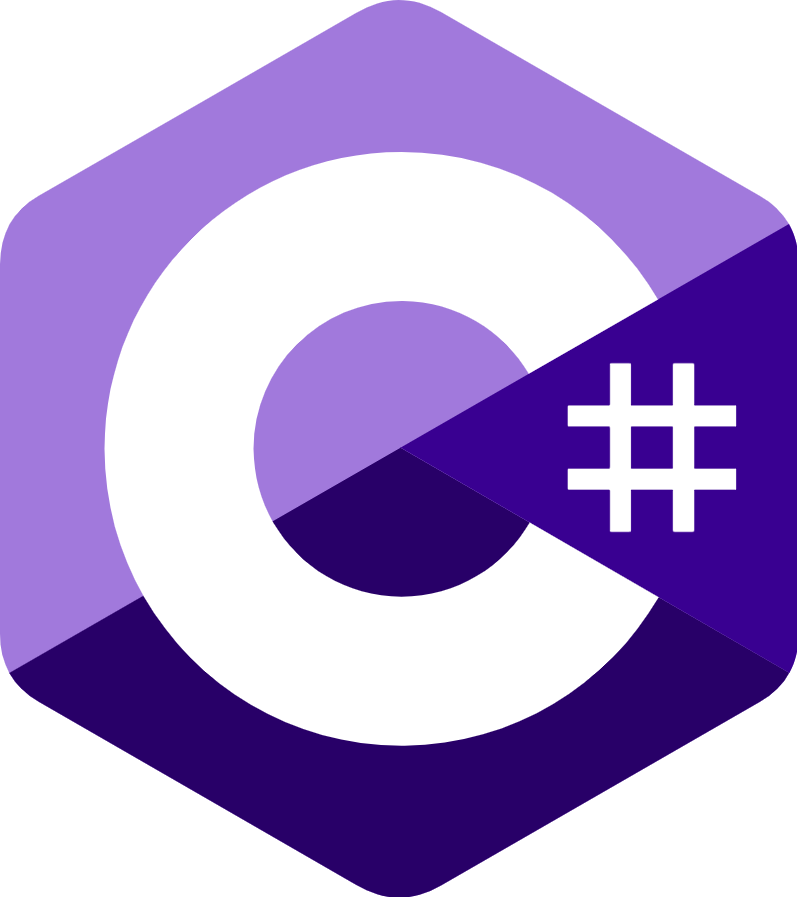
# Title

LẬP TRÌNH C# - SOLOLEARN

**Student : Green Wolf**

**Date : 27 / 12 / 2020**



# Table of Contents

[Title 1](#_Toc60203167)

[Table of Contents 2](#_Toc60203168)

[1. Basic Concepts 5](#_Toc60203169)

[1. What is C#? 5](#_Toc60203170)

[2. Variables 6](#_Toc60203171)

[3. Your First C# Program 9](#_Toc60203172)

[4. Printing Text 13](#_Toc60203173)

[5. Getting User Input 14](#_Toc60203174)

[6. Comments 16](#_Toc60203175)

[7. The var Keyword 18](#_Toc60203176)

[8. Constants 19](#_Toc60203177)

[9. Arithmetic Operators 20](#_Toc60203178)

[10. Assignment & Increment Operators 23](#_Toc60203179)

[11. Module 1 Quiz 27](#_Toc60203180)

[2. Conditionals and Loops 30](#_Toc60203181)

[1. The if-else Statement 30](#_Toc60203182)

[2. The switch Statement 36](#_Toc60203183)

[3. The while Loop 41](#_Toc60203184)

[4. The for Loop 45](#_Toc60203185)

[5. The do-while Loop 48](#_Toc60203186)

[6. break and continue 50](#_Toc60203187)

[7. Logical Operators 52](#_Toc60203188)

[8. The Conditional Operator 56](#_Toc60203189)

[9. Basic Calculator 57](#_Toc60203190)

[10. Module 2 Quiz 60](#_Toc60203191)

[3. Methods 63](#_Toc60203192)

[1. Introduction to Methods 63](#_Toc60203193)

[2. Method Parameters 67](#_Toc60203194)

[3. Multiple Parameters 70](#_Toc60203195)

[4. Optional & Named Arguments 72](#_Toc60203196)

[5. Passing Arguments 76](#_Toc60203197)

[6. Method Overloading 80](#_Toc60203198)

[7. Recursion 83](#_Toc60203199)

[8. Making a Pyramid 84](#_Toc60203200)

[9. Module 3 Quiz 87](#_Toc60203201)

[4. Classes & Objects 90](#_Toc60203202)

[1. Classes & Objects 90](#_Toc60203203)

[2. Value & Reference Types 92](#_Toc60203204)

[3. Class Example 94](#_Toc60203205)

[4. Encapsulation 97](#_Toc60203206)

[5. Constructors 101](#_Toc60203207)

[6. Properties 105](#_Toc60203208)

[7. Module 4 Quiz 112](#_Toc60203209)

[5. Arrays & Strings 117](#_Toc60203210)

[1. Arrays 117](#_Toc60203211)

[2. Using Arrays in Loops 120](#_Toc60203212)

[3. Multidimensional Arrays 124](#_Toc60203213)

[4. Jagged Arrays 127](#_Toc60203214)

[5. Array Properties & Methods 128](#_Toc60203215)

[6. Working with Strings 131](#_Toc60203216)

[7. Module 5 Quiz 134](#_Toc60203217)

[6. More On Classes 137](#_Toc60203218)

[1. Destructors 137](#_Toc60203219)

[2. Static Members 138](#_Toc60203220)

[3. Static Classes 142](#_Toc60203221)

[4. this & readonly 145](#_Toc60203222)

[5. Indexers 147](#_Toc60203223)

[6. Operator Overloading 150](#_Toc60203224)

[7. Module 6 Quiz 154](#_Toc60203225)

[7. Inheritance & Polymorphism 159](#_Toc60203226)

[1. Inheritance 159](#_Toc60203227)

[2. Protected Members 163](#_Toc60203228)

[3. Derived Class Constructor & Destructor 167](#_Toc60203229)

[4. Polymorphism 170](#_Toc60203230)

[5. Abstract Classes 175](#_Toc60203231)

[6. Interfaces 178](#_Toc60203232)

[7. Nested Classes 181](#_Toc60203233)

[8. Namespaces 182](#_Toc60203234)

[9. Module 7 Quiz 185](#_Toc60203235)

[8. Structs, Enums, Exceptions & Files 189](#_Toc60203236)

[1. Structs 189](#_Toc60203237)

[2. Enums 191](#_Toc60203238)

[3. Exception Handling 194](#_Toc60203239)

[4. Working with Files 200](#_Toc60203240)

[5. Module 8 Quiz 203](#_Toc60203241)

[9. Generics 207](#_Toc60203242)

[1. Generic Methods 207](#_Toc60203243)

[2. Generic Classes 210](#_Toc60203244)

[3. Collections 212](#_Toc60203245)

[4. Lists and BitArray 215](#_Toc60203246)

[5. Stack & Queue 222](#_Toc60203247)

[6. Dictionary & HashSet 226](#_Toc60203248)

[7. Module 9 Quiz 231](#_Toc60203249)

[Dictionary 238](#_Toc60203250)

[Noun 238](#_Toc60203251)

[Verb 241](#_Toc60203252)

[Adjective 243](#_Toc60203253)

[Other 244](#_Toc60203254)

[End – 30 / 12 / 2020! 244](#_Toc60203255)

# Basic Concepts

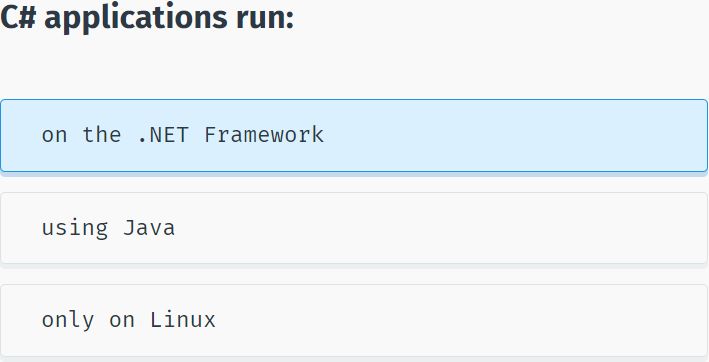
## What is C#?

**Welcome to C#**

C# is an elegant object-oriented language that enables developers to build a variety of secure and robust applications that run on the **.NET Framework.**

You can use C# to create Windows applications, Web services, mobile applications, client-server applications, database applications, and much, much more.

You will learn more about these concepts in the upcoming lessons!



**The .NET Framework**

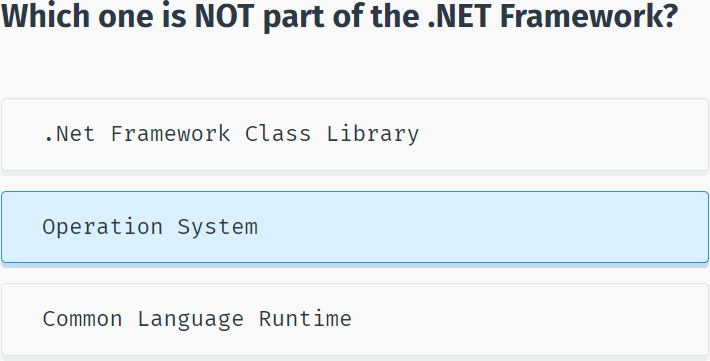
The .NET Framework consists of the **Common Language Runtime (CLR)** and the .NET Framework **class library**.

The **CLR** is the foundation of the .NET Framework. It manages code at execution time, providing core services such as memory management, code accuracy, and many other aspects of your code.

The **class library** is a collection of classes, interfaces, and value types that enable you to accomplish a range of common programming tasks, such as data collection, file access, and working with text.

C# programs use the .NET Framework class library extensively to do common tasks and provide various functionalities.

These concepts might seem complex, but for now just remember that applications written in C# use the **.NET Framework** and its components.



## Variables

**Variables**

Programs typically use data to perform tasks.

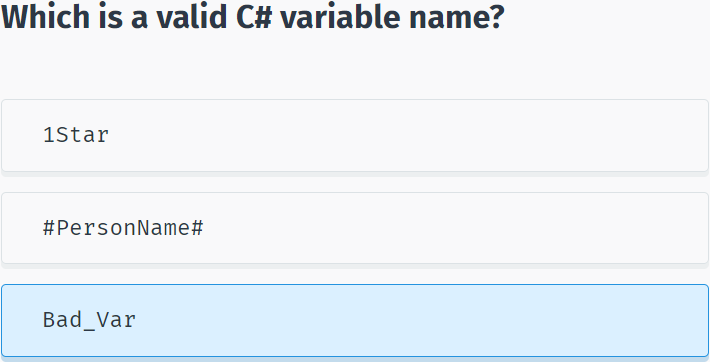
Creating a **variable** reserves a memory location, or a space in memory, for storing values. It is called **variable** because the information stored in that location can be changed when the program is running.

To use a variable, it must first be declared by specifying the **name** and **data type**.

A variable name, also called an **identifier**, can contain letters, numbers and the underscore character (\_) and must start with a letter or underscore.

Although the name of a variable can be any set of letters and numbers, the best identifier is descriptive of the data it will contain. This is very important in order to create clear, understandable and readable code!

For example, **firstName** and **lastName** are good descriptive variable names, while **abc** and **xyz** are not.



**Variable Types**

A **data type** defines the information that can be stored in a variable, the size of needed memory and the operations that can be performed with the variable.

For example, to store an integer value (a whole number) in a variable, use the **int** keyword:

int myAge;

The code above declares a variable named **myAge** of type **integer**.

A line of code that completes an action is called a statement. Each statement in C# must end with a **semicolon “;”**.

You can assign the value of a variable when you declare it:

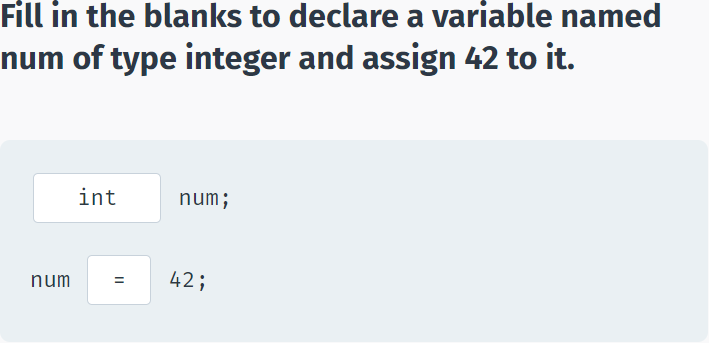
int myAge = 18;

or later in your code:

int myAge;

myAge = 18;

Remember that you need to declare the variable before using it.



**Built-in Data Types**

There are a number of built-in data types in C#. The most common are:

**int** - integer.

**float** - floating point number.

**double** - double-precision version of float.

**char** - a single character.

**bool** - Boolean that can have only one of two values: True or False.

**string** - a sequence of characters.

The statements below use C# data types:

int ​x = 42;

double ​pi = 3.14;

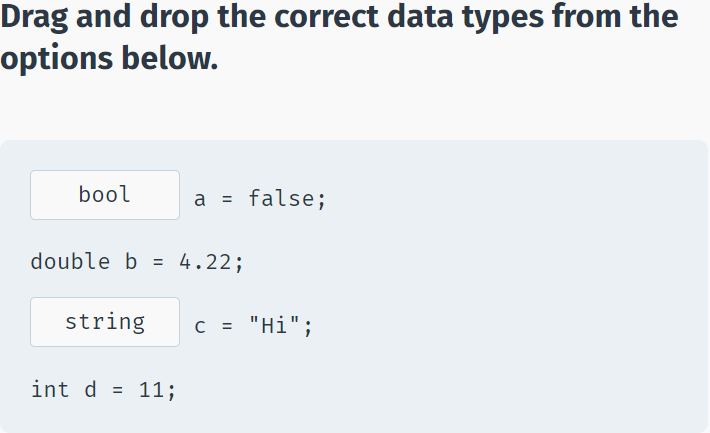
char y = 'Z’;

bool ​isOnline = true;

string ​firstName = "David”;

Note that **char** values are assigned using single quotes and **string** values require double quotes.

You will learn how to perform different operations with variables in the upcoming lessons!



## Your First C# Program

**Your First C# Program**

You can run, save, and share your C# codes on our **Code Playground**, without installing any additional software.

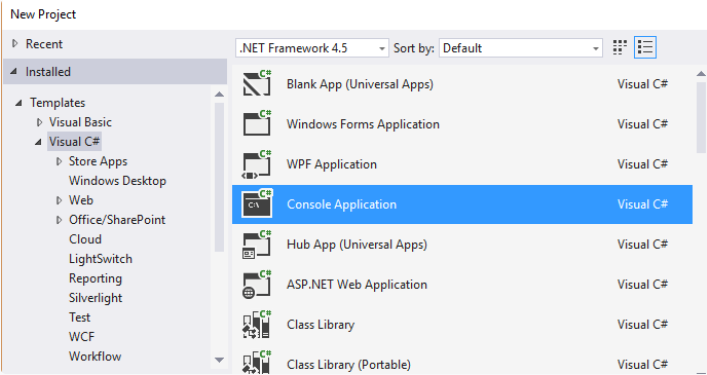
Reference this lesson if you need to install the software on your computer.

To create a C# program, you need to install an integrated development environment (IDE) with coding and debugging tools.

We will be using **Visual Studio Community Edition**, which is available to download for free.

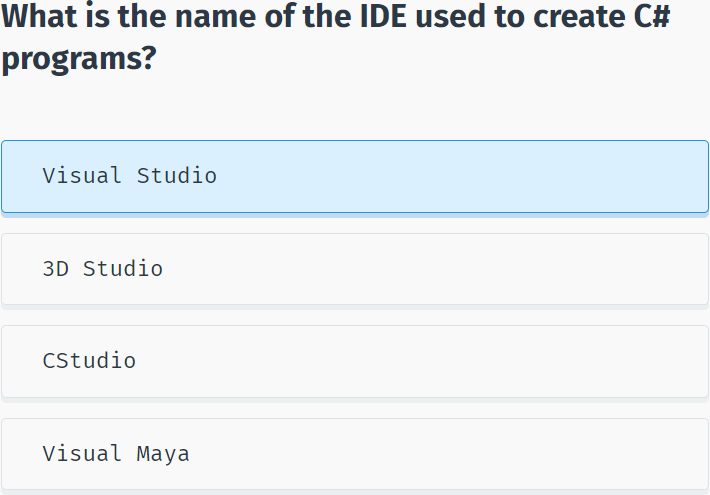
After installing it, choose the default configuration.

Next, click **File->New->Project** and then choose **Console Application** as shown below:



Enter a name for your Project and click OK.

**Console application** uses a text-only interface. We chose this type of application to focus on learning the fundamentals of C#.



Visual Studio will automatically generate some code for your project:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

}

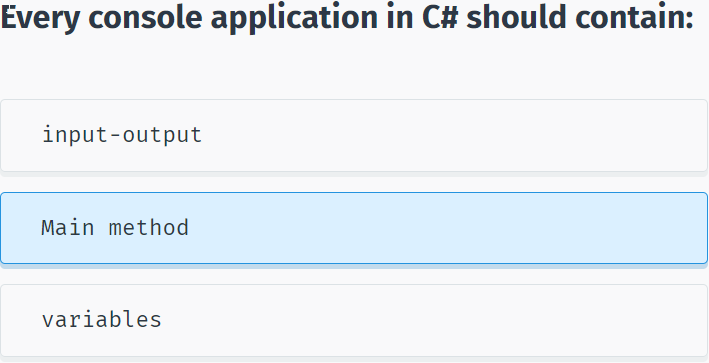
}

}

You will learn what each of the statements does in the upcoming lessons.

For now, remember that every C# console application must contain a **method (a function) named Main**. Main is the starting point of every application, i.e. the point where our program starts execution from.

We will learn about classes, methods, arguments, and namespaces in the upcoming lessons.

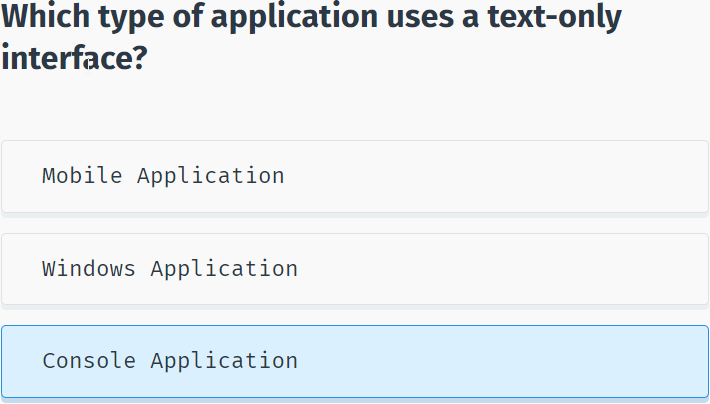


To run your program, press **Ctrl+F5**. You will see the following screen:



This is a console window. As we did not have any statements in our **Main** method, the program just produces a general message. Pressing any key will close the console.

Congratulations, you just created your first C# program.



## Printing Text

**Displaying Output**

Most applications require some **input** from the user and give **output** as a result.

To display text to the console window you use the **Console.Write** or **Console.WriteLine** methods. The difference between these two is that **Console.WriteLine** is followed by a line terminator, which moves the cursor to the next line after the text output.

The program below will display Hello World! to the console window:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

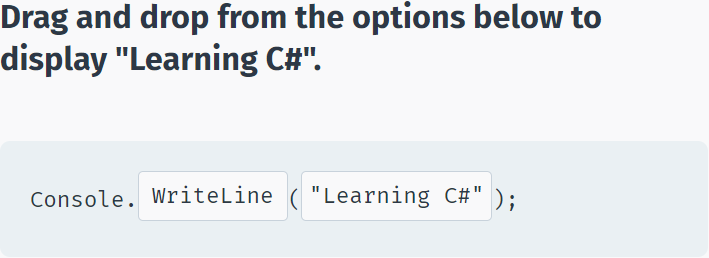
Console.WriteLine("Hello World!");

}

}

}

Note the **parentheses** after the **WriteLine** method. This is the way to pass data, or arguments, to methods. In our case **WriteLine** is the method and we pass "Hello World!" to it as an argument. String arguments must be enclosed in quotation marks.



We can display variable values to the console window:

int x = 89;

Console.WriteLine(x);//89

To display a **formatted string**, use the following syntax:

int x = 10;

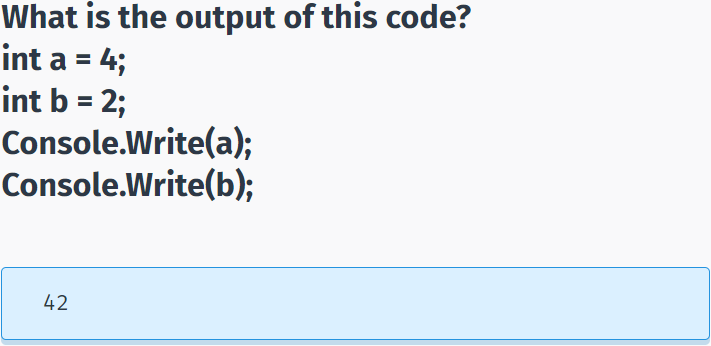
double y = 20;

Console.WriteLine("x = {0}; y = {1}", x, y);

//x = 10; y = 20

As you can see, the value of **x** replaced **{0}** and the value of **y** replaced **{1}**.

You can have as many variable placeholders as you need. (i.e.: {3}, {4}, etc.).



## Getting User Input

**User Input**

You can also prompt the user to enter data and then use the **Console.ReadLine** method to assign the input to a string variable.

The following example asks the user for a name and then displays a message that includes the input:

string yourName;

Console.WriteLine("What is your name?");

yourName = Console.ReadLine();//->Green

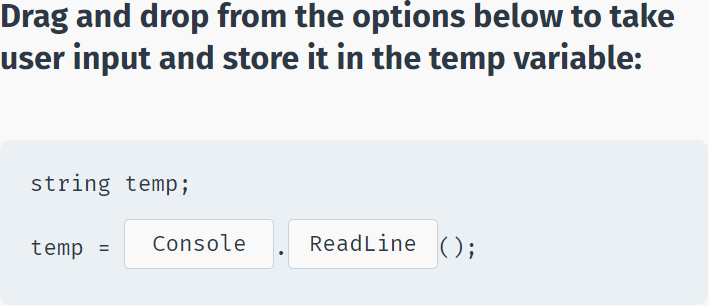
Console.WriteLine("Hello {0}", yourName);

//What is your name?

//Green

The **Console.ReadLine** method waits for user input and then assigns it to the variable. The next statement displays a formatted string containing Hello with the user input. For example, if you enter David, the output will be Hello David.

Note the empty parentheses in the **ReadLine** method. This means that it does not take any arguments.



The **Console.ReadLine()** method returns a **string** value.

If you are expecting another type of value (such as int or double), the entered data must be converted to that type.

This can be done using the **Convert.ToXXX** methods, where XXX is the .NET name of the type that we want to convert to. For example, methods include **Convert.ToDouble** and **Convert.ToBoolean**.

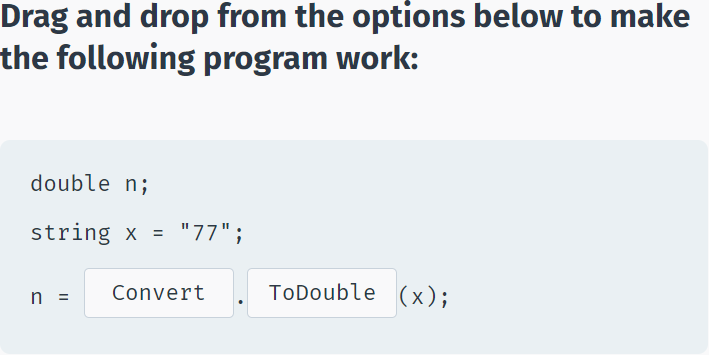
For integer conversion, there are three alternatives available based on the bit size of the integer: **Convert.ToInt16**, **Convert.ToInt32** and **Convert.ToInt64**. The default int type in C# is 32-bit.

Let’s create a program that takes an integer as input and displays it in a message:

int age = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("You are {0} years old", age);

If, in the program above, a non-integer value is entered (for example, letters), the **Convert** will fail and cause an error.



## Comments

**Comments**

**Comments** are explanatory statements that you can include in a program to benefit the reader of your code.

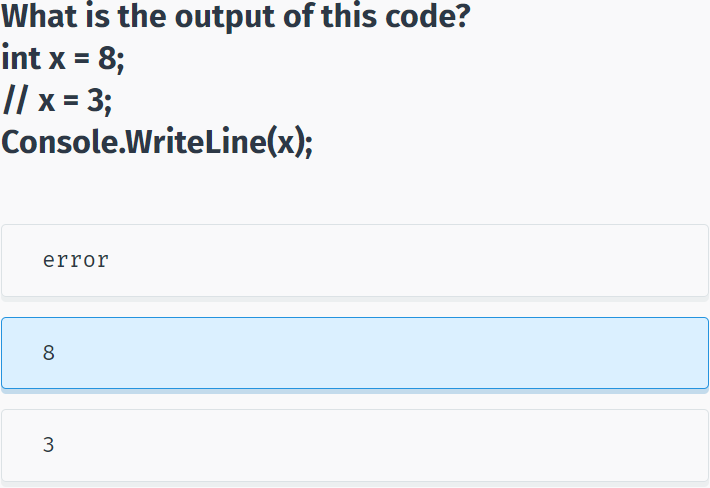
The compiler ignores everything that appears in the comment, so none of that information affects the result.

A comment beginning with two slashes (//) is called a single-line comment. The slashes tell the compiler to ignore everything that follows, until the end of the line.

// Prints Hello

Console.WriteLine("Hello");

When you run this code, Hello will be displayed to the screen. The // Prints Hello line is a comment and will not appear as output.



**Multi-Line Comments**

Comments that require multiple lines begin with /\* and end with \*/ at the end of the comment block.

You can place them on the same line or insert one or more lines between them.

/\* Some long

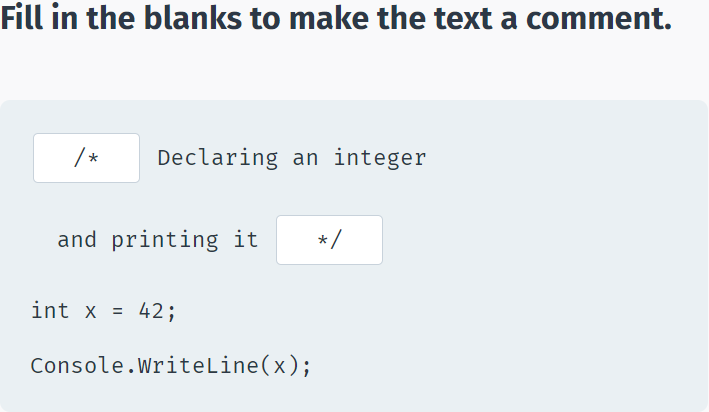
comment text

\*/

int x = 42;

Console.WriteLine(x);//42

Adding comments to your code is good programming practice. It facilitates a clear understanding of the code for you and for others who read it.



## The var Keyword

**The var Keyword**

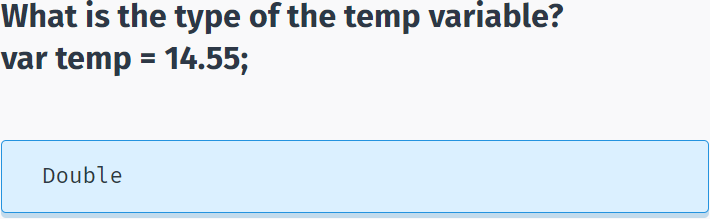
A variable can be explicitly declared with its **type** before it is used.

Alternatively, C# provides a handy function to enable the compiler to determine the type of the variable automatically based on the expression it is assigned to.

**The var keyword is used for those scenarios:**

var num = 15;

The code above makes the compiler determine the type of the variable. Since the value assigned to the variable is an integer, the variable will be declared as an integer automatically.



Variables declared using the var keyword are called **implicitly typed** variables.

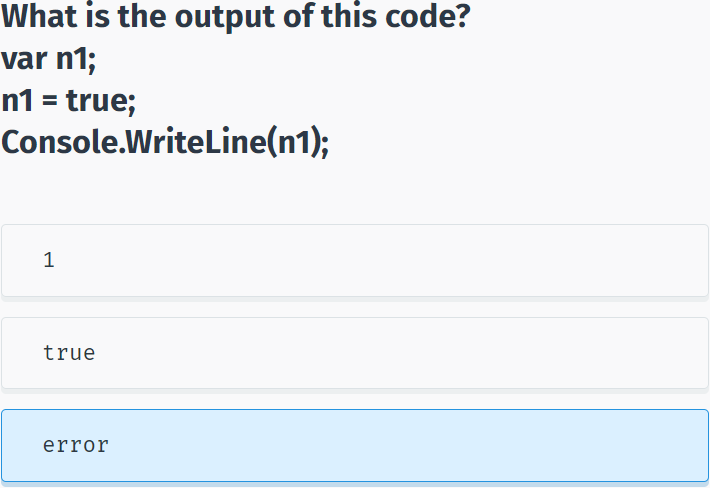
Implicitly typed variables must be initialized with a value.

