# UNIT I - JAVA FUNDAMENTALS

- Java Data types
- Class – Object
- I / O Streams
- File Handling concepts
- Threads
- Applets
- Swing Framework
- Reflection

Presented by,
B.Vijayalakshmi
Computer Centre
MIT Campus
Anna University

# Miscellaneous Topics

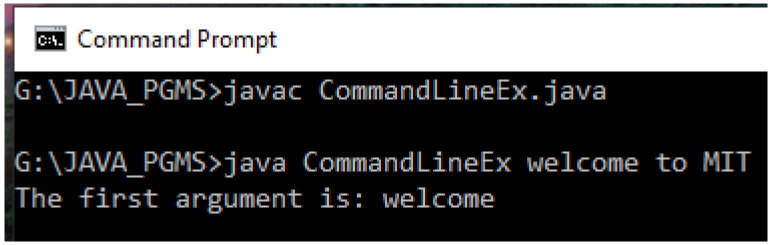
- ➢Command Line Arguments
- ➢Java Wrapper Class
- ➢Garbage Collection in Java
- ➢Exception Handling in Java

## Command Line Arguments

# Java Command line argument

- The command line argument is the **argument that passed to a program during runtime.**
- It is the way to **pass argument to the main method** in Java
- The arguments passed from the console can be received in the java program and it can be used as an input.
- These arguments store into the String type **args** parameter which is main method parameter.

```
class CommandLineEx
{
 public static void main(String args[])
 {
   System.out.println("The first argument is: "+args[0]);
 }
}
```



Command Prompt

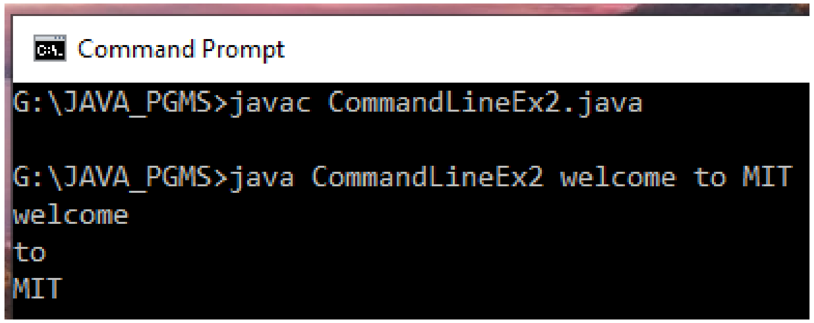
```
G:\JAVA_PGMS>javac CommandLineEx.java

G:\JAVA_PGMS>java CommandLineEx welcome to MIT
The first argument is: welcome
```

•In the above program, the main() method includes an array of strings named args as its parameter. The String array stores all the arguments passed through the command line.

**Note**: Arguments are always stored as strings and always separated by **white-space**.

## Command line argument example

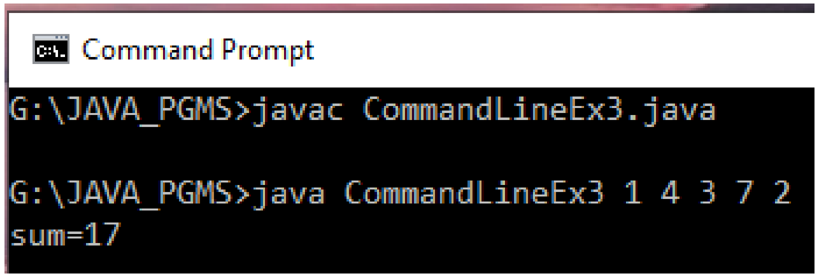
```
class CommandLineEx2
{
        public static void main(String args[])
        {
                for(String s:args)
                        System.out.println(s);
        }
}
```



```
Command Prompt
G:\JAVA_PGMS>javac CommandLineEx2.java

G:\JAVA_PGMS>java CommandLineEx2 welcome to MIT
welcome
to
MIT
```

- The main() method of every Java program only accepts string arguments. Hence it is not possible to pass numeric arguments through the command line.
- However, we can later convert string arguments into numeric values.

```
class CommandLineEx3
{
  public static void main(String args[])
  {
        int sum=0;
        for(String s:args)
                sum=sum+Integer.parseInt(s);
        System.out.println("sum="+sum);
  }
}
```



```
Command Prompt
G:\JAVA_PGMS>javac CommandLineEx3.java

G:\JAVA_PGMS>java CommandLineEx3 1 4 3 7 2
sum=17
```

Here, the parseInt() method of the Integer class converts the string argument into an integer. Similarly, we can use the parseDouble() and parseFloat() method to convert the string into double and float respectively.

