



La Programación Básica en Entorno Visual (I)

UNIDAD N° 2

(Parte 1/3)

2023

Temas

- Trabajando con variables, operadores y expresiones. Escribir métodos y definir ámbitos.
- Usando operadores de asignación compuestos y declaraciones de iteración.
- Manejando errores y excepciones.
- Repasando: tipos por valor y referencia, arreglos.
- Creando y manejando clases y objetos. Trabajando con herencia.
- Implementando propiedades para el acceso a campos.
- Creando tipos con enumeraciones y estructuras.
- Usando arrays. Entendiendo los parámetros de arrays.
- Trabajando con interfaces y definiendo clases abstractas.
- Usando y enumerando colecciones. Consultando datos en memoria con lenguaje de consulta.

CHAPTER 2

Working with variables, operators, and expressions

Understanding statements

Tip C# is a “free format” language, which means that white space, such as a space character or a new line, is not significant except as a separator. In other words, you are free to lay out your statements in any style you choose. However, you should adopt a simple, consistent layout style to make your programs easier to read and understand.

Using identifiers

- You can use only letters (uppercase and lowercase), digits, and underscore characters.
- An identifier must start with a letter or an underscore.

camelCase

variables

parámetros

UpperCamelCase o PascalCase

clases, interfaces, enums, structs, records

métodos (públicos y privados)

propiedades

Using identifiers

Identifying keywords

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

add	global	select
alias	group	remove
and	into	set
ascending	join	value
async	let	var
await	nameof	when
descending	not	where
dynamic	or	yield
from	orderby	
get	partial	

These identifiers are
not reserved by C#



Important C# is a case-sensitive language: *footballTeam* and *FootballTeam* are two different identifiers.

Uso de @

- Don't start an identifier with an underscore. Although this is legal in C#, it can limit the interoperability of your code with applications built by using other languages, such as Microsoft Visual Basic.
- Don't create identifiers that differ only by case. For example, do not create one variable named *myVariable* and another named *MyVariable* for use at the same time, because it is too easy to get them confused. Also, defining identifiers that differ only by case can limit the ability to reuse classes in applications developed by using other languages that are not case sensitive, such as Visual Basic.
- Start the name with a lowercase letter.
- In a multiword identifier, start the second and each subsequent word with an uppercase letter. (This is called *camelCase notation*.)
- Don't use Hungarian notation. (If you are a Microsoft Visual C++ developer, you are probably familiar with Hungarian notation. If you don't know what Hungarian notation is, don't worry about it!)

Declaring variables

```
int age;  
  
age = 42;  
  
Console.WriteLine(age);
```

Unassigned local variables

```
int age;  
Console.WriteLine(age); // compile-time error
```

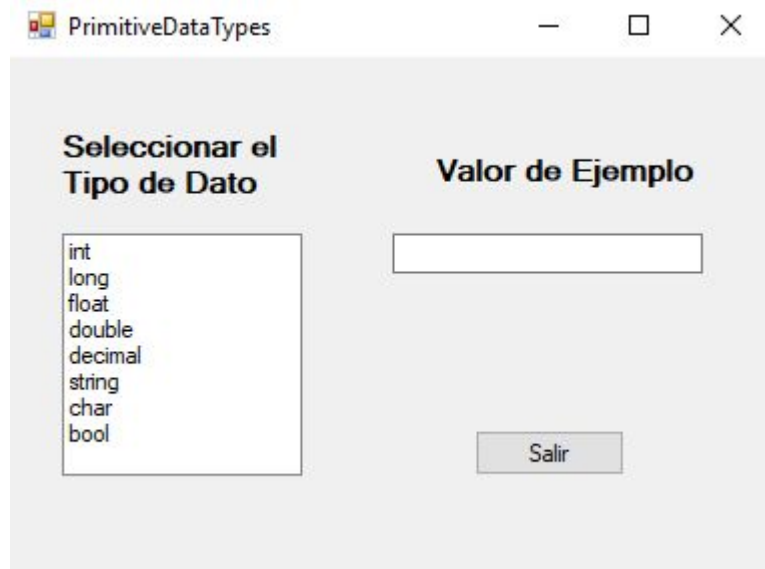
Specifying numeric values

```
float myVar; // declare a floating-point variable  
myVar = 42F; // assign a floating-point value to the variable
```

