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, that disrupts the normal flow of the program.**
- **Exception is a run-time error.**
- Exceptions can cause a program to terminate abruptly if they are not properly caught and handled.

```java
public class ExceptionClass {
    public static void main(String[] args){
        double a=5.0/0;
        System.out.println("I am executable");
        int b=5/0;
        System.out.println("I am not executable");
    }
}
```

```java
public class ExceptionClass {
    public static void main(String[] args){
        double a=5.0/0;
        System.out.println("I am executable");
        int b=5/0;
        System.out.println("I am not executable");
    }
}
```

```
I am executable
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at ExceptionClass.main(ExceptionClass.java:5)
```

In Java, dividing a floating-point number (5.0) by zero does not throw an exception. Instead, it results in positive or negative infinity.

Dividing an integer by zero in Java throws an ArithmeticException.
Integer division by zero is undefined and not allowed.
This exception causes the program to terminate abruptly at this point, and the statement System.out.println("I am not executable"); is never reachable

```java
public class ExceptionClass {
    public static void main(String[] args){
        double a=5.0/0;
        System.out.println("I am executable");
        int b=5/0;
        System.out.println("I am not executable");
    }
}
```

```
I am executable
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at ExceptionClass.main(ExceptionClass.java:5)
```

JVM print an error message as a stack trace that helps the programmer to fix the bug.

A stack trace, prints the exception message and the method name, line number, and file name where the exception was thrown.
Next the method that called this method along with line number and file name.
This continues until the main method is reached.

```java
public class ExceptionClass {
    public static void main(String[] args) {
        double a = 5.0 / 0;
        System.out.println("I am executable");
        try {
            int b = 5 / 0;
        } catch (ArithmeticException ae) {
            System.out.println(ae + "\n I am at catch ArithmeticException");
        }
        System.out.println("I am now executable!");
    }
}
```

```
I am executable
java.lang.ArithmeticException: / by zero
 I am at catch ArithmeticException
I am now executable!
```

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.**
- Handling exceptions properly is crucial to building robust applications that can deal with errors gracefully and continue to operate smoothly.
- **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

