Day 02

- Inheritance

.- Abstraction & Encapsulation

- Polymorphism
- Exception Handling
- Thread Programming
- File I/O
- SOLID Principles

Object Composition

(has-a relationship)

The composition is a design technique in java to implement a has-a relationship.

Java Inheritance is used for code reuse purposes and the same we can do by using composition.

The composition is achieved by using an instance variable that refers to other objects.

If an object contains the other object and the contained object cannot exist without the existence of that object, then it is called composition

Object Composition

(has-a relationship)

Composition is a way of describing reference between two or more classes using instance variable and an instance should be created before it is used.

```
class Customer
    name homeAddress workAddress

class Address
    doorNo, streetInfo, city, zipCode
```

Task : Object Composition

Write a program that manages Books and their Reviews:

- Book:
  - Id
  - Name
  - Review
- Review:
  - Id
  - Description
  - Rating

Inheritance

Use, Reuse And Extend

(is-a relationship)

Inheritance

Java supports one of the basic Object-Oriented Programming Paradigms : **Inheritance**.

- Inheritance is a mechanism of code reuse.
- Accomplished by using the Java keyword "***extends***"
- The "***super***" keyword allows a sub-class to access the attributes present in the super-class.

```
public class Person {
        // <Person Definition>
}

public class Student extends Person {
       // <Student Definition, after reusing Person Code>
}
```

Task : Inheritance

Create an Employee class extending Student Class with following attributes:

- Title
- Employer
- EmployeeGrade
- Salary

Create a method toString() within Employee to print all state attribute values, including those of Person.

Inheritance (Contd) How Constructors are called?

When a sub-class object is created -

- Sub-class constructor is called and it implicitly invokes its super-class constructor.
- The Java compiler inserts the code "***super()***" (if it is not explicitly added by the programmer) as the first statement in the body of the sub-class default constructor
- The statement ***super();*** is the invocation of the super-class default constructor.
- Hence, the body of the super-class constructor is always invoked before the body of the sub-class constructor

Multiple Inheritance

In Java, direct Multiple Inheritance is not allowed.

However we can create an inheritance chain.

- class C **is a** class B
- class B **is a** class A

In Java, it is permitted for a *super-class reference variable to reference a sub-class object instance*

- Pet pet = new Dog();

The operator `instanceof` is to find the relationship between an object and a class. If the object is an instance of the class or its sub class, it returns true.

- pet instanceof Pet
- pet instanceof Dog

Abstraction And More

The Hide And Show Game

Abstract Class

An abstract class is declared using the “abstract” keyword in its class definition.

Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties.

Points to remember about Abstract Classes -

- An instance of an abstract class can not be created.
- If a class contains at least one abstract method then compulsory should declare a class as abstract
- If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract methods.
- We can have an abstract class without any abstract method.
- abstract method in class(abstract class) can not be declared as final

Abstract Class

```
public abstract class AbstractRecipe {
                public void execute() {
                        prepareIngredients();
                        cookRecipe();
                        cleanup();
                }

                abstract void prepareIngredients();
                abstract void cookRecipe();
                abstract void cleanup();
        }
```

Cooking any dish normally involves following a tried and tested recipe, and its preparation boils down to these basic steps:

- ✓ Prepare the Ingredients
- ✓ Cook the Recipe
- ✓ Cleanup (the Mess created!)

These steps would be different for each dish but the order of steps remain the same.

Interfaces

It is the blueprint of the class.

Interfaces specify what a class must do and not how.

Interfaces in Java can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body)

If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

We can use Interface as type for reference variable

Interface Example

```java
public interface GamingConsole {
        public void up();
        public void down();
        public void left();
        public void right();
}
```

```java
public class MarioGame implements GamingConsole {
        // implement all unimplemented methods
}

public class ChessGame implements GamingConsole {
        // implement all unimplemented methods
}

GamingConsole game = new MarioGame();
GamingConsole game = new ChessGame();
```

Interface (Contd)

Interesting Facts

An interface can be extended by another interface. We can have an inheritance hierarchy purely consisting of interfaces.

An implementation of interface should implement all its methods including the methods in its super interfaces.

If a class is declared as abstract, it can skip providing implementations for all interface methods.

Interfaces cannot have member variables. An interface can only have declared **constants**

Starting from Java SE 8, an interface can provide a default implementation of the methods it provides.

