CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

Syllabus

MODULE III JAVA OBJECTS – 2	L	T	P	EL
	3	0	4	3
Inheritance and Polymorphism – Super classes and sub classes, overriding, object class and its methods, casting, instance of, Array list, Abstract Classes, Interfaces, Packages, Exception Handling				
SUGGESTED ACTIVITIES : • flipped classroom • Practical - implementation of Java programs – use Inheritance, polymorphism, abstract classes and interfaces, creating user defined exceptions • EL – dynamic binding, need for inheritance, polymorphism, abstract classes and interfaces				
SUGGESTED EVALUATION METHODS: • Assignment problems • Quizzes				

Google's self-driving car

Would you take a ride in a car that has no steering wheel, pedals, brakes or accelerator? How Google's self-driving car works:

A laser sensor scans 360 degrees around the vehicle for objects and helps determine its location.

A processor reads the data from the sensors and regulates vehicle behavior.

Radar helps determine speed by detecting and measuring the speed of vehicles ahead.

Orientation sensor located inside the car tracks the car's motion and balance.

Wheel hub sensor detects the number of rotations to help determine the car's location.

Articulating radars Scan a wide field to detect vehicles, pedestrians, objects

Cameras Provide a 360-degree view of surroundings

Lidar Bounces laser light off objects to detect shape and location

High-speed processors Crunch data from sensors to direct the car's movements

Vehicle location High-definition maps are created using lidar scans of roads

Short-range radar Detects objects, pedestrians, vehicles near the car

Long-range radar Detects and measures velocity of traffic down the road

Electric motor Battery power drives the wheels

Exception

Exception is an abnormal condition that arises in a code sequence at run time.

Exception is a run-time error.

Exception-Handling Fundamentals

- **A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.**
- **When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.**
  - That method may choose to handle the exception itself, or pass it on.
  - Either way, at some point, the exception is caught and processed.
- **Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.**
  - Exceptions thrown by Java relate to fundamental errors that or the constraints of the Java execution environment. violate the rules of the Java language
  - Manually generated exceptions are typically used to report some error condition to the caller of a method.

Exception-Handling Fundamentals

- **Java exception handling is managed via five keywords:**
  - **try, catch, throw, throws, and finally.**
- **Program statements to monitor for exceptions are contained within a try block.**
  - If an exception occurs within the try block, it is thrown.
  - The code can catch this exception (using catch) and handle it in some rational manner.
  - System generated exceptions are automatically thrown by the Java run-time system.
- **To manually throw an exception, use the keyword throw.**
- **Any exception that is thrown out of a method must be specified as such by a throws clause.**
- **Any code that absolutely must be executed after a try block completes is put in a finally block**

Why exception handling?

- In a language **To simplify programming and make applications more robust.**
- **What does robust mean?**

