CORE JAVA

JAVA EXCEPTIONS

1

LESSON

ERRORS AND EXCEPTION HANDLING

Learning Objectives

- ❑ Errors and Exception Handling
- ❑ Understanding Syntax Errors and Exceptions
- ❑ Exception Handling Techniques
- ❑ Custom Exceptions

WHAT IS AN EXCEPTION IN JAVA?

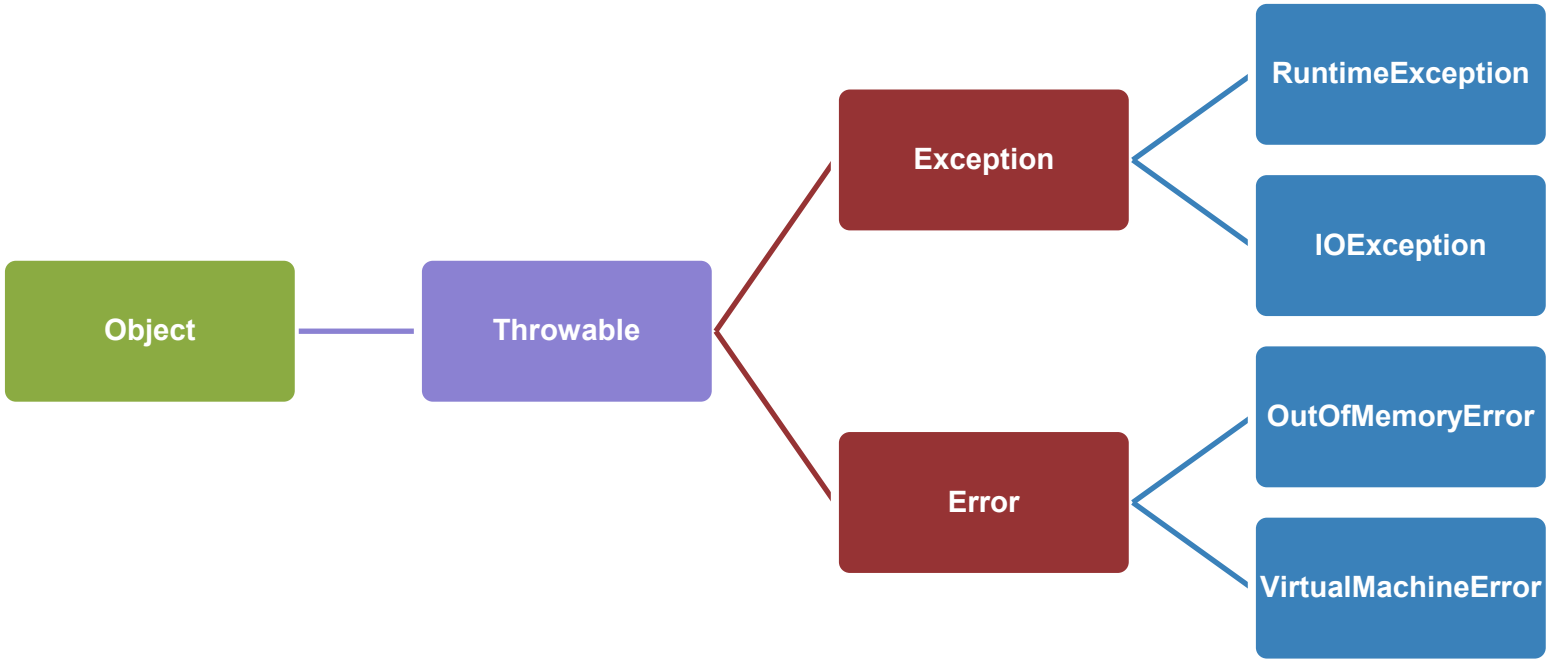
- Exceptions are abnormal events that disrupt the normal flow of execution in a program
- Java provides a robust mechanism to handle such situations and maintain the normal application flow

```
// Division by zero
try {
        int result = example.divide(10, 0); // This will throw
ArithmeticException
        System.out.println("Result: " + result);
} catch (ArithmeticException e) {
        System.out.println("Error: Division by zero is not allowed.");
}
```

EXCEPTION HIERARCHY

- **Throwable** is the base class for all errors and exceptions
- **Exception** (checked exceptions) and **RuntimeException** (unchecked exceptions) are subclasses

EXCEPTION HIERARCHY



TRY CATCH

The **try** statement lets you test a block of code for errors.

