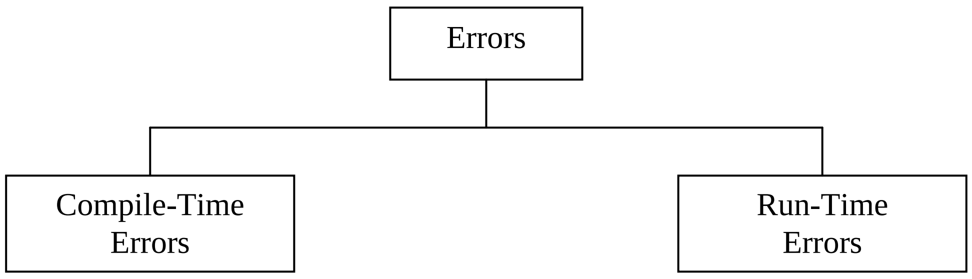
Errors

- Errors are the wrongs that can make a program go wrong.
- An error may lead to:
    - Produce an incorrect output
    - Terminate the execution of program
    - Cause the system to crash

Types of Errors



Compile-Time Errors

- All syntax errors that were detected and displayed by the java compiler.
- If a program has compile time error than *.class* file will not be created.
- It is necessary to fix the compile time errors for executing a program.
- Java compiler tell us where (Line number) the errors are in program.
- Most of the compile time errors are due to typing mistakes.
- Generally the earliest errors are the major source of problem.

Example:
- Missing semicolons
- Missing brackets in class or methods
- Misspelling of identifiers or keywords
- Use of undeclared variable
- Use of == in place of = operator
- Directory Path Error (javac : command not found)

Run-Time Errors

- Sometimes a program may compile successfully creating .class file but may not execute properly.
- Java compiler generates an error message and aborts the program.
- These errors may produce wrong results due to wrong logic or terminate the program or even cause the system to crash.

Example:
- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible type
- Attempting to use a negative size for an array

Exception

- An exception is condition that is caused by a run time error in the program.
- If exception is not handled properly then:
    - The interpreter will create an exception object and throws it
    - Display an error message
    - Terminate the program

Example-1: If exception is not handled.

three.java

```
public class three
{
        public static void main(String args[])
        {
                int data=50/0;
                System.out.println("Remaining Code...");
        }
}
```

Output:

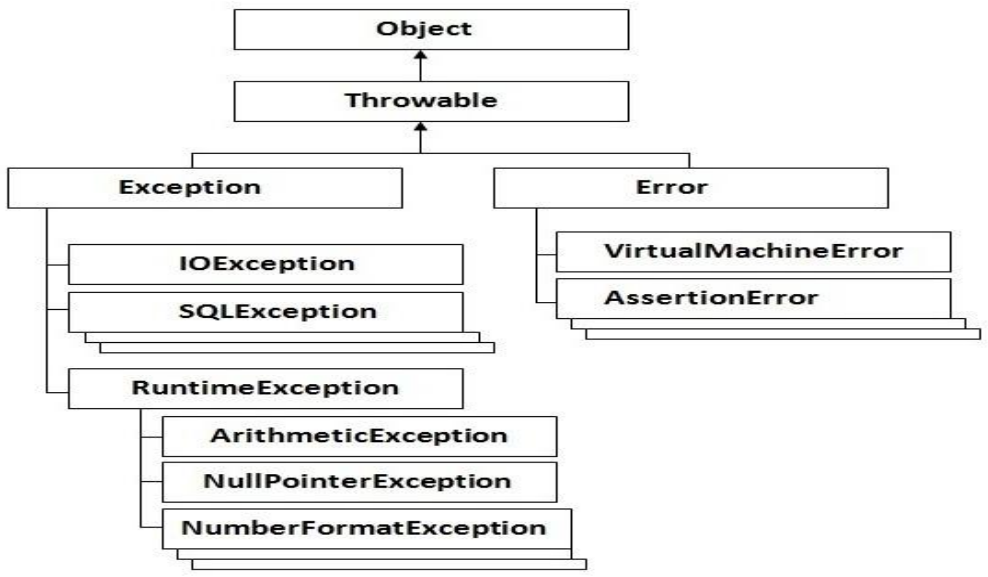
```
Exception in thread “main” java.lang.ArithmeticException: / by Zero
        at three.main (three.java:5)
```

Exception Handling

- If we want our program to continue with execution:
    - We should catch the exception object thrown by error condition
    - Display an appropriate message for taking corrective actions.

  This task is known as exception handling.
- Exception Handling is a mechanism to handle runtime errors.
- It consists of basically two segments
    - To detect errors and to throws exceptions
    - To catch exceptions and to take appropriate actions

Hierarchy of Java Exception classes



Types of Exception

1. **Checked Exception**
   - Checked Exception is direct sub-class of Exception class.
   - Checked exceptions are checked at compile-time.
   - Example- IOException, SQLException, etc.
2. **Unchecked Exception**
   - Unchecked Exception is direct sub-class of RuntimeException class.
   - Unchecked exceptions are checked at runtime.
   - Example- ArithmeticException, NullPointerException, etc.
3. **Error**
   - Error is irrecoverable
   - Example- OutOfMemoryError, VirtualMachineError, etc.

