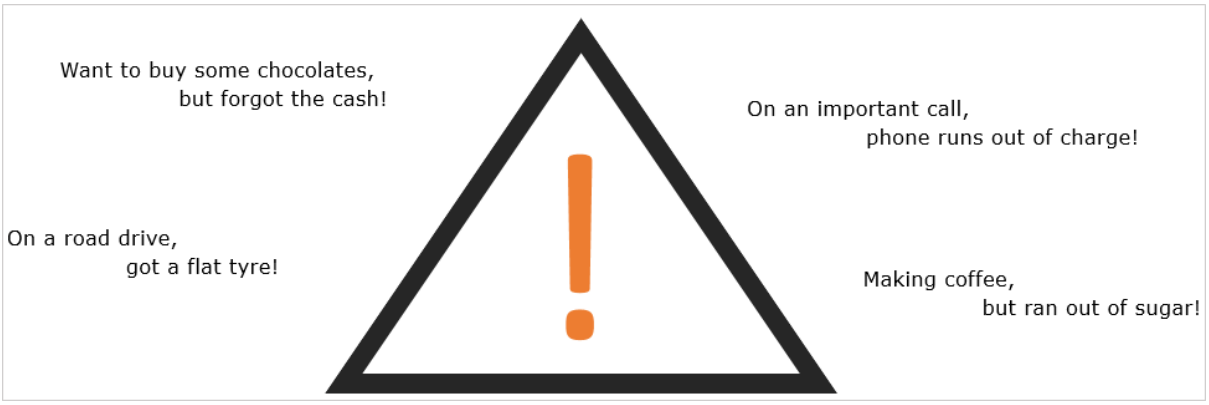
Exception-Introduction

Everyday we come across something that doesn't happen as expected, something that deviates from the normal expected flow of events. Unexpected situations or problems arise and we should deal with them.



Exception is an event which disrupts the normal flow of program during the program's execution. In an application, usually a certain input, or a programming mistake or an overlook can lead to exceptions. They come out as runtime errors and abnormally terminate the program.

Let's go through an example to understand how exceptions occur in code, and how we can take care of it.

Observe the code below:

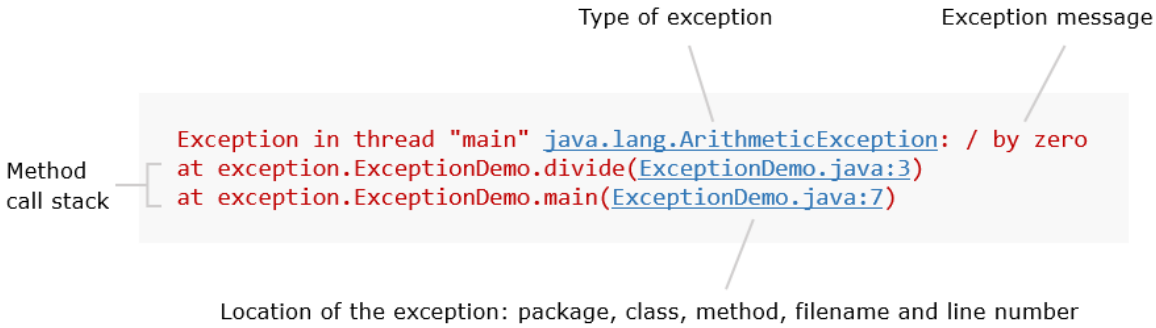
```
1. public class ExceptionDemo {
2.         public static void divide(int number1, int number2) {
3.                 int quotient = number1/number2;
4.                 System.out.println(quotient);
5.         }
6.
7.         public static void main(String args[]) {
8.                 divide(10, 0);
9.         }
10. }
```

What happens when we try to divide any number by 0? Mathematically it is infinity, but what will happen in the Java world? Let us execute the code and have a look.

