JAVA ASSIGNMENT 1

ROLL NO 1.

5TH SEM MCA

Q1. What is the Throwable class? Differentiate between Exception and Error class.

**Throwable** class is the superclass of all errors and exceptions in the Java. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.

Declaration of Throwable:

public class Throwable

extends Object

implements Serializable

**Error** class represents the errors which are mainly caused by the environment in which application is running. For example, **OutOfMemoryError** occurs when JVM runs out of memory or **StackOverflowError** occurs when stack overflows.

**Exception** class represents the exceptions which are mainly caused by the application itself. For example, **NullPointerException** occurs when an application tries to access null object.

|  |  |
| --- | --- |
| **Errors** | **Exceptions** |
| Errors in java are of type java.lang.Error. | Exceptions in java are of type java.lang.Exception. |
| All errors in java are unchecked type. | Exceptions include both checked as well as unchecked type. |
| Errors happen at run time. They will not be known to compiler. | Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time. |
| It is impossible to recover from errors. | You can recover from exceptions by handling them through try-catch blocks. |
| Errors are mostly caused by the environment in which application is running. | Exceptions are mainly caused by the application itself. |
| Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException |

Q2. What is the default behaviour if a RuntimeException occurs without using try and catch? Give an example.

Any exception that is not caught by our program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Example

public static void main(String args[]){

int a[] = new int[2];

a[0]=1;

a[1]=2;

a[2]=3;

a[3]=2;

for(int i=0;i<10;i++)

System.out.println(" s"+a[i]);

}

On running the program: java ExTest gives

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2

at ExTest.main(ExTest.java:12)

When the Java run-time system detects the attempt to access the array with an invalid index , it constructs a new exception object and then *throws* this exception. This causes the execution of the classto stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven’t supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Q3. Can we catch two or more exceptions occurring within the same try block?

Yes, we can catch two or more exceptions raised by a single piece of code.

To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try / catch block. The following example traps two different exception types:

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.");

}

Another way is to simply use a single catch block which can handle all types of exceptions by using an Exception object in the catch block,

try {

int a = args.length;

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

int b = 42 / a;

int c[] = { 1 };

c[42] = 99;

}

catch(Exception e) {

System.out.println(“Got Exception ”+e);

}

Q4. What do you understand by nested try statements?

Nested try statements means that a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement’s catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

// An example of nested try statements.

class NestTry {

public static void main(String args[]) {

try {

int a = args.length;

int b = 42 / a;

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

try { // nested try block

if(a==1) a = a/(a-a); // division by zero

if(a==2) {

int c[] = { 1 };

c[42] = 99;

}

}

catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Array index out-of-bounds: " + e);

}

}

catch(ArithmeticException e) {

System.out.println("Divide by 0: " + e);

}

}

}

Q5. Differentiate between **throw** and **throws** keyword.

|  |  |  |
| --- | --- | --- |
| **No.** | **Throw** | **Throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | We cannot throw multiple exceptions. | We can declare multiple exceptions  e.g.  public void method() throws IOException,SQLException. |

Q6. What is the significance of the finally block?

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism.

The finally keyword is designed to address this contingency.

The finally keyword 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.

// Demonstrate finally.

class FinallyDemo {

// Throw an exception out of the method.

static void procA() {

try {

System.out.println("inside procA");

throw new RuntimeException("demo");

}

finally {

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

}

}

// Return from within a try block.

static void procB() {

try {

System.out.println("inside procB");

return;

} finally {

System.out.println("procB's finally");

}

}

// Execute a try block normally.

static void procC() {

try {

System.out.println("inside procC");

}

finally {

System.out.println("procC's finally");

}

}

public static void main(String args[]) {

try {

procA();

}

catch (Exception e) {

System.out.println("Exception caught");

}

procB();

procC();

}

}

In this example, procA( ) prematurely breaks out of the try by throwing an exception.

The finally clause is executed on the way out. procB( )’s try statement is exited via a return

statement. The finally clause is executed before procB( ) returns. In procC( ), the try

statement executes normally, without error. However, the finally block is still executed.

Q7 . With proper examples illustrate unchecked and checked exceptions.

|  |  |
| --- | --- |
| ***Checked*** | ***Unchecked*** |
| The compiler checks the checked exception. | The compiler does not check the Unchecked exception. |
| Except "RuntimeException" class all the child classes of the class "Exception", and the "Error" class and its child classes are Checked Exception. | "RuntimeException" class and its child classes, are"Unchecked Exceptions". |
| If we do not handle the checked exception, then the compiler objects. | Even if we do not handle the unchecked exception, the compiler doesn't object. |
| The program doesn't compile if there is an unhandled checked exception in the program code. | The program compiles successfully even if there is an unhandled unchecked exception in the program code. |

Examples of Unchecked Exception:

class Example {

public static void main(String args[])

{

int arr[] ={1,2,3,4,5};

System.out.println(arr[7]);

}

}

Examples of Checked Exception:

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

fis = new FileInputStream("B:/myfile.txt");

//This constructor FileInputStream(File filename) throws FileNotFoundException which is a checked exception.

int k;

while(( k = fis.read() ) != -1)

{

//Method read() of FileInputStream class also throws a checked exception: IOException.

System.out.print((char)k);

}

fis.close();

//The method close() closes the file input stream it throws IOException.

}

}

Q8. Create user defined exceptions **StackFullException** and **StackEmptyException** that must be thrown when a stack object is full or empty respectively.