  without exception handling

  When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated

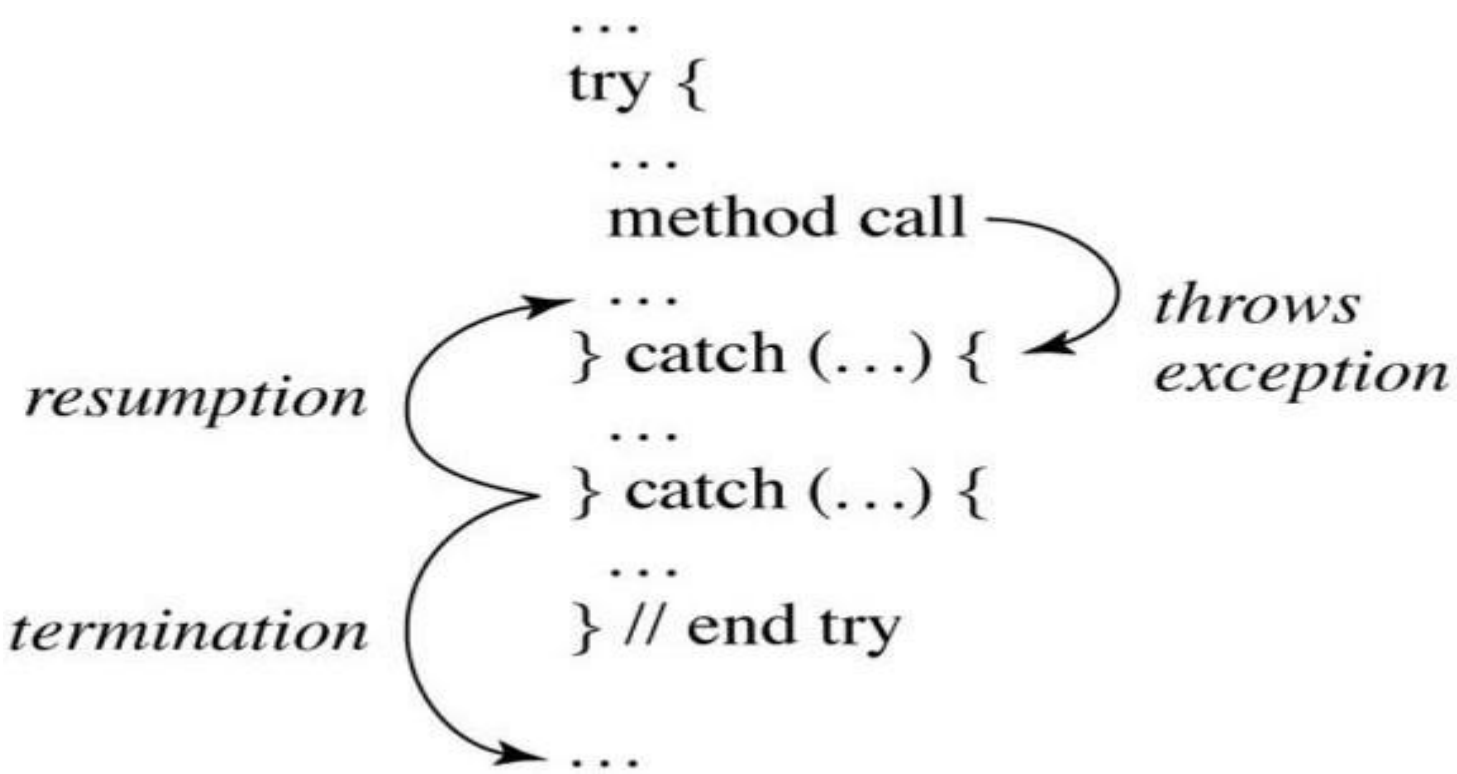
  In a language with exception handling

  Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing

Design Issues

How to create exception handlers and their scope?
How to bound an exception to a specific exception handler otherwise default exception handlers ?
How to Pass information about the exception be to the handler?
IS the excepting handling type is Continuation or Resumption?
Why to provide finalization?
How are user-defined exceptions are supported and handled?
Are there any predefined exceptions?
Can predefined exceptions be explicitly raised?
Are hardware-detectable errors can be handled?
Can exceptions be disabled, if at all?

Exception Handling type



Exception

- **An infrequent, abnormal situation in program logic at program execution.**
- **Not necessarily an error**

Reasons for exceptions:

- **hardware events**
  - **arithmetic overflow, parity errors**
- operating system events
  - **out of memory exception**
- **programming language properties**
  - **dynamic range checks**
- **application program properties**
  - **data structure overflow/underflow**

Basic Concepts

Many languages allow programs to trap input/output errors (including EOF)

- **An exception is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing**
- **The special processing that may be required after detection of an exception is called exception handling**
- **The exception handling code unit is called an exception handler**

Advantages of Built-in Exception Handling

- **Error detection code is tedious to write and it clutters the program**
- **Exception handling encourages programmers to consider many different possible error**

Exception Handling in Ada

The frame of an exception handler in Ada is either a subprogram body, a package body, a task, or a block

- Because exception handlers are usually local to the code in which the exception can be raised, they do not have parameters

Handler form:

```
exception
when exception_name {| exception_name} =>
statement_sequence
...
when ...
...
[when others =>
statement_sequence ]
```

Evaluation

- **Ada was the only widely used language with exception handling until it was added to C++**
- **Exceptions are added to C++ in 1990**
- **Design of exceptions is based on that of Ada, and ML**

C++ Exception Handlers

Exception Handlers Form:

```
try {
-- code that is expected to raise an exception
}
catch (formal parameter) {
-- handler code
}
...
catch (formal parameter) {
-- handler code
}
```