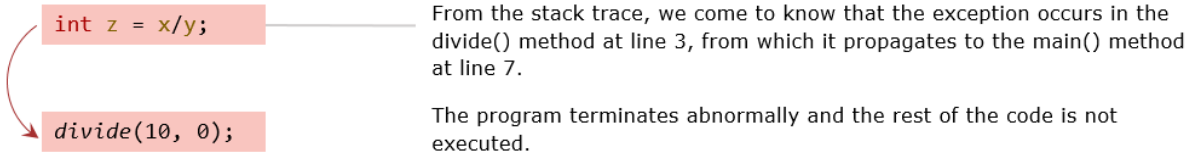
Exception- StackTrace

An exception occurs! The output is the stack trace of the exception. It tells us the type, message, method call stack, and the location of the exception, and hence, helps us debug it. Since we are trying to divide an integer by 0, Java throws an Arithmetic Exception as shown below.

Exception Hierarchy



As you can see above, the stack trace talks about the nature of Exception(Arithmetic Exception), the message contained in the exception (/ by zero) and where the exception was created (ExceptionDemo.java:3). But what does the last line of the stack trace indicate? Why is the line in the main method, where the divide() method called present in the stack trace?
This is due to the propagation of exception. The exception is travelling from the divide() function, to the main method and finally to the Runtime Environment of Java. More on exception propagation will be discussed later.



Till now we have seen the Arithmetic Exception in Java. Are there any more exceptions similar to it? If yes, is there any classification among them?

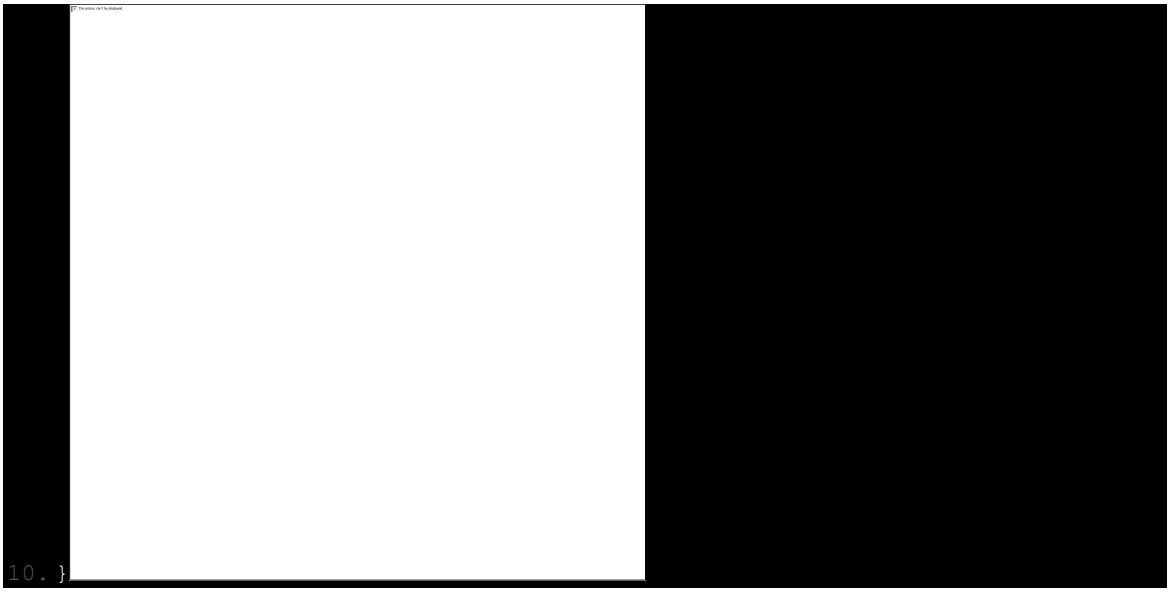
Exception Propagation

The exception objects in Java belong to the Exception class, and they contain the information pertaining to the exception, like the exception stack trace.