## Java Wrapper Class

## Java Wrapper Class

- It is used for **converting primitive data type into object and object into a primitive data type**.
- For each primitive data type, a pre-defined class is present which is known as Wrapper class.

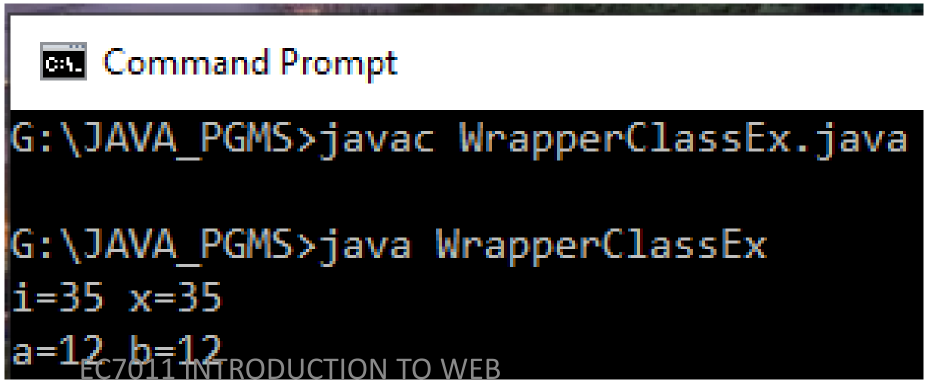
| Primitive Type | Wrapper Class |
|---|---|
| byte | Byte |
| boolean | Boolean |
| char | Character |
| double | Double |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |

# Autoboxing and Unboxing in Java

- **Autoboxing**
  - The automatic **conversion of primitive data type into its corresponding wrapper class** is known as autoboxing
  - Example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
- **Unboxing**
  - The **automatic conversion of wrapper type into its corresponding primitive type** is known as unboxing.
  - It is the reverse process of autoboxing.
- These conversions are done implicitly by the Java compiler during program execution.

## Autoboxing and Unboxing example

```
public class WrapperClassEx
{
        public static void main(String args[])
        {
                //Autoboxing
                int i=35;
                Integer x=i; //autoboxing -converting int into Integer
                System.out.println("i="+i+" x="+x);
                //unboxing
                Integer a=new Integer(12);
                int b=a;//unboxing Integer to int
                System.out.println("a="+a+" b="+b);
        }
}
```


Command Prompt

```
G:\JAVA_PGMS>javac WrapperClassEx.java

G:\JAVA_PGMS>java WrapperClassEx
i=35 x=35
a=12 b=12
```

# Use of Wrapper classes in Java

- Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc.
- Let us see the different scenarios, where we need to use the wrapper classes.
  - **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
  - **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
  - **Synchronization:** Java synchronization works with objects in Multithreading.
  - **java.util package:** The java.util package provides the utility classes to deal with objects.
  - **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.
- **Note**: Primitive types are more efficient than corresponding objects. Hence, when efficiency is the requirement, it is always recommended primitive types.

# Garbage Collection in Java

# Garbage Collection in Java

- In java, **garbage means unreferenced objects**.
- When a Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed.
- When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called **Garbage Collection**.
- To do so, we were using,
  - free() function in C language
  - delete() in C++
- But, **in java it is performed automatically**. So, java provides better memory management.

## How can an object be unreferenced?

- There are many ways:
  - By setting null to object reference

```
Student s1=new Student();
s1=null;
```

  - By assigning a reference to another

```
Student s1=new Student();
Student s2=new Student();
s1=s2;  //Now the first object referred by s1 is available for garbage collection
```

  - By anonymous object

```
new Student();
```

## Advantages of Garbage Collection

- Programmer doesn't need to worry about dereferencing an object.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.
- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory and decreases the chances for memory leak..