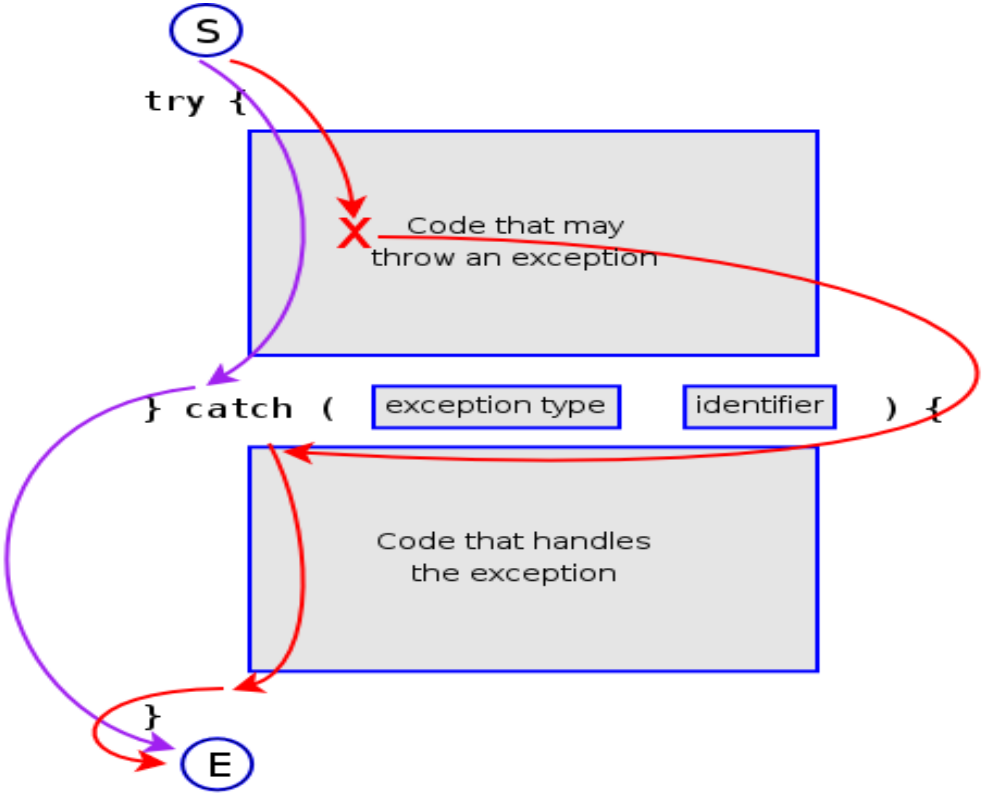
The **catch** statement lets you handle the error.

The **throw** statement lets you create custom errors.

The **finally** statement lets you execute code, after try and catch, regardless of the result.

> *The **try...catch** statement marks a block of statements to try and specifies one or more responses should an exception be thrown.*
>
> *If an exception is thrown, the **try...catch** statement catches it.*

WITH CATCH


S
try {
Code that may
throw an exception
} catch (
exception type
identifier
) {
Code that handles
the exception
}
E

TRY.. CATCH

- The **try...catch** statement consists of a try block, which contains one or more statements, and a **catch** block, containing statements that specify what to do if an exception is thrown in the **try** block.
- If the **try** block fails, control passes to the **catch** block.
- If any statement within the **try** block (or in a function called from within the try block) throws an exception, control immediately shifts to the **catch** block.

