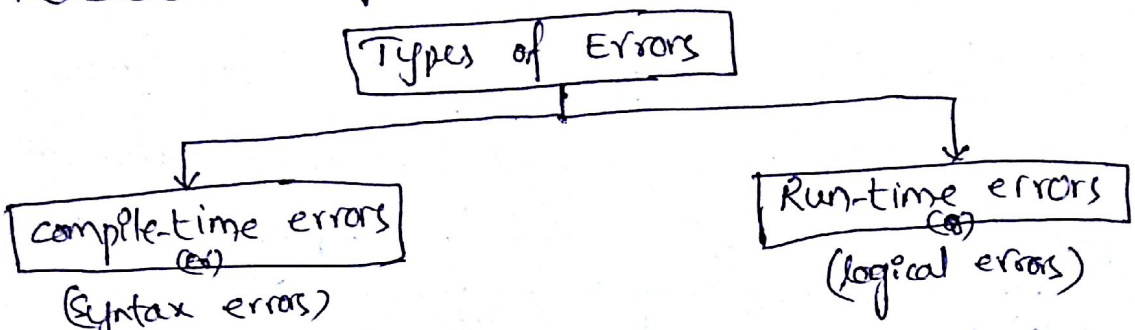
Exception Handling



→ Syntax errors are detected by the compiler during compilation time. most of the compile time errors are due to typing errors.

→ Logical errors arise because of deaviation from application logic also called as bugs. These are detected by JRE.

ex: ① Dividing an integer by zero.
② Accessing an element that is out of the bounds of an array.
③ trying to store an incompatible value in an array.
④ converting invalid string to a number
⑤ Entering of invalid input.
⑥ Attempt to open a file that does not exist.

Exception: An exception is an abnormal condition that arises in a code sequence at runtime. i.e an exception is a runtime error.

A java exception is an object that describes an exceptional (i.e 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.

Exception Handling

Java provides the capability to let the programmer handle runtime errors, this capability referred to as exception handling.

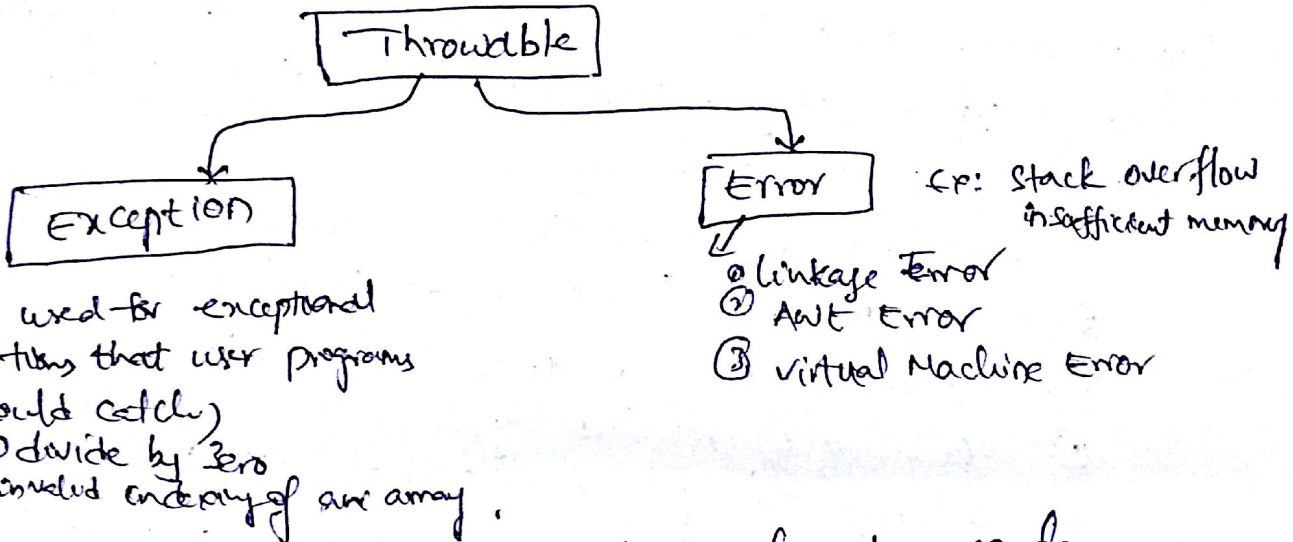
⇒ An exception is said to be thrown from the point where it occured and is said to be caught at the point to which control is transferred.

→ programs can also throw exceptions explicitly using throws stmts.

→ Every exception in java is represented by an instance of the class Throwable (or) one of its subclasses. Any exception should be handled by the default exception handler (or) compiler.

Exception types:

All exception types are subclasses of the built-in class Throwable.



Java exception handling is managed by five keywords:

try, catch, throw, throws, and finally.