## Can the Garbage Collection be forced explicitly ?

•No, the Garbage Collection can not be forced explicitly. We may request JVM for **garbage collection** by calling **System.gc()** method.

•But This does not guarantee that JVM will perform the garbage collection.

## gc() method

- It is used to **invoke the garbage collector to perform cleanup processing.**
- It is **found in System and Runtime classes**.

  **public static void** gc(){}

- **Note:** Garbage collection is performed by a daemon thread called Garbage Collector(GC). **This thread calls the finalize() method** before object is garbage collected.

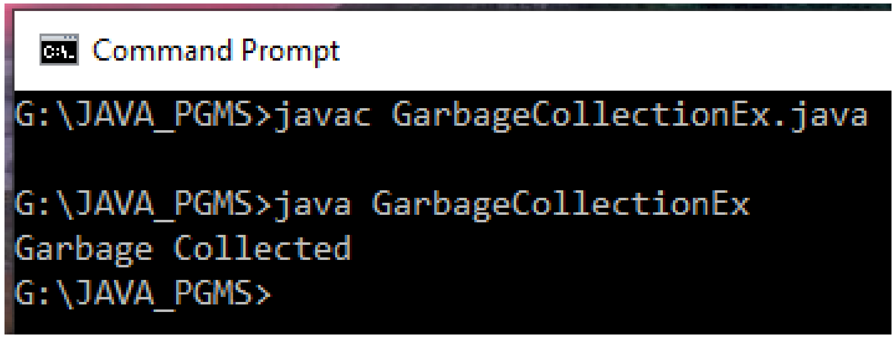
## finalize() method

- The **finalize()** method is **called by garbage collection thread before collecting object**.
- Its the last chance for any object to perform cleanup utility.

**protected void finalize() { //finalize-code }**

## Explicit Garbage Collection Example

```
public class GarbageCollectionEx
{
   public static void main(String[] args)
   {
      GarbageCollectionEx obj = new GarbageCollectionEx();
      obj=null;
      System.gc();
   }
   public void finalize()
   {
      System.out.println("Garbage Collected");
   }
}
```



```
G:\JAVA_PGMS>javac GarbageCollectionEx.java

G:\JAVA_PGMS>java GarbageCollectionEx
Garbage Collected
G:\JAVA_PGMS>
```

## Exception Handling in Java

# Exception Handling in Java

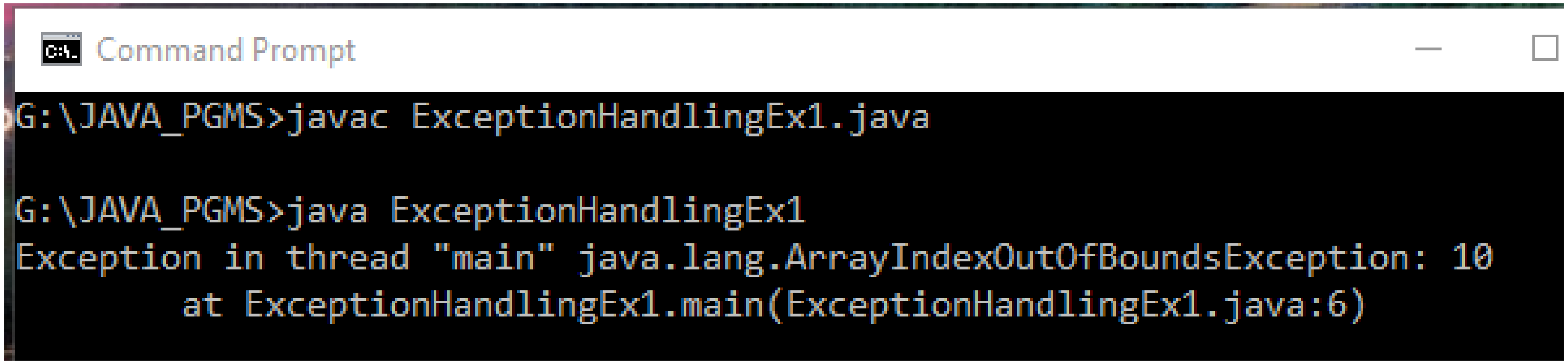
- **Exception:**
  - It is an **abnormal condition**, which **occurs during the execution** of a program.
  - **Example:**
    - Division by zero
    - Opening a file that does not exist
  - Java **Exception is an object** which describes an error condition that has materialized in the program.
  - Java handles exceptions using five keywords,
    - try
    - catch
    - finally
    - throws
    - throw