C++ Exception Handling example

```
#include <iostream.h>
int main () {
char A[10];
cin >> n;
try {
for (int i=0; i<n; i++){
if (i>9) throw "array index error";
A[i]=getchar();
}
}
catch (char* s)
{ cout << "Exception: " << s << endl; }
return 0;
}
```

```
int main()
{
int a,b;
cout<<“enter a,b values:”;
cin>>a>>b;
 try{
 if(b!=0)
 cout<<“result is:”<<(a/b);
 else
 throw b;
 }
catch(int e)
 {
 cout<<“divide by zero error occurred due
 to b= “ << e;
 }
}
```

The catch Function

- **Catch is the name of all handlers**
- **It is an overloaded name, so the formal parameter of each must be unique**
- **The formal parameter need not have a variable**
  - **It can be simply a type name to distinguish the handler it is in from others**
- **The formal parameter can be used to transfer information to the handler**
- **The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled**

Throwing Exceptions

Exceptions are all raised explicitly by the statement:

throw [expression];

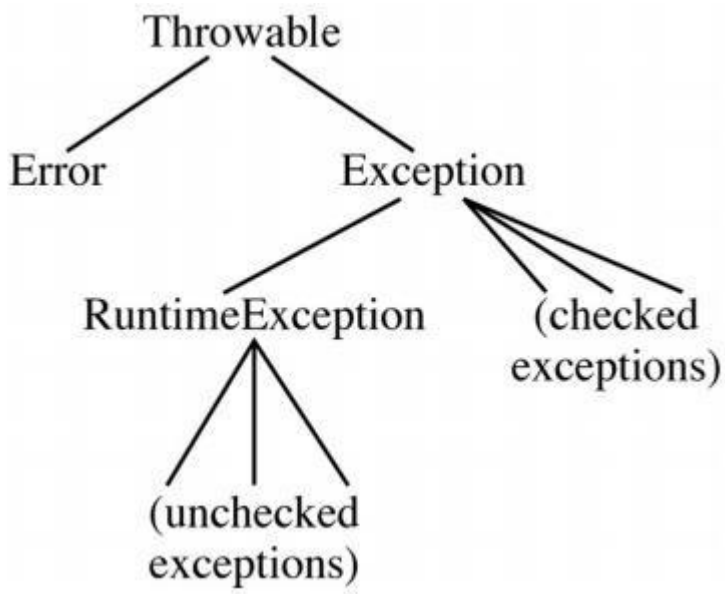
- The brackets are metasymbols
- A throw without an operand can only appear in a handler;
- when it appears, it simply re-raises the exception, which is then handled elsewhere
- The type of the expression disambiguates the intended handler

Exception Handling in Java

Based on that of C++, but more in line with OOP philosophy

- All exceptions are objects of classes that are descendants of the Throwable class

```
try { ... }
catch { ... }
// more handlers
finally { ... }
```



Classes of Exceptions

The Java library includes two subclasses of Throwable :

Error

- Thrown by the Java interpreter for events such as heap overflow
- Never handled by user programs

Exception

- User-defined exceptions are usually subclasses of this
- Has two predefined subclasses, IOException and RuntimeException

(e.g., ArrayIndexOutOfBoundsException and NullPointerException

RuntimeException sub classes :

ArithmeticException
ArrayIndexOutofBoundException
ArrayStoreException
ClassCasteException
IlegalArgumentException
IndexOutofBoundException
NegativeArraySizeException
NullPointerException
NumberFormatException
SecurityException
StringIndexOutofBoundException

Exception sub classes:

ClassNotFoundException
DataFormatException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchMethodException
RuntimeException

Error sub classes :

ClassCirculatoryError
ClassFormatError
Error
IllegalAccessError
IncompatibleClassChangeError
InstantiationError
LinkageError
NoCassDefFoundError
NoSuchFieldError
NoSuchMethodError
OutofMemoryError
StackOverflowError
Throwable
UnknownError
UnsatisfiedLinkError
VerifyError
VirtualMachineError

Java Exception Handlers

- Syntax of try clause is exactly that of C++, except for the finally clause
- Syntax of catch clause is like those of C++, except every catch requires a named parameter and all parameters must be descendants of Throwable class.

