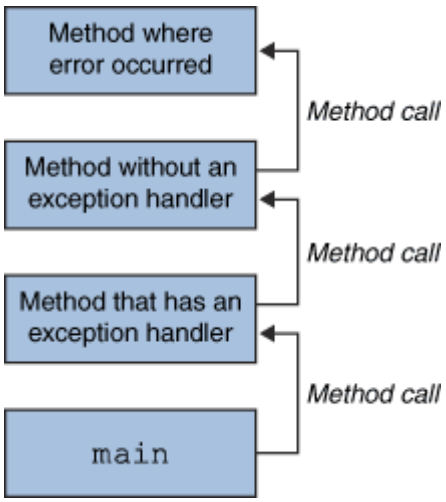
Exception Handling

- An exception is an abnormal condition that arises in a code sequence at run time.
- In other words, an exception is a runtime error.

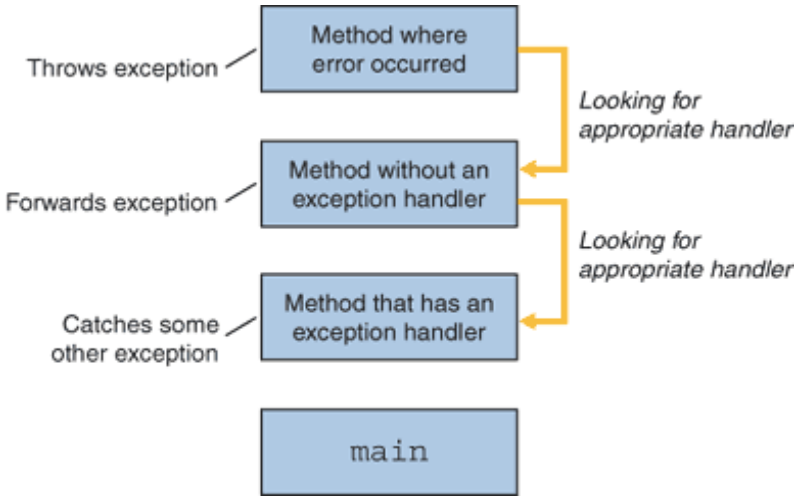
- When an error occurs within a method, the method creates an object and hands it off to the runtime system.
- The object, called an **exception object**, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called **throwing an exception**.

- After a method throws an exception, the runtime system attempts to find something to handle it.
- The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.
- The list of methods is known as the **call stack** (see the next figure).



- The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called **an exception handler**.
- The search begins with the method in which the error occurred and **proceeds through the call stack in the reverse order that the methods were called**.
- When an appropriate handler is found, the run-time system passes the exception to the handler.
- An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.
- The exception handler chosen is said to catch the exception.

- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



- Using exceptions to manage errors has some advantages over traditional error-management techniques.

Advantages of Exceptions

Advantage 1: Separating Error-Handling Code from "Regular" Code

- Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program.
- In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code.
- For example, consider the following pseudocode method that reads an entire file into memory:

```
readFile {
   open the file;
   determine its size;
   allocate that much memory;
   read the file into memory;
   close the file;
}
```

At first glance, this function seems simple enough, but it ignores all these potential errors.

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

To handle these cases, the readFile function must have more code to do error detection, reporting, and handling. The function might look like this:

```
errorCodeType readFile {
        initialize errorCode = 0;
        open the file;
        if (theFileIsOpen) {
                determine the length of the file;
                if (gotTheFileLength) {
                        allocate that much memory;
                        if (gotEnoughMemory) {
                                read the file into memory;
                                if (readFailed) {
                                        errorCode = -1;
                                }
                        } else {
                                errorCode = -2;
                        }
                } else {
                        errorCode = -3;
                }
                close the file;
                if (theFileDidntClose && errorCode == 0) {
                        errorCode = -4;
                } else {
                        errorCode = errorCode and -4;
                }
        } else {
                errorCode = -5;
        }
        return errorCode;
}
```

There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter.

- Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere.
- If the readFile function used exceptions instead of traditional error-management techniques, it would look more like this:

```
readFile {
        try {
                open the file;
                determine its size;
                 allocate that much memory;
                read the file into memory;
                close the file;
        } catch (fileOpenFailed) {
                doSomething;
        } catch (sizeDeterminationFailed) {
                doSomething;
        } catch (memoryAllocationFailed) {
                doSomething;
        } catch (readFailed) {
                doSomething;
        } catch (fileCloseFailed) {
                doSomething;
        }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

Advantage 2: Propagating Errors Up the Call Stack

- A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods.
- Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program:
- method1 calls method2, which calls method3, which finally calls readFile:

method1 {
call method2;
}
	method2 {
	call method3;
	}
		method3 {
		call readFile;
		}

Advantage 3: Grouping and Differentiating Error Types

- Because all exceptions thrown within a program are objects, grouping or categorizing of exceptions is a natural outcome of the class hierarchy.
- An example of a group of related exception classes in the Java platform are those defined in java.io: IOException and its descendants.
- IOException is the most general and represents any type of error that can occur when performing I/O. Its descendants represent more specific errors.
- For example, FileNotFoundException means that a file could not be located on disk.

- A method can write specific handlers that can handle a very specific exception.
- The FileNotFoundException class has no descendants, so the following handler can handle only one type of exception:

```
catch (FileNotFoundException e) {
   ...
}
```

- A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement.
- For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an IOException argument:

```
catch (IOException e) {
   ...
}
```

This handler will catch all I/O exceptions, including FileNotFoundException, EOFException, and so on.

You can find the details on what occurred by querying the argument passed to the exception handler. For example, to print the stack trace:

```
catch (IOException e) {
    e.printStackTrace();                // output goes to Sytem.err
    e.printStackTrace(System.out);  // send trace to stdout
}
```

You can set up an exception handler that handles any Exception with this handler:

```
catch (Exception e) {    // a (too) general exception handler
   ...
}
```

- The Exception class is close to the top of the Throwable class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and exit.

- However, in most situations, you want exception handlers to be as specific as possible. The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility.

- Exception handlers that are too general can make code more error prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

Types of Exception

1. **Checked Exception**
   - Exception that a well-wriiten program should anticipate and recover from it.
   - The exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself.
   - These are called **checked exceptions**.

No.	Exception	Meaning
1	ClassNotFoundException	Class not found.
2	CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
3	IllegalAccessException	Access to a class is denied.
4	InstantiationException	Attempt to create an object of an abstract class or interface.
5	InterruptedException	One thread has been interrupted by another thread.
6	NoSuchFieldException	A requested field does not exist.
7	NoSuchMethodException	A requested method does not exist.
8	ReflectiveOperationException	Superclass of reflection-related exceptions.

2.
3. **Error**
   - External to the application
   - Application usually cannot anticipate it and recover from it.
   - E.g. hardware or system malfunction causes unsuccessful read,
     Then IOError will be thrown.
   - Error and its subclasses
   -
4. Rutime Exception
   - Internal to application
   - Cannot anticipate it and recover from
   - E.g NullPointerException, ArithmeticException, etc.
   - Runtimexception and its subclasses

Unchecked Exception = Error + RuntimeException

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

Here is the output generated by running it both ways:

1. a = 0
   Divide by 0: java.lang.ArithmeticException: / by zero
   After try/catch blocks.

2. a = 1
   Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
   After try/catch blocks.