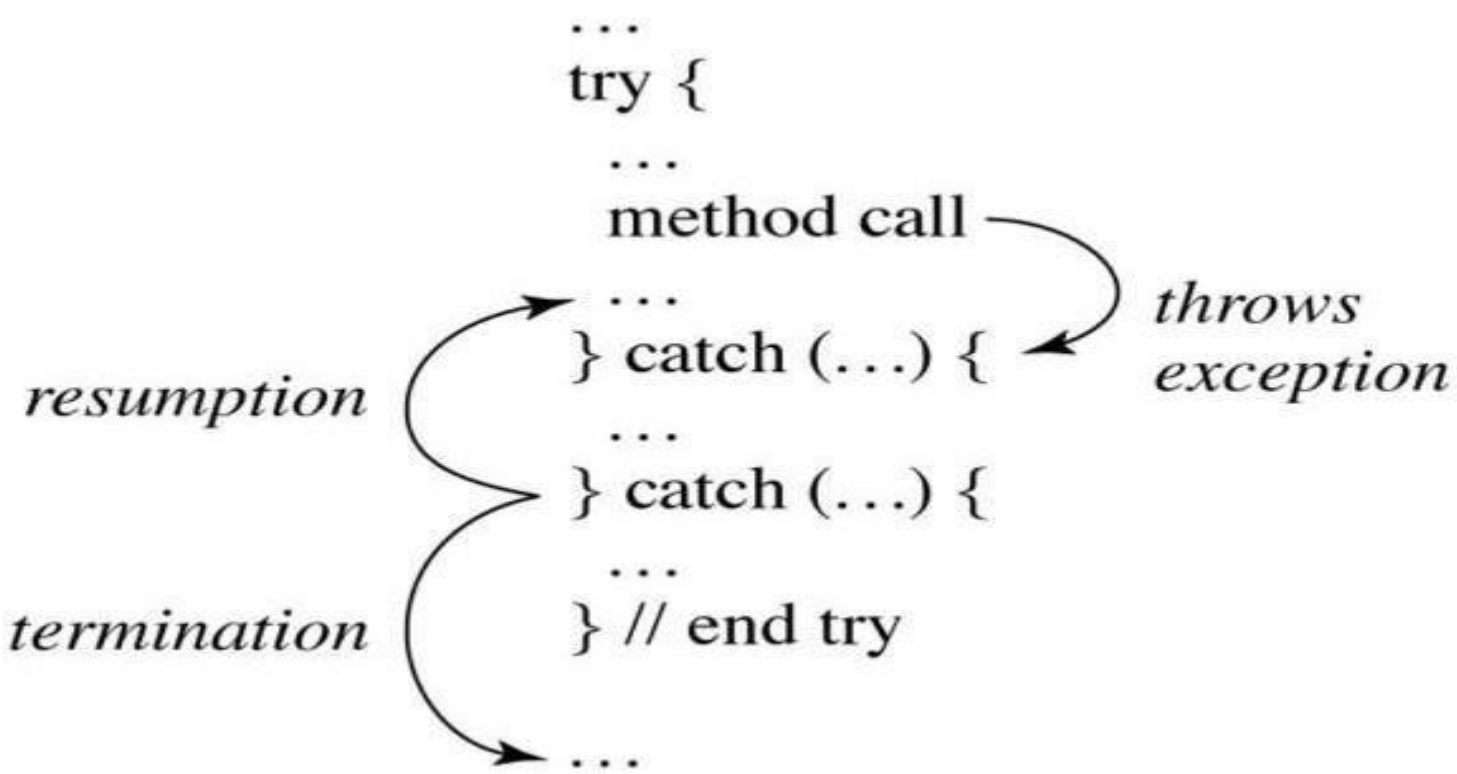
  In a language with exception handling

  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



```java
import java.util.Scanner;
public class ExceptionClassExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount( initialBalance: 1000);
        Scanner scanner = new Scanner(System.in);
        while (true) {
            try {
                // exception-throwing operation
                System.out.print("Enter amount to withdraw (or 'q' to quit): ");
                String input = scanner.nextLine();
                if (input.equalsIgnoreCase( anotherString: "q")) {
                    break; // Termination: exit the loop
                }
                double amount = Double.parseDouble(input);
                account.withdraw(amount);
                // Resumption: continue with next operation
                System.out.println("Withdrawal successful. New balance: Rs" + account.getBalance());
            } catch (NumberFormatException e) { !!!!!
                // Handle invalid input
                System.out.println("Invalid input. Please enter a valid number.");
                // Resumption: continue with next iteration
            } catch (InsufficientFundsException e) {
                // Handle insufficient funds
                System.out.println(e.getMessage());
                // Resumption: continue with next iteration
            } catch (Exception e) {
                // Handle any other unexpected exceptions
                System.out.println("An unexpected error occurred: " + e.getMessage());
                // Termination: exit the loop
                break;
            }
        } System.out.println("Thank you for using our banking system.");        }    }
```

#58!

```java
class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Insufficient funds. Current balance: $" + balance);
        }
        balance -= amount;
    }
    public double getBalance() {
        return balance;
    }
}
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

//method THROWSSSSS and exception!!!

manually throwing an error using THROW keyword!

//Custom exception!!!

```java
import java.util.Scanner;
public class ExceptionClassExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount( initialBalance: 1000);
        Scanner scanner = new Scanner(System.in);
        while (true) {
            try {
                // exception-throwing operation
                System.out.print("Enter amount to withdraw (or 'q' to quit): ");
                String input = scanner.nextLine();
                if (input.equalsIgnoreCase( anotherString: "q")) {
                    break; // Termination: exit the loop
                }
                double amount = Double.parseDouble(input);
                account.withdraw(amount);
                // Resumption: continue with next operation
