

# Core Java 8

## Lesson 05 : Flow control And Exception Handling



# 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





# 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



## 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”:

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

**Expression can be  
of String type!**



## 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.



# 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)
```



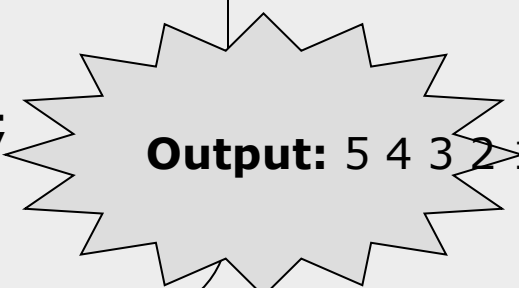
# Iteration Statements

- For Loop:

```
for( initialization ; condition ; iteration)
{ //body of the loop }
```

- Example

```
// Demonstrate the for loop.
class SampleFor {
    public static void main(String args[]) {
        int number;
        for(number =5; number >0; n--)
            System.out.print(number +"\t");
    }
}
```



**Output: 5 4 3 2 1**



# Demo

Data types in Java

Switch Statement using String as expression







## 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.



## 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 `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.





## 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.





# 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.





## 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.  
}
```



# 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 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 }
}
```



# 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

```
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);  
        }  
    }  
}
```



# Defining Exceptions

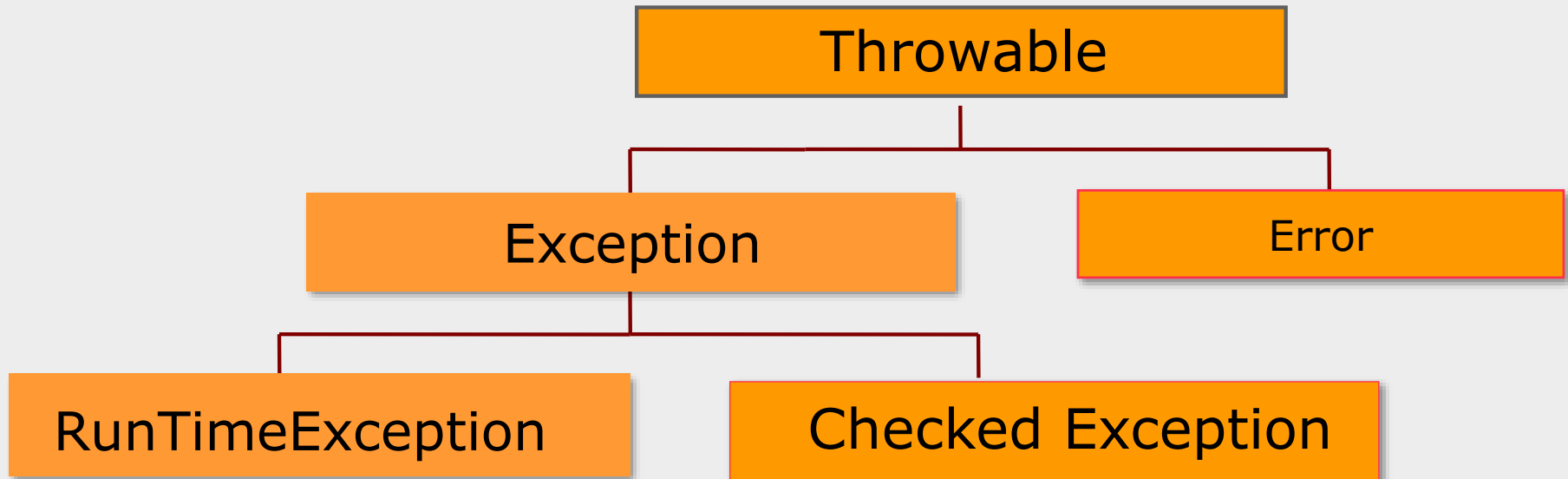
To create exceptions:

- Write a class that extends(indirectly) Throwable.
- Which 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"; } }  
}
```



# Exception Hierarchy



Unchecked Exception

Exception Sub-classes



## 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

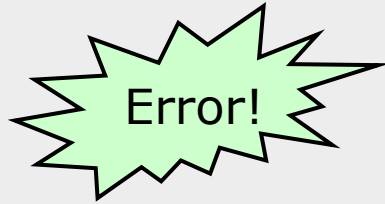




## Exception Matching -Example

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}
```



You must catch subclasses before their ancestors





## 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");  
        } } }
```



# 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

