



Control Flow, Conditional Statements and Loops

Correctly Organizing the Control Flow Logic

Telerik Software Academy
Learning & Development
<http://academy.telerik.com>



Table of Contents

- ◆ Organizing Straight-line Code
- ◆ Using Conditional Statements
- ◆ Using Loops
- ◆ Other Control Flow Structures



Organizing Straight-Line Code

Order and Separate Your
Dependencies Correctly



Straight-Line Code

◆ When statements' order matters

```
GetData();  
GroupData();  
Print();
```



- ◆ Make dependencies obvious
- ◆ Name methods according to dependencies
- ◆ Use method parameters

```
data = GetData();  
groupedData = GroupData(data);  
PrintGroupedData(groupedData);
```



- ◆ Document the control flow if needed

Straight-Line Code (2)

- ◆ When statements' order does not matter
 - ◆ Make code read from top to bottom like a newspaper
 - ◆ Group related statements together
 - ◆ Make clear boundaries for dependencies
 - ◆ Use blank lines to separate dependencies
 - ◆ Use separate method



Straight-Line Code – Examples



```
ReportFooter CreateReportFooter(Report report)
{
    // ...
}

ReportHeader CreateReportHeader(Report report)
{
    // ...
}

Report CreateReport()
{
    var report = new Report();
    report.Footer = CreateReportFooter(report);
    report.Content = CreateReportContent(report);

    report.Header = CraeteReportHeader(report);
    return report;
}

ReportContent CreateReportContent(Report report)
{
    // ...
}
```

Straight-Line Code – Examples



```
Report CreateReport()
```

```
{  
    var report = new Report();  
  
    report.Header = CreateReportHeader(report);  
    report.Content = CreateReportContent(report);  
    report.Footer = CreateReportFooter(report);  
  
    return report;  
}
```

```
ReportHeader CreateReportHeader(Report report)
```

```
{  
    // ...  
}
```

```
ReportContent CreateReportContent(Report report)
```

```
{  
    // ...  
}
```

```
ReportFooter CreateReportFooter(Report report)
```