                System.out.println("Withdrawal successful. New balance: Rs" + account.getBalance());
            } catch (NumberFormatException e) {
                // Handle invalid input
                System.out.println("Invalid input. Please enter a valid number.");
                // Resumption: continue with next iteration
            } catch (InsufficientFundsException e) {
                // Handle insufficient funds
                System.out.println(e.getMessage());
                // Resumption: continue with next iteration
            } catch (Exception e) {
                // Handle any other unexpected exceptions
                System.out.println("An unexpected error occurred: " + e.getMessage());
                // Termination: exit the loop
                break;
            }
        } System.out.println("Thank you for using our banking system.");        }    }
```

```
Enter amount to withdraw (or 'q' to quit): java
Invalid input. Please enter a valid number.

Enter amount to withdraw (or 'q' to quit): 2000
Insufficient funds. Current balance: $1000.0

Enter amount to withdraw (or 'q' to quit): 500
Withdrawal successful. New balance: $500.0

Enter amount to withdraw (or 'q' to quit): q
Thank you for using our banking system.

Process finished with exit code 0
```

```java
class BankAccount {
                                                                                    nt balance: $" + balance);
        }
        balance -= amount;
    }
    public double getBalance() {
        return balance;
    }
}
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

Exception

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

Reasons for exceptions:

- **hardware events**
  - **arithmetic overflow, parity errors**
    - adding two large integers exceeds the maximum value of integer type.
    - **parity errors :RAM read may result in** data corruption
- operating system events
  - **out of memory exception**
    - if the JVM cannot allocate sufficient heap space for large objects.
- **programming language properties**
  - **dynamic range checks**
    - y > Integer.MAX_VALUE
- **application program properties**
  - **data structure overflow/underflow**
    - int[] numbers = {1, 2, 3}; int value = numbers[5]; // ArrayIndexOutOfBoundsException

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;
         }
         return 0;
      }
```

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];

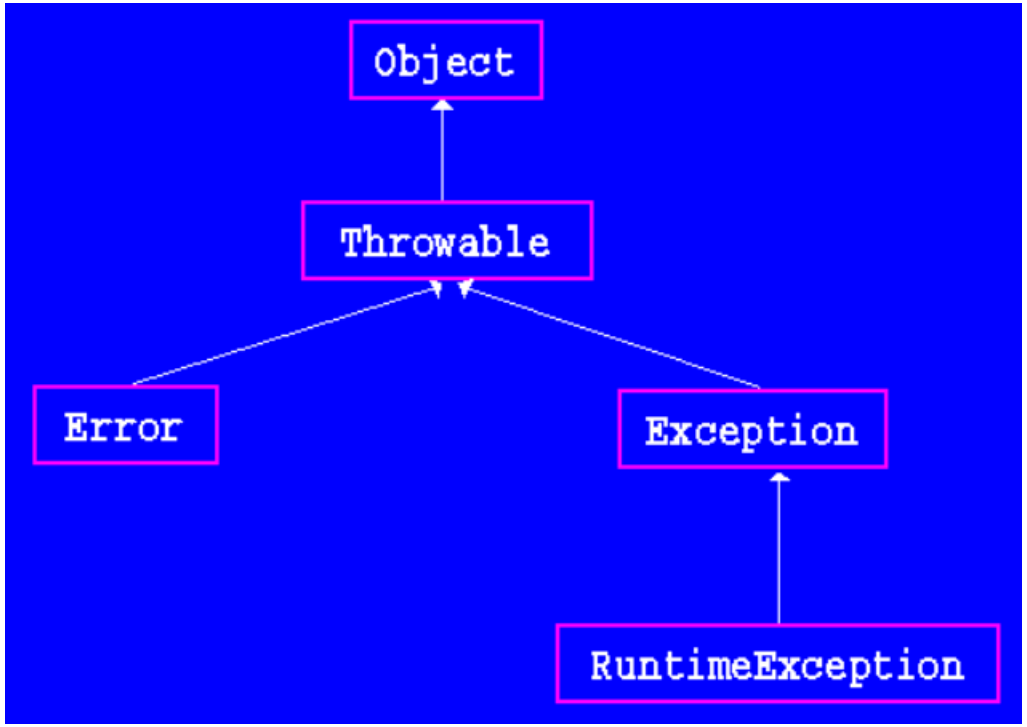
- The brackets are metasymbols
- A throw without an operand can only appear in a handler;
- when it appears, it simply re-raises the exception, which is then handled elsewhere
- The type of the expression disambiguates the intended handler

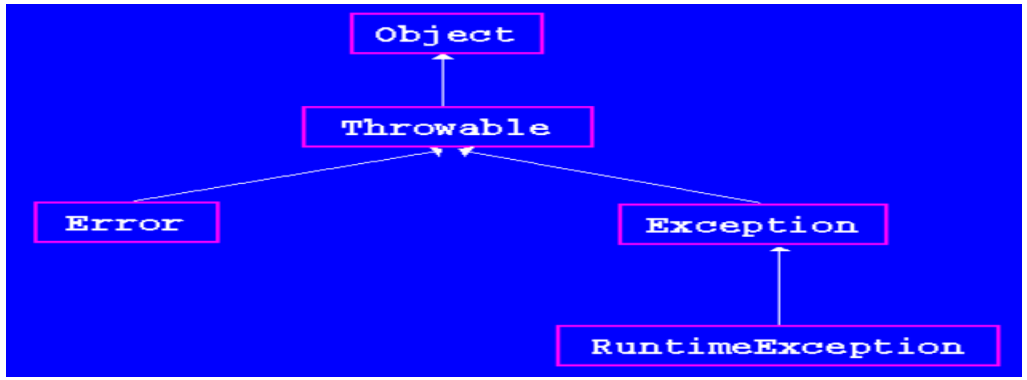
Exception Handling in Java

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 {
...
 }
```

