17.3 Exception handling using try with resources

💡 Java Exception Handling – Mastering `try-with-resources`

Java's exception handling mechanisms are designed to make your code more robust and readable. So far, we've explored `try-catch-finally`, `throw`, and `throws`. Now, let's take a closer look at one of the most elegant features introduced in Java 7: the `try-with-resources` statement.

If you've ever written code to read files, open database connections, or manage sockets, you know how important it is to **close resources**. Forgetting to do so can lead to memory leaks or file locks. That's exactly the problem `try-with-resources` solves!

🔍 What is `try-with-resources`?

The `try-with-resources` statement is used to **automatically close resources** such as files, streams, or sockets, once they are no longer needed.

To use it, the resource must implement the `AutoCloseable` interface (or `Closeable`, which extends `AutoCloseable`).

✅ Syntax:

```java
try (ResourceType resource = new ResourceType()) {
    // use resource
} catch (ExceptionType e) {
    // handle exception
}
```

Once the try block is exited—whether normally or via an exception—the resource is automatically closed.

🧪 Example: Reading a File Using `try-with-resources`

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

public class TryWithResourcesExample {
    public static void main(String[] args) {
        String filePath = "example.txt";

        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("Exception occurred while reading the file: " +
e.getMessage());
        }
    }
}
```

🔹 In this example, the `BufferedReader` is automatically closed when the try block is exited. No need for a `finally` block to close it manually!

🔄 Traditional `try-finally` vs `try-with-resources`

Here's how things looked before Java 7:

```java
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("example.txt"));
    // use reader
} catch (IOException e) {
    // handle exception
} finally {
    try {
        if (reader != null) {
            reader.close();
        }
    } catch (IOException ex) {
        // handle close exception
    }
}
```

😩 This approach is verbose and error-prone. Compare it with `try-with-resources`, which is much cleaner and safer.

🔁 Multiple Resources in One Try Block

You can manage **multiple resources** in a single `try-with-resources` statement, separated by semicolons:

```java
try (
    BufferedReader reader = new BufferedReader(new FileReader("example.txt"));
    FileWriter writer = new FileWriter("copy.txt")
) {
    String line;
    while ((line = reader.readLine()) != null) {
        writer.write(line + "\n");
    }
} catch (IOException e) {
    System.out.println("Exception: " + e.getMessage());
}
```

✅ Both `reader` and `writer` will be closed automatically, **in reverse order** of their creation.

🔧 Custom Resources with AutoCloseable

You can also create your own resource classes that implement `AutoCloseable`:

```java
class MyResource implements AutoCloseable {
    public void doSomething() {
        System.out.println("Using MyResource");
    }

    @Override
    public void close() {
        System.out.println("Closing MyResource");
    }
}

public class CustomResourceDemo {
    public static void main(String[] args) {
        try (MyResource resource = new MyResource()) {
            resource.doSomething();
        }
    }
}
```

📌 Output:

```
Using MyResource
Closing MyResource
```

📘 When to Use `try-with-resources`

Use `try-with-resources` when working with:

- File I/O (`BufferedReader`, `FileInputStream`, etc.)
- Database connections (`Connection`, `Statement`, `ResultSet`)
- Network sockets
- Any custom resource that implements `AutoCloseable`

📝 Final Thoughts

The `try-with-resources` statement is a cleaner, safer, and more concise way to handle resources in Java. It helps avoid boilerplate code and ensures that resources are always closed properly—even when exceptions occur.

Document by ***Suyash*** 😇