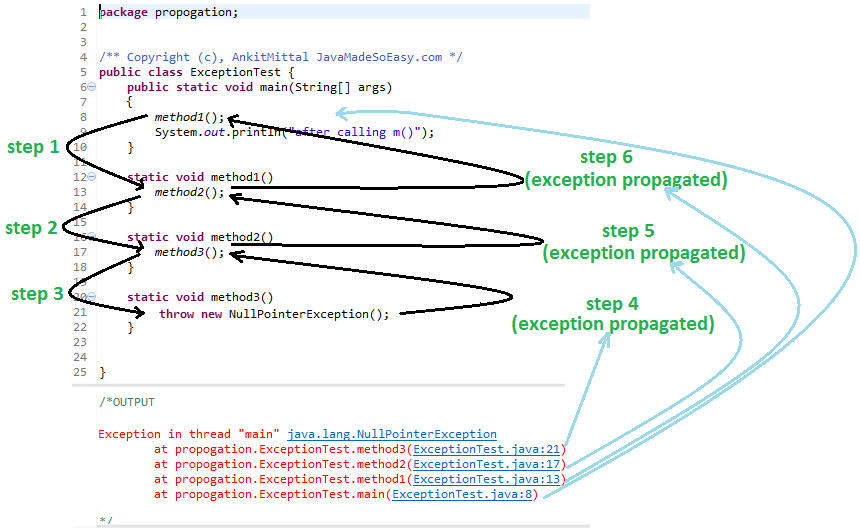
*Exception Propagation*

Whenever methods are called [stack](http://javamadesoeasy.com/2015/01/stacks.html) is formed and an exception is first thrown from the top of the stack and if it is not caught, it starts coming down the stack to previous methods until it is not caught.

If exception remains uncaught even after reaching bottom of the stack it is propagated to JVM and program is terminated.

*Propagating* [***unchecked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)*exception (NullPointerException)*

**unchecked** exceptions are **automatically propagated** in java.

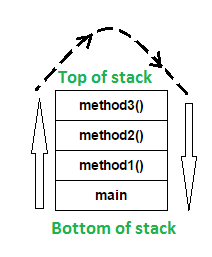


***Now, i’ll be explaining you how unchecked exception*** *was* ***propagated.***

***Let’s see step by step what happened in above program***

* ***JVM*** *called main method*
* ***step 1*** *- main called method1()*
* ***step 2*** *- method1 called method2()*
* ***step 3*** *- method2 called method3()*
* ***step 4*** *- method3* ***automatically propagated exception*** *to method2() [because, unchecked exceptions are propagated* ***automatically****]*
* ***step 5*** *- method2* ***automatically propagated exception*** *to method1() [because, unchecked exceptions are propagated* ***automatically****]*
* ***step 6*** *- method2* ***automatically propagated exception*** *to main() [because, unchecked exceptions are propagated* ***automatically****]*
* *main()* ***automatically propagated exception*** *to* ***JVM*** *[because, unchecked exceptions are propagated* ***automatically****]*

*Let's see how* [***stack***](http://javamadesoeasy.com/2015/01/stacks.html)*of methods is formed*

