**Exception Handling**

**-----------------------**

## **Introduction**

**Exception:** An unwanted unexpected event that disturbs normal flow of the program is called exception.

***Example:*** SleepingException  
 TyrePunchuredException  
 FileNotFoundException ...etc

* It is highly recommended to handle exceptions. The main objective of exception handling is graceful (normal) termination of the program.

**What is the meaning of exception handling?**

Exception handling doesn't mean repairing an exception. We have to define alternative way to continue rest of the program normally this way of "defining alternative is nothing but exception handling".

***Example:*** Suppose our programming requirement is to read data from remote file locating at London at runtime if London file is not available our program should not be terminated abnormally.

We have to provide a local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

Example:

Try

{

read data from london file

}

catch(FileNotFoundException e)

{

use local file and continue rest of the program normally

}

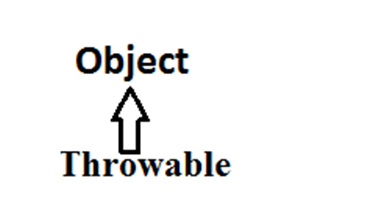
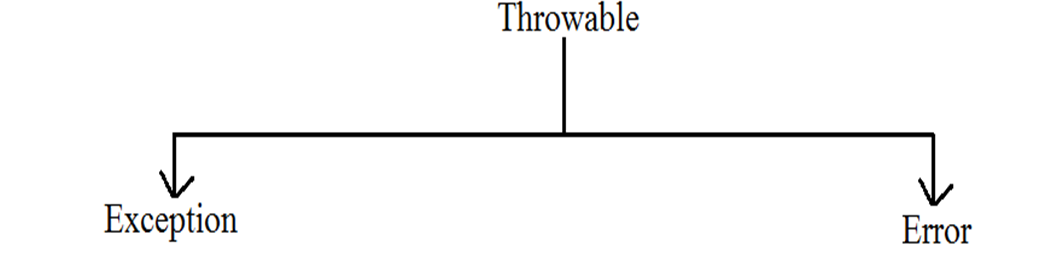
}

.

.

.

### **Exception hierarchy:**

**** Throwable acts as a root for exception hierarchy.  
 Throwable class contains the following two child classes.  


**Exception:** Most of the cases exceptions are caused by our program and these are recoverable.  
 **Ex :** If FileNotFoundException occurs we can use local file and we can continue rest of the program execution normally.  
  
 **Error:** Most of the cases errors are not caused by our program these are due to lack of system resources and these are non recoverable.  
 **Ex :** If OutOfMemoryError occurs being a programmer we can't do anything the program will be terminated abnormally.  
 System Admin or Server Admin is responsible to raise/increase heap memory.

## **Checked Vs Unchecked Exceptions:**

* The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.

1. HallTicketMissingException

2. PenNotWorkingException

3. FileNotFoundException

* The exceptions which are not checked by the compiler are called unchecked exceptions.

1. BombBlaustException

2. ArithmeticException

3. NullPointerException

**Note:** RuntimeException and its child classes, Error and its child classes are unchecked and all the remaining are considered as checked exceptions.  
 **Note:** Whether exception is checked or unchecked compulsory it should occur at runtime only there is no chance of occurring any exception at compile time.