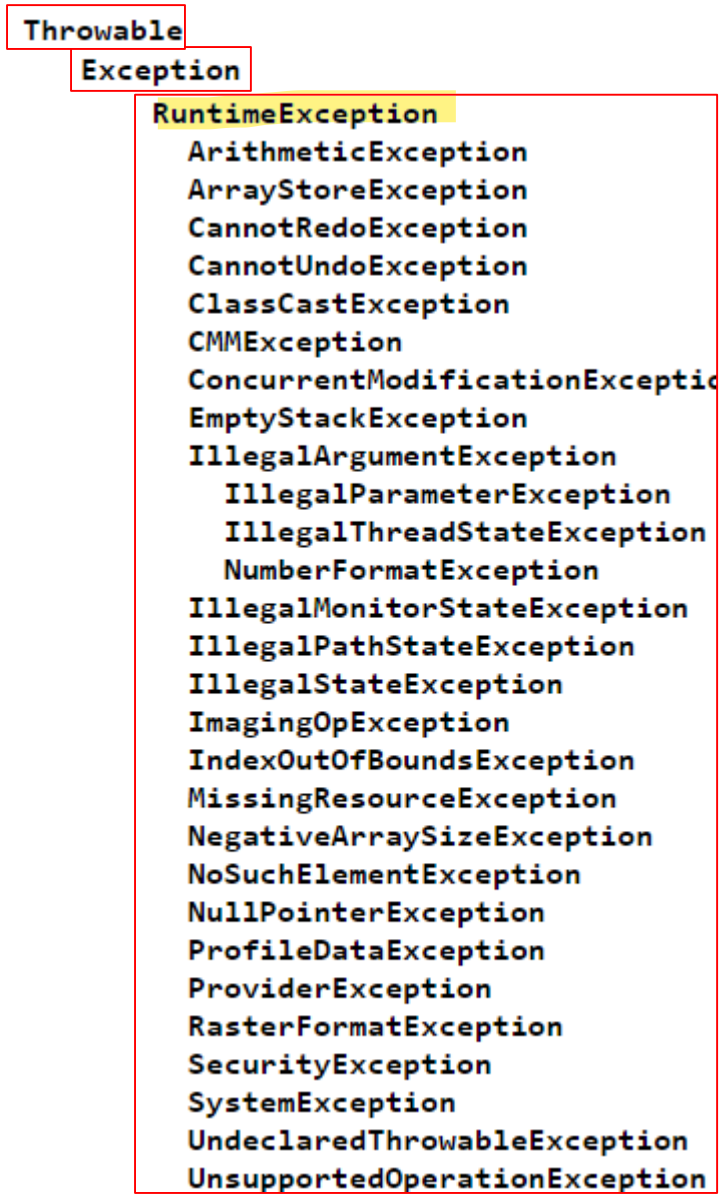


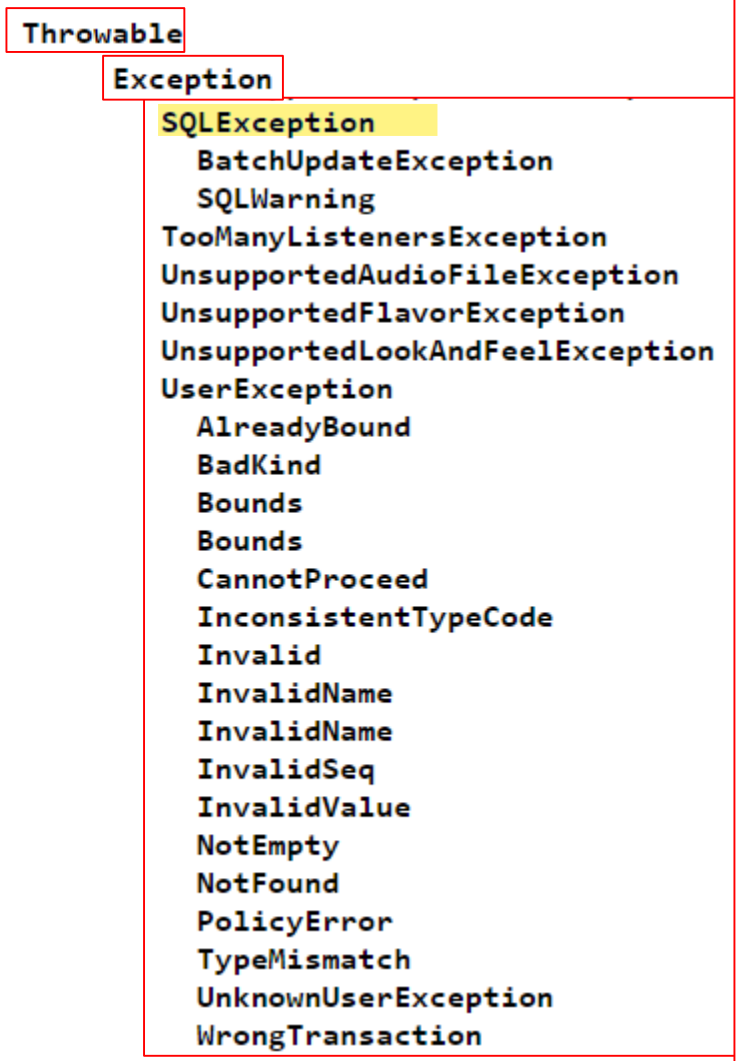


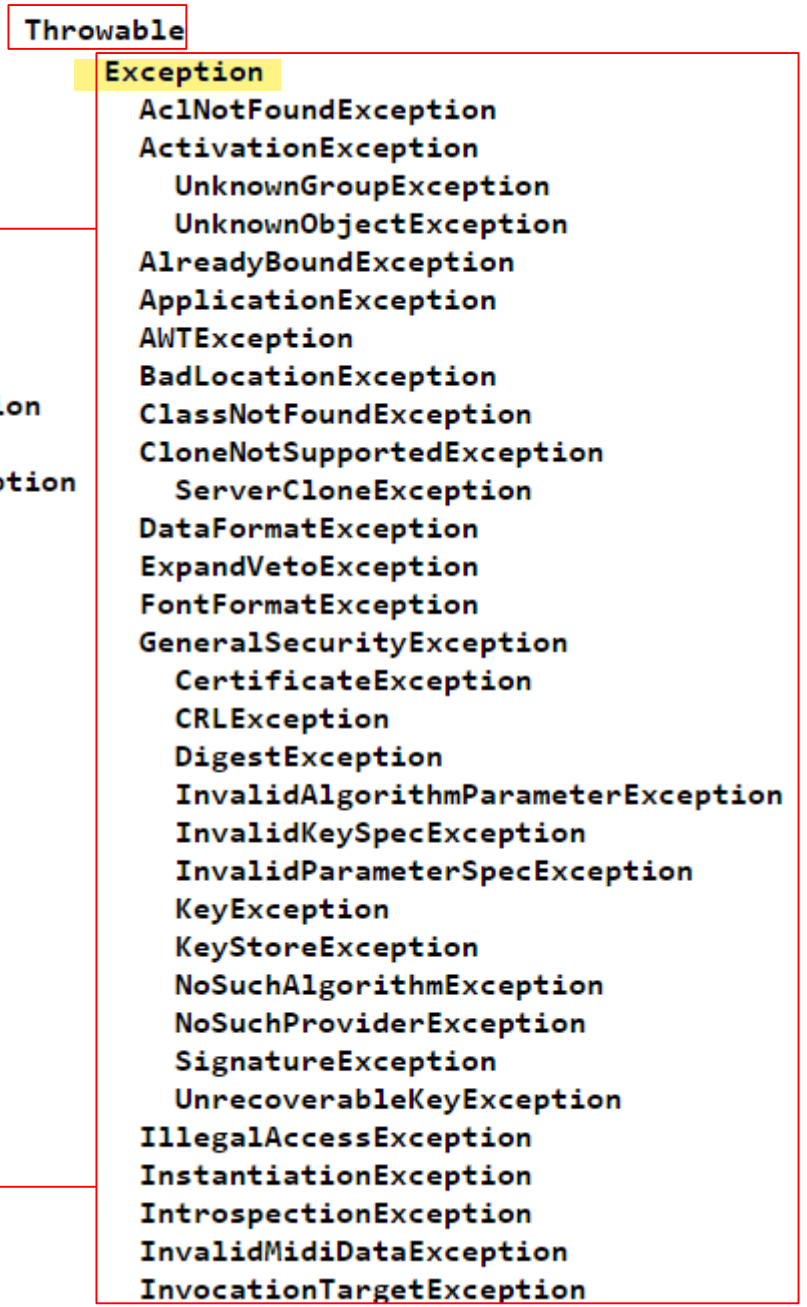

```
Throwable
  Error
    AWTError
    LinkageError
     ClassCircularityError
     ClassFormatError
     ExceptionInInitializerError
     IncompatibleClassChangeError
     NoClassDefFoundError
     UnsatisfiedLinkError
     VerifyError
    ThreadDeath
    VirtualMachineError
      InternalError
      OutOfMemoryError
      StackOverflowError
      UnknownError
```

```
Throwable
    Exception
      IOException
        ChangedCharSetException
        CharConversionException
        EOFException
        FileNotFoundException
        InterruptedIOException
        MalformedURLException
        ObjectStreamException
        ProtocolException
        RemoteException
        SocketException
        SyncFailedException
        UnknownHostException
        UnknownServiceException
        UnsupportedEncodingException
        UTFDataFormatException
        ZipExcept
```

```
Throwable
    Exception
      NamingException
        AttributeInUseException
        AttributeModificationException
        CannotProceedException
        CommunicationException
        ConfigurationException
        ContextNotEmptyException
        InsufficientResourcesException
        InterruptedNamingException
        InvalidAttributeIdentifierExcep
        InvalidAttributesException
        InvalidAttributeValueException
        InvalidNameException
        InvalidSearchControlsException
        InvalidSearchFilterException
        LimitExceededException
        LinkException
        NameAlreadyBoundException
        NameNotFoundException
        NamingSecurityException
        NoInitialContextException
        NoSuchAttributeException
        NotContextException
        OperationNotSupportedException
        ReferralException
        SchemaViolationException
        ServiceUnavailableException
      NoninvertibleTransformException
      NoSuchFieldException
      NoSuchMethodException
      NotBoundException
      NotOwnerException
      ParseException
      PartialResultException
      PrinterException
        PrinterAbortException
        PrinterIOException
      PrivilegedActionException
      RemarshalException
```


