C# Flow Control

In this article, we will cover flow control in C#, a fundamental topic for any software developer. Effective management of flow control ensures that your program behaves predictably, performs efficiently, and remains readable and maintainable. Based on industry best practices, including insights from Steve McConnell's *Code Complete*, we'll explore how to handle key flow control structures like `if`, `switch`, loops (`for`, `while`, `do-while`, `foreach`), and the `break`/`continue` statements.

The `if` Statement

The `if` statement is a fundamental control structure that evaluates a Boolean condition and executes a block of code if the condition is `true`. Writing effective `if` statements is crucial for maintaining clean, readable, and efficient code while improving its clarity and maintainability..

Example:

```csharp
int threshold = 25;
int userInput = 25;

if (userInput == threshold)
{
    Console.WriteLine("The input matches the threshold! 🥳");
}
```

In the example above, the `if` block executes only if `userInput` equals `25`. If the condition evaluates to `true`, the message is printed.

Best Practices

1. **Use Clear Conditions**
   The condition inside an `if` statement should be simple and easy to understand. Avoid writing complex or convoluted expressions that make the logic hard to follow. If necessary, break the condition into smaller, meaningful variables or expressions.

   Before:

   ```csharp
   if ((x > 10 && y < 20) || (x < 5 && y > 30))
   {
       // Complex logic...
   }
   ```

   After:

```
bool isInValidRange = (x > 10 && y < 20);
bool isOutOfRange = (x < 5 && y > 30);

if (isInValidRange || isOutOfRange)
{
    // Clearer logic...
}
```

Breaking down the condition into descriptive Boolean variables improves readability and makes the logic easier to understand.

2. **Avoid Nested `if` Statements**
While nesting `if` statements can sometimes be necessary, excessive nesting makes the code hard to follow. Instead of deeply nested `if` blocks, consider **guard clauses**—an early return or exit strategy that reduces nesting, which make your code flatter and easier to read.

Before:

```
if (user != null)
{
    if (user.IsActive)
    {
        if (user.HasPermissions)
        {
            // Perform action...
        }
    }
}
```

After (Using Guard Clauses):

```
if (user == null) return;
if (!user.IsActive) return;
if (!user.HasPermissions) return;

// Perform action...
```

Using guard clauses reduces unnecessary nesting and makes the main flow of the code more apparent.

3. **Favor Positive Conditions**
Whenever possible, write conditions in a positive manner. Positive conditions are easier to understand and often make your code more readable. For example, instead of negating a condition, use its positive equivalent.

Before (Negative Condition):

```
if (!isInvalid)
{
    // Logic for valid case...
}
```

After (Positive Condition):

```
if (isValid)
{
    // Logic for valid case...
}
```

This change makes the condition clearer and reduces cognitive load when reading the code.

4. **Handle Edge Cases and Errors First (Guard Clauses)**
   Guard clauses allow you to handle edge cases and errors early in your code, preventing unnecessary complexity. By handling invalid conditions upfront, the main logic becomes more readable and manageable.

   ```
   if (customer == null)
   {
       throw new ArgumentNullException(nameof(customer));
   }

   if (!customer.IsActive)
   {
       Console.WriteLine("Customer account is inactive.");
       return;
   }
   // Continue with core logic for active customers...
   ```

   By handling invalid conditions immediately, you keep the main logic simple and focused.

5. **Put the Normal Case First**
   As Steve McConnell suggests in *Code Complete*, always handle the normal or expected case first in an `if` block. This keeps the focus on the main flow of logic, while exceptional cases can be handled afterward. This makes the code easier to follow.

   Before:

   ```
   if (!isAuthorized)
   {
       // Handle unauthorized case...
   }
   else
   {
       // Main logic for authorized case...
   }
   ```

   After (Normal Case First):

   ```
   if (isAuthorized)
   {
       // Main logic for authorized case...
   }
   else
   {
       // Handle unauthorized case...
   }
   ```

6. **Consider using `switch` for multiple conditions**
   When dealing with multiple conditions related to a single variable, a `switch` statement is often more readable than a series of `if-else` blocks. It provides a structured way to handle multiple cases, making it easier to see all the conditions at a glance.

