TODAY'S AGENDA

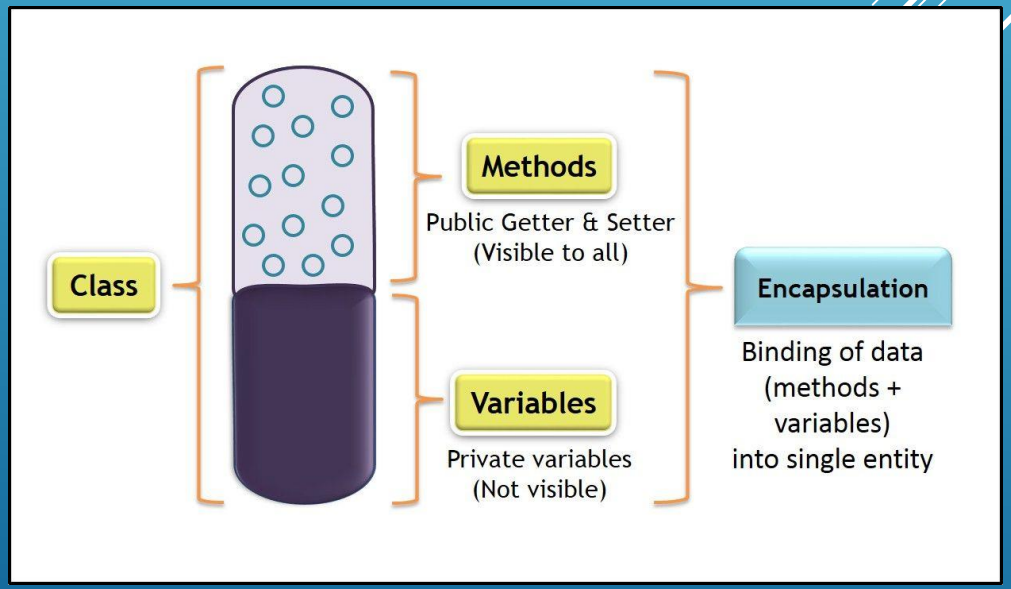
- Encapsulation
- Abstraction
  - Abstract Class
  - Interface
- Exceptional Handling (Try Catch)
- Keywords

ENCAPSULATION

- Encapsulation refers to wrapping up of data under a single unit. It is the mechanism that binds code and the data it manipulates.
- It is a protective shield that prevents the data from being accessed by the code outside this shield. In this, the variables or data of a class is hidden from any other class and accessed only through the methods of their current class. Therefore, it is also known as data hiding.
- It will create a capsule with data variables (private) and data methods (public).

Encapsulation in Java can be achieved by:

- Declaring the variables of a class as private.
- Providing public setter and getter methods to modify and view the variables values.



ENCAPSULATION

- **Data Hiding:** The user will have no idea about the inner implementation of the class. Even user will not be aware of how the class is storing values in the variables. They will only be aware that we are passing the values to a setter method and variables are getting initialized with that value.

- **Increased Flexibility:** Here, we can make the variables of the class as read-only or write-only depending on our requirement. In case you wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc.

- **Reusability:** It also improves the re-usability and easy to change with new requirements.

ENCAPSULATION EXAMPLE

```java
public class Company {

    public String name;
    public int totalEmp;
    private int budget;
    private String projects;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public int getTotalEmp() { return totalEmp; }

    public void setTotalEmp(int totalEmp) {  this.totalEmp = totalEmp; }

    public int getBudget() { return budget; }

    public void setBudget(int budget) { this.budget = budget; }

    public String getProjects() { return projects; }

    public void setProjects(String projects) { this.projects = projects; }
}
```

```java
public class CompanyTest {

    public static void main(String[] args) {

        Company obj = new Company();
        obj.setName("IBM");
        obj.setTotalEmp(5000);
        obj.setBudget(10000);
        obj.setProjects("AI - ROBOTICS");
  //With the help of all set methods I am trying to fill in the values. (setter)

//With the help of get methods I can access them. (getter)

        System.out.println(obj.getName());
        System.out.println(obj.getTotalEmp());
        System.out.println(obj.getProjects());
        System.out.println(obj.getBudget());
    }
}
```