General form:

```
try
{
  // block of code to monitor for errors.
}
catch (ExceptionType1 exOb)
{
  // Exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
  // Exception handler for ExceptionType2
}
// finally block of code to be executed before try block ends.
}
```

→ doubtful code (code which may raise an exception) should be written in a try block.

→ Even if there is a single stmt in try block it must be enclosed in a pair of brackets. ({ }).

→ A 'try' should be always followed by either catch or finally block.

→ when an exception is raised, then an object of that type (class) is constructed by the compiler and is thrown.

→ try can have any no of stmts that may raise different types of exceptions.

→ An exception object thrown in a try block will be handled by a 'catch' block.

→ One try can have any no of catch blocks.

→ whenever an exception is thrown, then the 'catch' blocks will be checked in order and the one matching with the exception will be executed.

→ 'catch' blocks are used for defining user-defined messages in case of exception.

→ After the catch the program continues its execution.

program stmts that to be monitor for exceptions are contained in try block. If an exception occurs, it is thrown. We have to catch this exception using 'catch'. 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 before a method returns is put in a 'finally' block.

Uncaught Exceptions

```
class Exception1
{
    P.S.V.main(String args[])
    {
        int d=0;
        int a=42/d;
        SOP(a);
    }
}
```

o/p

Java.lang.ArithmeticException:
/ by zero at
Exception1.main(Exception1.java:4)

Example

```
class Exc2
{
    static void fun()
    {
        int d=0;
        int a=10/d;
    }
    p.s.v.main(String args[])
    {
        Exc2.fun();
    }
}
```

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

java.lang.ArithmeticException : / by zero at Exc2.fun(Exc2.java:6)
at exc2.main(Exc2.java:10)

Using try and catch :

```
class Exc3
{
    p.s.v.main(String args[])
    {
        int d, a;
        try
        {
            d=0; a=42/d;
            S.o.pln("This will not be printed.");
        }
        catch(ArithmeticException e)
        {
            s.o.pln("Division by zero.");
        }
        s.o.pln("After catch statement.");
    }
}
```

op : Division by zero.
After catch statement.

Note: Here, println() inside the try block is never executed. once an exception is thrown, program control transfers out of the try block into catch block. once the catch stmts has executed, program control continues with the next line in the program following the entire try/catch mechanism.

Displaying a Description of an exception

↳ Throwable overrides the toString() method so that it returns a string containing a description of the exception.

```
class Exc4
{
    p.s.v.main(String args[])
    {
        int d, a;
        try
        {
            d=0;
            a= 45/d;
            s.o.pln("This will not be printed");
        }
        catch (Arithmetic Exception e)
        {
            s.o.pln("Exception:"+e);
        }
        s.o.pln("After the catch stmt");
    }
}
```

o/p: Exception: java.lang.ArithmeticException: / by zero..
After the catch stmt.

Multiple catch clauses

```
class Multicatch
{
    p.s.v.m(String args[])
    {
        try
        {
            int a= args.length;
            s.o.pln("a="+a);
            int b= 42/a;
            int c[]= {1};
            c[42]= 99;
        }
        catch(ArithmeticException e)
        {
            s.o.pln("Divide by 0:", +e);
        }
        catch(ArrayIndexOutofBoundsException e)
        {
            s.o.pln("ArrayIndex OOb:"+e);
        }
```

```
        S.O.println("After try/catch blocks.");
    }
}
```

Here, It gives a division by zero exception if it is started with no command line parameters, since 'a' will equal zero. If you provide a command line argument, arithmetic exception is not raised because a is not equal to zero. But, it causes ArrayIndexOutOfBounds exception, since the int array 'c' has length of 1, yet the program attempts to assign a value to c[42]

```
c:\> java Multicatch
    a=0
    Divide by 0: java.lang.ArithmeticException: / by zero
    After try/catch blocks.
c:\> java MulticatchTestArg
    a=1
    Array index oob: java.lang.ArrayIndexOutOfBoundsException.
    After try/catch blocks
```

Note: In multiple catch stmts, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch stmt that uses a superclass will catch exceptions of that type plus any of its subclasses.

```
class SuperSubCatch
{
  p.s.v.m(String args[])
  {
    try
    {
      int a=0;
      int b=42/a;
    }
    catch(Exception e)
    {
      S.O.println("Generic Exception catch");
    }
    catch(ArithmeticException e)
    {
      S.O.println("This is never reached");
    }
  }
}
```

o/p:

Error message stating that the second catch stmt is unreachable because exception has already been caught.

ArithmeticException is subclass of Exception.

To fix this problem, reverse the order of catch statements.