Throwable
Exception
RuntimeException
ArithmeticException
ArrayStoreException
CannotRedoException
CannotUndoException
ClassCastException
CMMException
ConcurrentModificationExceptic
EmptyStackException
IllegalArgumentException
IllegalParameterException
IllegalThreadStateException
NumberFormatException
IllegalMonitorStateException
IllegalPathStateException
IllegalStateException
ImagingOpException
IndexOutOfBoundsException
MissingResourceException
NegativeArraySizeException
NoSuchElementException
NullPointerException
ProfileDataException
ProviderException
RasterFormatException
SecurityException
SystemException
UndeclaredThrowableException
UnsupportedOperationException


Throwable
Exception
SQLException
BatchUpdateException
SQLWarning
TooManyListenersException
UnsupportedAudioFileException
UnsupportedFlavorException
UnsupportedLookAndFeelException
UserException
AlreadyBound
BadKind
Bounds
Bounds
CannotProceed
InconsistentTypeCode
Invalid
InvalidName
InvalidName
InvalidSeq
InvalidValue
NotEmpty
NotFound
PolicyError
TypeMismatch
UnknownUserException
WrongTransaction


Throwable
Exception
AclNotFoundException
ActivationException
UnknownGroupException
UnknownObjectException
AlreadyBoundException
ApplicationException
AWTException
BadLocationException
ClassNotFoundException
CloneNotSupportedException
ServerCloneException
DataFormatException
ExpandVetoException
FontFormatException
GeneralSecurityException
CertificateException
CRLException
DigestException
InvalidAlgorithmParameterException
InvalidKeySpecException
InvalidParameterSpecException
KeyException
KeyStoreException
NoSuchAlgorithmException
NoSuchProviderException
SignatureException
UnrecoverableKeyException
IllegalAccessException
InstantiationException
IntrospectionException
InvalidMidiDataException
InvocationTargetException

Classes of Exceptions

Throwable

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

The top three classes in this hierarchy ,**Throwable**, **Error**, and **Exception** classes are all defined in the **java.lang** package

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 exceptions
  - Exceptions of class Error and RunTimeException and all of their descendants are called unchecked exceptions
  - All other exceptions are called checked exceptions
  - These exceptions are not checked by the compiler, and hence, need not be caught or declared to be thrown in the program
  - They are programming logical errors that can be fixed in compiled-time, rather than leaving it to runtime exception handling.
- Checked exceptions :
  - They are checked by the compiler and must be caught or declared to be thrown.
    - Listed in the throws clause, or
    - Handled using try catch 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 DivisionExample {
    public static void main(String[] args) {
        int divisor = 0;
        int result = 42 / divisor;
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at ExceptionExamplee.main(ExceptionExamplee.java:4)

Process finished with exit code 1
```

```java
public class ExceptionHandlerExample1 {
    public static void main(String[] args) {
        int divisor, quotient;
        try { // monitor a block of code
            divisor = 0;
            quotient = 42 / divisor;
            System.out.println("This statement won't be executed.");
        }
        catch (ArithmeticException ex) { // catch divide-by-zero error
            System.out.println("Error: Division by zero occurred.");
        }
        System.out.println("Execution continues after try-catch block.");
    }
}
```

```
Error: Division by zero occurred.
Execution continues after try-catch block.

Process finished with exit code 0
```

```java
public class ExceptionExample1 {
    static void calculateQuotient() {
        int divisor = 0;
        int quotient = 10 / divisor;
    }

    public static void main(String[] args) {
        calculateQuotient(); //stack trace
    }
}
```

look (stack space tree)---from method--then_to---->main

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at ExceptionExample1.calculateQuotient(ExceptionExample1.java:4)
    at ExceptionExample1.main(ExceptionExample1.java:8)

Process finished with exit code 1
```

Example of Exception Handling in java

```java
import java.util.Scanner;
public class ExceptionExample {
    public static void main(String args[])  {
        int a[], b, c;
        Scanner input = new Scanner(System.in);
        a = new int[5];
        for (int i = 0; i < 5; i++) {
            a[i] = Integer.parseInt(input.nextLine());
        }
        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("Am I always executable?");
        }
    }
}
```

//IndexOutOfBoundError!!

```
1
2
3
4
5
1
2
3
4
5
The run time error is:java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
Am I always executable?
Process finished with exit code 0
```

```java
import java.util.Scanner;
public class ExceptionExample {
    public static void main(String args[])  {
        int a[], b, c;
        Scanner input = new Scanner(System.in);
        a = new int[5];
        for (int i = 0; i < 5; i++) {
            a[i] = Integer.parseInt(input.nextLine());
        }
        try {
            for (int i = 0; i < 7; i++) {
                if(i>4) System.exit( status: 0); //finally block not executed
                System.out.println(a[i]);
            }
        }
        catch(Exception e)
        {
            System.out.println("The run time error is:"+e);
        }
        finally
        {
            System.out.print("Am I always executable?");
        }
    }
}
```

using System.exit(0);

```
1
2
3
4
5
1
2
3
4
5

Process finished with exit code 0
```

```java
import java.util.Scanner;
public class ExceptionExample {
    public static void main(String args[])  {
        int a[], b, c;
        Scanner input = new Scanner(System.in);
        a = new int[5];
        for (int i = 0; i < 5; i++) {
            a[i] = Integer.parseInt(input.nextLine());
        }
        try {
            for (int i = 0; i < 7; i++) {
                if(i>4) return;
                System.out.println(a[i]);
            }
        }
        catch(Exception e)
        {
            System.out.println("The run time error is:"+e);
        }
        finally
        {
            System.out.print("Am I always executable?");
        }
    }
}
```

//Finally is always executed

```
1
2
3
4
5
1
2
3
4
5
Am I always executable?
Process finished with exit code 0
```