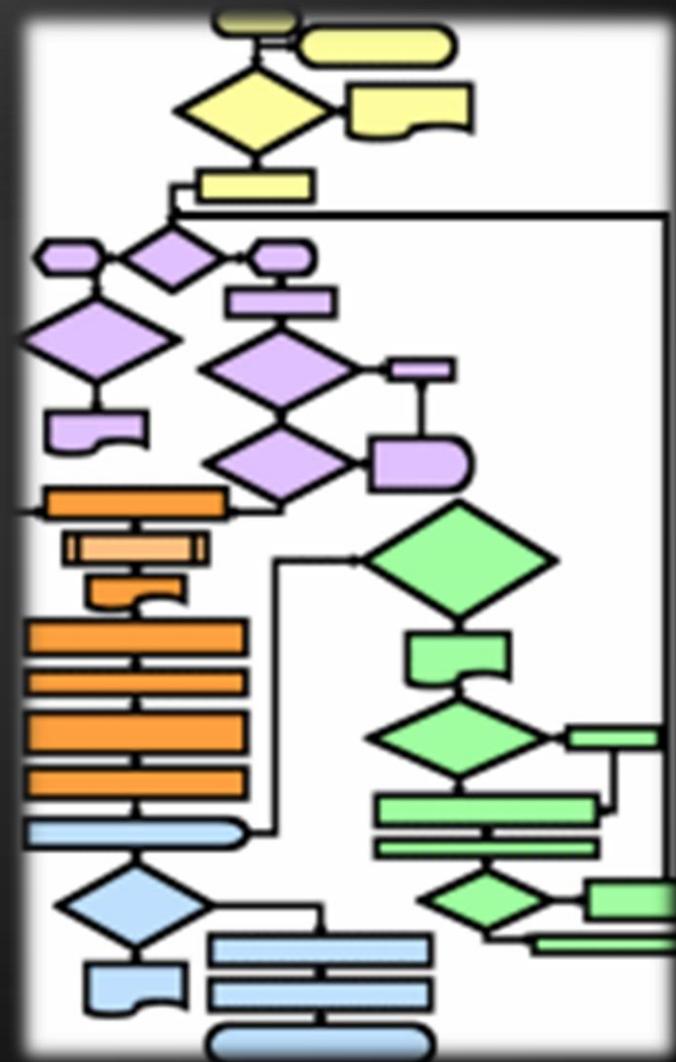
```
{  
    // ...  
}
```

Straight-Line Code – Summary

- ◆ The most important thing to consider when organizing straight-line code is
 - ◆ Ordering dependencies
- ◆ Dependencies should be made obvious
 - ◆ Through the use of good routine names, parameter lists and comments
- ◆ If code doesn't have order dependencies
 - ◆ Keep related statements together

Using Conditional Statements

Using Control Structures



Using Conditional Statements

- ◆ Always use { and } for the conditional statements body, even when it is a single line:

```
if (condition)
{
    DoSomething();
}
```



- ◆ Why omitting the brackets could be harmful?

```
if (condition)
    DoSomething();
    DoAnotherThing();
DoDifferentThing();
```



- ◆ This is misleading code + misleading formatting

Using Conditional Statements (2)

- ◆ Always put the normal (expected) condition first after the if clause

```
var response = GetHttpWebResponse();
if (response.Code == Code.NotFound)
{
    // ...
}
else
{
    if (response.Code == Code.OK)
    {
        // ...
    }
}
```



```
var response = GetHttpWebResponse();
if (response.Code == Code.OK)
{
    // ...
}
else
{
    if (response.Code == Code.NotFound)
    {
        // ...
    }
}
```



- ◆ Start from most common cases first, then go to the unusual ones

Using Conditional Statements (3)

- ◆ Avoid comparing to true or false:

```
if (HasErrors == true)  
{  
    ...  
}
```



```
if (HasErrors)  
{  
    ...  
}
```



- ◆ Always consider the else case
 - ◆ If needed, document why the else isn't necessary

```
if (parserState != States.Finished)  
{  
    // ...  
}  
else  
{  
    // Ignore all content once the parser has finished  
}
```



Using Conditional Statements (4)

- ◆ Avoid double negation

```
if (!HasNoError)  
{  
    DoSomething();  
}
```



```
if (HasErrors)  
{  
    DoSometing();  
}
```



- ◆ Write if clause with a meaningful statement

```
if (!HasError)  
;  
else  
{  
    DoSometing();  
}
```



```
if (HasErrors)  
{  
    DoSometing();  
}
```



- ◆ Use meaningful boolean expressions, which read like a sentence

Using Conditional Statements (5)

- ◆ Be aware of copy/paste problems in if-else bodies

```
if (SomeCondition)
{
    var p = GetSomePerson();
    p.SendMail();
    p.SendSms();
}
else
{
    var p = GetOtherPerson();
    p.SendMail();
    p.SendSms();
}
```



```
Person p = null;
if (SomeCondition)
{
    p = GetSomePerson();
}
else
{
    p = GetOtherPerson();
}

p.SendMail();
p.SendSms();
```



Use Simple Conditions

- ◆ Do not use complex if conditions
 - ◆ You can always simplify them by introducing boolean variables or boolean methods
 - ◆ Incorrect example:

```
if (x > 0 && y > 0 && x < Width-1 && y < Height-1 &&  
    matrix[x, y] == 0 && matrix[x-1, y] == 0 &&  
    matrix[x+1, y] == 0 && matrix[x, y-1] == 0 &&  
    matrix[x, y+1] == 0 && !visited[x, y]) ...
```



- ◆ Complex boolean expressions can be harmful
- ◆ How you will find the problem if you get IndexOutOfRangeException?