**For example, the following program will cause an error:**

var num;

num = 42;->error!

Although it is easy and convenient to declare variables using the **var** keyword, overuse can harm the readability of your code. Best practice is to explicitly declare variables.



## Constants

**Constants**

**Constants** store a value that cannot be changed from their initial assignment.

To declare a constant, use the **const** modifier.

**For example:**

const double PI = 3.14;

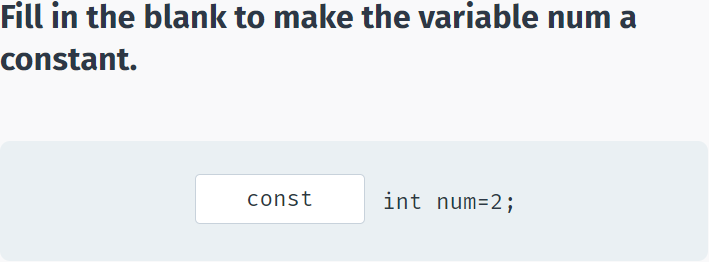
The value of const PI cannot be changed during program execution.

For example, an assignment statement later in the program will cause an error:

const double PI = 3.14;

PI = 8; ->error

Constants **must** be initialized with a value when declared.



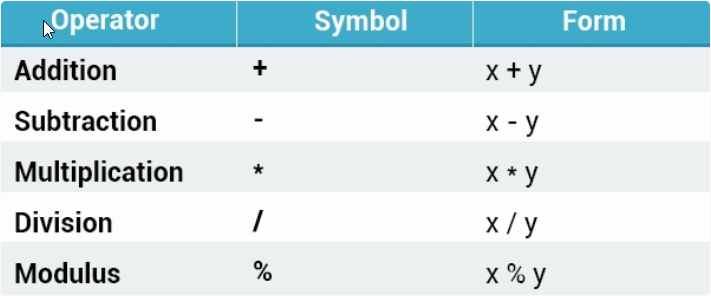
## Arithmetic Operators

**Operators**

An **operator** is a symbol that performs mathematical or logical manipulations.

**Arithmetic Operators**

C# supports the following arithmetic operators:



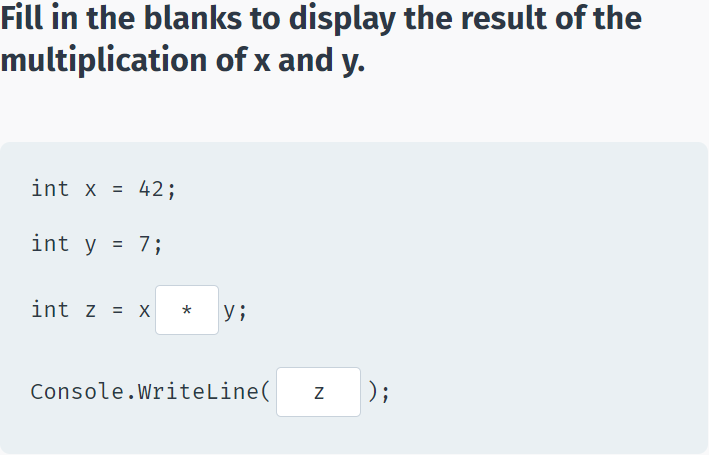
**For example:**

int x = 10;

int y = 4;

Console.WriteLine(x-y);//6

Tap **Try It Yourself** to play around with the code!



**Division**

The division operator (/) divides the first operand by the second. If the operands are both integers, any remainder is dropped in order to return an integer value.

**Example:**

int x = 10 / 4;

Console.WriteLine(x);//2

Division by 0 is undefined and will crash your program.



**Modulus**

The modulus operator (%) is informally known as the remainder operator because it returns the remainder of an integer division.

**For example:**

int x = 25 % 7;

Console.WriteLine(x);//4



**Operator Precedence**

Operator **precedence** determines the grouping of terms in an expression, which affects how an expression is evaluated. Certain operators take higher precedence over others; for example, the multiplication operator has higher precedence than the addition operator.

**For example:**

int x = 4+3\*2;

Console.WriteLine(x);//10

The program evaluates 3\*2 first, and then adds the result to 4.

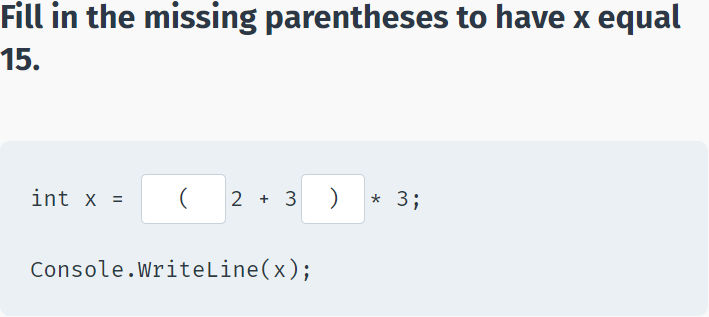
As in mathematics, using **parentheses** alters operator precedence.

int x = (4 + 3) \*2;

Console.WriteLine(x);//14

The operations within parentheses are performed first. If there are parenthetical expressions nested within one another, the expression within the innermost parentheses is evaluated first.

If none of the expressions are in parentheses, multiplicative (multiplication, division, modulus) operators will be evaluated before additive (addition, subtraction) operators. Operators of equal precedence are evaluated from left to right.



## Assignment & Increment Operators

**Assignment Operators**

The = **assignment** operator assigns the value on the right side of the operator to the variable on the left side.

C# also provides **compound assignment operators** that perform an operation and an assignment in one statement.

**For example:**

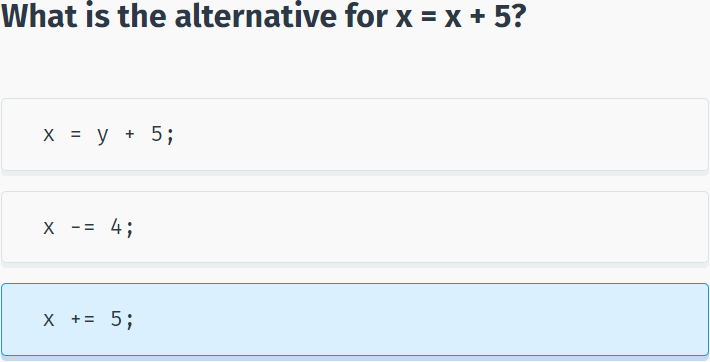
int x = 42;

x += 2; // equivalent to x = x + 2

Console.WriteLine(x);//44

x -= 6; // equivalent to x = x - 6

Console.WriteLine(x);//38



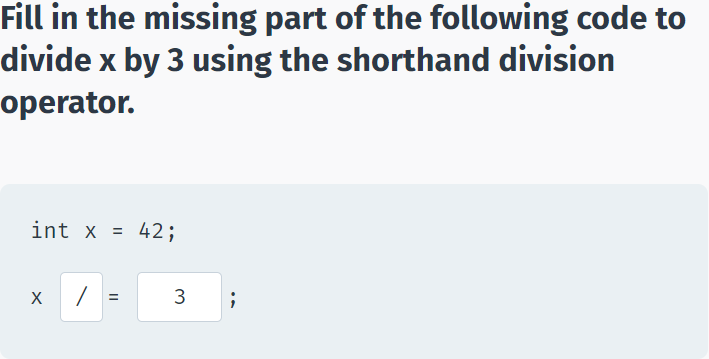
The same shorthand syntax applies to the multiplication, division, and modulus operators.

x \*= 8; // equivalent to x = x \* 8

x /= 5; // equivalent to x = x / 5

x %= 2; // equivalent to x = x % 2

The same shorthand syntax applies to the multiplication, division, and modulus operators.



**Increment Operator**

The **increment** operator is used to increase an integer's value by one, and is a commonly used C# operator.

x++; ​//equivalent to x = x + 1

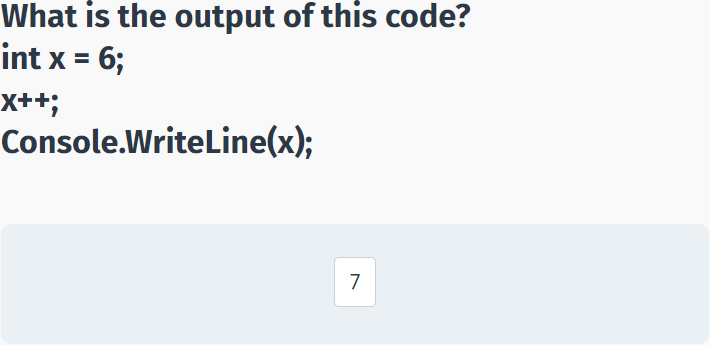
**For example:**

int x = 10;

x++;

Console.WriteLine(x);//11

The increment operator is used to increase an integer's value by one.



**Prefix & Postfix Forms**

The increment operator has two forms, **prefix** and **postfix**

++x; //prefix

x++; //postfix

**Prefix** increments the value, and then proceeds with the expression.

**Postfix** evaluates the expression and then performs the incrementing.

**Prefix example:**

int x = 3;

int y = ++x;

//x is 4; y is 4

**Postfix example:**

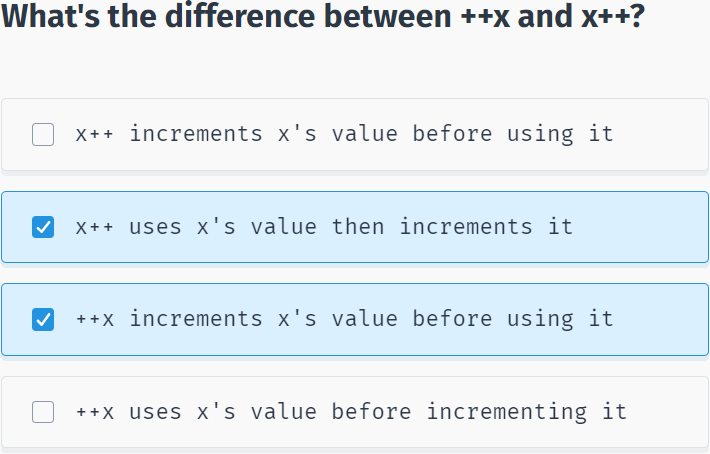
int x = 3;

int y = x++;

//x is 4; y is 3

The **prefix** example increments the value of x, and then assigns it to y.

The **postfix** example assigns the value of x to y, and then increments x.



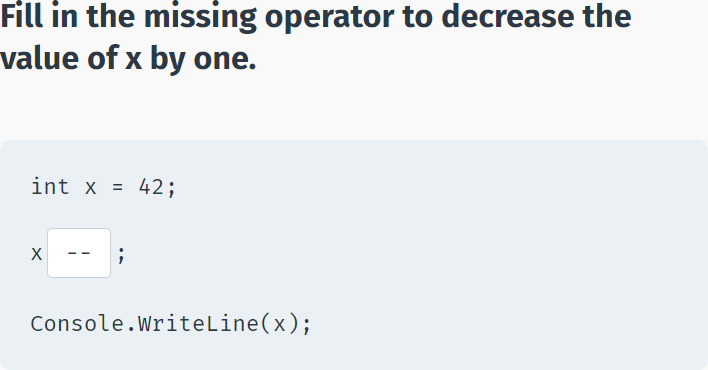
**Decrement Operator**

The **decrement** operator (--) works in much the same way as the increment operator, but instead of increasing the value, it decreases it by one.

--x; // prefix

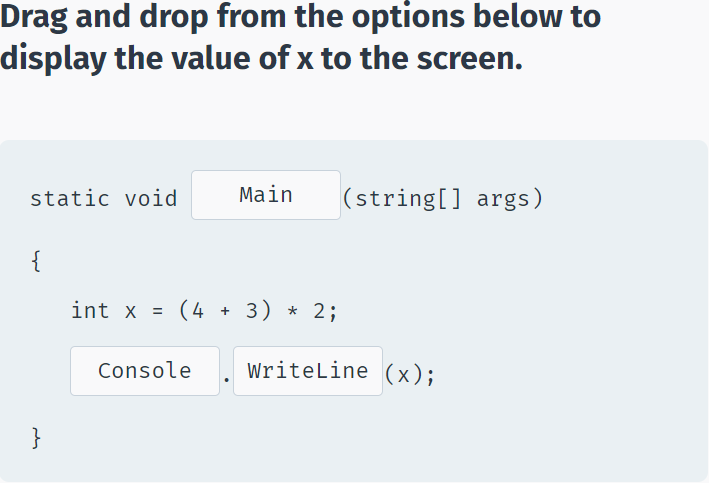
x--; // postfix

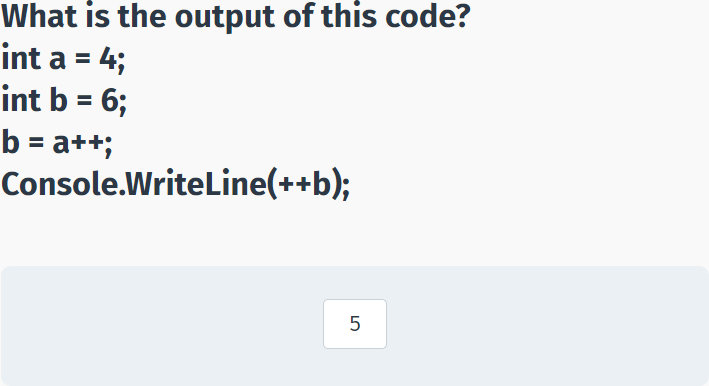
The decrement operator (--) works in much the same way as the increment operator.

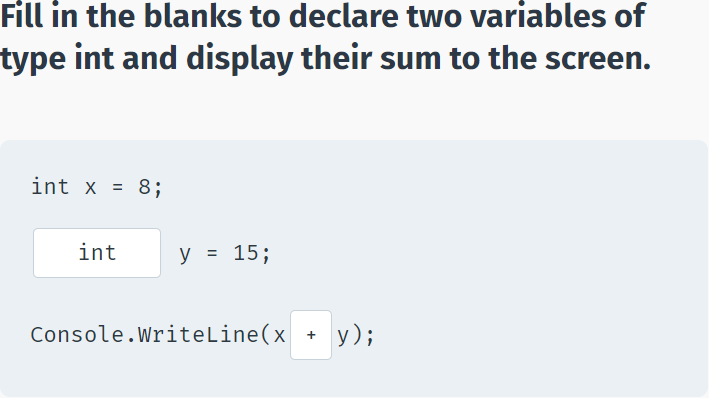


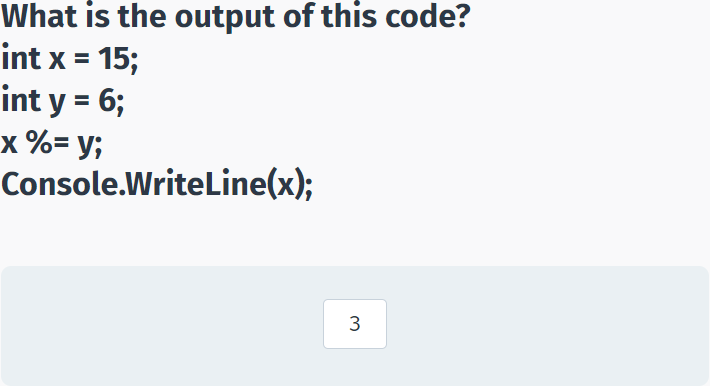
## Module 1 Quiz











# Conditionals and Loops

## The if-else Statement

**The if Statement**

The **if** statement is a conditional statement that executes a block of code when a condition is true.

**The general form of the if statement is:**

if (condition)

{

// Execute this code when condition is true

}

The condition can be any expression that returns true or false.

**For example:**

int x = 8;

int y = 3;

if (x > y)

{

Console.WriteLine("x is greater than y");

//x is greater than y

}

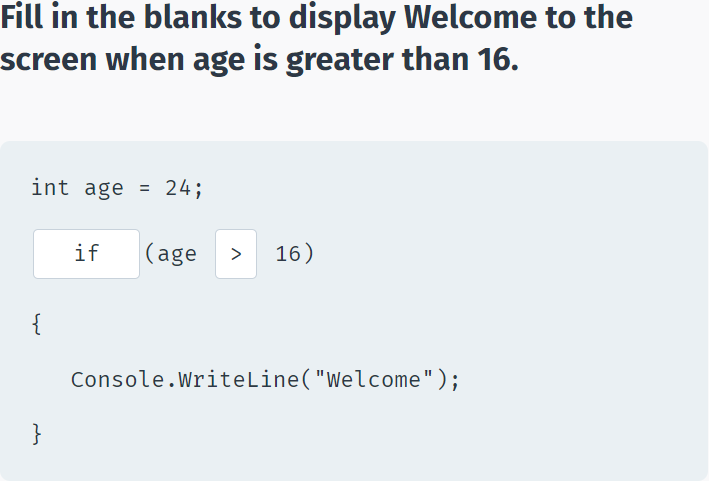
The code above will evaluate the condition **x > y**. If it is true, the code inside the if block will execute.

When only one line of code is in the if block, the curly braces can be omitted.

For example:

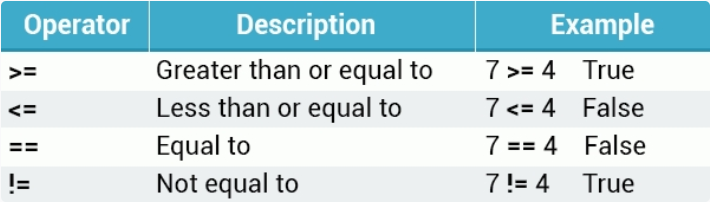
if (x > y)

Console.WriteLine("x is greater than y");



**Relational Operators**

Use **relational operators** to evaluate conditions. In addition to the less than (<) and greater than (>) operators, the following operators are available:



**Example:**

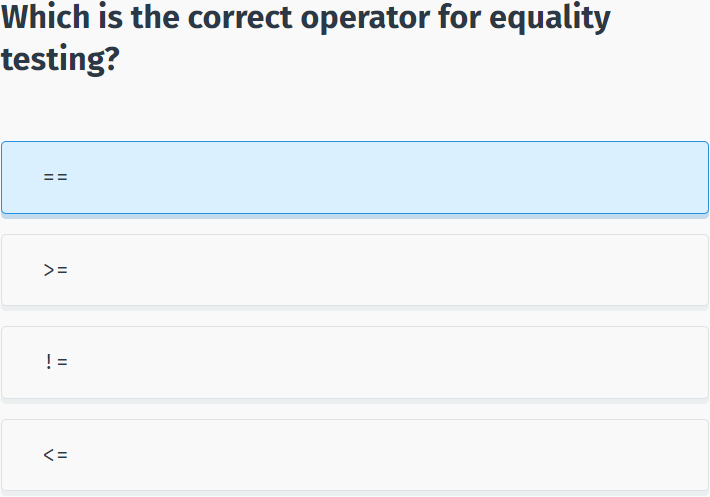
int a=7, b=7;

if (a == b) {

Console.WriteLine("Equal");

//Equal

}



**The else Clause**

An optional **else** clause can be specified to execute a block of code when the condition in the **if** statement evaluates to **false**.

**Syntax:**

if (condition)

{

//statements

}

else

{

//statements

}

**For example:**

int mark = 85;

if (mark < 50)

{

Console.WriteLine("You failed.");

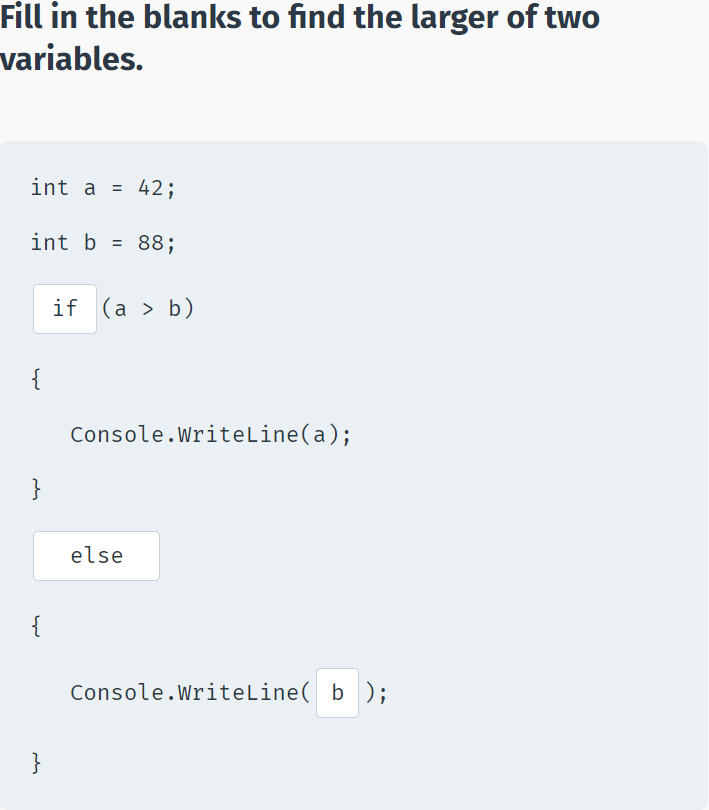
}

else

{

Console.WriteLine("You passed.");

}



**Nested if Statements**

You can also include, or **nest**, if statements within another if statement.

**For example:**

int mark = 100;

if (mark >= 50) {

Console.WriteLine("You passed.");

if (mark == 100) {

Console.WriteLine("Perfect!");

}

else {

Console.WriteLine("You failed.");

}

You can nest an unlimited number of if-else statements.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int age = 17;

if (age > 14) {

if(age > 18) {

Console.WriteLine("Adult");

}

else {

Console.WriteLine("Teenager");

}

}

else {

if (age > 0) {

Console.WriteLine("Child");

}

else {

Console.WriteLine("Something's wrong");

}

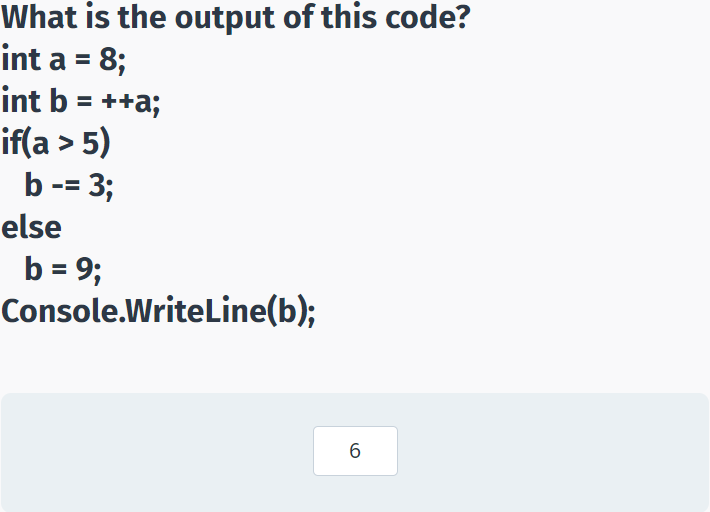
}

}

}

}

Remember that all **else** clauses must have corresponding **if** statements.



**The if-else if Statement**

The **if-else if** statement can be used to decide among three or more actions.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int x = 33;

if (x == 8) {

Console.WriteLine("Value of x is 8");

}

else if (x == 18) {

Console.WriteLine("Value of x is 18");

}

else if (x == 33) {

Console.WriteLine("Value of x is 33");

}

else {

Console.WriteLine("No match");

}

}

}

}

Remember, that an **if** can have zero or more **else if**'s and they must come before the last **else**, which is optional.

Once an **else if** succeeds, none of the remaining **else if**'s or **else** clause will be tested.



## The switch Statement

**switch**

The **switch** statement provides a more elegant way to test a variable for equality against a list of values.

Each value is called a **case**, and the variable being switched on is checked for each switch case.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int num = 3;

switch (num)

{

case 1:

Console.WriteLine("one");

break;

case 2:

Console.WriteLine("two");

break;

case 3:

Console.WriteLine("three");

break;

}

}

}

}

Each **case** represents a value to be checked, followed by a colon, and the statements to get executed if that case is matched.

A **switch** statement can include any number of **cases**. However, no two case labels may contain the same constant value.

The **break**; statement that ends each **case** will be covered shortly.



**The default Case**

In a switch statement, the optional **default** case is executed when none of the previous cases match.

**Example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int age = 88;

switch (age) {

case 16:

Console.WriteLine("Too young");

break;

case 42:

Console.WriteLine("Adult");

break;

case 70:

Console.WriteLine("Senior");

break;

default:

Console.WriteLine("The default case");

break;

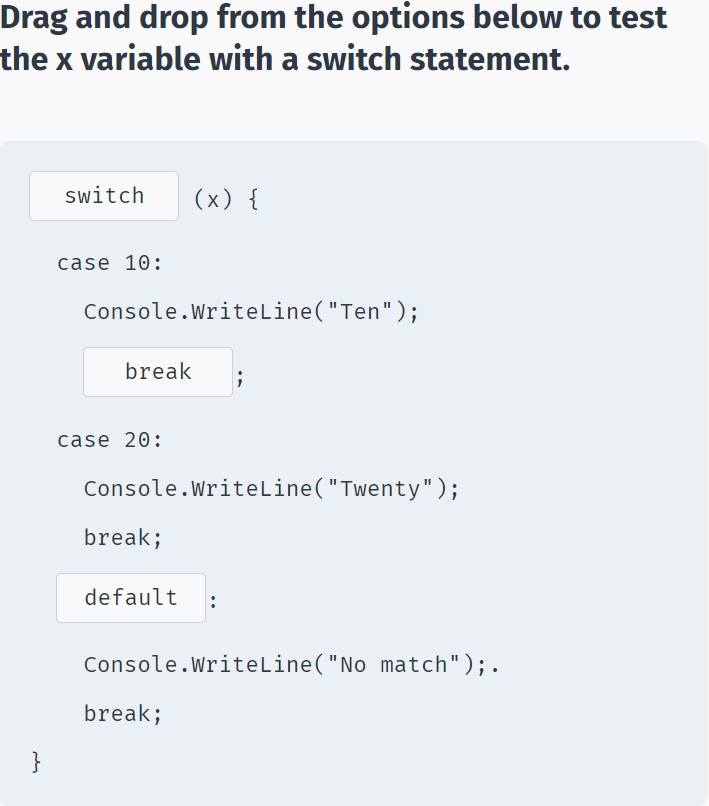
}

}

}

}

The **default** code executes when none of the cases matches the switch expression.



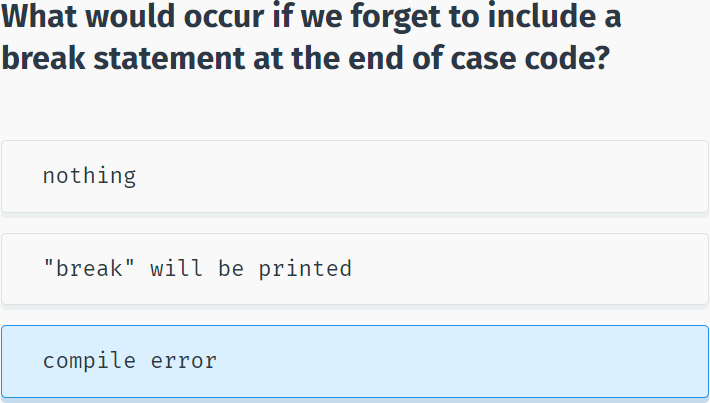
**The break Statement**

The role of the **break** statement is to terminate the **switch** statement.

Without it, execution continues past the matching **case** statements and falls through to the next case statements, even when the case labels don’t match the switch variable.

This behavior is called **fallthrough** and modern C# compilers will not compile such code. All case and default code must end with a **break** statement.

The **break** statement can also be used to break out of a loop. You will learn about loops in the coming lessons.



## The while Loop

**while**

A **while** loop repeatedly executes a block of code as long as a given condition is **true**.