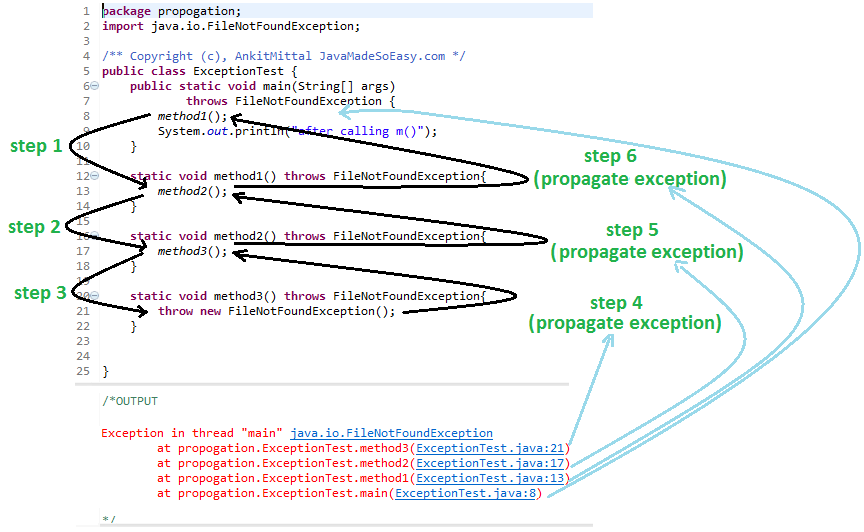


In the above program, stack is formed and an exception is first thrown from the top of the stack [ **method3()** ] and it remains uncaught there, and starts coming down the stack to previous methods to **method2()**,then to **method1()**,than to **main()** and it remains uncaught throughout.

exception remains uncaught even after reaching bottom of the stack [ **main()** ] so it is propagated to JVM and ultimately program is terminated by throwing exception [ as shown in output ].

*Propagating* [***checked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)*exception (FileNotFoundException) using throws keyword*

For **propagating checked** exceptions method must throw exception by using [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html)keyword.



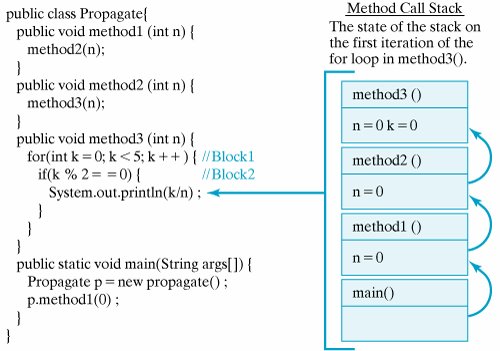
***Now, i’ll be explaining you how checked exception*** *was* ***propagated****.*

***Let’s see step by step what happened in above program***

* ***JVM*** *called main method*
* ***step 1*** *- main called method1()*
* ***step 2*** *- method1 called method2()*
* ***step 3*** *- method2 called method3()*
* ***step 4*** *- method3* ***propagated exception*** *to method2() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* ***step 5*** *- method2* ***propagated exception*** *to method1() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* ***step 6*** *- method2* ***propagated exception*** *to main() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* *main()* ***propagated exception*** *to* ***JVM*** *using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*

**Exception Propagation: Searching for a Catch Block**

**The method call stack for the Propagate program. The curved arrows give a trace of the method calls leading to the program's present state.**



# Naming Conventions

It's common to use a variable naming convention to distinguish between fields, arguments, and local variables

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

|  |  |
| --- | --- |
| **Name** | **Convention** |
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | Methods in Java also follow the same lowerCamelCase convention like Objects and variables.should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX\_PRIORITY etc. |

| Java Bitwise and Bit Shift Operators | |
| --- | --- |
| Operator | Description |
| | | [Bitwise OR](https://www.programiz.com/java-programming/bitwise-operators#or) |
| & | [Bitwise AND](https://www.programiz.com/java-programming/bitwise-operators#and) |
| ~ | [Bitwise Complement](https://www.programiz.com/java-programming/bitwise-operators#complement) |
| ^ | [Bitwise XOR](https://www.programiz.com/java-programming/bitwise-operators#xor) |
| << | [Left Shift](https://www.programiz.com/java-programming/bitwise-operators#left-shift) |
| >> | [Right Shift](https://www.programiz.com/java-programming/bitwise-operators#right-shift) |
| >>> | [Unsigned Right Shift](https://www.programiz.com/java-programming/bitwise-operators#unsigned-right-shift) |

## Bitwise OR

Bitwise OR is a binary operator (operates on two operands). It's denoted by |.

The | operator compares corresponding bits of two operands. If either of the bits is 1, it gives 1. If not, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

00001100

| 00011001

\_\_\_\_\_\_\_\_

00011101 = 29 (In decimal)

### Example 1: Bitwise OR

class BitwiseOR {

public static void main(String[] args) {

int number1 = 12, number2 = 25, result;

result = number1 | number2;

System.out.println(result);

}

}

When you run the program, the output will be:

29

## Bitwise AND

Bitwise AND is a binary operator (operates on two operands). It's denoted by &.