## Default Exception Handling

- When a program throws an Exception, it comes to stop
- This Exception should be caught by an Exception handler and dealt with immediately
- In case of provision of an **exception handler**, the **handling is done by the handler**
- Suppose there are **no handlers, the java runtime system** provides default handlers
- This **displays a string** which **describes the exception and the point at which the exception occurred** after which **the program terminates**

## Default Exception Handling Example

```
public class ExceptionHandlingEx1
{
  public static void main(String[] args)
  {
    int i[]={2};
    i[10]=5;
  }
}
```



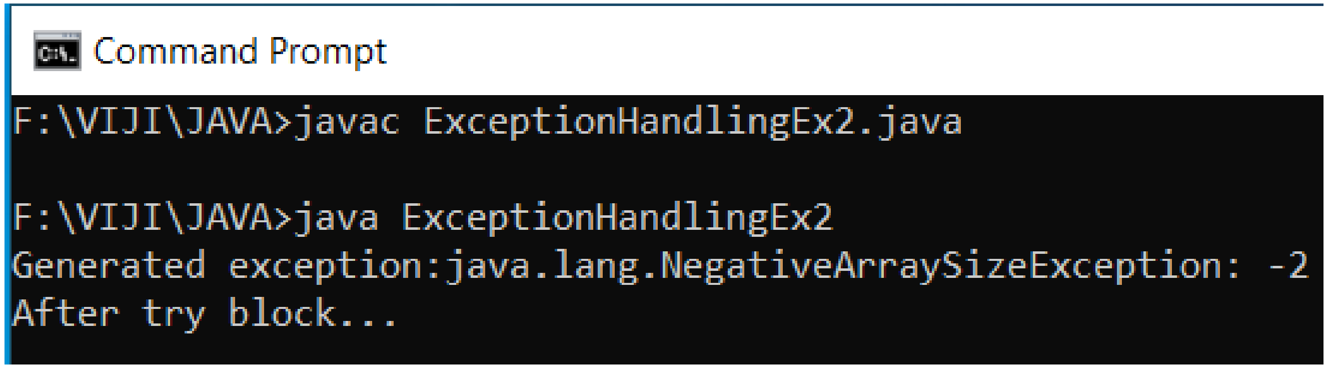
```
G:\JAVA_PGMS>javac ExceptionHandlingEx1.java

G:\JAVA_PGMS>java ExceptionHandlingEx1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at ExceptionHandlingEx1.main(ExceptionHandlingEx1.java:6)
```

## try and catch

- Whenever there is **a possibility of an exception** being generated in a program**, it is better to handle it explicitly**
- This can be done using the **try and catch**
- It fixes the error and prevents the program from terminating abruptly
- The **code sequence which is to be guarded**, should be placed **inside the try block**.
- **catch clause should be immediately follow the try block**
- **catch clause can have statements** explaining the cause of the exception
- **It is possible to have multiple catch statements for one try block**

## try and catch Example

```
class ExceptionHandlingEx2
{
        public static void main(String args[])
        {
                try
                {
                        int a[]=new int[-2];
                        System.out.println("First element:"+a[0]);
                }
                catch(NegativeArraySizeException n)
                {
                        System.out.println("Generated exception:"+n);
                }
                System.out.println("After try block...");
        }
}
```