### **Partially checked Vs fully checked :**

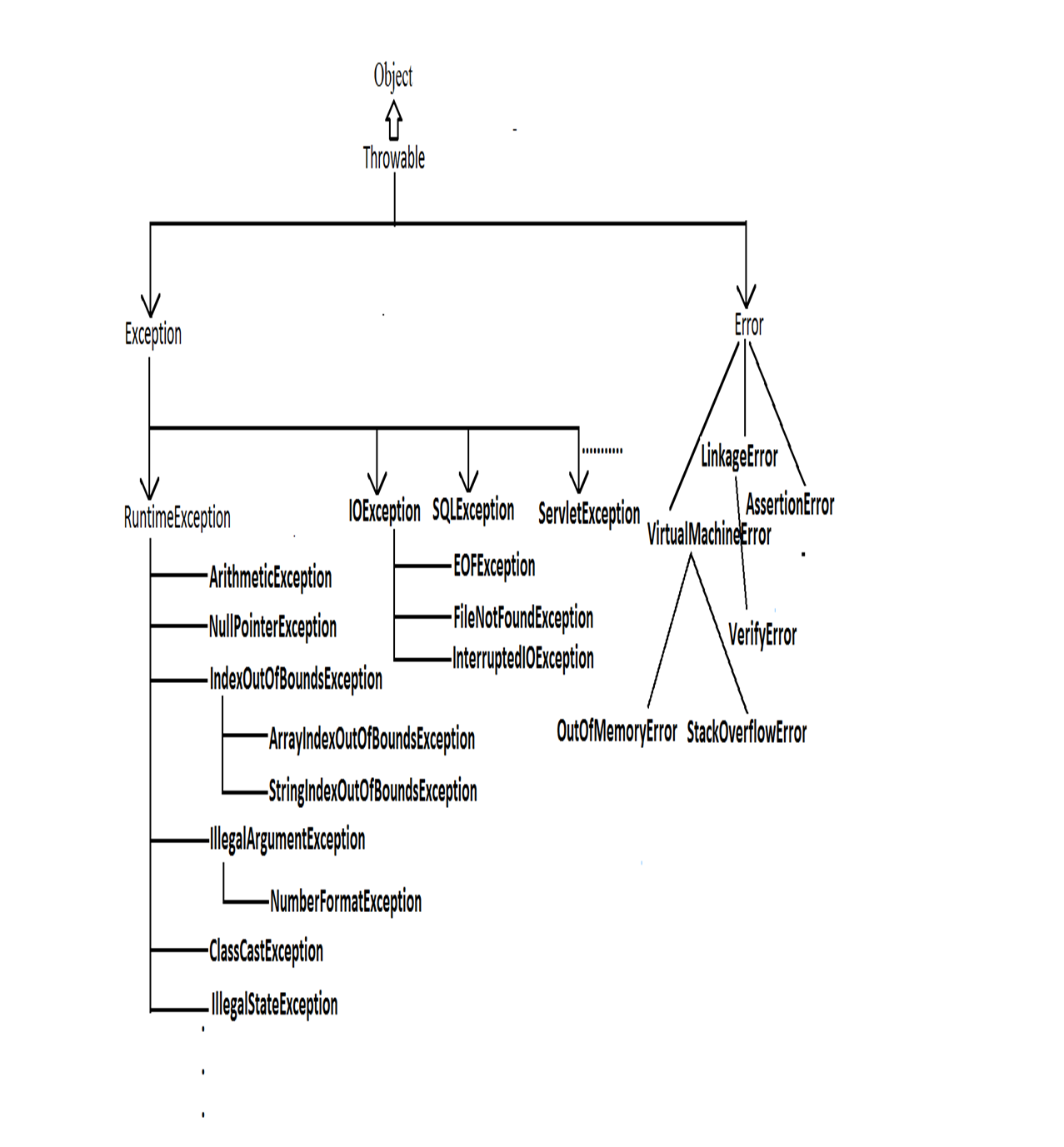
A checked exception is said to be fully checked if and only if all its child classes are also checked.  
 **Example:** 1) IOException  
 2) InterruptedException

A checked exception is said to be partially checked if and only if some of its child classes are unchecked.

**Example:** Exception The only partially checked exceptions available in java are:

1. Throwable.

2. Exception.

*Diagram:  
*

### **Customized exception handling by try catch:**

* It is highly recommended to handle exceptions.
* In our program the code which may cause an exception is called risky code, we have to place risky code inside try block and the corresponding handling code inside catch block.