   Before (Multiple `if` Statements):

   ```
   if (status == "Pending")
   {
       // Handle pending status...
   }
   else if (status == "Approved")
   {
       // Handle approved status...
   }
   else if (status == "Rejected")
   {
       // Handle rejected status...
   }
   ```

   After (`switch` Statement):

   ```
   switch (status)
   {
       case "Pending":
           // Handle pending status...
           break;
       case "Approved":
           // Handle approved status...
           break;
       case "Rejected":
           // Handle rejected status...
           break;
       default:
           // Handle unknown status...
           break;
   }
   ```

   Using a `switch` statement provides a clearer and more maintainable structure when dealing with multiple cases.

7. **Refactor Complex `if` Logic**
   If your `if` conditions become too complex, it’s a sign that you should refactor the code. Extracting logic into separate methods or using design patterns (like the **Strategy Pattern**) can simplify decision-making and improve code readability.

   ```
   if (ShouldSendNotification(user, notificationType))
   {
       SendNotification(user, notificationType);
   }
   ```

   By abstracting the logic into a method, you encapsulate the decision-making process and keep the `if` statement concise.

8. Test and Document Your Conditions
   Always ensure that your conditions are thoroughly tested, including edge cases, null checks, and unexpected inputs. Use comments to document complex conditions or edge cases, making your code easier to maintain.

   ```
   // Ensure the user is not null and has the necessary permissions
   if (user != null && user.HasPermissions)
   {
       // Perform action...
   }
   ```

   A simple comment clarifies the intent of the condition and makes the code easier to understand for others.

The `switch` Statement

The `switch` statement is a control structure used to select one of many code blocks to execute based on the value of a variable. It's particularly useful when you need to compare a variable against several discrete values.

Example:

```
string status = "Approved";

switch (status)
{
    case "Pending":
        Console.WriteLine("The request is pending.");
        break;
    case "Approved":
        Console.WriteLine("The request is approved.");
        break;
    case "Rejected":
        Console.WriteLine("The request is rejected.");
        break;
    default:
        Console.WriteLine("Unknown status.");
        break;
}
```

In this example, the program evaluates the value of `status` and executes the corresponding block of code. If none of the cases match, the `default` block is executed.

Best Practices for `switch` Statements:

1. **Always Include a `default` Case**: A `default` case ensures that your program handles unexpected or unhandled values gracefully.
2. **Keep Cases Short and Concise**: Each case in a `switch` should perform a specific task. If the logic is complex, consider refactoring it into a separate method to keep the `switch` statement readable.

3. **Use `switch` When Handling Multiple Discrete Values**: If your control flow depends on evaluating the same variable against multiple constant values, `switch` is more readable and maintainable than multiple `if-else` statements.

In recent versions of C# (starting with C# 8.0), the `switch` statement has been significantly enhanced with pattern matching, allowing you to write more concise and expressive code through `switch` **expressions**. These enhancements make it easier to handle multiple conditions and pattern matching scenarios efficiently, often reducing the need for longer, more verbose `switch` statements.

Traditional `switch` vs `switch` Expressions

A traditional `switch` statement typically looks like this:

```
public string GetDayName(int day)
{
    switch (day)
    {
        case 1:
            return "Monday";
        case 2:
            return "Tuesday";
        case 3:
            return "Wednesday";
        default:
            return "Unknown";
    }
}
```

However, with `switch` **expressions** introduced in C# 8.0, you can achieve the same result with a more concise and expressive syntax:

```
public string GetDayName(int day) => day switch
{
    1 => "Monday",
    2 => "Tuesday",
    3 => "Wednesday",
    _ => "Unknown"
};
```

Benefits of `switch` Expressions:

1. **More concise**: You no longer need to write `case` and `break` for each branch.
2. **Functional**: `switch` expressions return a value directly, which aligns with functional programming principles.
3. **Exhaustive**: The compiler checks if all cases are covered, ensuring that any missing cases are flagged as warnings or errors.

