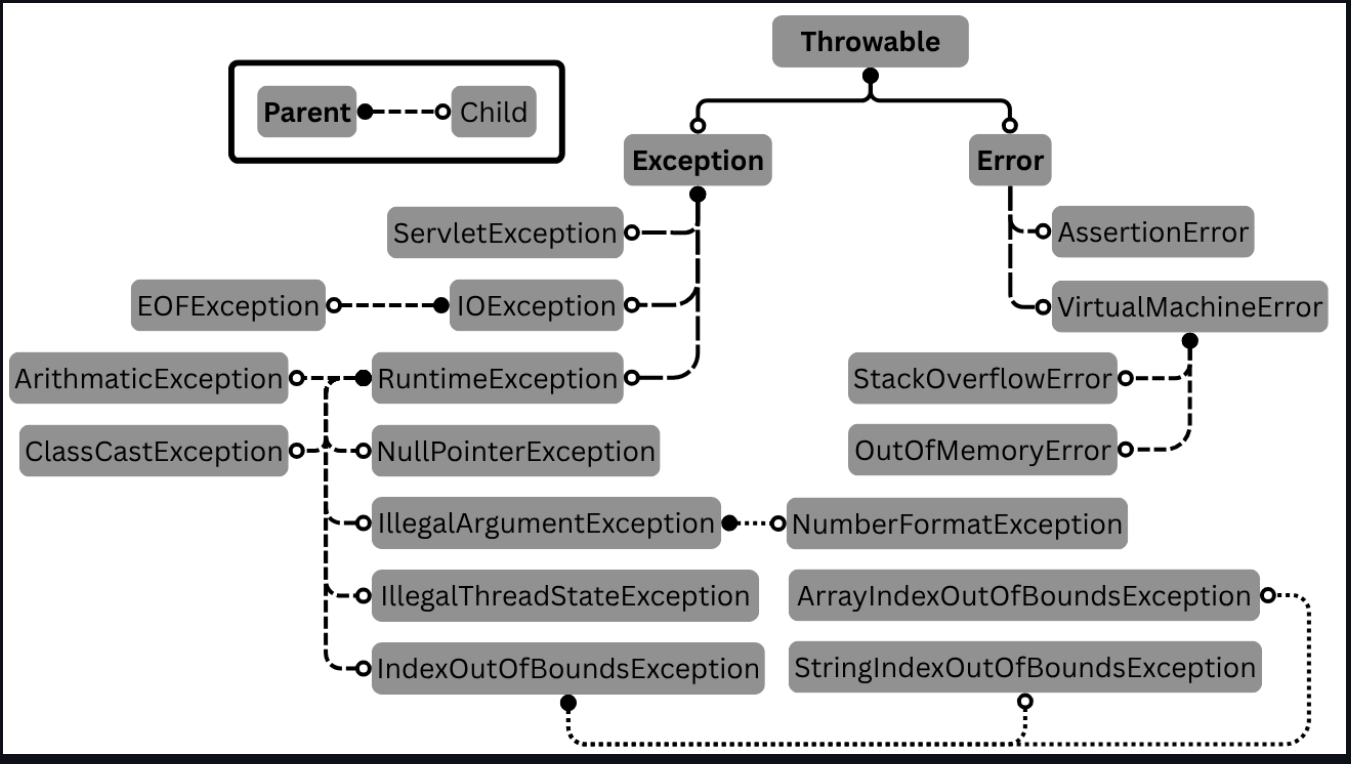
Exception Handling

- **Exception:** An unexpected, unwanted event that disturbs the normal flow of the program.
- **Exception Handling:** The process of providing an alternative course of action to continue program execution after an exception.
    - The primary objective of exception handling is to ensure **normal termination** of the program.

Types of Exceptions

- **Exception:** Typically caused by programming errors and are generally **recoverable**.
- **Error:** Usually caused by system-level issues (e.g., memory exhaustion) and are typically **non-recoverable**.

Classification



- **Checked Exceptions:**
    - Checked at **compile-time**.
    - Must be either caught or declared in the method signature.

- **Unchecked Exceptions:**
    - Not checked at compile-time.
    - Includes `RuntimeException` and its subclasses, as well as `Error` and its subclasses.

> **Note:** Regardless of type, **all exceptions occur at run-time**.

