Java String Oops Exception

Here are some of the **most useful `String` methods in Java** with brief explanations and examples:

✅ 1. `length()`
Returns the length of the string.

```
java

String str = "Hello";
// Prints 5
System.out.println(str.length());
```

✅ 2. `isEmpty()`
Checks if the string is empty.

```
java

String str = "";
// Prints true
System.out.println(str.isEmpty());
```

✅ 3. `trim()`
Removes leading and trailing whitespaces.

```
java

String str = "   Hello   ";
// Prints "Hello"
System.out.println(str.trim());
```

✅ 4. `substring()`
Extracts a portion of the string.

```
java

String str = "Hello, world";
// Prints "Hello"
System.out.println(str.substring(0, 5));
// Prints "world"
System.out.println(str.substring(7));
```

✅ 5. `indexOf()`
Returns the index of the first occurrence of a character or string.

```
java

String str = "Hello, world";
// Prints 0
System.out.println(str.indexOf("H"));
// Prints 7
System.out.println(str.indexOf("world"));
```

✅ 6. `lastIndexOf()`

Returns the index of the last occurrence of a character or string.

```
java

String str = "Hello, world, Hello";
// Prints 13
System.out.println(str.lastIndexOf("Hello"));
```

✅ 7. `contains()`

Checks if the string contains a specified sequence.

```
java

String str = "Hello, world";
// Prints true
System.out.println(str.contains("world"));
```

✅ 8. `startsWith()` / `endsWith()`

Checks if the string starts or ends with a specified prefix or suffix.

```
java

String str = "Hello, world";
// Prints true
System.out.println(str.startsWith("Hello"));
// Prints true
System.out.println(str.endsWith("world"));
```

✅ 9. `replace()`

Replaces all matching subsequences with another.

```
java

String str = "Hello, world";
// Prints "Hello, Java"
System.out.println(str.replace("world", "Java"));
```

✅ 10. `toUpperCase()` / `toLowerCase()`

Converts the case of the string.

```
java

String str = "Hello";
// Prints "HELLO"
System.out.println(str.toUpperCase());
// Prints "hello"
System.out.println(str.toLowerCase());
```

✅ 11. `equals()` and `equalsIgnoreCase()`

Compares two strings.

```
java

String str1 = "Hello";
// Prints true
System.out.println(str1.equals("Hello"));
```

```
// Prints true
System.out.println(str1.equalsIgnoreCase("HELLO"));
```

✅ 12. `split()`

Splits a string into an array of substrings.

```
java

String str = "apple,banana,cherry";
// Prints [apple, banana, cherry]
String[] arr = str.split(",");
```

✅ 13. `charAt()`

Returns the character at a specified index.

```
java

String str = "Hello";
// Prints 'H'
System.out.println(str.charAt(0));
```

✅ 14. `compareTo()`

Compares two strings lexographically.

```
java

String str1 = "apple";
String str2 = "banana";
// Prints a negative number
System.out.println(str1.compareTo(str2));
```

✨ If you'd like, I can go into more depth (with code samples, comparison with **`StringBuilder`**) or a complete **String Cheat Sheet**. Just let me know! 🌟

Excellent question — Exception Handling in Java is a key topic.

Let's go through it **steps by example**.

🔥 What is Exception Handling?

- Exception = An event that disrupts the normal flow of your code.
- Exception Handling lets you gracefully handle these events instead of crashing your application.

🔹 Exception Handling keywords:

✅ **try** — Block that contains code which might throw an Exception.
✅ **catch** — Block that handles the Exception if it occurs.
✅ **finally** — Block that executes **always**, whether Exception occurs or not (typically used for cleanup).
✅ **throw** — Throws a custom Exception.
✅ **throws** — Declares that a method might throw Exception(s).

Exception Handling — example:

```java
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int a = 5;
            int b = 0;
            int c = a / b; // Exception: division by zero
        } catch (ArithmeticException e) {
            System.out.println("Can't divide by zero.");
        } finally {
            System.out.println("Execution finished.");
        }
    }
}
```

Output:

```rust
Can't divide by zero.
Execution finished.
```

Multiple `catch` Blocks:

```java
try {
    int[] arr = new int[5];
    arr[10] = 100;
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Index is invalid.");
} catch (Exception e) {
    System.out.println("Some other Exception.");
}
```

Exception Propagation with `throws`:

```java
public class ExceptionPropagation {

    static void methodWithException() throws Exception {
        throw new Exception("Something went wrong.");
    }

    public static void main(String[] args) {
        try {
            methodWithException();
        } catch (Exception e) {
            System.out.println("Exception caught in main.");
        }
    }
}
```

🔹 `throw` — Manual Exception:

```
java

public class ManualThrowExample {
    static void validateAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible.");
        }
        System.out.println("Eligible.");
    }

    public static void main(String[] args) {
        try {
            validateAge(16);
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage()); // Prints: Not eligible.
        }
    }
}
```

🔹 Exception Hierarchy:

✅ All Exception classes **derive from `java.lang.Exception`.**
✅ **`Error`** (like **`OutOfMemoryError`**) is **not meant to be caught by your code**.
✅ **`Checked Exception`** (like **`IOException`**) **should be either caught or thrown upward with `throws`.**
✅ **`Unchecked Exception`** (like **`NullPointerException`**) **does not need to be explicitly dealt with**.

✨ If you'd like, I can go into **more depth** with:

- Exception best practices
- Custom Exception classes
- Exception chaining
- Exception messages
- Exception logging
- Exception handling in multithreading or enterprise applications

✅ Just let me know!