SYNTAX

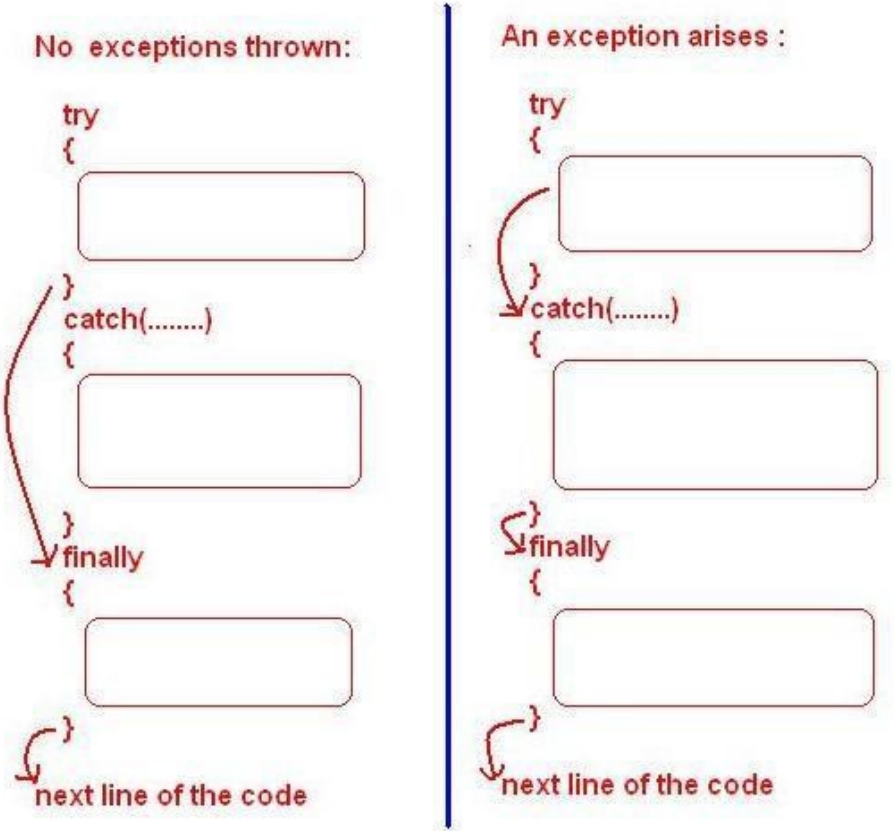
The try.. catch .. finally, syntax

```
Syntax:
try {
  Block of code to try
}
catch(Exception ex) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of the try / catch result
}
```

TRY.. CATCH

- If no exception is thrown in the **try** block, the **catch** block is skipped
- The **finally** block executes after the **try** and **catch** blocks execute but before the statements following the **try...catch** statement

WITH FINALLY



RUNTIME VS. APPLICATION EXCEPTIONS

- Runtime Exception:
  - ✓ These exceptions occur during the program's execution and are typically not recoverable (e.g., NullPointerException, ArrayIndexOutOfBoundsException).

- Checked (Application) Exception:
  - ✓ These exceptions are expected, and you are forced to handle them in your code (e.g., IOException, SQLException).

CREATE CUSTOM EXCEPTION TYPES

- Custom exceptions allow you to define application-specific error handling

```
class InvalidFileFormatException extends Exception {
    @Serial
    private static final long serialVersionUID = 1L;

    public InvalidFileFormatException(String message) {
       super(message);
    }
}
```

USING CUSTOM EXCEPTION TYPES

- Throw a custom exception when an abnormal condition reached

```
class FileProcessor {
    public void processFile(String file) throws InvalidFileFormatException {
        if (!file.endsWith(".txt")) {
            throw new InvalidFileFormatException("Invalid file format. Only .txt
files are supported.");
        }
        System.out.println("Processing file: " + file);
    }
}
```

CATCH CUSTOM EXCEPTION TYPES

- Use **try.. catch** block to handle a custom exception

```
public static void main(String[] args) {
        FileProcessor processor = new FileProcessor();
        try {
                processor.processFile("document.pdf");
        } catch (InvalidFileFormatException e) {
                System.out.println("Error: " + e.getMessage());
        }
}
```

PRACTICAL APPLICATION AND BEST PRACTICES

Banking

Handle insufficient balance, failed transactions, or unauthorized access using custom exceptions like **InsufficientFundsException** or **TransactionFailedException**

Finance

Use exception handling for missing data scenarios such as missing financial records or when a calculation cannot be performed due to incomplete input

School System

Handle exceptions when students or records cannot be found or when invalid data is entered into the system

Technology

Use exceptions to manage invalid data inputs, file processing issues, or system failures due to resource unavailability

THANKS!