Examples of Unchecked Exceptions

1. **ArithmeticException**
   - If we divide any integer number by zero.
     Example:
     int a=50/0;
2. **NullPointerException**
   - If we have null value in any variable and performing any operation on it.
     Example:
     String s=null;
     System.out.println(s.length());
3. **NumberFormatException**
   - The wrong formatting of any value.
     Example:
     String s="abc";
     int i=Integer.parseInt(s);
4. **ArrayIndexOutOfBoundsException**
   - If we are Accessing/inserting any value in the wrong index.
     Example:
     int a[]=new int[5];
     a[10]=50;

Exception Handling Keywords

- try
- catch
- finally
- throw
- throws

Try block

- It is used to enclose the code that might throw an exception.
- It must be used within the method.
- It must be followed by either catch or finally block.

Syntax of try-catch

```
try
{
        statements; //generates an exception
}
catch (exception-type e)
{
        statements; //processes the exception
}
```

Example:

four.java

```
public class four
{
        public static void main(String args[])
        {
                try
                {
                        int data=50/0;
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e);
                }
                System.out.println("Remaining Code...");
        }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
Remaining Code . . .
```

Multi-catch

- At a time only one Exception is occurred and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general.
  Example:
  catch for ‘*ArithmeticException*’ must come before catch for ‘*Exception*’**.**

Syntax of Multi-catch

```
try
{
        statements; //generates an exception
}
catch (exception-type-1 e)
{
        statements; //processes the exception
}
```

```
catch (exception-type-2 e)
{
        statements; //processes the exception
}
```

Example

five.java

```
public class five
{
        public static void main(String args[])
        {
                try
                {
                        int a[]=new int[5];
                        a[5]=30/0;
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println(e);
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
                System.out.println("Remaining Code...");
        }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
Remaining Code . . .
```

Nested Try Block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax of nested try block

```
try
{
        statements;
        try
        {
                statements;
        }
        catch(Exception e)
        {
                statements;
        }
}
catch(Exception e)
{
        statement;
}
```

Example:

six.java

```
class six
{
        public static void main(String args[])
        {
                try
                {
                        try
                        {
                                int a =30/0;
                        }
                        catch(ArithmeticException e)
                        {
                                System.out.println(e);
                        }
                        try
                        {
                                int b[]=new int[5];
                                b[5]=4;
                        }
                        catch(ArrayIndexOutOfBoundsException e)
                        {
                                System.out.println(e);
                        }
                        System.out.println("Other Statement");
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
                System.out.println("Remaining Code...");
        }
}
```

Output

```
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
Other Statement
Remaining Code . . .
```

Finally Block

- It is always executed whether exception is handled or not.

- It follows try or catch block.
- It is used *to execute important code* such as closing connection, closing file etc.
- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block will not be executed
  - if program exits by calling System.exit()
  - if program exits by causing a fatal error
- The different cases where finally block can be used are:
  - *Case 1*: Where exception doesn't occur
  - *Case 2*: Where exception occurs and not handled
  - *Case 3*: Where exception occurs and handled

Example: Where exception doesn't occur (Case-1)

seven.java

```
class seven
{
	public static void main(String args[])
	{
		try
		{
			int a=25/5;
			System.out.println(a);
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
		finally
		{
			System.out.println("Finally block");
		}
		System.out.println("Remaining Code...");
	}
}
```

Output:

```
5
Finally block
Remaining Code . . .
```

Example: Where exception occurs and not handled (Case-2)

eight.java

```
class eight
{
	public static void main(String args[])
	{
		try
		{
			int a=25/0;
			System.out.println(a);
		}
		catch(NullPointerException e)
```

```
                {
                        System.out.println(e);
                }
                finally
                {
                        System.out.println("Finally block");
                }
                System.out.println("Remaining Code...");
        }
}
```

Output:

```
Finally block
Exception in thread “main” java.lang.ArithmeticException: / by zero
        at eight.main(eight.java:7)
```

Example: Where exception occurs and handled (Case-3)

nine.java

```
class nine
{
        public static void main(String args[])
        {
                try
                {
                        int a=25/0;
                        System.out.println(a);
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e);
                }
                finally
                {
                        System.out.println("Finally block");
                }
                System.out.println("Remaining Code...");
        }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
Finally block
Remaining Code . . .
```

throw keyword

- It is used to explicitly throw an exception.
- With *throw* keyword we can throw either checked or uncheked exception.
- It is mainly used to throw custom exception.