Simplifying Boolean Conditions

- ◆ The last example can be easily refactored into self-documenting code:

```
bool inRange = x > 0 && y > 0 && x < Width-1 && y < Height-1;  
if (inRange)  
{  
    bool emptyCellAndNeighbours =  
        matrix[x, y] == 0 && matrix[x-1, y] == 0 &&  
        matrix[x+1, y] == 0 && matrix[x, y-1] == 0 &&  
        matrix[x, y+1] == 0;  
    if (emptyCellAndNeighbours && !visited[x, y]) ...  
}
```



- ◆ Now the code is:
 - ◆ Easy to read – the logic of the condition is clear
 - ◆ Easy to debug – breakpoint can be put at the if

- ◆ Use object-oriented approach

```
class Maze
{
    Cell CurrentCell { get; set; }
    IList<Cell> VisitedCells { get; }
    IList<Cell> NeighbourCells { get; }
    Size Size { get; }

    bool IsCurrentCellInRange()
    {
        return Size.Contains(CurrentCell);
    }

    bool IsCurrentCellVisited()
    {
        return VisitedCells.Contains(CurrentCell);
    }
}
```

(continues on the next slide)



Simplifying Boolean Conditions (3)

```
bool AreNeighbourCellsEmpty()
{
    ...
}

bool ShouldVisitCurrentCell()
{
    return
        IsCurrentCellInRange() &&
        CurrentCell.IsEmpty() &&
        AreNeighbourCellsEmpty() &&
        !IsCurrentCellVisited()
}
```



- ◆ Now the code:
 - ◆ Models the real scenario
 - ◆ Stays close to the problem domain

- ◆ Sometimes a decision table can be used for simplicity

```
var table = new Hashtable();
table.Add("A", new AWorker());
table.Add("B", new BWorker());
table.Add("C", new CWorker());

string key = GetWorkerKey();

var worker = table[key];
if (worker != null)
{
    ...
    worker.Work();
    ...
}
```



Positive Boolean Expressions

- ◆ Starting with a positive expression improves the readability

```
if (IsValid)
{
    DoSometing();
}
else
{
    DoSometingElse();
}
```



```
if (!IsValid)
{
    DoSometingElse();
}
else
{
    DoSomething();
}
```



- ◆ Use De Morgan's laws for negative checks

```
if (!IsValid || !IsVisible)
```



```
if (!(IsValid && IsVisible))
```

Use Parentheses for Simplification

- ◆ Avoid complex boolean conditions without parenthesis

```
if (a < b && b < c || c == d)
```



- ◆ Using parenthesis helps readability as well as ensure correctness

```
if (( a < b && b < c ) || c == d)
```



- ◆ Too many parenthesis have to be avoided as well
 - Consider separate Boolean methods or variables in those cases

Boolean Expression Evaluation

- ◆ Most languages evaluate from left to right
 - ◆ Stop evaluation as soon as some of the boolean operands is satisfied

```
if (FalseCondition && OtherCondition) = false
```

```
if (TrueCondition || OtherCondition) = true
```

- ◆ Useful when checking for null

```
if (list != null && list.Count > 0) ...
```

- ◆ There are languages that does not follow this “short-circuit” rule

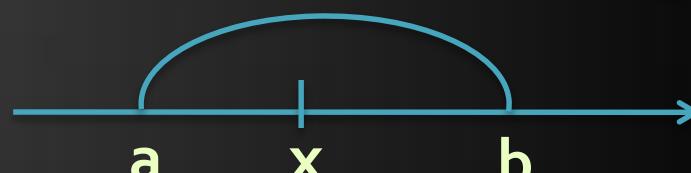
Numeric Expressions as Operands

- ◆ Write numeric boolean expressions as they are presented on a number line
 - ◆ Contained in an interval

`if (x > a && b > x)`



`if (a < x && x < b)`

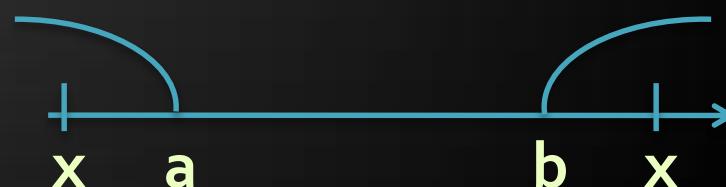


- ◆ Outside of an interval

`if (a > x || x > b)`



`if (x < a || b < x)`



Avoid Deep Nesting of Blocks

- ◆ Deep nesting of conditional statements and loops makes the code unclear
 - ◆ More than 2-3 levels is too deep
 - ◆ Deeply nested code is complex and hard to read and understand
 - ◆ Usually you can extract portions of the code in separate methods
 - ◆ This simplifies the logic of the code
 - ◆ Using good method name makes the code self-documenting