Nested try statements

The try stmt can be nested. i.e. a try stmt can be inside the block of another try. Each time a try stmt is entered, the context of that exception is pushed on the stack. If an inner try stmt does not have a catch handler for a particular exception, the stack is unwound and the next try stmt's catch handlers are inspected for a match. This continues until one of the catch stmt succeeds. If no catch stmt matches, the Java run-time system will handle the exception.

```
class NestTry
{
    p.s.v.m(String args[])
    {
        try
        {
            int a = args.length;
            int b = 42/a;
            S.O.pln("a = "+a);
            try
            {
                if (a==1)
                {
                    a = a/(a-a);
                }
                if (a==2)
                {
                    int c[] = {1};
                    c[42] = 99;
                }
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                S.O.pln("A.I.O.O.B.E = "+e);
            }
        }
        catch(ArithmeticException e)
        {
            S.O.pln("Divide by 0 : "+e);
        }
    }
}
```

Inner try & catch.

throw:

It is possible for your program to throw an exception explicitly, using the **throw** stmt.

Syntax:

```
throw ThrowableInstance;
```

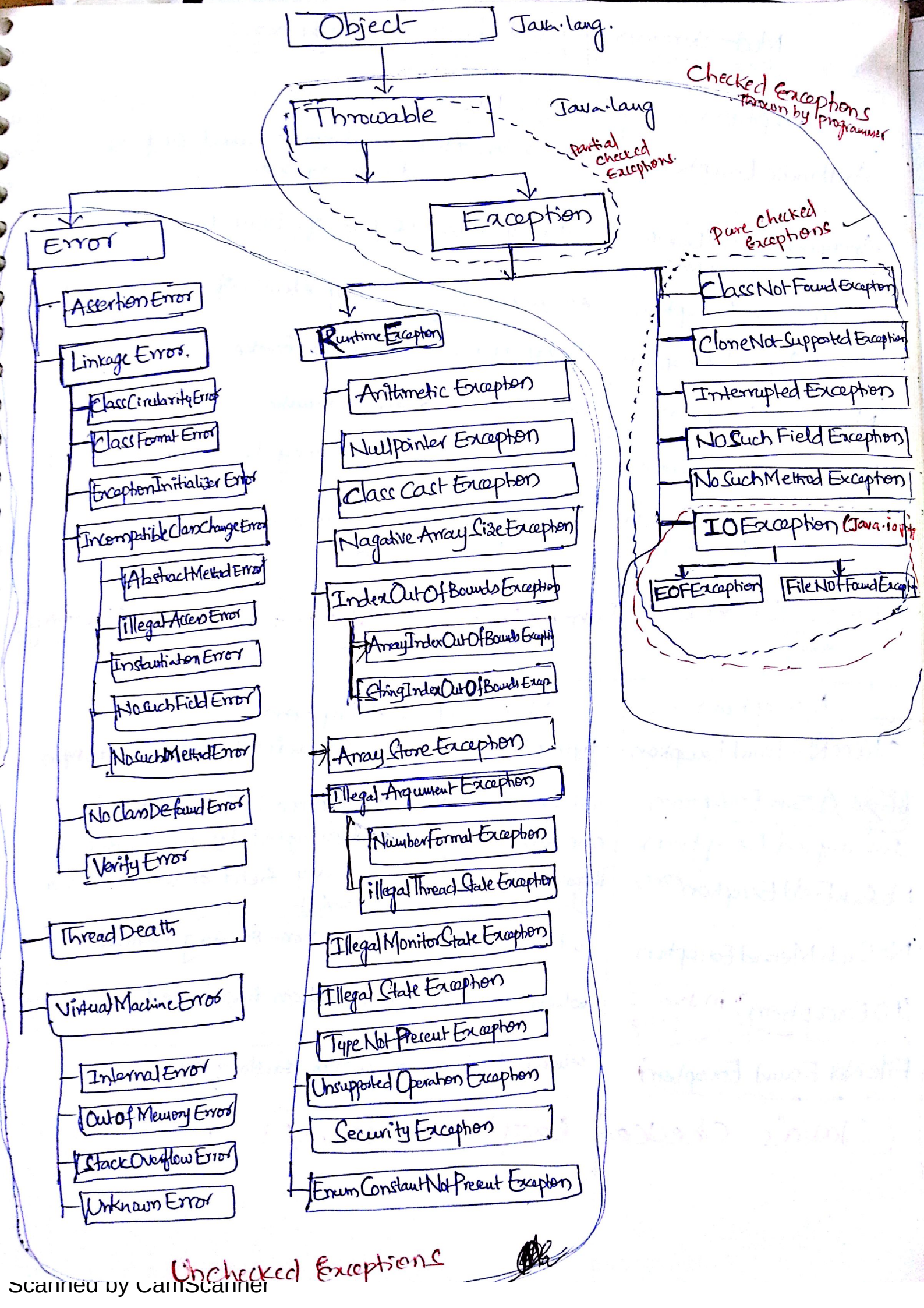
↳ must be an object of type Throwable (or) a subclass of Throwable.