Working with primitive data types

Data type	Description	Size (bits)	Range	Sample usage
int	Whole numbers (integers)	32	-2^{31} through $2^{31} - 1$	int count; count = 42;
long	Whole numbers (bigger range)	64	-2^{63} through $2^{63} - 1$	long wait; wait = 42L;
float	Floating-point numbers	32	$\pm 1.5 \times 10^{-45}$ through $\pm 3.4 \times 10^{38}$	float away; away = 0.42F;
double	Double-precision (more accurate) floating-point numbers	64	$\pm 5.0 \times 10^{-324}$ through $\pm 1.7 \times 10^{308}$	double trouble; trouble = 0.42;
decimal	Monetary values	128	28 significant figures	decimal coin; coin = 0.42M;
string	Sequence of characters	16 bits per character	Not applicable	string vest; vest = "forty two";
char	Single character	16	0 through $2^{16} - 1$	char grill; grill = 'x';
bool	Boolean	8	True or false	bool teeth; teeth = false;

Ejercicio N° 1



The image shows a Java Swing window titled "PrimitiveDataTypes". The window has a standard title bar with a minimize button, a maximize button, and a close button. The main content area is divided into two sections. The left section is titled "Seleccionar el Tipo de Dato" and contains a list of primitive data types: int, long, float, double, decimal, string, char, and bool. The right section is titled "Valor de Ejemplo" and contains a text input field. Below the input field is a button labeled "Salir".

Seleccionar el Tipo de Dato	Valor de Ejemplo
int long float double decimal string char bool	<input type="text"/>
	<input type="button" value="Salir"/>

Using arithmetic operators

Operators and types

Not all operators are applicable to all data types. The operators that you can use on a value depend on the value's type. For example, you can use all the arithmetic operators on values of type *char*, *int*, *long*, *float*, *double*, or *decimal*. However, with the exception of the plus operator, *+*, you can't use the arithmetic operators on values of type *string*, and you cannot use any of them with values of type *bool*. So, the following statement is not allowed, because the *string* type does not support the minus operator (subtracting one string from another is meaningless):

```
// compile-time error  
Console.WriteLine("Gillingham" - "Forest Green Rovers");
```



Tip The .NET Framework provides a method called *Int32.Parse* that you can use to convert a string value to an integer if you need to perform arithmetic computations on values held as strings.

Using arithmetic operators

String interpolation

C# renders many uses of the + operator obsolete for concatenating strings.

A common use of string concatenation is to generate string values that include variable values. For example, consider the following statements:

```
string username = "John";  
string message = "Hello " + userName;
```

String interpolation lets you use the following syntax instead:

```
string username = "John";  
string message = ($"Hello {userName }");
```

The \$ symbol at the start of the string indicates that it is an interpolated string and that any expressions between the { and } characters should be evaluated, and the result substituted in their place. Without the leading \$ symbol, the string {username} would be treated literally.

String interpolation is more efficient than using the + operator; string concatenation using the + operator can be memory hungry because of how strings are handled by the .NET libraries. String interpolation is also arguably more readable and less error prone.

Numeric types and infinite values

There are other features of numbers in C# that you should be aware of. For example, the result of dividing any number by zero is infinity, which is outside the range of the `int`, `long`, and `decimal` types; consequently, evaluating an expression such as `5/0` results in error. However, the `double` and `float` types have a special value that can represent infinity, and the value of the expression `5.0/0.0` is `Infinity`. The one exception to this rule is the value of the expression `0.0/0.0`. Usually, if you divide zero by anything, the result is zero, but if you divide anything by zero, the result is infinity. The expression `0.0/0.0` results in a paradox; the value must be zero and infinity at the same time. C# has another special value for this situation, called NaN, which stands for *not a number*. So if you evaluate `0.0/0.0`, the result is NaN.