ABSTRACTION

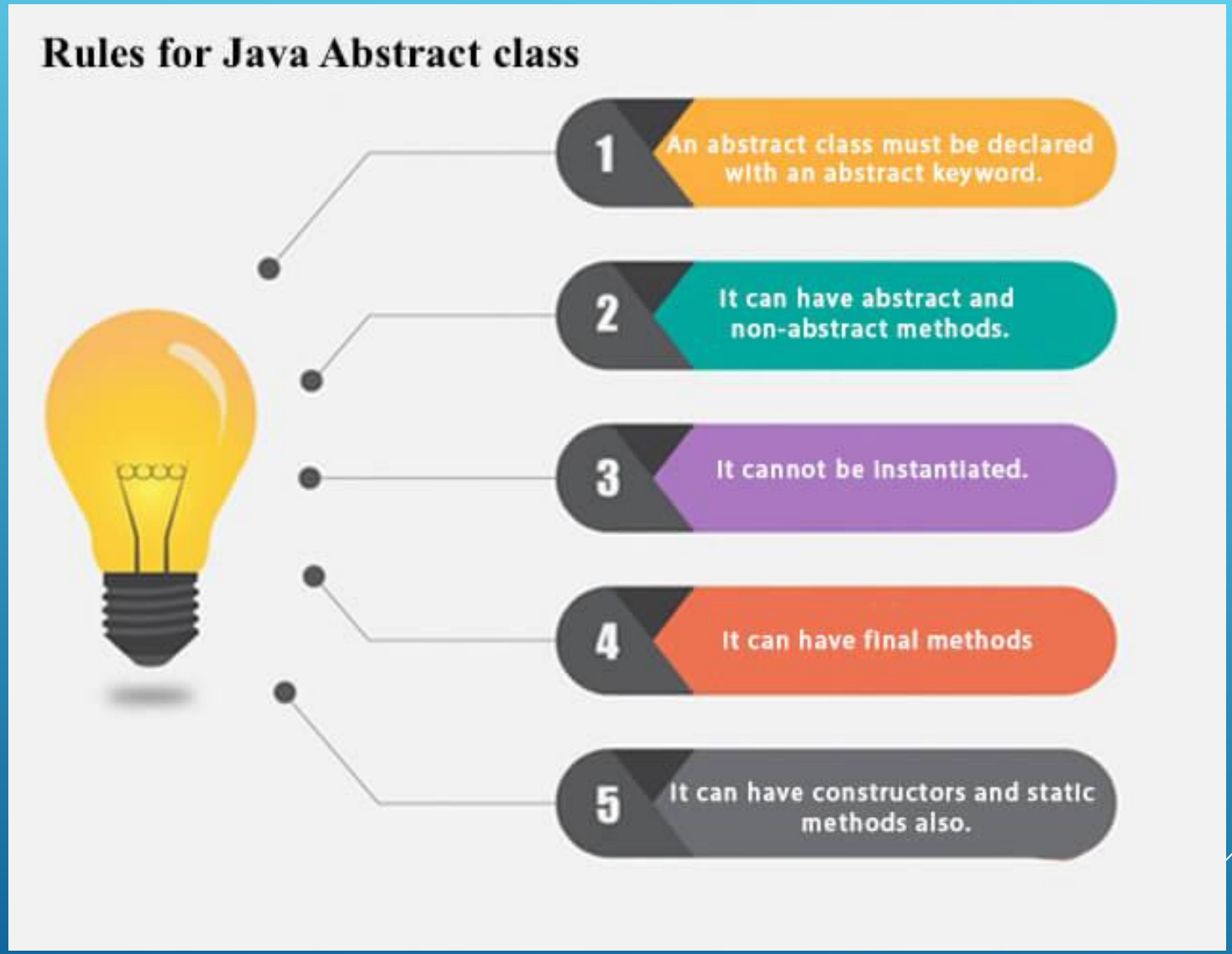
- Abstraction is a process of hiding the implementation details from the user, only showing the necessary functionality to the user. In other words, the user will have the information on what the object does instead of how it does it.

- Java Abstraction can be achieved in two ways:
  - Abstract Class (provides 0-100% abstraction)
  - Interface (provides 100% abstraction)

Without Abstraction

With Abstraction

ABSTRACT CLASS



EXAMPLE

```java
public abstract class Pages {

    public abstract void header();
    public abstract void title();

    public void logout() {
        System.out.println(“Non-abstract method");
    }
}
```

```java
public class TestPage {

    public static void main(String[] args) {

        LoginPage lp = new LoginPage();
        lp.header();
        lp.title();
        lp.login("admin", “test123");
        lp.logout();
    }
}
```

```java
public class LoginPage extends Pages{

    @Override
    public void header() {
        System.out.println("Login page - Header");
    }

    @Override
    public void title() {
        System.out.println("Login page - Title");
    }

    public int login(String un, String pwd){
        System.out.println("LP - login");
        System.out.println("login with "+ un + " and "+ pwd);
    }
}
```

INTERFACE

- An interface is declared by using the interface keyword.
- It provides 100% abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.
- Along with abstraction, the interface also helps to achieve multiple inheritance in Java.
- Cannot create object for interface

