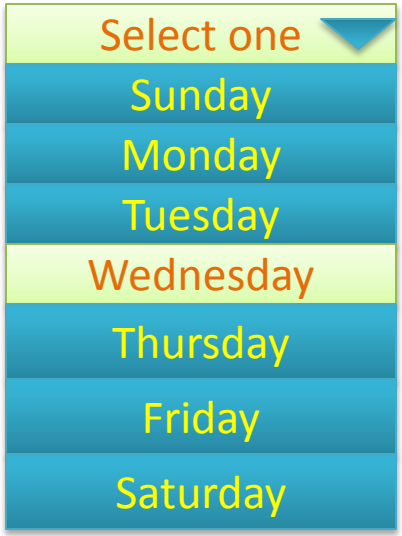
edureka!

Enums in Java

Enums

- **Enums** helps to relate the variables with related constants so that it will be flexible to work.
- We use "**enum**" keyword.
- E.g: enums can be used in dropdown boxes.



Why we need to use Enum?

edureka!

- Enum is **type-safe** i.e any constants can not be assigned to that variables outside the enum definition.
- **Adding new constants** will be easy without disturbing already present code.
- You can also **assign different constants** to variables other than default values.

How to declare an Enum?

edureka!

- Declaring an **enum** is similar to class.
- Should be declared **outside the class** in which it has to be used **or** in an **interface**.

variables which will be assigned constant values

- **enum** Colors{

  Red, Green, Blue, White, Yellow

  }

name of enum **0** **1** **2** **3** **4(default constants assigned)**

Simple program for Enum

edureka!

```
enum Colors_enum{red , green , blue , white , yellow}
public class Main {
public static void  main(String args[]) {
    Colors_enum colors[]=Colors_enum.values();
    for(Colors_enum c:colors)
        {
            System.out.println(c);
        }
    }
}
```

How to assign constants to Enum by user

edureka!

```
enum Chocolates{
    dairymilk(20) ,
    kitkat(10) ,
    munch(5);
    int cost;
    Choloclates(int cost)
    {
        this.cost=cost;
    }
}
```

```
public class Main {
public static void  main(String args[]) {
        Chocolates favouritechoco=Chocolates.dairymilk;
        switch(favouritechoco)
        {
            case dairymilk: System.out.println("Dairy Milk");
                                break;
            case kitkat: System.out.println("Kitkat");
                                break;
            case munch: System.out.println("Munch");
                                break;
        }
    }
}
```

