Control Flow

C# Programming

In this chapter we will learn ...

- Controlling execution flow
- if, for, foreach, switch and while statements
- How to handle exceptions

Question

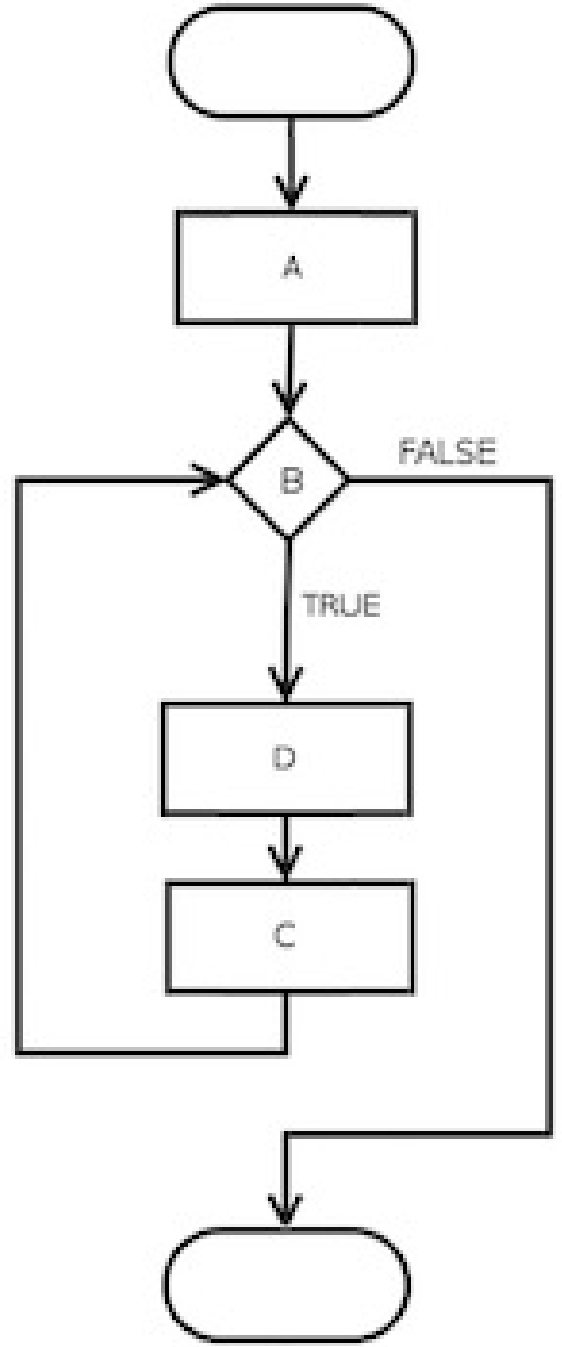
Name some control flow statements

What happens next?

Every programming language has a way to

- Make decisions about which code to execute
  - And NOT execute
- Keep executing the same code until complete/finished
- Process the contents of an array or list
- Just run some code x number of times

Control what happens after an error occurs



Control Flow Statements

Main Control Flow statements in C# are:

Statement	Purpose
if, else if, else	Controls what executes next
while, do while	Executes until a set state is reached
foreach	Processes the contents of a collection
for	Executes code a set number of times
try, catch	Handles exceptions

if statement

Purpose

- To selectively determine which line(s) of code to execute next

Basic Syntax

```
if ( <boolean expression(s) evaluates to true> )
{
    // Code to execute
}
```

- Code inside if may or may not execute
    - Depending on current state

Simple if - example

```
int age = 21;
decimal salary = 60000;
decimal bonus = 0;

if (age >= 21 && salary < 100000) // Age not relevant! Convenient example :)
{
    bonus = salary * 0.05m;
}

Console.WriteLine($"Bonus is {bonus:c}")
```

Try expressing the rule in words ...

What happens if the rule fails?

The else clause

- When the if clause evaluates to false the else provides a default action

```
...

if (age >= 21 && salary < 100000) {
    bonus = salary * 0.05m;
}
else {
    bonus = salary * 0.02m;
}
Console.WriteLine($"Bonus is {bonus:c}")
```

- Only one else clause is allowed and must be placed at the end

But what if bonus rules were more complex?

