Programming in C#. Fundamentals

Lesson 3
Control Flow

Control Flow

Branching
Switching
Iterating
Jumping
Chaining
Error throwing
Error handling

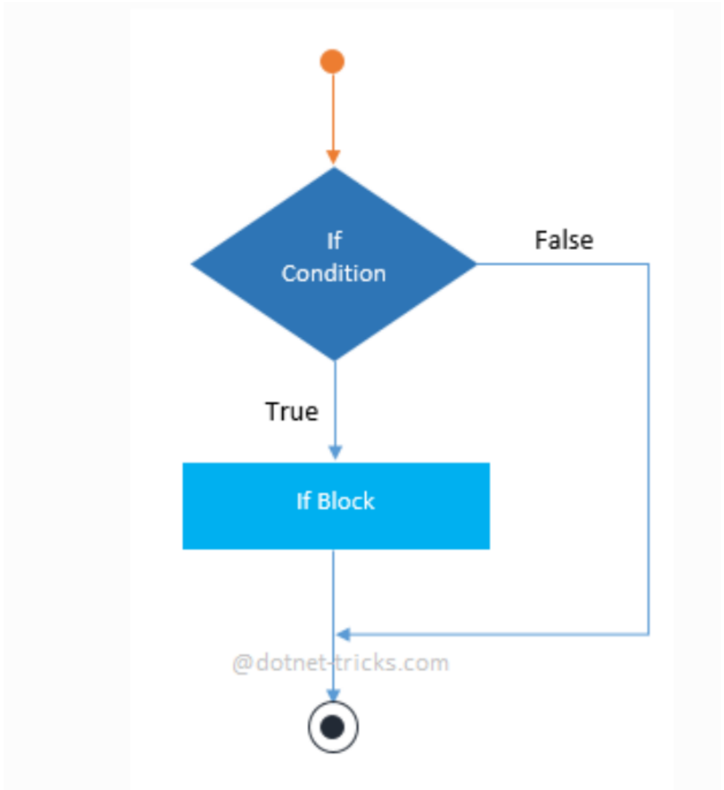
SIGMA Software

Branching

- **If** statement
- **If-Else** statement
- **If-Else-If** statement or ladder
- **Switch** statement

If statement

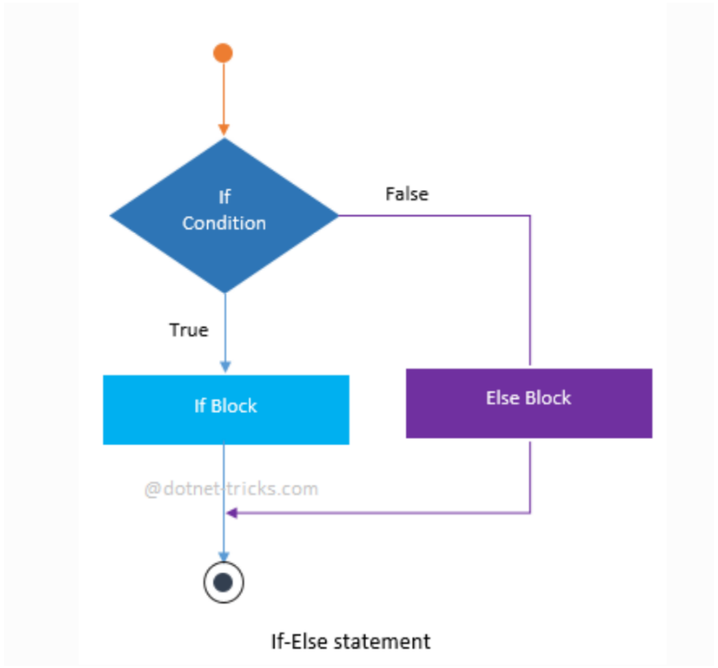
An if statement consists of a boolean expression which is evaluated to a boolean value. If the value is true then if block is executed otherwise next statement(s) would be executed.