**For example, the following code displays the numbers 1 through 5:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int num = 1;

while(num < 6)

{

Console.WriteLine(num);

num++;

}

}

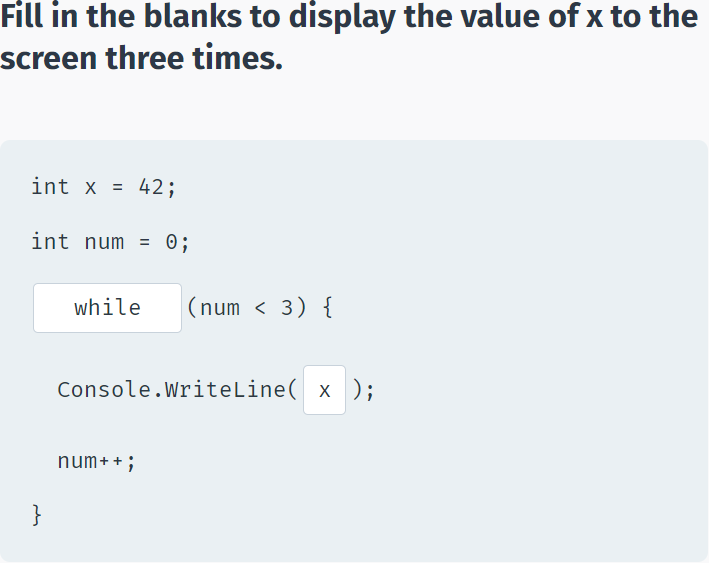
}

}

The example above declares a variable equal to 1 (int num = 1). The **while** loop checks the condition (num < 6) and, if **true**, executes the statements in its body, which increment the value of **num** by one, before checking the loop condition again.

After the 5th iteration, **num** equals 6, the condition evaluates to **false**, and the loop stops running.

The **loop body** is the block of statements within curly braces.



**The while Loop**

The compound arithmetic operators can be used to further control the number of times a loop runs.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int num = 1;

while(num < 6)

{

Console.WriteLine(num);

num+=2;

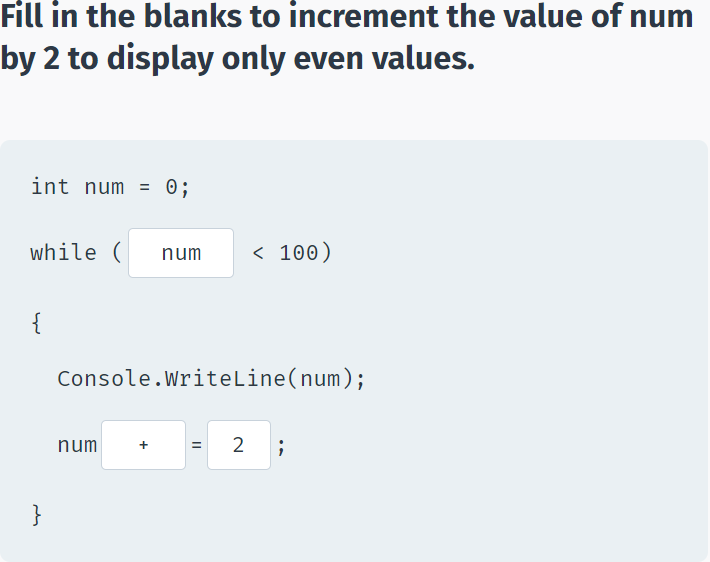
}

}

}

}

Without a statement that eventually evaluates the loop condition to **false**, the loop will continue indefinitely.



We can shorten the previous example, by incrementing the value of **num** right in the condition:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int num = 0;

while(++num < 6)

Console.WriteLine(num);

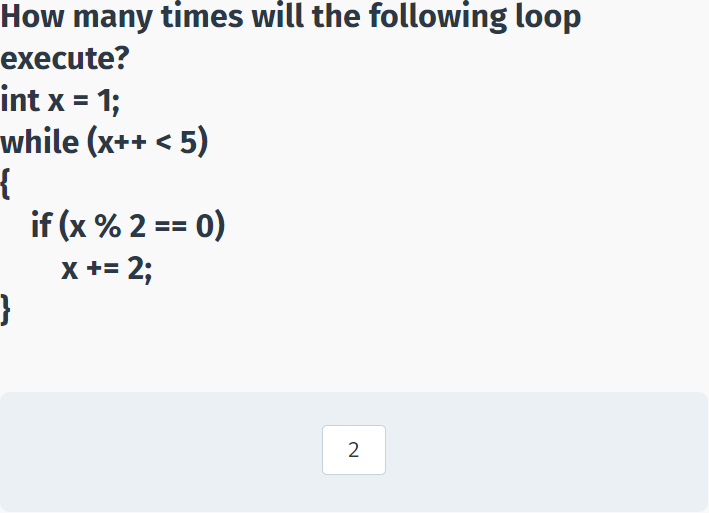
}

}

}

What do you think, is there a difference between **while(num++ < 6)** and **while(++num < 6)**?

Yes! The loop **while(++num < 6)** will execute 5 times, because pre-increment increases the value of x before checking the num < 6 condition, while post-increment will check the condition before increasing the value of num, making **while(num++ < 6)** execute 6 times.



## The for Loop

**The for Loop**

A **for** loop executes a set of statements a specific number of times, and has the syntax:

for ( init; condition; increment ) {

statement(s);

}

A counter is declared once in **init**.

Next, the **condition** evaluates the value of the counter and the body of the loop is executed if the condition is **true**.

After loop execution, the **increment** statement updates the counter, also called the loop control variable.

The condition is again evaluated, and the loop body repeats, only stopping when the condition becomes **false**.

**For example:**

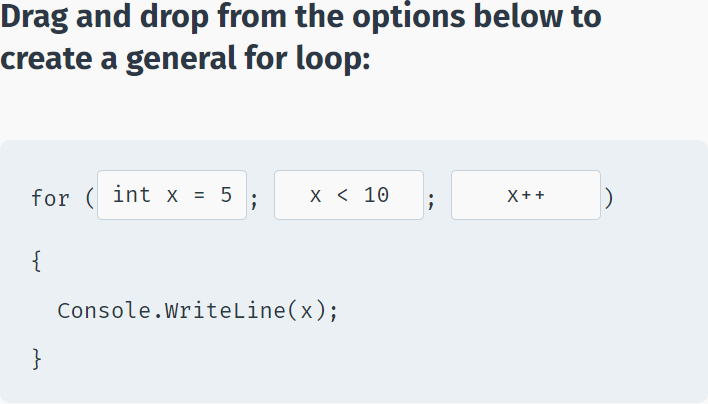
for (int x = 10; x < 15; x++)

{

Console.WriteLine("Value of x: {0}", x);

}

Note the **semicolons** in the syntax.



Compound arithmetic operators can be used to further control loop iterations.

**For example:**

for (int x = 0; x < 10; x+=3)

{

Console.WriteLine(x);

}

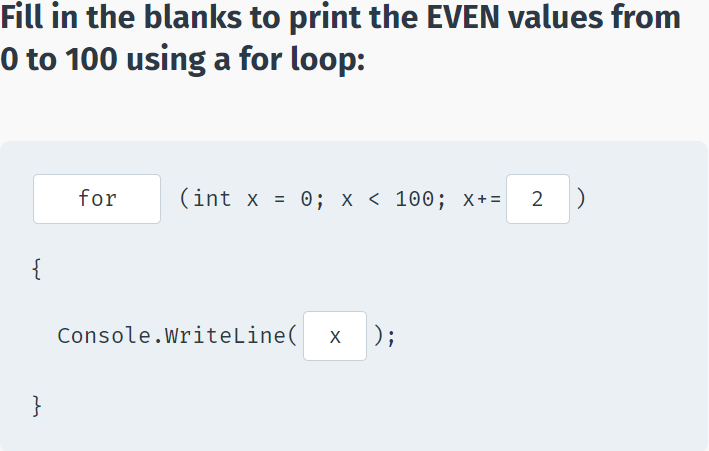
You can also decrement the counter:

for (int x = 10; x > 0; x-=2)

{

Console.WriteLine(x);

}



The **init** and **increment** statements may be left out, if not needed, but remember that the semicolons are mandatory.

**For example, the init can be left out:**

int x = 10;

for ( ; x > 0; x -= 3)

{

Console.WriteLine(x);

}

You can have the increment statement in the for loop body:

int x = 10;

for ( ; x > 0 ; )

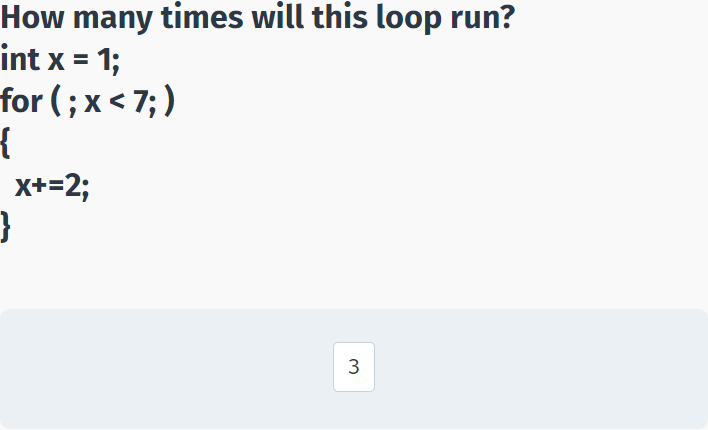
{

Console.WriteLine(x);

x -= 3;

}

**for (; ;) {}** is an infinite loop.



## The do-while Loop

**do-while**

A **do-while** loop is similar to a **while** loop, except that a **do-while** loop is guaranteed to execute at least one time.

**For example:**

int a = 0;

do

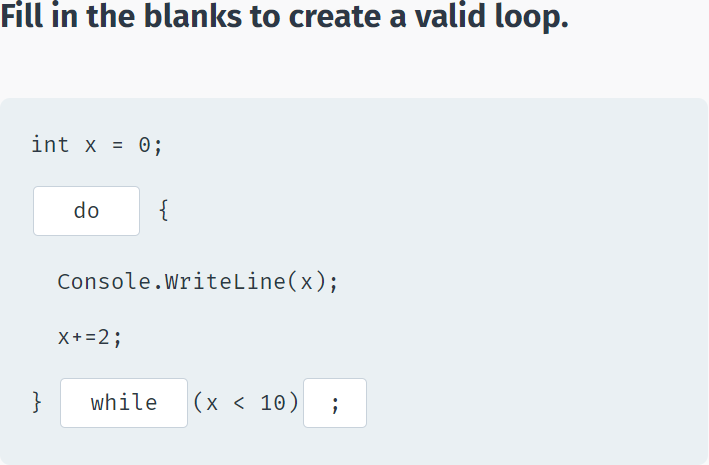
{

Console.WriteLine(a);

a++;

} while (a < 5);

Note the **semicolon** after the while statement.



**do-while vs. while**

If the condition of the **do-while** loop evaluates to **false**, the statements in the **do** will still run once:

int x = 42;

do {

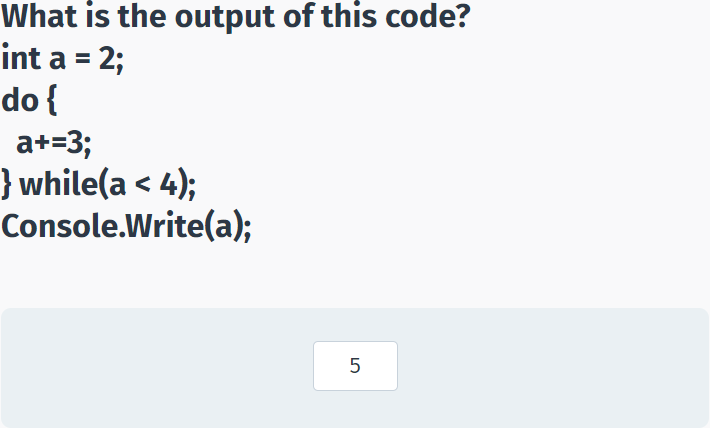
Console.WriteLine(x);

x++;

} while(x < 10);

The **do-while** loop executes the statements at least once, and then tests the condition.

The **while** loop executes the statement only after testing condition.



## break and continue

**break**

We saw the use of **break** in the switch statement.

Another use of **break** is in loops: When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program execution moves on to the next statement following the loop body.

**For example:**

int num = 0;

while (num < 20)

{

if (num == 5)

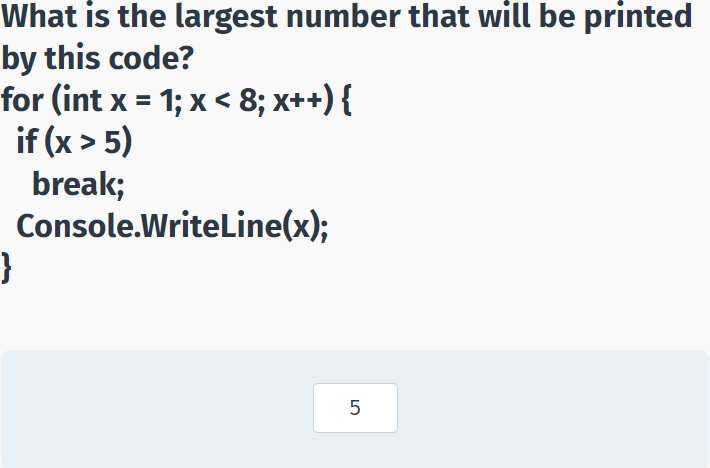
break;

Console.WriteLine(num);

num++;

}

If you are using nested loops (i.e., one loop inside another loop), the **break** statement will stop the execution of the innermost loop and start executing the next line of code after the block.



**continue**

The **continue** statement is similar to the **break** statement, but instead of terminating the loop entirely, it skips the current iteration of the loop and continues with the next iteration.

**For example:**

for (int i = 0; i < 10; i++) {

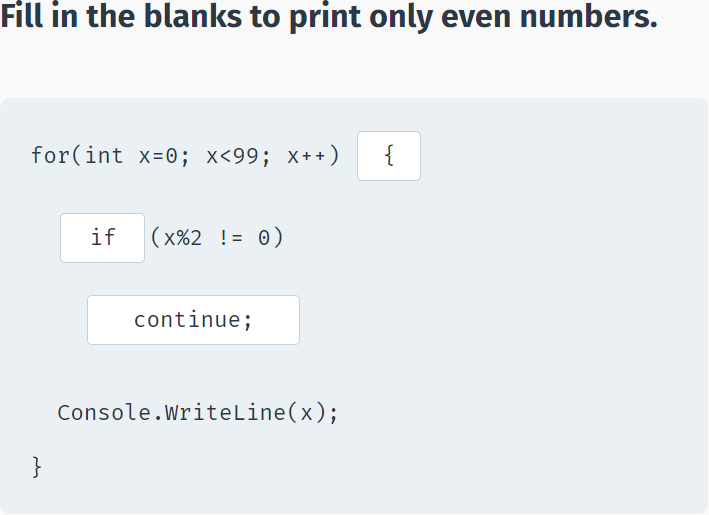
if (i == 5)

continue;

Console.WriteLine(i);

}

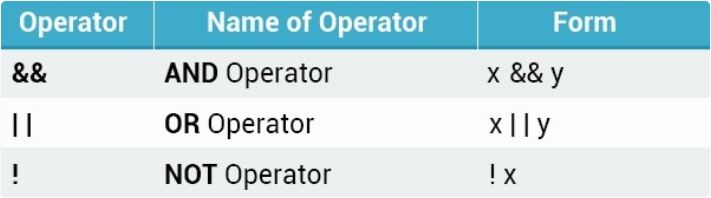
As you can see, number 5 is not printed, as the **continue** statement skips the remaining statements of that iteration of the loop.



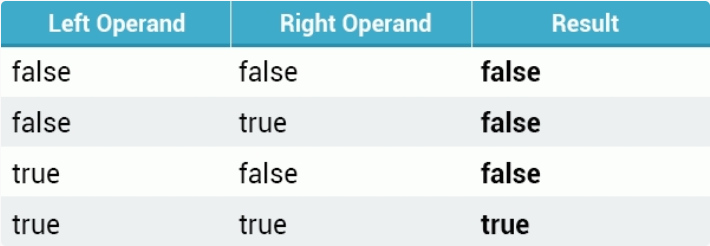
## Logical Operators

**Logical Operators**

Logical operators are used to join multiple expressions and return **true** or **false**.



The **AND** operator (&&) works the following way:



For example, if you wish to display text to the screen only if **age** is greater than 18 AND **money** is greater than 100:

int age = 42;

double money = 540;

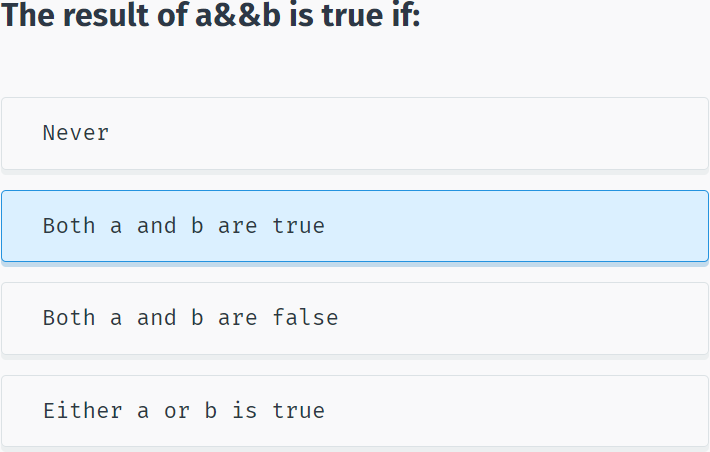
if(age > 18 && money > 100) {

Console.WriteLine("Welcome");

}

The AND operator was used to combine the two expressions.

With the AND operator, both operands must be **true** for the entire expression to be **true**.



**AND**

You can join more than two conditions:

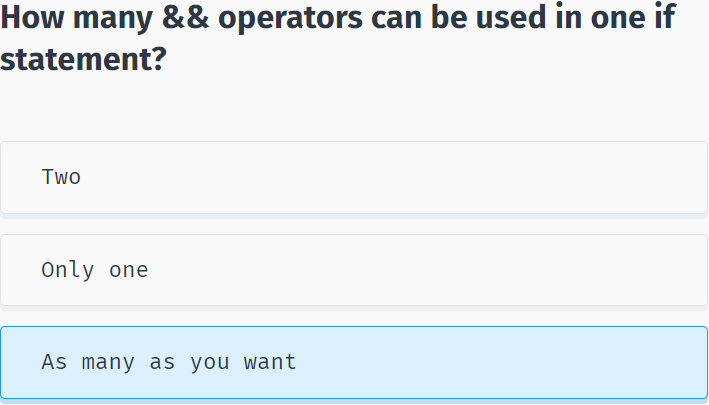
int age = 42;

int grade = 75;

if(age > 16 && age < 80 && grade > 50)

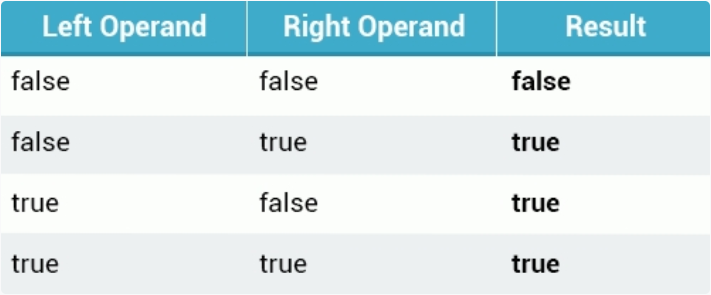
Console.WriteLine("Hey there");

The entire expression evaluates to **true** only if all of the conditions are **true**.



**The OR Operator**

The **OR** operator (||) returns **true** if any one of its operands is **true**.



**For example:**

int age = 18;

int score = 85;

if (age > 20 || score > 50) {

Console.WriteLine("Welcome");

}

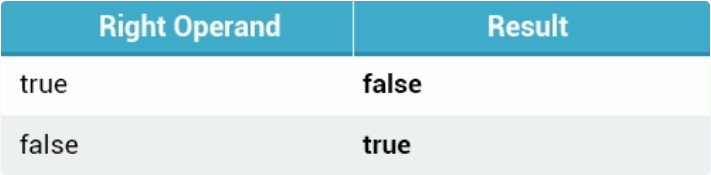
You can join any number of logical **OR** statements you want.

In addition, multiple **OR** and **AND** statements may be joined together.



**Logical NOT**

The logical **NOT** (!) operator works with just a single operand, reversing its logical state. Thus, if a condition is **true**, the NOT operator makes it **false**, and vice versa.

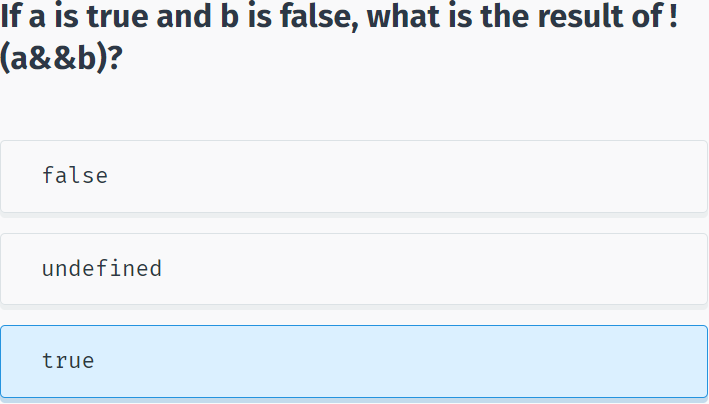


int age = 8;

if ( !(age > 16) ) {

Console.Write("Your age is less than 16");

}



## The Conditional Operator

**The ? : Operator**

**Consider the following example:**

int age = 42;

string msg;

if(age >= 18)

msg = "Welcome";

else

msg = "Sorry";

Console.WriteLine(msg);

The code above checks the value of the **age** variable and displays the corresponding message to the screen.

This can be done in a more elegant and shorter way by using the **?: operator**, which has the following form:

Exp1 ? Exp2 : Exp3;

The ?: operator works the following way: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

So, the example above can be replaced by the following:

int age = 42;

string msg;

msg = (age >= 18) ? "Welcome" : "Sorry";

Console.WriteLine(msg);



## Basic Calculator

**Basic Calculator**

Now let's create a simple project that repeatedly asks the user to enter two values and then displays their sum, until the user enters exit.

We start with a **do-while** loop that asks the user for input and calculates the **sum**:

do {

Console.Write("x = ");

int x = Convert.ToInt32(Console.ReadLine());

Console.Write("y = ");

int y = Convert.ToInt32(Console.ReadLine());

int sum = x+y;

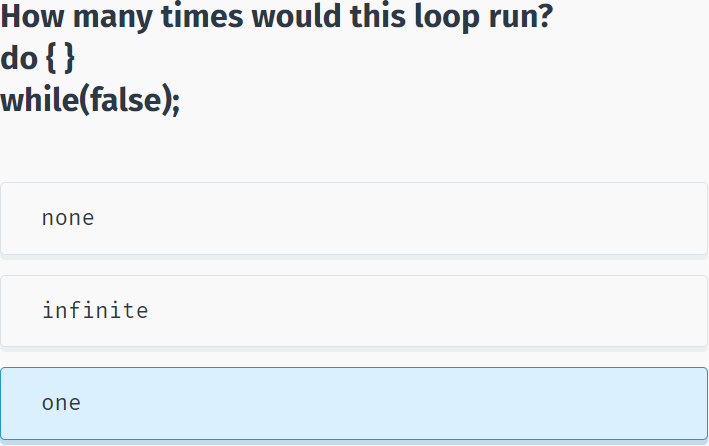
Console.WriteLine("Result: {0}", sum);

}

while(true);

This code will ask for user input infinitely. Now we need to handle the "exit".

If the user enters a non-integer value, the program will crash from a conversion error. We will learn how to handle errors like that in the coming modules.



If the user enters "exit" as the value of x, the program should quit the loop. To do this, we can use a **break** statement:

Console.Write("x = ");

string str = Console.ReadLine();

if (str == "exit")

break;

int x = Convert.ToInt32(str);

Here we compare the input with the value "exit" and break the loop.

**So the whole program looks like:**

do {

Console.Write("x = ");

string str = Console.ReadLine();

if (str == "exit")

break;

int x = Convert.ToInt32(str);

Console.Write("y = ");

int y = Convert.ToInt32(Console.ReadLine());

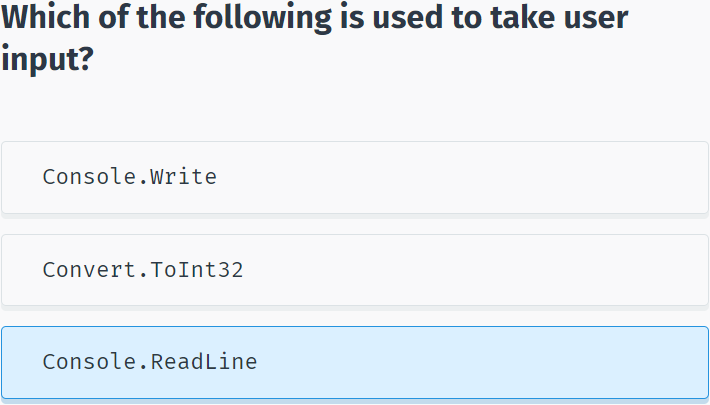
int sum = x + y;

Console.WriteLine("Result: {0}", sum);

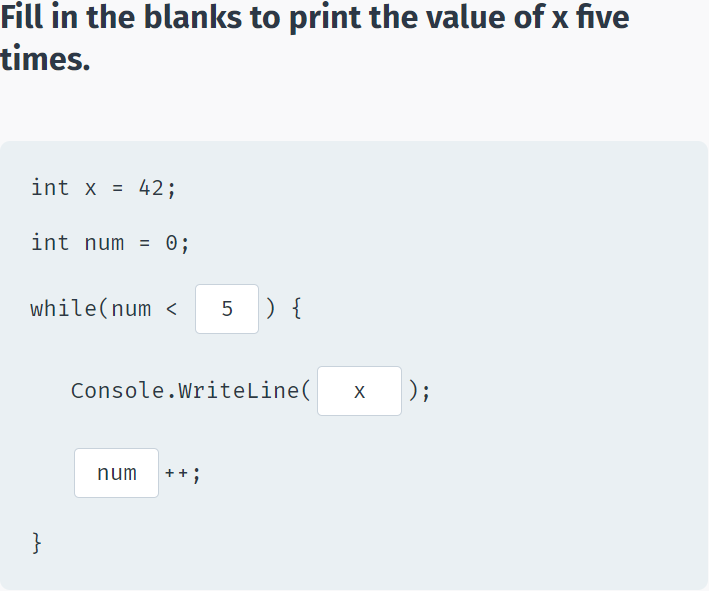
}

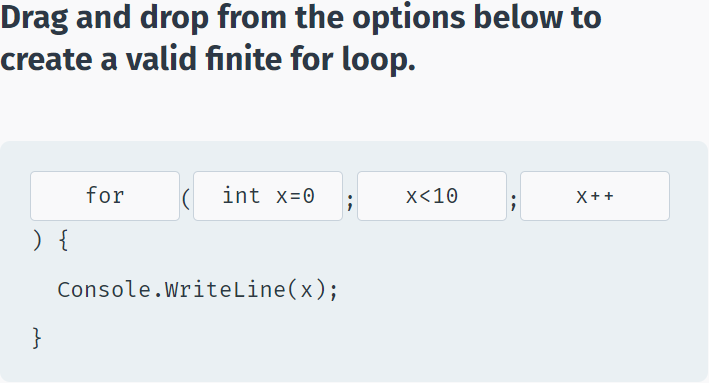
while (true);

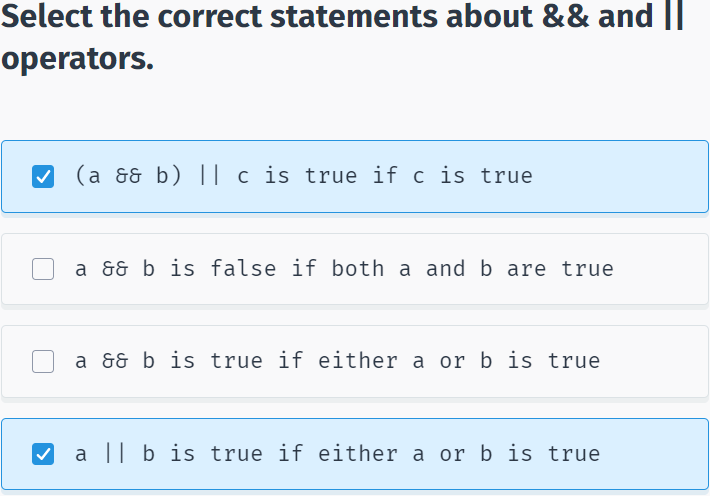
If the user enters "exit" as the value of x, the program should quit the loop.