Provide multiple alternative if statements using else if

```
if (age >= 21 && salary < 100000) {
    bonus = salary * 0.05m;
}
else if (age < 21 and salary < 100000) {
    bonus = 0.08m;
}
else {
    bonus = salary * 0.02m;
}
```

- Multiple else if clauses are allowed

Single statement if

if can be written without braces { } - but there's a problem ...

```
int age = 21;
decimal salary = 60000;
decimal bonus;

if (age >= 21 && salary < 100000)
    bonus = salary * 0.05m;
    Console.WriteLine($"Bonus is {bonus:c}"); // ERROR! bonus is uninitialized
```

What's the problem?

if statements without braces- GOTCHA

In a single statement if

- Only the first statement is guarded by the if clause
- Subsequent statements sit outside of the if logic

So ...

- Console.WriteLine line will execute regardless of the values of age and salary

```
if (age >= 21 && salary < 100000)
    bonus = salary * 0.05m;
    Console.WriteLine($"Bonus is {bonus:c}"); // ERROR! bonus is uninitialized
```

Ternary Operator

Acts as a single line if statement

- Facilitates selectively initializing variables

Syntax

```
<variable> = <boolean expression> ? <value to assign if true> : <value to assign of false>
```

Made up of

- Boolean expression
- ? - Value/Variable to assign if to variable if expression is true
- : - Value/Variable to assign if to variable if expression is false

Ternary Operator - Example

Instead of

```
if (age >= 21 && salary < 100000)
    bonus = salary * 0.05m;
else
    bonus = salary * 0.02m;
```

We could use a ternary operator and write

```
decimal bonus =  (age >= 21 && salary < 100000) ? salary * 0.05m : salary * 0.02m;
```

for loop

Purpose

- To execute code a defined number of times

Basic Syntax

```
for ( <one or more loop counter variables>; <loop exit clause(s)>; <loop counter modifier> )
{
    // Code to execute
}
```

for loop statement - Example

```
for (int count = 0; count < 20; count++)
{
    Console.WriteLine(count);
}
```

for loop - Early Exit

break can be used to exit from the loop

```
for (int count = 0; count < 20; count++)
{
    if (count == 3) break;
    Console.WriteLine(count);
}
```

0

1

2

for loop - Next Iteration

continue moves immediately to the next iteration

- Also works with while

```
for (int count = 0; count < 20; count++) {
    if (count == 3) continue;
    Console.WriteLine(count);
}
```

0

1

2

4

Processing Collections

To process every element in an array we could use

- for loop

```
string[] names = {"Fred", "Mina", "Amy", "Tam", "Dhami"};

for (int index = 0; index < names.Length; index++)
{
    string name = names[index]; // Index into array using the loop counter variable
    Console.WriteLine($"Hello {name}.");
}
```

- for loop allows read/write access to array elements

foreach loop

Purpose

- To access the contents of an array or list
- Can process any type implementing IEnumerable
- OR any type that implements GetEnumerator() public function

Basic Syntax

```
foreach (<type> <variable name> in <IEnumerable object> )
{
    // Code to execute
}
```

foreach loop statement - Example

Let's say "Hello" to every friend in the names array

- An array implements IEnumerable

```
string[] names = {"Fred", "Mina", "Amy", "Tam", "Dhami"};

foreach (string name in names)
{
    Console.WriteLine($"Hello {name}.");
    Console.WriteLine("I hope you have a nice day.");
}
```

foreach - Gotcha

foreach loops are by default readonly

- Below code will give an error

```
string[] names = {"Fred", "Mina", "Amy", "Tam", "Dhami"};

foreach (string name in names)
{
    Console.WriteLine($"Hello {name}.");
    name += "s"; // Compile Error! Can't change foreach loop variable
}
```

foreach writable - Solution - C# 7.3

- Use a foreach by reference
  - Arrays must be cast to Span<T>
  - Declare loop variable as ref

```
Span<string> names = new string[] { "Fred", "Mina", "Amy", "Tam", "Dhami" };
foreach (ref string name in names)
{
    Console.WriteLine($"Hello {name}.");
    name += "s";
}
```