There are two ways to obtain a Throwable object: Using a parameter into a catch clause, (or) creating one with the **new** operator

```
class ThrowDemo
{
    static void demoproc()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            S.O.println("caught inside demoproc");
            throw e;
        }
    }
    P.S.V.main(String args[])
    {
        try
        {
            demoproc();
        }
        catch(NullPointerException e)
        {
            S.O.println("Recaught :" + e);
        }
    }
}
```

o/p: caught inside demoproc.

Recaught : java.lang.NullPointerException: demo



throws

If a method is capable of causing an exception that it does not handle, then it is the responsibility of the calling method to handle that exception and it should declare the same using the keyword 'throws'. The calling method should place that method in a 'try' block.

General form

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

Example

```
class ThrowsDemo
{
    static void throwOne() throws IllegalAccessException
    {
        System.out.println("Inside throwOne method.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[])
    {
        try
        {
            throwOne();
        }
        catch (IllegalAccessException e)
        {
            System.out.println("caught : "+e);
        }
    }
}
```

o/p : Inside throwOne method.
caught : java.lang.IllegalAccessException : demo

finally:

'finally' creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The 'finally' block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no 'catch' stmt matches the exception. This can be useful for closing file handles and freeing up any other resources. The 'finally' clause is optional. However, each 'try' stmt requires atleast one 'catch' or a 'finally' stmt.

Example:

```
class FinallyDemo
{
    static void procA()
    {
        try
        {
            S.O.pln("Inside procAmethod");
            throw new RuntimeException("demo");
        }
        finally
        {
            S.O.pln("procA's finally block");
        }
    }
    static void procB()
    {
        try
        {
            S.O.pln("Inside procB method");
            return;
        }
        finally
        {
            S.O.pln("procB's finally block");
        }
    }
    static void procC()
    {
        try
        {
            S.O.pln("Inside procc method");
        }
        finally
        {
            S.O.pln("procc's finally block");
        }
    }
```

```
public static void main( String args[])
{
    try
    {
        procA();
    }
    catch ( Exception e)
    {
        S.o.println(" Exception caught");
    }
    procB();
    procC();
}
}
```

output: Inside procA method
procA's finally block
Exception caught
Inside procB method
procB's finally block
Inside procC method
procC's finally block.

Checked & Unchecked Exceptions

The exception classes "Error" and it's subclasses and RuntimeException and its subclasses need not be included in the throws list for need not be handled in the method of that method is raising the exception. Such exception classes are known as Unchecked exceptions, because the compiler does not check if the user has handled them.

All the subclasses of 'Exception' class, except RuntimeException are known as checked exceptions because the compiler will check if the user has handled these types of exceptions.

Eg: ClassNotFoundException, IllegalAccessException... etc.

Exception Hierarchy

Java Built-in Exceptions

- Object
  - Throwable
    - Error
      - Linkage Error
      - Virtual Machine Error
      - Awt Error
      - Internal Error
      - out of Memory Error
      - Stack Overflow Error
      - Unknown Error.
    - Exception
      - Runtime Exception ⟸ checked exception thrown by programmer
      - ClassNotFound Exception
      - cloneNotSupported Exception
      - IllegalAccessException
      - InstantiationException
      - NoSuchMethod Exception
      - Interrupted Exception
      - IO Exception
        - EOF Exception
        - FileNotFound Exception

- Runtime Exception
  - Arithmetic Exception
  - Nullpointer Exception
  - ClassCast Exception
  - NegativeArraySizeException
  - IndexOutOfBoundsException
    - ArrayIndexOutOfBoundsExcptn
    - StringIndexOutOfBoundsExcptn
  - ArrayStoreException
  - IllegalArgumentException
    - NoFormatException
    - IllegalThreadStateException
  - SecurityException.
  - IllegalMonitorStateException
  - IllegalStateException
  - UnsupportedOperationException
  - EnumConstantNotPresentException.

unchecked exceptions thrown by JVM.

User-defined Exceptions

If a user wants to create a user-defined exception, then that class has to be derived from the superclass "Exception"

example: (User-defined)

```
class MyException extends Exception
{
    MyException()
    {
        S.o.pln("Invalid");
    }
}
class Test
{
    static void checkTest(int x)
    {
        try
        {
            if(x<18)
            {
                throw new MyException();
            }else S.o.pln("No exception");
        }
        catch(MyException e)
        {
            S.o.pln("caught:"+e);
        }
    }
    p.s.v.main(String args[])
    {
        checkTest(15);
        checkTest(25);
    }
}
```

op:

Multithreading

→ A multithreaded program contains two (or) more parts that can run concurrently. Each part of such a program is called a thread and each thread defines a separate path of execution. Thus, multi-threading is a specialized form of multitasking

Multitasking
- ① process-based
- ② Thread-based

} Differences b/w Multi-process and Multi-threading

A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two (or) more programs concurrently. It enables you to run Java compiler at the same time that you are using a text editor.