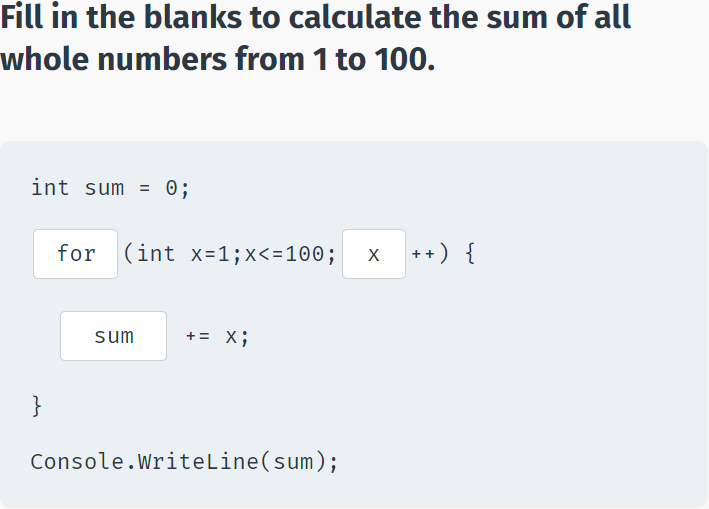


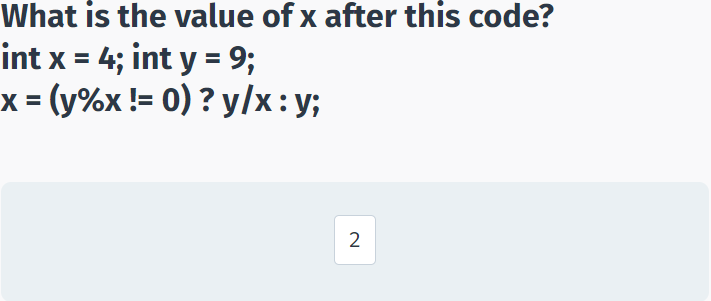
## Module 2 Quiz











# Methods

## Introduction to Methods

**What is a Method?**

A **method** is a group of statements that perform a particular task.

In addition to the C# built-in methods, you may also define your own.

Methods have many advantages, including:

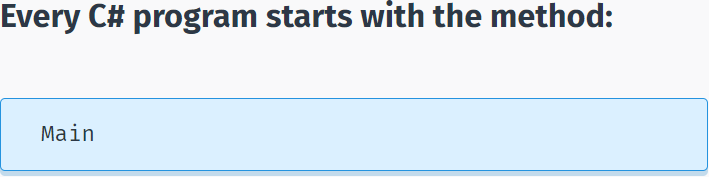
- Reusable code.

- Easy to test.

- Modifications to a method do not affect the calling program.

- One method can accept many different inputs.

Every valid C# program has at least one method, the **Main** method.



**Declaring Methods**

To use a method, you need to **declare** the method and then **call** it.

Each method declaration includes:

- the return type

- the method name

- an optional list of parameters.

<return type> name(type1 par1, type2 par2, … , typeN parN)

{

List of statements

}

For example, the following method has an int parameter and returns the number squared:

int Sqr(int x)

{

int result = x\*x;

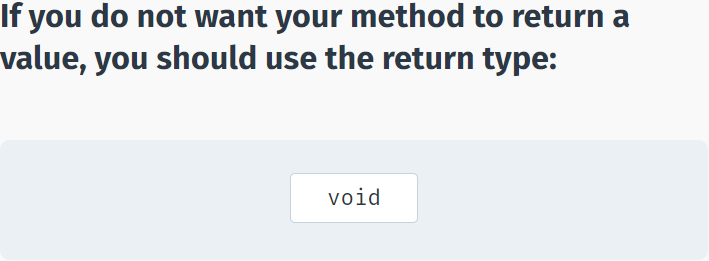
return result;

}

The **return** type of a method is declared before its name. In the example above, the return type is **int**, which indicates that the method returns an integer value. When a method returns a value, it must include a **return** statement. Methods that return a value are often used in assignment statements.

Occasionally, a method performs the desired operations without returning a value. Such methods have a return type **void**. In this case, the method cannot be called as part of an assignment statement.

**void** is a basic data type that defines a valueless state.



**Calling Methods**

Parameters are optional; that is, you can have a method with no parameters.

As an example, let's define a method that does not return a value, and just prints a line of text to the screen.

static void SayHi()

{

Console.WriteLine("Hello");

}

Our method, entitled **SayHi**, returns **void**, and has no parameters.

To execute a method, you simply call the method by using the name and any required arguments in a statement.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void sayHi()

{

Console.WriteLine("Hello");

}

static void Main(string[] args)

{

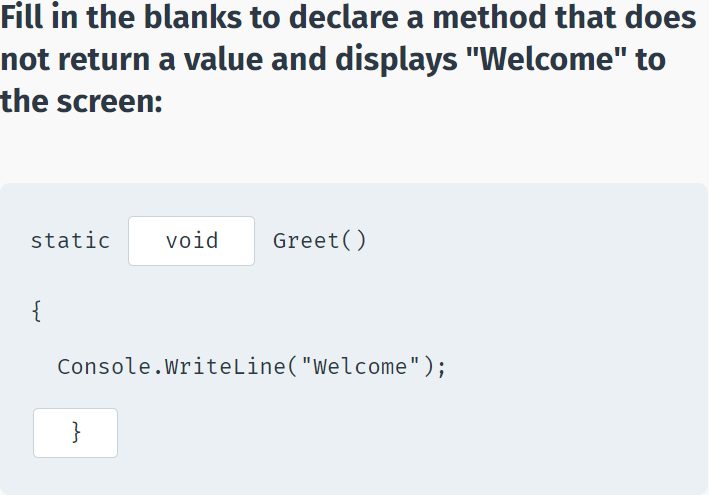
sayHi();

}

}

}

The **static** keyword will be discussed later; it is used to make methods accessible in Main.



You can call the same method multiple times:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void SayHi()

{

Console.WriteLine("Hello");

}

static void Main(string[] args)

{

SayHi();

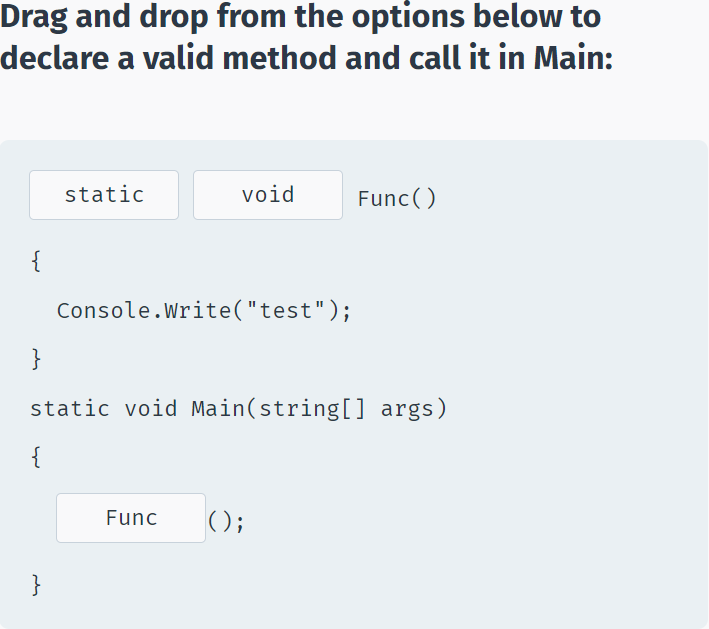
SayHi();

SayHi();

}

}

}



## Method Parameters

**Parameters**

Method declarations can define a list of **parameters** to work with.

Parameters are variables that accept the values passed into the method when called.

**For example:**

void Print(int x)

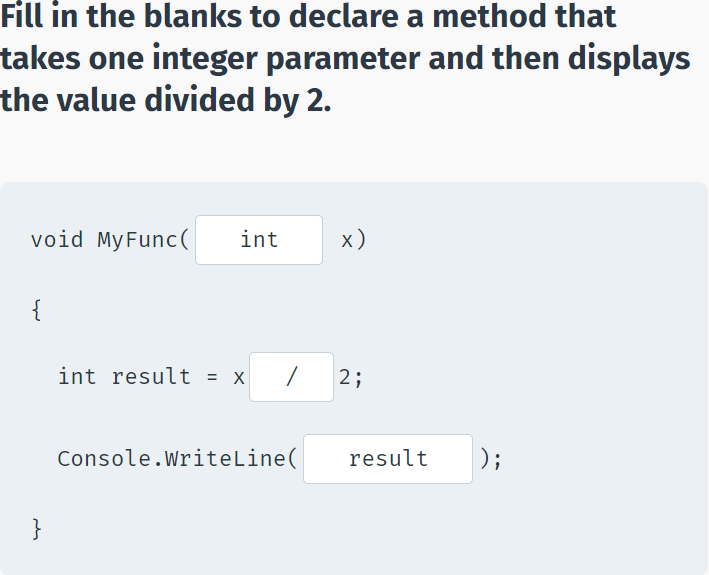
{

Console.WriteLine(x);

}

This defines a method that takes one integer parameter and displays its value.

Parameters behave within the method similarly to other local variables. They are created upon entering the method and are destroyed upon exiting the method.



Now you can call the method in Main and pass in the value for its parameters (also called **arguments**):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Print(int x)

{

Console.WriteLine(x);

}

static void Main(string[] args)

{

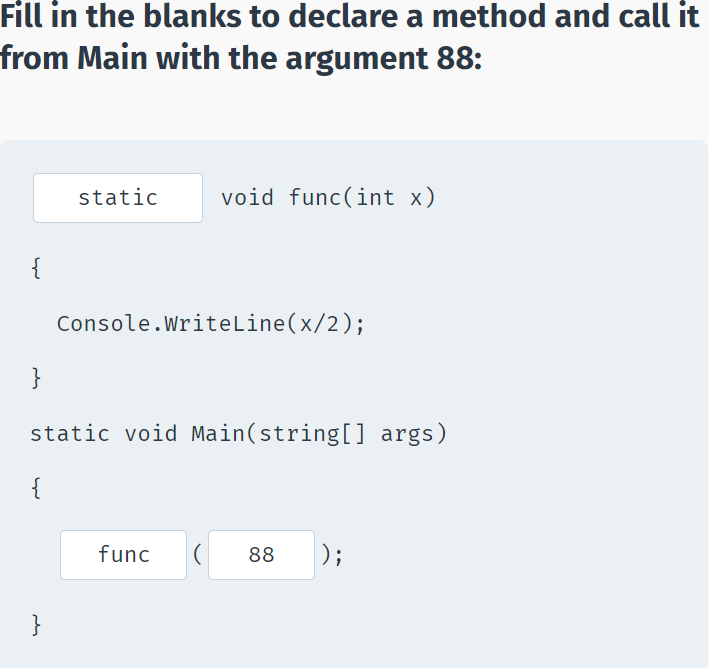
Print(42);

}

}

}

The value 42 is passed to the method as an argument and is assigned to the formal parameter x.



You can pass different arguments to the same method as long as they are of the expected type.

**For example:**

static void Func(int x)

{

Console.WriteLine(x\*2);

}

static void Main(string[] args)

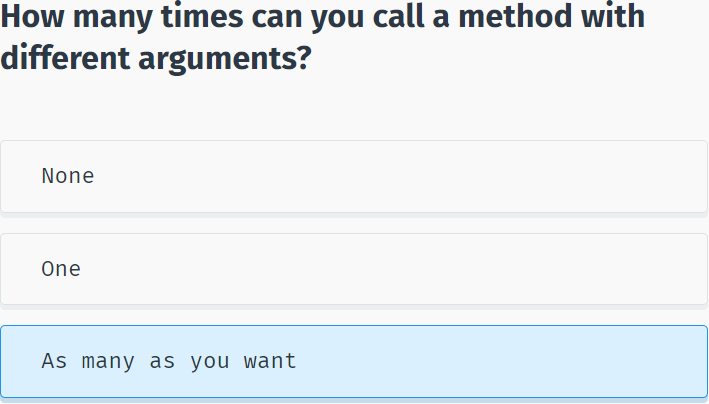
{

Func(5);

Func(12);

Func(42);

}



## Multiple Parameters

**Multiple Parameters**

You can have as many parameters as needed for a method by separating them with **commas** in the definition.

Let's create a simple method that returns the sum of two parameters:

int Sum(int x, int y)

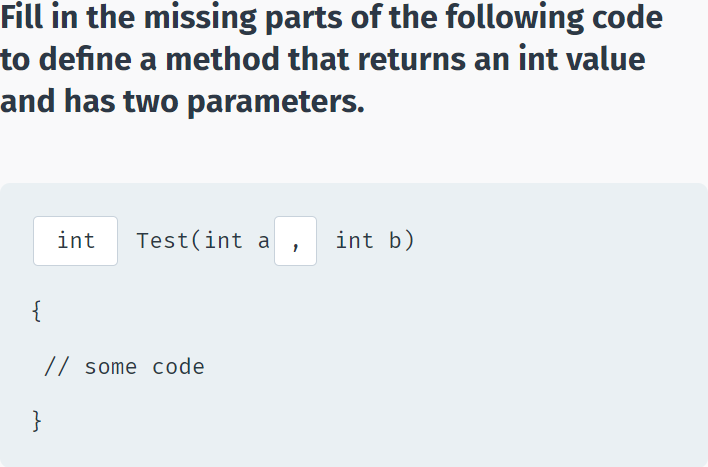
{

return x+y;

}

The **Sum** method takes two integers and returns their sum. This is why the return type of the method is **int**. Data **type** and **name** should be defined for each parameter.

Methods return values using the **return** statement.



A method call with multiple parameters must separate arguments with **commas**.

For example, a call to **Sum** requires two arguments:

static int Sum(int x, int y)

{

return x+y;

}

static void Main(string[] args)

{

Console.WriteLine(Sum(8, 6));

}

In the call above, the return value was displayed to the console window. Alternatively, we can assign the return value to a variable, as in the code below:

static int Sum(int x, int y)

{

return x+y;

}

static void Main(string[] args)

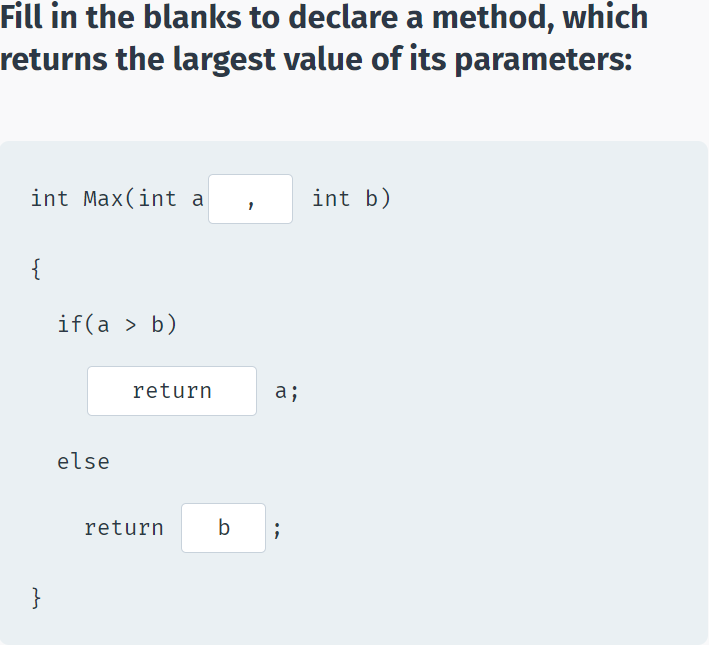
{

int res = Sum(11, 42);

Console.WriteLine(res);

}

You can add as many parameters to a single method as you want. If you have multiple parameters, remember to separate them with **commas**, both when declaring them and when calling the method.



## Optional & Named Arguments

**Optional Arguments**

When defining a method, you can specify a **default value** for optional parameters. Note that optional parameters must be defined after required parameters. If corresponding arguments are missing when the method is called, the method uses the default values.

To do this, assign values to the parameters in the method definition, as shown in this example.

static int Pow(int x, int y=2)

{

int result = 1;

for (int i = 0; i < y; i++)

{

result \*= x;

}

return result;

}

The Pow method assigns a default value of 2 to the y parameter. If we call the method without passing the value for the y parameter, the default value will be used.

static int Pow(int x, int y=2)

{

int result = 1;

for (int i = 0; i < y; i++)

{

result \*= x;

}

return result;

}

static void Main(string[] args)

{

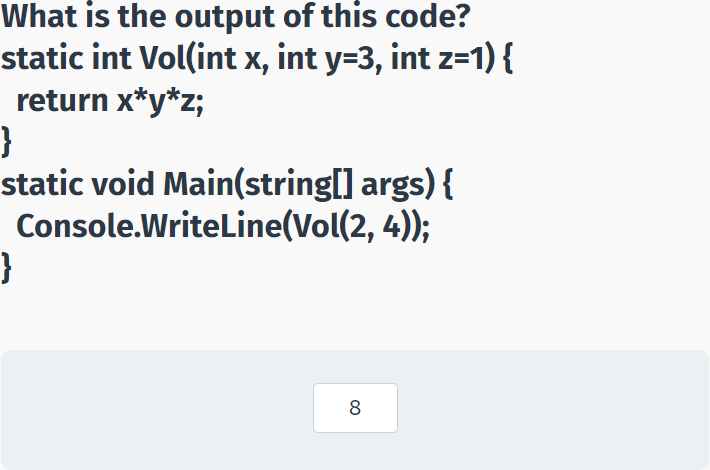
Console.WriteLine(Pow(6));

Console.WriteLine(Pow(3, 4));

}

As you can see, default parameter values can be used for calling the same method in different situations without requiring arguments for every parameter.

Just remember, that you must have the parameters with default values at the **end** of the parameter list when defining the method.



**Named Arguments**

Named arguments free you from the need to remember the order of the parameters in a method call. Each argument can be specified by the matching parameter name.

For example, the following method calculates the area of a rectangle by its height and width:

static int Area(int h, int w)

{

return h \* w;

}

When calling the method, you can use the parameter names to provide the arguments in any order you like:

static int Area(int h, int w)

{

return h \* w;

}

static void Main(string[] args)

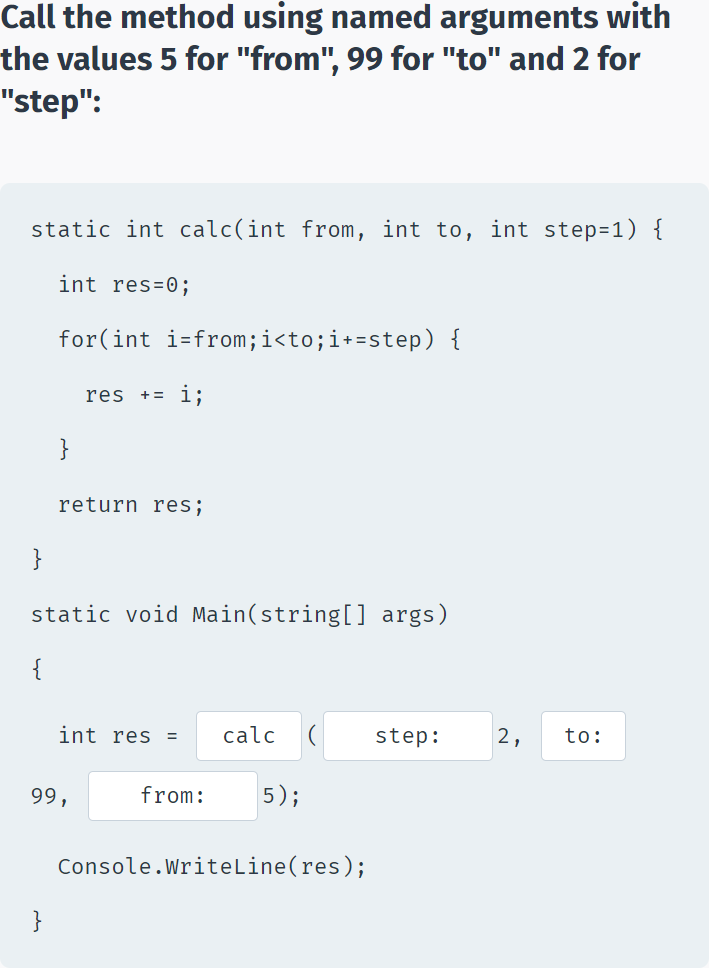
{

int res = Area(w: 5, h: 8);

Console.WriteLine(res);

}

Named arguments use the **name** of the parameter followed by a **colon** and the **value**.



## Passing Arguments

**Passing Arguments**

There are three ways to pass arguments to a method when the method is called: By **value**, By **reference**, and as **Output**.

By **value** copies the argument's value into the method's formal parameter. Here, we can make changes to the parameter within the method without having any effect on the argument.

By default, C# uses call by **value** to pass arguments.

**The following example demonstrates by value:**

static void Sqr(int x)

{

x = x \* x;

}

static void Main(string[] args)

{

int a = 3;

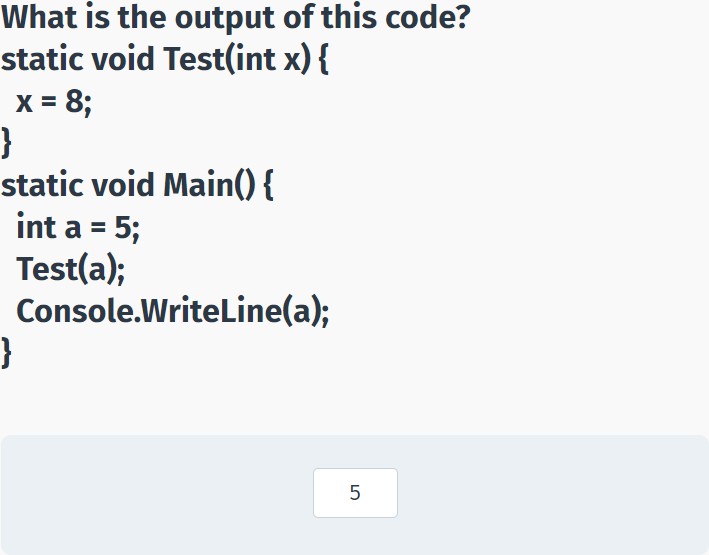
Sqr(a);

Console.WriteLine(a);

}

In this case, **x** is the parameter of the **Sqr** method and **a** is the actual argument passed into the method.

As you can see, the **Sqr** method does not change the original value of the variable, as it is passed by **value**, meaning that it operates on the **value**, not the actual variable.



**Passing by Reference**

Pass by **reference** copies an argument's memory address into the formal parameter. Inside the method, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

To pass the value by reference, the **ref** keyword is used in both the call and the method definition:

static void Sqr(ref int x)

{

x = x \* x;

}

static void Main(string[] args)

{

int a = 3;

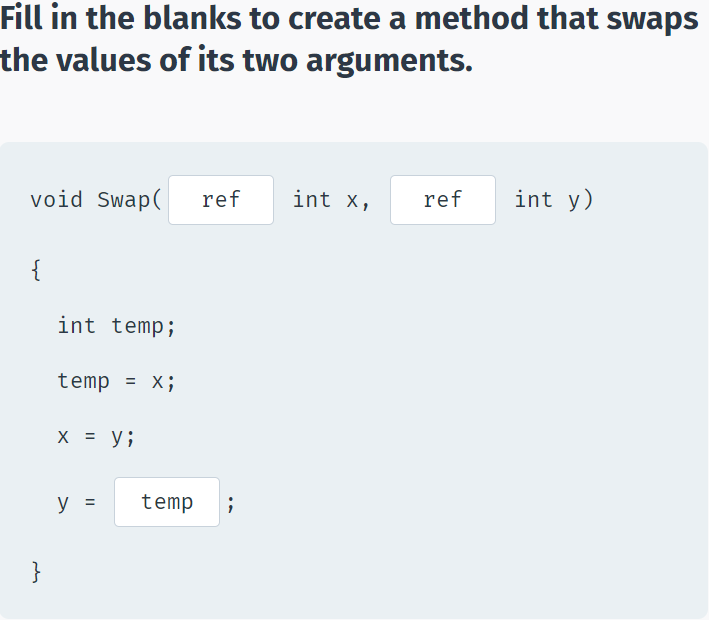
Sqr(ref a);

Console.WriteLine(a);

}

The **ref** keyword passes the memory address to the method parameter, which allows the method to operate on the actual variable.

The **ref** keyword is used both when defining the method and when calling it.



**Passing by Output**

**Output** parameters are similar to reference parameters, except that they transfer data out of the method rather than accept data in. They are defined using the **out** keyword.

The variable supplied for the output parameter need not be initialized since that value will not be used. Output parameters are particularly useful when you need to return multiple values from a method.

**For example:**

static void GetValues(out int x, out int y)

{

x = 5;

y = 42;

}

static void Main(string[] args)

{

int a, b;

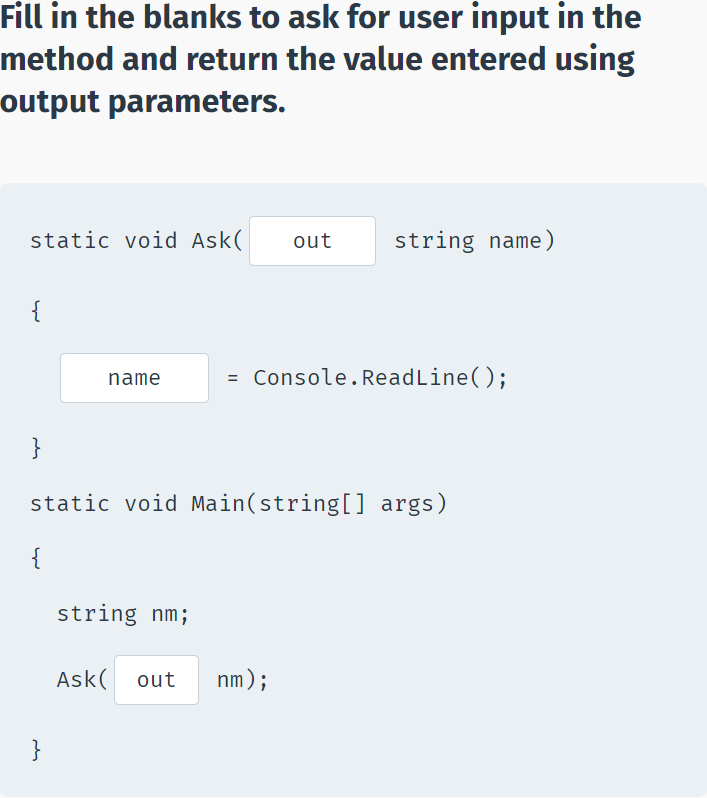
GetValues(out a, out b);

Console.WriteLine(a+" "+b);

}

Unlike the previous reference type example, where the value 3 was referred to the method, which changed its value to 9, output parameters get their value from the method (5 and 42 in the above example).

Similar to the **ref** keyword, the **out** keyword is used both when defining the method and when calling it.



## Method Overloading

**Overloading**

Method **overloading** is when multiple methods have the **same name**, but **different parameters**.

For example, you might have a **Print** method that outputs its parameter to the console window:

void Print(int a)

{

Console.WriteLine("Value: "+a);

}

The + operator is used to concatenate values. In this case, the value of a is joined to the text "Value: ".

This method accepts an integer **argument** only.

Overloading it will make it available for other types, such as **double**

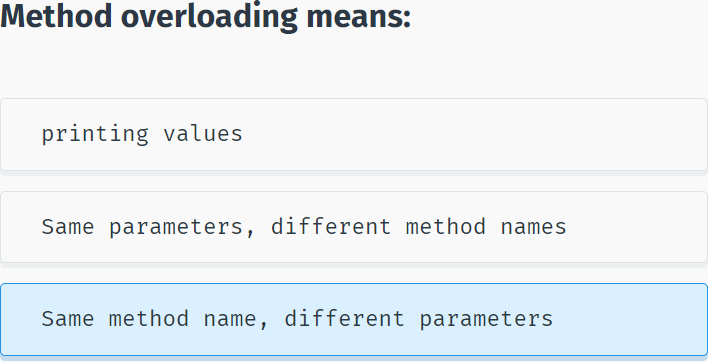
void Print(double a)

{

Console.WriteLine("Value: "+a);

}

Now, the same **Print** method name will work for both integers and doubles.



When overloading methods, the definitions of the methods must differ from each other by the types and/or number of parameters.

When there are overloaded methods, the method called is based on the arguments. An **integer** argument will call the method implementation that accepts an **integer** parameter. A **double** argument will call the implementation that accepts a **double** parameter. Multiple arguments will call the implementation that accepts the same number of arguments.

static void Print(int a) {

Console.WriteLine("Value: " + a);

}

static void Print(double a) {

Console.WriteLine("Value: " + a);

}

static void Print(string label, double a) {

Console.WriteLine(label + a);

}

static void Main(string[] args)

{

Print(11);

Print(4.13);

Print("Average: ", 7.57);

}

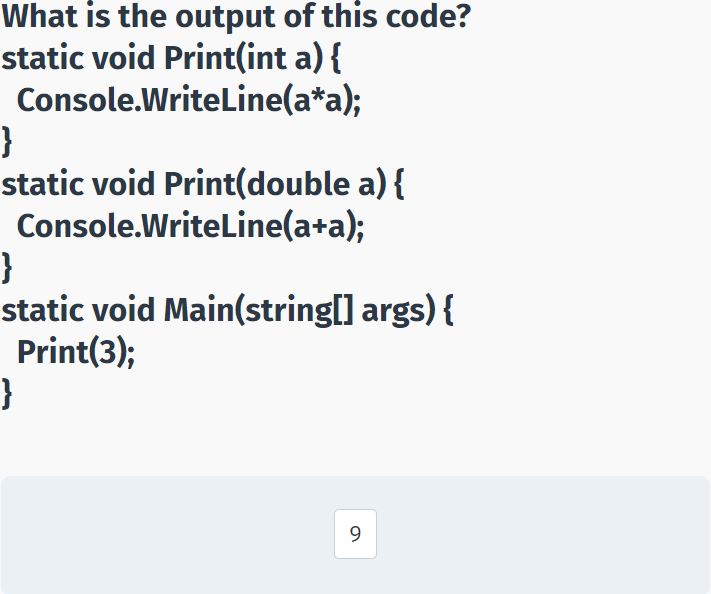
You cannot overload method declarations that differ only by return type.