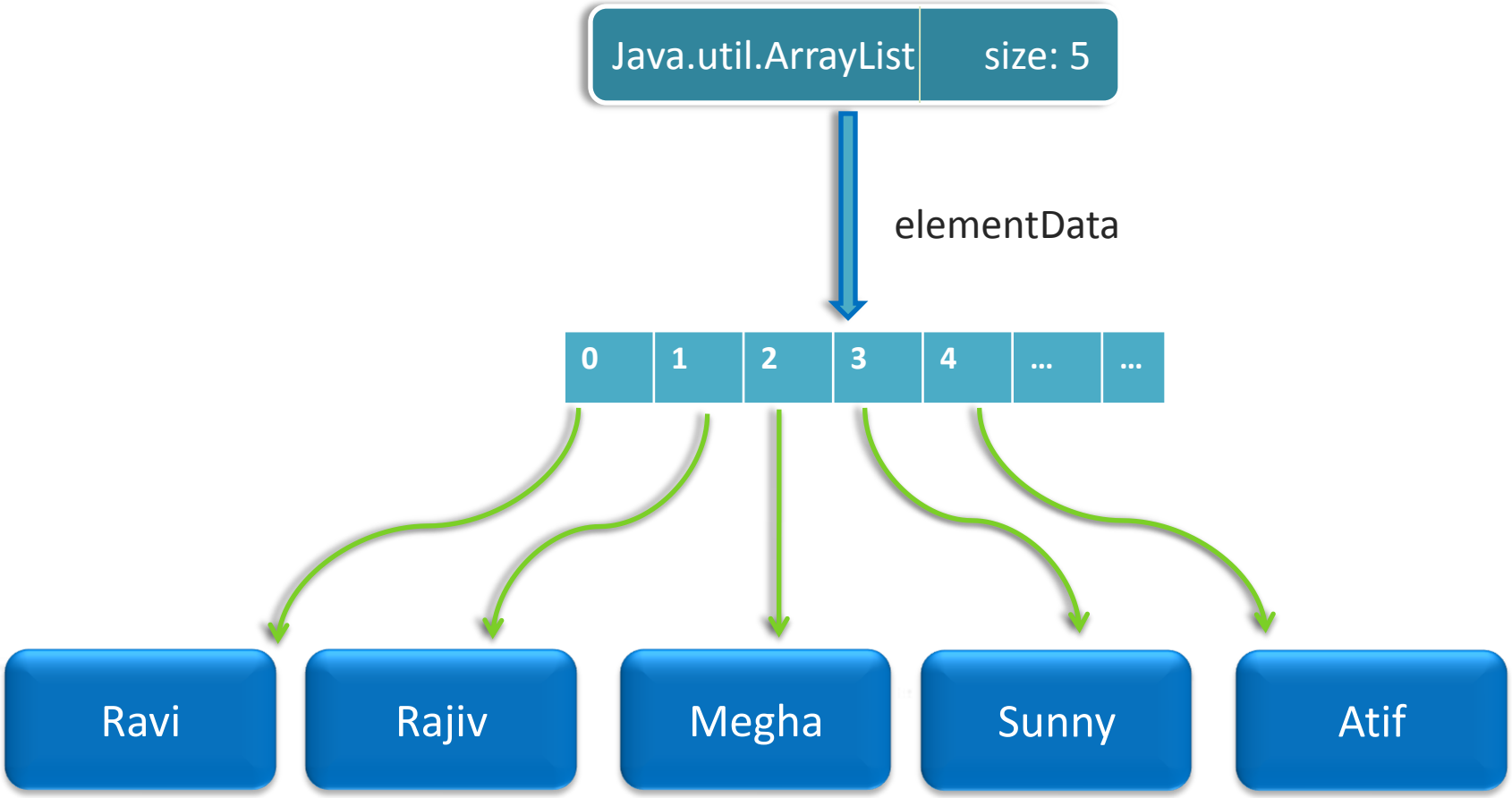
Array List

ArrayList class

edureka!

- The **ArrayList** class is a concrete implementation of the List interface.
- **Allows duplicate** elements.
- A list can **grow or shrink dynamically** where as array size is fixed once it is created.
  - If your application does not require insertion or deletion of elements, the most efficient data structure is the array

ArrayList class



Methods in ArrayList

edureka!

- boolean add(Object e)
- void add(int index, Object element)
- boolean addAll(Collection c)

- Object get(int index)
- Object set(int index,Object element)

- Object remove(int index)

- Iterator iterator()
- ListIterator listIterator()

- int indexOf()
- int lastIndexOf()

- int index(Object element)
- int size()
- void clear()

ArrayList - Insertion

edureka!

```
// Create an arraylist
ArrayList arraylist = new ArrayList();

// Adding elements
arraylist.add("Rose");
arraylist.add("Lilly");
arraylist.add("Jasmine");
arraylist.add("Rose");

//removes element at index 2
arraylist.remove(2);
```

How to trace the elements of ArrayList?

edureka!

- Iterator
- ListIterator
- For-each loop
- Enumeration

Iterator

edureka!

- **Iterator** is an interface that is used to traverse through the elements of collection.
- It traverses only in **forward direction** with the help of methods.

Iterator Methods

- boolean hasNext()
- element next()
- void remove ()

Displaying Items using Iterator

edureka!

```
Iterator iterator = arraylist.iterator();

while (iterator.hasNext()) {
    Object object = iterator.next();
    System.out.print(object + " ");
}
```

ListIterator

- **ListIterator** is an interface that traverses through the elements of the collection.
- It traverses in both **forward and reverse direction**.