In thread-based multitasking, a single program can perform two (or) more tasks simultaneously. A text-editor can format text at the same time that it is printing, two actions are being performed by two separate threads.

Multitasking threads require less overhead than multitasking processes. processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context-switching from one process to another is costly. processes are independent.

Threads are lightweight. They share the same address space and cooperatively share the same heavyweight process. Inter Thread communication is inexpensive, and context switching from one thread to the next is low cost. Threads are dependent on the main thread.

Multithreading enables you to write very efficient programs that make maximum use of CPU, because idle time can be kept to a minimum.

If one process gets blocked, other process are not affected.

If the parent thread terminates, the child thread also gets terminated

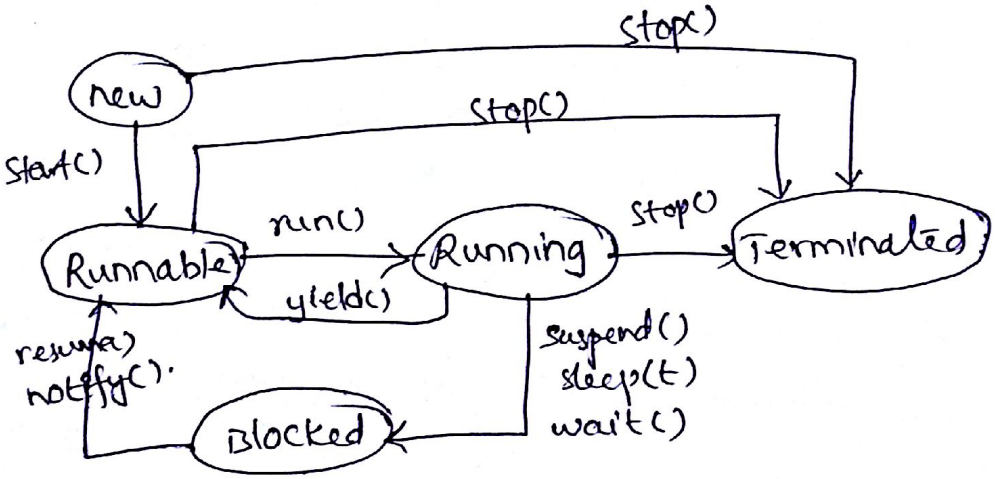
A Thread is called as a light-weight process because it shares the memory assigned to a process.

A process is called heavy-weight process, because every process has its own memory space.

P_1, P_2 & P_3 are processes

T_1, T_2 & T_3 are Threads

Thread Life cycle :



① **new born state** :
when a thread object is created, then the thread is said to be in new born state.

Runnable state
when the thread object invokes the start() method, then the thread is said to be in Runnable state i.e. it is ready for execution and is waiting in a queue for the processor.

Running state
The thread scheduler uses a scheduling algorithm (priority, roundrobin) and picks one from the ready queue and gives it to the processor. Then, the thread is said to be in

running state and the method that is being executed is run().
The task we want our thread to perform should be written in run() method. run() method is automatically invoked by start() method.

Blocked state:
A thread is said to be in block state in three cases:

① Invoking a suspend() method: The thread will stop running and will start again, only when resume() method is invoked.

② Invoking a sleep(t) method: The thread sleeps for the amount of time mentioned as the parameter. It wakes up automatically after the time is elapsed.

③ wait() method: When two or more threads communicate with each other, to achieve synchronization of data, threads will be made to wait (may be for an event, resource or...) for another thread to notify() them.

Terminated state:
A thread is said to be in this state when it has finished execution of run() method. If the user wants to kill a thread, then he can invoke stop() method.

The Thread class and the Runnable Interface:
Java's multithreading system built upon the Thread class, its methods and interface Runnable.
To create a new thread, the program will either extend Thread or implement the Runnable interface.
The Thread class defines several methods that help to manage threads:

Methods

① public final void setName (String name) → to set a name to thread
② public final String getName() :- to obtain a thread's name
③ public void start() :- starts a thread by calling its run method.
④ public static Thread currentThread() :→ obtaining reference of current thread
⑤ public void run() → Entry point for the thread
⑥ public static void sleep(long ms) → Suspend a thread for a period of time.
⑦ public void join() - wait for a thread to terminate
⑧ public int getPriority() : to obtain a thread's priority
⑨ public void setPriority (int p): to change a thread's priority
⑩ public boolean isAlive(): determine if a thread is still running.
⑪ public void resume():
⑫ public void stop();
⑬ public void suspend():
⑭ public void yield()

The Main Thread

When a java program starts up, one thread begins running immediately It is called the main thread of your program, because it is the one that is executed when your program begins. It is important for
① It is the thread from which other child threads will be spawned.
② It must be the last thread to finish execution.