Abstract Class and Interface : A Comparison

Factors	Interface	Abstract Class
Definition	Interface is a contract	Example of Abstraction
Usage	An interface is primarily used when you have two software components that need to communicate with each other, and there is a need to establish a contract.	An abstract class is primarily used when you want to generalize behavior by creating a super class.
Methods Access Specifier	No method declared inside an interface can be qualified with a private access specifier.	An abstract class can have private/protected methods.
Member variable declaration	An interface cannot have declared member variables	An abstract class can have member variable declarations.
Multiple Inheritance	An interface can extend only one interface,	A class or an abstract class can implement multiple interfaces.
Implementation	A Java interface can be implemented using the keyword **“implements”.**	An abstract class can be extended using the keyword **“extends”**.

Polymorphism

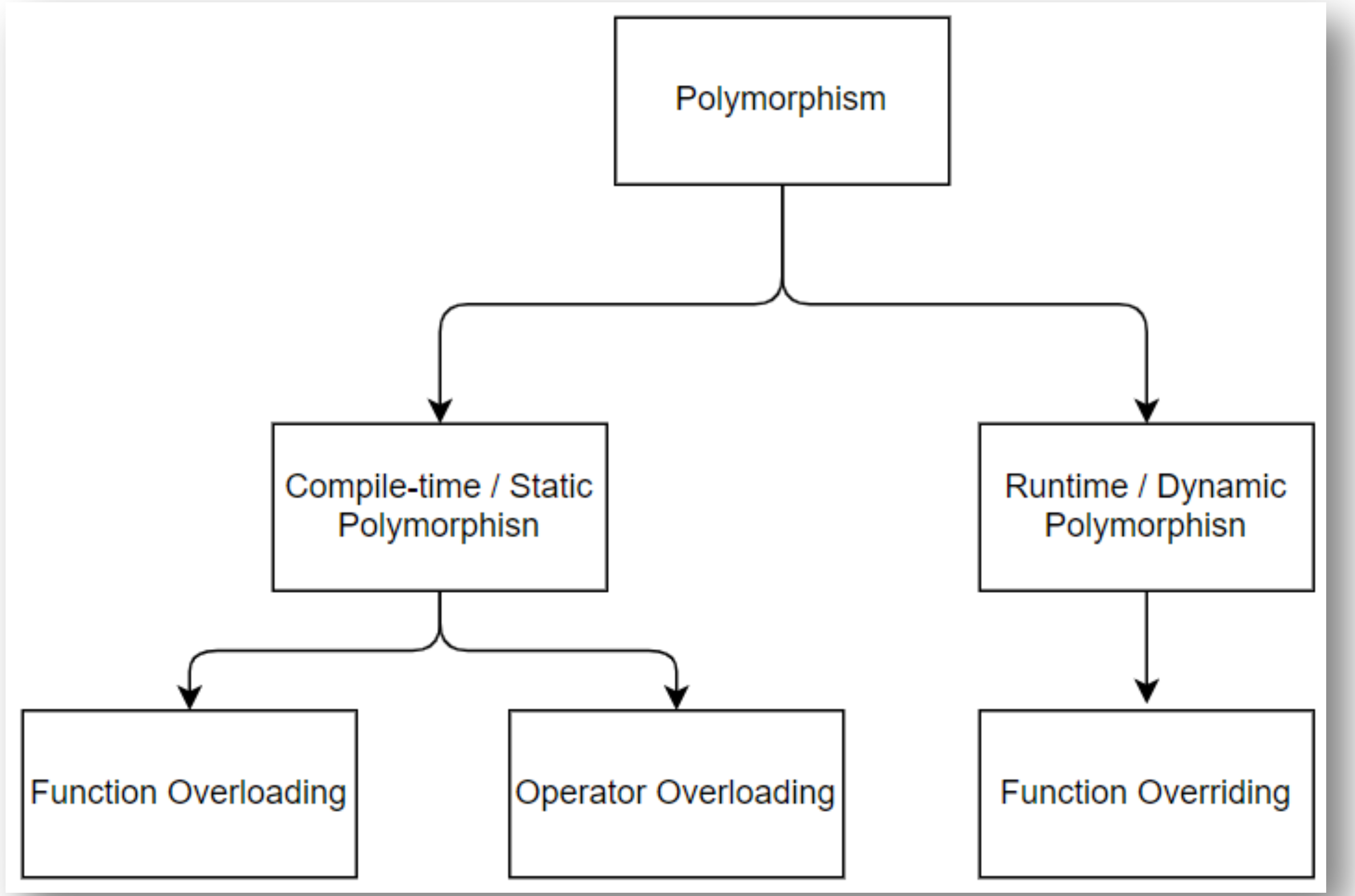
The word “poly” means many and “morphs” means forms, So it means many forms.