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

```java
public class RuntimeExceptionExample {
    public static void main (String args[ ]){
        int number, InvalidCount = 0, validCount = 0;
        String[] str={"1", "2", "a","4"};
        for (int i = 0; i < str.length; i++){
            try {
                number = Integer.parseInt(str[i]);
                validCount++;
                System.out.println ("Valid number at " + i+" "+ str[i]);
            }
            catch (NumberFormatException e){
                InvalidCount++;
                System.out.println ("Invalid number at " + i +" "+ str[i]);
            }
        }
        System.out.println ("Invalid entries: " + InvalidCount);
        System.out.println ("Valid entries: " + validCount);
    }
}
```

//Runt-time exception+NumberFormatException

```
Valid number at 0 1
Valid number at 1 2
Invalid number at 2 a
Valid number at 3 4
Invalid entries: 1
Valid entries: 3
```

/* Multiple errors with single catch block */

```java
public class MultipleErrorSingleCatch {
    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);
               System.out.println (e.getMessage() );
           }
           }
       }
   }
```

```
case#0
/ by zero
case#1
Cannot load from int array because "b" is null
case#2
Index 10 out of bounds for length 2
case#3
String index out of range: 9

Process finished with exit code 0
```

```java
import java.util.Random;

// Handle an exception and continue execution.
class ExceptionHandlerExample {
    public static void main(String[] args) {
        int x=0 , y = 0, z = 0;
        Random randomGenerator = new Random();
        for (int i = 0; i < 5; i++) {
            try {
                y = randomGenerator.nextInt();
                z = randomGenerator.nextInt();
                x = 123 / (y / z);
            } catch (ArithmeticException ex) {
                System.out.println("I : "+ i+ " x: " + x + " y: " + y + " z: " + z + " Error: Division by zero detected.");
                x = 0; // set x to zero and continue
            }
            System.out.println("I : "+ i+ " x: " + x + " y: " + y + " z: " + z);
        }
    }
}
```

```
I : 0 x: 0 y: -109831734 z: 787149652 Error: Division by zero detected.
I : 0 x: 0 y: -109831734 z: 787149652
I : 1 x: 61 y: -1717928116 z: -857147401
I : 2 x: 61 y: 936171523 z: -1918638870 Error: Division by zero detected.
I : 2 x: 0 y: 936171523 z: -1918638870
I : 3 x: 0 y: 1590138376 z: -1824645867 Error: Division by zero detected.
I : 3 x: 0 y: 1590138376 z: -1824645867
I : 4 x: 0 y: -1177849097 z: 1597335688 Error: Division by zero detected.
I : 4 x: 0 y: -1177849097 z: 1597335688
```

```java
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int x = args.length;
            System.out.println("x = " + x);
            int y = 42 / x;
            int[] arr = { 1 };
            arr[42] = 99;
        } catch(ArithmeticException ex) {
            System.out.println("Arithmetic error: " + ex);
        } catch(ArrayIndexOutOfBoundsException ex) {
            System.out.println("Array index error: " + ex);
        }
        System.out.println("Execution continues after try-catch blocks.");
    }
}
```

```
x = 0
Arithmetic error: java.lang.ArithmeticException: / by zero
Execution continues after try-catch blocks.
```

/*Arithmetic and ArrayIndexOutOfBoundException */

```java
public class MultipleCatchExample {
    public static void main(String[] args) {
            try {
                int x = 1;
                System.out.println("x = " + x);
                int y = 42 / x;
                int[] arr = { 1 };
                arr[42] = 99;
            } catch(ArithmeticException ex) {
                System.out.println("Arithmetic error: " + ex);
            } catch(ArrayIndexOutOfBoundsException ex) {
                System.out.println("Array index error: " + ex);
            }
            System.out.println("Execution continues after try-catch blocks.");
        }
    }
```

```
x = 1
Array index error: java.lang.ArrayIndexOutOfBoundsException: Index 42 out of bounds for length 1
Execution continues after try-catch blocks.
```

/* This code demonstrates an error in exception handling.

In a series of catch blocks, subclass exceptions should be caught before superclass exceptions.
Otherwise, unreachable code will be created, resulting in a compile-time error.
*/

```java
class ExceptionOrderDemo {
    public static void main(String[] args) {
        try {
            int denominator = 0;
            int result = 100 / denominator;
        } catch(Exception ex) {
            System.out.println("Caught general exception.");
        }
        /* This catch block is unreachable because
           ArithmeticException is a subclass of Exception. */
        catch(ArithmeticException ex) { // ERROR - unreachable
            System.out.println("This block is never executed.");
        }
    }
}
```

Superclass shouldn't come before in try-catch block!!!

```
java: exception java.lang.ArithmeticException has already been caught
```

/* This code demonstrates an error in exception handling.

In a series of catch blocks, subclass exceptions should be caught before superclass exceptions.
Otherwise, unreachable code will be created, resulting in a compile-time error.
*/

```java
class ExceptionOrderDemo {
    public static void main(String[] args) {
        try {
            int denominator = 0;
            int result = 100 / denominator;
        } catch(ArithmeticException ex) {
            System.out.println("Caught specific exception.");
        }
        catch(Exception ex) {
            System.out.println("This block is never executed.");
        }
    }
}
```

```
Caught specific exception.

Process finished with exit code 0
```

```
// Example of nested try statements
class NestedTryExample {
    public static void main(String[] args) {
        try {
            int argCount = args.length;
             int result = 100 / argCount;
            System.out.println("Number of arguments: " + argCount);
            try { // nested try block
            /* If exactly one command-line arg is used, this will cause a divide-by-zero exception. */
                if(argCount == 1) result = argCount / (argCount - argCount);
            /* If two command-line args are used,this will generate an out-of-bounds exception. */
                if(argCount == 2) {
                    int[] numbers = { 1 };
                    numbers[10] = 99; // generate an out-of-bounds exception
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index error: " + e);
            }
        } catch(ArithmeticException e) {
            System.out.println("Arithmetic error: " + 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
```

ry Statements

```
/* Try statements can be implicitly nested via method calls. */
class MethodNestedTryDemo {
   static void nestedTryMethod(int value) {
      try { // nested try block
         /* If value is 1, this will cause a divide-by-zero exception. */
         if(value == 1) value = value / (value - value);
          /* If value is 2, this will generate an out-of-bounds exception. */
         if(value == 2) {
            int[] numbers = { 1 };
            numbers[10] = 99; // generate an out-of-bounds exception
         }
      } catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Array bounds error: " + e);
      }   }
      public static void main(String[] args) {
      try {
         int argCount = args.length;
         /* If no command-line args are present,this will cause a divide-by-zero exception. */
         int result = 100 / argCount;
         System.out.println("Argument count: " + argCount);
         nestedTryMethod(argCount);
      } catch(ArithmeticException e) {
         System.out.println("Calculation error: " + e);
      }   } }
```