If statement

If-Else statement

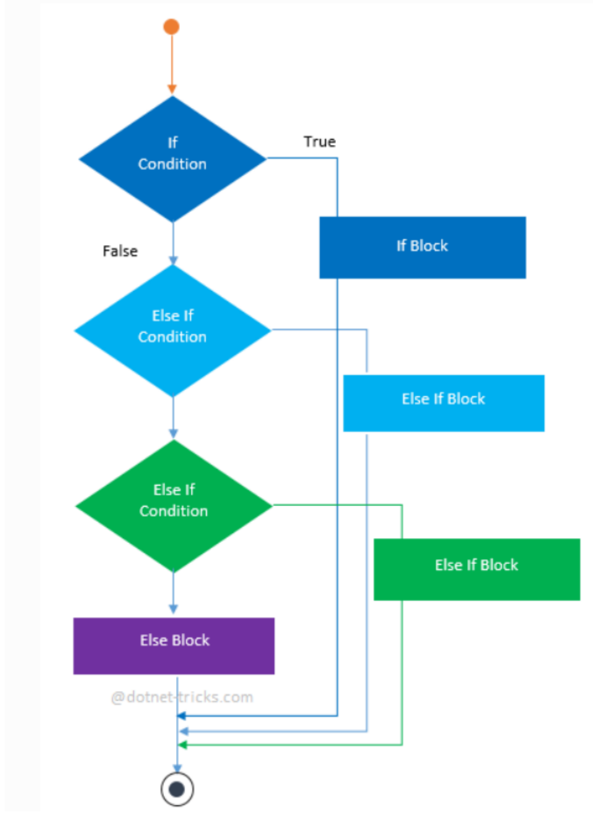
An if-else statement consists of two statements – if statement and else statement. When the expression in an if-statement is evaluated to true then if block is executed otherwise the else block would be executed.



If-Else statement

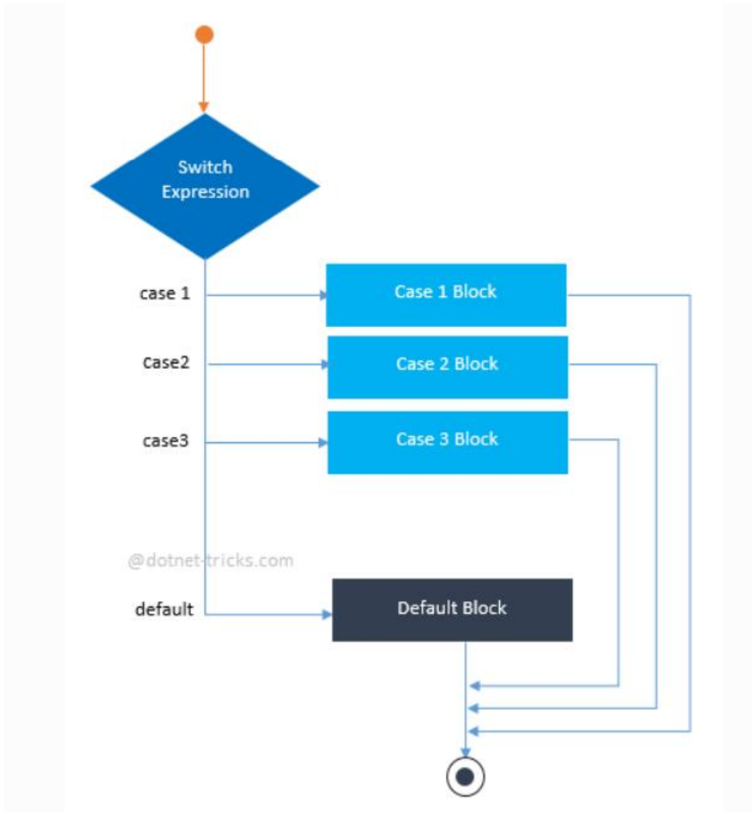
If-Else-If statement

The If-Else-If ladder is a set of statements that is used to test a series of conditions. If the first if statement meet the result then code within the if block executes. If not, control passes to the else statement, which contains a second "if" statement. If second one meet the result then code within the if block executes.



Switch statement

Switch statement acts as a substitute for long If-Else-If ladder that is used to test a series of conditions. A switch statement contains one or more case labels which are tested against the switch expression.