Deep Nesting – Example



```
if (maxElem != Int32.MaxValue)
{
    if (arr[i] < arr[i + 1])
    {
        if (arr[i + 1] < arr[i + 2])
        {
            if (arr[i + 2] < arr[i + 3])
            {
                maxElem = arr[i + 3];
            }
            else
            {
                maxElem = arr[i + 2];
            }
        }
        else
        {
            if (arr[i + 1] < arr[i + 3])
            {
                maxElem = arr[i + 3];
            }
            else
            {
                maxElem = arr[i + 1];
            }
        }
    }
}
```

(continues on the next slide)

Deep Nesting – Example (2)



```
else
{
    if (arr[i] < arr[i + 2])
    {
        if (arr[i + 2] < arr[i + 3])
        {
            maxElem = arr[i + 3];
        }
        else
        {
            maxElem = arr[i + 2];
        }
    }
    else
    {
        if (arr[i] < arr[i + 3])
        {
            maxElem = arr[i + 3];
        }
        else
        {
            maxElem = arr[i];
        }
    }
}
```

Avoiding Deep Nesting – Example



```
private static int Max(int i, int j)
{
    if (i < j)
    {
        return j;
    }
    else
    {
        return i;
    }
}

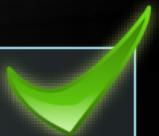
private static int Max(int i, int j, int k)
{
    if (i < j)
    {
        int maxElem = Max(j, k);
        return maxElem;
    }
    else
    {
        int maxElem = Max(i, k);
        return maxElem;
    }
}
```

(continues on the next slide)

Avoiding Deep Nesting – Example (2)

```
private static int FindMax(int[] arr, int i)
{
    if (arr[i] < arr[i + 1])
    {
        int maxElem = Max(arr[i + 1], arr[i + 2], arr[i + 3]);
        return maxElem;
    }
    else
    {
        int maxElem = Max(arr[i], arr[i + 2], arr[i + 3]);
        return maxElem;
    }

    if (maxElem != Int32.MaxValue) {
        maxElem = FindMax(arr, i);
    }
}
```



Using Case Statement

- ◆ Choose the most effective ordering of cases
 - ◆ Put the normal (usual) case first
 - ◆ Order cases by frequency
 - ◆ Put the most unusual (exceptional) case last
 - ◆ Order cases alphabetically or numerically
- ◆ Keep the actions of each case simple
 - ◆ Extract complex logic in separate methods
- ◆ Use the default clause in a case statement or the last else in a chain of if-else to trap errors

Incorrect Case Statement



```
void ProcessNextChar(char ch)
{
    switch (parseState)
    {
        case InTag:
            if (ch == ">")
            {
                Console.WriteLine("Found tag: {0}", tag);
                text = "";
                parseState = ParseState.OutOfTag;
            }
            else
            {
                tag = tag + ch;
            }
            break;
        case OutOfTag:
            ...
    }
}
```

Improved Case Statement



```
void ProcessNextChar(char ch)
{
    switch (parseState)
    {
        case InTag:
            ProcessCharacterInTag(ch);
            break;
        case OutOfTag:
            ProcessCharacterOutOfTag(ch);
            break;
        default:
            throw new InvalidOperationException(
                "Invalid parse state: " + parseState);
    }
}
```

- ◆ Avoid using fallthroughs
- ◆ When you do use them, document them well

```
switch (c)
{
    case 1:
    case 2:
        DoSomething();
        // FALLTHROUGH
    case 17:
        DoSomethingElse();
        break;
    case 5:
    case 43:
        DoOtherThings();
        break;
}
```



Case – Best Practices(2)

- ◆ Overlapping control structures is evil:

```
switch (inputVar)
{
    case 'A': if (test)
    {
        // statement 1
        // statement 2
    case 'B': // statement 3
        // statement 4
        ...
    }
    ...
    break;
}
```



