
Core Java 8
Flow control And Exception Handling
Capgemini

Lesson Objectives

After completing this lesson, participants will be able to:

- Understand Basic Java Language constructs like:
  - Write Java programs using control structures
  - Exceptions
  - User Defined Exceptions
- Best Practices

Lesson Outline:

3.1: Keywords
3.2: Primitive Data Types
3.3: Operators and Assignments
3.4: Variables and Literals
3.5: Flow Control: Java's Control Statements
3.6: Best Practices

Control Statements

Use control flow statements to:

- Conditionally execute statements
- Repeatedly execute a block of statements
- Change the normal, sequential flow of control

Categorized into two types:

- Selection Statements
- Iteration Statements

Java being a programming language, offers a number of programming constructs for decision making and looping.

Selection Statements

Allows programs to choose between alternate actions on execution.

"if" used for conditional branch:

```
if (condition) statement1;
else statement2;
```

"switch" used as an alternative to multiple "if's":

Expression can be of String type!

```
switch(expression){
    case value1:    //statement sequence
                    break;
    case value2:    //statement sequence
                    break; …
    default:                //default statement sequence
}
```

If Statement:

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths.

Each statement may be a single statement or a compound statement enclosed in curly braces. The condition is any expression that returns a boolean value. The else clause is optional.

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
if(a < b) a = 0;
else b = 0;
```

switch case : an example

```
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<=4; i++)
            switch(i) {
        case 0:
              System.out.println("i is zero."); break;
        case 1:
               System.out.println("i is one."); break;
        case 2:
               System.out.println("i is two."); break;
        case 3:
               System.out.println("i is three."); break;
        default:
               System.out.println("i is greater than 3.");
             }
        }}
```

Output:
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.

Switch – Case:
The *switch* statement is Java's multi-way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
The switch-expression must evaluate to any of the following type:

- byte
- short
- char
- int
- enum
- **String**.

As such, it often provides a better alternative than a large series of *if-else-if* statements.

Iteration Statements

Allow a block of statements to execute repeatedly

- While Loop: Enters the loop if the condition is true

```
while (condition)
{  //body of loop
 }
```

- Do – While Loop: Loop executes at least once even if the condition is false