Exceptions are thrown with throw, often the throw includes the new operator to create the object:

throw new MyException();

Checked and Unchecked Exceptions

The Java throws clause is quite different from the throw clause of C++

- unchecked and checked exceptions
  - Exceptions of class Error and RunTimeException and all of their descendants are called unchecked exceptions;
  - All other exceptions are called checked exceptions
- Checked exceptions that may be thrown by a method must be either:
  - Listed in the throws clause, or
  - Handled in the method

During the execution of a program, when an exceptional condition arises, an object of the respective exception class is created and thrown in the method which caused the exception.
That method may choose to catch the exception and then can guard against premature exit or may have a block of code execute.
Java exception handling is managed via five key words : try, catch, throw, throws, and finally.
Here is the basic form of an exception handling block.

```
try {
        // block of code
}
catch ( ExceptionType1 e) {
        // Exception handling routine for ExceptionType1 (optional)
}
catch (ExceptionType2 e ) {
        // Exception handling routine for ExceptionType2 (optional)
}
.
.
.
catch (ExceptionTypen e) {
        // Exception handling routine for ExceptionType_n (optional)
}
finally {
        // Program code of exit (optional)
}
```

```
class Exc0 {
  public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
  }
}
```

```
java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:4)
```

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
     d = 0;
     a = 42 / d;
     System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
     System.out.println("Division by zero.");
    }

    System.out.println("After catch statement.");
  }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Example of Exception Handling in java

```
import java.io.*;
class Test
{
public static void main(String args[]) throws IOException
{
int a[],b,c;
Scanner in=new Scanner(System.in);
a=new int[5];
for(int i=0;i<5;i++)
{
a[i]=Integer.parseInt(in.readLine());
}
//displaying the values from array
try{
for(int i=0;i<7;i++)
{
System.out.println(a[i]);
}
}
```

```
catch(Exception e)
{
System.out.println("The run time error is:"+e);
}
finally
 {
System.out.print(“I am always executable");
 }
}
}
```

O/P: 1 2 3 4 5

1 2 3 4 5
The runtime error is :
ArrayIndexOutOfBoundsException: 5
I am always executable

Evaluation

The types of exceptions makes more sense than in the case of C++

- The throws clause is better than that of C++

  The throw clause in C++ says little to the programmer

- The finally clause is often useful

- The Java interpreter throws a variety of exceptions that can be handled by user programs

```
class Example{
        public static void main (String args[ ]){
                int number, InvalidCount = 0, validCount = 0;
                for (int i = 0; i < args.length; i++)
                {
                        try {
                                number = Integer.parseInt(args[i]);
                        } catch (NumberFormatException e){
                                InvalidCount++;
                                System.out.println ("Invalid number at " + i +"  "+ args[i]);
                        }
                        validCount++;
                        System.out.println ("Valid number at " + i+" "+ args[i]);
                }
                System.out.println ("Invalid entries: " + InvalidCount);
                System.out.println ("Valid entries: " + validCount);
        }
}
```

/* Multiple errors with single catch block */

```
class Example  {
   public static int j;
   public static void main (String args[ ] ) {
            for (int i = 0; i < 4; i++ ) {
     try {
                                switch (i) {
                                        case 0 :
                                                int zero = 0;
                                                j = 999/ zero; // Divide by zero
                                                break;
                                        case 1:
                                                int b[ ] = null;
                                                j = b[0] ; // Null pointer error
                                                break;
                                        case 2:
                                                int c[] = new int [2] ;
                                                j = c[10]; // Array index is out-of-bound
                                                break;
                                        case 3:
                                                char ch = "Java".charAt(9) ;// String index is out-of-bound
                                                break;
                                }
     } catch (Exception e) {
        System.out.println("case#"+i+ "\n");
        System.out.println (e.getMessage() );
                        }   } }
```

oUTPUT:

```
case#0
/ by zero
case#1
null
case#2
10
case#3
String index out of range: 9
```

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```

```java
// Demonstrate multiple catch statements.
class MultipleCatches {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:
Index 42 out of bounds for length 1
After try/catch blocks.
```

```
/* This program contains an error.

   A subclass must come before its superclass in
   a series of catch statements. If not,
   unreachable code will be created and a
   compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
     } catch(Exception e) {
       System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

Nested try Statements

```
// An example of nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command-line arg is used,
           then a divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command-line args are used,
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:
  Index 42 out of bounds for length 1
```

Nested try Statements

```
/* Try statements can be implicitly nested via
   calls to methods. */
class MethNestTry {
  static void nesttry(int a) {
    try { // nested try block
      /* If one command-line arg is used,
         then a divide-by-zero exception
         will be generated by the following code. */
       if(a==1) a = a/(a-a); // division by zero

