnotation(author = "Neeraj", date = "22-09-2025")
    public void myMethod() {
```

```
        System.out.println("Custom Annotation Example");
    }
}
```

5. Accessing Annotations via Reflection

```
import java.lang.reflect.*;

public class ReadAnnotation {
    public static void main(String[] args) throws Exception {
        Method method = Test.class.getMethod("myMethod");
        MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);

        System.out.println("Author: " + annotation.author());
        System.out.println("Date: " + annotation.date());
    }
}
```

6. Predefined Annotations in java.lang.annotation

Annotation	Use
`@Retention`	How long annotation is kept
`@Target`	Where annotation can be applied
`@Inherited`	Inherited by subclasses
`@Documented`	Included in Javadocs

Anonymous Classes and Lambda Expressions in Java

1. Anonymous Classes

- A class **without a name**.
- Declared and instantiated at the **same time**.
- Useful when:
    - You need a **short-term implementation** of an interface or abstract class.
    - You don't want to create a full class file.

Example 1: Anonymous Class with an Interface

```
interface Greeting {
    void sayHello();
}

public class Test {
    public static void main(String[] args) {
        Greeting g = new Greeting() {
            @Override
            public void sayHello() {
                System.out.println("Hello from Anonymous Class!");
            }
        };
        g.sayHello();
    }
```

```
}
```

Example 2: Anonymous Class with Abstract Class

```
abstract class Animal {
    abstract void sound();
}

public class Test {
    public static void main(String[] args) {
        Animal dog = new Animal() {
            @Override
            void sound() {
                System.out.println("Dog barks");
            }
        };
        dog.sound();
    }
}
```

2. Lambda Expressions

- Introduced in **Java 8**.
- Provide a **cleaner, shorter way** to implement **functional interfaces** (interfaces with exactly one abstract method).
- Improves readability and reduces boilerplate code compared to anonymous classes.

Syntax

```
(parameters) -> expression
(parameters) -> { statements }
```

Example 1: Runnable with Lambda

```
public class Test {
    public static void main(String[] args) {
        // Before (Anonymous Class)
        Runnable r1 = new Runnable() {
            public void run() {
                System.out.println("Running with Anonymous Class");
            }
        };
        r1.run();

        // After (Lambda Expression)
        Runnable r2 = () -> System.out.println("Running with Lambda
Expression");
        r2.run();
    }
}
```

Example 2: Custom Functional Interface

```
@FunctionalInterface
interface Calculator {
    int add(int a, int b);
}
```

```
public class Test {
    public static void main(String[] args) {
        // Lambda expression implementing add method
        Calculator calc = (a, b) -> a + b;
        System.out.println("Sum: " + calc.add(5, 3));
    }
}
```

Example 3: Lambda with Collections

```
import java.util.*;

public class Test {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Neeraj", "Amit", "Priya");

        // Using Lambda with forEach
        names.forEach(name -> System.out.println(name));
    }
}
```

3. Differences Between Anonymous Classes and Lambda Expressions

Feature	Anonymous Class	Lambda Expression
Introduced in	Java 1.1	Java 8
Syntax	Verbose	Concise
Used for	Any interface/abstract class	Only **functional interfaces**
`this` keyword	Refers to anonymous class object	Refers to enclosing class object
Readability	Less readable (boilerplate code)	More readable (short syntax)

Java Generics

- **Generics** were introduced in Java 5.
- They allow us to write **type-safe** and **reusable** code.
- Generics enable classes, interfaces, and methods to operate on **any data type** without losing **type safety**.
- They provide **compile-time type checking** and eliminate the need for **type casting**.

2. Advantages of Generics

- **Type Safety** → Only the specified data type can be stored.
- **Elimination of Type Casting** → No need to explicitly cast objects.
- **Code Reusability** → Write one class/method, use it for any type.
- **Compile-time Checking** → Errors caught during compilation.

3. Generic Class

We can create a class that works with any type using `<T>` (Type parameter).