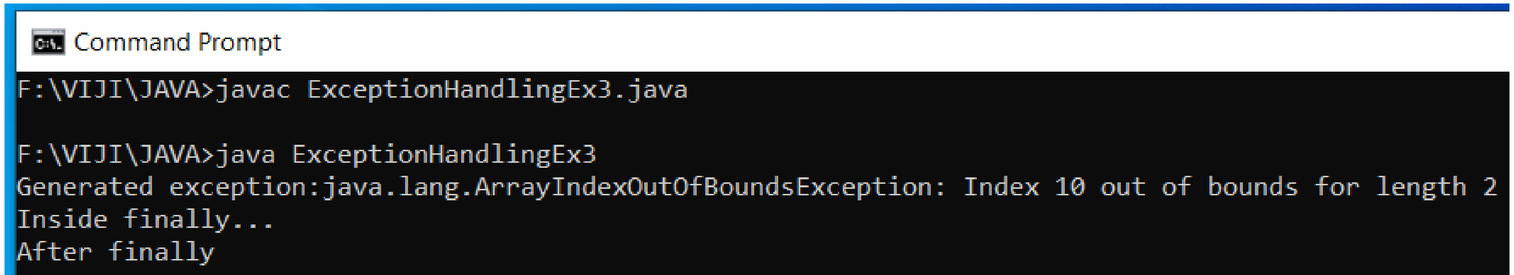
```
F:\VIJI\JAVA>javac ExceptionHandlingEx2.java

F:\VIJI\JAVA>java ExceptionHandlingEx2
Generated exception:java.lang.NegativeArraySizeException: -2
After try block...
```

# try, catch, finally

- When an Exception is generated in a program, sometimes it may be necessary to perform certain activities before termination.
- The activities may be closing a file, deleting a graphics object created etc.
- In such cases, the try block, apart from the catch clause also has finally block within which the  activities mentioned above can be performed.
- This block is executed after the execution of statements within the try & catch block
- In case an Exception is thrown, the finally block will be executed even if no catch matches it.
- **try block requires either a finally clause or catch clause**

## try, catch, finally Example

```
class ExceptionHandlingEx3
{
	public static void main(String args[])
	{
		try
		{
			int a[]=new int[2];
			System.out.println("10th element:"+a[10]);
		}
		catch(Exception e)
		{
			System.out.println("Generated exception:"+e);
		}
		//System.out.println("After try block...");
		//ExceptionHandlingEx3.java:15: error: 'finally' without 'try'
		finally
		{
			System.out.println("Inside finally...");
		}
		System.out.println("After finally");
	}
}
```



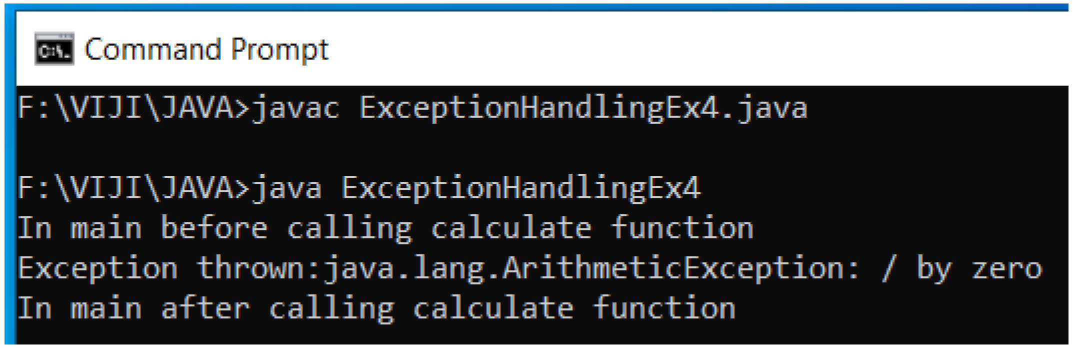
```
F:\VIJI\JAVA>javac ExceptionHandlingEx3.java

F:\VIJI\JAVA>java ExceptionHandlingEx3
Generated exception:java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 2
Inside finally...
After finally
```

# throws

- Handling all the Exceptions using the try and catch block could be cumbersome and will hinder the coders throughput.
- So java provides an option, wherein whenever we are using a risky piece of code in the method definition we declare it throws an Exception without implementing try catch
- The **throws keyword is used in method declaration** in order to explicitly specify the exceptions that a particular method might throw
- When a method declaration has one or more exceptions defined using throws clause then the method-call must handle all the defined exceptions
- If a method is using throws clause along with few exceptions then this implicitly tells other methods that – "if you call me you must handle these exceptions that i throw"