```
class SuperSubCatch {
        public static void main(String args[]) {
                try {
                        int a = 0;
                        int b = 42 / a;
                } catch(Exception e) {
                        System.out.println("Generic Exception catch.");
                }
                /* This catch is never reached because
                ArithmeticException is a subclass of Exception. */

                catch(ArithmeticException e) { // ERROR – unreachable
                        System.out.println("This is never reached.");
}       }       }
```

```
// An example of nested try statements.
class NestTry {
        public static void main(String args[]) {
                try {
                        int a = args.length;
                        /* If no command-line args are present, the following statement
                        will generate a divide-by-zero exception. */
                        int b = 42 / a;
                        System.out.println("a = " + a);

                        try { // nested try block
                                /* If one command-line arg is used, then a divide-by-zero
                                exception will be generated by the following code. */
                                if(a==1)
                                        a = a/(a-a); // division by zero
                                /* If two command-line args are used,
                                then generate an out-of-bounds exception. */
                                if(a==2) {
                                        int c[] = { 1 };
                                        c[42] = 99; // generate an out-of-bounds exception
                                }
                        }
                        catch(ArrayIndexOutOfBoundsException e) {
                                System.out.println("Array index out-of-bounds: " + e);
                        }

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

OUTPUT:

java NestTry	Divide by 0: java.lang.ArithmeticException: / by zero
java NestTry One	a = 1 Divide by 0: java.lang.ArithmeticException: / by zero
java NestTry One Two	a = 2 Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException:42

throw keyword

- So far, the exceptions caught were thrown by the Java run-time system.
- However, it is possible to throw an exception explicitly, using the throw statement.

The general form of throw is shown here: **throw ThrowableInstance;**

- 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:

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

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
- The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.
- If it does find a match, control is transferred to that statement.
- If not, then the next enclosing try statement is inspected, and so on.
- If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

- Many of Java's built-in run-time exceptions have at least two constructors:
- One with no parameter and one that takes a string parameter.
- When the second form is used, the argument specifies a string that describes the exception.
- This string is displayed when the object is used as an argument to print( ) or println( ).
- It can also be obtained by a call to getMessage( ), which is defined by Throwable.
- Example: in the following programs if we write **System.out.println(e.getMessage());** in catch block then **demo** will be printed.

Program 1

```
class ThrowDemo {
        static void demoproc() {
                try {
                        throw new NullPointerException("demo");
                }catch(NullPointerException e) {
                        System.out.println("Caught inside demoproc. "+e);
        }       }
        public static void main(String args[]) {
                demoproc();
}       }
```

OUTPUT:Caught inside demoproc. java.lang.NullPointerException: demo

Program 2 : throw and catch in calling method

```
class ThrowDemo {
	static void demoproc() {
		throw new NullPointerException("demo");
	}
	public static void main(String args[]) {
		try {
			demoproc();
		} catch(NullPointerException e) {
			System.out.println("Caught: " + e);
}	}	}
```

OUTPUT:Caught: java.lang.NullPointerException: demo

Program 3: rethrow the exception from catch

```
class ThrowDemo {
	static void demoproc() {
		try {
			throw new NullPointerException("demo");
		}
		catch(NullPointerException e) {
			System.out.println("Caught inside demoproc. "+e);
			throw e; // rethrow the exception
	}	}
	public static void main(String args[]) {
			demoproc();
}	}
```

OUTPUT:

```
Caught inside demoproc. java.lang.NullPointerException: demo
Exception in thread "main" java.lang.NullPointerException: demo
	at ThrowDemo.demoproc(ThrowDemo.java:4)
	at ThrowDemo.main(ThrowDemo.java:13)
```

Program 4: catch the rethrown exception in calling method

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

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

OUTPUT:
Caught inside demoproc. java.lang.NullPointerException: demo
Caught: java.lang.NullPointerException: demo

Throws

- If a 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.
- This can be done by including a throws clause in the method's declaration.
- 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. (these exceptions are checked exceptions which the program should anticipate)
- This is the general form of a method declaration that includes a throws clause:

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

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
class ThrowsDemo {
        static void throwOne() {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[]) {
                throwOne();
}       }
```

OUTPUT: compile time error
ThrowsDemo.java:4: error: unreported exception IllegalAccessException; must be caught or declared to be thrown
                throw new IllegalAccessException("demo");
1 error

Program 2

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

OUTPUT: compile time error

ThrowsDemo.java:7: error: unreported exception IllegalAccessException; must be caught or declared to be thrown