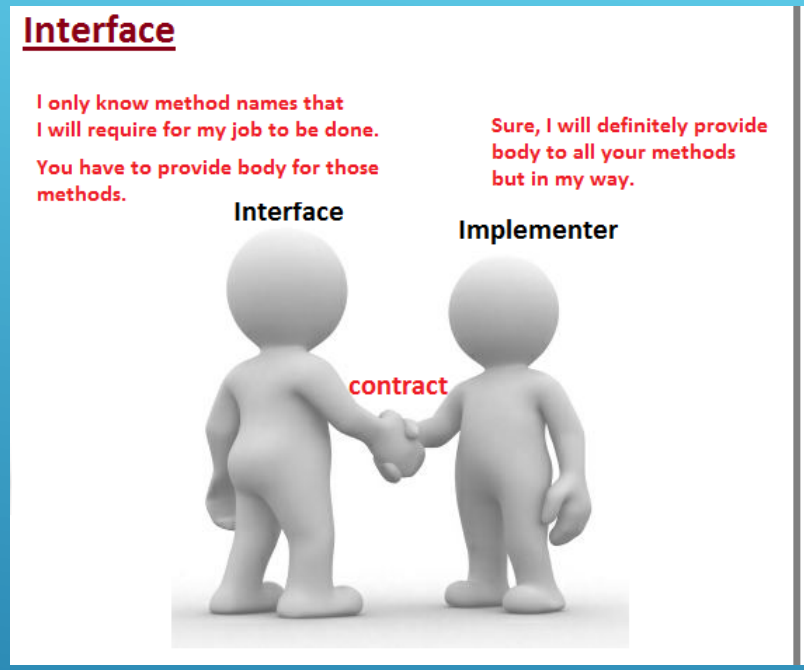
```
1. Interface <interface_name>{
2.
3.    // declare constant fields
4.    // declare methods that abstract
5. }
```

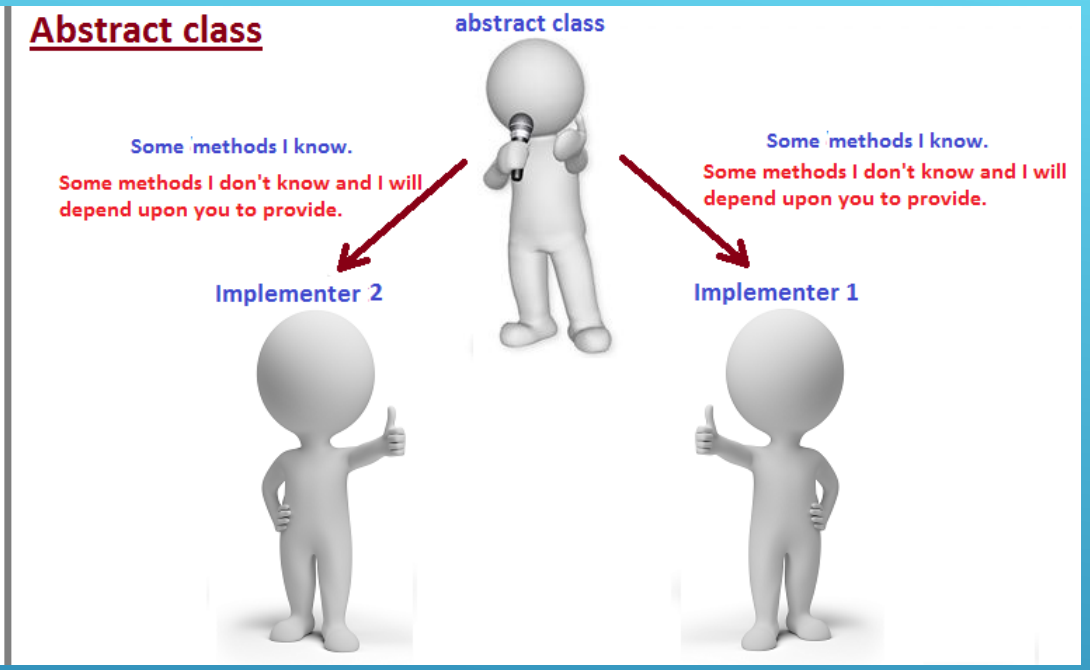
INTERFACE EXAMPLE

```
interface FirstInterface {

// interface method
public void myMethod();
}


interface SecondInterface {

// interface method
public void myOtherMethod();
}
```

```
class DemoClass implements FirstInterface, SecondInterface {

public void myMethod() {
        System.out.println("Implementation...");
}

public void myOtherMethod() {
        System.out.println(“Implementation...");
   }
}

class RunTest {

public static void main(String[] args) {

        DemoClass myObj = new DemoClass();
        myObj.myMethod(); myObj.myOtherMethod();
   }
}
```