ListIterator Methods

- boolean hasNext()
- element next()
- void remove ()
- boolean hasPrevious()
- element previous()

Displaying Items using ListIterator

edureka!

```
// To modify objects we use ListIterator
ListIterator listiterator =
arraylist.listIterator();

  while (listiterator.hasNext()) {
     Object object = listiterator.next();
     System.out.print(+ object + " ");
  }
```

For-each loop

edureka!

- It's action **similar to for loop**. It traces through all the elements of array or arraylist.
- **No need to mention size** of Arraylist.
- for (String s : arraylist_name)

Keyword — type of data stored in arraylist — name of arraylist

Enumeration

- **Enumeration** is an interface whose action is similar to iterator.
- But the difference is that it have **no method for deleting** an element of arraylist.

Enumeration Methods

- boolean hasMoreElement()
- element nextElement()

Displaying Items using Enumeration

edureka!

```
Enumeration  enumeration =
Collections.enumeration(arraylist);

while (enumeration.hasMoreElements()) {
    Object object = enumeration.nextElement();
    System.out.print(object + " ");
}
```

HashMaps

HashMap Class

edureka!

- The **HashMap** is a class which is used to perform operations such as inserting, deleting, and locating elements in a Map .
- The Map is an interface maps keys to the elements.
- Maps are **unsorted** and **unordered.**
- Map allows one null key and multiple null values
- **HashMap < K, V >**

  key **value** associated with key
- key act as indexes and can be any objects.

Methods in HashMap

edureka!

- Object put(Object key, Object value)

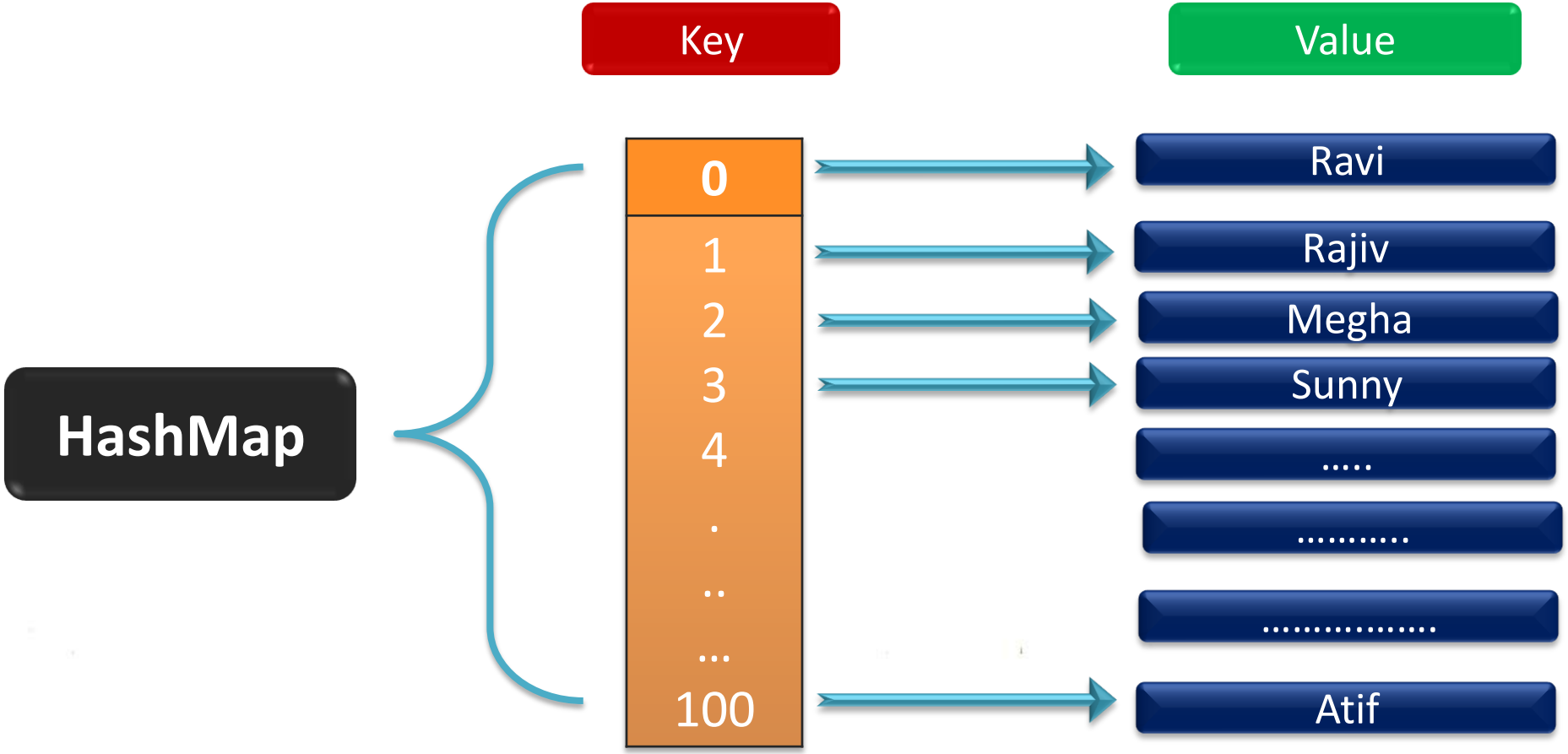
- Enumeration keys()
- Enumeration elements()
- Object get(Object keys)