- ◆ This code will not compile in C# but may compile in other languages

Control Statements – Summary

- ◆ For simple if-else-s, pay attention to the order of the if and else clauses
 - ◆ Make sure the nominal case is clear
- ◆ For if-then-else chains and case statements, choose the most readable order
- ◆ Optimize boolean statements to improve readability
- ◆ Use the default clause in a case statement or the last else in a chain of if-s to trap errors

Using Loops

Choose Appropriate Loop Type
and Don't Forget to Break



- ◆ Choosing the correct type of loop:
 - ◆ Use **for** loop to repeat some block of code a certain number of times
 - ◆ Use **foreach** loop to process each element of an array or a collection
 - ◆ Use **while** / **do-while** loop when you don't know how many times a block should be repeated
- ◆ Avoid deep nesting of loops
 - ◆ You can extract the loop body in a new method

Loops: Best Practices

- ◆ Keep loops simple
 - ◆ This helps readers of your code
- ◆ Treat the inside of the loop as it were a routine
 - ◆ Don't make the reader look inside the loop to understand the loop control
- ◆ Think of a loop as a black box:

```
while (!inputFile.EndOfFile() && !hasErrors)
{
    (black box code)
}
```

Loops: Best Practices (2)

- ◆ Keep loop's housekeeping at the start or at the end of the loop block

```
while (index < 10)
{
    ...
    index += 2;
    ...
}
```



```
while (index < 10)
{
    ...
    ...
    index += 2;
}
```



- ◆ Use meaningful variable names to make loops readable

```
for(i=2000, i<2011, i++)
{
    for(j=1, j<=12, j++)
    ...
}
```



```
for (year=2000, year<2011, year++)
{
    for(month=1, month<=12, month++)
    ...
}
```



Loops: Best Practices (3)

- ◆ Avoid empty loops

```
while ((inputChar = Console.Read()) != '\n')  
{  
}
```



```
do  
{  
    inputChar = Console.Read();  
}  
while (inputChar != '\n');
```



- ◆ Be aware of your language (loop) semantics
 - C# – access to modified closure

Loops: Tips on for-Loop

- ◆ Don't explicitly change the index value to force the loop to stop
 - Use while-loop with break instead
- ◆ Put only the controlling statements in the loop header

```
for (i = 0, sum = 0;  
     i < length;  
     sum += arr[i], i++)  
{  
    ;  
}
```



```
sum = 0;  
for (i = 0; i < length; i++)  
{  
    sum += arr[i];  
}
```



Loops: Tips on for-Loop(2)

- ◆ Avoid code that depends on the loop index's final value

```
for (i = 0; i < length; i++)
{
    if (array[i].id == key)
    {
        break;
    }
}

// Lots of code
...

return (i >= length);
```



```
bool found = false;
for (i = 0; i < length; i++)
{
    if (array[i].id == key)
    {
        found = true;
        break;
    }
}

// Lots of code
...

return found;
```



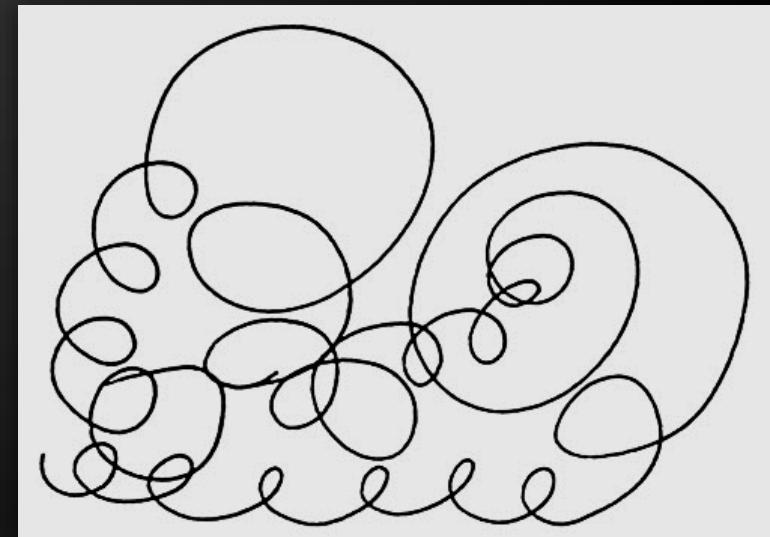
Loops: break and continue

- ◆ Use `continue` for tests at the top of a loop to avoid nested `if`-s
- ◆ Avoid loops with lots of `break`-s scattered through it
- ◆ Use `break` and `continue` only with caution



How Long Should a Loop Be?