OVERVIEW





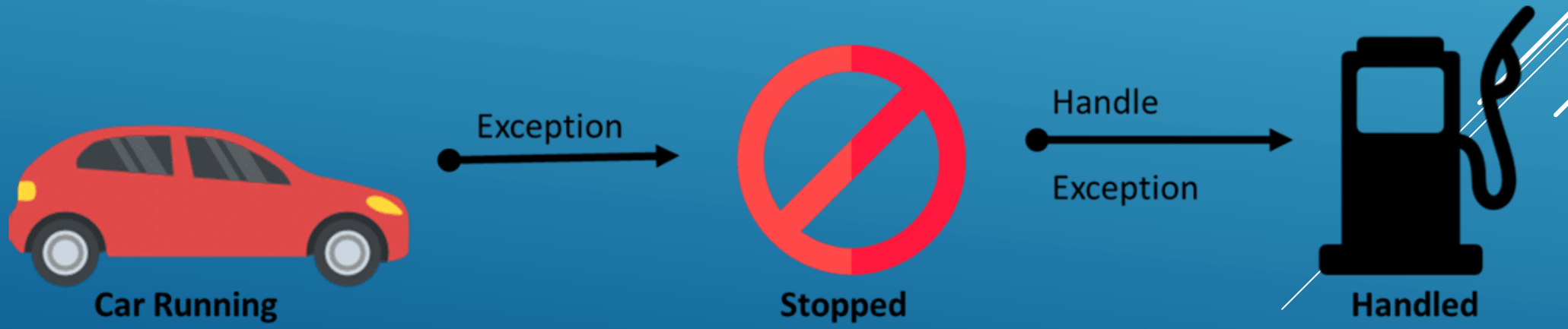
Abstraction	Encapsulation
Solves the problem in design level	Solves the problem in the implementation level
Used for hiding unwanted data and giving relevant results	Encapsulation means hiding the code and data into a single unit to protect data from the outside world
Outer layout – used in terms of design	Inner layout – used in terms of implementation

EXCEPTION HANDLING

- Exception handling is one of the most important feature of java programming that allows us to handle the runtime errors caused by exceptions.

What is an exception?

- An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.