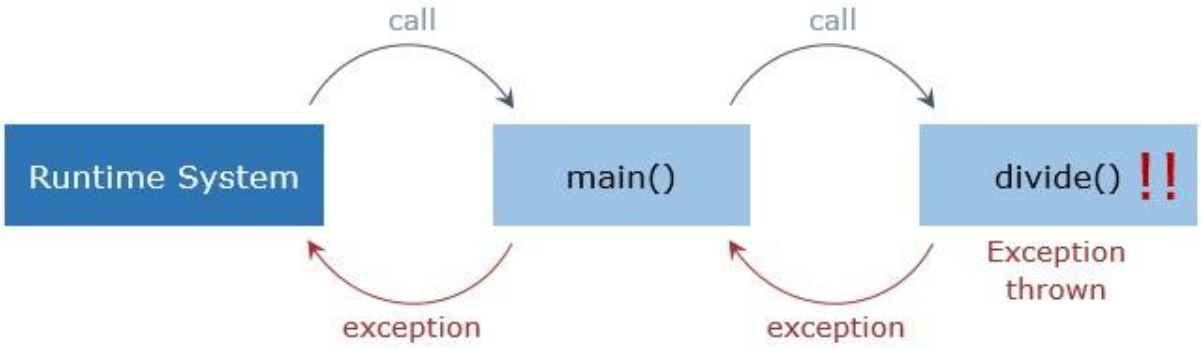
As soon as any exceptional event arises, an exception object is generated and thrown by the Java Runtime Environment(JRE). Any and all the executions stop as soon as an exception object is thrown. If the exception is not handled, it will be sent back to the calling environment which can be a calling method, or the runtime system.

Considering the previous code,

```
1.  public class ExceptionDemo {
2.          public static void divide(int number1, int number2) {
3.                  int quotient = number1/number2;
4.                  System.out.println(quotient);
5.          }
6.
7.          public static void main(String args[]) {
8.                  divide(10, 0);
9.          }
```



The control starts with the main method. The main method then calls on the divide method. Due to incorrect division operator in the divide method, an exception is created and thrown. This exception terminates any further execution of the program, but internally, the flow is different. The exception raised in the divide method is sent back to the main method. From the main method, the exception is given to the Runtime environment which executed the main method. This Runtime environment display the stack trace to the end user.



If there were any more non-exception throwing methods between main and divide, then the exception would have travelled through all of them before reaching the main method.

But why did the compiler not tell us about this exception before-hand? Are there any exceptions that even the compiler cannot detect?

Exception Types

To make things easier and convenient, Java provides excellent exception handling mechanisms.

Whenever there is a chance of an exception to occur in a method, we have two choices:

- Handle the Exception in the method
- Propagate it to the called method.

Exceptions are broadly classified into two types,

Checked Exceptions:

- If not handled by the programmer, these exceptions will be **detected during the compilation of the program** which will result in compilation errors.
- Programmers **are forced** to handle these exceptions or declare its propagation to the calling environment.

Unchecked Exceptions:

- These exceptions are **detected during the execution of the program or the runtime**, hence causing an error.
- Programmers **are neither forced** to handle it **nor** declare its propagation.

The try - catch Block

Handling exceptions involves using the try-catch block for constructing an exception handler.

```
try {
    // Code that can throw exceptions
}
catch(Exception1 exception1) {
    // Code for handling Exception1
}
catch(Exception2 exception2) {
    // Code for handling Exception2
}
```

The code that can throw an exception is enclosed inside the try block. One or many catch blocks succeed the try block.

A catch block handles the exception specified as its argument. A catch block can accept objects of type Throwable or its subclasses only.

Now let's create an exception handler block for our ArithmeticException example.

```
public static void divide(int number1, int number2) {
    try {
        int quotient = number1/number2;
        System.out.println(quotient);
    }
    catch(ArithmeticException exception) {
        System.out.println("The divisor should not be zero");
    }
}
```

Observe how the code has now become free from conditional logic making it more readable.

Catching Exceptions

Whenever the statements in the try block throw an exception, it is immediately caught by the first matching catch block which can handle it. The code inside the try block following the line causing the exception is ignored.

Once the catch block catches and handles the thrown exception, the program execution starts from the end of the try-catch block.