- ◆ Try to make the loops short enough to view it all at once (one screen)
- ◆ Use methods to shorten the loop body
- ◆ Make long loops especially clear
- ◆ Avoid deep nesting in loops



Other Control Flow Structures

To Understand Recursion,
One Must First Understand
Recursion



The return Statement

- ◆ Use **return** when it enhances readability
- ◆ Use **return** to avoid deep nesting

```
void ParseString(string str)
{
    if (string != null)
    {
        // Lots of code
    }
}
```



```
void ParseString(string str)
{
    if (string == null)
    {
        return;
    }

    // Lots of code
}
```



- ◆ Avoid multiple **return**-s in long methods

- ◆ Useful when you want to walk a tree / graph-like structures
- ◆ Be aware of infinite recursion or indirect recursion
- ◆ Recursion example:

```
void PrintWindowsRecursive(Window w)
{
    w.Print()
    foreach(childWindow in w.ChildWindows)
    {
        PrintWindowsRecursive(childWindow);
    }
}
```



- ◆ Ensure that recursion has end (bottom)
- ◆ Verify that recursion is not very high-cost
 - Check the occupied system resources
 - You can always use stack classes and iteration
- ◆ Don't use recursion when there is better linear (iteration based) solution, e.g.
 - Factorials
 - Fibonacci numbers
- ◆ Some languages optimize tail-call recursions

- ◆ Avoid goto-s, because they have a tendency to introduce spaghetti code
- ◆ [“A Case Against the GOTO Statement”](#)
by Edsger Dijkstra
- ◆ Use goto-s as a last resort
 - If they make the code more maintainable
- ◆ C# supports goto with labels, but avoid it!



Control Flow, Conditional Statements and Loops

Questions?

1. Refactor the following class using best practices for organizing straight-line code:

```
public void Cook()  
public class Chef  
{  
    private Bowl GetBowl()  
    {  
        //...  
    }  
    private Carrot GetCarrot()  
    {  
        //...  
    }  
    private void Cut(Vegetable potato)  
    {  
        //...  
    }  
}
```

// continue on the next slide

```
public void Cook()
{
    Potato potato = GetPotato();
    Carrot carrot = GetCarrot();
    Bowl bowl;
    Peel(potato);

    Peel(carrot);
    bowl = GetBowl();
    Cut(potato);
    Cut(carrot);
    bowl.Add(carrot);

    bowl.Add(potato);
}
private Potato GetPotato()
{
    //...
}
```

2. Refactor the following if statements:

```
Potato potato;  
//...  
if (potato != null)  
    if(!potato.HasNotBeenPeeled && !potato.IsRotten)  
        Cook(potato);
```

```
if (x >= MIN_X && (x <= MAX_X && ((MAX_Y >= y &&  
MIN_Y <= y) && !shouldNotVisitCell)))  
{  
    VisitCell();  
}
```

3. Refactor the following loop:

```
int i=0;
for (i = 0; i < 100;)
{
    if (i % 10 == 0)
    {
        Console.WriteLine(array[i]);
        if ( array[i] == expectedValue )
        {
            i = 666;
        }
        i++;
    }
    else
    {
        Console.WriteLine(array[i]);
        i++;
    }
}
// More code here
if (i == 666)
{
    Console.WriteLine("Value Found");
}
```

Free Trainings @ Telerik Academy

- ◆ C# Programming @ Telerik Academy

- ◆ csharpfundamentals.telerik.com



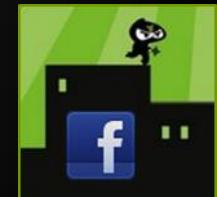
- ◆ Telerik Software Academy

- ◆ academy.telerik.com



- ◆ Telerik Academy @ Facebook

- ◆ facebook.com/TelerikAcademy



- ◆ Telerik Software Academy Forums

- ◆ forums.academy.telerik.com