The following declaration results in an **error**.

**int** PrintName(int a) { }

**float** PrintName(int b) { }

**double** PrintName(int c) { }



## Recursion

**Recursion**

A **recursive** method is a method that calls itself.

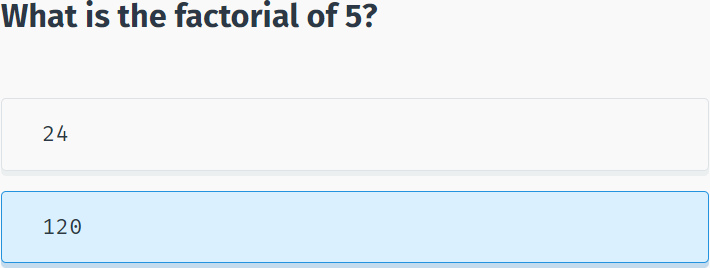
One of the classic tasks that can be solved easily by recursion is calculating the **factorial** of a number.

In mathematics, the term **factorial** refers to the product of all positive integers that are less than or equal to a specific non-negative integer (n). The factorial of n is denoted as **n!**

**For example:**

4! = 4 \* 3 \* 2 \* 1 = 24

A recursive method is a method that calls itself.



As you can see, a factorial can be thought of as repeatedly calculating num \* num-1 until you reach 1.

Based on this solution, let's define our method:

static int Fact(int num) {

if (num == 1) {

return 1;

}

return num \* Fact(num - 1);

}

In the **Fact** recursive method, the **if** statement defines the exit condition, a base case that requires no recursion. In this case, when **num** equals one, the solution is simply to return 1 (the factorial of one is one).

The recursive call is placed after the exit condition and returns **num** multiplied by the factorial of n-1.

For example, if you call the **Fact** method with the argument 4, it will execute as follows:

return 4\*Fact(3), which is 4\*3\*Fact(2), which is 4\*3\*2\*Fact(1), which is 4\*3\*2\*1.

**Now we can call our Fact method from Main:**

static int Fact(int num) {

if (num == 1) {

return 1;

}

return num \* Fact(num - 1);

}

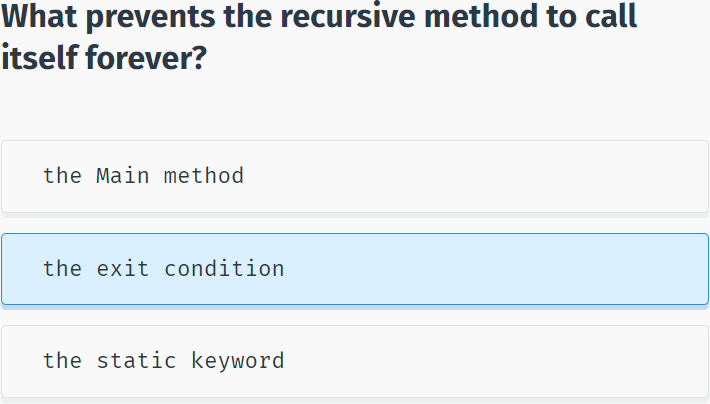
static void Main(string[] args)

{

Console.WriteLine(Fact(6));

}

The factorial method calls itself, and then continues to do so, until the argument equals 1. The exit condition prevents the method from calling itself indefinitely.



## Making a Pyramid

**Making a Pyramid**

Now, let's create a method that will display a pyramid of any height to the console window using star (\*) symbols.

Based on this description, a parameter will be defined to reflect the number of rows for the pyramid.

**So, let's start by declaring the method:**

static void DrawPyramid(int n)

{

//some code will go here

}

**DrawPyramid** does not need to return a value and takes an integer parameter n.

In programming, the step by step logic required for the solution to a problem is called an **algorithm**. The algorithm for MakePyramid is:

1. The first row should contain one star at the top center of the pyramid. The center is calculated based on the number of rows in the pyramid.

2. Each row after the first should contain an odd number of stars (1, 3, 5, etc.), until the number of rows is reached.

Based on the algorithm, the code will use for loops to display spaces and stars for each row:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void DrawPyramid(int n)

{

for (int i=1; i<=n; i++)

{

for (int j=i; j<=n; j++)

{

Console.Write(" ");

}

for (int k=1; k<=2\*i-1; k++)

{

Console.Write("\*"+" ");

}

Console.WriteLine();

}

}

static void Main(string[] args)

{

DrawPyramid(5);

}

}

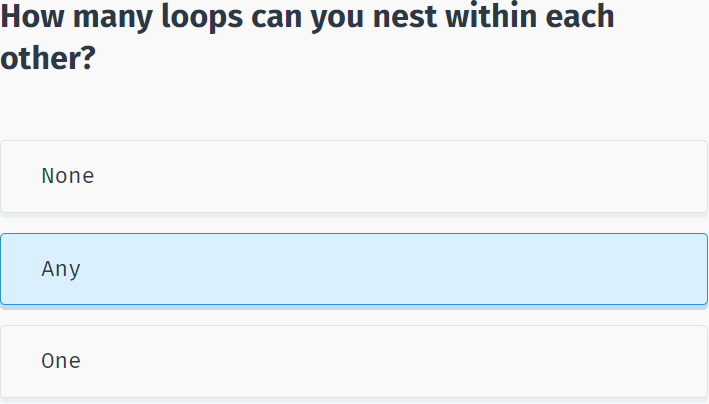
}

The first **for** loop that iterates through each row of the pyramid contains two **for** loops.

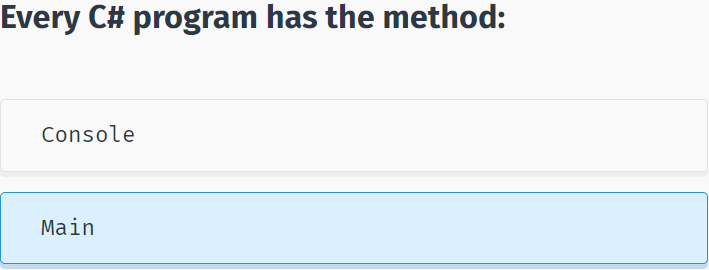
The first inner loop displays the spaces needed before the first star symbol. The second inner loop displays the required number of stars for each row, which is calculated based on the formula (2\*i-1) where i is the current row.

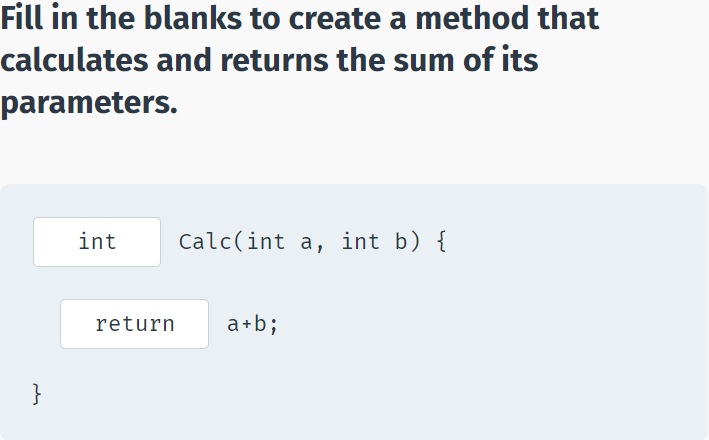
The final **Console.WriteLine();** statement moves the cursor to the next row.

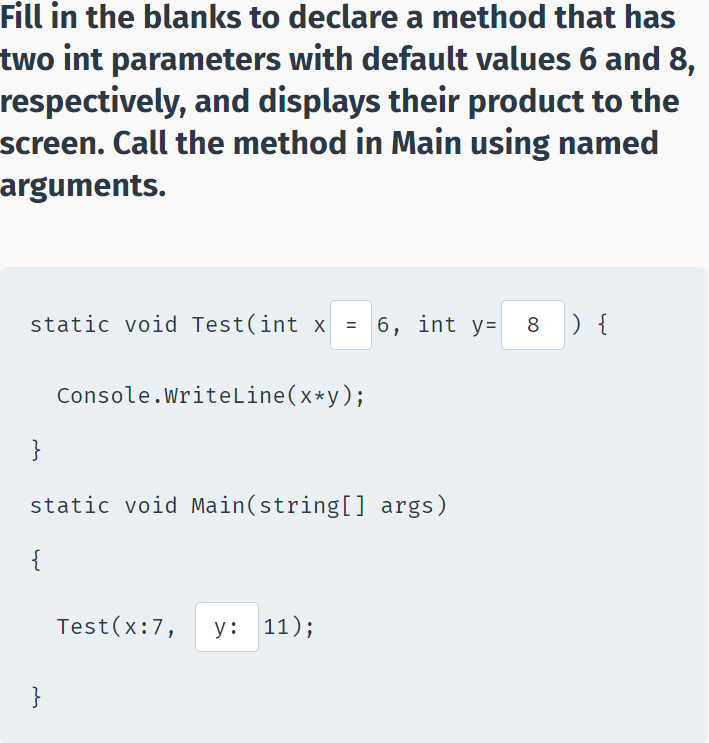
Now, if we call the **DrawPyramid** method, it will display a pyramid having the number of rows we pass to the method.

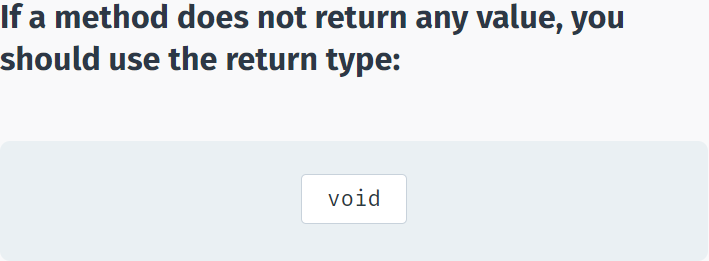


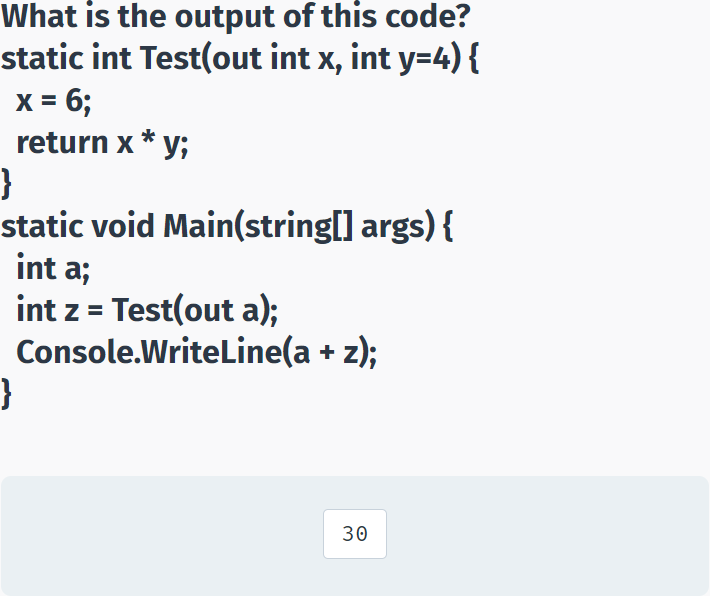
## Module 3 Quiz











# Classes & Objects

## Classes & Objects

**Classes**

As we have seen in the previous modules, built-in data types are used to store a single value in a declared variable. For example, **int x** stores an integer value in a variable named **x**.

In object-oriented programming, a **class** is a data type that defines a set of variables and methods for a declared **object**.

For example, if you were to create a program that manages bank accounts, a **BankAccount** class could be used to declare an object that would have all the properties and methods needed for managing an individual bank account, such as a **balance** variable and **Deposit** and **Withdrawal** methods.

A class is like a **blueprint**. It defines the data and behavior for a type. A class definition starts with the keyword **class** followed by the class name. The class body contains the data and actions enclosed by curly braces.

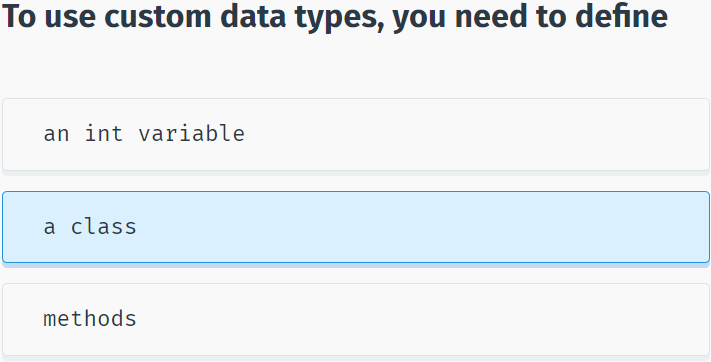
class BankAccount

{

//variables, methods, etc.

}

The class defines a data type for objects, but it is not an object itself. An **object** is a concrete entity based on a class, and is sometimes referred to as an **instance of a class**.



**Objects**

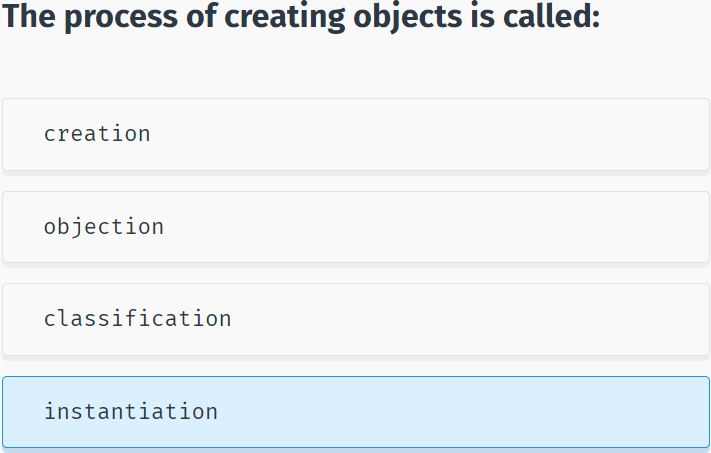
Just as a built-in data type is used to declare multiple variables, a class can be used to declare multiple **objects**. As an analogy, in preparation for a new building, the architect designs a blueprint, which is used as a basis for actually building the structure. That same blueprint can be used to create multiple buildings.

Programming works in the same fashion. We define (design) a class that is the blueprint for creating objects.

In programming, the term **type** is used to refer to a class **name**: We're creating an object of a particular **type**.

Once we've written the class, we can create objects based on that class. Creating an object is called **instantiation**.

An object is called an instance of a class.



Each object has its own characteristics. Just as a person is distinguished by name, age, and gender, an object has its own set of values that differentiate it from another object of the same type.

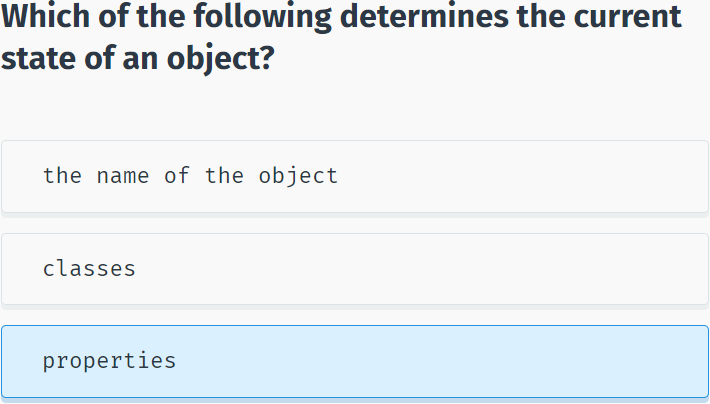
The characteristics of an object are called **properties**.

Values of these properties describe the current state of an object. For example, a Person (an object of the class Person) can be 30 years old, male, and named Antonio.

Objects aren't always representative of just physical characteristics.

For example, a programming object can represent a date, a time, and a bank account. A bank account is not tangible; you can't see it or touch it, but it's still a well-defined object because it has its own properties.

Let's move on and see how to create your own custom classes and objects!



## Value & Reference Types

**Value Types**

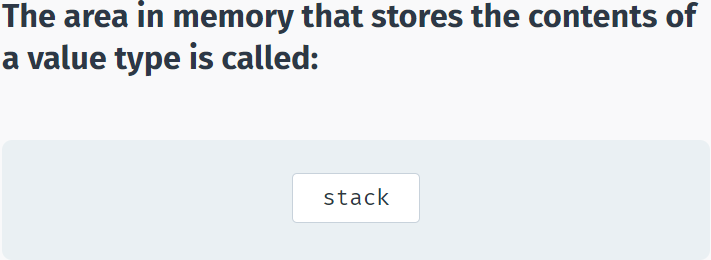
C# has two ways of storing data: by **reference** and by **value**.

The built-in data types, such as int and double, are used to declare variables that are **value types**. Their value is stored in memory in a location called the **stack**.

For example, the declaration and assignment statement **int x = 10**; can be thought of as:



The value of the variable x is now stored on the **stack**.



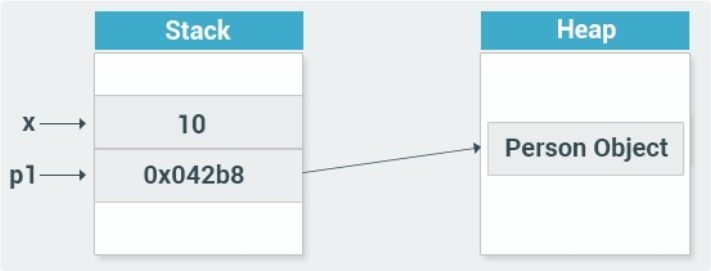
**Reference Types**

**Reference** types are used for storing objects. For example, when you create an object of a class, it is stored as a reference type.

Reference types are stored in a part of the memory called the **heap**.

When you instantiate an object, the data for that object is stored on the heap, while its heap memory address is stored on the stack.

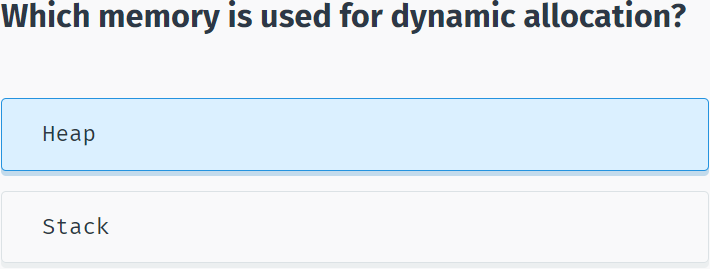
That is why it is called a reference type - it contains a reference (the memory address) to the actual object on the heap.



As you can see, the **p1** object of type Person on the stack stores the memory address of the heap where the actual object is stored.

**Stack** is used for static memory allocation, which includes all your value types, like x.

**Heap** is used for dynamic memory allocation, which includes custom objects, that might need additional memory during the runtime of your program.



## Class Example

**Example of a Class**

Let’s create a **Person** class:

class Person

{

int age;

string name;

public void SayHi()

{

Console.WriteLine("Hi");

}

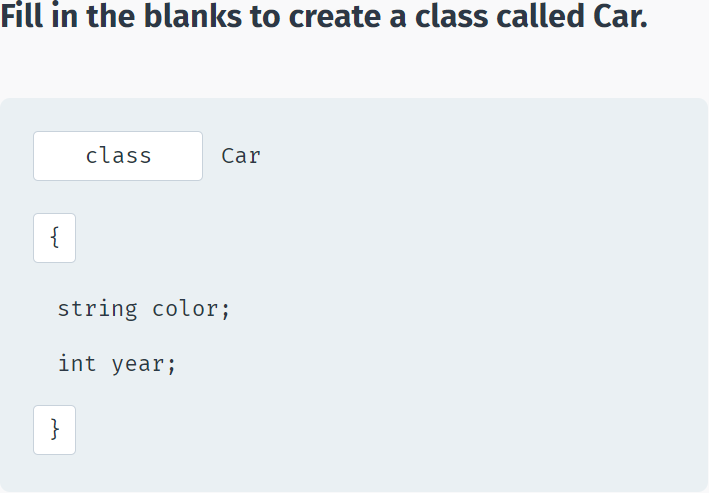
}

The code above declares a class named **Person**, which has **age** and **name** fields as well as a **SayHi** method that displays a greeting to the screen.

You can include an access modifier for fields and methods (also called **members**) of a class. **Access modifiers** are keywords used to specify the accessibility of a member.

A member that has been defined **public** can be accessed from outside the class, as long as it's anywhere within the scope of the class object. That is why our **SayHi** method is declared **public**, as we are going to call it from outside of the class.

You can also designate class members as **private** or **protected**. This will be discussed in greater detail later in the course. If no access modifier is defined, the member is **private** by default.



Now that we have our Person class defined, we can instantiate an object of that type in Main.

The **new** operator instantiates an object and returns a reference to its location:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Person {

int age;

string name;

public void SayHi() {

Console.WriteLine("Hi");

}

}

static void Main(string[] args)

{

Person p1 = new Person();

p1.SayHi();

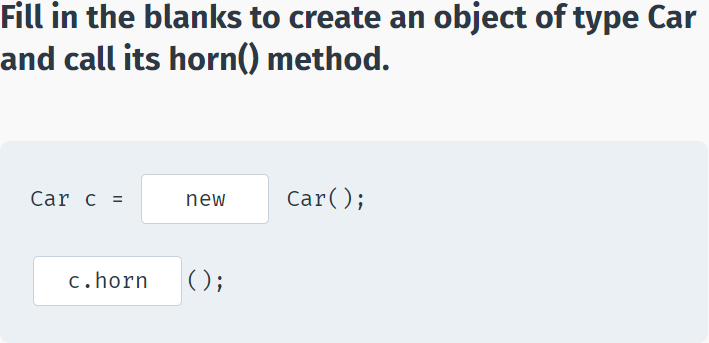
}

}

}

The code above declares a Person object named **p1** and then calls its public **SayHi()** method.

Notice the **dot operator (.)** that is used to access and call the method of the object.



You can access all public members of a class using the dot operator.

Besides calling a method, you can use the dot operator to make an assignment when valid.

**For example:**

class Dog

{

public string name;

public int age;

}

static void Main(string[] args)

{

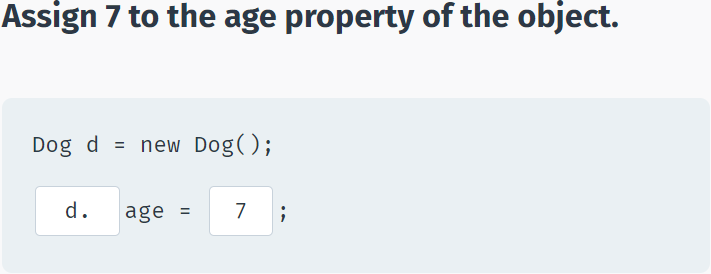
Dog bob = new Dog();

bob.name = "Bobby";

bob.age = 3;

Console.WriteLine(bob.age);

}



## Encapsulation

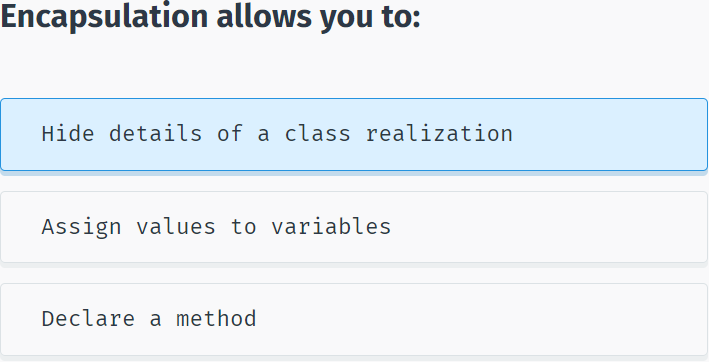
**Encapsulation**

Part of the meaning of the word **encapsulation** is the idea of "surrounding" an entity, not just to keep what's inside together, but also to protect it.

In programming, encapsulation means more than simply combining members together within a class; it also means restricting access to the inner workings of that class.

Encapsulation is implemented by using **access modifiers**. An access modifier defines the scope and visibility of a class member.

Encapsulation is also called **information hiding**.

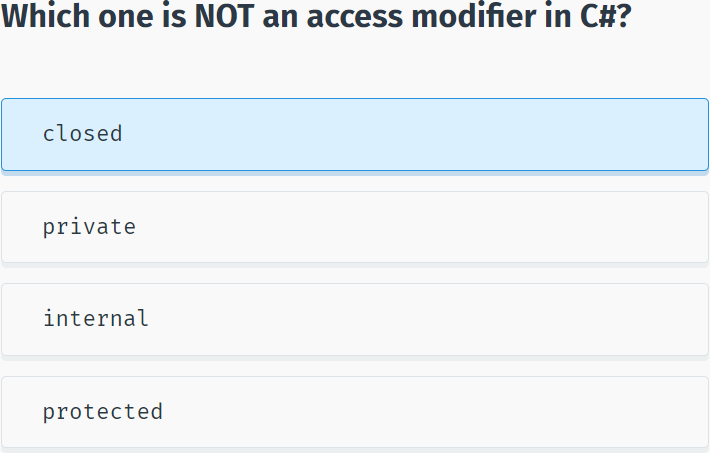


C# supports the following access modifiers: **public, private, protected, internal, protected internal**.

As seen in the previous examples, the **public** access modifier makes the member accessible from the outside of the class.

The **private** access modifier makes members accessible only from within the class and hides them from the outside.

**protected** will be discussed later in the course.



To show encapsulation in action, let’s consider the following example:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class BankAccount {

private double balance=0;

public void Deposit(double n) {

balance += n;

}

public void Withdraw(double n) {

balance -= n;

}

public double GetBalance() {

return balance;

}

}

class Program

{

static void Main(string[] args)

{

BankAccount b = new BankAccount();

b.Deposit(199);

b.Withdraw(42);

Console.WriteLine(b.GetBalance());

}

}

}

We used encapsulation to hide the **balance** member from the outside code. Then we provided restricted access to it using public methods. The class data can be read through the **GetBalance** method and modified only through the **Deposit** and **Withdraw** methods.

You cannot directly change the **balance** variable. You can only view its value using the public method. This helps maintain data integrity.

We could add different verification and checking mechanisms to the methods to provide additional security and prevent errors.

In summary, the benefits of encapsulation are:

- Control the way data is accessed or modified.

- Code is more flexible and easy to change with new requirements.

- Change one part of code without affecting other parts of code.



## Constructors

**Constructors**

A class constructor is a special member method of a class that is executed whenever a new object of that class is created.

A constructor has exactly the same name as its class, is public, and does not have any return type.

**For example:**

class Person

{

private int age;

public Person()

{

Console.WriteLine("Hi there");

}

}

Now, upon the creation of an object of type Person, the constructor is automatically called.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Person

{

private int age;

public Person()

{

Console.WriteLine("Hi there");

}

}

static void Main(string[] args)

{

Person p = new Person();

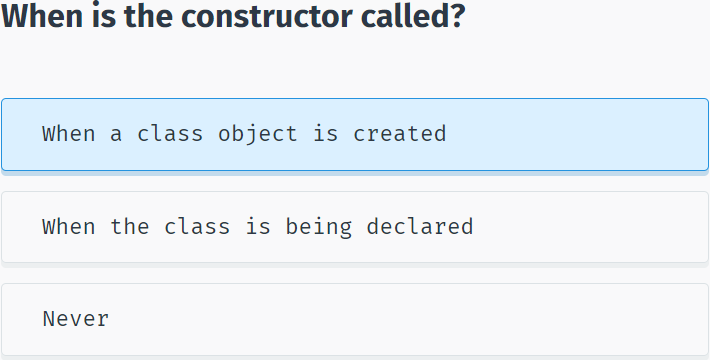
}

}

}

This can be useful in a number of situations. For example, when creating an object of type BankAccount, you could send an email notification to the owner.

The same functionality could be achieved using a separate public method. The advantage of the constructor is that it is called automatically.



Constructors can be very useful for setting initial values for certain member variables.