Example:

```
class CurrentThreadDemo
{
    p.s.v. main(String args[])
    {
        Thread t = Thread.currentThread();
        S.o.pln("Current thread: "+t);
        t.setName("My Thread");
        S.o.pln("After name change: "+t);
```

```
try
{
  for(int n=5; n>0; n--)
  {
    System.out.println(n);
    Thread.sleep(1000);
  }
}
catch (InterruptedException e)
{
  Sys.o.pln("Main Thread Interrupted");
}
}
```

→ milliseconds of type long

o/p: Current Thread : Thread[main, 5, main]

(name of thread, priority (default), group of thread.)

After name change : Thread[MyThread, 5, main]

5
4
3
2
1

Creating a Thread
- ① Extended by a Thread class
- ② Implement the Runnable Interface

① Extending Thread

```
class Mythread extends Thread
{
  MyThread()
  {
    start();
  }
  public void run()
  {
    try
    {
      for(int i=5; i>0; i--)
      { s.o.pln("child thread:"+i);
        Thread.sleep(500);
      }}
    catch (InterruptedException e)
    {
      s.o.pln("child Interrupted");
    }
  }
}
```

```
class ThreadDemo1
{
    p.s.v. main (String args[])
    {
        new NewThread(); // create a new thread.
        try
        {
            for (i=5; i>0; i--)
            {
                s.o.pln ("Main thread : "+i);
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                s.o.pln ("Main Thread interrupted.");
            }
        }
        s.o.pln ("Main thread exiting.");
    }
}
```

→ To create a thread is to create a new class that extends Thread, and then to create instance of that class. The extending class must override the run() method, which is the entry point for the new thread. It must also call start() to begin the execution of the new thread.

In the above program, new MyThread() creates thread object. The control will goto MyThread constructor, in which we have invoked start(). The start() method implicitly invokes run(). and displays the stmt in the for loop.

When start() invokes run(), new thread starts executing simultaneously with the main thread. Here, the thread object is created in the main thread. So, main thread now is the parent thread which spawns the child thread. The sleep() raises an interrupted exception by default. So, it has to always in a try block. To ensure that the main thread gets terminated, after the child threads, the sleep time in main thread should be more than the sleep time in the child threads.

② Implementing Runnable Interface

Runnable abstracts a unit of executable code. To implement Runnable, a class need only implement a single method called run(). Inside run(), you will define the code that constitutes the new thread. run() establishes the entry point for another concurrent thread of execution with in the program.

After creating a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors.

```
Thread( Runnable threadOb, String threadName);
```

After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. start() executes a call to run().

```
void start();
```

Example

```
class MyThread2 implements Runnable
{
    Thread t;
    MyThread2()
    {
        t = new Thread(this, "Child Thread");
        S.O.println("child thread : "+t);
        t.start(); // Start the thread.
    }
    public void run()
    {
        try
        {
            for(int i=5; i>0; i--)
            {
                S.O.println("Child Thread : "+i);
                Thread.sleep(500);
            }
        }
        catch(InterruptedException e)
        {
            S.O.println("Child Interrupted");
        }
        S.O.println("Exiting Child Thread");
    }
}
```

```
class ThreadDemo
{
    p.s.v.main(String args[])
    {
        new MyThread2();
        try
        {
            for(int j=5; j>=1; j--)
            {
                s.o.pln("MainThread: "+j);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
            s.o.pln("MainThread interrupted");
        }
        s.o.pln("MainThread exiting");
    }
}
```

A new 'Thread' object is created by the stmt

t = new Thread(this, "ChildThread");

Next, start() is called to start the thread of execution beginning at the run() method. Here ChildThread's for loop to begin. After calling start(), MyThread2 constructor returns to main(). When the main() thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.

output

```
ChildThread: Thread[ChildThread, 5, main]
main Thread: 5
Child Thread: 5
ChildThread: 4
MainThread: 4
Child Thread: 3
ChildThread: 2
MainThread: 3
ChildThread: 1
Exiting ChildThread
MainThread: 2
MainThread: 1
MainThread exiting
```

creating multiple Threads

```
class NewThread implements Runnable
{
    String name;
    Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        S.o.pln("NewThread :" + t);
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i=5; i>0; i--)
            {
                S.o.pln(name + ": " + i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
            S.o.pln(name + " Interrupted");
        }
        S.o.pln(name + "existing");
    }
}
class MultiThreadDemo
{
    P.S.V.main(String args[])
    {
        new NewThread("one");
        new NewThread("two");
        new NewThread("three");
        try
        {
            Thread.sleep(10000);
        }
        catch(InterruptedException e)
        {
            S.o.pln("Main Thread Interrupted");
        }
        S.o.pln("Main Thread existing");
    }
}
```

output

```
NewThread : Thread[one, 5, main]
NewThread : Thread[two, 5, main]
NewThread : Thread[three, 5, main]
one : 5
two : 5
three : 5
  :
  :
one : 1
two : 1
three : 1
one existing
two existing
three existing
Main Thread existing
```

Q) Write a java program to create multiple threads (3 threads) which displays:

① "Good Morning" for every 1 sec
② "welcome" for every 2 secs
③ "Hi" for every 3 secs.

isAlive() and join():

Two ways exist to determine whether a thread has finished.