- For example : a person at the same time can have different characteristics/roles

In Java, polymorphism is mainly divided into two types -

- Compile-time / Static Polymorphism
- Runtime / Dynamic Polymorphism

Polymorphism (Contd)



Method Overloading

- Multiple functions with the same name but different parameters

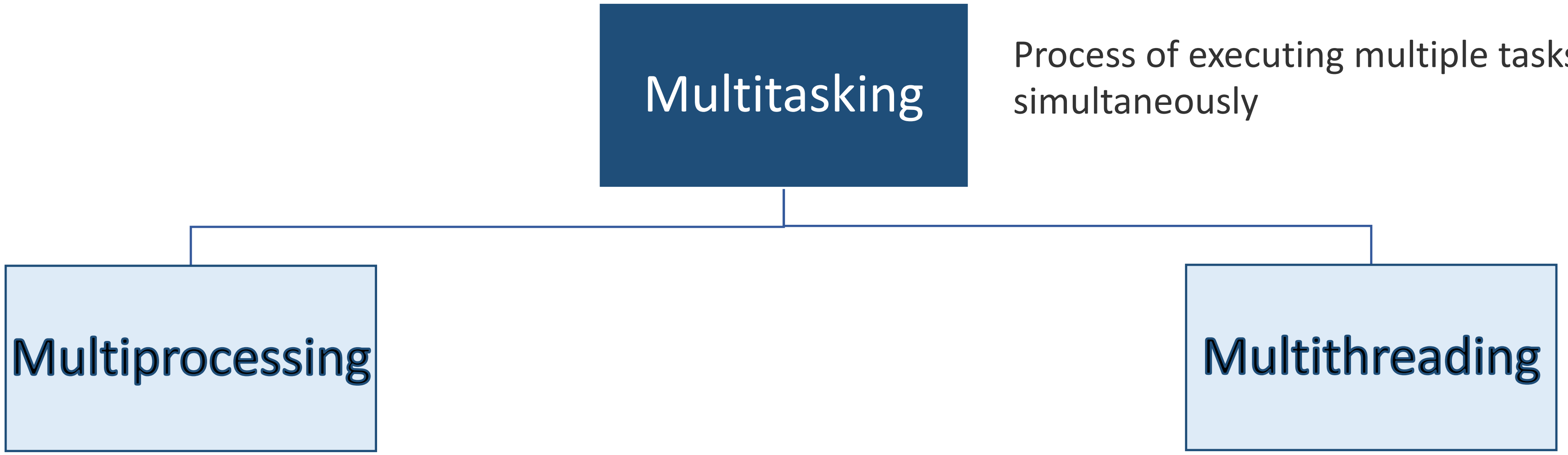
Method Overriding

- A derived class has a definition for one of the member functions of the base class