Calculation error: / by zero

Argument count: 1
Calculation error: / by zero

Argument count: 2
Array bounds error:
java.lang.ArrayIndexOutOfBoundsException: Index 10 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 exception throwing and rethrowing
class ExceptionPropagationDemo {
    static void exceptionGenerator() {
        try {
            throw new NullPointerException("Example exception");
        } catch(NullPointerException ex) {
            System.out.println("Exception caught in exceptionGenerator.");
            throw ex; // rethrow the exception
        }
    }

    public static void main(String[] args) {
        try {
            exceptionGenerator();
        } catch(NullPointerException ex) {
            System.out.println("Exception recaught in main: " + ex);
        }
    }
}
```

Output:
Exception caught in exceptionGenerator.
Exception recaught in main: java.lang.NullPointerException:
Example exception

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();
   }
}
```

```
java: unreported exception java.lang.IllegalAccessException; must be caught or declared to be thrown
```

It should either have a try-catch block to handle or should extend THROWS!!!

not extend,it should throws that exceptions

```java
// 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);
        }
    }
}
```

```
Inside throwOne.
Caught java.lang.IllegalAccessException: demo

Process finished with exit code 0
```

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

NEW!!!!

- 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.

How many times will this fact be repeated:{

```
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      }    }
```

```
Hello James Gosling !
 Hi !
Hello Java !
 Hi !
Hello World !
 Hi !
Quit and reset the index value

Process finished with exit code 0
```

```
public class FinallyBlockExample {
        // 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");
        }
    }
```

error raised!!

no-error!
but finally is executed regardless of return in function!!

```
// 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();
    }
}
```

no-error in code!!

MAIN!!

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

Process finished with exit code 0
```

```
// custom exception type.        //Example for custom exception!!!
class MyException extends Exception {
    private int age;
    MyException(int age) {
      this.age = age;
    }
    public String toString() {
        return "Exception[" + age + "]";
    } }
class CustomException  {
    static void checkAge(int age) throws MyException {
        System.out.println("Called checkAge(" + age + ")");
        if (age <18)
            throw new MyException(age);
        else
            System.out.println("you are eligible to vote");
        System.out.println("Normal exit");     }
    public static void main(String args[]) {      //MAIN!!!!
        try {
            checkAge(18);
            checkAge(15);
        } catch (MyException e) {
            System.out.println("Exception Caught " + e);
        }     }  }
```

```
Called checkAge(18)
you are eligible to vote
Normal exit
Called checkAge(15)
Exception Caught Exception[15]
```

Overriding method

list count should match!

return-type should be the same!

An overriding method must have the same argument list and return-type.

can go to less restricted acesss!

An overriding method cannot have more restricted access. a method with protected access may be overridden to have protected or public access but not private or default access.

An overloading method must have different argument list, but it can have any return-type.

Key Points:
1.For primitive types and void, the return type must match exactly.
2.For object types, the return type can be a subclass (covariant return).
3.If the superclass method returns an interface, the overriding method can return any class that implements that interface.
In summary, an overriding method cannot have any return type—it must either match the return type of the superclass method or return a subtype in the case of reference types.

Exceptions in overriding

- **An overriding method cannot declare new exception types**
- **An overriding method can declare exception types that are the same as or subclasses of the original**
- **An overriding method can declare fewer exceptions than the original**

```java
public class MethodStack {
    public void methodA() {
        System.out.println("Enter methodA()");
        methodB();
        System.out.println("Exit methodA()");
    }
    public void methodB() {
        System.out.println("Enter methodB()");
        methodC();
        System.out.println("Exit methodB()");
    }
    public void methodC() {
        System.out.println("Enter methodC()");
        System.out.println("Exit methodC()");
    }
    public static void main(String[] args) {
        MethodStack obj=new MethodStack();
        System.out.println("Enter main()");
        obj.methodA();
        System.out.println("Exit main()");
    }

}
```

//how function call works is explained here!!!

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit methodC()
Exit methodB()
Exit methodA()
Exit main()
```

```java
public class MethodStack {
    public void methodA() {
        System.out.println("Enter methodA()");
        methodB();
        System.out.println("Exit methodA()");
    }
    public void methodB() {
        System.out.println("Enter methodB()");
        methodC();
        System.out.println("Exit methodB()");
    }
    public void methodC() {
        System.out.println("Enter methodC()");
         System.out.println(5 / 0);
        System.out.println("Exit methodC()");
    }
    public static void main(String[] args) {
        MethodStack obj=new MethodStack();
        System.out.println("Enter main()");
        obj.methodA();
        System.out.println("Exit main()");
    }

}
```

//stackTrace() of function calls!!!