① isAlive(): It returns 'true' if the thread upon which it is called is still running. It returns 'false' otherwise.

```
final boolean isAlive()
```

② join():

A method which is used to wait for a thread to finish is join().

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates.

```
class NewThread implements Runnable
{
    String name;
    Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("NewThread : "+t);
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i=5; i>0; i--)
            {
                System.out.println(name+":"+i);
            }
            Thread.sleep(1000);
```

```
        catch (InterruptedException e)
        {
            S.O.pln( name + " interrupted ");
        }
        S.O.pln( name + " exiting.");
    }
}

class AliveJoinDemo
{
    p. s. v. main ( String args[])
    {
        NewThread ob1 = new NewThread ("one");
        NewThread ob2 = new NewThread ("two");
        NewThread ob3 = new NewThread ("three");

        S.O.pln("Thread one is alive : " + ob1.isAlive());
        S.O.pln("Thread two is alive : " + ob2.isAlive());
        S.O.pln("Thread three is alive: " + ob3.isAlive());

        try
        {
            S.O.pln("waiting for threads to finish");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch(InterruptedException e)
        {
            S.O.pln("Main thread Interrupted");
        }
        S.O.pln("Thread one is alive: " + ob1.isAlive());
        S.O.pln("Thread two is alive: " + ob2.isAlive());
        S.O.pln("Three three is alive: " + ob3.isAlive());
        S.O.pln("main thread existing");
    }
}
```

Note:
→ Generally, the child-thread class and main-thread class are written by two different users, so main() thread user will not be able to know the sleep() time of the child thread. To overcome this problem and to ensure termination of main thread last, join() method was used.

In the above program, the stmts ob1.t.join(), ob2.t.join() and ob3.t.join() were written in main thread. That means, the main thread waits till the child-threads of ob1, ob2 and ob3 gets terminated and ~~join~~ joins the main thread. i.e main-thread will not be terminated till the objects ob1, ob2 & ob3 terminates.

Thread priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. Higher-priority threads get more cpu time than lower-priority threads. A higher-priority thread can also preempt a lower-priority one. Threads of equal priority should get equal access to the CPU.

setpriority() : to set a thread's priority.

```
final void setPriority (int level)
```

The value of level must be within the range MIN-PRIORITY and MAX-PRIORITY. [1 and 10].

To return a thread to default priority, specify NORM-PRIORITY, which is currently 5.

getpriority() : to obtain the current priority of thread

```
final int getPriority();
```

Example:

```
class clicker implements Runnable
{
    int click=0;
    Thread t;
    private volatile boolean running=true;
    public clicker(int p)
    {
        t=new Thread(this);
        t.setPriority(p);
    }
```

```
public void run()
{
    while (running)
    {
        click++;
    }
}
public void stop()
{
    running = false;
}
public void start()
{
    t.start();
}
}
class HiLoPri
{
    p. s. v. main( String args[])
    {
    Thread.currentThread().setPriority (Thread.MAX_PRIORITY);
        clicker hi = new clicker (Thread.NORM_PRIORITY+2);
        clicker lo = new clicker (Thread.NORM_PRIORITY-2);

          lo.start();
          hi.start();
        try
        {
          Thread.sleep( 10000);
        }
        catch( InterruptedException e)
        { s.o.pln("MainThread interrupted");
        }
        lo.stop();
        hi.stop();
        try
        { hi.t.join(); lo.t.join();
        }
        catch ( Interrupted Exception e)
        { s.o.pln("Interrupted Exception caught");
        }
      s.o.pln (" low priority thread: " + lo.click);
       s.o.p.ln(" high priority thread: " + hi.click);
    }
}
```

Synchronization

When two (or) more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. It is achieved by a process known as Synchronization.

Two types

1. 'synchronized' keyword
2. synchronized (object) { }.

The method to be shared is declared as 'synchronized', which will stop other threads to access that method when already a thread is using it. Generally these shared methods are written by third party. So, instead of declaring the method as synchronized the method call is placed in synchronized block.