NaN and `Infinity` propagate through expressions. If you evaluate `10 + NaN`, the result is NaN, and if you evaluate `10 + Infinity`, the result is `Infinity`. The value of the expression `Infinity * 0` is NaN.

Ejercicio N° 2

MathsOperators

Operando Izquierda

Operando Derecha

☐ [+] Suma

☐ [-] Resta

☐ [*] Multiplicación

☐ [/] División

☐ [%] Resto

Calcular

Expresión:

Resultado:

Salir

Controlling precedence

Precedence governs the order in which an expression's operators are evaluated. Consider the following expression, which uses the + and * operators:

$2 + 3 * 4$

$* / \% \rightarrow + -$

Using associativity to evaluate expressions

Operator precedence is only half the story. What happens when an expression contains different operators that have the same precedence? This is where *associativity* becomes important. Associativity is the direction (left or right) in which the operands of an operator are evaluated. Consider the following expression that uses the / and * operators:

$4 / 2 * 6$



Associativity and the assignment operator

In C#, the equal sign (=) is an operator. All operators return a value based on their operands. The assignment operator = is no different. It takes two operands: the operand on the right side is evaluated and then stored in the operand on the left side. The value of the assignment operator is the value that was assigned to the left operand. For example, in the following assignment statement, the value returned by the assignment operator is 10, which is also the value assigned to the variable *myInt*:

```
int myInt;  
myInt = 10; // value of assignment expression is 10
```

```
int myInt;  
int myInt2;  
myInt2 = myInt = 10;
```

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

```
int myInt, myInt2, myInt3 = 10;  
myInt3 = myInt / myInt2;
```

the compiler generates the following errors:

```
Use of unassigned local variable 'myInt'  
Use of unassigned local variable 'myInt2'
```

Incrementing and decrementing variables

If you want to add 1 to a variable, you can use the + operator, as demonstrated here:

```
count = count + 1;
```

However, adding 1 to a variable is so common that C# provides its own operator just for this purpose: the ++ operator. To increment the variable *count* by 1, you can write the following statement:

```
count++;
```

Similarly, C# provides the -- operator that you can use to subtract 1 from a variable, like this:

```
count--;
```

The ++ and -- operators are *unary* operators, meaning that they take only a single operand. They share the same precedence and are both left-associative.

Prefix and postfix

```
count++; // postfix increment
++count; // prefix increment
count--; // postfix decrement
--count; // prefix decrement
```

```
int x;
x = 42;
Console.WriteLine(x++); // x is now 43, 42 written out
x = 42;
Console.WriteLine(++x); // x is now 43, 43 written out
```

Declaring implicitly typed local variables

```
int myInt;
```

It was also mentioned that you should assign a value to a variable before you attempt to use it. You can declare and initialize a variable in the same statement, such as illustrated in the following:

```
int myInt = 99;
```

```
var myVariable = 99;  
var myOtherVariable = "Hello";
```

```
var yetAnotherVariable; // Error - compiler cannot infer type
```

CHAPTER 3

Writing methods and applying scope

Creating methods

A *method* is a named sequence of statements. If you have previously programmed by using a language such as C, C++, or Microsoft Visual Basic, you will see that a method is similar to a function or a subroutine. A method has a name and a body. The method name should be a meaningful identifier that indicates the overall purpose of the method (*calculateIncomeTax*, for example). The method body contains the actual statements to be run when the method is called. Additionally, methods can be given some data for processing and can return information, which is usually the result of the processing. Methods are a fundamental and powerful mechanism.

```
returnType methodName ( parameterList )
{
    // method body statements go here
}
```