Threads & Concurrency

Let’s run it in parallel

Multitasking, Multiprocessing & Multithreading



Process of executing multiple tasks simultaneously

- Each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

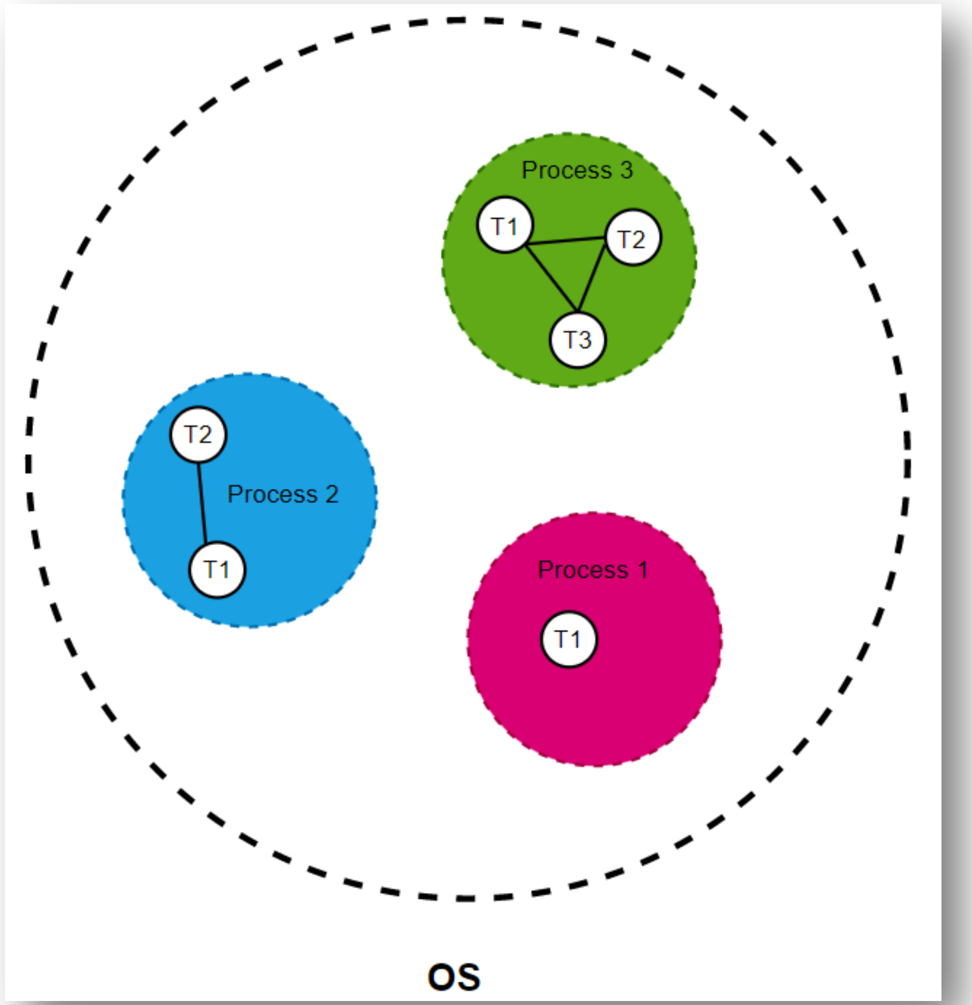
Threads in Java

A thread is a lightweight subprocess, the smallest unit of processing.

- It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads.

- It uses a shared memory area.



Threads in Java

Java provides two ways to create a thread programmatically.

Implementing the **Runnable** interface.	To make a class runnable, we can implement `Runnable` interface and provide implementation in `public void run()` method.
	To use this class as Thread, we need to create a Thread object by passing object of this runnable class and then call `start()` method to execute the `run()` method in a separate thread.
Extending the **Thread** class	We can extend **`Thread`** class to create our own java thread class and override `run()` method.
	Then we can create its object and call `start()` method to execute our custom java thread class run method.

The Thread LifeCycle

A Java Thread goes through a sequence of **states** during its lifetime.

The term life-cycle defines what specific state a thread could be in, at various points of time.

NEW	A thread is in this state as soon as it's been created, but its start() method hasn't yet been invoked.
TERMINATED / DEAD	When all the statements inside a thread's method have been completed, that thread is said to have terminated.
RUNNING	If the thread is currently running.
RUNNABLE	If the thread is not currently running but is ready to do so at any time.
BLOCKED / WAITING	If the thread is not currently running on the processor but is not ready to execute either.

`synchronized` Methods And Thread Utilities

Few of the methods for synchronization in the class

- `start()`
- `join()`
- `sleep()`
- `wait()`

Above methods have a few drawbacks:

- No Fine-Grained Control
- Difficult to maintain
- NO Sub-Task Return Mechanism

Exception Handling

Try It, Catch It!

Exception Handling

An *exception* is an error event that can happen during the execution of a program and disrupts its normal flow.

Java provides a robust and object-oriented way to handle exception scenarios known as Java Exception Handling.

The code that specifies what to do in specific exception scenarios is called exception handling.

Java creates an *exception object* when an error occurs while executing a statement

- . The exception object contains a lot of debugging information such as method hierarchy, line number where the exception occurred, and type of exception.

How Java handles the exceptions?

If an exception occurs in a method, the process of creating the exception object and handing it over to the runtime environment is called *“throwing the exception”.*

The normal flow of the program halts and the Java Runtime Environment (JRE) tries to find the handler for the exception. Exception Handler is the block of code that can process the exception object.