Example:

Try

{

risky code

}

catch(Exception e)

{

handling code

}

| **Without try catch** | **With try catch** |
| --- | --- |
| class Test  {  public static void main(String[] args){  System.out.println("statement1");  System.out.println(10/0);  System.out.println("statement3");  }  }  output:  statement1  RE:AE:/by zero  at Test.main()    Abnormal termination. | class Test{  public static void main(String[] args){  System.out.println("statement1");  try{  System.out.println(10/0);  }  catch(ArithmeticException e){  System.out.println(10/2);  }  System.out.println("statement3");  }}  Output:  statement1  5  statement3    Normal termination. |

### **Control flow in try catch:**

try{

statement1;

statement2;

statement3;

}

catch(X e) {

statement4;

}

statement5;

* **Case 1:** There is no exception.  
   1, 2, 3, 5 normal termination.
* **Case 2:** if an exception raised at statement 2 and corresponding catch block matched 1, 4, 5 normal termination.
* **Case 3**: if an exception raised at statement 2 but the corresponding catch block not matched , 1 followed by abnormal termination.
* **Case 4:** if an exception raised at statement 4 or statement 5 then it's always abnormal termination of the program.

***Note:***

1. Within the try block if anywhere an exception raised then rest of the try block won't be executed even though we handled that exception. Hence we have to place/take only risk code inside try and length of the try block should be as less as possible.

2. If any statement which raises an exception and it is not part of any try block then it is always abnormal termination of the program.

3. There may be a chance of raising an exception inside catch and finally blocks also in addition to try block.

### **Various methods to print exception information:**

Throwable class defines the following methods to print exception information to the console.

| **printStackTrace():** | This method prints exception information in the following format.  **Name of the exception: description of exception  Stack trace** |
| --- | --- |
| **toString():** | This method prints exception information in the following format.  **Name of the exception: description of exception** |
| **getMessage():** | This method returns only description of the exception.  **Description.** |

**Example:**

**Note:** Default exception handler internally uses printStackTrace() method to print exception information to the console.

## Try with multiple catch blocks:

The way of handling an exception is varied from exception to exception hence for every exception raise a separate catch block is required that is try with multiple catch blocks is possible and recommended to use.

**Example:**

| try  {  .  .  .  .  }  catch(Exception e)  {  default handler  } | try  {  .  .  .  .  catch(FileNotFoundException e)  {  use local file  }  catch(ArithmeticException e)  {  perform these Arithmetic operations  }  catch(SQLException e)  {  don't use oracle db, use mysql db  }  catch(Exception e)  {  default handler  } |
| --- | --- |
| This approach is not recommended  because for any type of Exception  we are using the same catch block. | This approach is highly recommended  because for any exception raise  we are defining a separate catch block. |

* If try with multiple catch blocks presents then order of catch blocks is very important it should be from child to parent by mistake if we are taking from parent to child then we will get Compile time error saying "exception xxx has already been caught".

**Example:**

| class Test  {  public static void main(String[] args)  {  try  {  System.out.println(10/0);  }  catch(Exception e)  {  e.printStackTrace();  }  catch(ArithmeticException e)  {  e.printStackTrace();  }}}  Output:  Compile time error.  Test.java:13: exception  java.lang.ArithmeticException has already  been caught catch(ArithmeticException e) | class Test  {  public static void main(String[] args)  {  try  {  System.out.println(10/0);  }  catch(ArithmeticException e)  {  e.printStackTrace();  }  catch(Exception e)  {  e.printStackTrace();  }}}  Output:  Compile successfully. |
| --- | --- |

### **Finally block:**

* It is never recommended to take clean up code inside try block because there is no guarantee for the execution of every statement inside a try.
* It is never recommended to place clean up code inside catch block because if there is no exception then catch block won't be executed.
* We require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled such type of place is nothing but finally block.
* Hence the main objective of finally block is to maintain cleanup code.

Example:

Try

{

risky code

}

catch(x e)

{

handling code

}

finally

{

cleanup code

}

The speciality of finally block is it will be executed always irrespective of whether the exception raised or not raised and whether handled or not handled.