## Throws example

```
class ExceptionHandlingEx4
{
 void calculate() throws ArithmeticException
 {
  int i=0;
  int n=25/i;
  System.out.println("n="+n);
  System.out.println("After arithmetic calculation");
  throw new ArithmeticException();
 }
 public static void main(String args[])
 {
     ExceptionHandlingEx4 obj=new ExceptionHandlingEx4();
     System.out.println("In main before calling calculate function");
     try
     {
          obj.calculate();
     }
    catch(Exception e)
    {
          System.out.println("Exception thrown:"+e);
    }
    System.out.println("In main after calling calculate function");
 }
}
```


Command Prompt

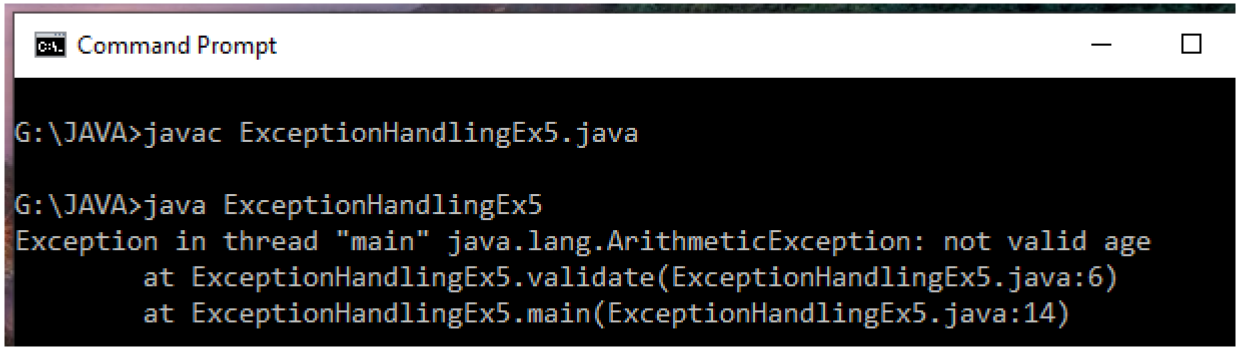
```
F:\VIJI\JAVA>javac ExceptionHandlingEx4.java

F:\VIJI\JAVA>java ExceptionHandlingEx4
In main before calling calculate function
Exception thrown:java.lang.ArithmeticException: / by zero
In main after calling calculate function
```

# throw

- The throw keyword is used to explicitly throw a single exception.
- We can throw either checked or unchecked exception in java by throw keyword.
  - **Checked Exceptions:** They are **checked at compile-time**. For example, IOException , InterruptedException , etc.
  - **Unchecked Exceptions:** They are **not checked at compile-time** but at run-time. For example: ArithmeticException , NullPointerException , ArrayIndexOutOfBoundsException , exceptions under Error class, etc.
- The throw keyword is **mainly used to throw custom exception**.
- The throw keyword is used to **throw an exception explicitly**.
- **Only object of Throwable class or its sub classes can be thrown**.
- Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.
- **Syntax :** throw throwableObject;

## Throw example

```
public class ExceptionHandlingEx5
{
        void validate(int age)
        {
                if(age<18)
                        throw new ArithmeticException("not valid age");
                else
                        System.out.println("welcome to vote");
        }

        public static void main(String args[])
        {
                ExceptionHandlingEx5 obj=new ExceptionHandlingEx5();
                obj.validate(13);
                System.out.println("After validate function call...");
        }
}
```