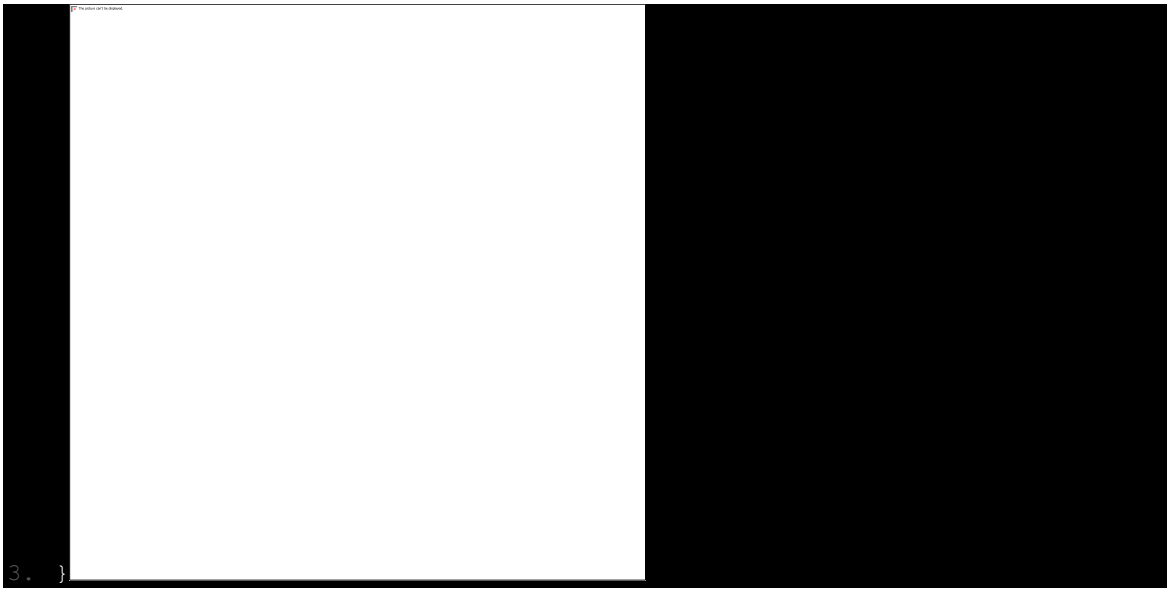
```
1.  public static void divide(int number1, int number2) {
2.      try {
3.          int quotient = number1/number2;       // If an exception occurs
    here, the control jumps to the first matching catch block
4.          System.out.println(quotient);      // Execution of this line will be
    skipped
5.      }
6.      catch(ArrayIndexOutOfBoundsException exception) {
7.          System.out.println("Index not found");
8.      }
9.      catch(ArithmeticException exception) {       // This is the matching
    exception handler
10.         System.out.println("The divisor should not be zero");
11.     }
12.     System.out.println("Method execution ends");       // Program execution
    will continue from this line
13. }
```

The exception will remain unhandled and will be propagated to the calling function if a matching catch block is not found.

If no exception is thrown inside a try block, the catch blocks following it are ignored.

A catch block that can handle objects of Exception class can catch all the exceptions. This should always be the last catch block in the catch sequence.

```
1.  catch(Exception e) {
2.       // Code for handling exception
```



```
}
```

Note: Nesting is allowed in try-catch blocks.

From Java 7 on wards, a multi-catch block is also allowed. This is where a single catch block is used to catch different exceptions in the same level of hierarchy.

```
catch(ExceptionType1 | ExceptionType2 | ExceptionType3 exception) {
    // Code for exception handling
}
```

Consider the division scenario, but now instead of primitive data type int, we use a wrapper class Integer. In this scenario, there can be the case where the divisor is 0, or there can also be the case where one Integer object is null instead of holding a numeric value. Here we have two exceptions to be careful of, ArithmeticException for when the divisor is 0 and NullPointerException for when any Integer object is null. Since both of these exceptions enjoy the same level of hierarchy, we can use a multi-catch block for them.

```
public static void divide(Integer number1, Integer number2) {
    try {
        int quotient = number1/number2;
        System.out.println(quotient);
    }
    catch(ArithmeticException | NullPointerException exception) {
        System.out.println("Some Error Occurred");
    }
}
```

finally Block

An exception inside a try block causes the rest of the code to be skipped. This might lead to the important parts of the code not being executed.

Code which closes connections, releases resources, etc. need to be executed in all the conditions. Keeping them inside the try block cannot guarantee their execution.

In such situations, the finally block plays an important role. It always executes, irrespective of occurrence of any exception.

```
try {
    // Code that can throw exceptions
}
catch(Exception1 exception1) {

```

```
}
finally {
    // Code to be executed no matter what
}
```

If any exception occurs or System.exit() is invoked in the finally block, the finally block will disrupt.