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at MethodStack.methodC(MethodStack.java:19)
    at MethodStack.methodB(MethodStack.java:9)
    at MethodStack.methodA(MethodStack.java:4)
    at MethodStack.main(MethodStack.java:25)
```

```
public class MethodStack {
    public void methodA() {
        System.out.println("Enter methodA()");
        methodB();
        System.out.println("Exit methodA()");
    }
    public void methodB() {
        System.out.println("Enter methodB()");
        methodC();
        System.out.println("Exit methodB()");
    }
    public void methodC() {
        System.out.println("Enter methodC()");
       //unchecked exception propagate
        System.out.println(5 / 0);
        System.out.println("Exit methodC()");
    }
```

```
    public static void main(String[] args) {
        public static void main(String[] args) {
        try {
            MethodStack obj = new MethodStack();
            System.out.println("Enter main()");
            obj.methodA();
        }
        catch(ArithmeticException e){
            System.out.println(e.getMessage());
        }
        finally {
            System.out.println("I am from finally block");
        }
        System.out.println("Exit main()");
    }
}
```

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
/ by zero
I am from finally block
Exit main()
```

```java
//use throws keyword for checked exception
// No need to use throws for unchecked exception
public class MethodStack {
    public void methodA() throws IOException{
        System.out.println("Enter methodA()");
        methodB();
        System.out.println("Exit methodA()");
    }
    public void methodB()  {
        System.out.println("Enter methodB()");
        methodC();
        System.out.println("Exit methodB()");
    }
    public void methodC() throws IOException {
      System.out.println("Enter methodC()");
       throw new IOException("Exception from method C");
      //ERROR : Unreachable statement
      //System.out.println("Exit methodC()");
    }
```

```java
public static void main(String[] args) {
        public static void main(String[] args) {
        try {
            MethodStack obj = new MethodStack();
            System.out.println("Enter main()");
            obj.methodA();
        }
        catch(ArithmeticException e){
            System.out.println(e.getMessage());
        }
        finally {
            System.out.println("I am from finally block");
        }
        System.out.println("Exit main()");
    }
}
```

No, unchecked exceptions do not require the throws keyword.

```
java: unreported exception java.io.IOException; must be caught or declared to be thrown
```

```java
//use throws keyword for checked exception
// No need to use throws for unchecked exception
public class MethodStack {
    public void methodA() throws IOException{
        System.out.println("Enter methodA()");
        methodB();
        System.out.println("Exit methodA()");
    }
    public void methodB() throws IOException  {
        System.out.println("Enter methodB()");
        methodC();
        System.out.println("Exit methodB()");
    }
    public void methodC() throws IOException {
      System.out.println("Enter methodC()");
        throw new IOException("Exception from method C");
      //ERROR : Unreachable statement
      //System.out.println("Exit methodC()");
    }
```

No, throw itself is not a checked exception. Rather, it is a keyword used to explicitly throw an exception in Java. The exception being thrown can be either a checked exception or an unchecked exception.

```java
public static void main(String[] args) {
        public static void main(String[] args) {
        try {
            MethodStack obj = new MethodStack();
            System.out.println("Enter main()");
            obj.methodA();
//checked exception propagate only through throws statement
        }
        catch(IOException e){
            System.out.println(e.getMessage());
        }
       finally {
            System.out.println("I am from finally block");
        }
        System.out.println("Exit main()");
    }
}
```

NOTE!!!!!

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception from method C
I am from finally block
Exit main()
```

```
import java.io.FileNotFoundException; import java.io.IOException;
import java.sql.SQLException;
   class GrandParent {
      void method1() throws Exception {
         System.out.println("GrandParent's method1");        }
      void method2() throws IOException, SQLException {
         System.out.println("GrandParent's method2");        }
   }
   class Parent extends GrandParent {
      @Override
      void method1() throws IOException { //a more specific exception
         System.out.println("Parent's method1");        }
      @Override
      void method2() throws IOException { // with fewer exceptions
         System.out.println("Parent's method2");        }
   }
   class Child extends Parent {
      @Override
      void method1() throws FileNotFoundException { // more specific
            System.out.println("Child's method1");        }
      @Override
      void method2() { // Overriding without declaring any exceptions
         System.out.println("Child's method2");        }
// void method2() throws Exception { // Error: a new exception
   //    System.out.println("Child's method3");        }    }
```

a subclass of that exception!

less no of exception count!!

```
public class ExceptionInOverriding  {      MAIN!!!
      public static void main(String[] args) {
         Child child = new Child();
         try {
            child.method1(); // Only catch FileNotFoundException
         }
         catch (FileNotFoundException e) {
            System.out.println("FileNotFoundException caught");
         }
         child.method2(); // No need to catch any exceptions
         GrandParent gp = new Child();
         try {
            gp.method1(); // only catch FileNotFoundException
            gp.method2(); // No need to catch any exceptions
         } catch (Exception e) {
   // use broadest exception type declared in GrandParent
            System.out.println("Exception caught");
         }
      }    }
```

```
Child's method1
Child's method2
Child's method1
Child's method2
```

Exceptions in overloading

- Overloaded methods must be differentiated by their argument list.
- They cannot be differentiated solely by return type, exceptions, or modifiers.
- They can have any return type, access modifier, and exceptions, as long as the argument list is different.

```
import java.io.IOException;
public class ExceptionInOverloading {
   // Overloaded methods with different argument lists
   public void method(int a) {
      System.out.println("int: " + a);     }
   public void method(int a, int b) {
      System.out.println(" ints: " + a + ", " + b);     }
   // Overloaded method with different return type
   public double method(double a) {
      System.out.println(" double: " + a);
      return a * 2;     }
   // Overloaded method with different access modifier
   protected void method(String s) {
      System.out.println(" String: " + s);     }
   // Overloaded method that throws an exception
   public void method(boolean b) throws IOException {
      System.out.println(" boolean: " + b);
      if (!b) {
         throw new IOException("False condition");  }     }
// public int method (int a) { //error
//     return a;
// }
```

```
public static void main(String[] args) {
   ExceptionInOverloading  obj = new ExceptionInOverloading ();
   obj.method(5);
   obj.method(5, 10);
   double result = obj.method(3.14);
   System.out.println("Result : " + result);
   obj.method("Hello");
   try {
      obj.method(true);
      obj.method(false);
   } catch (IOException e) {
      System.out.println("IOException caught: " + e.getMessage());
   }
} }
```

```
int: 5
 ints: 5, 10
 double: 3.14
Result : 6.28
 String: Hello
 boolean: true
 boolean: false
IOException caught: False condition
```

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.**

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.

NOTE!!!!!

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); }
  }
```

NOTE!!

#50

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