switch statement

Purpose

- To evaluate/match a single/multiple variables for specific processing

Basic Syntax - Simple

```
switch ( <variable to evalue> ) {
    case <value to match>: // One or more case statements allowed
         // Code to execute
         break; // To ensure exit from the switch
    default:
         // Code to execute if match not found
         break;

}
```

switch (Constant Pattern) - Example

Below weather conditions is evaluated against a constant/fixed pattern

```
WeatherConditions conditions = WeatherConditions.Raining;
switch(conditions) {
    case WeatherContitions.Raining:
        Console.WriteLine("It's raining!");
        break;
    case WeatherConditions.Sunny:
        Console.WriteLine("It's a beautiful day!");
        break;
    default:
        Console.WriteLine("What's the weather like?");
        break;
}
```

switch - with fall-through

If you don't use a break statement execution falls through to case below

Here Raining and Sleeting are handled the same way

```
WeatherConditions conditions = WeatherConditions.Raining;
switch(conditions) {
    case WeatherContitions.Raining:
    case WeatherContitions.Sleeting:
        Console.WriteLine("It's a miserable day!");
        break;
    case WeatherConditions.Sunny:
        Console.WriteLine("It's a beautiful day!");
        break;
}
```

switch - Relational pattern

Allows use of standard relational operators

```
byte age = 17;
switch(age) {
    case < 16:
        Console.WriteLine("Child");
        break;
    case < 18:
        Console.WriteLine("Young adult");
        break;
    case >= 18 and < 65:
        Console.WriteLine("Adult");
        break;
}
```

Pattern Matching with type and when clause

Here a nullable byte is cast to a byte if pattern matches

```
byte? enteredAge = null;
switch (enteredAge)
{
    case byte age when age >= 18:
        Console.WriteLine("Adult");
        break;
    default:
        Console.WriteLine("Age Not Supplied");
        break;
}
```

- No match is found because enteredAge is null

Using switch as a function guard

switch statements can be used to guard execution of critical code

```
void DoSomething(int a, int b) {

    switch ((a, b)) {
        case (>5, >10) when b % a == 0:
            Console.WriteLine("All is well!");
            break;
        default:
            Console.WriteLine("Ooops");
            break;
    }
}
```

switch expressions

A switch can be expressed as a lambda expression

- Expression

```
string HowsTheweather(WeatherContitions condition) => condition switch {

    WeatherContitions.Raining => "It's raining!";
    WeatherConditions.Sunny => "It's a beautiful day!";
    _  => "What's the weather like?";

}
```

while statement

Purpose

- To repeatedly execute a block of code while a boolean test evaluates to true

Basic Syntax

```
while ( <boolean expression(s) evaluates to true> )
{
    // Code to execute
}
```

- Note: The code inside the while may never execute!
    - if the boolean expression evaluates to false at the start

while statement - Example

```
int count = 0;
while(count < 20)
{
    Console.WriteLine(count);
    count++;
}
```

while loop keeps executing

- Until count is equal to 20
- Incrementing count leads to loop clause eventually evaluating to false

while statement - Early Termination

Use break to exit loop early

```
int count = 0;
while(count < 20) {
    if (count == 15) break;
    Console.WriteLine(count);
    count++;
}
```

Ensuring Loop code executes once

while loops may never execute code within

- Loop clause may prevent entry into code block

do while inverts the loop

```
int count = 20;
do
{
    Console.WriteLine(count);
    count++;
}while(count < 20);
```

What will happen in above code?