Pattern Matching with `switch` Expressions

`switch` expressions become even more powerful with pattern matching, allowing for checks beyond simple equality, such as type checks and property matches. For example:

```
public string DescribeNumber(object number) => number switch
{
    int n when n > 0 => "Positive integer",
    int n when n < 0 => "Negative integer",
    double d => "Double",
    null => "No value",
    _ => "Unknown type"
};
```

This example uses **relational patterns** and **type patterns** to match different types of numbers and handle them accordingly.

Advanced Pattern Matching:

1. **Relational Patterns**: You can check ranges of values using conditions like `<` and `>`.

```
string WaterState(int temperature) => temperature switch
{
    < 32 => "Solid",
    > 212 => "Gas",
    _ => "Liquid"
};
```

2. **Property Patterns**: These allow matching based on object properties.

```
public record Person(string Name, int Age);

string CheckPerson(Person person) => person switch
{
    { Age: >= 18 } => "Adult",
    { Age: < 18 } => "Minor",
    _ => "Unknown"
};
```

3. **Tuple Patterns**: Useful when working with multiple values.

```
string ClassifyCoordinates(int x, int y) => (x, y) switch
{
    (0, 0) => "Origin",
    (_, 0) => "On the X axis",
    (0, _) => "On the Y axis",
    _ => "Somewhere in the plane"
};
```

By using these new features, you can write cleaner and more readable code, especially when handling complex scenarios that involve multiple conditions or data structures.

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns

Loops in C#

Loops are used to execute a block of code repeatedly based on a condition. There are four primary looping structures in C#: `for`, `while`, `do-while`, and `foreach`.

The `for` Loop

A `for` loop repeats a block of code a specific number of times, which makes it ideal for iterating over arrays or performing a task a known number of times.

Example:

```csharp
for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Iteration {i + 1}");
}
```

In this example, the loop will execute five times, printing the iteration number to the console.

Best Practices for `for` Loops:

1. **Use Descriptive Variable Names**: Instead of `i`, use meaningful variable names if the context requires it (e.g., `index` or `counter`), especially in nested loops.
2. **Avoid Magic Numbers**: Rather than hardcoding values like `5` directly in the condition, use a named constant or variable to improve readability.

```csharp
int numberOfIterations = 5;

for (int index = 0; index < numberOfIterations; index++)
{
    Console.WriteLine($"Iteration {index + 1}");
}
```

The `while` Loop

A `while` loop repeats as long as its condition evaluates to `true`. It's useful when you don't know the number of iterations ahead of time and want to continue looping until a certain condition is met.

```csharp
int counter = 0;

while (counter < 5)
{
    Console.WriteLine($"Counter: {counter}");
    counter++;
}
```

This loop will continue running until the condition `counter < 5` is no longer `true`.

Best Practices for `while` Loops:

1. **Ensure Your Loop Will Eventually Terminate**: Always update variables inside the loop to ensure the condition will eventually become `false`, avoiding infinite loops.
2. **Consider Loop Conditions Carefully**: If the condition may never be met or is complex, consider adding a safety mechanism or logging to debug the loop's behavior.

The `do-while` Loop

A `do-while` loop is similar to a `while` loop, but it guarantees that the code inside the loop will run at least once, even if the condition is `false` initially.

```
int number = 0;

do
{
    Console.WriteLine($"Number: {number}");
    number++;
} while (number < 5);
```

In this example, the block of code executes first, and then the condition is evaluated. The loop will continue as long as `number` is less than 5.

Best Practices for `do-while` Loops:

1. **Use When You Want at Least One Execution**: The primary use case for a `do-while` loop is when you need the loop to execute at least once, even if the condition is not met from the start.

The `foreach` Loop

The `foreach` loop is a more convenient way to iterate over collections, such as arrays, lists, or dictionaries. It automatically iterates through each element in the collection.

```
string[] fruits = { "Apple", "Banana", "Cherry" };

foreach (string fruit in fruits)
{
    Console.WriteLine(fruit);
}
```

In this example, the loop iterates over the `fruits` array and prints each fruit.