If we have to include the finally block in the divide scenario, it will look like,

```
public static void divide(int number1, int number2) {
    try {
        int quotient = number1/number2;
        System.out.println(quotient);
    }
    catch(ArithmeticException exception) {
        System.out.println("The divisor should not be zero");
    }
    finally {
        System.out.println("At the end of the divide method");
    }
}
```

Note: One or more catch blocks or a finally block should succeed a try block.

throw Keyword

Exceptions are generated when some predefined exceptional events like division by zero occurs in a program. But what if there is a need to terminate a process in our program due to some circumstances?

Consider a scenario of a customer trying to purchase something on a E-Retail website. The customer wants to buy 3 pairs of socks. But the website has only 2 pairs. Shouldn't the program stop its execution or at least halt it, while it conveys this issue to the customer? Using any conditional statement in this scenario will be very expensive. Hence to halt this process, we make use of the throwability factor of the exception. Till now the the Java program created an threw any exception on its own. But now, we have to handle the reins of the same.

Java allows us to explicitly generate or throw exceptions using the throw keyword:

```
Exception e = new Exception();
throw e;
```

Any object of type Throwable can be thrown.

Exceptions also accept a message for themselves:

```
throw new Exception(<<message in String format>>);
```

Having exceptions with custom messages increases the readability of our applications.

Now let's see how we can use this in our example.

```
public static void checkStock(int stockAvailable, int quantityRequired) {
    try {
        if(stockAvailable < quantityRequired)
            throw new Exception("There is not enough stock available.");
        System.out.println("Please proceed to the check-out");
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

The above code will generate an exception with the given message if the condition is satisfied. Here we have created an exception object of the Exception class. If need be, you can use any of the sub-classes of the Exception class to build your exception object. You can also create your own custom exceptions, which we will see further on.

throws Keyword

Until now, we have been handling exceptions in the method in which they are thrown. What if we need to propagate and handle the exceptions elsewhere!

If there is a checked exception which the method doesn't handle, it has to be declared using the **throws** clause:

```
1. public static void checkStock(int stockAvailable, int quantityRequired) {
2.     if(stockAvailable < quantityRequired)
3.         throw new Exception("There is not enough stock available.");
4.     System.out.println("Please proceed to the check-out");
5. }
```

There are mainly three scenarios for when the **throws** keyword will come in play in a Java program. To look into these scenarios, let us consider the following code where the stock availability is checked. (Note: The below code will have compilation errors in checkStock() method due to the exception not being handled.)

```
1. class MobileShopee{
2.     static int stockAvailable = 400;
3.     public static void checkStock(int quantityRequired) {
4.         if(stockAvailable < quantityRequired)
5.             throw new Exception("There is not enough stock available.");
6.         System.out.println("Please proceed to the check-out");
7.     }
8.     public static void buyMobiles(int quantityRequired) {
9.         checkStock(550);
10.        System.out.println("Please pay for the items in your cart.");
11.    }
12.    public static void main(String[] args) {
13.        buyMobiles(550);
14.    }
15. }
```

First Scenario:

The exception is not handled in the method in which the exception is not being created rather is being propagated and handled by the method calling it.

```
1. class MobileShopee {
2.     static int stockAvailable = 400;
3.     public static void checkStock(int quantityRequired) throws Exception{
4.         if(stockAvailable < quantityRequired)
5.             throw new Exception("There is not enough stock available.");
6.         System.out.println("Please proceed to the check-out");
7.     }
8.     public static void buyMobiles(int quantityRequired) {
9.         try{
10.            checkStock(550);
11.            System.out.println("Thank you for shopping at MobileShopee");
12.        } catch(Exception exception) {
13.            System.out.println(exception.getMessage());
14.        }
15.    }
16.    public static void main(String[] args) {
17.        buyMobiles(550);
18.    }
19. }
```

In this case, if the exception is thrown in the checkStock() method, the exception will get propagated to the buyMobiles() method and will be handled there.

Second Scenario:

The exception is not handled the method in which the exception is created nor in any intermediate methods, rather will be propagated to and handled by the main method.