```
// Generic Class
class Box<T> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<String> b1 = new Box<>();
        b1.set("Hello");
        System.out.println(b1.get());

        Box<Integer> b2 = new Box<>();
        b2.set(100);
        System.out.println(b2.get());
    }
}
```

4. Generic Methods

We can define methods with generics.

```
class GenericMethodDemo {
    // Generic Method
    public <T> void printData(T data) {
        System.out.println(data);
    }

    public static void main(String[] args) {
        GenericMethodDemo obj = new GenericMethodDemo();
        obj.printData("Hello");   // String
        obj.printData(123);       // Integer
        obj.printData(45.67);     // Double
    }
}
```

5. Bounded Types

We can restrict the type parameter using **extends**.

```
class NumberBox<T extends Number> { // Only Numbers allowed
    private T num;

    public NumberBox(T num) {
        this.num = num;
    }

    public double getDoubleValue() {
        return num.doubleValue();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        NumberBox<Integer> n1 = new NumberBox<>(10);
        System.out.println(n1.getDoubleValue());

        NumberBox<Double> n2 = new NumberBox<>(20.5);
        System.out.println(n2.getDoubleValue());

        // NumberBox<String> n3 = new NumberBox<>("Hello"); // ERROR
    }
}
```

6. Wildcards in Generics

Wildcards are represented by `?` and provide flexibility.

Unbounded Wildcard (`?`)

Accepts any type.

```
public void printList(List<?> list) {
    for(Object obj : list) {
        System.out.println(obj);
    }
}
```

Upper Bounded Wildcard (`? extends Type`)

Accepts type or subtype of given class.

```
public void sumOfNumbers(List<? extends Number> list) {
    double sum = 0;
    for(Number n : list) {
        sum += n.doubleValue();
    }
    System.out.println("Sum: " + sum);
}
```

Lower Bounded Wildcard (`? super Type`)

Accepts type or its superclass.

```
public void addNumbers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
}
```

7. Generic Interfaces

Interfaces can also be generic.

```
interface Container<T> {
    void add(T item);
    T get();
}
```

```
class MyContainer<T> implements Container<T> {
    private T item;

    public void add(T item) {
        this.item = item;
    }

    public T get() {
        return item;
    }
}
```

File Handling in Java (CRUD Operations)

Java provides the **`java.io`** package for file handling.
CRUD stands for **Create, Read, Update, Delete**.

1. Import Required Classes

```
import java.io.File;
import java.io.FileWriter;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
```

2. Create a File

```
public class CreateFileExample {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

3. Write to a File (Create/Update)

```
public class WriteFileExample {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("example.txt"); // overwrite
            writer.write("Hello Java File Handling\n");
            writer.write("CRUD Operations Example");
            writer.close();
            System.out.println("Successfully written to file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Append Mode:

```
FileWriter writer = new FileWriter("example.txt", true); // append mode
```

4. Read a File

Use **FileReader** and **BufferedReader**.

```
public class ReadFileExample {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");
            BufferedReader br = new BufferedReader(new FileReader(file));

            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

5. Update a File

Read the file, modify content, and write back.

```
import java.io.*;

public class UpdateFileExample {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");
            BufferedReader br = new BufferedReader(new FileReader(file));

            StringBuilder sb = new StringBuilder();
            String line;
            while ((line = br.readLine()) != null) {
                sb.append(line.replace("Java", "JAVA")).append("\n");
            }
            br.close();

            FileWriter writer = new FileWriter(file);
            writer.write(sb.toString());
            writer.close();
            System.out.println("File updated successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

6. Delete a File

```
public class DeleteFileExample {
    public static void main(String[] args) {
        File file = new File("example.txt");
        if (file.delete()) {
            System.out.println("Deleted the file: " + file.getName());
        } else {
```

```
            System.out.println("Failed to delete the file.");
        }
    }
}
```

7. Key Points

File class is used to represent files/folders.
FileWriter → Writing text to file.
FileReader & BufferedReader → Reading text.
Updating → Read → Modify → Write.
Deletion → `file.delete()`.
Exception Handling is necessary (`IOException`).