A default constructor has no parameters. However, when needed, parameters can be added to a constructor. This makes it possible to assign an initial value to an object when it's created, as shown in the following example:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Person

{

private int age;

private string name;

public Person(string nm)

{

name = nm;

}

public string getName()

{

return name;

}

}

static void Main(string[] args)

{

Person p = new Person("David");

Console.WriteLine(p.getName());

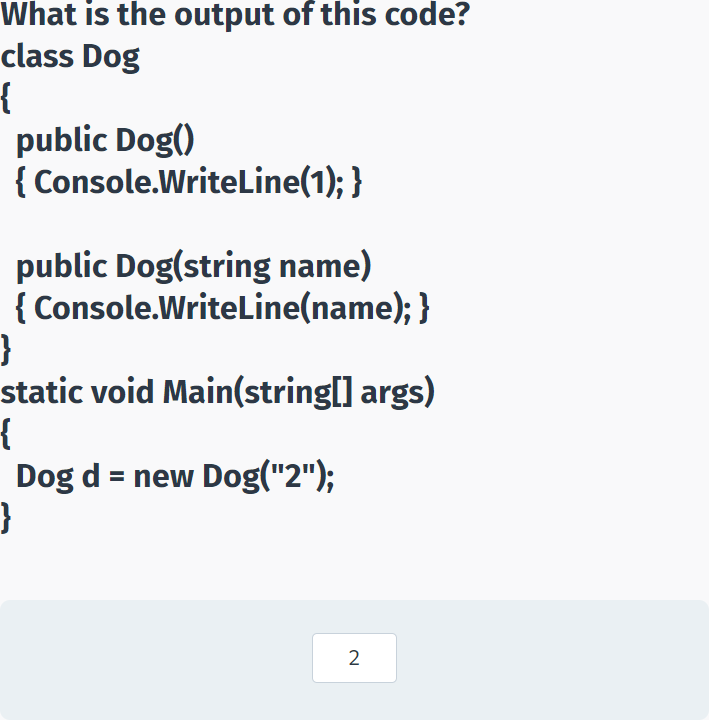
}

}

}

Now, when the object is created, we can pass a parameter that will be assigned to the **name** variable.

Constructors can be **overloaded** like any method by using different numbers of parameters.



## Properties

**Properties**

As we have seen in the previous lessons, it is a good practice to encapsulate members of a class and provide access to them only through public methods.

A **property** is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they actually include special methods called accessors.

The **accessor** of a property contains the executable statements that help in getting (reading or computing) or setting (writing) a corresponding field. Accessor declarations can include a **get** accessor, a **set** accessor, or both.

**For example:**

class Person

{

private string name; //field

public string Name //property

{

get { return name; }

set { name = value; }

}

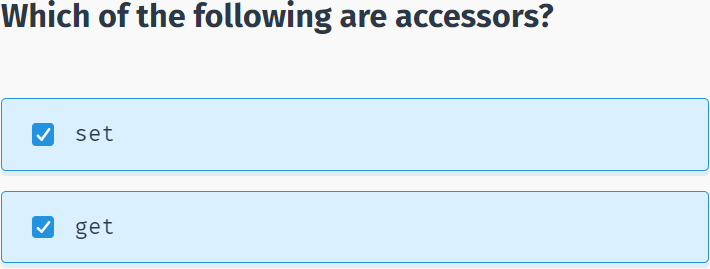
}

The Person class has a **Name** property that has both the **set** and the **get** accessors.

The set accessor is used to assign a value to the name variable; get is used to return its value.

**value** is a special keyword, which represents the value we assign to a property using the **set** accessor.

The name of the property can be anything you want, but coding conventions dictate properties have the same name as the private field with a capital letter.



Once the property is defined, we can use it to assign and read the private member:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Person

{

private string name;

public string Name

{

get { return name; }

set { name = value; }

}

}

static void Main(string[] args)

{

Person p = new Person();

p.Name = "Bob";

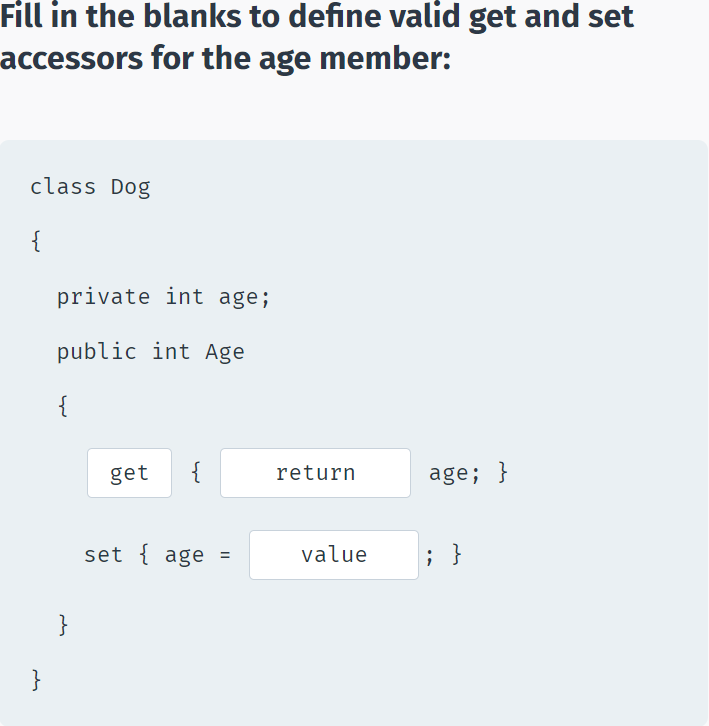
Console.WriteLine(p.Name);

}

}

}

The property is accessed by its name, just like any other public member of the class.



Any accessor of a property can be omitted.

For example, the following code creates a property that is read-only:

class Person

{

private string name;

public string Name

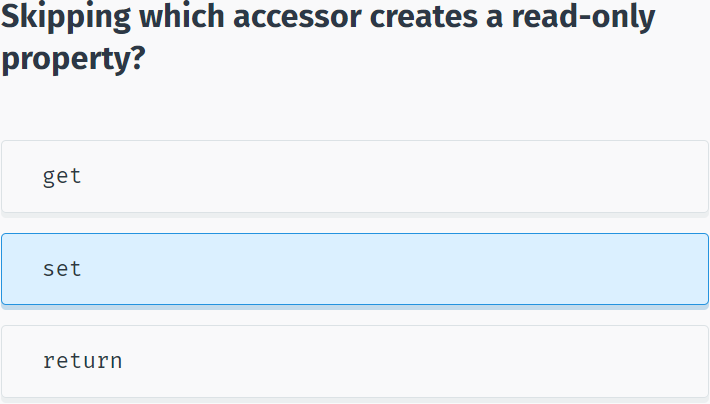
{

get { return name; }

}

}

A property can also be **private**, so it can be called only from within the class.



So, why use properties? Why not just declare the member variable public and access it directly?

With properties you have the option to control the logic of accessing the variable.

For example, you can check if the value of **age** is greater than 0, before assigning it to the variable:

class Person

{

private int age=0;

public int Age

{

get { return age; }

set {

if (value > 0)

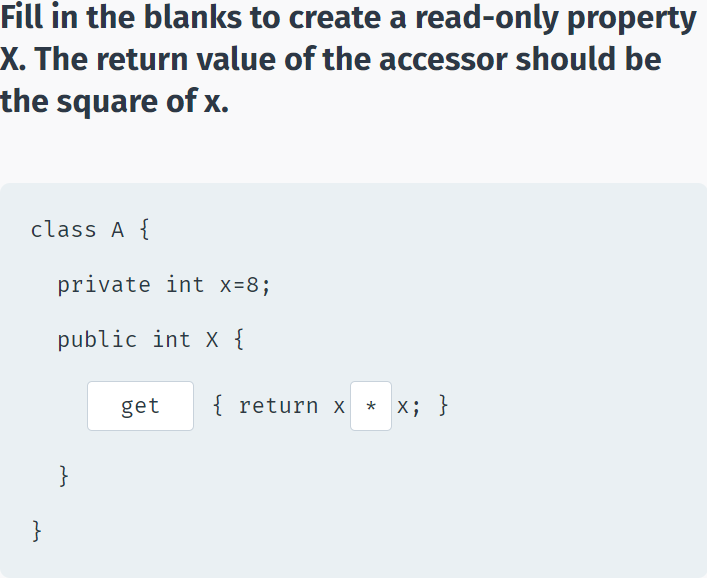
age = value;

}

}

}

You can have any custom logic with **get** and **set** accessors.



**Auto-Implemented Properties**

When you do not need any custom logic, C# provides a fast and effective mechanism for declaring private members through their properties.

For example, to create a private member that can only be accessed through the **Name** property's **get** and **set** accessors, use the following syntax:

public string Name { get; set; }

As you can see, you do not need to declare the private field name separately - it is created by the property automatically. **Name** is called an **auto-implemented property**. Also called auto-properties, they allow for easy and short declaration of private members.

We can rewrite the code from our previous example using an auto-property:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Person

{

public string Name { get; set; }

}

static void Main(string[] args)

{

Person p = new Person();

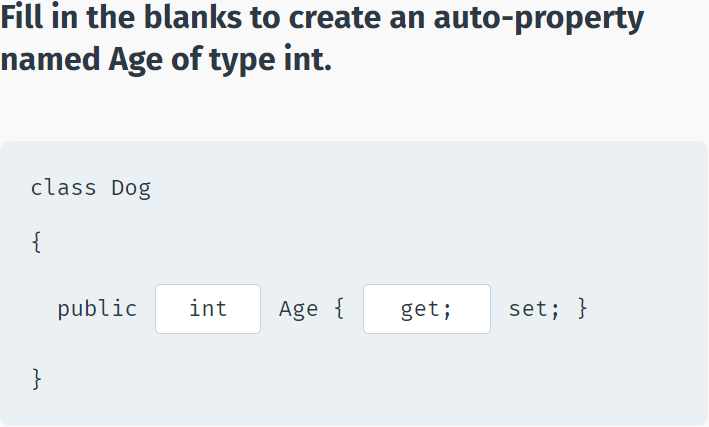
p.Name = "Bob";

Console.WriteLine(p.Name);

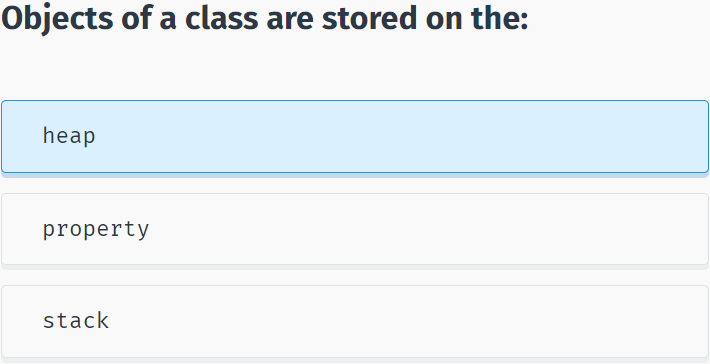
}

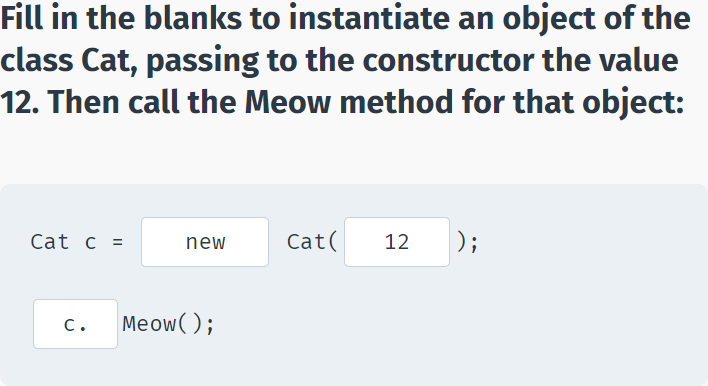
}

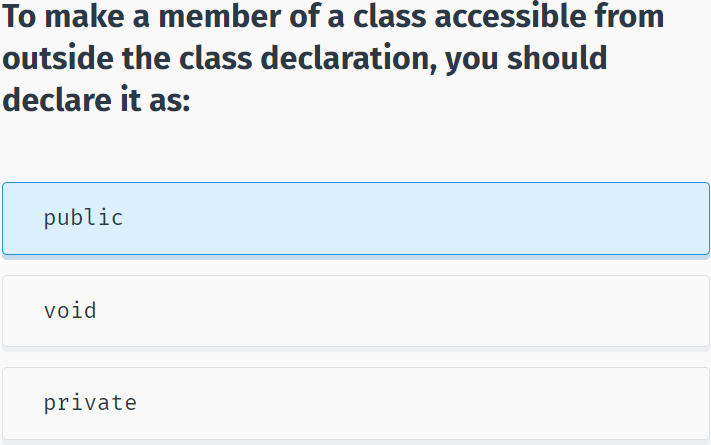
}



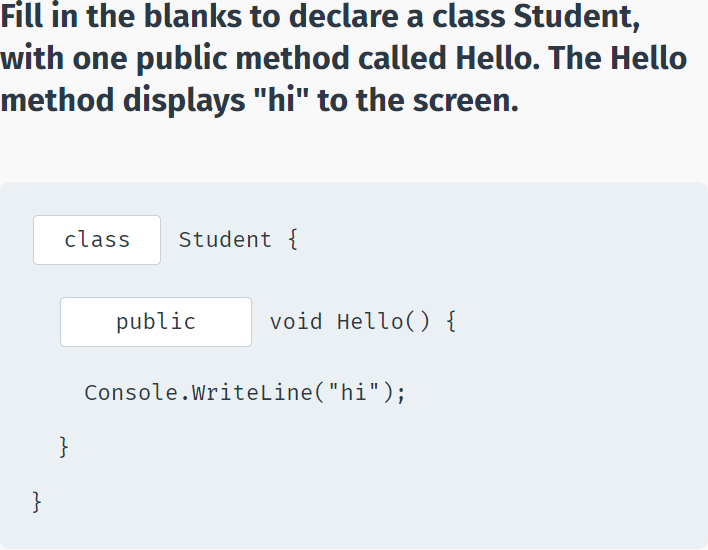
## Module 4 Quiz

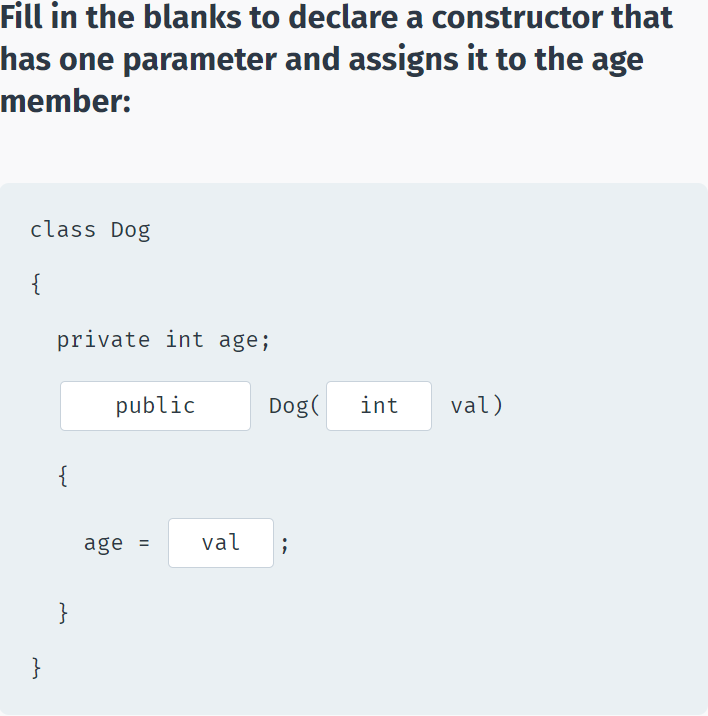


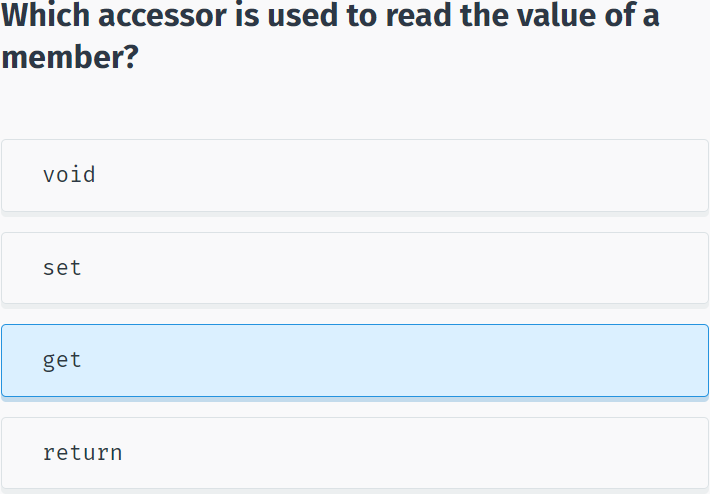












# Arrays & Strings

## Arrays

**Arrays**

C# provides numerous built-in classes to store and manipulate data.

One example of such a class is the **Array** class.

An array is a data structure that is used to store a collection of data. You can think of it as a collection of variables of the **same type**.

For example, consider a situation where you need to store 100 numbers. Rather than declare 100 different variables, you can just declare an array that stores 100 **elements**.

To declare an array, specify its element types with square brackets:

int[] myArray;

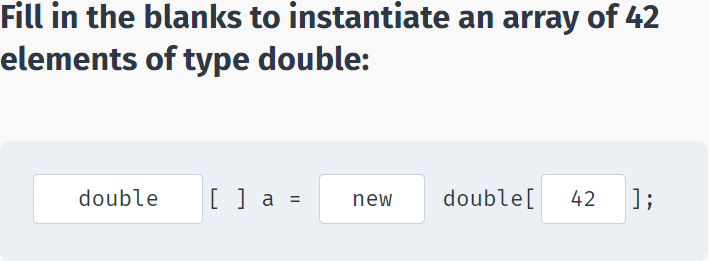
This statement declares an array of integers.

Since arrays are objects, we need to instantiate them with the **new** keyword:

int[] myArray = new int[5];

This instantiates an array named myArray that holds 5 integers.

Note the **square brackets** used to define the number of elements the array should hold.



After creating the array, you can assign values to individual elements by using the **index number**:

int[] myArray = new int[5];

myArray[0] = 23;

This will assign the value 23 to the first element of the array.

Arrays in C# are zero-indexed meaning the first member has index 0, the second has index 1, and so on.



We can provide initial values to the array when it is declared by using curly brackets:

string[] names = new string[3] {"John", "Mary", "Jessica"};

double[] prices = new double[4] {3.6, 9.8, 6.4, 5.9};

We can omit the size declaration when the number of elements are provided in the curly braces:

string[] names = new string[] {"John", "Mary", "Jessica"};

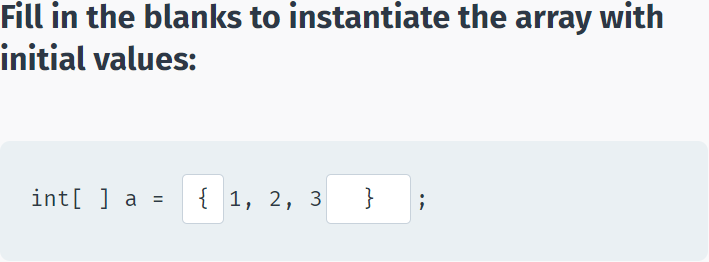
double[] prices = new double[] {3.6, 9.8, 6.4, 5.9};

We can even omit the **new** operator. The following statements are identical to the ones above:

string[] names = {"John", "Mary", "Jessica"};

double[] prices = {3.6, 9.8, 6.4, 5.9};

Array values should be provided in a comma separated list enclosed in {curly braces}.



As mentioned, each element of an array has an index number.

**For example, consider the following array:**

int[] b = {11, 45, 62, 70, 88};

The elements of **b** have the following indexes:



To access individual array elements, place the element's index number in square brackets following the array name.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[] b = {11, 45, 62, 70, 88};

Console.WriteLine(b[2]);

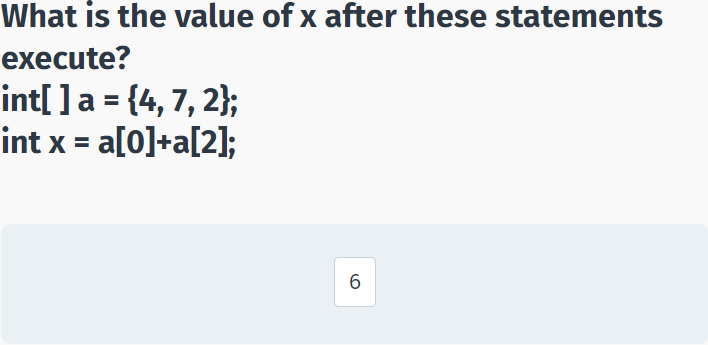
Console.WriteLine(b[3]);

}

}

}

Remember that the first element has index 0.



## Using Arrays in Loops

**Arrays & Loops**

It's occasionally necessary to iterate through the elements of an array, making element assignments based on certain calculations. This can be easily done using loops.

For example, you can declare an array of 10 integers and assign each element an even value with the following loop:

int[] a = new int[10];

for (int k = 0; k < 10; k++) {

a[k] = k\*2;

}

We can also use a loop to read the values of an array.

For example, we can display the contents of the array we just created:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[] a = new int[10];

for (int k = 0; k < 10; k++) {

a[k] = k\*2;

}

for (int k = 0; k < 10; k++) {

Console.WriteLine(a[k]);

}

}

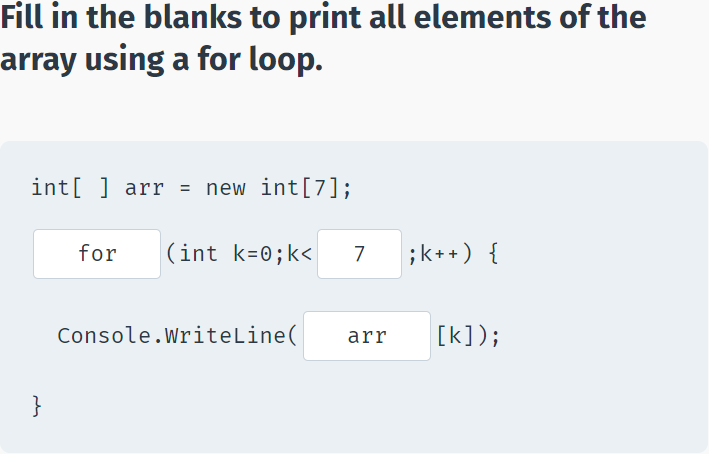
}

}

This will display the values of the elements of the array.

The variable **k** is used to access each array element.

The last index in the array is 9, so the for loop condition is k<10.



**The foreach Loop**

The **foreach loop** provides a shorter and easier way of accessing array elements.

The previous example of accessing the elements could be written using a **foreach loop**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[ ] a = new int[10];

for (int k = 0; k < 10; k++) {

a[k] = k\*2;

}

foreach (int k in a) {

Console.WriteLine(k);

}

}

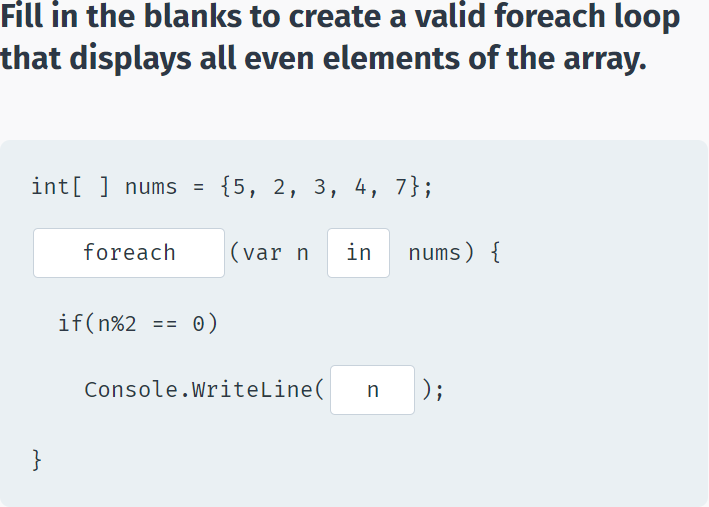
}

}

The **foreach loop** iterates through the array a and assigns the value of the current element to the variable **k** at each iteration of the loop. So, at the first iteration, k=a[0], at the second, k=a[1], etc.

The data type of the variable in the **foreach loop** should match the type of the array elements.

Often the keyword **var** is used as the type of the variable, as in: **foreach (var k in a)**. The compiler determines the appropriate type for var.



**Arrays**

The following code uses a **foreach** loop to calculate the sum of all the elements of an array:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[ ] arr = {11, 35, 62, 555, 989};

int sum = 0;

foreach (int x in arr) {

sum += x;

}

Console.WriteLine(sum);

}

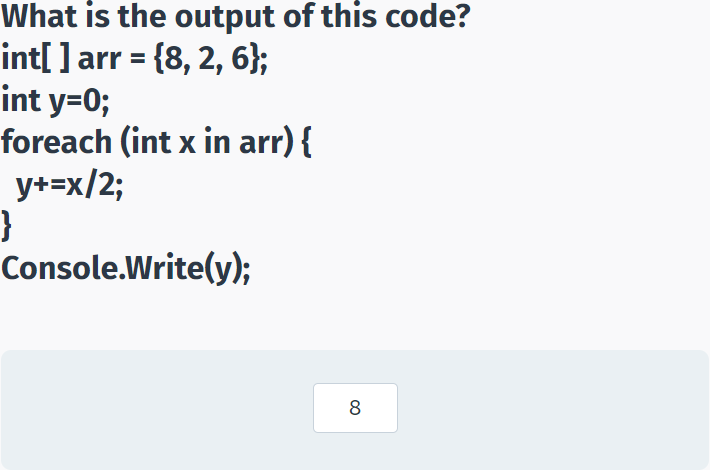
}

}

To review, we declared an array and a variable **sum** that will hold the sum of the elements.

Next, we utilized a **foreach** loop to iterate through each element of the array, adding the corresponding element's value to the **sum** variable.

The **Array** class provides some useful methods that will be discussed in the coming lessons.



## Multidimensional Arrays

**Multidimensional Arrays**

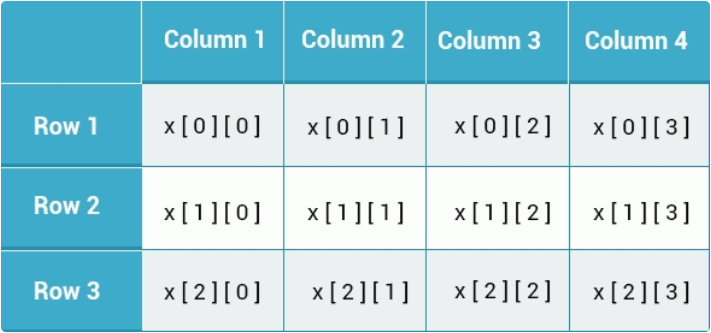
An array can have multiple dimensions. A **multidimensional array** is declared as follows:

type[, , … ,] arrayName = new type[size1, size2, …, sizeN];

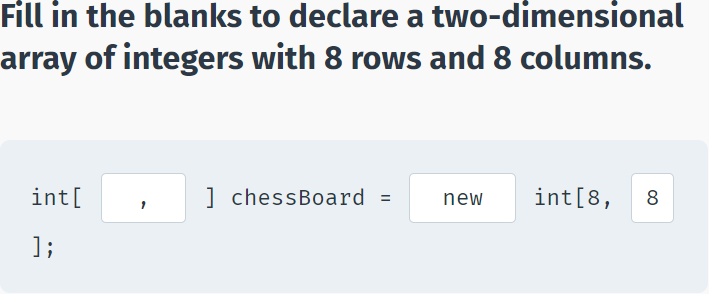
For example, let's define a two-dimensional 3x4 integer array:

int[ , ] x = new int[3,4];

Visualize this array as a table composed of 3 rows and 4 columns:



Array indexing starts from 0.



We can initialize multidimensional arrays in the same way as single-dimensional arrays.

**For example:**

int[ , ] someNums = { {2, 3}, {5, 6}, {4, 6} };

This will create an array with three rows and two columns. Nested curly brackets are used to define values for each row.

To access an element of the array, provide both indexes. For example **someNums[2, 0]** will return the value **4**, as it accesses the first column of the third row.

Let's create a program that will display the values of the array in the form of a table.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[ , ] someNums = { {2, 3}, {5, 6}, {4, 6} };

for (int k = 0; k < 3; k++) {

for (int j = 0; j < 2; j++) {

Console.Write(someNums[k, j]+" ");

}

Console.WriteLine();

}

}

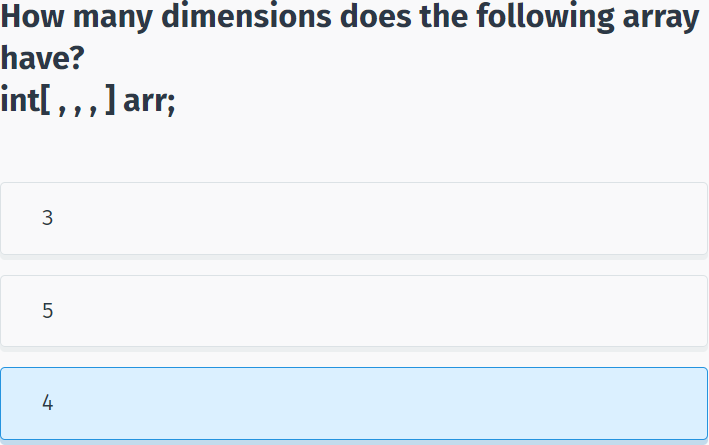
}

}

We have used two nested **for** loops, one to iterate through the rows and one through the columns.