Exceptions are cool

An exception is a report of an unhandled exceptional state

When an exception occurs we have some choices

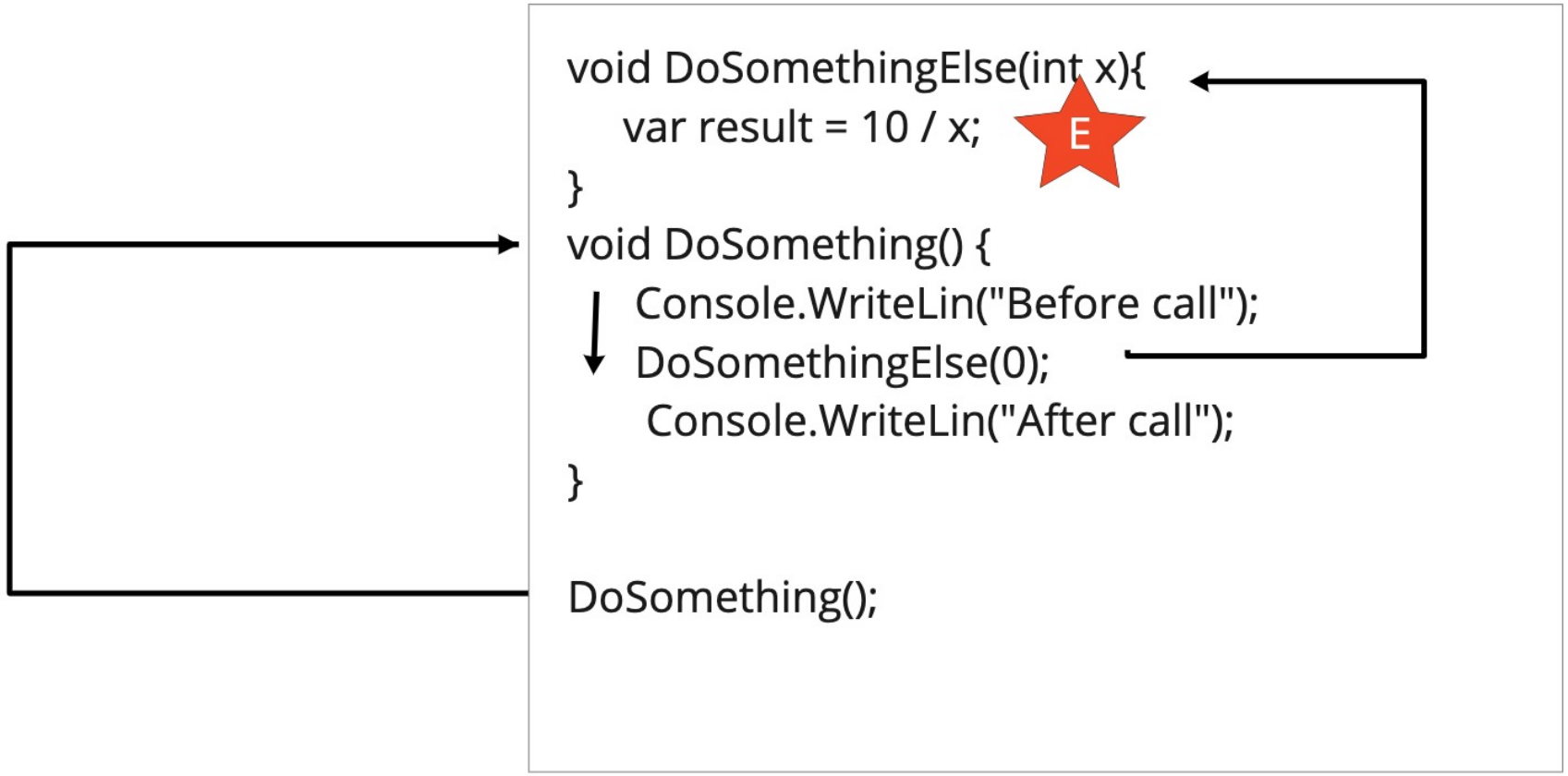
- Should code execution just continue
- Should we at least check what the error is and ...
    - see if there is something we can do about it
    - report it - to the user or to a log
    - should we ignore it?

Exceptions in C#

- Are deliberately fatal to the execution of the app
  - If you don't handle am exception your app/transaction will fail
- Are hierarchical
  - They are passed between functions from child to parent
  - Then recursively up through the call stack