       /* If two command-line args are used,
          then generate an out-of-bounds exception. */
      if(a==2) {
        int c[] = { 1 };
        c[42] = 99; // generate an out-of-bounds exception
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index out-of-bounds: " + e);
    }
  }

  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;
      System.out.println("a = " + a);

      nesttry(a);
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:
  Index 42 out of bounds for length 1
```

throw

- **Program to throw an exception explicitly, using the throw statement.**
- **The general form of throw is shown here:**

  throw ThrowableInstance;
- **Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.**
- **Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.**
- **There are two ways you can obtain a Throwable object:**

  using a parameter in a catch clause or creating one with the new operator.

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

throws

- **Method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.**
- **A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.**
- **All other exceptions that a method can throw must be declared in the throws clause.**
  - If they are not, a compile-time error will result.

```
type method-name(parameter-list) throws exception-list
{
  // body of method
}
```

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

Finally clause

- **Exception handling has try-catch-finally construct, although the finally clause is purely optional.**
- **The finally clause defines a block of code which will be executed always of whether an exception was caught or not. .**
- **For example, before exiting a program, it may have to close some opened files and freeing up any other resources that might have been allocated at the beginning of a method.**

Finally clause

finally clause

- Can appear at the end of a try construct
- Form:

finally {

...

}

Example:

```
try {
for (index = 0; index < 100;
index++) {
…
if (…) {
return;
}
}
} // end of try construct
finally {
…
}
```

- Purpose: To specify code that is to be executed, regardless of what happens in the try construct
- A try construct with a finally clause can be used outside exception handling

/* finally in try-catch block */ code in finally block will be executed always the loop is iterated.

```
class Example    {
        public static void main (String [ ] args ) {
          int i = 0;
          String greetings[] = {"Hello James Gosling !", "Hello Java !", "Hello World ! "};
          while ( i < 4) {
            try {
                    System.out.println (greetings [i] );
                     i++;
                  }catch (Exception e ) {
                       System.out.println (e.toString() );
                    }
                finally {
                  System.out.println (" Hi !");
                   if (i < 3);
                     else  {System.out.println("Quit and reset the index value");break;}
                     }
               } //while
  }
}
```

```
// Demonstrate finally.
class FinallyDemo {
  // Throw an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");

    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }

  // Execute a try block normally.
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
      System.out.println("procC's finally");
    }
  }

  public static void main(String args[]) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Exception caught");
    }

    procB();
    procC();
  }
}
```

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Creating Your Own Exception Subclasses

- **Create the own exception types to handle situations specific to your applications.**
  - Exception( )
  - Exception(String msg)
- **The first form creates an exception that has no description.**
- **The second form specify a description of the exception**
- **The version of toString( ) defined by Throwable (and inherited by Exception) first displays the name of the exception followed by a colon, which is then followed by your description.**
- **By overriding toString( ), prevent the exception name and colon from being displayed.**

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}

class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Normal exit");
  }

  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

In Java, throw key word is known by which user can throw an exception of their own instead of automatic exception object generated by Java run time.
To use this, we have to create an instance of Throwable object. Then the throw key word can be used to throw this exception.

```
// File Name BankDemo.java
                public class BankDemo
                {
                  public static void main(String [] args)
                  {
                        CheckingAccount c = new CheckingAccount(1001);
                        System.out.println("Deposit INR 500...");
                        c.deposit(500.00);
                        try
                        {       System.out.println("\nWithdrawing INR100...");
                                c.withdraw(100.00);
                                System.out.println("\nWithdrawing INR 500...");
                                c.withdraw(600.00);
                        }catch(InsufficientBalanceException e)
                        {
                                System.out.println("Sorry, out of balance"+ e.getAmount());
                                e.printStackTrace();
                        }
                }}
```

```
// File Name CheckingAccount.java
public class CheckingAccount
{
  private double balance;
  private int number;
  public CheckingAccount(int number)
  {
        this.number = number;
  }
  public void deposit(double amount)
  {
        balance += amount;
  }
  public void withdraw(double amount) throws InsufficientBalanceException
  {
        if(amount <= balance)
        {   balance -= amount;   }
        else   {
        double needs = amount - balance;
        throw new InsufficientBalanceException(needs);  }
    }
  public double getBalance()   {
        return balance;
  }
 public int getNumber() {
        return number;
  }
}
```