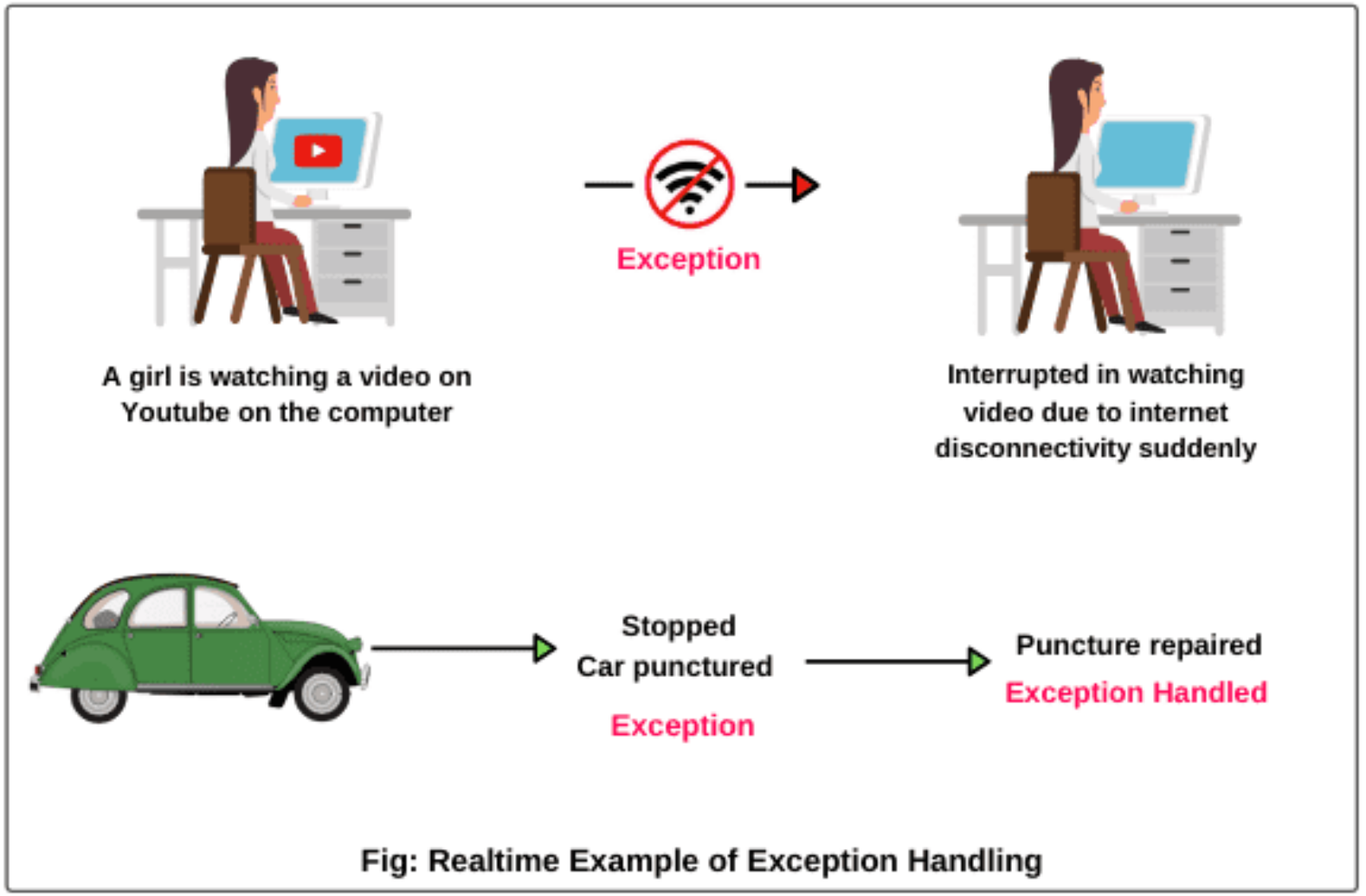


Fig: Realtime Example of Exception Handling

Why an exception occurs?

- There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