- boolean containsKey(Object key)
- boolean containsValue(Object key)

- Object remove(Object key)
- int size()
- String toString()

HashMap Class

edureka!



HashMap - Insertion

edureka!

```
// Create a hash map
HashMap hashmap = new HashMap();

// Putting elements
hashmap.put("Ankita", 9634.58);
hashmap.put("Vishal", 1283.48);
hashmap.put("Gurinder", 1478.10);
hashmap.put("Krishna", 199.11);
```

HashMap - Display

edureka!

```
// Get an iterator
Iterator iterator = hashmap.entrySet().iterator();

// Display elements
while (iterator.hasNext()) {
    Map.Entry entry = (Map.Entry) iterator.next();
    System.out.print(entry.getKey() + ": ");
    System.out.println(entry.getValue());
}
```

Hashtable

Hashtable Class

edureka!

- **Hashtable** is a class which is used to perform operations such as inserting, deleting, and locating elements similar to HashMap .
- Similar to HashMap it also have key and value.
- It does not allow null keys and null values.
- The only difference between them is **Hashtable is synchronized** where as HashMap is not by default.

Methods in Hashtable

edureka!

- Object put(Object key, Object value)

- Enumeration keys()
- Enumeration elements()
- Object get(Object keys)

- boolean containsKey(Object key)
- boolean containsValue(Object key)

- Object remove(Object key)
- int size()
- String toString()

Hashtable - Insertion

edureka!

```
// Create a hash table
Hashtable hashtable = new Hashtable();

// Putting elements
hashtable.put("Ankita", 9634.58);
hashtable.put("Vishal", 1283.48);
hashtable.put("Gurinder", 1478.10);
hashtable.put("Krishna", 199.11);
```

Hashtable - Display

edureka!

```
// Using Enumeration
Enumeration enumeration = hashtable.keys();

// Display elements
while (enumeration.hasMoreElements()) {
    String key = enumeration.nextElement().toString();

    String value = hashtable.get(key).toString();

    System.out.println(key + ":"+value);
}
```

edureka!