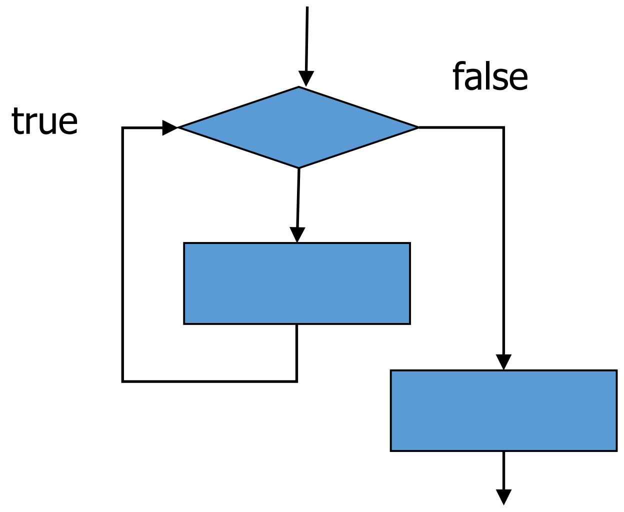


Loops

- Repeated execution of one or more statements until a terminating condition occurs
- Pre-test and post-test loops

Types of loops:

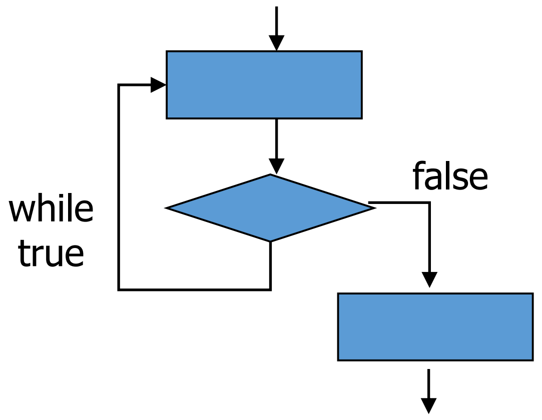
- *Pre-test loops*
  - while
  - for
  - foreach
- *Post-test loop*
  - do…while



While

Used to repeat a portion of based on
the evaluation of a Boolean expression

The Boolean expression is checked before the loop body is executed

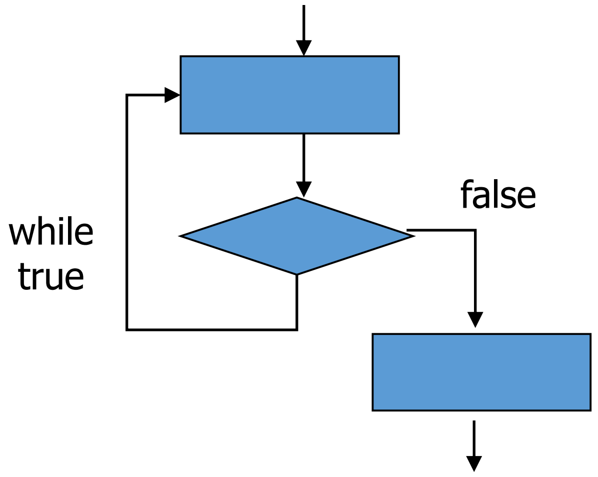


```
while (boolean_expression) {
  Statement_1;
  Statement_2;
  . . .
  Statement_Last;
}
```

Do-while

Used to execute a portion of, and then repeat it based on the evaluation of a Boolean expression

The loop body is executed at least once



```
do {
   Statement_1;
   Statement_2;
   . . .
   Statement_Last;
}  while (boolean_expression);
```