- The logic to find the exception handler begins with searching in the method where the error occurred.
- If no appropriate handler is found, then it will move to the caller method.
- If an appropriate exception handler is found, the exception object is passed to the handler to process it.

Java Exception handling framework is used to handle runtime errors only.

Exception Handling : Keywords

throw	used to throw exceptions to the runtime to handle it

- For example, in a user authentication program, we should throw exceptions to clients if the password is NULL.

throws	throws keyword in the method signature

- When we are throwing an exception in a method and not handling it, then we must use the `throws` keyword in the method signature to let the caller program know the exceptions that might be thrown by the method.

try…catch	block for exception handling

- try is the start of the block and catch is at the end of the try block to handle the exceptions.
- The catch block requires a parameter that should be of type Exception.

Finally	optional block

- Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use the finally block.
- The finally block always gets executed, whether an exception occurred or not.

Exception Hierarchy

ERRORS

- Errors are exceptional scenarios that are out of the scope of application, and it’s not possible to anticipate and recover from them.
- For example, hardware failure, Java virtual machine (JVM) crash, or out-of-memory error.

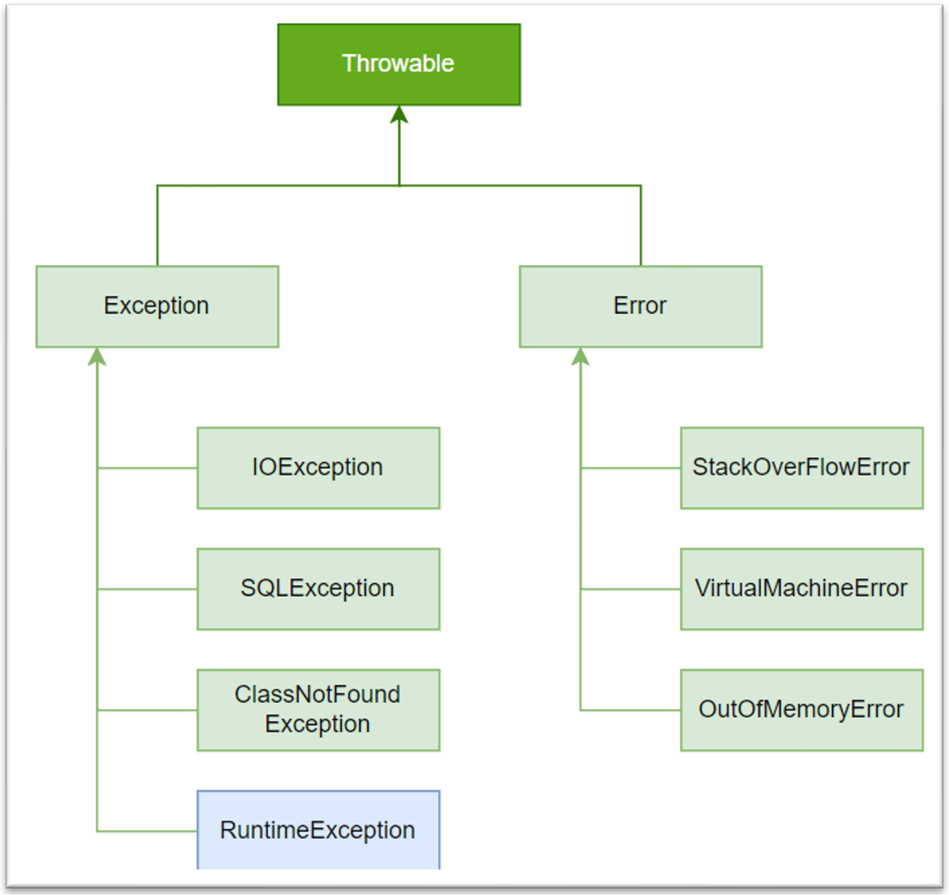
CHECKED EXCEPTIONS

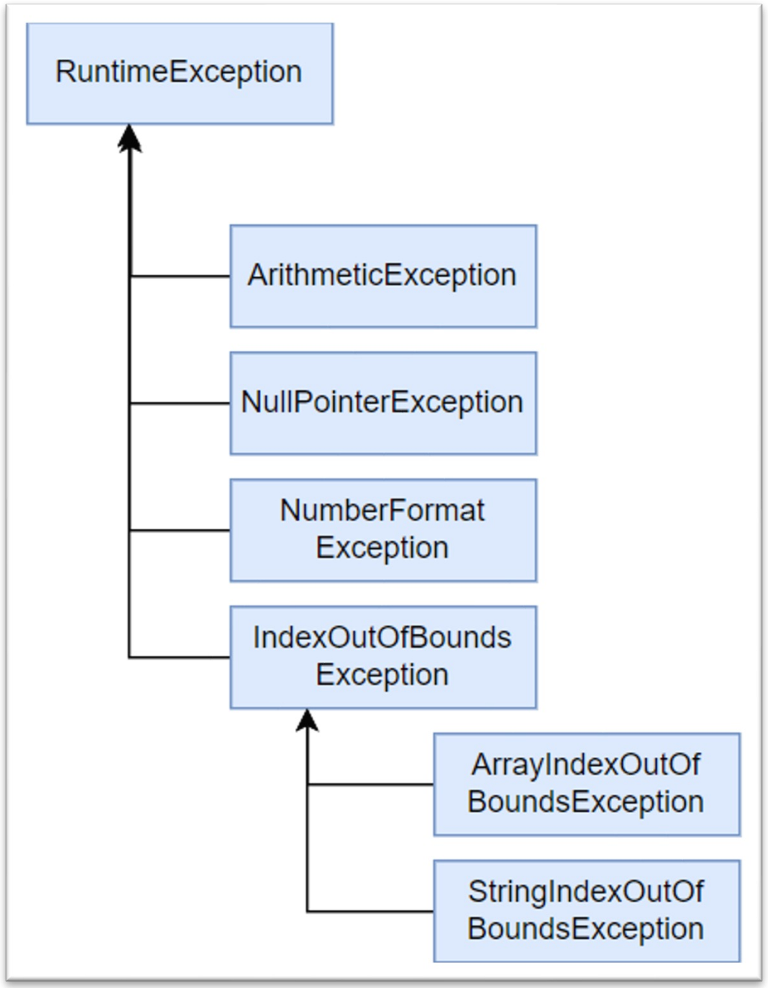
- Checked Exceptions are exceptional scenarios that we can anticipate in a program and try to recover from it.
- For example, FileNotFoundException.
- If we are throwing a Checked Exception, we must catch it in the same method, or we have to propagate it to the caller using the throws keyword.

RUNTIME EXCEPTIONS

- Runtime Exceptions are caused by bad programming.
- For example, trying to retrieve an element from an array which does NOT exist.
- If we are throwing any Runtime Exception in a method, it’s not required to specify them in the method signature throws clause. Runtime exceptions can be avoided with better programming.

Exception Handling Hierarchy



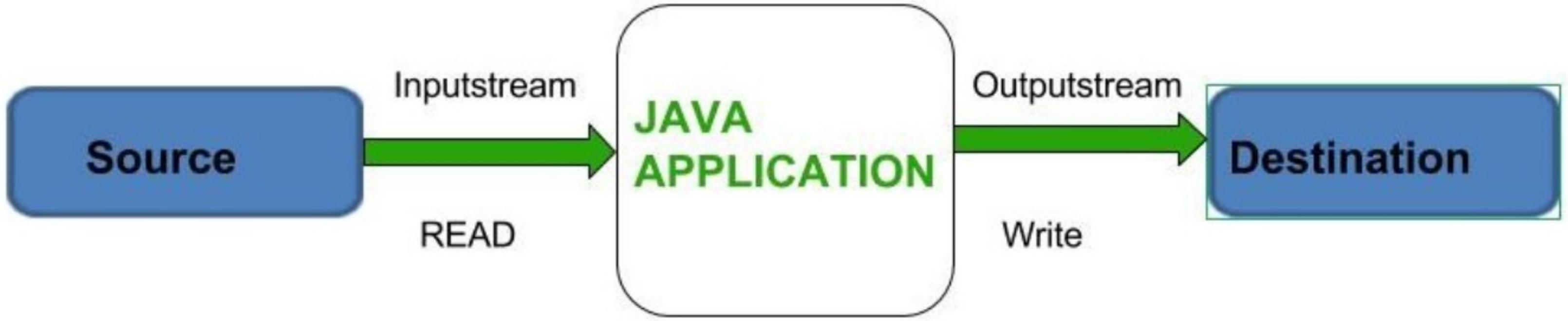

RuntimeException
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOfBounds Exception
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

Working with Files

Streams Are The Essence

Streams

Java provides various Streams with its **I/O package** that helps the user to perform all the input-output operations.



