

Core Java 8 and Development Tools

Lesson 09 : Exception Handling

Lesson Objectives

- On completion of this lesson, you will be able to:
 - Explain the concept of Exception
 - Describe types of Exceptions
 - Handle Exception in Java
 - Create your own Exceptions
 - State best practices on Exception



2

This lesson deals with Java's exception-handling mechanism. An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.

Lesson Outline:

- 9.1: Introduction
- 9.2: Exception Types
- 9.3: Exception Hierarchy
- 9.4: Try-catch-finally
- 9.5: Try-with-resources
- 9.6: Multi catch blocks
- 9.7: Throwing exceptions using throw
- 9.8: Declaring exceptions using throws
- 9.9: User defined Exceptions
- 9.10: Best Practices

9.1: Exception Handling – Fundamentals

Why is exception handling used?

- No matter how well-designed a program is, there is always a chance that some kind of error will arise during its execution, for example:
 - Attempting to divide by 0
 - Attempting to read from a file which does not exist
 - Referring to non-existing item in array
- An exception is an event that occurs during the execution of a program that disrupts its normal course.



3

Why Exception Handling?

Java was designed with the understanding that errors occur, that unexpected events happened and the programmer should always be prepared for the worst. The preferred way of handling such conditions is to use exception handling, an approach that separates a program's normal code from its error-handling code.

9.1: Exception Handling – Fundamentals

Exception Handling

- Exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions:
 - Eg: Hard disk crash, Out of bounds array access, Divide by zero etc
- When an exception occurs, the executing method creates an Exception object and hands it to the runtime system — “throwing an exception”
- The runtime system searches the runtime call stack for a method with an appropriate handler, to handle/catch the exception.



end 4

Exception Handling:

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Exceptions are used as a way to report occurrence of some exceptional condition. Exception provides a means of communicating information about errors up through a chain of methods until one of them handles it.

Note: Java exception handling is similar to the one used in C++.

The exception mechanism is built around the throw-and-catch paradigm. When an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system. This process is known as throwing an exception. The exception object contains information about the exception, including its type and the state of the program when the error occurred. To catch an exception is to take appropriate action to deal with the exception.

Java's Exception handling mechanism is managed by five keywords: try, catch, throw, throws and finally.

9.1: Exception Handling – Fundamentals

Exception Handling

- The general form of exception handling block:

```
try {  
    //code to be monitored.  
}  
catch (Exception1 e1 ) {  
    //exception handler for Type Exception1  
}  
catch (Exception2 e2 ) {  
    //exception handler for Type Exception2  
}  
finally {  
    // code that must be executed.  
}
```

Exception Handling (Contd.):

Normally, the program code that you want to observe for exceptions is written in the try block. If an exception occurs within a try block, it is thrown. Your code can catch this exception (using the catch block), handle the situation gracefully and continue to be in a program. Any code, that absolutely must be executed, regardless of whether exception has occurred or not, can be put into finally block.


In the above code fragment, Exception1 and Exception2 are being caught. The default handler ultimately processes an exception that is not caught by your program. The default handler displays a string describing the exception, and prints a stack trace from the point at which the exception occurred.

9.1: Exception Handling – Fundamentals

Demo

- Execute the DefaultDemo.java program

```
class DefaultDemo {  
    public static void main(String a[]) {  
        String str = null;  
        str.equals("Hello");  
    }  
}
```



Output:
Exception in thread "main"
java.lang.NullPointerException at
com.igatepatni.lesson5.DefaultDemo.main(DefaultDemo.java:6)

The code throws a Exception:

Exception in thread "main" java.lang.NullPointerException
at DefaultDemo.main(DefaultDemo.java:5)

This is because the String Object is not created and is therefore Null. When methods are invoked on such referenced objects, a NullPointerException is thrown!