The Console.WriteLine(); statement moves the output to a new line after one row is printed.

Arrays can have any number of dimensions, but keep in mind that arrays with more than three dimensions are harder to manage.



## Jagged Arrays

**Jagged Arrays**

A **jagged array** is an array whose elements are arrays. So it is basically an **array of arrays**.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

int[ ][ ] jaggedArr = new int[3][ ];

Each dimension is an array, so you can also initialize the array upon declaration like this:

int[ ][ ] jaggedArr = new int[ ][ ]

{

new int[ ] {1,8,2,7,9},

new int[ ] {2,4,6},

new int[ ] {33,42}

};

You can access individual array elements as shown in the example below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[ ][ ] jaggedArr = new int[ ][ ]

{

new int[ ] {1,8,2,7,9},

new int[ ] {2,4,6},

new int[ ] {33,42}

};

int x = jaggedArr[2][1];

Console.WriteLine(x);

}

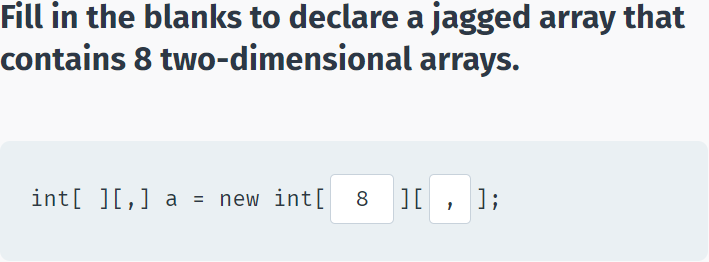
}

}

This accesses the second element of the third array.

A **jagged array** is an array-of-arrays, so an int[ ][ ] is an array of int[ ], each of which can be of different lengths and occupy their own block in memory.

A **multidimensional array** (int[,]) is a single block of memory (essentially a matrix). It always has the same amount of columns for every row.



## Array Properties & Methods

**Arrays Properties**

The Array class in C# provides various properties and methods to work with arrays.

For example, the **Length** and **Rank** properties return the number of elements and the number of dimensions of the array, respectively. You can access them using the dot syntax, just like any class members:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[ ] arr = {2, 4, 7};

Console.WriteLine(arr.Length);

Console.WriteLine(arr.Rank);

}

}

}

The **Length** property can be useful in **for** loops where you need to specify the number of times the loop should run.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[ ] arr = {2, 4, 7};

for(int k=0; k<arr.Length; k++) {

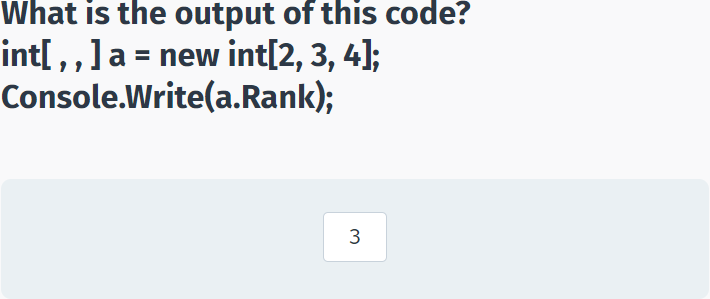
Console.WriteLine(arr[k]);

}

}

}

}



**Array Methods**

There are a number of methods available for arrays.

**Max** returns the largest value.

**Min** returns the smallest value.

**Sum** returns the sum of all elements.

For example:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int[ ] arr = { 2, 4, 7, 1};

Console.WriteLine(arr.Max());

Console.WriteLine(arr.Min());

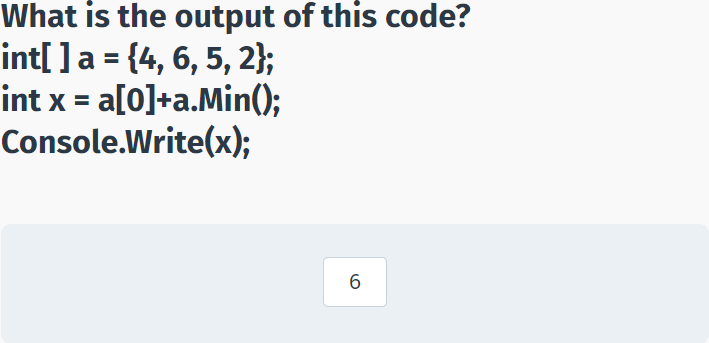
Console.WriteLine(arr.Sum());

}

}

}

C# also provides a static Array class with additional methods. You will learn about those in the next module.



## Working with Strings

**Strings**

It’s common to think of strings as arrays of characters. In reality, strings in C# are objects.

When you declare a **string** variable, you basically instantiate an object of type **String**.

String objects support a number of useful properties and methods:

**Length** returns the length of the string.

**IndexOf(value)** returns the index of the first occurrence of the value within the string.

**Insert(index, value)** inserts the value into the string starting from the specified index.

**Remove(index)** removes all characters in the string from the specified index.

**Replace(oldValue, newValue)** replaces the specified value in the string.

**Substring(index, length)** returns a substring of the specified length, starting from the specified index. If length is not specified, the operation continues to the end of the string.

**Contains(value)** returns true if the string contains the specified value.

**The examples below demonstrate each of the String members:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

string a = "some text";

Console.WriteLine(a.Length);

//Outputs 9

Console.WriteLine(a.IndexOf('t'));

//Outputs 5

a = a.Insert(0, "This is ");

Console.WriteLine(a);

//Outputs "This is some text"

a = a.Replace("This is", "I am");

Console.WriteLine(a);

//Outputs "I am some text"

if(a.Contains("some"))

Console.WriteLine("found");

//Outputs "found"

a = a.Remove(4);

Console.WriteLine(a);

//Outputs "I am"

a = a.Substring(2);

Console.WriteLine(a);

//Outputs "am"

}

}

}

You can also access characters of a string by its index, just like accessing elements of an array:

static void Main(string[] args)

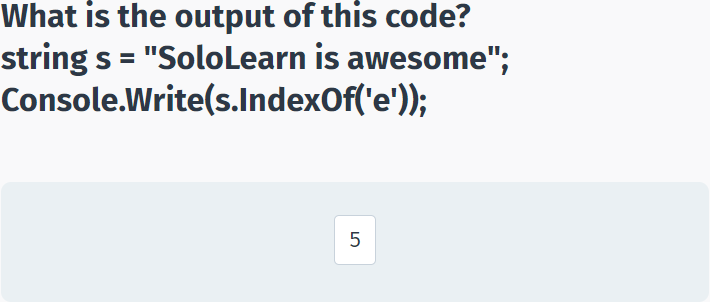
{

string a = "some text";

Console.WriteLine(a[2]);

}

Indexes in strings are similar to arrays, they start from 0.



**Working with Strings**

Let's create a program that will take a string, replace all occurrences of the word "dog" with "cat" and output the first sentence only.

static void Main(string[] args)

{

string text = "This is some text about a dog. The word dog appears in this text a number of times. This is the end.";

text = text.Replace("dog", "cat");

text = text.Substring(0, text.IndexOf(".")+1);

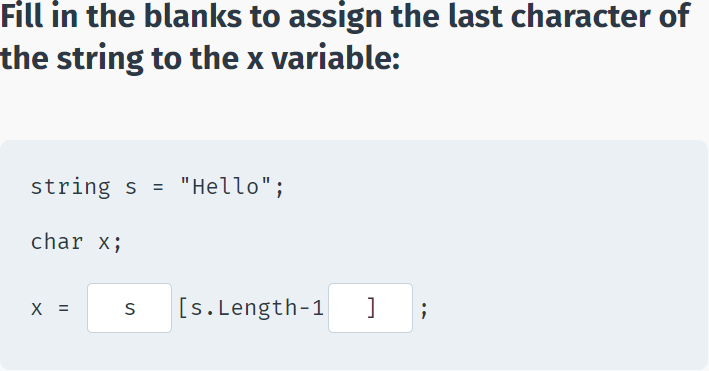
Console.WriteLine(text);

}

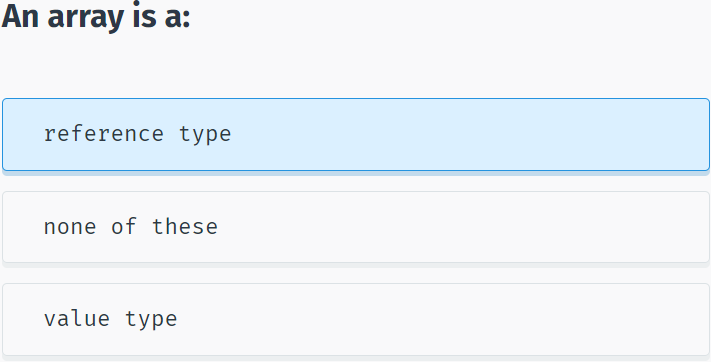
The code above replaces all occurrences of "dog" with "cat". After that it takes a substring of the original string starting from the first index until the first occurrence of a period character.

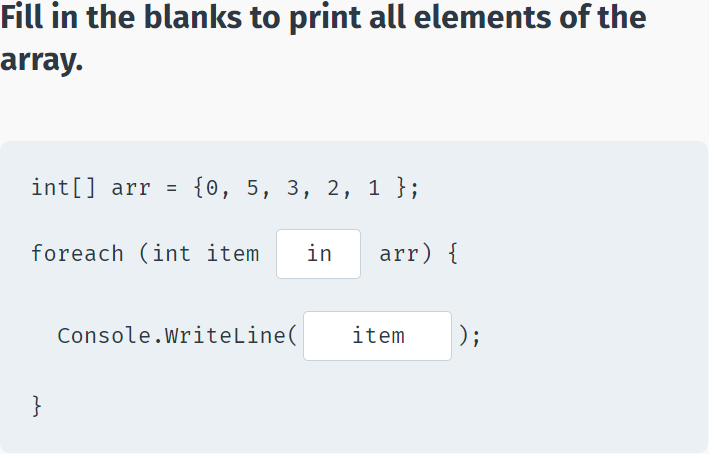
We add one to the index of the period to include the period in the substring.

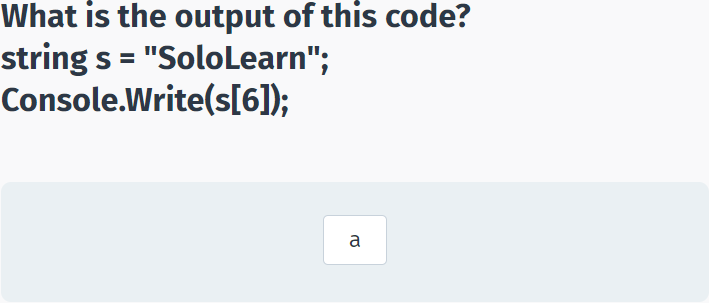
C# provides a solid collection of tools and methods to work and manipulate strings. You could, for example, find the number of times a specific word appears in a book with ease, using those methods.

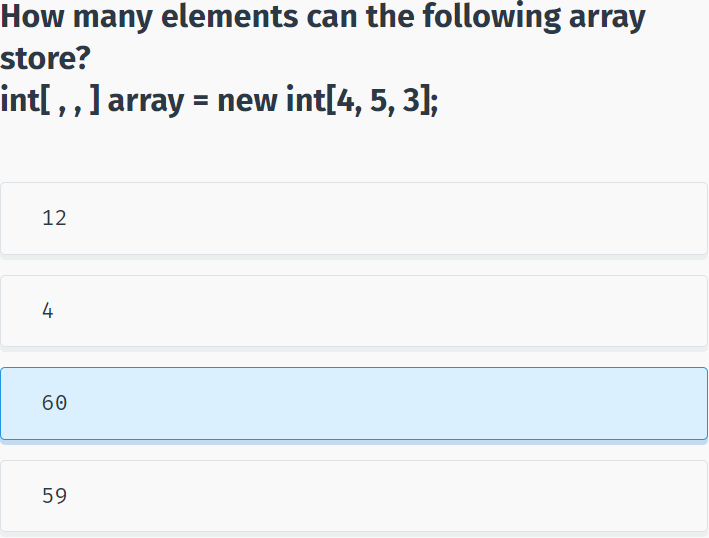


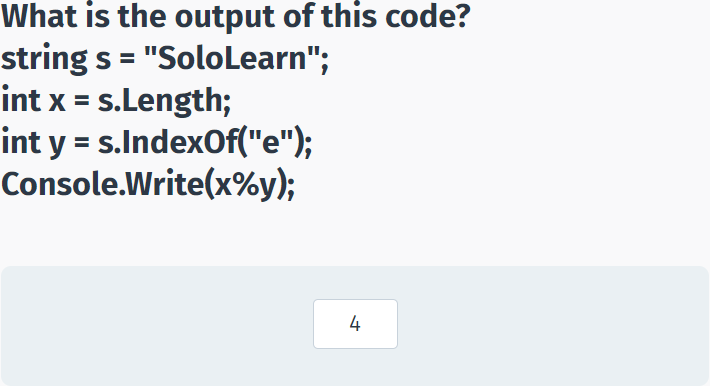
## Module 5 Quiz











# More On Classes

## Destructors

Destructors

As constructors are used when a class is instantiated, **destructors** are automatically invoked when an object is destroyed or deleted.

Destructors have the following attributes:

- A class can only have **one** destructor.

- Destructors cannot be called. They are invoked automatically.

- A destructor does not take modifiers or have parameters.

- The name of a destructor is exactly the same as the class prefixed with a **tilde (~)**.

**For Example:**

class Dog

{

~Dog()

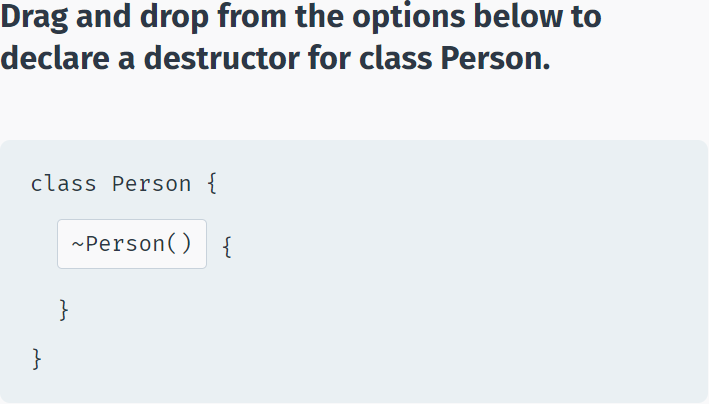
{

// code statements

}

}

Destructors can be very useful for releasing resources before coming out of the program. This can include closing files, releasing memory, and so on.



Let’s include WriteLine statements in the destructor and constructor of our class and see how the program behaves when an object of that class is created and when the program ends:

class Dog

{

public Dog() {

Console.WriteLine("Constructor");

}

~Dog() {

Console.WriteLine("Destructor");

}

}

static void Main(string[] args)

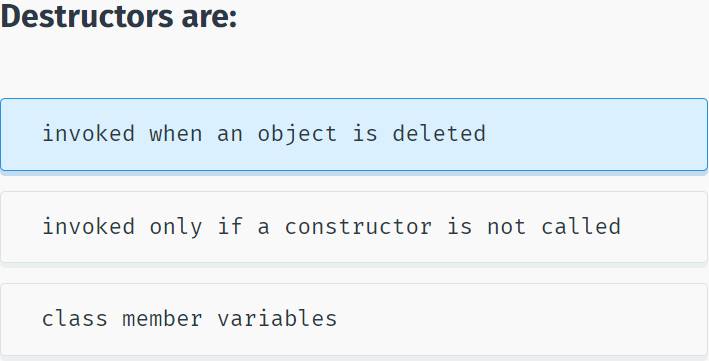
{

Dog d = new Dog();

}

When the program runs, it first creates the object, which calls the constructor. The object is deleted at the end of the program and the destructor is invoked when the program's execution is complete.

This can be useful, for example, if your class is working with storage or files. The constructor would initialize and open the files. Then, when the program ends, the destructor would close the files.



## Static Members

**Static**

Now it's time to discuss the **static** keyword.

You first noticed it in the Main method's declaration:

static void Main(string[] args)

Class members (variables, properties, methods) can also be declared as **static**. This makes those members belong to the class itself, instead of belonging to individual objects. No matter how many objects of the class are created, there is only **one** copy of the static member.

**For example:**

class Cat {

public static int count=0;

public Cat() {

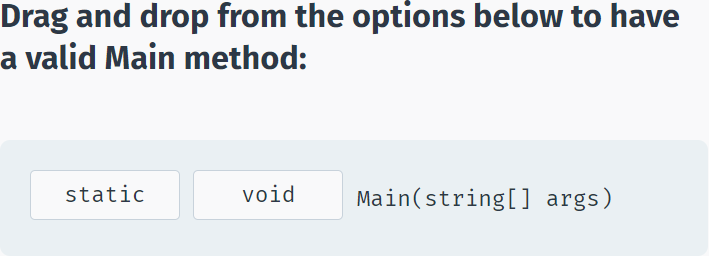
count++;

}

}

In this case, we declared a public member variable **count**, which is **static**. The constructor of the class increments the **count** variable by one.

No matter how many **Cat** objects are instantiated, there is always only one **count** variable that belongs to the **Cat** class because it was declared **static**.



Because of their global nature, static members can be accessed directly using the **class name** without an object.

**For example:**

class Cat {

public static int count=0;

public Cat() {

count++;

}

}

static void Main(string[] args)

{

Cat c1 = new Cat();

Cat c2 = new Cat();

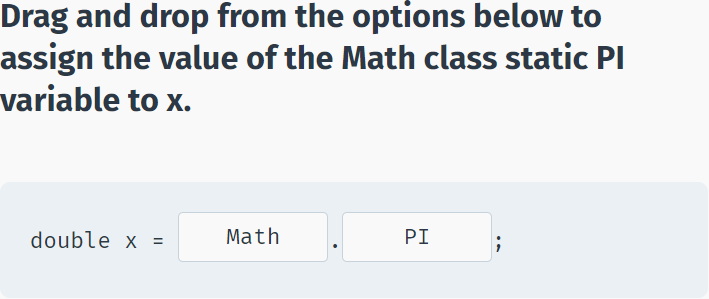
Console.WriteLine(Cat.count);

}

As you can see, we can access the static variable using the class name: **Cat.count**.

The **count** variable is shared between all Cat objects. For this class, each time an object is created, the static value is incremented. The program above demonstrates this when 2 is displayed after creating two objects of that class.

You must access **static** members using the class name. If you try to access them via an object of that class, you will generate an error.



**Static Methods**

The same concept applies to static methods.

**For example:**

class Dog

{

public static void Bark() {

Console.WriteLine("Woof");

}

}

static void Main(string[] args)

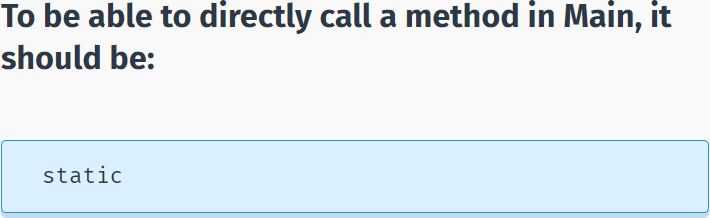
{

Dog.Bark();

}

Static methods can access **only** static members.

The Main method is **static**, as it is the starting point of any program. Therefore any method called directly from Main had to be **static**.



**Static**

**Constant** members are static by definition.

**For example:**

class MathClass {

public const int ONE = 1;

}

static void Main(string[] args)

{

Console.Write(MathClass.ONE);

}

As you can see, we access the property **ONE** using the name of the class, just like a static member. This is because all **const** members are **static** by default.

**Static Constructors**

Constructors can be declared **static** to initialize static members of the class.

The static constructor is automatically called once when we access a static member of the class.

**For example:**

class SomeClass {

public static int X { get; set; }

public static int Y { get; set; }

static SomeClass() {

X = 10;

Y = 20;

}

}

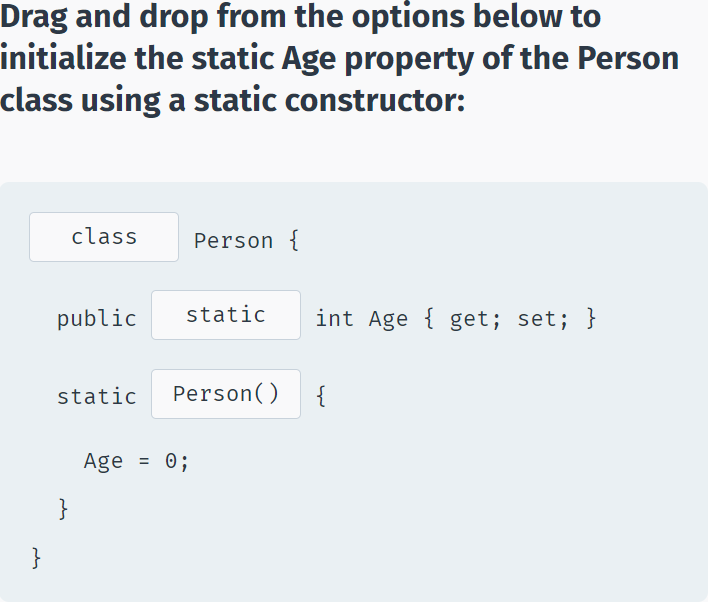
static void Main(string[] args)

{

Console.WriteLine(SomeClass.X);

}

The constructor will get called once when we try to access SomeClass.X or SomeClass.Y.



## Static Classes

**Static Classes**

An entire class can be declared as **static**.

A **static class** can contain only static members.

You cannot instantiate an object of a static class, as only one instance of the static class can exist in a program.

Static classes are useful for combining logical properties and methods. A good example of this is the **Math** class.

It contains various useful properties and methods for mathematical operations.

**For example, the Pow method raises a number to a power:**

static void Main(string[] args)

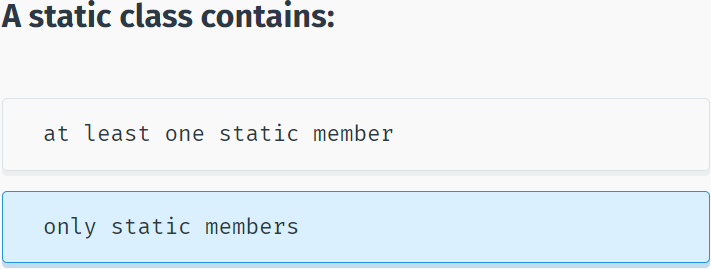
{

Console.WriteLine(Math.Pow(2, 3));

}

You access all members of the Math class using the class name, without declaring an object.

Tap next to learn about the available methods of the Math class.



There are a number of useful static methods and properties available in C#:

**Math**

Math.**PI** the constant PI.

Math.**E** represents the natural logarithmic base e.

Math.**Max()** returns the larger of its two arguments.

Math.**Min()** returns the smaller of its two arguments.

Math.**Abs()** returns the absolute value of its argument.

Math.**Sin()** returns the sine of the specified angle.

Math.**Cos()** returns the cosine of the specified angle.

Math.**Pow()** returns a specified number raised to the specified power.

Math.**Round()** rounds the decimal number to its nearest integral value.

Math.**Sqrt()** returns the square root of a specified number.

**Array**

The **Array** class includes some static methods for manipulating arrays:

int[] arr = {1, 2, 3, 4};

Array.Reverse(arr);

//arr = {4, 3, 2, 1}

Array.Sort(arr);

//arr = {1, 2, 3, 4}

**String**

string s1 = "some text";

string s2 = "another text";

String.Concat(s1, s2); // combines the two strings

String.Equals(s1, s2); // returns false

**DateTime**

The **DateTime** structure allows you to work with dates.

static void Main(string[] args)

{

Console.WriteLine(DateTime.Now);

Console.WriteLine(DateTime.Today);

Console.WriteLine(DateTime.DaysInMonth(2016, 2));

}

The **Console** class is also an example of a static class. We use its static **WriteLine()** method to output to the screen, or the static **ReadLine()** method to get user input.

The **Convert** class used to convert value types is also a static class.



## this & readonly

**The this Keyword**

The **this** keyword is used inside the class and refers to the current instance of the class, meaning it refers to the current object.

One of the common uses of **this** is to distinguish class members from other data, such as local or formal parameters of a method, as shown in the following example:

class Person {

private string name;

public Person(string name) {

this.name = name;

}

}

Here, **this.name** represents the member of the class, whereas **name** represents the parameter of the constructor.

Another common use of **this** is for passing the current instance to a method as parameter: ShowPersonInfo(**this**);



**The readonly Modifier**

The **readonly** modifier prevents a member of a class from being modified after construction. It means that the field declared as **readonly** can be modified only when you declare it or from within a constructor.

**For example:**

class Person {

private readonly string name = "John";

public Person(string name) {

this.name = name;

}

}

If we try to modify the **name** field anywhere else, we will get an error.

There are three major differences between **readonly** and **const** fields.

First, a constant field must be initialized when it is declared, whereas a readonly field can be declared without initialization, as in:

**For example:**

readonly string name; // OK

const double PI; ->error

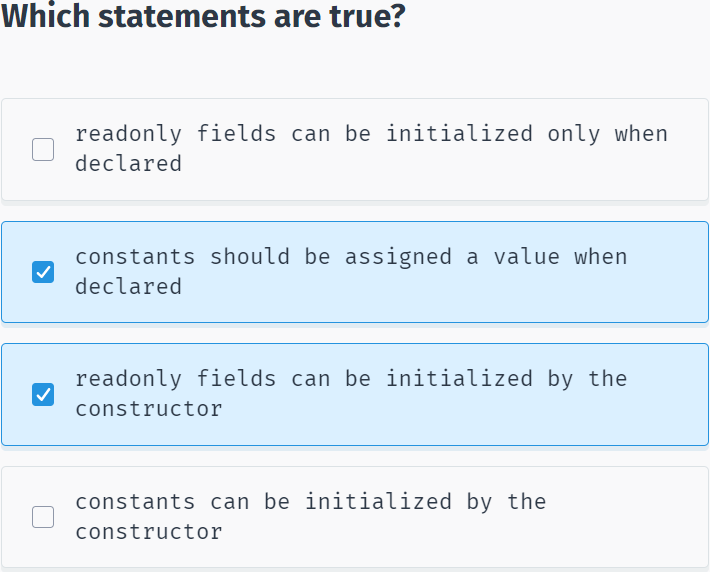
Second, a **readonly** field value can be changed in a constructor, but a constant value cannot.

Third, the **readonly** field can be assigned a value that is a result of a calculation, but constants cannot, as in:

readonly double a = Math.Sin(60); // OK

const double b = Math.Sin(60); ->error

The readonly modifier prevents a member of a class from being modified after construction.



## Indexers

**Indexers**

An **indexer** allows objects to be indexed like an array.

As discussed earlier, a string variable is actually an object of the **String** class. Further, the String class is actually an array of Char objects. In this way, the string class implements an indexer so we can access any character (Char object) by its index:

static void Main(string[] args)

{

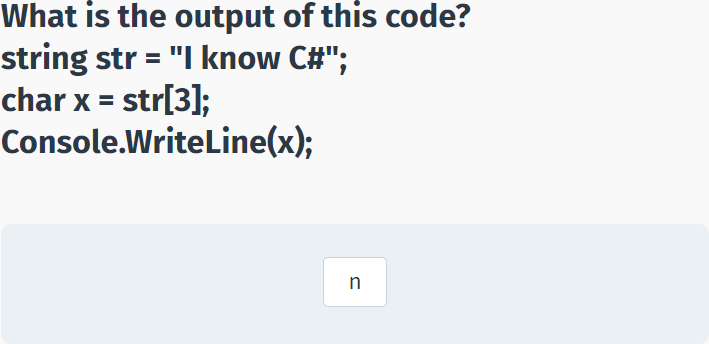
string str = "Hello World";

char x = str[4];

Console.WriteLine(x);

}

Arrays use integer indexes, but indexers can use any type of index, such as strings, characters, etc.



Declaration of an indexer is to some extent similar to a property. The difference is that indexer accessors require an **index**.

Like a property, you use **get** and **set** accessors for defining an indexer. However, where properties return or set a specific data member, indexers return or set a particular value from the object instance.

Indexers are defined with the **this** keyword.