Exception Handling

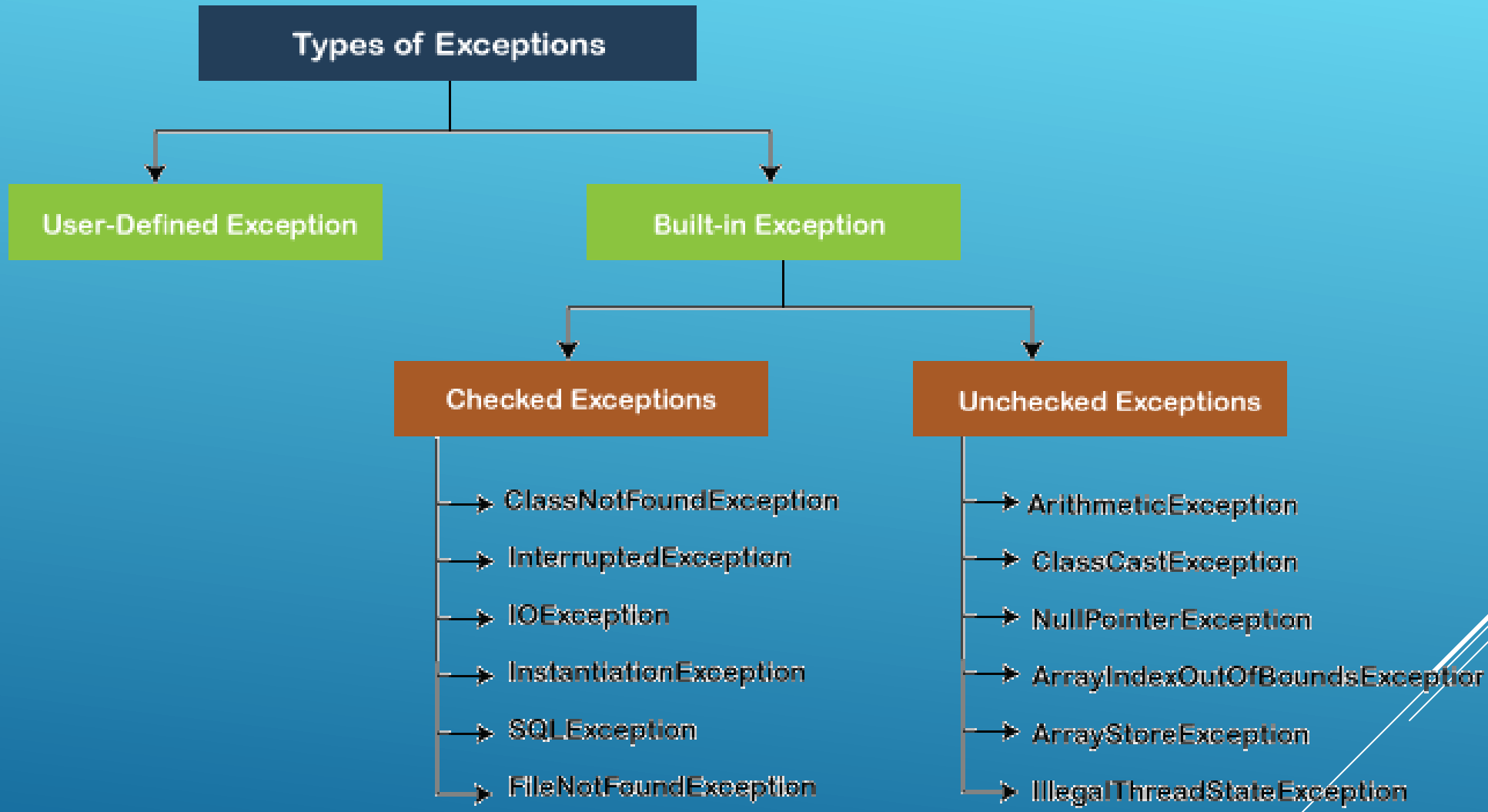
- If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user.