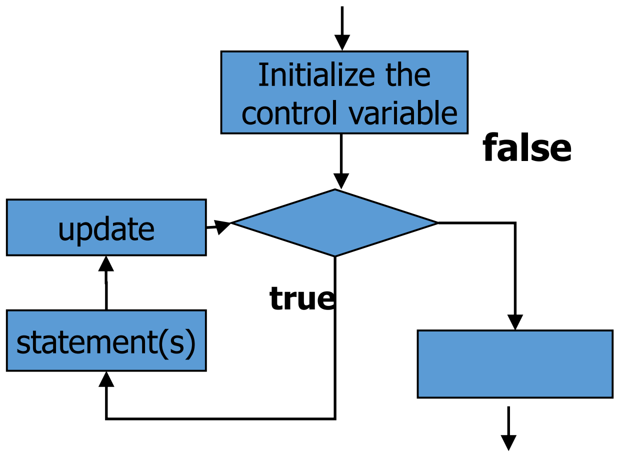
For

Used to step through an integer variable in equal increments

It begins with the keyword for,

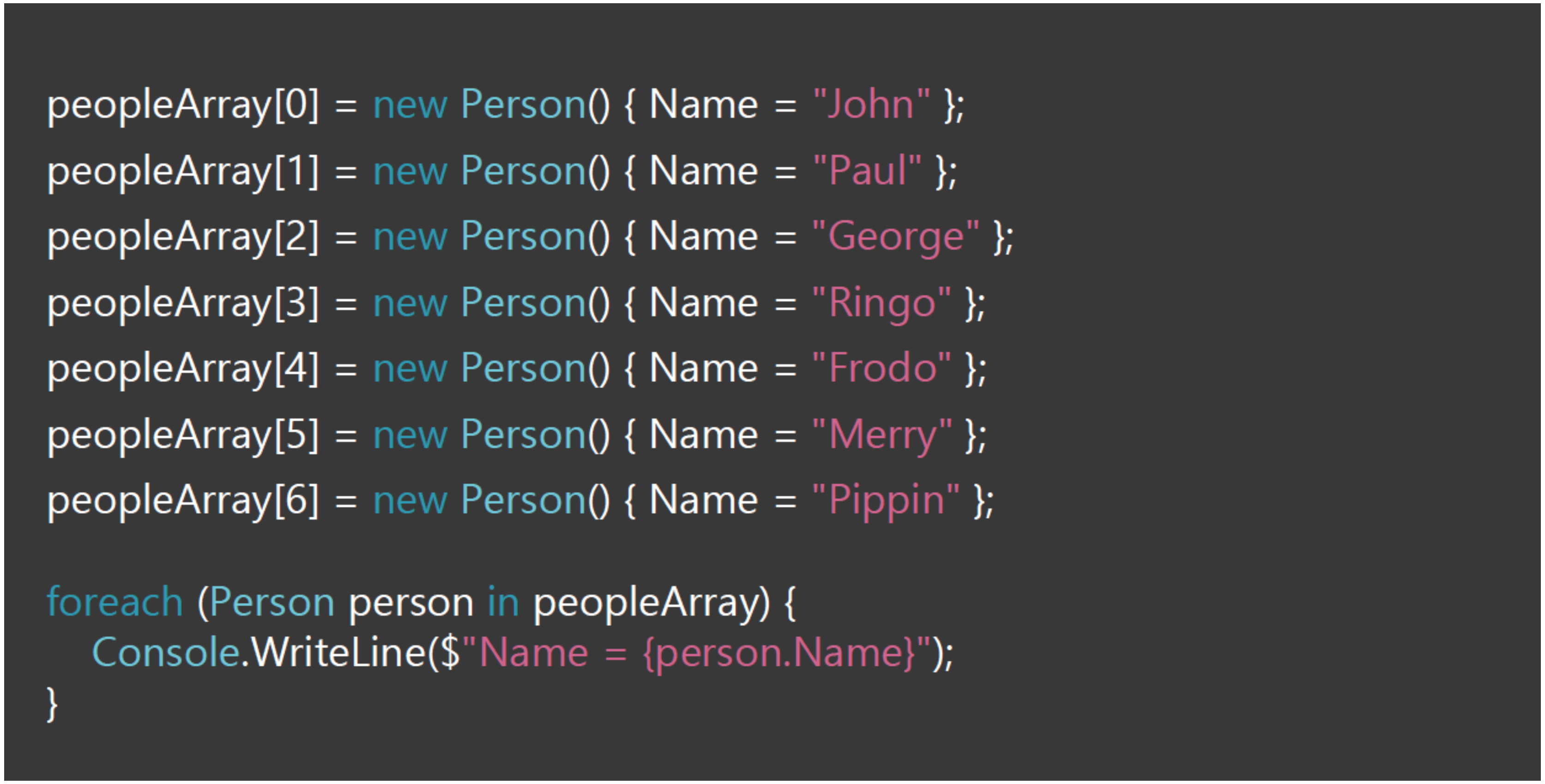
followed by three expressions in parentheses that describe

what to do with one or more controlling variables



```
for (initialization; boolean_expression; update)
{
   Statements_Body;
}
```

Foreach



```
peopleArray[0] = new Person() { Name = "John" };
peopleArray[1] = new Person() { Name = "Paul" };
peopleArray[2] = new Person() { Name = "George" };
peopleArray[3] = new Person() { Name = "Ringo" };
peopleArray[4] = new Person() { Name = "Frodo" };
peopleArray[5] = new Person() { Name = "Merry" };
peopleArray[6] = new Person() { Name = "Pippin" };

foreach (Person person in peopleArray) {
    Console.WriteLine($"Name = {person.Name}");
}
```

Break & continue

The break statement terminates the closest enclosing loop or switch statement in which it appears.