```
1.  class MobileShopee{
2.      static int stockAvailable = 400;
3.      public static void checkStock(int quantityRequired) throws Exception{
4.          if(stockAvailable < quantityRequired)
5.              throw new Exception("There is not enough stock available.");
6.          System.out.println("Please proceed to the check-out");
7.      }
8.      public static void buyMobiles(int quantityRequired) throws Exception{
9.          checkStock(550);
10.         System.out.println("Please pay for the items in your cart.");
11.     }
12.     public static void main(String[] args) {
13.         try{
14.             buyMobiles(550);
15.         } catch (Exception exception) {
16.             System.out.println(exception.getMessage());
17.         }
18.     }
19. }
```

In this case, if the exception is thrown in the checkStock() method, the exception will get propagated to the buyMobiles() method. Since we have declared a throws keyword even in the buyMobiles() method, this exception will get propagated to the main method where it will be handled.

Third Scenario:

The exception will not be handled by any method of the program, rather will be propagated to the Runtime Environment.

```
1.  class MobileShopee{
2.      static int stockAvailable = 400;
3.      public static void checkStock(int quantityRequired) throws Exception{
4.          if(stockAvailable < quantityRequired)
5.              throw new Exception("There is not enough stock available.");
6.          System.out.println("Please proceed to the check-out");
7.      }
8.      public static void buyMobiles(int quantityRequired) throws Exception{
9.          checkStock(550);
10.         System.out.println("Please pay for the items in your cart.");
11.     }
12.     public static void main(String[] args) throws Exception{
13.         buyMobiles(550);
14.     }
15. }
```

In this case, if any exception is thrown from the checkStock() method, it will get propagated to the buyMobiles() method, which will send it to the main method. Since we have used the throws keyword even in the main method, the exception will get propagated to the Runtime exception. The Runtime exception will then print the exception stack trace in the output window.

Note: Usage of the third scenario is frowned upon as the exception is not being handled in it. One of the main purpose of handling the exception was so that the end user is not shown the exception stack trace, which is neglected in the third scenario. Hence avoid the use of it as much as possible.

User - Defined Exceptions

Till now you have seen and used the exception provided by Java. But what if you wanted to create an exception that suited your business requirements?

Let us understand this with a scenario. Consider an online retail shop. It allows the customers to buy the products displayed to them. The customer can buy anything he/she wants until and unless the product is in stock. What should happen if the product goes out of stock, or the quantity requested by the customer is not available? The program should stop with an appropriate message displayed to the customer.

In such a scenario, the default exceptions of Java may not help as much, instead we need an exception of our own creation. Having user-defined exceptions not only increases the flexibility of our applications but also makes the code more manageable.

Creating a user-defined exception is very simple. A user-defined exception is any class that is a subclass of the Exception class. In other terms, if we extend the Exception class, we get our user-defined exception.

```
1. public class StockNotAvailableException extends Exception {}
```

With this, a user-defined exception class, StockNotAvailableException is created. StockNotAvailableException is an exception class which does not contain any message. If we want an exception message to be associated with this class, we have to add a parameterised constructor to this exception class as shown in the below code.

```
1. public class StockNotAvailableException extends Exception {
2.     public StockNotAvailableException(String message){
3.         super(message);
4.     }
5. }
```

Use of a parameterised constructor allows us to access and modify the message attribute of the superclass. Now let us see where and how we can use this user-defined exception.