Excellent. Here's a clear breakdown of **common Error and Exception types in Java**:

🔹 Common Exception Types in Java (Checked and Unchecked)

✅ **Checked Exceptions** (Must be caught or thrown)

- **IOException** — Input/Output related (like `FileNotFoundException`)

```
java

BufferedReader br = new BufferedReader(new FileReader("file.txt"));
```

- **ClassNotFoundException** — Loading a class that cannot be found

```
java

Class.forName("com.example.UnknownClass");
```

- **SQLException** — Database-related issues

```
java

java.sql.Connection conn = DriverManager.getConnection(url, user, password);
```

✅ **Unchecked Exceptions** (Runtime — do not need to be explicitly dealt with)

- **NullPointerException** — Accessing a `null` object

```
java

String str = null;
str.length();
```

- **ArrayIndexOutOfBoundsException** — Accessing invalid array index

```
java

int[] arr = new int[5];
arr[10] = 100;
```

- **ArithmeticException** — Division by zero

```
java

int a = 5/0;
```

- **ClassCastException** — Incorrect casting

```java
Object o = "string";
// This will cause ClassCastException
Integer number = (Integer) o;
```

- **NumberFormatException** — Failed number parsing

```java
String str = "abc";
// This will cause NumberFormatException
int number = Integer.parseInt(str);
```

🔹 Common Error Types in Java (Not Exception)

✅ **Error** — Happens due to problems outside the control of your application.
Usually, you **should not try to handle them**.

- **OutOfMemoryError** — JVM runs out of memory

```java
java.lang.OutOfMemoryError: Java heap space
```

- **StackOverflowError** — Method calls pile up without base case

```java
java.lang.StackOverflowError
```

- **VirtualMachineError** — Generic error related to the JVM itself (like `OutOfMemoryError`)

✨ If you'd like, I can:

- List **all Exception subclasses**,
- Explain their **common causes and solutions**,
- Provide **code samples** for each,
- Or even demonstrate **with a small project**.
  ✅ Just let me know!

Excellent — OOP (Object-Oriented Programming) is a **core concept in Java**.
Let's go through it **steps by example**.

🔹 What is OOP?

✅ OOP = **Object-Oriented Programming** — it's a way of organizing code by focusing on **objects** instead of just functions.
✅ An object typically **combines data (fields)** and **behavior (methods)**.

🔹 OOP Concept #1 — Class and Object:

✅ **Class** — Blueprint or template.
✅ **Object** — An instance of that class.

```
java

// Class
class Car {
    // Fields (state)
    String color = "Red";

    // Method (behavior)
    void drive(){
        System.out.println("Car is driving.");
    }
}

public class OOPExample {
    public static void main(String[] args) {
        // Create object from class
        Car myCar = new Car();

        // Access fields and methods
        System.out.println(myCar.color);
        myCar.drive();
    }
}
```

🔹 OOP Concept #2 — Encapsulation:

✅ Encapsulation = **hiding internal details and exposing only what's necessary**.
✅ Implemented by **making fields private and accessing them with getters/setters**.

```
java

class Person {
    // private field
    private String name;

    // getter
    public String getName(){
        return name;
    }

    // setter
    public void setName(String newName){
        this.name = newName;
```

```
    }
}

public class EncapsulationExample {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Alice");

        System.out.println(p.getName()); // Prints: Alice
    }
}
```

🔹 OOP Concept #3 — Inheritance:

✅ Allows **child class to reuse code** from parent class.
✅ **`extends`** keyword is used.

```
java


// Parent class
class Animal {
    void eat(){
        System.out.println("This animal eats food.");
    }
}

// Child class
class Dog extends Animal {
    void bark(){
        System.out.println("Dog barks.");
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
        d.bark();
    }
}
```

🔹 OOP Concept #4 — Polymorphism:

✅ **One action — many forms.**
✅ Mainly implemented in two ways:

1. **Method Overriding** (runtime) — child class provides its own implementation.

```
java


class Parent {
    void show(){
        System.out.println("Parent.");
    }
}

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

```
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show();
    }
}
```

Output:

```
Child.
```

2. **Method Overloading** (compile-time) — multiple methods with the SAME NAME but different parameters.

```
java

class Calculator {
    void add(int a, int b){
        System.out.println(a + b);
    }

    void add(int a, int b, int c){
        System.out.println(a + b + c);
    }
}

public class MethodOverloadingExample {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        calc.add(5, 10);
        calc.add(5, 10, 15);
    }
}
```

🔹 OOP Concept #5 — Abstraction:

✅ **Showing essential details and hiding implementation details.**
✅ Implemented with **abstract classes or interfaces**.

```
java

// Abstract class
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw(){
        System.out.println("This is a circle.");
    }
}

public class AbstractionExample {
    public static void main(String[] args) {
        Shape circle = new Circle();
        circle.draw();
```

```
    }
}
```

🔹 OOP Concept #6 — Interface:

✅ An **interface** contains **abstract methods**.
✅ A class **implements** the interface.

```
java

interface Printable {
    void print();
}

class Document implements Printable {
    @Override
    public void print(){
        System.out.println("Document is printing.");
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Printable p = new Document();
        p.print();
    }
}
```

✨ If you'd like, I can:
✅ Provide **practice problems**,
✅ Explain **association, aggregation, composition**,
✅ Explain **inner classes, anonymous classes**,
✅ Or even create **a complete project** demonstrating OOP in Java.
✨ Just let me know! 🌟