The & operator compares corresponding bits of two operands. If both bits are 1, it gives 1. If either of the bits is not 1, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

00001100

& 00011001

\_\_\_\_\_\_\_\_

00001000 = 8 (In decimal)

### Example 2: Bitwise AND

class BitwiseAND {

public static void main(String[] args) {

int number1 = 12, number2 = 25, result;

result = number1 & number2;

System.out.println(result);

}

}

When you run the program, the output will be:

8

## Bitwise Complement

Bitwise complement is an unary operator (works on only one operand). It is denoted by ~.

The ~ operator inverts the bit pattern. It makes every 0 to 1, and every 1 to 0.

35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

\_\_\_\_\_\_\_\_

11011100 = 220 (In decimal)

### Example 3: Bitwise Complement

class Complement {

public static void main(String[] args) {

int number = 35, result;

result = ~number;

System.out.println(result);

}

}

When you run the program, the output will be:

-36

Why are we getting output -36 instead of 220?

It's because the compiler is showing 2's complement of that number; negative notation of the binary number.

For any integer n, 2's complement of n will be -(n+1).

Decimal Binary 2's complement

--------- --------- ---------------------------------------

0 00000000 -(11111111+1) = -00000000 = -0(decimal)

1 00000001 -(11111110+1) = -11111111 = -256(decimal)

12 00001100 -(11110011+1) = -11110100 = -244(decimal)

220 11011100 -(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

## Bitwise XOR

Bitwise XOR is a binary operator (operates on two operands). It's denoted by ^.

The ^ operator compares corresponding bits of two operands. If corresponding bits are different, it gives 1. If corresponding bits are same, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

00001100

| 00011001

\_\_\_\_\_\_\_\_

00010101 = 21 (In decimal)

### Example 4: Bitwise XOR

class Xor {

public static void main(String[] args) {

int number1 = 12, number2 = 25, result;

result = number1 ^ number2;

System.out.println(result);

}

}

When you run the program, the output will be:

21

## Signed Left Shift

The left shift operator << shifts a bit pattern to the left by certain number of specified bits, and zero bits are shifted into the low-order positions.

212 (In binary: 11010100)

212 << 1 evaluates to 424 (In binary: 110101000)

212 << 0 evaluates to 212 (In binary: 11010100)

212 << 4 evaluates to 3392 (In binary: 110101000000)

### Example 5: Signed Left Shift

class LeftShift {

public static void main(String[] args) {

int number = 212, result;

System.out.println(number << 1);

System.out.println(number << 0);

System.out.println(number << 4);

}

}

When you run the program, the output will be:

424

212

3392

## Signed Right Shift

The right shift operator >> shifts a bit pattern to the right by certain number of specified bits.

212 (In binary: 11010100)

212 >> 1 evaluates to 106 (In binary: 01101010)

212 >> 0 evaluates to 212 (In binary: 11010100)

212 >> 8 evaluates to 0 (In binary: 00000000)

If the number is a 2's complement signed number, the sign bit is shifted into the high-order positions.

### Example 6: Signed Right Shift

class RightShift {

public static void main(String[] args) {

int number = 212, result;

System.out.println(number >> 1);

System.out.println(number >> 0);

System.out.println(number >> 8);

}

}

When you run the program, the output will be:

106

212

0

## Unsigned Right Shift

The unsigned right shift operator << shifts zero into the leftmost position.

### Example 7: Signed and UnSigned Right Shift

class RightShift {

public static void main(String[] args) {

int number1 = 5, number2 = -5;

// Signed right shift

System.out.println(number1 >> 1);

// Unsigned right shift

System.out.println(number1 >>> 1);

// Signed right shift

System.out.println(number2 >> 1);

// Unsigned right shift

System.out.println(number2 >>> 1);

}

}

When you run the program, the output will be:

2

2

-3

2147483645

Notice, how signed and unsigned right shift works differently for 2's complement.

The 2's complement of 2147483645 is 3.