```
		throwOne();
```

1 error

Program 3

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

OUTPUT:

```
Inside throwOne.
Exception in thread "main" java.lang.IllegalAccessException: demo
	at ThrowsDemo.throwOne(ThrowsDemo.java:4)
	at ThrowsDemo.main(ThrowsDemo.java:7)
```

Program 4

```
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(e);
}	}	}
```

OUTPUT:

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

Program 5

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

OUTPUT:

```
ThrowsDemo.java:15: error: unreported exception IllegalAccessException; must be caught or declared to be thrown
		throwOne();
1 error
```

Program 6

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

OUTPUT:

```
Inside throwOne.
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 statement matches the exception.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

```
import java.util.Scanner;
class FinallyDemo2 {
	public static void main(String args[]) {
		Scanner sc = new Scanner(System.in);
		int a = sc.nextInt();
		int b = sc.nextInt();
		try{
			System.out.println(a/b);
		}
		catch(ArithmeticException e){
			System.out.println("cannot divide by zero");
		}
		finally {
			System.out.println("after trycatch");
}	}	}
```

OUTPUT: Code with finally

1. **no exception a=10 b=2**

```
5
after trycatch
```

2. **if catch(ArrayIndexOutOfBoundsException e) instead of catch(ArithmeticException e) a=10 b=0**

```
after trycatch
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at FinallyDemo2.main(FinallyDemo.java:9)
```

3. **correct handler available a=10 b=0**

```
cannot divide by zero
after trycatch
```

OUTPUT: if finally block is not written.

1. **no exception a=10 b=2**

5
after trycatch

2. **if catch(ArrayIndexOutOfBoundsException e) instead of catch(ArithmeticException e) a=10 b=0**

Exception in thread "main" java.lang.ArithmeticException: / by zero
at FinallyDemo2.main(FinallyDemo.java:9)

3. **correct handler available a=10 b=0**

cannot divide by zero
after trycatch

Example:

```
class FinallyDemo {
        static void procA() {
                try {
                        System.out.println("inside procA");
                        throw new RuntimeException("demo");
                } finally {     System.out.println("procA's finally"); }
        }
        static void procB() {
                try {
                        System.out.println("inside procB");
                        return;
                } finally {     System.out.println("procB's finally");     }
        }
        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();
        }
}
```

OUTPUT:
inside procA
procA's finally

Exception caught
inside procB
procB's finally

inside procC
procC's finally

Create a **custom exception InvalidAgeException**. If the entered age is less than 18 years then InvalidAgeException gets thrown.

```
class InvalidAgeException extends Exception{
	InvalidAgeException(String msg) {
		super(msg);
	}
}
class CustomExceptionDemo{
	static void validate(int age)throws InvalidAgeException{
		if(age<18)
			throw new InvalidAgeException("not allowed to vote");
		else
			System.out.println("Can vote");
	}
	public static void main(String args[])throws InvalidAgeException{
		int age;
		age=13;
		try{
			validate(age);
		}
		catch(InvalidAgeException e){
			System.out.println("Exception occurred: "+e);
		}
		age=20;
		try{
			validate(age);
		}
		catch(InvalidAgeException e){
			System.out.println("Exception occurred: "+e);
		}
	}
}
```

OUTPUT:

```
Exception occurred: InvalidAgeException: not allowed to vote
Can vote
```

```
class MyStack{
	int tos;
	int []elements;
	MyStack(){
		tos=-1;
		elements = new int[10];
	}
	void push(int n) throws StackOverFlowException{
		tos++;
		if(tos>=9){
			tos--;
			throw new StackOverFlowException("Stack is full");
		}
		else
			elements[tos]=n;
	}
	int pop() throws StackUnderFlowException{
		int x=-1;
		if(tos>=0){
			x = elements[tos];
			tos--;
		}
		else{
			throw new StackUnderFlowException("Stack is empty");
		}
		return x;
	}
}

class StackOverFlowException extends Exception{
	StackOverFlowException(String msg){
		super(msg);
	}
}

class StackUnderFlowException extends Exception{
		StackUnderFlowException(String msg)   {
			super(msg);
		}
}
```

```
class StackExceptionDemo{
	public static void main(String args[]){
		MyStack s = new MyStack();
		for(int i=0;i<12;i++){
			try{
				s.push(i);
				System.out.println("Pushed "+i);
			}
			catch(Exception e)   {
				System.out.println("Exception occurred: "+e);
		}	}
		for(int i=0;i<10;i++){
			try{
				int x = s.pop();
				System.out.println("Popped" + x);
			}
			catch(Exception e){
				System.out.println("Exception occurred: "+e);
}	}	}	}
```