Example 1:

class Test

{

public static void main(String[] args)

{

try

{

System.out.println("try block executed");

}

catch(ArithmeticException e)

{

System.out.println("catch block executed");

}

finally

{

System.out.println("finally block executed");

}}}

Output:

Try block executed

Finally block executed

Example 2:

class Test

{

public static void main(String[] args)

{

try

{

System.out.println("try block executed");

System.out.println(10/0);

}

catch(ArithmeticException e)

{

System.out.println("catch block executed");

}

finally

{

System.out.println("finally block executed");

}}}

Output:

Try block executed

Catch block executed

Finally block executed

Example 3:

class Test

{

public static void main(String[] args)

{

try

{

System.out.println("try block executed");

System.out.println(10/0);

}

catch(NullPointerException e)

{

System.out.println("catch block executed");

}

finally

{

System.out.println("finally block executed");

}}}

Output:

Try block executed

Finally block executed

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Test.main(Test.java:8)

### **Difference between final, finally, and finalize:**

**Final:**

* Final is the modifier applicable for class, methods and variables.
* If a class declared as the final then child class creation is not possible.
* If a method declared as the final then overriding of that method is not possible.
* If a variable declared as the final then reassignment is not possible.

**Finally:**

* It is the block always associated with try catch to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled.

**Finalize:**

* It is a method which should be called by garbage collector always just before destroying an object to perform cleanup activities.

**Note:**

To maintain clean up code faunally block is recommended over finalize() method because we can't expect exact behavior of GC.

### **Throw statement:**

Sometimes we can create Exception object explicitly and we can hand over to the JVM manually by using throw keyword.

*Example:* The result of following 2 programs is exactly same.

| class Test  {  public static void main(String[] args){  System.out.println(10/0);  }}  In this case creation of ArithmeticException object and handover to the jvm will be performed automatically by the main() method. | class Test  {  public static void main(String[] args){  throw new ArithmeticException("/ by zero");  }}  In this case we are creating exception object explicitly and handover to the JVM manually. |
| --- | --- |

*Note:* In general we can use throw keyword for customized exceptions but not for predefined exceptions.

**Throws statement:**

In our program if there is any chance of raising checked exception compulsory we should handle either by try catch or by throws keyword otherwise the code won't compile.

Example:

import java.io.\*;

class Test3

{

public static void main(String[] args){

PrinterWriter out=new PrintWriter("abc.txt");

out.println("hello");

}

}

CE :

Unreported exception java.io.FileNotFoundException;

must be caught or declared to be thrown.

Example:

class Test3

{

public static void main(String[] args){

Thread.sleep(5000);

}

}

Unreported exception java.lang.InterruptedException;

must be caught or declared to be thrown.

We can handle this compile time error by using the following 2 ways.  
 Example:

| **By using try catch** | **By using throws keyword** |
| --- | --- |
| class Test3  {  public static void main(String[] args){  try{  Thread.sleep(5000);  }  catch(InterruptedException e){}  }  }  Output:  Compile and running successfully | We can use throws keyword to delicate the responsibility of exception handling to the caller method. Then caller method is responsible to handle that exception.  class Test3  {  public static void main(String[] args)throws  InterruptedException{  Thread.sleep(5000);  }  }  Output:  Compile and running successfully |

**Note :**

* Hence the main objective of "throws" keyword is to delicate the responsibility of exception handling to the caller method.
* "throws" keyword required only checked exceptions. Usage of throws for unchecked exception there is no use.
* "throws" keyword required only to convenes complier. Usage of throws keyword doesn't prevent abnormal termination of the program.  
   Hence recommended to use try-catch over throws keyword.

Example:

class Test

{

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

doStuff();

}

public static void doStuff()throws InterruptedException{

doMoreStuff();

}

public static void doMoreStuff()throws InterruptedException{

Thread.sleep(5000);

}

}

Output:

Compile and running successfully.

In the above program if we are removing at least one throws keyword then the program won't compile.

**Exception handling keywords summary:**

1. **try:** To maintain risky code.

2. **catch:** To maintain handling code.

3. **finally:** To maintain cleanup code.

4. **throw:** To handover our created exception object to the JVM manually.

5. **throws:** To delegate responsibility of exception handling to the caller method.

**Customized Exceptions (User defined Exceptions):**

Sometimes we can create our own exception to meet our programming requirements. Such type of exceptions are called customized exceptions (user defined exceptions).

***Example:***

1. InSufficientFundsException

2. TooYoungException

3. TooOldException

### **Steps to Create Customized Exceptions:**

1. **Create a Custom Exception Class:** Define a new class that extends either Exception (checked exception) or RuntimeException (unchecked exception) depending on whether you want your exception to be checked or unchecked.