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
    return leftHandSide + rightHandSide;
}
```

```
void showResult(int answer)
{
    // display the answer
    ...
    return;
}
```



Important If you program in C, C++, or Visual Basic, you should note that C# does not support global methods. You must write all your methods inside a class; otherwise, your code will not compile.

Using expression-bodied methods

Some methods can be very simple, performing a single task or returning the results of a calculation without involving any additional logic. C# supports a simplified form for methods that comprise a single expression. These methods can still take parameters and return values, and they operate in the same way as the methods that you have seen so far. The following code examples show simplified versions of the `addValues` and `showResult` methods written as expression-bodied methods:

```
int addValues(int leftHandSide, int rightHandSide) => leftHandSide + rightHandSide;  
void showResult(int answer) => Console.WriteLine($"The answer is {answer}");
```



Important If you program in C, C++, or Visual Basic, you should note that C# does not support global methods. You must write all your methods inside a class; otherwise, your code will not compile.

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
}
```

```
int arg1 = 99;
int arg2 = 1;
addValues(arg1, arg2);
```

```
addValues;           // compile-time error, no parentheses
addValues();         // compile-time error, not enough arguments
addValues(39);       // compile-time error, not enough arguments
addValues("39", "3"); // compile-time error, wrong types for arguments
```

```
int result = addValues(39, 3); // on right-hand side of an assignment
showResult(addValues(39, 3));  // as argument to another method call
```

Ver múltiples
valores de
retorno

Ejercicios N° 3 y N° 4

Applying scope

```
class Example
{
    void firstMethod()
    {
        int myVar;
        ...
    }
    void anotherMethod()
    {
        myVar = 42; // error - variable not in scope
        ...
    }
}
```

```
class Example
{
    void firstMethod()
    {
        myField = 42; // ok
        ...
    }
    void anotherMethod()
    {
        myField++; // ok
        ...
    }

    int myField = 0;
}
```

Overloading methods

Ejercicio N° 5

Nesting methods

```
long CalculateFactorial(string input)
{
    int inputValue = int.Parse(input);
    long factorial (int dataValue)
    {
        if (dataValue == 1)
        {
            return 1;
        }
        else
        {
            return dataValue * factorial(dataValue - 1);
        }
    }

    long factorialValue = factorial(inputValue);
    return factorialValue;
}
```

Using optional parameters and named arguments

```
public void DoWorkWithData(int intData, float floatData, int moreIntData)
{
    ...
}
```

```
public void DoWorkWithData(int intData, float floatData)
{
    ...
}
```

```
int arg1 = 99;
float arg2 = 100.0F;
int arg3 = 101;
```

```
DoWorkWithData(arg1, arg2, arg3); // Call overload with three parameters
DoWorkWithData(arg1, arg2);       // Call overload with two parameters
```

```
public void DoWorkWithData(int intData)
{
    ...
}
```

```
public void DoWorkWithData(int moreIntData)
{
    ...
}
```



```
int arg1 = 99;
int arg3 = 101;
```

```
DoWorkWithData(arg1);
DoWorkWithData(arg3);
```

Defining optional parameters

```
void optMethod(int first, double second = 0.0, string third = "Hello")  
{  
    ...  
}
```

```
optMethod(99, 123.45, "World"); // Arguments provided for all three parameters  
optMethod(100, 54.321);         // Arguments provided for first two parameters only
```

Passing named arguments

```
optMethod(first : 99, second : 123.45, third : "World");  
optMethod(first : 100, second : 54.321);  
  
optMethod(third : "World", second : 123.45, first : 99);  
optMethod(second : 54.321, first : 100);
```

Resolving ambiguities with optional parameters and named arguments

```
void optMethod(int first, double second = 0.0, string third = "Hello")  
{  
    ...  
}
```

```
void optMethod(int first, double second = 1.0, string third = "Goodbye", int fourth = 100 )  
{  
    ...  
}
```

optMethod(1, 2.5, "World"); ok

optMethod(1, fourth : 101); ok

optMethod(1, 2.5); ?

```
optMethod(1, third : "World");  
optMethod(1);  
optMethod(second : 2.5, first : 1);
```


Ejercicio N° 6

Using decision statements

Using Boolean operators

Short-circuiting

Summarizing operator precedence and associativity

Category	Operators	Description	Associativity
Primary	() ++ --	Precedence override Post-increment Post-decrement	Left
Unary	! + - ++ --	Logical NOT Returns the value of the operand unchanged Returns the value of the operand negated Pre-increment Pre-decrement	Left
Multiplicative	* / %	Multiply Divide Division remainder (modulus)	Left

Category	Operators	Description	Associativity
Additive	+ -	Addition Subtraction	Left
Relational	<	Less than	Left
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
Equality	==	Equal to	Left
	!=	Not equal to	
Conditional AND	&&	Conditional AND	Left
Conditional OR		Conditional OR	Left
Assignment	=	Assigns the right-hand operand to the left and returns the value that was assigned	Right

Notice that the && operator and the || operator have a different precedence: && is higher than ||.

Pattern matching

```
bool validPercentage;  
int percent = ... ;  
validPercentage = (percent >= 0) && (percent <= 100);
```

Pattern matching enables you to simplify the expression to:

```
validPercentage = (percent is >= 0 and <= 100);
```

The `is` operator works with the Boolean operator `and` (known as the *conjunctive* pattern operator) and the Boolean operator `or` (the *disjunctive* pattern operator). You cannot use `&&` or `||` with `is`. Additionally, pattern matching only matches a single variable against a pattern. For example, the following code, which attempts to evaluate two variables, `dancingAbility` and `singingAbility`, won't work (it doesn't compile):

```
canMoveToNextStage = (dancingAbility is >= 7 and singingAbility is >= 7);
```

C# pattern matching also provides the `not` operator for use with `is`. It is similar to the `!` operator but works only for patterns. The following code shows another way of determining whether the value in the `percent` variable lies within the range 0 to 100:

```
validPercentage = (percent is not < 0 and not > 100);
```

Using *if* statements to make decisions

```
if ( booleanExpression )
    statement-1;
else
    statement-2;
```

```
int seconds = 0;
int minutes = 0;
...
if (seconds == 59)
{
    seconds = 0;
    minutes++;
}
else
{
    seconds++;
}
```

type variable = booleanExpression ? value or expression :
value or expression;

```
if (day == 0)
{
    dayName = "Sunday";
}
else if (day == 1)
{
    dayName = "Monday";
}
else if (day == 2)
{
    dayName = "Tuesday";
}
else if (day == 3)
{
    dayName = "Wednesday";
}
else if (day == 4)
{
    dayName = "Thursday";
}
else if (day == 5)
{
    dayName = "Friday";
}
else if (day == 6)
{
    dayName = "Saturday";
}
else
{
    dayName = "unknown";
}
```

Using *switch* statements

```
switch ( controllingExpression )
{
    case constantExpression :
        statements
        break;
    case constantExpression :
        statements
        break;
    ...
    default :
        statements
        break;
}

switch (trumps)
{
    case Hearts :
    case Diamonds :      // Fall-through allowed - no code between labels
        color = "Red";   // Code executed for Hearts and Diamonds
        break;
    case Clubs :
        color = "Black";
    case Spades :        // Error - code between labels
        color = "Black";
        break;
}
```

```
switch (day)
{
    case 0 :
        dayName = "Sunday";
        break;
    case 1 :
        dayName = "Monday";
        break;
    case 2 :
        dayName = "Tuesday";
        break;
    ...
    default :
        dayName = "Unknown";
        break;
}
```

Using switch expressions with pattern matching

```
int measurement = ...;

string range = measurement switch
{
    < 0 => "negative",
    0 => "zero",
    >= 1 and <= 9 => "singledigit",
    >= 10 and <= 99 => "doubledigit",
    >= 100 => "large"
};
```

Using switch expressions with pattern matching

The syntactic points to note are:

- The `switch` keyword acts as an operator. The measurement variable in the example and the pattern-matching block (the code between the braces) are the operands.
- Each line in the pattern-matching block comprises a pattern and an expression to evaluate as the result. The character sequence `=>` separates the two elements.
- Each pattern/expression pair is separated by a comma. There is no `break` statement.
- The patterns follow the same syntax described for those in `if` statements described earlier.
- The expression to the right of `=>` is evaluated if the pattern matches the first operand of the `switch` operator.
- The `switch` expression must catch every possible value that can be found in the first operand. At runtime, if there is no matching pattern for the operand, you'll receive the error message "The switch expression does not handle all possible values of its input type (it is not exhaustive)." *You can use the underscore character as a catchall that matches any cases not previously handled.*

CHAPTER 5

Using compound assignment and iteration statements

Using compound assignment operators

Don't write this	Write this
<code>variable = variable * number;</code>	<code>variable *= number;</code>
<code>variable = variable / number;</code>	<code>variable /= number;</code>
<code>variable = variable % number;</code>	<code>variable %= number;</code>
<code>variable = variable + number;</code>	<code>variable += number;</code>
<code>variable = variable - number;</code>	<code>variable -= number;</code>



Tip The compound assignment operators share the same precedence and right associativity as the corresponding simple assignment operators.



Tip Use the increment (++) and decrement (--) operators instead of a compound assignment operator when incrementing or decrementing a variable by 1. For example, replace

`count += 1;`

with:

`count++;`

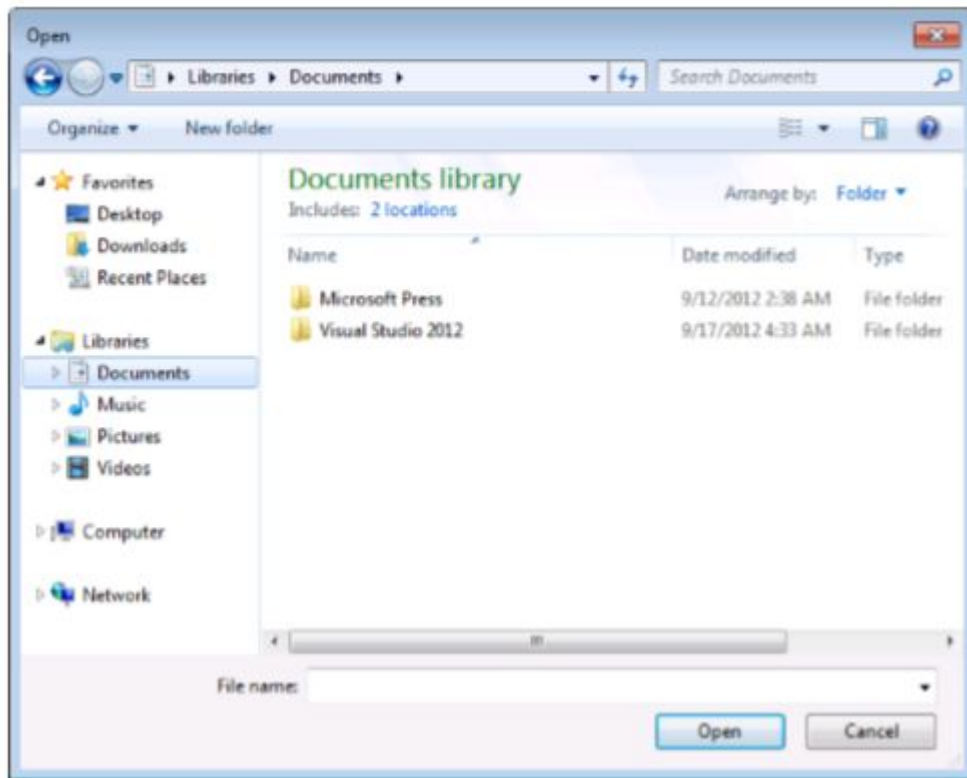
Writing *while* statements

You use a *while* statement to run a statement repeatedly for as long as some condition is true. The syntax of a *while* statement is as follows:

```
while ( booleanExpression )  
    statement
```

```
int i = 0;  
while (i < 10)  
{  
    Console.WriteLine(i);  
    i++;  
}
```

Ejercicio N° 7



Writing *for* Statements

```
for (initialization; Boolean expression; update control variable)
    statement
```

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
    ...
}
```

Writing *do* statements

```
do
    statement
while (booleanExpression);
```

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
}
while (i < 10);
```

The *break* and *continue* statements

In Chapter 4, you saw how to use the *break* statement to jump out of a *switch* statement. You can also use a *break* statement to jump out of the body of an iteration statement. When you break out of a loop, the loop exits immediately and execution continues at the first statement that follows the loop. Neither the update nor the continuation condition of the loop is rerun.

In contrast, the *continue* statement causes the program to perform the next iteration of the loop immediately (after reevaluating the Boolean expression). Here's another version of the example that writes the values 0 through 9 to the console, this time using *break* and *continue* statements:

```
int i = 0;
while (true)
{
    Console.WriteLine("continue " + i);
    i++;
    if (i < 10)
        continue;
    else
        break;
}
```

This code is absolutely ghastly. Many programming guidelines recommend using *continue* cautiously or not at all because it is often associated with hard-to-understand code. The behavior of *continue* is also quite subtle. For example, if you execute a *continue* statement from within a *for* statement, the update part runs before performing the next iteration of the loop.

Ejercicio N° 8

TRABAJO PRÁCTICO N° 2

CHAPTER 6

Managing errors and exceptions

```
try
{
    var numerador = int.Parse("1");
    var denominador = int.Parse("1");
    var operacion = numerador / denominador;
    long numero = int.MaxValue;
    numerador = int.Parse((numero + 1).ToString());
}
catch (FormatException fex)
{
    Console.WriteLine(fex.Message);
}
catch (DivideByZeroException dzex)
{
    Console.WriteLine(dzex.Message);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```



Important Notice that after catching an exception, execution continues in the method containing the *catch* block that caught the exception. If the exception occurred in a method other than the one containing the *catch* handler, control does *not* return to the method that caused the exception.

Filtering exceptions

You can filter the exceptions that are matched against catch handlers to ensure that a catch handler is triggered only when additional conditions are met. These conditions take the form of a Boolean expression prefixed by the *when* keyword. The following example illustrates the syntax:

```
bool catchErrors = ...;
try
{
    ...
}
catch (Exception ex) when (catchErrors == true)
{
    // Handle exceptions only if the catchErrors variable is true
}
```

This example catches all exceptions (the `Exception` type) depending on the value of the `catchErrors` Boolean variable. If this variable is `false`, then no exception handling occurs, and the default exception handling mechanism for the application is used. If `catchErrors` is `true`, then the code in the `catch` block runs to handle the exception.

Ejercicio N° 9

Writing checked statements

A checked statement is a block preceded by the *checked* keyword. All integer arithmetic in a checked statement always throws an *OverflowException* if an integer calculation in the block overflows, as shown in this example:

```
int number = int.MaxValue;
checked
{
    int willThrow = number++;
    Console.WriteLine("this won't be reached");
}
```



Important Only integer arithmetic directly inside the *checked* block is subject to overflow checking. For example, if one of the checked statements is a method call, checking does not apply to code that runs in the method that is called.

You can also use the *unchecked* keyword to create an *unchecked* block statement. All integer arithmetic in an *unchecked* block is not checked and never throws an *OverflowException*. For example:

```
int number = int.MaxValue;
unchecked
{
    int wontThrow = number++;
    Console.WriteLine("this will be reached");
}
```

Throwing exceptions

```
public static String monthName(int month)
{
    switch (month)
    {
        case 1 :
            return "January";
        case 2 :
            return "February";
        ...
        case 12 :
            return "December";
        default :
            throw new ArgumentOutOfRangeException("Bad month");
    }
}
```

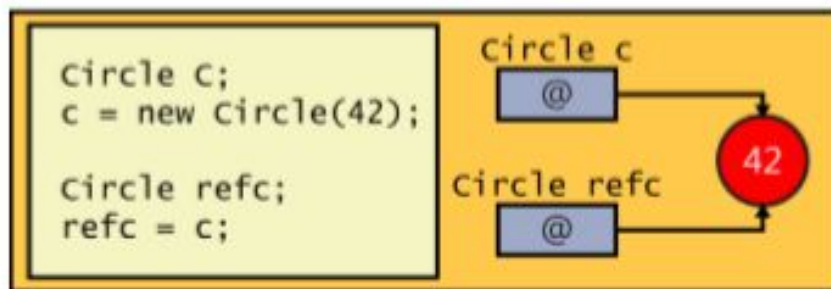
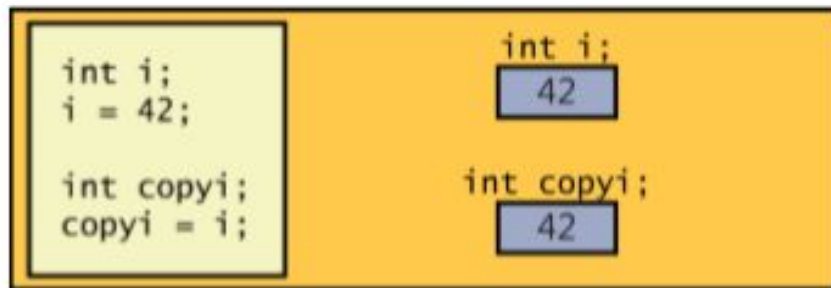
Ejercicio N° 10

Using a *finally* block

```
TextReader reader = ...;
...
try
{
    string line = reader.ReadLine();
    while (line != null)
    {
        ...
        line = reader.ReadLine();
    }
}
finally
{
    if (reader != null)
    {
        reader.Dispose();
    }
}
```

CHAPTER 8

Understanding values and references



Copying reference types and data privacy

If you actually want to copy the contents of a `Circle` object, `c`, into a different `Circle` object, `refc`, instead of just copying the reference, you must make `refc` refer to a new instance of the `Circle` class and then copy the data, field by field, from `c` into `refc`, like this:

```
var refc = new Circle();  
refc.radius = c.radius; // Don't try this
```

However, if any members of the `Circle` class are private (like the `radius` field), you won't be able to copy this data. Instead, you can make the data in the private fields accessible by exposing them as properties and then use these properties to read the data from `c` and copy it into `refc`. You'll learn how to do this in Chapter 15, "Implementing properties to access fields."

Alternatively, a class could provide a `Clone` method that returns another instance of the same class but populated with the same data. The `Clone` method would have access to the private data in an object and could copy this data directly to another instance of the same class. For example, the `Clone` method for the `Circle` class could be defined as shown here:

```
class Circle  
{  
    private int radius;  
    // Constructors and other methods omitted  
    ...  
    public Circle Clone()  
    {  
        // Create a new Circle object  
        Circle clone = new Circle();  
  
        // Copy private data from this to clone  
        clone.radius = this.radius;  
  
        // Return the new Circle object containing the copied data  
        return clone;  
    }  
}
```

Ejercicio N° 11