Best Practices for `foreach` Loops:

1. **Use When Iterating Over Collections**: A `foreach` loop is simpler and more readable than a `for` loop when iterating over collections because you don’t need to manage index values manually.

2. **Avoid Modifying Collections Inside a `foreach`**: Modifying the collection (e.g., adding or removing elements) during iteration can lead to runtime exceptions. If you need to modify the collection, consider using a `for` loop or iterating over a copy of the collection.

The `break` and `continue` Statements

In C#, `break` and `continue` are control flow statements that are used to alter the execution of loops and `switch` statements. They provide a way to control when to exit a loop early or skip to the next iteration based on certain conditions. Proper usage of these statements can make your loops more efficient and readable.

The `break` Statement

The `break` statement is used to immediately exit the loop or `switch` statement in which it is placed. When the `break` statement is executed, control is transferred to the code following the loop or `switch` block.

```
for (int i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break; // Exit the loop when i is 5
    }
    Console.WriteLine($"Iteration {i}");
}
```

In this example, the loop will terminate as soon as `i` equals 5, so it will print iterations 0 to 4, and then exit the loop.

Best Practices for Using `break`:

1. **Use `break` to Exit Early from Loops When Necessary**: You can use `break` to terminate loops early when a certain condition is met. This can improve performance if continuing the loop would be unnecessary.
2. **Avoid Overusing `break` in Complex Loops**: While `break` can simplify loop logic, overusing it can make loops harder to understand. Use it sparingly and consider refactoring complex loops to improve readability.
3. **Always Include `break` in `switch` Statements**: Ensure that each case in a `switch` statement has a `break` (or equivalent control flow like `return` or `throw`), unless you intend to let execution fall through to the next case.

The `continue` Statement

The `continue` statement skips the remaining code inside the loop for the current iteration and proceeds to the next iteration. Unlike `break`, `continue` does not terminate the loop—it just skips to the next loop cycle.

```
for (int i = 0; i < 10; i++)
{
    if (i % 2 == 0)
    {
        continue; // Skip the rest of the loop body for even numbers
    }
    Console.WriteLine($"Odd number: {i}");
}
```

In this example, the `continue` statement causes the loop to skip over even numbers and only prints odd numbers.

Best Practices for Using `continue`:

1. **Use `continue` to Skip Unnecessary Processing**: When certain iterations in a loop don't require further processing, `continue` can help you skip unnecessary steps, making the loop more efficient.
2. **Avoid Overusing `continue`**: While `continue` can simplify loops, excessive use may make the loop logic harder to follow. Ensure that it is used only when necessary to improve clarity.
3. **Prefer `continue` Over Complex `if` Logic**: In some cases, using `continue` is clearer than adding multiple nested `if` conditions. For example, instead of writing complex conditional logic inside a loop, you can use `continue` to skip iterations early.

Before (Complex `if` Condition):

```
for (int i = 0; i < 10; i++)
{
    if (i % 2 != 0)
    {
        // Perform action only for odd numbers
        Console.WriteLine(i);
    }
}
```

After (Using `continue` for Clarity):

```
for (int i = 0; i < 10; i++)
{
    if (i % 2 == 0)
    {
        continue; // Skip even numbers
    }
    Console.WriteLine(i); // Perform action only for odd numbers
}
```

Using `continue` simplifies the loop and avoids unnecessary nesting.

Conclusion

Mastering flow control in C# is vital to writing clean, maintainable, and efficient code. Whether it's structuring `if` statements clearly, leveraging the power of `switch` expressions, or using loops effectively, following best practices will lead to better software development.

For further reading on these principles, consider *Code Complete* by Steve McConnell, which offers deeper insights into writing clean, high-quality code.

References

McConnell, Steve. Code Complete. Microsoft Press.
Microsoft Documentation: C# Patterns and Flow Control