9.1: Exception Handling – Fundamentals

Advantages of Exceptions

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

Code without Exception handling

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

Code with Exception handling

```
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 error handling code and regular code are separate

7

Advantages of Exceptions:

Exceptions provide the means to separate the details of what to do when something exceptional happens, which differs from the main logic of a program. Refer to the code snippet without Exception handling. At first glance, this function seems simple enough, but it ignores all the following 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?

Advantages of Exceptions (Contd.):

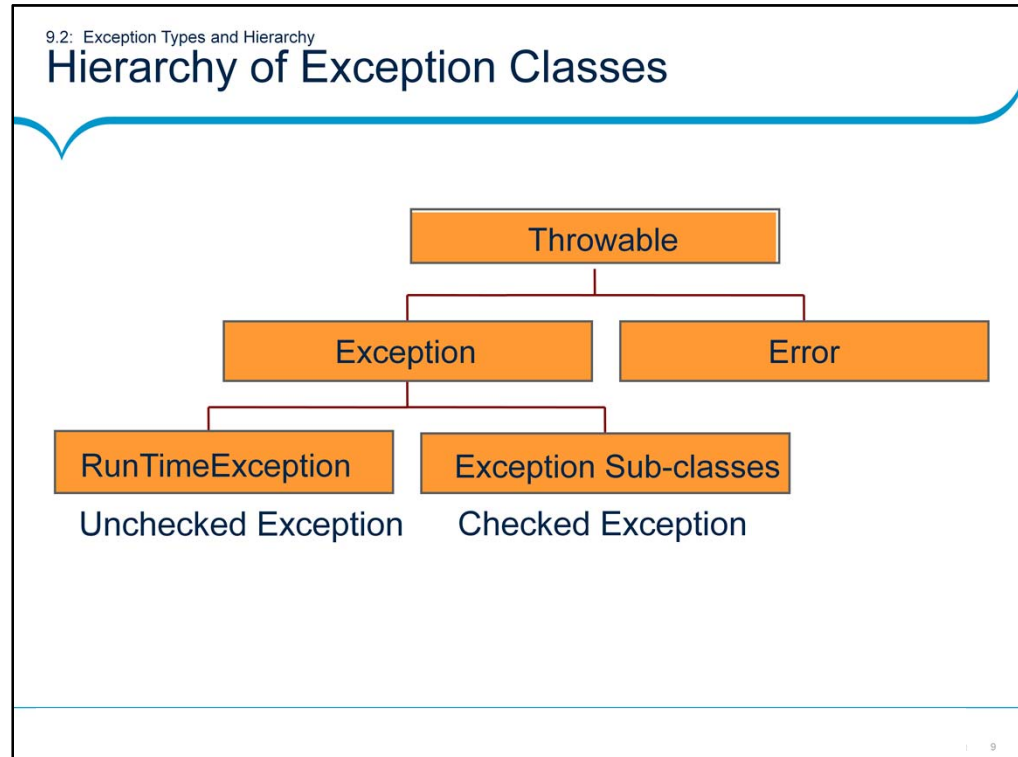
To handle such cases, the **readFile** function must have more code to do error detection, reporting, and handling. Here is an example of the function code.

```
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 is so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Moreover, the logical flow of the code has also been lost. This makes it difficult to check whether the code is performing the right function: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to carry out the right function when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

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 be coded as shown in box #2 in the previous slide.

Note that exceptions do not spare you the effort involved in detecting, reporting, and handling errors; however, they help you organize the work more effectively.



Hierarchy of Exception Classes:

All exceptions are derived from the `java.lang.Throwable` class. `Throwable` is at top of the execution class hierarchy. Immediately below `Throwable` are two subclasses, `Error` and `Exception`, which categorize exceptions into two distinct branches.

Exception class: This class is used for exceptional conditions that user programs should catch. This is also the class that you use as a subclass to create your own custom exception types. There is an important subclass of `Exception`, called `RuntimeException`.

Error class: This class defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type `Error` are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

9.2: Exception Types and Hierarchy

Error

- An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch.
- Most such errors are abnormal conditions.
- Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
 - Stack overflow is an example of such an error.

est 10

Error:

Instances of Error are internal errors in Java runtime environment. These are rare and usually fatal and therefore not supposed to be handled by the program. Instances of error are thrown, when the Java Virtual Machine faces some memory leakage problem, insufficient memory problem, dynamic linking failure or when some other "hard" failure in the virtual machine occurs. Obviously, in case of an error, the program stops executing.