OUTPUT:

```
Pushed 0
Pushed 1
Pushed 2
Pushed 3
Pushed 4
Pushed 5
Pushed 6
Pushed 7
Pushed 8
Exception occurred: StackOverFlowException: Stack is full
Exception occcured: StackOverFlowException: Stack is full
Exception occcured: StackOverFlowException: Stack is full
Popped8
Popped7
Popped6
Popped5
Popped4
Popped3
Popped2
Popped1
Popped0
Exception occcured: StackUnderFlowException: Stack is empty
```

Chained Exception

To allow chained exceptions, two constructors and two methods were added to Throwable. The constructors are shown here:

1. **Throwable(Throwable cause)** - the cause is the current exception.
2. **Throwable(String msg, Throwable cause)** - msg is the exception message, the cause is the current exception.

Methods:

1. Throwable getCause( ) :- returns actual cause.
2. Throwable initCause(Throwable causeExc):- sets the cause for calling an exception.

- The **getCause( )** method returns the exception that underlies the current exception.
- If there is no underlying exception, **null** is returned.

- The **initCause( )** method associates *causeExc* with the invoking exception and returns a reference to the exception.
- A cause can be associated with an exception after the exception has been created.

- However, the cause exception can be set only once.
- Thus, you can call **initCause( )** only once for each exception object.
- Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause( )**.

```
class TestChainException{
	static void method1(){
		Integer a=null;
		try	{
			int b= new java.util.Scanner(System.in).nextInt();
			a=10/b;
		}catch(ArithmeticException e){	}
		int x = a.intValue();
		System.out.println("x = "+x);
	}
	public static void main(String args[]){
		try	{
			method1();
		}catch(NullPointerException e){
			System.out.println("Exception occured: "+ e);
}	}	}
```

OUTPUT:

```
0
Exception occured: java.lang.NullPointerException
```

Now actual cause of NullPointerException is the ArithmeticException

```
class TestChainException
{
	static void method1(){
		Integer a=null;
		ArithmeticException ae = null;
		try	{
			int b= new java.util.Scanner(System.in).nextInt();
			a=10/b;
		}catch(ArithmeticException e){
			//System.out.println("Exception occured: "+ e);
			ae=e;
		}
		try	{
			int x = a.intValue();
			System.out.println("x = "+x);
		}catch(NullPointerException e){
			e.initCause(ae);
			throw e;
	}	}
	public static void main(String args[]){
		try	{
			method1();
		}catch(NullPointerException e){
			System.out.println("Exception occured: "+ e);
			System.out.println("Cause :"+e.getCause());
		}
	}
}
```

OUTPUT:

```
0
Exception occured: java.lang.NullPointerException
Cause :java.lang.ArithmeticException: / by zero
```

Program2

```
class ChainExcDemo {
	static void demoproc() {
		// create an exception
		NullPointerException e = new NullPointerException("top layer");
		// add a cause
		e.initCause(new ArithmeticException("cause"));
		throw e;
	}
	public static void main(String args[]) {
		try {
			demoproc();
		} catch(NullPointerException e) {
			// display top level exception
			System.out.println("Caught: " + e);
			// display cause exception
			System.out.println("Original cause: " +
			e.getCause());
		}
	}
}
```

OUTPUT:
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause