I want to buy the book NOW!
Feel free to give this document to anyone you think can have use of the information.

This document must NOT be altered in any way if you distribute it, it must be delivered as is.

Bonus: Videos and Source Code

Speed up your learning with the free video course and the source code from the book, available at:

Course 1: Chapter 1-3 of this book

Source Code Academy

C# Programming

All rights reserved. The content is presented as is and the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information in the book.

It is strictly prohibited to reproduce or transmit the whole book, or any part of the book, in any form or by any means without the written permission of the author.

You can reach the author at: info@SourceCodeAcademy.com.

Copyright © 2013 by Jonas Fagerberg
Source code

The source code accompanying this book is shared under the MIT License and can be downloaded here: http://www.sourcecodeacademy.com/

Intended audience

This book targets beginner and intermediate developers. If the reader is a beginner programmer he/she should have some basic knowledge of programming; my recommendation is to start from the beginning of the book.

Developers with intermediate skills might want to skip the first chapters which teach basic concepts and the C# language.

This book is ideal for developers that are preparing for the MCSD Exam 70-483. The book covers the most important concepts that are needed to successfully take that exam. I suggest that the reader starts from the beginning of the book even though he/she already has the knowledge presented in those chapters. I also suggest that the reader follows the links provided in the book and searches on the internet for more information prior to taking the exam. The author and publisher give no guarantees that the reader will pass the exam.

You can find the bold text “Additional reading:” followed by a search term, which you can use to search for more information about the current topic. If you are reading the e-book then you can click the search term to open a browser to view the underlying web page. If a link is broken then you simply search for the search term specified.
About the author

Jonas started a company back in 1994 focusing on education in Microsoft Office and the Microsoft operating systems. While still studying at the university in 1995, he wrote his first book about Windows 95 as well as a number of course materials.

In the year 2000, after working as a Microsoft Office developer consultant for a couple of years, he wrote his second book about Visual Basic 6.0.

Between 2000 and 2004 he worked as a Microsoft instructor with two of the largest educational companies in Sweden. First teaching Visual Basic 6.0, and when Visual Basic.NET and C# were released he started teaching these languages as well as the .NET Framework. Teaching classes on all levels for beginner to advanced developers.

From the year 2005, Jonas shifted his career towards consulting once again, working hands on with the languages and framework he taught.

This is the third book Jonas has written and it explains key concepts of C# programming for beginners to intermediate developers. He is an author with great knowledge about the C# language and the .NET framework and he has a passion for learning new things and teaching them to others. He has a knack for explaining difficult subjects in a way that is easy to understand, even for a beginner.
CONTENTS

Variables and values

   Data Types

   Expressions

   Using Variables
Variables and values

All applications use data from different sources such as user interfaces, databases, network services or other sources. Variables are the way to go when storing values; operators and expressions are used to manipulate those values.

Data Types

Variables are declared as specific data types and because C# is a type-safe language, the compiler guarantees that the value stored in a variable is the correct type. The list below shows the most commonly used data types (Type ~ Description ~ Size (bytes) ~ Range).

- **Int**
  - Whole numbers
  - 4 bytes
  - -2,147,483,648 to 2,147,483,647

- **Long**
  - Whole numbers
  - 8 bytes
  - -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

- **Float**
  - Floating point numbers
  - 4 bytes
  - +/-3.4 x 10^38
• **Double**
  • Floating point numbers
  • 8 bytes
  • +/-1.7 x 10^308

• **Decimal**
  • Monetary values
  • 16 bytes
  • 28 significant figures

• **Char**
  • Single character
  • 2 bytes
  • N/A

• **Bool**
  • Boolean
  • 1 byte
  • True or False

• **DateTime**
  • Moments in time
  • 8 bytes
  • 00:00:00 01/01/0001 to 23:59:59 12/31/9999

• **String**
  • Sequence of characters
  • 2 bytes per character
  • N/A

**Additional reading:** “Reference Tables for Types (C# Reference)”

### Expressions

An expression is made up of operands and operators. You use expressions to evaluate or manipulate data. All expressions evaluate to one value when the application runs. The result will depend on the data types and the operators that are used.

### Operand

An operand is a value; it can be a number, a string or of another type.
Operands can be constants (literals), variables, properties or function returns.

Operators

Operators define operations on operands. There are operators for the basic mathematical operations such as addition, multiplication, division and subtraction. There are also logical operations and operators for bit manipulation of a value.

Many of the operators in C# fall into the following categories:

1. Unary
   This type of operator is used to convert the value of a single operand and is achieved by placing the operator before the operand. One example is when you change a positive value to a negative with the minus sign (-). The positive value 1 changes to negative 1 when the – operator is placed in front of the 1; this would be the same as multiplying the value by -1.
   
   -1 is the same as \((-1) * 1\)

2. Binary
   This type of operator is placed between two operands. Commonly used operations of this type are multiplying or adding values.
   
   \(2 * 3\) or \(3 + 4\)

3. Ternary
   There is only one ternary operator in C#, ?. You can use this operator in conditional expressions instead of using a traditional if...then construct.

   For example
   
   ```csharp
   var result = x < 11 ? 10 : 100;
   ```
   if \(x\) is less than 11 then \(result\) will contain 10, otherwise \(result\) will contain 100.

   **Important**
One important rule regarding expressions is that the type of expression must be the same as the type of the operands. This means that some operands might need to be converted to ensure compatibility.

For instance, if you divide two int values, you might expect the same result as when dividing a double with an int, but that is not the case.

Example: \( 5/2 = 2 \) (not 2.5) and \( 5.0/2 = 2.5 \)

The following list shows the operators available in C# (Type ~ Description).

- **Arithmetic**: +, -, *, /, %
- **Increment, decrement**: ++, --
- **Comparison**: ==, !=, <, >, <=, >=, is
- **String concatenation**: +
- **Logical/bitwise operations**: &!, |, ^, !, &&, ||
- **Indexing (counting starts from element 0)**: [ ]
- **Casting**: ( ), as
- **Assignment**: =, +=, -=, *=, /=, %=, &=, |&=, ^=, <<=, >>=, ??
- **Bit shift**: <<, >>
- **Type information**: sizeof, typeof
- **Delegate concatenation and removal**: +, -
- **Overflow exception control**: checked, unchecked
- **Indirection and Address (unsafe code only)**: *, ->, [ ], &
- **Conditional (ternary operator)**: ?:

**Important**

There are classes in the .NET Framework that contains methods that you can use when performing mathematical and string operations; one of those classes is `System.Math`.

Additional reading: “C# Operators”
Using Variables

Before you can use a variable, you must declare it and give it an identifier. The identifier is the name you access the variable through. When you declare a variable, it will take up a small portion of memory to store the value associated with that variable.

You can declare multiple variables on the same line separating them with commas; all variables declared this way will have the same type.

Declaring a variable

```
int amount;
int vat;
// or
int amount, vat;
```

Assigning a variable

```
int amount;
amount = 100;
```

Declaring and assigning a variable

```
double discount = 0.5;
```

Important

In C# you must assign a variable before using it. C# is implemented this way to avoid using variables with random values; this was a source of problems in C and C++.

Implicitly Declaring Variables

A great way to declare variables is to declare them implicitly. Instead of explicitly stating the variable type such as `int`, `double` or `string`, you can use the `var` keyword. The `var` keyword uses the assigned value to determine the data type. Once a value has been assigned, the data type cannot change.
In many cases, the code will be cleaner and easier to read when using
the \texttt{var} keyword.

\begin{verbatim}
var discount = 0.5;
\end{verbatim}

\textbf{Additional reading:} “\textit{Implicitly Typed Local Variables (C\# Programming Guide)}”

\textbf{Object Variables}

When creating a variable from a class, it is undefined before the \texttt{new}
operator is used to create an object instance. The instance is placed on
the heap, a memory area designated for objects. The \texttt{new} operator
does two things: First, it causes the CLR to allocate memory for the
object on the heap. Second, it invokes a constructor in the object to
initialize its fields. A class can contain multiple overloaded constructors;
which is determined by the parameters you specify when using the \texttt{new}
operator.

\textbf{Accessing Type Members}

After an instance has been created by the new operator, you can use
the variable to access the content of that object by using a period after
the variable name.

When accessing a method, put a parenthesis after the method name; if
the method takes parameters then pass them to the method in the
parenthesis.

When accessing a property you use the property name; you can set the
value if the property has a set block and get the value if it has a get
block defined.

\textbf{Accessing an object}

\begin{verbatim}
// Create an instance
MyClass obj = new MyClass();
// Accessing the MyValue property
var value = obj.MyValue;
\end{verbatim}
// Assigning a value to the MyValue property
obj.MyValue = 100;
// Calling the GetValue function that
// returns a value
value = obj.GetValue()

Additional reading (properties): “Properties (C# Programming Guide)”

Additional reading (methods): “Methods (C# Programming Guide)”

Important

When a variable is declared, it will take up a piece of memory when the application is running. Where in the memory it ends up is determined by the data type. Value type variables such as int and double will end up on the stack which is a fast memory storage where the values can be removed as soon as the variable goes out of scope (no longer used). Object variables on the other hand will end up on the heap that handles large variables. When an object variable goes out of scope, the object will undergo a clean-up process through the Garbage Collection (GC) which is slower that the stack.

Additional reading: “Garbage Collector Basics and Performance Hints”

Variable naming rules

There are rules for naming variables to which you must adhere.

Rule 1: An identifier can only contain letters, digits and underscore characters.

Rule 2: An identifier must start with a letter or an underscore character.

Rule 3: The identifier cannot be the same as a reserved C# keyword.

Important

C# is case sensitive which means that you potentially could use the same variable name only changing the casing. The names myVariable and MyVariable would be two different variables. One instance when you
A naming convention for variables should follow a consistent convention and be uniformly applied in the codebase. This helps in maintaining code readability and maintainability.

**Casting**

In C#, it is common to convert values from one data type to another. One example is when storing a value from a text box or user interface control into a variable, or utilizing the value in a calculation. Changing a value from one type to another is called *casting*. There are two types of casting: *implicit* and *explicit*.

**Implicit Casting**

Implicit conversion is done automatically by the CLR as long as no information is lost during the cast; however, this process allows for some loss of precision.

Widening conversions are allowed; for example, converting from a smaller data type to a larger one, like casting an `int` to a `long`. The reverse conversion (from `long` to `int`) is not permitted because it might lose data.

```csharp
int x = 100;
long y;
y = x; // Implicit casting from int to long
```

The following list shows the implicit conversions that are allowed (From: To):
- `sbyte`: `short`, `int`, `long`, `float`, `double`, `decimal`
- `byte`: `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`
- `short`: `int`, `long`, `float`, `double`, `decimal`
- `ushort`: `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`
- `int`: `long`, `float`, `double`, `decimal`
- `uint`: `long`, `ulong`, `float`, `double`, `decimal`
- `long`, `ulong`: `float`, `double`, `decimal`
- `float`: `double`

**Example:**

```csharp
int x = 100;
long y;
y = x; // Implicit casting from int to long
```
char: ushort, int, uint, long, ulong, float, double, decimal

Explicit Casting

Explicit conversions require you to explicitly write code to do the conversion. This is done when a conversion must be made and information potentially could be lost or produce an error. Beware that an explicit casting can produce an unexpected result.

This type of casting can be performed only where it makes sense, such as converting from a long to an int. You cannot use it to convert from a string to an int where the format of the data has to physically change.

```csharp
int x;
long y = 1000;
x = (int)y; // Explicit casting from long to int
```

The System.Convert Class

You can do explicit conversions using the System.Convert class in cases where implicit or explicit casting isn’t possible, or if you think it’s easier for you to use this class for all conversions. The class contains conversion functions such as ToDecimal, ToInt32 and ToString.

```csharp
string myIntString = "1234";

// Conversion from string to int
int myInt = Convert.ToInt32(myIntString);
```

The TryParse Method

You can use the TryParse method on the data type to try and see if a conversion is possible. The function takes two parameters; the first is the value to parse and the second is a variable that will contain the parsed value if the conversion succeeds. The second parameter must be passed as an out parameter which means that it can pass out a value from the method.

```csharp
string parseValue = "1234";
int parsedInt = 0;
```
if (int.TryParse(parseValue, out parsedInt)) {
  // On success
}
else {
  // On failed parse
}

**Additional reading:** “[Casting and Type Conversions (C# Programming Guide)](https://example.com)”

### String Manipulation

Strings are used to store alphanumerical values and can be used, for example, to store values from text boxes in a UI.

#### String Concatenation

You can use the + operator to concatenate a string, but this method of concatenation should be used sparingly because it causes overhead; every time the + operator is used, a new string is created in memory and the old string is discarded.

```csharp
string myString = "first part";
myString = myString + "second part";
myString = myString + "third part";
```

You should instead use an instance of the `StringBuilder` class and append values to the instance.

```csharp
StringBuilder mySB =
    new StringBuilder("first part");

mySB.Append("second part");
mySB.Append("third part");

string concatenatedString = mySB.ToString();
```
String Validation

String validation is very important, especially if the value comes from a UI. One way to avoid `InvalidCastException` is to use regular expressions.

The `Regex` class is located in the `System.Text.RegularExpressions` namespace. You can use the `IsMatch` method to validate if the string matches the specified criteria.

The following code validates if the string contains numerical digits.

```csharp
var textToTest = "hell0 w0rld";
var regularExpression = "\d";
var result = Regex.IsMatch(textToTest, regularExpression, RegexOptions.None);
{
    // The text matched the expression.
}
```

Additional reading: “Regex Class”