Advantage of exception handling

- Exception handling ensures that the flow of the program doesn't break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.
  By handling we make sure that all the statements execute and the flow of program doesn't break.

Errors Vs. Exceptions

Errors	Exceptions
Errors in java are of type java.lang.Error.	Exceptions in java are of type java.lang.Exception.
All errors in java are unchecked type.	Exceptions include both checked as well as unchecked type.
Errors happen at run time. They will not be known to compiler.	Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time.
It is impossible to recover from errors.	You can recover from exceptions by handling them through try-catch blocks.
Errors are mostly caused by the environment in which application is running.	Exceptions are mainly caused by the application itself.
Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError	Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundsException, ClassCastException, NullPointerException


Types of Exceptions
User-Defined Exception
Built-in Exception
Checked Exceptions
Unchecked Exceptions
ClassNotFoundException
InterruptedException
IOException
InstantiationException
SQLException
FileNotFoundException
ArithmeticException
ClassCastException
NullPointerException
ArrayIndexOutOfBoundsExceptior
ArrayStoreException
IllegalThreadStateException

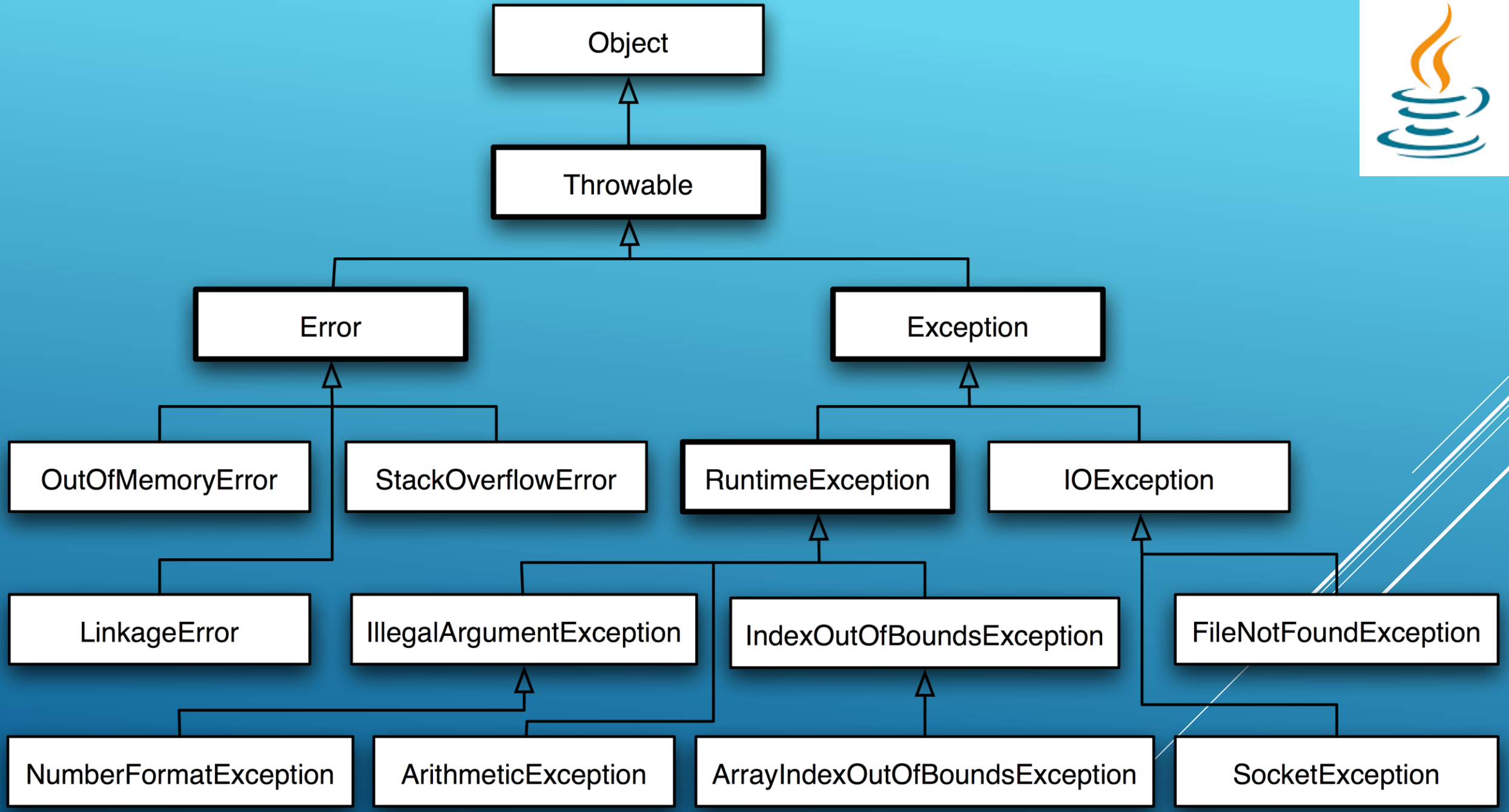
Checked exceptions

- All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

- Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Compile-time	Runtime
The compile-time errors are the errors which are produced at the compile-time, and they are detected by the compiler.	The runtime errors are the errors which are not generated by the compiler and produce an unpredictable result at the execution time.
In this case, the compiler prevents the code from execution if it detects an error in the program.	In this case, the compiler does not detect the error, so it cannot prevent the code from the execution.
It contains the syntax and semantic errors such as missing semicolon at the end of the statement.	It contains the errors such as division by zero, determining the square root of a negative number.


Object
Throwable
Error
Exception
OutOfMemoryError
StackOverflowError
RuntimeException
IOException
LinkageError
IllegalArgumentException
IndexOutOfBoundsException
FileNotFoundException
NumberFormatException
ArithmeticException
ArrayIndexOutOfBoundsException
SocketException

TRY CATCH IN JAVA – EXCEPTION HANDLING

Try block
The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax of try block

try{ //statements that may cause an exception }

Catch block
A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes

Syntax of try catch
try { //statements that may cause an exception }
catch (exception(type) e(object)) { //error handling code }