9.2: Exception Types and Hierarchy

Exception

- The Exception class and its subclasses are a form of Throwables. They indicate conditions, which a reasonable application may want to catch.
- Two Types:
 - Checked Exception
 - UnChecked Exception

and 11

Java distinguishes between two categories of exceptions:

Checked exceptions

Unchecked exceptions

This distinction is important, because the Java compiler enforces a catch-or-declare requirement for checked exceptions. An exception's type determines whether the exception is checked or unchecked. All classes that inherit from the Exception class but not the RuntimeException class are considered to be checked exceptions. The compiler checks each method call and method declaration to determine whether the method throws checked exceptions. If so, the compiler ensures that the checked exception is caught or is declared in a throws clause.

9.2: Exception Types and Hierarchy

Checked/Compile Time Exceptions

■ Characteristics of Checked Exceptions:

- They are checked by the compiler at the time of compilation.
- They are inherited from the core Java class Exception.
- They represent exceptions that are frequently considered “non-fatal” to program execution.
- They must be handled in your code, or passed to parent classes for handling.
- Some examples of Checked exceptions include: IOException, SQLException, ClassNotFoundException

12

Example:

```
try {
    DriverManager.registerDriver (new
oracle.jdbc.driver.OracleDriver());
    Connection conn =
DriverManager.getConnection (
        "jdbc:oracle:thin:@DbServer:trgdb",
        "scott", "tiger");
    .....
} catch (SQLException e) {

    System.out.println(e.getMessage());
}
```

An SQLException can be thrown while attempting to connect to database. Hence must be caught.

9.2: Exception Types and Hierarchy

Unchecked/Runtime Exceptions

- Unchecked exceptions represent error conditions that are considered “fatal” to program execution.
 - Runtime exceptions are exceptions which are not detected at the time of Compilation.
 - They are encountered only when the program is in execution.
 - It is called unchecked exception because the compiler does not check to see if a method handles or throws these exceptions.

and 13

Unchecked/Runtime Exceptions:

These kind of exceptions are encountered only while the program is in execution. Program may take actions to handle them explicitly or are propagated till java run-time handles them. In the latter case, program is terminated abruptly. Before you learn to handle exceptions in a program, it is useful to understand what happens when they are not handled. This program includes a code that causes `NullPointerException` because an object of the `String` class is not created.

```
class DefaultDemo {  
    public static void main(String a[]) {  
        String str = null;  
        str.equals("Hello");  
    }  
}
```

This program will compile properly. When java run-time system detects the attempt to call `equals()` method on null object, it constructs a new exception object and throws this exception. This halts the execution of program, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. As there is no explicit exception handler defined, it is handled by the default handler. The default handler displays a string describing the exception; prints stack trace and terminates the program.

Unchecked/Runtime Exceptions (Contd.):

Exception in thread "main" java.lang.NullPointerException
at DefaultDemo.main(DefaultDemo.java:4)

In this, the class name **DefaultDemo**; the method name **main**, the file name **DefaultDemo.java** and line number **4**, are all included in simple stack trace. The type of thrown exception is **NullPointerException**.

Many exception types are in this category. They are always thrown automatically by Java and they need not be included in your exception specifications.

Conveniently enough, they're all grouped together by putting them under a single base class called **RuntimeException**, which is a perfect example of inheritance: it establishes a family of types that have some characteristics and behaviors in common.

In addition, you never need to write an exception specification saying that a method might throw a **RuntimeException**, since that's just assumed. Because they indicate bugs, you virtually never catch a **RuntimeException** – it's dealt with automatically. If you are forced to check for **RuntimeExceptions**, your code could get messy. Even though you don't typically catch **RuntimeExceptions**, in your own packages you might choose to throw some of the **RuntimeExceptions**.

What happens when you don't catch such exceptions? Since the compiler doesn't enforce exception specifications for these, it's quite plausible that a **RuntimeException** could percolate all the way to your **main()** method without being detected.

9.3: Handling Exceptions

Keywords for handling Exceptions

- **try** : This marks the start of a block associated with a set of exception handlers.
- **catch** : The control moves here if an exceptions is generated.
- **finally** :This is called irrespective of whether an exception has occurred or not.
- **throws** : This describes the exceptions which can be raised by a method.
- **throw** : This raises an exception to the first available handler in the call stack, unwinding the stack along the way.

15

Mnemonic Hint: Throw two and try to catch finally.

9.3: Handling Exceptions

Why to handle exceptions?

▪ Without Exception handling

```
class WithoutException {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
        System.out.println("Will not be printed"); } }
```

▪ With Exception handling

```
class WithExceptionHandling {  
    public static void main(String args[]) {  
        int d=0, a;  
        try {  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        } catch (ArithmeticException e) {  
            System.out.println("Division by zero.");  
        }  
        System.out.println("This will get printed"); } }
```

16

Why to handle exceptions?

The code in box #1 includes an expression that intentionally causes a divide-by-zero error.

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of `WithoutException` to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

In this example, exception handlers haven't been supplied, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by this program is ultimately 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. Here is the output generated when this example is executed.

```
java.lang.ArithmeticException: / by zero  
at WithoutException.main(Exc0.java:4)
```

Handling an Exception yourself provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. The user of your application would be confused if your program stopped running and printed a stack trace whenever an error occurred.

Why to handle exceptions? (Contd.)

Fortunately, it is quite easy to prevent this. To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block.

Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. The code in box #2 includes a try block and a catch clause which processes the `ArithmeticException` generated by the division-by-zero error.

This program generates the following output:

Division by zero.

This will get printed

Notice that the call to `println()` inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. Thus, the line "This will not be printed." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

9.3: Handling Exceptions

Try and Catch

- The try structure has three parts:
 - The try block : Code in which exceptions are thrown
 - One or more catch blocks : Respond to different Exceptions
 - An optional finally block : Contains code that will be executed regardless of exception occurring or not
- The catch Block:
 - If exception occurs in try block, program flow jumps to the catch blocks.
 - Any catch block matching the caught exception is executed.

out 18

Syntax for the try and catch block is shown on page 5-05.

The default exception handler provided by java run-time system is useful for debugging. However, it's a good programming practice for the user to handle the exceptions.

The same example with exception handling is shown below. Observe the difference in the output. A try statement must be accompanied by at least one catch block and if required, one finally block.

```
class TryCatchDemo {  
    public static void main(String a[]) {  
        String str = null;  
        try {  
            str.equals("Hello");  
        } catch (NullPointerException ne) {  
            str = new String("Hello");  
            System.out.println(str.equals("Hello"));  
        }  
        System.out.println("Continuing in the program");  
    }  
}
```

The output is:

True

Continuing in the program...

9.3: Handling Exceptions

Using Try and Catch

- Execute the TryCatchDemo.java program

19

The code is shown in the previous page.

9.4: Try-with-resources

Try-with-resources

- Resources used by java programs like file or database connection needs to be closed properly
- To close resource automatically in exception handling, Java 7 has added try-with-resources:

```
try (resource1; resource 2; ..... resource n) {  
    //resource related work  
}  
catch (Exception e) {  
    //handle exception  
}
```

- It can be used to close resources that implement `java.lang.AutoCloseable`

20

Many times Java programs need to work with resources like file, database connection or network socket etc. After work, such resources must be closed gracefully to avoid loss of data. There are two places in exception handling where resource can be closed.

Finally Block (will be discussed later)
try-with-resources

A try-with-resources is a new feature added in java 7, where resources are closed automatically. Any block after try (either catch or finally block) will be executed only after the resource is closed.

9.5: Multi catch blocks

Multiple Catch Blocks

- If you include multiple catch blocks, the order is important.

```
public void divide(int x,int y)
{
    int ans=0;
    try{
        ans=x/y;
    }catch(Exception e) {
        //handle }
    catch(ArithmeticException f) {
        //handle}
}
```



You must catch subclasses before their ancestors



21

Multiple Catch Blocks:

The following example shows the uses of multiple catch clauses.

```
class MultiCatch {
    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); }
        catch(Exception e) {
            System.out.println("Generic Exception: " + e); }
        System.out.println("After try/catch blocks.");
    }
}
```

Output:

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

9.5: Multi catch blocks

Multi-Catch Blocks

- Starting from Java 7, a single catch block can be used to catch multiple exceptions
- Multi-catch block separates handled exceptions by vertical bar (|)

```
try {  
    }  
catch (exception 1 | exception 2 | .....| exception n) {  
    }
```

- Alternatives in multi-catch block is not allowed

22

Multiple-catch blocks:

Java 7 allows a single catch block to catch multiple exceptions. The only precaution required is not to list multiple exceptions which are related by subclassing. It means you cannot catch child and parent exception in multi-catch block. The following example throws an error as `ArithmeticException` is subclass of `Throwable` and hence cannot be combined in multi-catch block.

```
class MultiCatch {  
    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 | Throwable e) {  
            System.out.println("Exception: " + e);  
        }  
        System.out.println("After try/catch blocks.");  
    }  
}
```

Error!

9.5: Multi catch blocks

Multiple Catch Blocks

- Execute the MultiCatch.java class

23

The code for this example is shown in the previous page.

The last catch block is a generic catch block, which can catch all kinds of exceptions. This is a generalized catch block and should always appear as a last block in the list of catch blocks. If first two blocks cannot handle the exception thrown, it will be definitely handled by a generic block.

Try and catch blocks can be nested and if inner catch blocks are unable to handle an exception, it's escalated to the outer catch blocks. This continues until either one of the catch blocks handles the exception or all the try statements are exhausted.

9.5: Handling Exceptions

Nested Try Catch Block

```
try {
    int a = arg.length;
    int b = 10 / a;
    System.out.println("a = " + a);
    try {
        if(a==1)
            a = a/(a-a);
        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); }
```

24

Nested Try Catch Block:

The try statement can be nested. 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 handles the exception.

```
class Nesteddemo {
    public static void main(String arg[]) {
        try {
            int a = arg.length;
            int b = 10 / a;
            System.out.println("a = " + a);
            try {
                if(a==1)
                    a = a/(a-a);
                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); } } }
```


9.5: Handling Exceptions

The Finally Clause

- The finally block is optional.
- It is executed whether or not exception occurs.

```
public void divide(int x,int y)
{
    int ans;
    try{
        ans=x/y;
    }
    catch(Exception e) {
        ans=0; }
    finally{
        return ans; // This is always executed }
}
```

25

The Finally Clause:

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, the method may return prematurely. For example, if a method opens a database connection on 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.

Finally creates a block of code that is executed after try/catch block has completed and before the code following the try/catch block. The finally block executes whether or not an exception is thrown.

If an exception is thrown, the finally block executes even if no catch statements matches the exception. Finally is guaranteed to execute, even if no exceptions are thrown. The Finally block is an ideal position for closing the resources such as file handle or a database connection, and so on.

A finally block typically contains code to release resources acquired in its corresponding try block; this is an effective way to eliminate resource leaks. For example, the finally block should close any files opened in the try block.

9.5: Handling Exceptions

Demo: The Finally Clause

- Execute the FinallyDemo.java program

Output of this prg:

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

```
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() { // Return from within a try block.
        try {
            System.out.println("inside procB");
            return;
        } finally { System.out.println("procB's finally"); }
    }
    static void procC() { // Execute a try block normally.
        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();
    }
}
```

9.6: Throwing Exceptions

Throwing an Exception

- You can throw your own runtime errors:
 - To enforce restrictions on use of a method
 - To "disable" an inherited method
 - To indicate a specific runtime problem
- To throw an error, use the throw Statement
 - throw ThrowableInstance
where ThrowableInstance is any Throwable Object

end 27

Throwing an Exception:

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

9.6: Throwing Exceptions

Throwing an Exception

```
class ThrowDemo {  
    void proc() {  
        try {  
            throw new FileNotFoundException ("From Exception");  
        } catch(FileNotFoundException e) {  
            System.out.println("Caught inside demoproc.");  
            throw e; // rethrow the exception  
        }  
    }  
    public static void main(String args[]) {  
        ThrowDemo t=new ThrowDemo();  
        try {  
            t.proc();  
        } catch(FileNotFoundException e) {  
            System.out.println("Recaught: " + e);  
        }  
    }  
}
```

end 28

Throwing an Exception:

This program gets two chances to deal with the same error. First, `main()` sets up an exception context and then calls `proc()`. The `proc()` method then sets up another exception-handling context and immediately throws a new instance of `FileNotFoundException`, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

Caught inside demoproc.

Recaught: java.lang.FileNotFoundException: From Exception

The program also illustrates how to create one of Java's standard exception objects.

Example:

```
throw new FileNotFoundException();
```

Here, `new` is used to construct an instance of `FileNotFoundException`. All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.

9.7: Declaring exceptions using throws

Using The Throws Clause

- If a method might throw an exception, you may declare the method as “throws” that exception and avoid handling the exception yourself.

```
public class ThrowsDemo {  
    public static void main(String[] args) {  
        try {  
            fileOpen();  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
            System.out.println("File name specified does not exist "  
                + e.getMessage());  
        }  
  
        static void fileOpen() throws FileNotFoundException {  
            FileReader fileReader = new FileReader("test.txt");  
        } } }
```

29

Using The Throws Clause:

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 achieved by using the Throws clause in the method declaration.

The Throws clause can throw multiple exceptions separated by commas. It lists the type of exceptions that a method might throw.

Here is the output generated by running this example program:

```
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 100  
at ThrowsDemo.doWork(ThrowsDemo.java:11)  
at ThrowsDemo.main(ThrowsDemo.java:4)
```

The Throws clause is added to an application to indicate to the rest of the program that this method may throw an `ArithmeticException`. Clients of method `doWork()` are thus informed that the method may throw an `ArithmeticException` and that the exception should be caught.

9.8: User defined Exceptions

User specific Exception

- To create exceptions:

- Write a class that extends(indirectly) Throwable.
- What Superclass to extend?
 - For unchecked exceptions: RuntimeException
 - For checked exceptions: Any other Exception subclass or the Exception itself

```
class AgeException extends Exception {  
    private int age;  
    AgeException(int a) {  
        age = a;  
    }  
    public String toString() {  
        return age+" is an invalid age";  
    }  
}
```

30

User-Specific Exception:

To create your own exception class, you must inherit from an existing exception class, preferably one that is close in meaning to your new exception. Define a subclass of Exception. Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them. The following are some of the useful methods.

String toString(): This exception returns a String object containing a description of the exception. This method is called by **println()** when outputting a Throwable object.

String getMessage(): This exception returns a description of the exception.

void printStackTrace(): This exception displays the stack trace.

User Specific Exception (Contd.):

Example:

```

import java.util.*;
class AgeException extends Exception {
    private int age;
    AgeException(int a) {
        age = a; }
    public String toString() {
        return age+" is an invalid age"; } }
class emp {
    String name;
    int age;
    void getDetails() throws AgeException {
        System.out.println("Enter your name:");
        Scanner sc=new Scanner(System.in);
        name=sc.next();
        System.out.println("Enter your age:");
        age=sc.nextInt();
        if (age<16)
            throw new AgeException(age);
    }
}
class ExceptionDemo {
    public static void main(String args[]) {
        try {
            emp e=new emp();
            e.getDetails();
        }catch (AgeException e) {
            System.out.println( e); }
    }
}

```

This example defines a subclass of **Exception** called **AgeException**. This subclass is quite simple: it has only a constructor and an overloaded **toString()** method that displays the value of the exception. The **ExceptionDemo** class invokes **getDetails()** method of **emp** class. The **getDetails** method throws **AgeException** object if age is less than 16. The **main()** method sets up an exception handler for **AgeException**, then calls **getDetails()**.

The output generated is as follows:

```

Enter your name:
Suman
Enter your age:
12
12 is an invalid age

```

9.8: User defined Exceptions

Demo: User Specific Exception

- Execute UserException.java program

served 32

```
class ApplicationException extends Exception {
    private int detail;
    ApplicationException(int a) { detail = a; }
    ApplicationException(String args) { super(args); }
    public String toString() {return "ApplicationException["+detail+"]"; }
}
class UserException {
    static void compute(int a) throws ApplicationException {
        System.out.println("called compute("+a+")");
        if (a>10) throw new ApplicationException(a);
        System.out.println("Normal Exit");
    }
    public static void main(String arg[]) {
        try {
            compute(1);
            compute(20);
        } catch (ApplicationException e) { System.out.println("caught "+e); }
    }
}
```

Output:

```
called compute(1)
Normal Exit
called compute(20)
caught ApplicationException[20]
```


9.8: User defined Exceptions

Lab : Exception

- Lab 6: Exception Handling



9.9: Best Practices on Exception Handling

The Best Practices

- Avoid empty catch blocks.
- Avoid throwing and catching a generic exception class.
- Pass all the pertinent data to exceptions.
- Use the finally block to release the resources

34

The Best Practices:

Avoid empty catch blocks:

Most contend that it is usually not preferable to have an empty catch block. When the exception occurs, nothing happens, and the program fails for unknown reasons.

For example, if a problem with user input is detected and an exception is thrown as a result, then merely informing the user of the problem might be all that is required. For example, a message might read : Age must be in range 0..120

Avoid throwing and catching generic exception class:

If the exception is known then, catch a specific exception class.

In the Throws clause of a method header, be as specific as possible. Do not group together related exceptions in a generic exception class - that would represent a loss of possibly important information.

Use the finally block to release the resources like a database connection, closing a file or socket connection, and others.

This prevents resource leaks even if an exception occurs. The Finally block always execute irrespective of whether exception is thrown or not.

9.9: Best Practices on Exception Handling

Best Practices

- Avoid throwing unnecessary exceptions.
- Finalize method is not reliable.
- Exception thrown by finalizers are ignored.
- While using method calls, always handle the exceptions in the method where they occur.
- Do not use loops for exception handling.

Best Practices:

Avoid Throwing Unnecessary Exceptions: Creating and throwing an exception is a time-consuming process, so you should avoid throwing exceptions when it's not necessary to do so. For example, you might define a method such as the one shown here that returns an object from a list until there are no more in the list. If the method is invoked when there are no objects remaining in the list, a `NoMoreObjectsException` is thrown:

```
public Object getNextObject() throws NoMoreObjectsException {  
    // ...  
}
```

Instead of throwing a `NoMoreObjectsException` when there are no objects remaining in the list, you might change this method so that it returns a null value, which will allow your code to execute more quickly when that occurs.

Finalize method is not reliable: Since the Garbage collector is JVM specific and it is not always sure when the garbage collector is invoked, it is thus not sure when the finalize method is executed. So, avoid releasing resources which are important and are to be referred again in the code using the finalize method.

9.9: Best Practices on Exception Handling

Best Practices

- When deciding on checked exceptions vs. unchecked exceptions, ask yourself, "What action can the client code take when the exception occurs?"

36

Best Practices:

Always remember that exceptions thrown by finalizers are ignored.

While using method calls, always handle the exceptions in the method where they occur. Do not allow them to propagate to the calling method unless it is specifically required. It is efficient to handle them locally since allowing them to propagate to the calling method takes more execution time.

Do not use Exception handling in loops. It is better to place loops inside the try or catch block than vice versa.

Using Check or Uncheck Exception:

If the client can take some alternate action to recover from the exception, make it a checked exception. If the client cannot do anything useful, then make the exception unchecked. Being useful means, able to take steps to recover from the exception and not just log the exception. To summarize:

Client's reaction when exception happens	Exception type
1.Client code cannot do anything	1.Make it an unchecked exception
2.Client code will take some useful recovery	2. Make it a checked action based on information in exception

Summary

- Exceptions are powerful error handling mechanisms.
- A program can catch exceptions by using a combination of the try, catch, and finally blocks:
 - The try block identifies a block of code in which an exception can occur.
 - The catch block identifies a block of code, known as an exception handler.
 - The finally block identifies a block of code that is guaranteed to execute.
 - Try-with-resources
 - Multi-catch blocks
- Throw is used to throw an exception by the user.



est 37

Add the notes here.

Review – Match the Following format

1. CheckedException	A. Compulsory to use if a method throws a checked exception and doesn't handle it
2. finally	B. Inherited from RuntimeException
3. throws	C. Can have any number of catch blocks
4. Unchecked Exception	D. Used to avoid "resource leak"
5. try	E. Inherited from Exception