Control is passed to the statement that follows the terminated loop (or switch), if any.

```
for( int i = 0; i < row; i ++) {
   for (int j = 0; j < row; j++) {
      if ( i+j >= row )
        break;
      Console.Write("*");
   }
   Console.WriteLine();
 }
```

For row=3

```
***
**
*
```

```
for (int i= 1; i <= row; i++ ) {
  if( i % 2 == 0)
    continue;   // Go back to for
  Console.WriteLine("i = " +  i );
}
```

```
i = 1
i = 3
```

Return

The return statement terminates execution of the method in which it appears and returns control to the calling method.

```
public static void returnMethod(int row) {
  for (int i = 1; i <= row; i++) {
    if (i % 2 == 0)
      return;
    Console.WriteLine("i = " + i);
  }
}
```

i	Loop test	outcome
1	1<=4	Prints i=1
2	2<=4	Returns

Using

Defines a scope, outside of which an object or objects will be disposed of.

- It is usually best to release limited resources such as file handles and network connections as quickly as possible.

```
using (Font font1 = new Font("Arial", 10.0f)) {
}
```

Exceptions

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

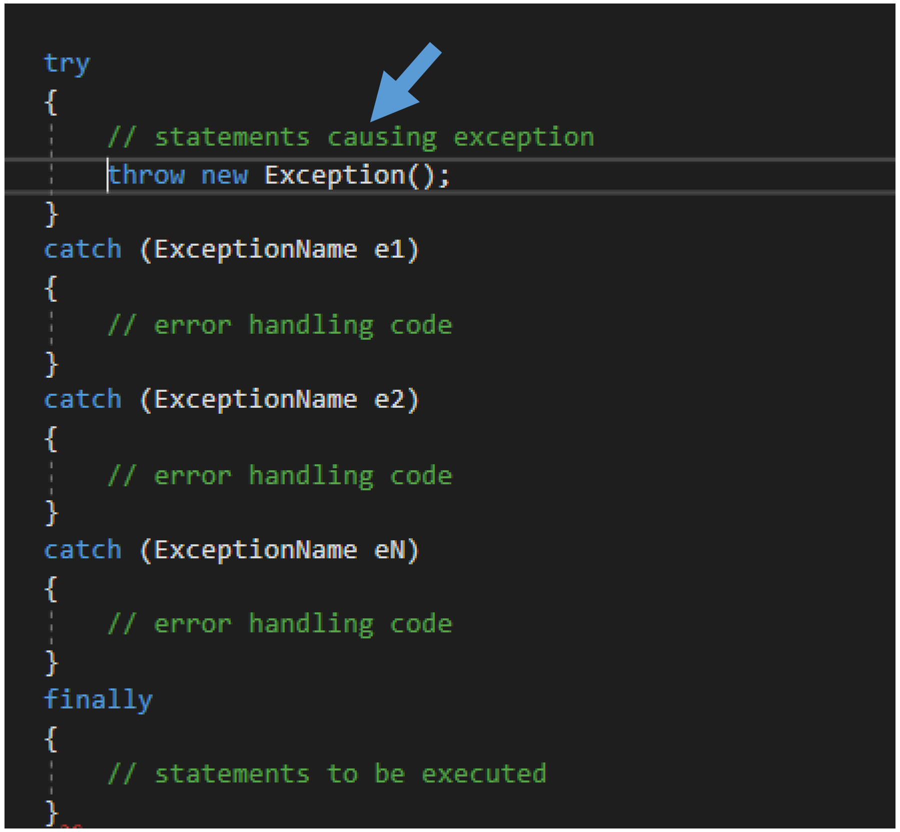
Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: try, catch, finally, and throw.

Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from **the System.Exception** class. Some of the exception classes derived from the **System.Exception** class are the **System.ApplicationException** and **System.SystemException** classes.

Throw statement

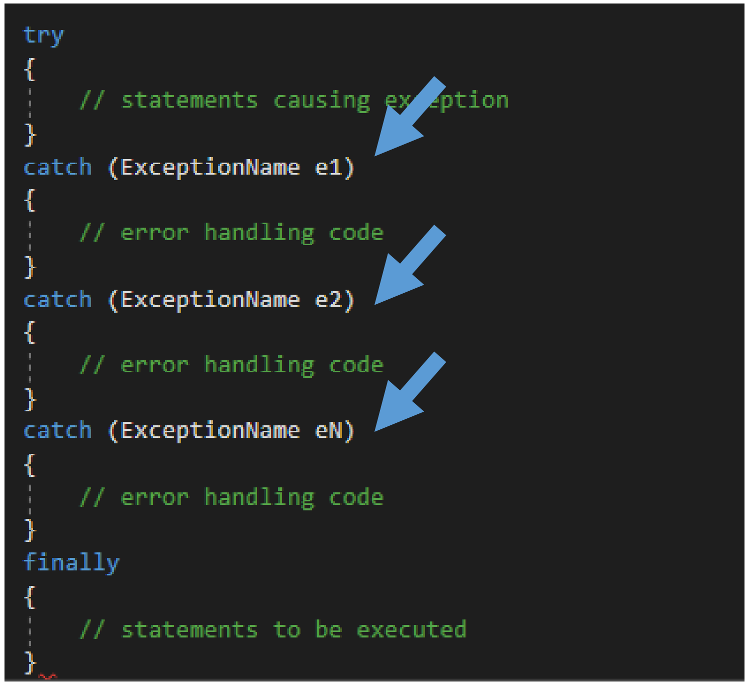
A program throws an exception when a problem shows up. This is done using a throw keyword.



```
try
{
    // statements causing exception
    throw new Exception();
}
catch (ExceptionName e1)
{
    // error handling code
}
catch (ExceptionName e2)
{
    // error handling code
}
catch (ExceptionName eN)
{
    // error handling code
}
finally
{
    // statements to be executed
}
```

Catching an Exception

A **try** block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.

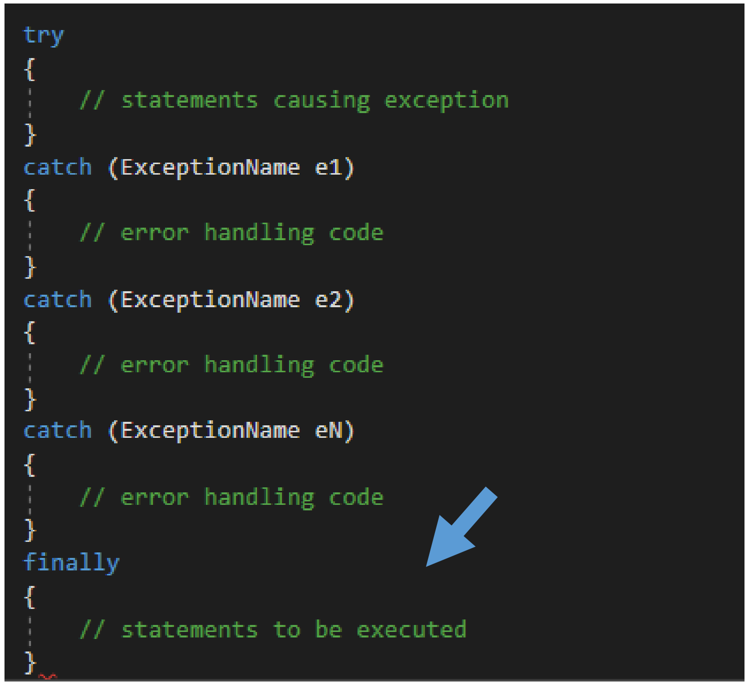


```
try
{
    // statements causing exception
}
catch (ExceptionName e1)
{
    // error handling code
}
catch (ExceptionName e2)
{
    // error handling code
}
catch (ExceptionName eN)
{
    // error handling code
}
finally
{
    // statements to be executed
}
```

catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

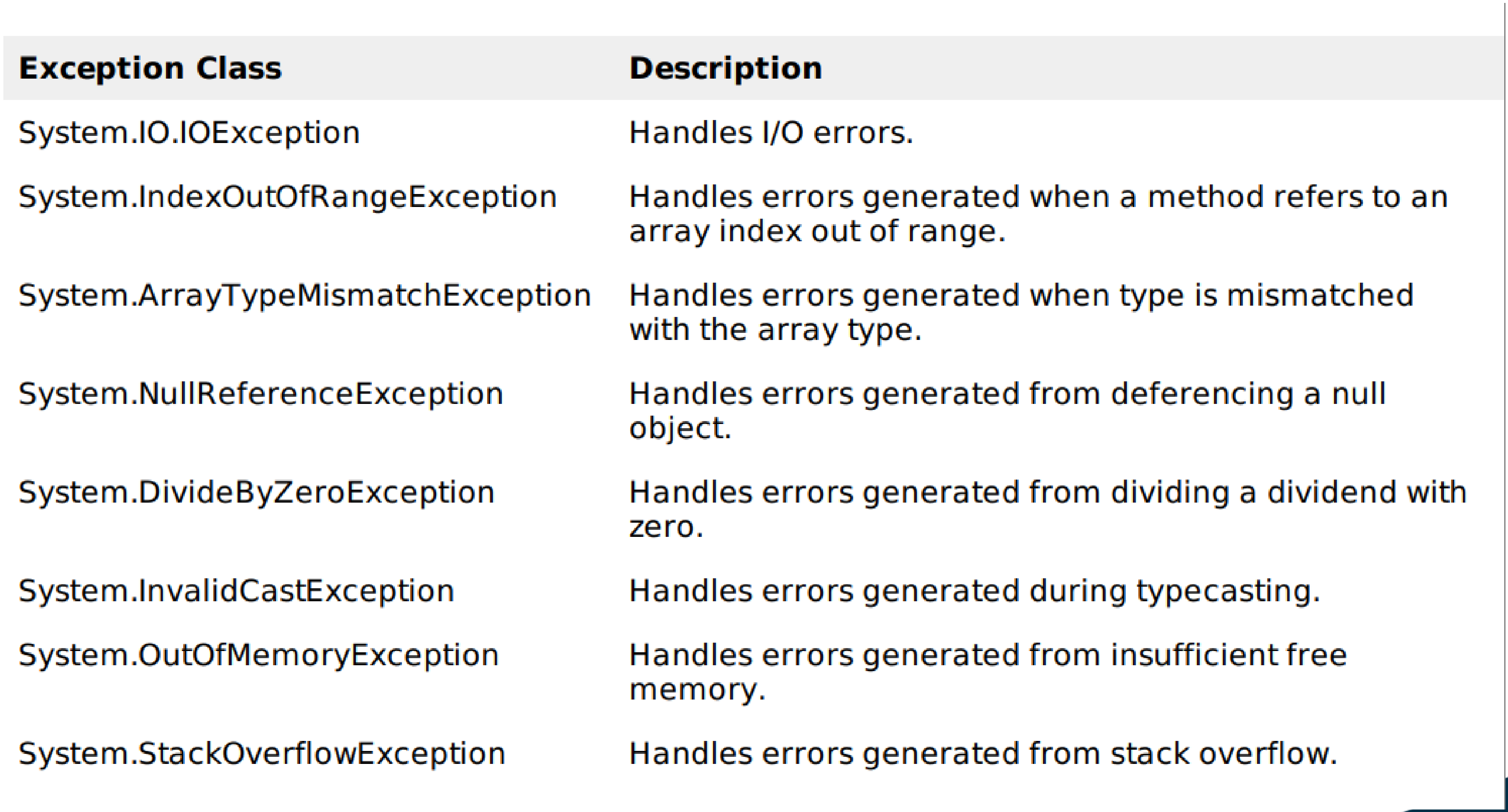
Finally block

The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.



```
try
{
    // statements causing exception
}
catch (ExceptionName e1)
{
    // error handling code
}
catch (ExceptionName e2)
{
    // error handling code
}
catch (ExceptionName eN)
{
    // error handling code
}
finally
{
    // statements to be executed
}
```

Library Exceptions



Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

Q & A

Practice
Lesson 3

Home work