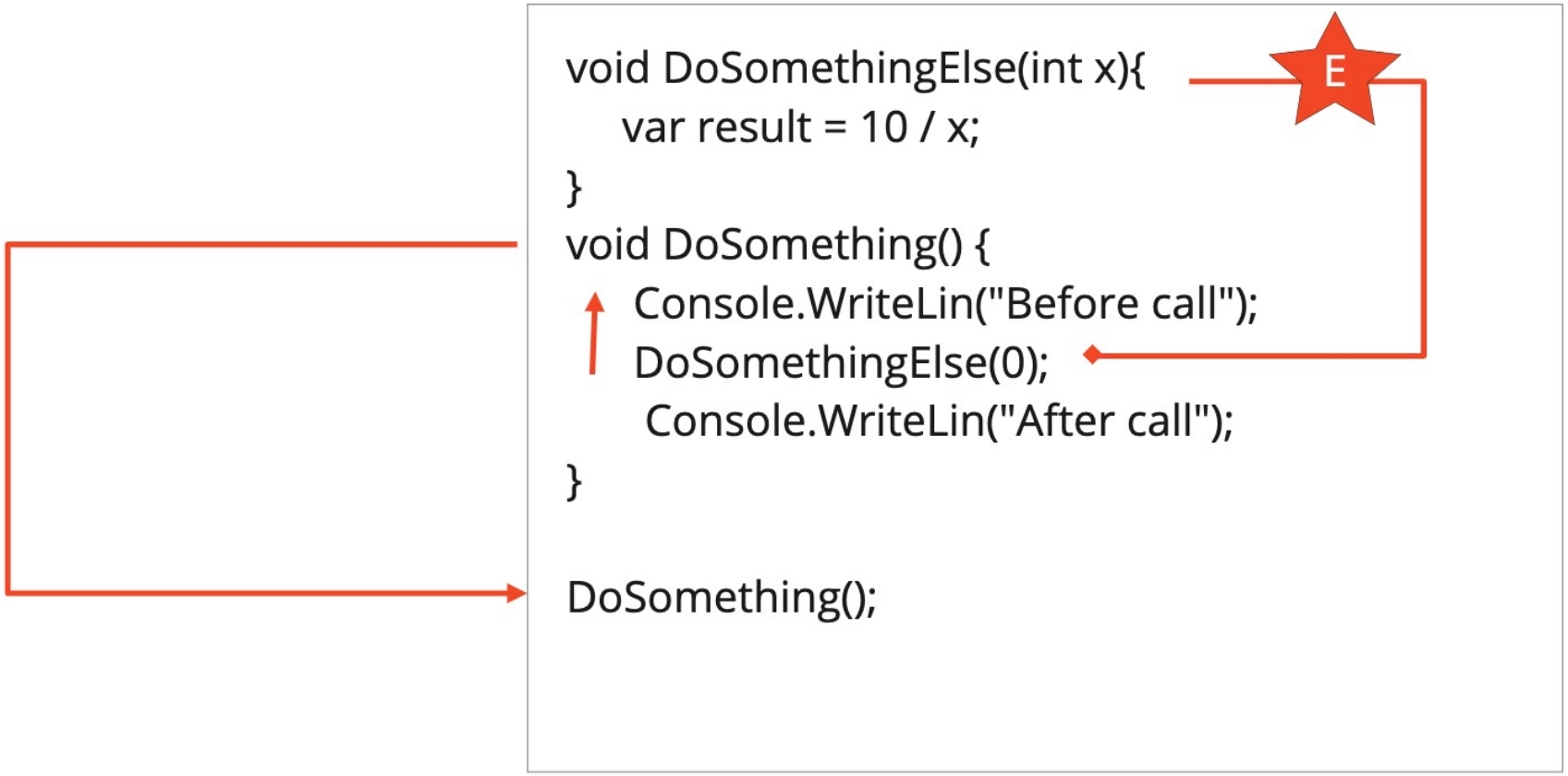
Exceptions are generated at the point of error

- Here a divide by zero occurs at the red star



Exception Propagation

- Unless handled an exception propagates back through the call stack



Handling an Exception

If an exception reaches the top of the call stack

- Execution of the application code terminates
- Control passes to apps the execution engine
    - Could be the OS or a host process

Catching exceptions stops exception propagation

- Exception can be dealt with
- After exception catching normal execution flow resumes

Catching Exceptions

Simple try catch block

```
try {
    <do something that might fail>
}
catch (<Exception>) {
    <code to handle the failure>
}
```

try catch - Example

```
void DoSomething(int x) {
    try {
        var result = 10 / x;
    }
    catch (Exception){
        Console.WriteLine("An error has occurred!")
    }
    Console.WriteLine("Finished executing DoSomething()")
}
```

> DoSomething(0);

An error has occurred!

Finished Executing DoSomething

In this chapter we learned ...

- Controlling execution flow
- if, for, foreach, switch and while statements
- How to handle exceptions