Types of Streams

Depending on the type of operations, streams can be divided into two primary classes:

Input Stream	Used to read data that must be taken as an input from a source array or file or any peripheral device
	Examples are `FileInputStream, BufferedInputStream, ByteArrayInputStream` etc.
Output Stream	used to write data as outputs into an array or file or any output peripheral device.
	Examples are `FileOutputStream, BufferedOutputStream, ByteArrayOutputStream` etc

Exercise : Streams

- Prints the characters available in file **`source.txt`** to the standard output using **`FileReader`**.
- Copy the data from **`source.txt`** to **`destination.txt`** using **`FileInputStream`** and **`FileOutputStream`**

SOLID Principles

Don’t Violate Them

SOLID Principles

A set of five design principles in object-oriented programming.

Promote the creation of more maintainable, flexible, and scalable software.

Introduced by **Robert C. Martin** and are a cornerstone of good software design.

The acronym SOLID stands for:

- SRP: Single Responsibility Principle
- OCP: Open/Closed Principle
- LSP: Liskov Substitution Principle
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

Single Responsibility Principle

A class should have **only one reason to change**, meaning it should have only one primary responsibility.

```
class UserManager {
   public void saveUsername(String username){}
   public void sendEmail(String emailBody){}
}
```

Promotes high cohesion and reduces coupling between different parts of the system.

```
class UserRegistration {
   public void saveUsername(String username){}
}

class EmailNotification {
   public void sendEmail(String emailBody){}
}
```

Open/Closed Principle

Software entities (classes, modules, functions) should be **open for extension but closed for modification.**

New functionality can be added without altering existing code, typically achieved through inheritance or interface

```
class BadAreaCalculator {
    public double calculateArea(Object shape){
        if(shape instanceof Circle){
            Circle circle = (Circle) shape;
            return Math.PI * circle.radius * circle.radius;
        } else if(shape instanceof Rectangle){
            Rectangle rectangle = (Rectangle) shape;
            return rectangle.width * rectangle.height;
        }
        return 0;
    }
}
```

Open/Closed Principle [Contd]

```
abstract class Shape{
    abstract public double calculateArea();
}
```

```
class Circle extends Shape{
    double radius;
    public Circle(double radius){
        this.radius = radius;
    }
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

```
class Rectangle extends Shape{
    double width;
    double height;
    public Rectangle(double w, double h){
        this.width = w;
        this.height = h;
    }
    @Override
    public double calculateArea() {
        return this.width * this.height;
    }
}
```

```
class GoodAreaCalculator {
    public double calculateAre(Shape shape){
        return shape.calculateArea();
    }
}
```

Liskov Substitution Principle

Subtypes must be **substitutable for their base types** without altering the program.

Ensures inheritance hierarchies are correctly designed and polymorphism works as expected.

```
class Bad_bird{
    public void fly(){
        System.out.println("Flying...");
    }
}

class Ostrich extends Bad_bird{
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Oops!
Can't fly.");
    }
}
```

Interface Segregation Principle

Clients should not be **forced to depend on interfaces** they do not use.

This encourages creating smaller, more specific interfaces rather than large, monolithic ones, leading to more focused and usable interfaces.

```
interface Animal{
   void feed();
}

class Dog implements Animal{
   @Override
   public void feed() {
      System.out.println("Dog can eat.");
   }
}

class Tiger implements Animal{
   @Override
   public void feed() {
      System.out.println("Tiger can eat.");
   }
}
```

Dependency Inversion Principle

High level modules should not depends on low level modules. Both should depend upon abstractions (interfaces), not on concreate classes

Promotes loose coupling and makes code more testable and flexible by relying on interfaces or abstract classes instead of concrete implementations.

```
// Low level module
class LightBulb {
    public void turnOn(){
        System.out.println("Light bulb is On.");
    }
    public void turnOff(){
        System.out.println("Light buld is off.");
    }
}
// High level module
class Switch {
    private LightBulb lightBulb;
    public void switchOn(){
        lightBulb.turnOn();
    }
    public  void switchOff(){
        lightBulb.turnOff();
    }
}
```

Day 02

Conclusion

- Inheritance
- Abstraction and Encapsulation
- Method overloading / Method overriding
- Threads & Concurrency
- Exception Handling
- Working with Files
- SOLID Principles