Your feedback is welcome
support@kavinschool.com

2

ADDITIONAL REFERENCES

1

INTRODUCTION TO

JAVA PATTERNS

INTRODUCTION TO OO PROGRAMMING IN JAVA

KavinSchool

BUILDER PATTERN IN JAVA

KavinSchool

Learning Objectives

- What is Builder Pattern?
- Why do we need Builder Pattern?
- Advantages & Disadvantages
- Different ways to build
- Examples

BUILDER PATTERN

Is a design pattern providing solution for object creation problem

Is to separate the construction of a complex object from its representation

Is one of the Go4 design pattern

NEED OF BUILDER PATTERN

Too many arguments to be passed to the constructor

The type of arguments are the same with different overloaded constructor and sending the constructor values correctly is hard

Some of the parameters are optional but you are forced to send all the parameters, and optional parameters values that are sent with NULL

ADVANTAGES

Allows you to vary a products internal representation

Encapsulates code for construction and representation

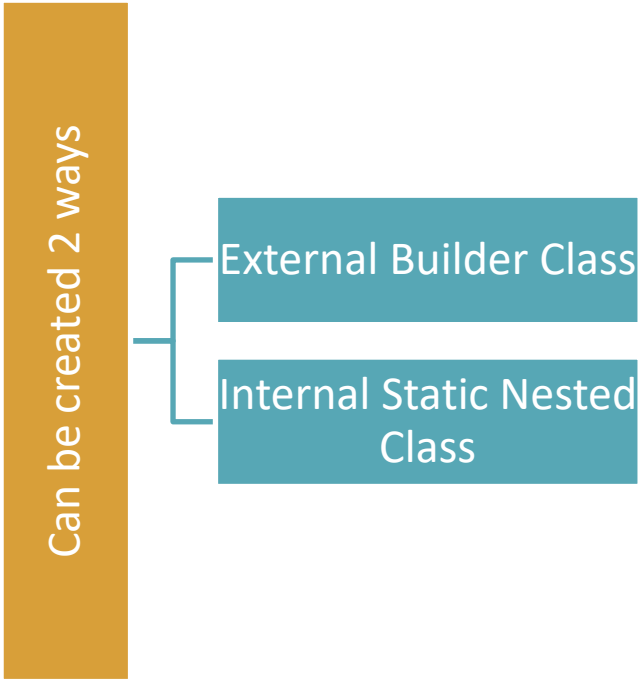
Provide control oversteps of construction process

You can force immutability of an object once its created

DISADVANTAGES

- Requires creating a separate Concrete Builder
- Requires the builder classes to be mutable
- Data Members of classes aren’t guaranteed to be initialized
- Dependency Injection less supported

BUILDER DESIGN PATTERN



External Builder Class

ADDRESS CLASS

Code

```
public class Address {

    private String addressLine1;
    private String addressLine2;
    private String city;
    private String state;
    private String zipCode;
    private String country;

    public Address(final String addressLine1, final String addressLine2, final String city, final
String state, final String zipCode, final String country) {
        this.addressLine1 = addressLine1;
        this.addressLine2 = addressLine2;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
        this.country = country;
    }
```

ADDRESS BUILDER CLASS

Code

```
public class AddressBuilder {
    private String addressLine1;
    private String addressLine2;
    private String city;
    private String state;
    private String zipCode;
    private String country;

    public AddressBuilder setAddressLine1(final String addressLine1) {
        this.addressLine1 = addressLine1;
        return this;
    }

    public AddressBuilder setAddressLine2(final String addressLine2) {
        this.addressLine2 = addressLine2;
        return this;
    }
```