```
do
{ //body of the loop
}  while (condition)
```

while statement:

The body of the loop is executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. Example:

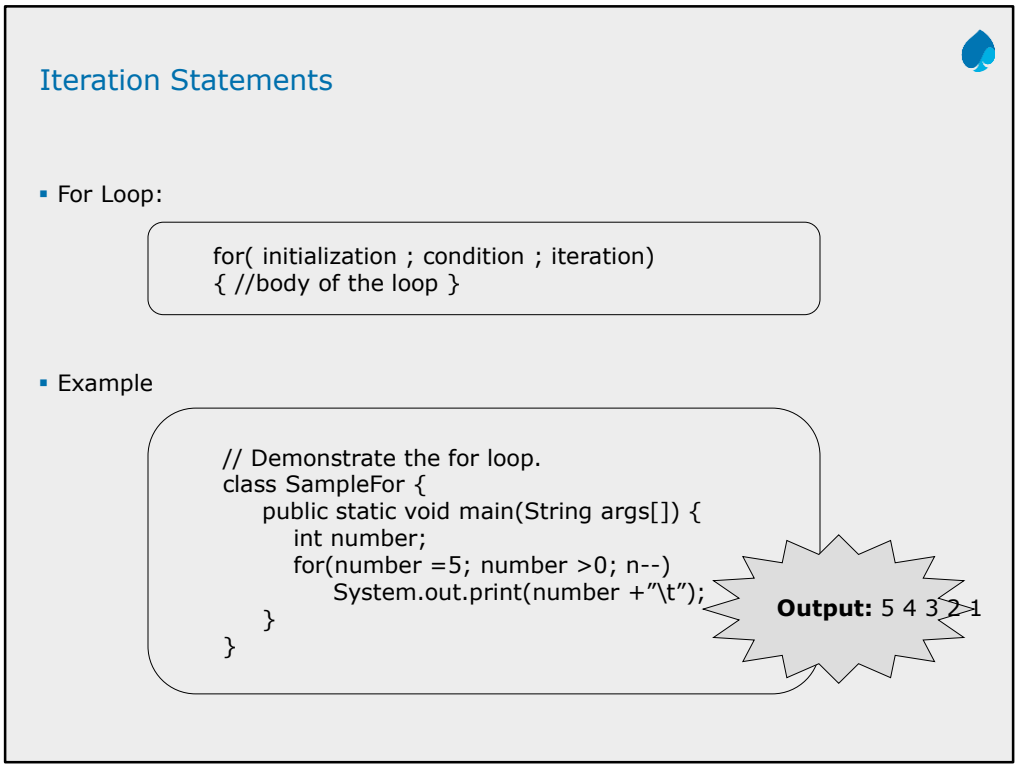
```
class Samplewhile {
        public static void main(String args[]) {
            int n = 5;
            while(n > 0) {
               System.out.print(n+"\t");
                n--;
    }}}
```

Output: 5 4 3 2 1

The while loop evaluates its conditional expression at the top of the loop. Hence, if the condition is false to begin with, the body of the loop will not execute even once.

do – while loop:

This construct executes the body of a **while** loop at least once, even if the conditional expression is false to begin with. The termination expression is tested at the **end** of the loop rather than at the beginning.



for loop:

When the **for** loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable,* which acts as a counter that controls the loop. The initialization expression is only executed once. Next, *condition* is evaluated. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed, else the loop terminates. Next, the *incremental* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Demo

Data types in Java

Switch Statement using String as expression

Add the notes here.

Best practices: Iteration Statements

Always use an int data type as the loop index variable whenever possible

Use for-each liberally

Switch case statement

Terminating conditions should be against 0

Loop invariant code motion

E.g If you call length() in a tight loop, there can be a performance hit.

Always use an int data type as the loop index variable whenever possible.
It is efficient when compared to using byte or short data types. This is because when we use byte or short data type as the loop index variable they involve implicit type cast to int data type.

Use for-each liberally.
The for-each loop is used with both collections and arrays. It is intended to simplify the most common form of iteration, where the iterator or index is used solely for iteration, and not for any other kind of operation, such as removing or editing an item in the collection or array.

When there is a choice, the for-each loop should be preferred over the for loop, since it increases legibility.

Switch Case statement.
One peculiar fact about switch statements is that code consisting of case with consecutive constants like 0,1,2 are faster than with case with constants like 2, 6, 7, 14, etc. This is because in the former switching between options requires less offset and it takes only 16 bytes. But in the later the offset is higher and it might take 32 or more bytes.

Using Break and Continue

It is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

In such cases, break and continue statements are used.

break statement:

The break statement terminates the loop, whereas continue statement forces the next iteration of the loop.

Continue statement :

The continue statement skips statements after it inside the loop. Its syntax is:

```
continue;
```

Unlabeled Statements

Unlabeled continue :

```
public static void main(String[] args) {
    String[] listOfNames = { "Ravi", "Soma",
            "null", "Colin", "Harry", "null",
            "Smith" };

        for (int i = 0; i < listOfNames.length; i++) {
           if (listOfNames[i].equals("null"))
              continue;
           System.out.println(listOfNames[i]);
        }
```

Unlabeled break :

```
public static void main(String[] args) {
    int i = 0;

      for (i = 9999; i <= 99999; i++) {
         if (i % 397 == 0)
         break;
      }
      System.out.println("First number "
         + "divisible by 397 between "
         + "9999 and 99999 is = " + i);
```

Labeled Statements

Java labeled blocks are logically similar to to goto statements in C/C++.

The labeled *break* and *continue* statements are the only way to write statements similar to *goto*. Java does not support *goto* statements. It is good programming practice to use fewer or no *break* and *continue* statements in program code to leverage readability. It is almost always possible to design program logic in a manner never to use *break* and *continue* statements. Too many labels of nested controls can be difficult to read. Therefore, it is always better to avoid such code unless there is no other way.

Labeled Statement

Labeled continue :

```
public static void main(String[] args) {
    start: for (int i = 0; i < 5; i++) {
         System.out.println();
         for (int j = 0; j < 10; j++) {
            System.out.print("#");
            if (j >= i)
               continue start;
         }
         System.out.println("This will never"
            + " be printed");
    }
```

Labeled break :

```
public static void main(String[] args) {
     int counter = 0;
      start: {
         for (int i = 0; i <= 10; i++) {
            for (int j = 0; j <= 10; j++) {
               if (i == 5)
                  break start;
            }
            counter++;
         }
      }
      System.out.println(counter);
```

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 disrupt its normal course.

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.

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.

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.

Handling Exceptions Using 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.

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…

Catching Exception Using try and catch

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.

.

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!

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 }
}
```

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.

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

Throwing an Exception:

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

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);
       }
} }
```

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.

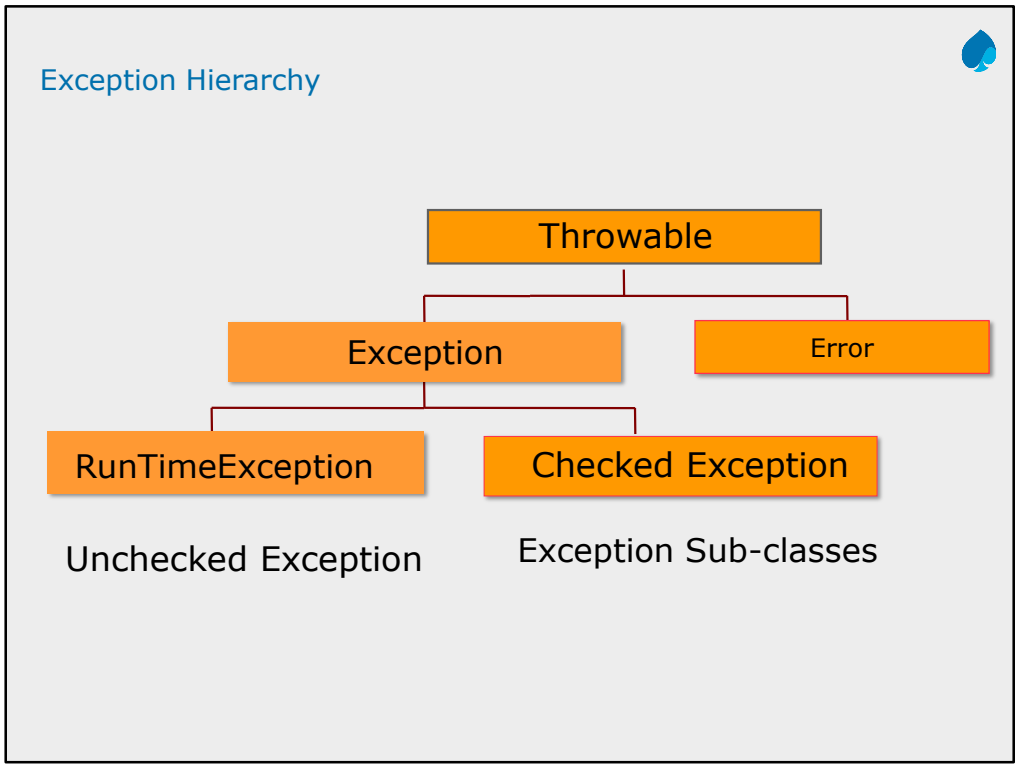
User Specific Exception (Contd.):

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



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.

Exception Matching

When an exception is thrown, the exception handling system looks through the “nearest” handlers in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs.

Matching an exception doesn’t require a perfect match between the exception and its handler. A derived-class object will match a handler for the base class

Exception Matching -Example

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}
```

Error!

You must catch subclasses before their ancestors

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.
```

Rethrowing the Same Exception

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");
    } } }
```

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.

Summary

In this lesson you have learnt:

- Flow Control: Java's Control Statements
- Exception Handling
- Best Practices

Review Question :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