```
// File Name InsufficientBalanceException.java
public class InsufficientBalanceException extends Exception
        {
          private double amount;
          public InsufficientBalanceException(double amount)
          {          this.amount = amount;          }
          public double getAmount()
          {          return amount;       }
}
```

Assertions

- **The primary use of assertion statements is for debugging and testing.**
- **Assertions are used to stop execution when "impossible" situations are detected**

  example: parameter can't possibly be null“

```
assert condition;
assert condition : expression;
assert expression1 : expression2;
```

- **Any failure in the assertion statement, then the JVM will throw an error that is labelled as an AssertionError.**
- **An effective way of detecting and correcting errors in a program**
- **There are two things that is needed to implement assertions in program**
  - **assert keyword and a boolean condition.**

Assertions

Statements in the program declaring a boolean expression regarding the current state of the computation

- **When evaluated to true nothing happens**
- **When evaluated to false an AssertionError exception is thrown**
- **Disabled during runtime without program modification or recompilation**

Assertions

```
public class AssertionExample {
    public static void main(String[] args) {
        int argCount = args.length;
        assert argCount == 5 : "The number of arguments must be 5";
        System.out.println("OK");
    }
}
```

- **Generally, assertion is enabled during development time to defect and fix bugs.**
- **Disabled at deployment or production to increase performance.**

```
java -ea AssertionExample 1 2 3 4

Exception in thread "main" java.lang.AssertionError: The number of arguments must be 5
        at AssertionExample.main(AssertionExample.java:6)
```

Exception handling

- **What is an exception?**
  - **An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.**
- **What is error?**
  - **An Error indicates that a non-recoverable condition has occurred that should not be caught.**
  - **Error, a subclass of Throwable, is intended for drastic problems, such as OutOfMemoryError, which would be reported by the JVM itself.**

Exception handling

What are the types of Exceptions in Java?

There are two types of exceptions in Java, unchecked exceptions and checked exceptions.

Checked exceptions: A checked exception is some subclass of Exception (or Exception itself), excluding class RuntimeException and its subclasses.

Each method must either handle all checked exceptions by supplying a catch clause or list each unhandled checked exception as a thrown exception.

Unchecked exceptions: All Exceptions that extend the RuntimeException class are unchecked exceptions. Class Error and its subclasses also are unchecked.

Exception handling

- **Why exception handling?**
  - **To Separate Error Handler Code from "Regular" Code.**
  - **To Propagate Errors Up the Call Stack.**
  - **To Group Error Types**
  - **To perform Error Differentiation.**

Can there be try block without catch block?

Yes, there can be try block without catch block, but finally block should follow the try block.

However it is invalid to use a try clause without either a catch clause or a finally clause.

What is the use of finally block?

Used to close files, release network sockets, connections, and any other cleanups.

When was the execution of finally clause happens?

The finally block code is always executed, whether an exception was thrown or not.

If the try block executes with no exceptions, the finally block is executed immediately after the try block completes.

If the try block throws an exception, the finally block executes immediately after the proper catch block completes

State the difference between throw and throws?

throws: Used in a method's signature and is capable of causing an exception that it does not handle and hence callers of the method must either have a try-catch clause to handle that exception or must be declared to throw that exception (or its superclass) itself..

```
public void throwFun(int arg) throws MyException {
  }
```

throw: Used to trigger an user-defined exception within a block. The exception will be caught by the try-catch clause that can catch that type of exception. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

```
throw new UserException("thrown an exception!");
```

Why Runtime Exceptions are Not Checked?

- The runtime exception classes (RuntimeException and its subclasses) are exempted from compile-time checking because such exceptions would not support the correctness of programs.

How to create user defined exceptions?

By extending the Exception class or one of its subclasses.

```
Example:
class MyException extends Exception {
  public MyException() { super(); }
  public MyException(String s) { super(s); }
  }
```

How to handle exceptions?

There are two ways to handle exceptions:

Try-catch clause

throws clause

Why Errors are Not Checked?

Error and its subclasses are an unchecked exception classes that are exempted from compile-time checking

Error can occur at many points in the program and recovery from them is difficult or impossible and hence checking such exceptions would be useless .

Example memory out of space