ADDRESS BUILDER CLASS

Code

```
… some methods are omitted for brevity
 …

public AddressBuilder setZipCode(final String zipCode) {
    this.zipCode = zipCode;
    return this;
}

public AddressBuilder setCountry(final String country) {
    this.country = country;
    return this;
}

public Address createAddress() {
    return new Address(addressLine1, addressLine2, city, state, zipCode, country);
}
}
```

ADDRESS USING EXTERNAL BUILDER

Code

```
public class CreateAddressUsingExternalBuilderDemo {

    public static void main(String[] args) {
        Address address1 = new AddressBuilder().setAddressLine1("100 1st St")
.setCity("San Francisco").setState("CA").setZipCode("94001").createAddress();
        System.out.println("address1 = " + address1);
        Address address2 = new AddressBuilder()
.setAddressLine1("1334 Mission St")
.setCity("Fremont").setAddressLine1("CA").createAddress();
        System.out.println("address2 = " + address2);

    }
}
```

Internal Static Nested Class

CAR CLASS

Code

```
public class Car {
    private String color;
    private int currentSpeed;
    private final int year;
    private final int wheels;
    private final String make;
    private final String model;
    private final int maximumSpeed;
    private final int fuelCapacity;

    private Car(final Builder builder) {
        color = builder.color;
        currentSpeed = builder.currentSpeed;
        year = builder.year;
        wheels = builder.wheels;
        make = builder.make;
        model = builder.model;
        maximumSpeed = builder.maximumSpeed;
        fuelCapacity = builder.fuelCapacity;
    }
```

Code

```
public static Builder newBuilder() {
    return new Builder();
}

public static Builder newBuilder(final Car copy) {
    Builder builder = new Builder();
    builder.color = copy.getColor();
    builder.currentSpeed = copy.getCurrentSpeed();
    builder.year = copy.getYear();
    builder.wheels = copy.getWheels();
    builder.make = copy.getMake();
    builder.model = copy.getModel();
    builder.maximumSpeed = copy.getMaximumSpeed();
    builder.fuelCapacity = copy.getFuelCapacity();
    return builder;
}

public String getColor() {return color;}
```

BUILDER CLASS WITHIN CAR

Code

```
public static final class Builder {
   private String color;
   private int currentSpeed;
   private int year;
   private int wheels;
   private String make;
   private String model;
   private int maximumSpeed;
   private int fuelCapacity;

   private Builder() { }
   /**
    * Sets the color and returns a reference to this Builder so
that the methods can be chained together.
    */
   public Builder withColor(final String color) {
      this.color = color;
      return this;
   }
```

BUILDER CLASS WITHIN CAR

Code

→ *some method are omitted for briefness*

```
   public Builder withModel(final String model) {
      this.model = model;
      return this;
   }
   public Builder withMaximumSpeed(final int
maximumSpeed) {
      this.maximumSpeed = maximumSpeed;
      return this;
   }
   public Builder withFuelCapacity(final int fuelCapacity) {
      this.fuelCapacity = fuelCapacity;
      return this;
   }
   public Car build() {
      return new Car(this);
   }
}
```

CAR INTERNAL BUILDER DEMO

Code

```
public class CarInternalBuilderDemo {
   public static void main(String[] args) {
      Car car1 = Car.newBuilder().build();
      System.out.println("car1 = " + car1);

      Car car2 = Car.newBuilder().withColor("Blue")
.withCurrentSpeed(80).withFuelCapacity(20).withMake("Toyota")
.withModel("Sienna").withMaximumSpeed(150).withWheels(4).withYear(2012).build();
      System.out.println("car2 = " + car2);

     //cloning car2 to car3 using copy method
      Car car3 = Car.newBuilder(car2).build();
      System.out.println("car3 = " + car3);

      //You can clone car3, with some property changes
      Car car4 = Car.newBuilder(car3).withColor("Red").build();
      System.out.println("car4 = " + car4);
      }
}
```


Thank You!

THANKS!

Your feedback is welcome
support@kavinschool.com