```
1. public void checkStock (int stockAvailable, int quantityRequired) throws
   StockNotAvailableException {
2.     if(stockAvailable < quantityRequired)
3.         throw new StockNotAvailableException("The required quantity is not
   available.");
4. }
```

The above code will throw an instance of StockNotAvailableException with the given message, when the stock available is less than the quantity required for the customer.

Here we have extended the main Exception class thereby making our exception a checked exception. A user-defined exception can also be made an unchecked exception by extending the RuntimeException class or any of its subclasses.

Best Practices – Introduction

Every application developed in any language will always have one thing in common, that is a way to handle the various exceptions that are thrown during the execution. And just like any other concept in Java, there are certain rules/practices that every developer should follow while handling exceptions to get the best results and performance from the application.

The following are the best practices that will vastly improve the *reliability*, *readability*, and *maintainability* of any application by many folds.

- Throw specific Exceptions rather than generic ones
- Proper usage of **finally** block
- Use the **catch** block only if an exception is supposed to be handled
- Give proper exception message when throwing an exception

- Include the cause of exception when rethrowing an exception
- **NullPointerException** should never be caught

Let us discuss each best practice in detail.

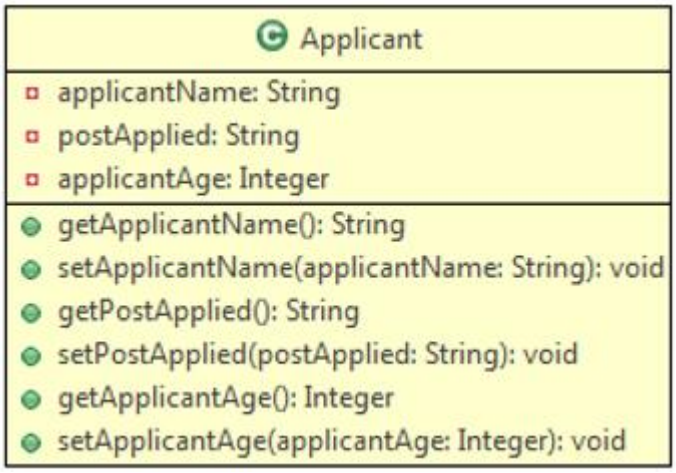
Exception Handling - Exercise 1

Problem Statement

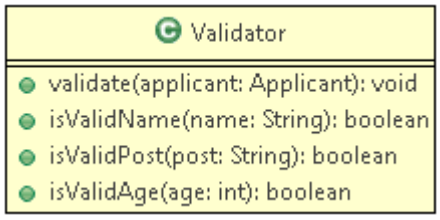
Infy Bank wants to conduct examinations for the post of Probationary Officers, Assistants, and Special Cadre Officers. It has rolled out an online application which is available on the Bank's website. The applicants can fill in the application form and submit it with accurate details.

Assuming that each **Applicant** is represented by the following class diagram:

Define an user-defined Exception **InfyBankException** according to following class diagram:



Design a **Validator** class which has methods for validating applicant details according to the following class diagram:



Method Description:

- **validate(Applicant applicant):**
    - This method receives the Applicant and calls the respective methods to validate the values. If validation fails, it throws user-defined exceptions **InfyBankException** with the exception message as given below:

Violation for	User defined exception	Exception message
Applicant name	InvalidNameException	Invalid applicant name
Post	InvalidPostException	Invalid post
Age	InvalidAgeException	Invalid age exception

If all values are valid print the following message:

```
All the values are valid.
```

- **isValidApplicantName(String name):**
    - This method validates applicantName.
    - It cannot be null or empty.
    - If the rule is violated then it should return false else it should return true.

- **isValidPost(String post):**
    - This method validates the post the applicant applied for.
    - It should be one among one of the following posts: "Probationary Officer", "Assistant", or "Special Cadre Officer".
    - If the rule is violated then it should return false else it should return true.

- **isValidAge(Integer age):**
    - This method validates the age of the applicant.
    - It should be greater than 18 years and less than 35 years.
    - If the rule is violated then it should return false else it should return true.

Create a class **Tester** and implement it as shown below:

1. Create an object of the Applicant class and populate it with values.

2. Invoke validate() method of Validator class to validate the values.

3. If any exception is thrown, catch the exception and print the exception message.

Note: You can change the input as needed to get a different output.

Sample:

Input:

Name	Jason
Post	Assistant
Age	37

Output:

Invalid age exception

Note: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

Code in java

```
class Applicant{
	private String applicantName;
	private String postApplied;
	private Integer applicantAge;
	public String getApplicantName() {
		return applicantName;
	}
	public void setApplicantName(String applicantName) {
		this.applicantName = applicantName;
	}
	public String getPostApplied() {
		return postApplied;
```

```
	}
	public void setPostApplied(String postApplied) {
		this.postApplied = postApplied;
	}
	public Integer getApplicantAge() {
		return applicantAge;
	}
	public void setApplicantAge(Integer applicantAge) {
		this.applicantAge = applicantAge;
	}

}
class InfyBankException extends Exception{
    public InfyBankException(String message){
       super(message);
    }
}

class Validator {
	public void validate(Applicant applicant){
		//code here
	}

	public boolean isValidName(String name) {
		//code here
		return false;
	}

	public boolean isValidAge(int age) {
```

```
            //code here
            return false;
        }

        public boolean isValidPost(String name) {
            //code here
            return false;
        }

}
class Tester{
        public static void main(String[] args) {
            //code here
        }
}
```