2. **Throwing Custom Exceptions:** You can throw your custom exceptions using the throw keyword whenever a particular condition or error occurs in your code.

**3.** **Handling Custom Exceptions:** When invoking methods that throw your custom exceptions, you need to handle or propagate them accordingly.

**Customized Checked Example-1**

**------------------------------**

**package** com.codegnan.exceptionhandling;

//Custom exception for too young

**class** TooYoungException **extends** Exception {

**public** TooYoungException(String message) {

**super**(message);

}

}

//Custom exception for too old

**class** TooOldException **extends** Exception {

**public** TooOldException(String message) {

**super**(message);

}

}

//Class to check marriage eligibility based on age

**public** **class** MarriageEligibilityChecker {

// Method to check eligibility based on age

**public** **void** checkEligibility(**int** age) **throws** TooYoungException, TooOldException {

**if** (age < 18) {

**throw** **new** TooYoungException("Marriage cannot be approved: Age is below 18 years.");

} **else** **if** (age > 60) {

**throw** **new** TooOldException("Marriage cannot be approved: Age is above 60 years.");

} **else** {

System.***out***.println("Marriage approved! Details will be processed soon.");

// Additional logic for marriage approval can be added here

}

}

// Example usage

**public** **static** **void** main(String[] args) {

MarriageEligibilityChecker checker = **new** MarriageEligibilityChecker();

**try** {

// Trying to check eligibility for a person aged 20

checker.checkEligibility(20);

// Trying to check eligibility for a person aged 16 (will throw

// TooYoungException)

checker.checkEligibility(16);

// Trying to check eligibility for a person aged 65 (will throw TooOldException)

checker.checkEligibility(65);

} **catch** (TooYoungException e) {

System.***out***.println("Exception caught: " + e.getMessage());

// Additional handling logic can be added here

} **catch** (TooOldException e) {

System.***out***.println("Exception caught: " + e.getMessage());

// Additional handling logic can be added here

}

}

}

**Customized checked example-2**

**------------------------------**

**package** com.codegnan.exceptionhandling;

//Custom checked exception for insufficient funds

**class** InsufficientFundsException **extends** Exception {

// Constructor to initialize the exception with a message

InsufficientFundsException(String message) {

**super**(message);

}

}

//Class representing a bank account

**class** BankAccount {

**private** String accountNumber;

**private** **double** balance;

// Constructor

**public** BankAccount(String accountNumber, **double** initialBalance) {

**this**.accountNumber = accountNumber;

**this**.balance = initialBalance;

}

// Method to withdraw money from the account

**public** **void** withdraw(**double** amount) **throws** InsufficientFundsException {

**if** (amount > balance) {

**throw** **new** InsufficientFundsException("Withdrawal amount exceeds available balance.");

}

balance -= amount;

System.***out***.println("Withdrawal of $" + amount + " successfully processed.");

}

// Method to get current balance

**public** **double** getBalance() {

**return** balance;

}

}

//Main class to demonstrate usage of the customized checked exception

**public** **class** BankDemo {

**public** **static** **void** main(String[] args) {

// Create a bank account with initial balance

BankAccount account = **new** BankAccount("12345", 1000.0);

**try** {

// Attempt to withdraw an amount greater than the balance

account.withdraw(1500.0);

} **catch** (InsufficientFundsException e) {

// Handle insufficient funds exception

System.***err***.println("Error: " + e.getMessage());

System.***err***.println("Available balance is $" + account.getBalance());

}

}

}

**Customized uncheckedExample-1**

**----------------------------**

**package** com.codegnan.exceptionhandling;

//Custom unchecked exception for too young

**class** TooYoungException **extends** RuntimeException {

**public** TooYoungException(String message) {

**super**(message);

}

}

//Custom unchecked exception for too old

**class** TooOldException **extends** RuntimeException {

**public** TooOldException(String message) {

**super**(message);

}

}

//Class to check marriage eligibility based on age

**public** **class** MarriageEligibilityChecker {

// Method to check eligibility based on age

**public** **void** checkEligibility(**int** age) {

**if** (age < 18) {

**throw** **new** TooYoungException("Marriage cannot be approved: Age is below 18 years.");

} **else** **if** (age > 60) {

**throw** **new** TooOldException("Marriage cannot be approved: Age is above 60 years.");

} **else** {

System.***out***.println("Marriage approved! Details will be processed soon.");

// Additional logic based on eligibility

**if** (age >= 18 && age <= 25) {

System.***out***.println("You are considered a young adult.");

System.***out***.println("Additional counseling may be recommended.");

} **else** **if** (age > 60) {

System.***out***.println("Consideration for elder care may be advised.");

System.***out***.println("Legal and financial implications may need special attention.");

} **else** {

System.***out***.println("Standard procedures for marriage registration will be followed.");

}

// Simulate sending confirmation email or notification

sendConfirmationEmail();

// Additional processing logic can be added here, such as database updates, etc.

}

}

// Method to simulate sending confirmation email

**private** **void** sendConfirmationEmail() {

System.***out***.println("Confirmation email sent to the provided email address.");

}

// Example usage

**public** **static** **void** main(String[] args) {

MarriageEligibilityChecker checker = **new** MarriageEligibilityChecker();

**try** {

// Trying to check eligibility for a person aged 20

checker.checkEligibility(20);

// Trying to check eligibility for a person aged 16 (will throw

// TooYoungException)

checker.checkEligibility(16);

// Trying to check eligibility for a person aged 65 (will throw TooOldException)

checker.checkEligibility(65);

} **catch** (TooYoungException e) {

System.***out***.println("Exception caught: " + e.getMessage());

// Additional handling logic for too young can be added here

} **catch** (TooOldException e) {

System.***out***.println("Exception caught: " + e.getMessage());

// Additional handling logic for too old can be added here

}

}

}

**Customized unchecked example-2**

**--------------------------------**

**package** com.codegnan.exceptionhandling;

//Custom unchecked exception for underage voters

**class** UnderAgeException **extends** RuntimeException {

**public** UnderAgeException(String message) {

**super**(message);

}

}

//Custom unchecked exception for overage voters

**class** OverAgeException **extends** RuntimeException {

**public** OverAgeException(String message) {

**super**(message);

}

}

//Class to check voter eligibility based on age

**public** **class** VoterEligibilityChecker {

// Method to check eligibility based on age

**public** **void** checkEligibility(**int** age) {

**final** **int** VOTING\_AGE = 18;

**final** **int** MAXIMUM\_AGE = 120; // Just for example, can be adjusted as needed

**if** (age < VOTING\_AGE) {

**throw** **new** UnderAgeException("You are not eligible to vote: Age is below " + VOTING\_AGE + " years.");

} **else** **if** (age > MAXIMUM\_AGE) {

**throw** **new** OverAgeException("You are not eligible to vote: Age is above " + MAXIMUM\_AGE + " years.");

} **else** {

System.***out***.println("You are eligible to vote.");

// Additional logic based on eligibility

**if** (age >= 18 && age <= 25) {

System.***out***.println("You belong to the youth category.");

} **else** **if** (age > 60) {

System.***out***.println("You are a senior citizen.");

} **else** {

System.***out***.println("You are in the adult age group.");

}

// Additional processing logic can be added here, such as voter registration,

// etc.

}

}

// Example usage

**public** **static** **void** main(String[] args) {

VoterEligibilityChecker checker = **new** VoterEligibilityChecker();

**try** {

// Trying to check eligibility for a voter aged 20

checker.checkEligibility(20);

// Trying to check eligibility for a voter aged 16 (will throw

// UnderAgeException)

checker.checkEligibility(16);

// Trying to check eligibility for a voter aged 130 (will throw

// OverAgeException)

checker.checkEligibility(130);

} **catch** (UnderAgeException e) {

System.***out***.println("Exception caught: " + e.getMessage());

// Additional handling logic for underage voters can be added here

} **catch** (OverAgeException e) {

System.***out***.println("Exception caught: " + e.getMessage());

// Additional handling logic for overage voters can be added here

}

}

}