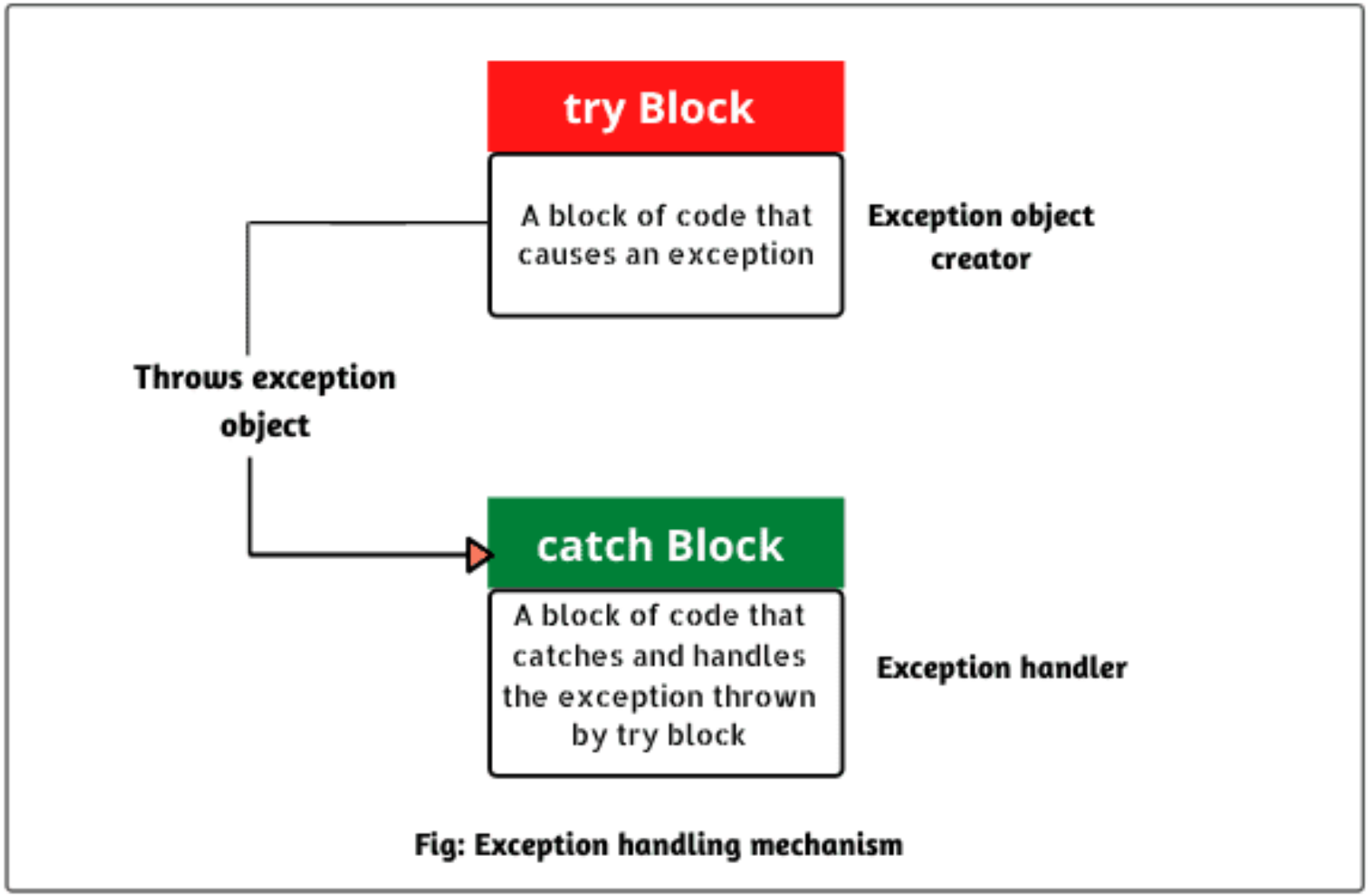
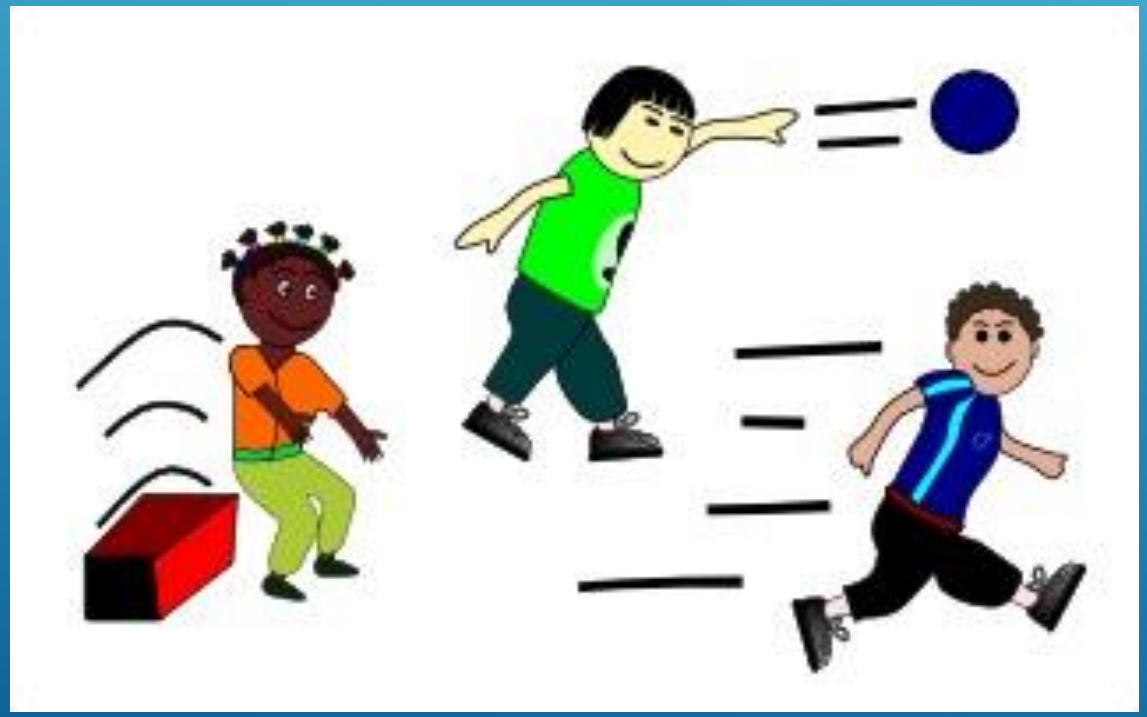


Fig: Exception handling mechanism

THROWS KEYWORD

- The throws keyword is used to declare the exception information that may occur during the program execution. It gives information about the exception to the programmer. It is better to provide the exception handling code so that the normal flow of program execution can be maintained.



HOMEWORK

- Check your classmate homework and identify mistakes
- Check in all code so far in GitHub
- Revise topics so far we have learnt in Java.

Deadline: Wednesday Midnight Latest