Run-time Stack Mechanism

```java
class RSM {
    public static void func() { func2(); }
    public static void func2() {
        System.out.println("func2");
    }
    public static void main(String[] args) {
        func();
    }
}
// Stack Illustration:
| func2(); |
| func();  |
| main();  |
```

- Each thread is assigned a separate **stack** by the JVM at creation.
- Method calls are stored as **activation records** (or **stack frames**) within the stack.
- When a method completes, its stack frame is removed.
- Upon completion of all method calls, the stack is destroyed and the thread terminates.

Default Exception Handler

```java
class DEH {
    public static void func() { func2(); }
    public static void func2() {
        System.out.println(10 / 0);
    }
    public static void main(String[] args) {
        func();
    }
}
```

- When an exception occurs, an **exception object** is created containing:
    - Type
    - Description
    - Location
- If the method lacks a handler, the JVM propagates the exception up the call stack.
- If `main()` is also unhandled, the **default exception handler** prints:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DEH.func2(Main.java:4)
    at DEH.func(Main.java:2)
    at DEH.main(Main.java:6)
```

Customized Exception Handling

```java
class CEH {
    public static void main(String[] args) {
        System.out.println("Hi");
        System.out.println(10 / 0);
        System.out.println("Bye");
    }
}
```

`try-catch`

```java
class TryCatchBlock {
    public static void main(String[] args) {
        System.out.println("Hi");
        try {
            System.out.println(10 / 0);
        } catch (ArithmeticException e) {
            System.out.println(10 / 2);
        }
        System.out.println("Bye");
    }
}
```

Flow Control in `try-catch`

```java
class TCB {
    public static void main(String[] args) {
        try {
            System.out.println("St 1");
            System.out.println("St 2");
            System.out.println("St 3");
        } catch (Exception e) {
            System.out.println("St 4");
        }
        System.out.println("St 5");
    }
}
```

Scenarios

- **Case 1**: No exception → `St 1`, `St 2`, `St 3`, `St 5`
- **Case 2**: Exception at `St 2` → `St 1`, `St 4`, `St 5`
- **Case 3**: Unhandled exception → `St 1`, program termination

> If no exception occurs, `catch` is ignored.
> If an exception occurs, remaining `try` statements are skipped.

Exception Methods (`Throwable` class)

- `.printStackTrace()` — Full trace with type, message, and location
- `.getMessage()` — Only the message
- `.toString()` — Type and message in one line

```java
class ExceptionExample {
    public static void main(String[] args) {
        try {
            System.out.println(10 / 0);
        } catch (ArithmeticException e) {
            e.printStackTrace();
            // System.out.println(e.getMessage());
            // System.out.println(e.toString());
        }
    }
}
```

Multiple `catch` Blocks

- Multiple `catch` blocks allowed.
- **Order** must be from **child** to **parent**.

```java
class MultipleCatch {
    public static void main(String[] args) {
        try {
            System.out.println(10 / 0);
        } catch (ArithmeticException e) {
            System.out.println("AE");
        } catch (Exception e) {
            System.out.println("E");
        }
    }
}
```

`finally` Block

- Always executes after `try`, regardless of exception occurrence or handling.
- Not executed if `System.exit()` is invoked.

```
class Finally {
    public static void main(String[] args) {
        try {
            System.out.println("try");
            System.out.println(10 / 0);
        } catch (ArithmeticException e) {
            System.out.println("catch");
        } finally {
            System.out.println("finally");
        }
    }
}
```

`finally` vs `return`

```
class FinallyVsReturn {
    public static int m1() {
        try { return 111; }
        catch (Exception e) { return 222; }
        finally { return 333; }
    }
    public static void main(String[] args) {
        System.out.println(m1());
    }
}
```

> **Output:** `333`

`throw` Keyword

- Used to explicitly throw an exception.
- Only applicable to `Throwable` objects.

- After `throw`, remaining statements become unreachable (unless inside a `try`).

```
class THROW {
    public static void main(String[] args) {
        throw new ArithmeticException("Demo");
        // System.out.println("Unreachable");
    }
}
class THROW2 {
    static ArithmeticException e;
    public static void main(String[] args) {
        throw e; // NullPointerException due to null reference
    }
}
Exception in thread "main" java.lang.NullPointerException: Cannot throw exception because "THROW2.e" is null
    at THROW2.main(Main.java:5)
```