Exceptions

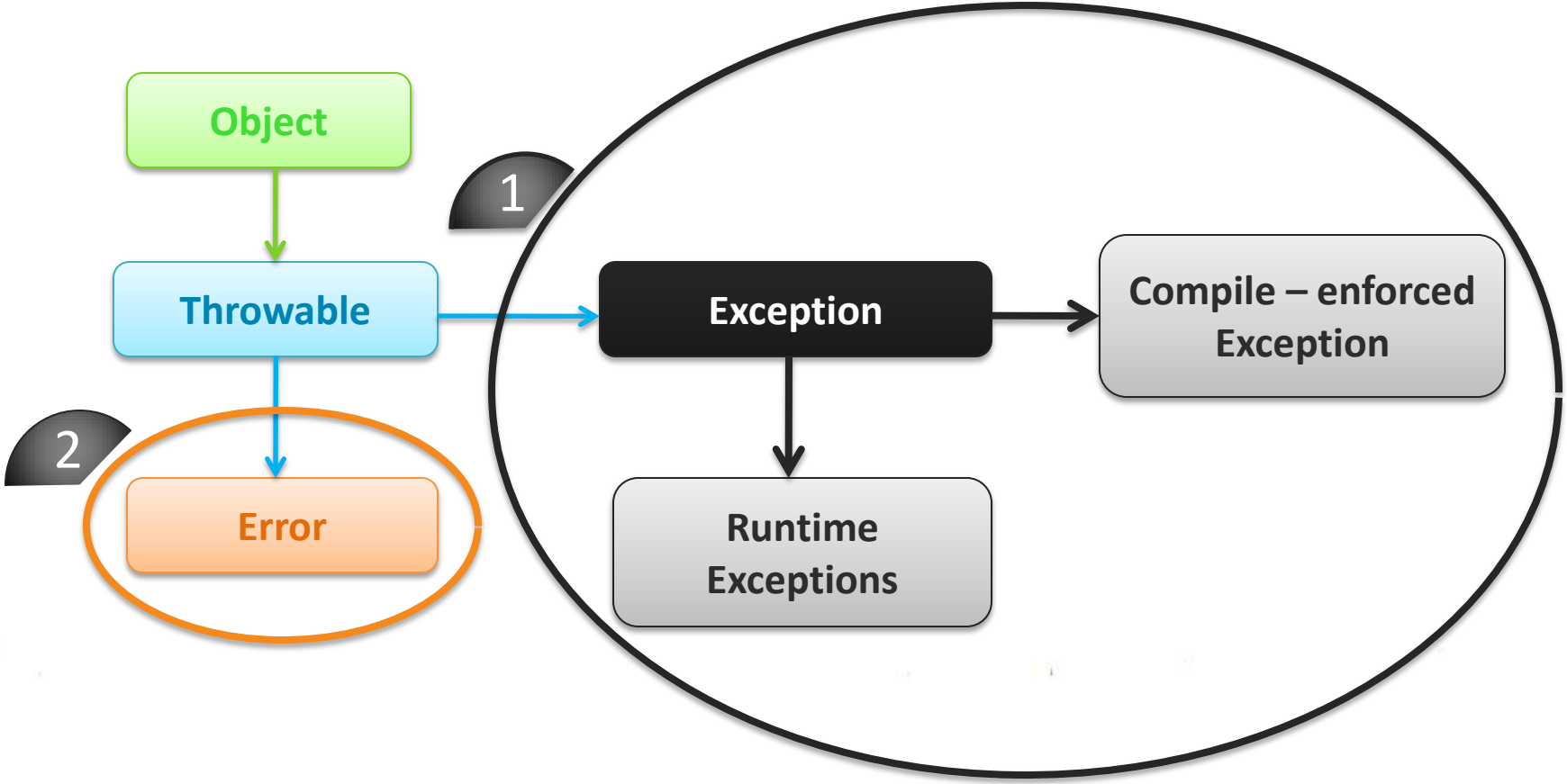
What is Exception Handling?

edureka!

- **Exception** is the one that stops the execution of the program unexpectedly.
- The process of handling these exceptions is called **Exception Handling**.

Exception Classes

edureka!



Types of Exception

edureka!

- **Run-time Exceptions.**
- **Compile Enforced Exception**

Run-Time Exceptions

edureka!

- Are also called as **Unchecked Exception.**
- These exceptions are **handled** at run-time i.e **by JVM** after they have occurred by using try and catch block.
- **Eg:** ArrayIndexOutOfBoundsException,

  ArithmeticException

  NullPointerException

Complier-enforced Exceptions

edureka!

- Are also called as **Checked Exceptions**.
- These exceptions are **handled by java complier** before they occur by using throws keyword.
- **Eg:** IOException,

  FileNotFoundException

Exception Handling Mechanism

edureka!