Syntax:

```
throw exception;
```

Example:
throw new ArithmeticException("Not eligible");

Example Program:

ten.java

```
class ten
{
	void verify(int age)
	{
		if(age<18)
			throw new ArithmeticException("Not eligible");
		else
			System.out.println("Eligible");
	}
	public static void main(String args[])
	{
		ten obj1=new ten();
		obj1.verify(15);
		System.out.println("Remaining Code...");
	}
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Not eligible
	at ten.verify(ten.java:6)
	at ten.main(ten.java:13)
```

Exception propagation

- An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.
- By default Unchecked Exceptions are forwarded in calling chain.
- By default, Checked Exceptions are not forwarded in calling chain.

Example Program: (Unchecked Exception are forwarded in calling chain)

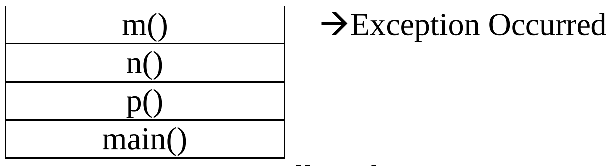
thirteen.java

```
class thirteen
{
	void m()
	{
		int data=50/0;
	}
	void n()
	{
		m();
	}
	void p()
	{
		try
```

```
                {
                        n();
                }
                catch(Exception e)
                {
                        System.out.println("Exception handled");
                }
        }
        public static void main(String args[])
        {
                thirteen obj=new thirteen();
                obj.p();
                System.out.println("Remaining Code...");
        }
}
```

Output:

```
Exception handled
Remaining Code...
```



Call stack

In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method ,p() method, n() method or m() method.

Example Program: (Checked Exception are not forwarded in calling chain)

<u>fourteen.java</u>

```
class fourteen
{
        void m()
        {
                throw new java.io.IOException("Error");
        }
        void n()
        {
                m();
        }
        void p()
        {
                try
                {
                        n();
                }
```

```
		catch(Exception e)
		{
			System.out.println("Exception handeled");
		}
	}
	public static void main(String args[])
	{
		fourteen obj=new fourteen();
		obj.p();
		System.out.println("Remaining Code");
	}
}
```

Output: *Compile Time Error*

```
fourteen.java:5: error: unreported exception IOException; must be caught or declared to be
thrown
		throw new java.io.IOException("Error");
1 error
```

throws Keyword

- It is used to declare checked exception, because:
    - **Unchecked Exception:** Under programmer control so he can correct his code.
    - **Error:** Beyond programmer control e.g. we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.
- Checked Exception can also be propagated (forwarded in call stack).

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax:

```
return_type method_name() throws exception_class_name
{
    //method code
}
```

Example Program: (Checked Exception are forwarded in calling chain)

eleven.java

```
import java.io.*;
class eleven
{
	void m()throws IOException
	{
		throw new IOException("Error");
	}
	void n()throws IOException
	{
		m();
	}
	void p()
	{
		try
		{
			n();
		}
		catch(Exception e)
		{
			System.out.println("Exception handled");
		}
	}
	public static void main(String args[])
	{
		eleven obj=new eleven();
		obj.p();
		System.out.println("Remaining Code...");
	}
}
```

Output:

```
Exception handled
Remaining Code...
```

Note: *If we are calling a method that declares an exception, we must either caught or declare the exception.*

Difference Between 'throw' and 'throws'

Throw	throws
"*throw*" keyword is used to explicitly throw an exception	"*throws"* keyword is used to declare an exception
Checked exception cannot be propagated using throw only	Checked exception can be propagated with throws
It is followed by an instance	It is followed by class
It is used within the method	It used with the method signature
We cannot throw multiple exceptions	We can declare multiple exceptions

Example
void m()
{
throw new ArithmeticException("sorry");
}
	Example
	void m()throws ArithmeticException
	{
	----------- 	
	}

Debugging

Programming is a complex process and it is done by human beings, so errors may often occur. Programming errors are called '***bugs'*** and the process of tracking them down and correcting them are called '***debugging'.*** Debugging is also known as '***debug'***. In other words Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

The debugging process can be made easier by using strategies such as:

- Unit testing
- Code reviews
- Pair programming

Questions asked in semester paper

Question-What are Exceptions and how they are handled in java? Explain the keywords try, catch, throw, throws and finally with example?

[2017-2018]

Question-What is an exception? How the exception is handled in java? Differentiate between throw and throws.

[2015-2016]

Question-What are exceptions and how they are handled? Explain with an example. How we define a try and catch block? Is it essential to catch all types of exceptions?

[2011-2012]

Question- Write short notes on Error Handling and Debugging.

[2011-2012]

Question- How exception handling is done in Java? Explain with suitable examples.

[2007-2008]

Question- How error handling and debugging are done in Java? Explain with suitable examples.

[2006-2007]

Question- What do you mean by Exception? Also write a program to handle exception in Java.

[2002-2003]