Example:

```
class Callme
{
    void call (String msg)
    {
        System.out.println("[" + msg);
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller (Callme targ, String s)
    {
        target = targ;
        msg = s;
        t = new Thread (this);
        t.start();
    }
```

```
        public void run()
        {
            target.call(msg);
        }
    }
```

Unsynchronized block

```
class SynchDemo
{
    P.S.V.main(String args[])
    {
        Callme target = new Callme();
        ~~callerObject~~
        caller ob1 = new caller(target, "Hello");
        caller ob2 = new caller(target, "Synchronized");
        caller ob3 = new caller(target, "World");
        try
        {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
    }
}
```

output:

```
Hello [Synchronized [World]
    ]
    ]
```

Here, three threads are allowed to access call() method without synchronization which is known as race condition. This problem can be solved by restricting a thread to access the call() method when already another thread is accessing it.
In Java, this is achieved by declaring such non-shareable methods as synchronized i.e

```
Synchronized void call(String s)
{
    =
}
```

Allowing only one process/thread to access the critical section and stopping others, is known as mutual exclusion.

Unshareable code of data is known as Critical section.

Using the keyword 'synchronized' is known as monitor (or synchronized tool)

Usage of the keyword 'synchronized' solves the problems of synchronization. But, in real world applications, the non-sharable methods may or may not be synchronized.

To synchronize the method call(), using a synchronized block the run() method in the above program has to be modified as:

```
public void run()
{
    synchronized (target);   → synchronized block.
     target.call(msg);
}
```

Now, output is

```
[Hello]
[synchronized]
 [world]
```

Interthread communication

The classical queuing problem (producer-consumer problem) where one thread is producing some data and another is consuming it.

To interprocess communication, java provides the methods like wait(), notify() and notifyAll().

Declaration

```
final void wait() throws Interrupted exception
final void notify()
final void notifyAll()
```

wait(): tells the calling thread to give up and go to sleep until some other thread enters the same and calls notify()

notify(): wakes up the first thread that called wait() on the same object.

notifyAll(): wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

// An incorrect implementation of a producer and consumer Pwk

```
class Q
{
   int n;
   synchronized int get()
   {
      S.o.pln("Got: "+n);
      return n;
   }
   synchronized void put(int n)
   {
      this.n = n;
      S.o.pln("put: "+n);
   }
}
```

class Q, the queue that you are trying to synchronize

```
class producer implements Runnable
{
   Q q;
   producer(Q q)
   {
      this.q = q;
      new Thread(this, "producer").start();
   }
   public void run()
   {
      int i=0;
      while(true)
      { q.put(i++); }
   }
}
```

producer thread — the Thread object producing queue entries

```
class consumer implements Runnable
{
    Q q;
    consumer(Q q)
    {
        this.q = q;
        new Thread(this, "consumer").start();
    }
    public void run()
    {
        while(true)
        {
            q.get();
        }
    }
}

class pcdemo
{
    p.s.v.main(String args[])
    {
        Q q = new Q();
        new producer(q);
        new consumer(q);
        S.o.pln("press control.c to stop");
    }
}
```

consumer thread
It is for consuming queue entries.

o/p:

```
put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
put: 2
Got: 3
put: 4
put: 5
put: 6
put: 7
Got: 7
```

// Correct implementation of a producer and consumer

```
class Q
{
    int n;
    boolean valueset = false;
    synchronized int get()
    {
        if( !valueset)
        {
            try
            {
                wait();
            }
            catch( InterruptedException e)
            {
                System.out.println(" Exception caught ");
            }
        }
        System.out.println("Got: "+n);
        valueset = false;
        notify();
        return n;
    }
    synchronized void put (int n)
    {
        if (valueset)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
                System.out.println ("Exception caught ");
            }
        }
        this.n = n;
        valueset = true;
        System.out.println(" put : "+n);
        notify();
    }
}
```

```
class producer Implements Runnable
{
    Q q;
    producer(Q q)
    { this.q=q;
      new Thread(this, "producer").start();
    }
    public void run()
    { int i=0;
      while(true)
      { q.put(i++);
      }
    }
}
class Consumer implements Runnable
{
    Q q;
    Consumer(Q q)
    { this.q=q;
      new Thread(this, "consumer").start();
    }
    public void run()
    { while(true)
      { q.get();
      }
    }
}
class PCDemo
{
    P.S.V.main(String args[])
    { Q q = new Q();
      new producer(q);
      new Consumer(q);
      S.O.println("press control-c to stop.");
    }
}
```

Inside get(), wait() is called. It causes its execution to suspended until the Producer notifies you that some data is ready. When it happens, execution inside get() resumes. After the data has been obtained, get() calls notify(). It tells producer that it is okay to put more data in the queue. Inside put(), wait() suspends execution until the consumer has removed the item from the queue. When the execution resumes, the next item is put in the queue, and notify(). This tells the consumer that it should now remove it.

o/p:

```
put: 1
Got: 1
put: 2
Got: 2
put: 3
Got: 3
put: 4
Got: 4
put: 5
Got: 5
```