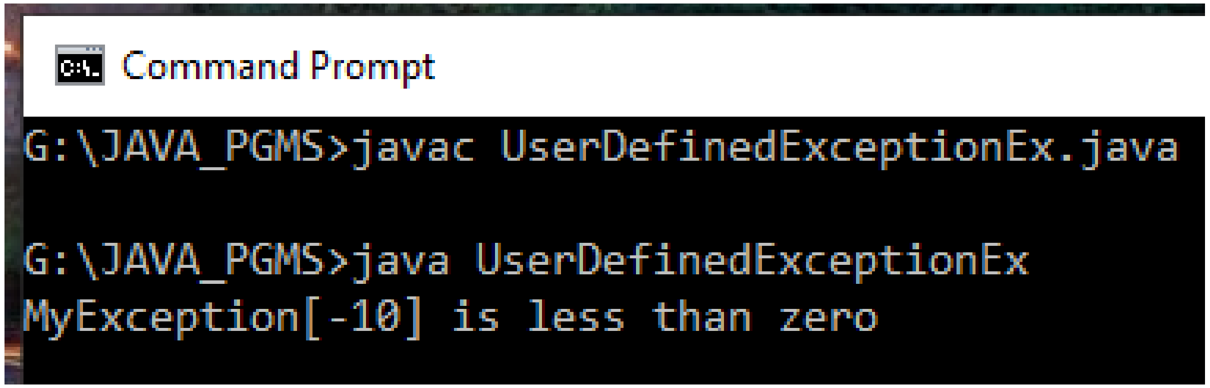
```
G:\JAVA>javac ExceptionHandlingEx5.java

G:\JAVA>java ExceptionHandlingEx5
Exception in thread "main" java.lang.ArithmeticException: not valid age
        at ExceptionHandlingEx5.validate(ExceptionHandlingEx5.java:6)
        at ExceptionHandlingEx5.main(ExceptionHandlingEx5.java:14)
```

## user-defined or custom exceptions

- In java we can create our own exception class and throw that exception using throw keyword.
- These exceptions are known as **user-defined** or **custom** exceptions.
- we can create our own exception simply by extending java **Exception** class.
- **Constructors in Exception class**:
  - public Exception( ) →Constructs a new exception with null as its detail message.
  - public Exception(String message)→Constructs a new exception with the specified detail message.
- We can define a constructor for our Exception (not compulsory) and we can override the toString() function to display our customized message on catch

## user-defined  exceptions Example1

```
class LessThanZeroException extends Exception
{
  private int ex;
  LessThanZeroException(int a)
  {
               ex = a;
  }
  public String toString()
  {
               return "MyException[" + ex +"] is less than zero";
  }
}
class UserDefinedExceptionEx
{
  static void sum(int a,int b) throws LessThanZeroException
  {
     if(a<0)
     {
      throw new LessThanZeroException(a); //calling constructor of user-defined exception class
      }
      else
       {
               System.out.println(a+b);
      }
 }
public static void main(String[] args)
{
   try
    {
            sum(-10, 10);
    }
    catch(LessThanZeroException e)
   {
            System.out.println(e); //it calls the toString() method of user-defined Exception
    }
  }
}
```

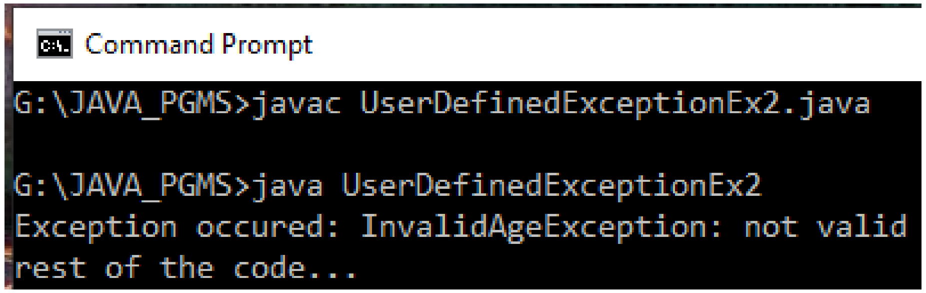


Command Prompt

```
G:\JAVA_PGMS>javac UserDefinedExceptionEx.java

G:\JAVA_PGMS>java UserDefinedExceptionEx
MyException[-10] is less than zero
```

## user-defined exceptions Example2

```
class InvalidAgeException extends Exception
{
 InvalidAgeException(String s)
 {
  super(s);  //calling parameterized constructor of Exception class
 }
}
class UserDefinedExceptionEx2
{
    static void validate(int age)throws InvalidAgeException
    {
     if(age<18)
      throw new InvalidAgeException("not valid");
     else
      System.out.println("welcome to vote");
   }
    public static void main(String args[])
    {
      try
     {
      validate(13);
      }catch(Exception m){System.out.println("Exception occured: "+m);}
      System.out.println("rest of the code...");
  }
}
```



```
G:\JAVA_PGMS>javac UserDefinedExceptionEx2.java

G:\JAVA_PGMS>java UserDefinedExceptionEx2
Exception occured: InvalidAgeException: not valid
rest of the code...
```