Exception can be handled in 3 ways:

- **try** block
- **Catch** block
- **Finally** block

Try and Catch block

edureka!

```
try
 {
 //code where you think exception would occur
 }

 catch(Exception_Class reference)
 {
 //Catch the exception and displays that exception
 }
```

Try – Catch example

edureka!

```
public class Try_Catch {
public static void main(String[] args)  {
                    int y=0;
                    try {
                              System.out.println(5/y);
                        }
                    catch(Exception e) {
                              System.out.println("Divide By Zero Exception");
                        }
    }
}
```

Multiple Catches

edureka!

- When there is a chance of getting different types of exceptions we use multiple catch block for a try block.

```
try
 {
 //statements
 }

 catch(Exception_Class reference)
 {
 //statements for one type of exception
 }

catch(Exception_Class reference)
 {
 //statements for other type of exception
 }
```

Multiple- Catch Example

edureka!

```
package com.edureka.exception.multiplecatch;
class Multiple_Catch {

    int n;
    int array[]=new int[3];
    Multiple_Catch(int n)
    {
        try{
            if(n==0)
                System.out.println(5/n);
            else{
                array[3]=n;
                System.out.println(array);
            }
        }
catch(ArrayIndexOutOfBoundsException
        arrayexception)
{
        System.out.println(arrayexception);
}

catch(ArithmeticException divideexception)
{
        System.out.println(divideexception);
}
    }
}
```

Multiple- Catch Example

edureka!

```
package com.edureka.exception.multiplecatch;
class Main {

    public static void main(String[] args)
    {

    Multiple_Catch  multiplecatch1= new Multiple_Catch(0);
    Multiple_Catch  multiplecatch2= new Multiple_Catch(5);

     }
}
```

What is throw keyword?

edureka!

- **throw** is a keyword which is used to call the sub class of an exception class.

- This keyword is also used to throw the exception occurred in try block to catch block.

```
try{
        throw new Exception_class("message");
    }
catch(Exception_class reference){
        //statements
    }
```

Example using throw keyword

edureka!

```
package com.edureka.exception.throwkeyword;
public class Student {

    Student(int studentid, String name){

        try{
            if(studentid==0)
                throw new Exception("id can not be zero");
            else
                System.out.println("The id of "+name+"
                                    is:"+studentid);
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

```
package com.edureka.exception.throwkeyword;

public class Main {

public static void main(String[] args) {

Student student1 = new Student(0,"STUDENT1");
Student student2 = new Student(1,"STUDENT2");

}

}
```

What is throws keyword?

edureka!

- **throws** is a keyword applied to methods for which an exception has raised during its execution.

```
returntype method_name throws Exception_Class
{
// statements
}
```

Example using throws keyword

edureka!

```
package com.edureka.throwskeyword;
public class GiveInput {

    void takeInput() throws IOException
     {
            BufferedReader reader=new
                        BufferedReader(new
                        InputStreamReader(System.in));

            System.out.println("Enter your name");
            String name=reader.readLine();

            System.out.println("Your name is: "+name);
     }
}
```

```
package com.edureka.throwskeyword;
public class Main {

public static void main(String[] args) throws
            IOException {

GiveInput input=new GiveInput();
input.takeInput();

}
}
```

Uses of finally keyword

edureka!

- When we want a set of statements to be executed even after an exception has occurred then we use finally block.
- finally

```
{
    //statements that needs to be executed after exception
}
```

User-defined Exceptions

edureka!

- **Across built-in exceptions user can also define his own exceptions.**
- It can be done by defining a class that **extends Exception class** and creating a constructor of the class (user-defined) with string argument to print that message when exception occurs.

Advantages of Exception

edureka!

- The program will still execute even if an exception arises i.e **finally block.**
- If you can't handle the exception JVM will handle the exception if we use **throws keyword**.
- We can differentiate the exceptions that have occurred.

Errors and Error Handling

edureka!

- **Design-time error:** These are the errors that occur while designing the programs.

  Eg: Syntax errors

These errors will be shown with a red mark in eclipse IDE so that you can easily find and correct it.

Errors and Error Handling

edureka!

- **Logical error:** These are the errors done by programmer. The programs with these errors will run but does not produce desired results.

  Eg: getting division of two numbers as output but expected is multiplication of numbers.

  These errors can be rectified by understanding the logic and checking whether it is works out correctly or not.

- Q& A..?

Thanks..!