**For example:**

class Clients {

private string[] names = new string[10];

public string this[int index] {

get {

return names[index];

}

set {

names[index] = value;

}

}

}

As you can see, the indexer definition includes the **this** keyword and an index, which is used to get and set the appropriate value.

Now, when we declare an object of class Clients, we use an index to refer to specific objects like the elements of an array:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Clients {

private string[] names = new string[10];

public string this[int index] {

get {

return names[index];

}

set {

names[index] = value;

}

}

}

static void Main(string[] args)

{

Clients c = new Clients();

c[0] = "Dave";

c[1] = "Bob";

Console.WriteLine(c[1]);

}

}

}

You typically use an indexer if the class represents a list, collection, or array of objects.



## Operator Overloading

**Operator Overloading**

Most operators in C# can be **overloaded**, meaning they can be redefined for custom actions.

For example, you can redefine the action of the plus (+) operator in a custom class.

Consider the **Box** class that has **Height** and **Width** properties:

class Box {

public int Height {get; set;}

public int Width {get; set;}

public Box(int h, int w) {

Height = h;

Width = w;

}

}

static void Main(string[] args) {

Box b1 = new Box(14, 3);

Box b2 = new Box(5, 7);

}

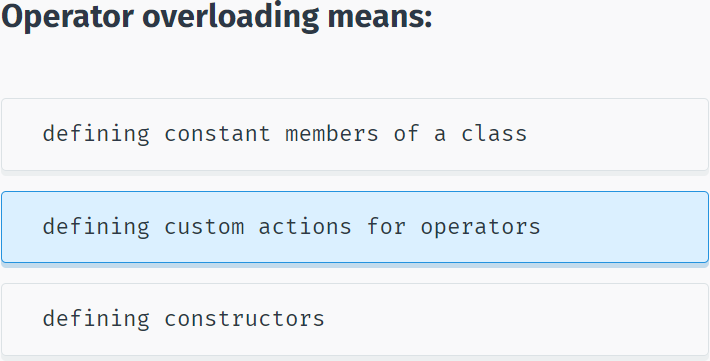
We would like to add these two Box objects, which would result in a new, bigger Box.

So, basically, we would like the following code to work:

Box b3 = b1 + b2;

The Height and Width properties of object b3 should be equal to the sum of the corresponding properties of the b1 and b2 objects.

This is achieved through **operator overloading**. Tap next to learn more!



Overloaded operators are methods with special names, where the keyword **operator** is followed by the symbol for the operator being defined.

Similar to any other method, an overloaded operator has a return type and a parameter list.

For example, for our **Box** class, we overload the + operator:

public static Box operator+ (Box a, Box b) {

int h = a.Height + b.Height;

int w = a.Width + b.Width;

Box res = new Box(h, w);

return res;

}

The method above defines an overloaded **operator +** with two Box object parameters and returning a new Box object whose Height and Width properties equal the sum of its parameter's corresponding properties.

Additionally, the overloaded operator must be **static**.

**Putting it all together:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Box {

public int Height { get; set; }

public int Width { get; set; }

public Box(int h, int w) {

Height = h;

Width = w;

}

public static Box operator+(Box a, Box b) {

int h = a.Height + b.Height;

int w = a.Width + b.Width;

Box res = new Box(h, w);

return res;

}

}

static void Main(string[] args)

{

Box b1 = new Box(14, 3);

Box b2 = new Box(5, 7);

Box b3 = b1 + b2;

Console.WriteLine(b3.Height);

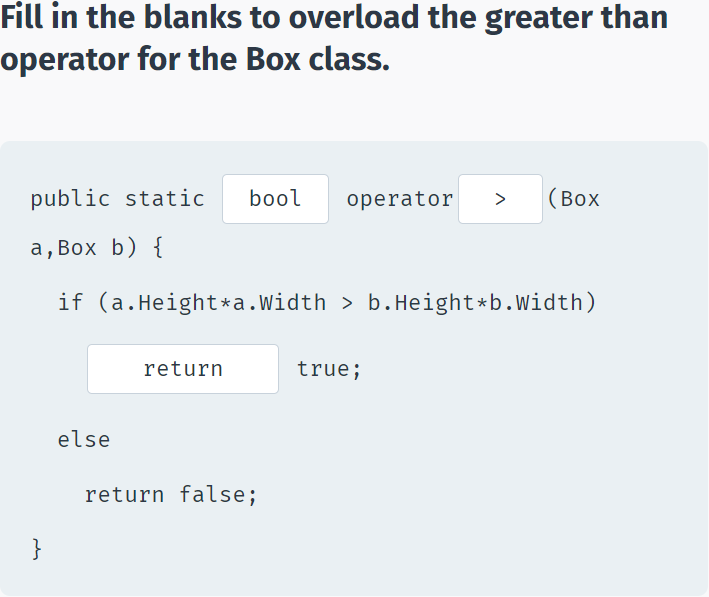
Console.WriteLine(b3.Width);

}

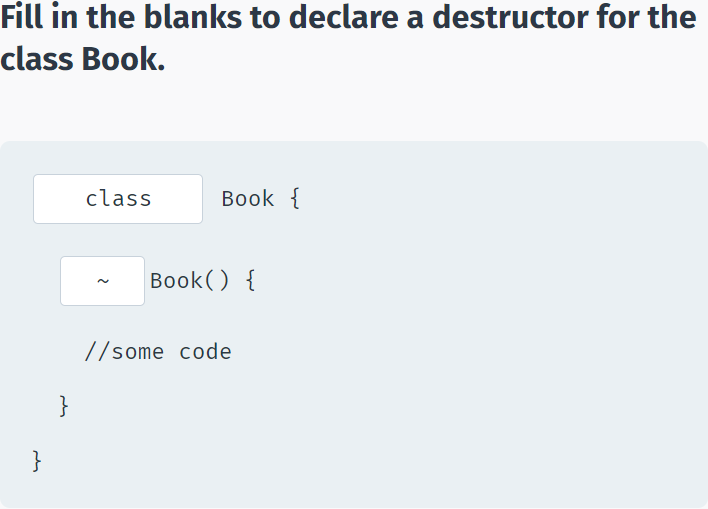
}

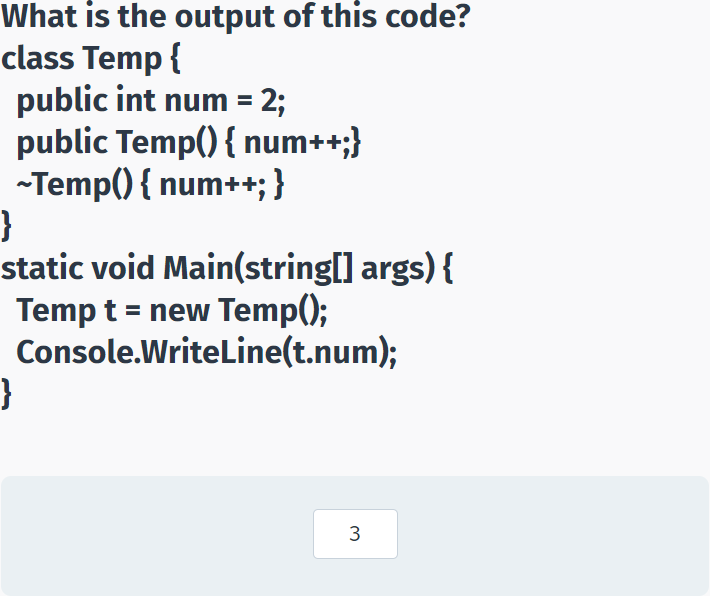
}

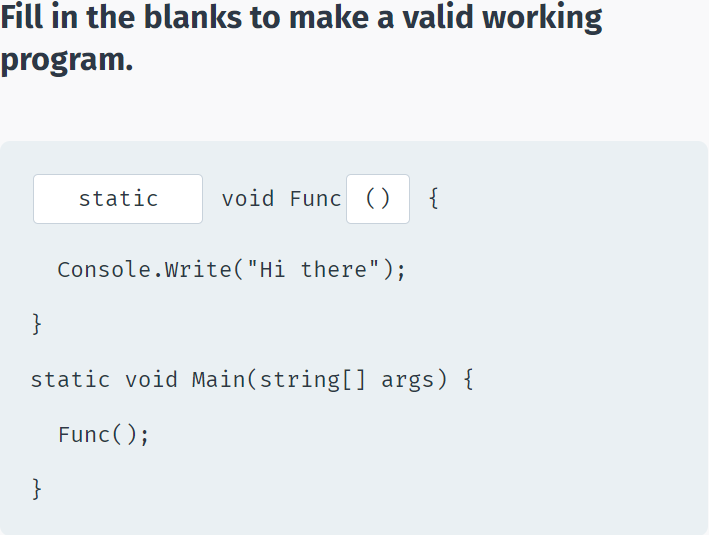
All arithmetic and comparison operators can be overloaded. For instance, you could define greater than and less than operators for the boxes that would compare the Boxes and return a **boolean** result. Just keep in mind that when overloading the greater than operator, the less than operator should also be defined.

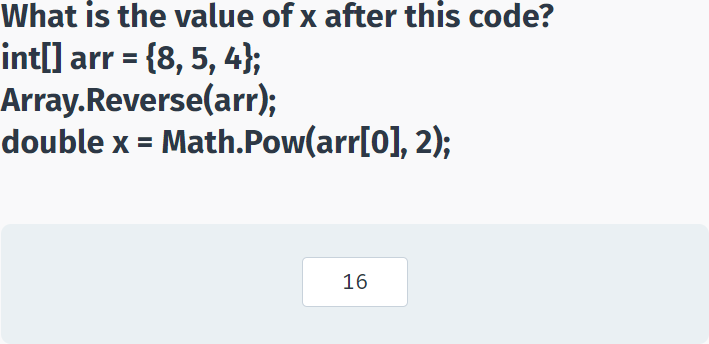


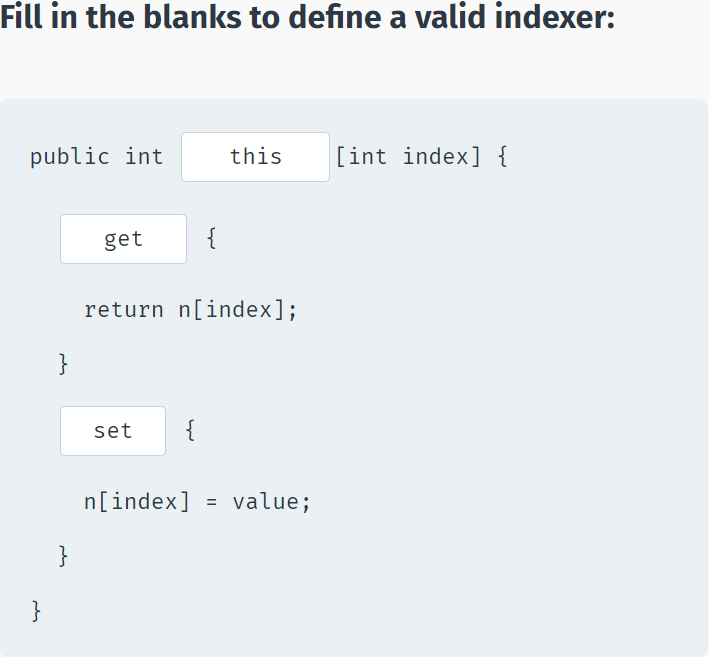
## Module 6 Quiz

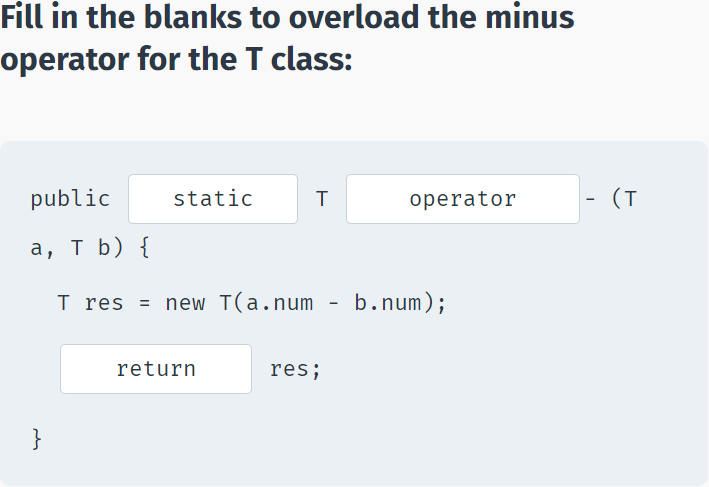












# Inheritance & Polymorphism

## Inheritance

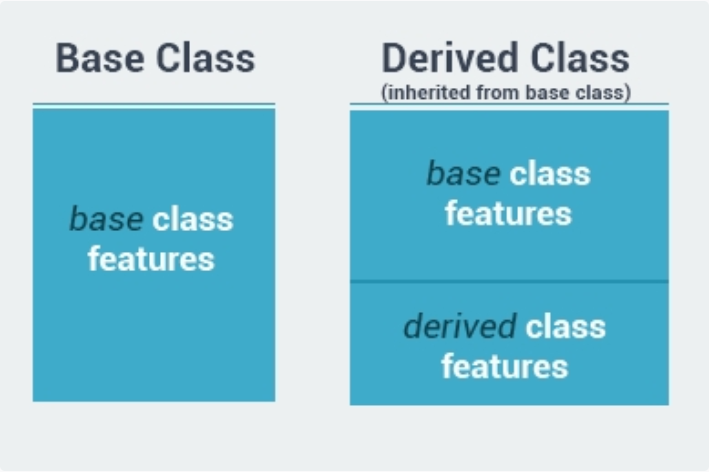
**Inheritance**

**Inheritance** allows us to define a class based on another class. This makes creating and maintaining an application easy.

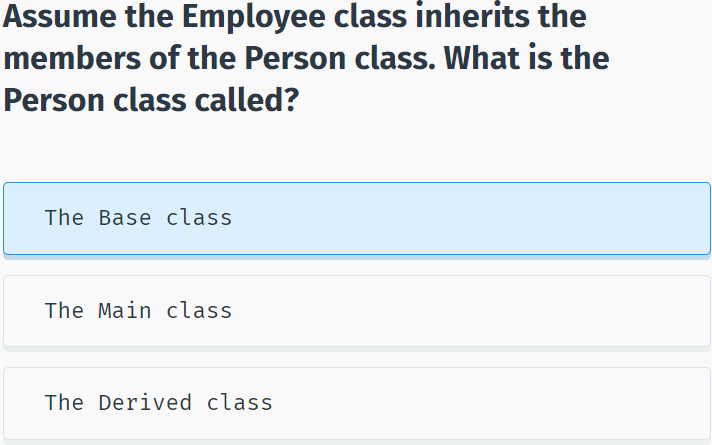
The class whose properties are inherited by another class is called the **Base** class. The class which inherits the properties is called the **Derived** class.

For example, base class **Animal** can be used to derive **Cat** and **Dog** classes.

The derived class inherits all the features from the base class, and can have its own additional features.



Inheritance allows us to define a class based on another class.



Let's define our base class **Animal**:

class Animal {

public int Legs {get; set;}

public int Age {get; set;}

}

Now we can derive class **Dog** from it:

class Dog : Animal {

public Dog() {

Legs = 4;

}

public void Bark() {

Console.Write("Woof");

}

}

Note the syntax for a derived class. A **colon** and the name of the **base** class follow the name of the derived class.

All public members of **Animal** become public members of Dog. That is why we can access the **Legs** member in the **Dog** constructor.

Now we can instantiate an object of type **Dog** and access the inherited members as well as call its own **Bark** method.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Animal {

public int Legs {get; set;}

public int Age {get; set;}

}

class Dog : Animal {

public Dog() {

Legs = 4;

}

public void Bark() {

Console.Write("Woof");

}

}

static void Main(string[] args)

{

Dog d = new Dog();

Console.WriteLine(d.Legs);

d.Bark();

}

}

}

A base class can have multiple derived classes. For example, a **Cat** class can inherit from **Animal**.

Inheritance allows the derived class to reuse the code in the base class without having to rewrite it. And the derived class can be customized by adding more members. In this manner, the derived class extends the functionality of the base class.



A derived class inherits all the members of the base class, including its methods.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Person {

public void Speak() {

Console.WriteLine("Hi there");

}

}

class Student : Person {

int number;

}

static void Main(string[] args)

{

Student s = new Student();

s.Speak();

}

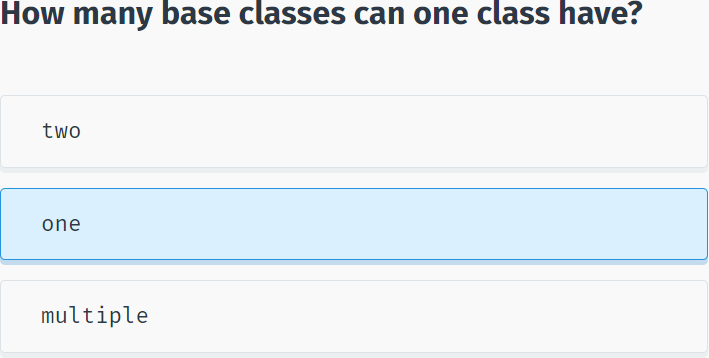
}

}

We created a **Student** object and called the **Speak** method, which was declared in the base class **Person**.

C# does not support multiple inheritance, so you cannot inherit from multiple classes.

However, you can use **interfaces** to implement multiple inheritance. You will learn more about **interfaces** in the coming lessons.



## Protected Members

**protected**

Up to this point, we have worked exclusively with **public** and **private** access modifiers.

Public members may be accessed from anywhere outside of the class, while access to **private** members is limited to their class.

The **protected** access modifier is very similar to **private** with one difference; it can be accessed in the derived classes. So, a **protected** member is accessible only from derived classes.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Person {

protected int Age {get; set;}

protected string Name {get; set;}

}

class Student : Person {

public Student(string nm) {

Name = nm;

}

public void Speak() {

Console.Write("Name: "+Name);

}

}

static void Main(string[] args)

{

Student s = new Student("David");

s.Speak();

}

}

}

As you can see, we can access and modify the **Name** property of the base class from the derived class.

But, if we try to access it from outside code, we will get an error:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Person {

protected int Age {get; set;}

protected string Name {get; set;}

}

class Student : Person {

public Student(string nm) {

Name = nm;

}

public void Speak() {

Console.Write("Name: "+Name);

}

}

static void Main(string[] args)

{

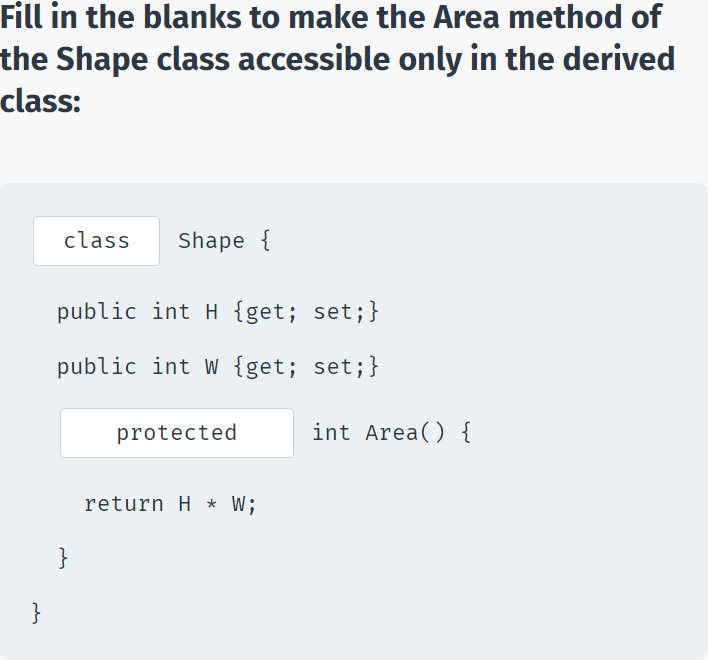
Student s = new Student("David");

s.Name = "Bob";

}

}

}



**sealed**

A class can prevent other classes from inheriting it, or any of its members, by using the **sealed** modifier.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

sealed class Animal {

//some code

}

class Dog : Animal { } ->error

static void Main(string[] args)

{

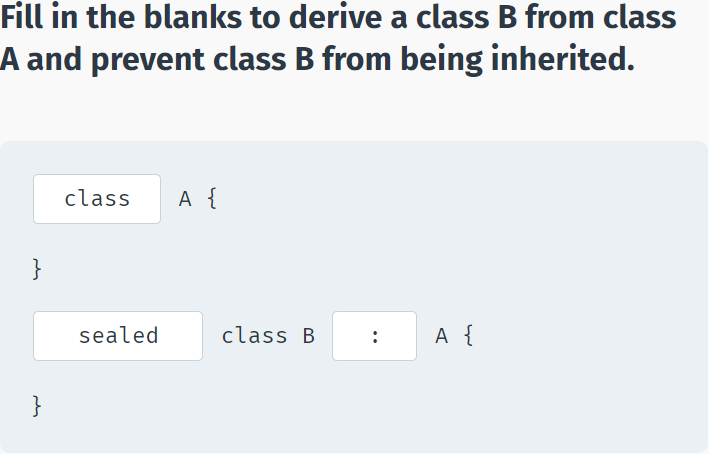
}

}

}

In this case, we cannot derive the Dog class from the Animal class because Animal is **sealed**.

The **sealed** keyword provides a level of protection to your class so that other classes cannot inherit from it.



## Derived Class Constructor & Destructor

**Inheritance**

Constructors are called when objects of a class are created. With inheritance, the base class constructor and destructor are not inherited, so you should define constructors for the derived classes.

However, the base class constructor and destructor are being invoked automatically when an object of the derived class is created or deleted.

**Consider the following example:**

class Animal {

public Animal() {

Console.WriteLine("Animal created");

}

~Animal() {

Console.WriteLine("Animal deleted");

}

}

class Dog: Animal {

public Dog() {

Console.WriteLine("Dog created");

}

~Dog() {

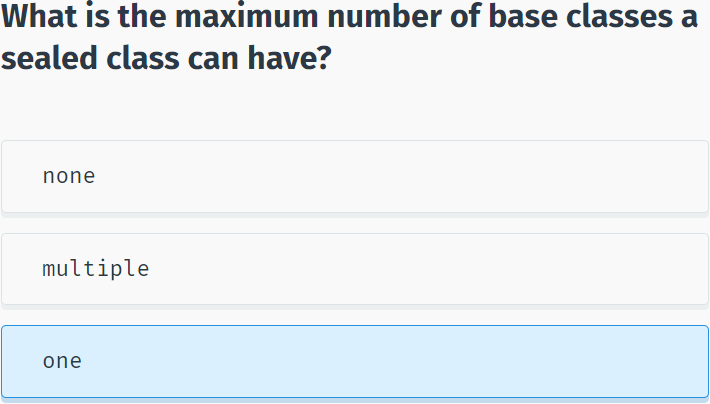
Console.WriteLine("Dog deleted");

}

}

We have defined the Animal class with a constructor and destructor and a derived Dog class with its own constructor and destructor.

So what will happen when we create an object of the derived class? Tap next to find out!



Let's create a Dog object:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Animal {

public Animal() {

Console.WriteLine("Animal created");

}

~Animal() {

Console.WriteLine("Animal deleted");

}

}

class Dog: Animal {

public Dog() {

Console.WriteLine("Dog created");

}

~Dog() {

Console.WriteLine("Dog deleted");

}

}

static void Main(string[] args)

{

Dog d = new Dog();

}

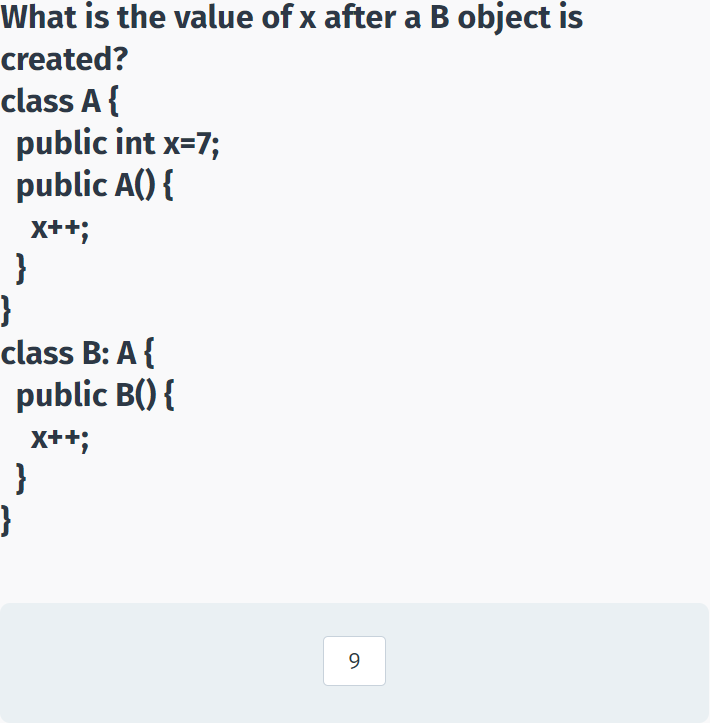
}

}

Note that the base class constructor is called first and the derived class constructor is called next.

When the object is destroyed, the derived class destructor is invoked and then the base class destructor is invoked.

You can think of it as the following: The derived class needs its base class in order to work, which is why the base class constructor is called first.



## Polymorphism

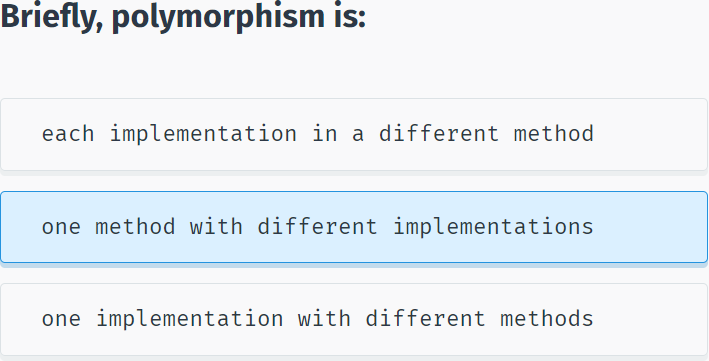
**Polymorphism**

The word **polymorphism** means "having many forms".

Typically, polymorphism occurs when there is a hierarchy of classes and they are related through inheritance from a common base class.

Polymorphism means that a call to a member method will cause a different implementation to be executed depending on the **type** of object that invokes the method.

Simply, polymorphism means that a single method can have a number of different implementations.



Consider having a program that allows users to draw different shapes. Each shape is drawn differently, and you do not know which shape the user will choose.

Here, polymorphism can be leveraged to invoke the appropriate **Draw** method of any derived class by overriding the same method in the base class. Such methods must be declared using the **virtual** keyword in the base class.

**For example:**

class Shape {

public virtual void Draw() {

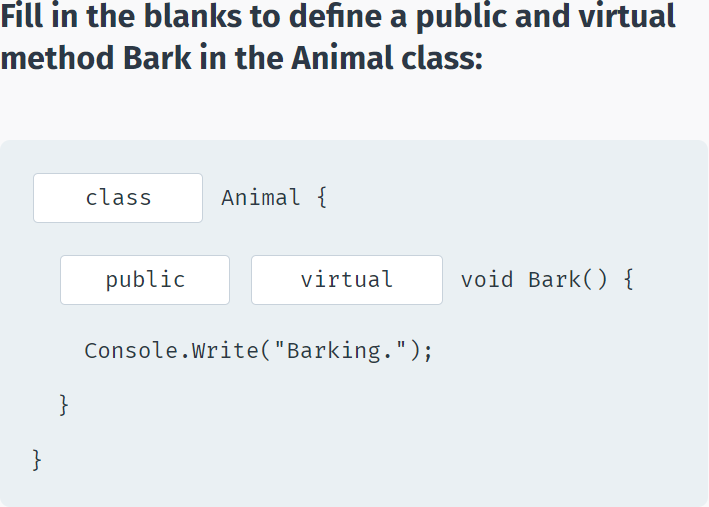
Console.Write("Base Draw");

}

}

The **virtual** keyword allows methods to be overridden in derived classes.

Virtual methods enable you to work with groups of related objects in a uniform way.



Now, we can derive different shape classes that define their own Draw methods using the **override** keyword:

class Circle : Shape {

public override void Draw() {

// draw a circle...

Console.WriteLine("Circle Draw");

}

}

class Rectangle : Shape {

public override void Draw() {

// draw a rectangle...

Console.WriteLine("Rect Draw");

}

}

The virtual Draw method in the Shape base class can be overridden in the derived classes. In this case, Circle and Rectangle have their own Draw methods.

Now, we can create separate Shape objects for each derived type and then call their Draw methods:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Shape {

public virtual void Draw() {

Console.Write("Base Draw");

}

}

class Circle : Shape {

public override void Draw() {

// draw a circle...

Console.WriteLine("Circle Draw");

}

}

class Rectangle : Shape {

public override void Draw() {

// draw a rectangle...

Console.WriteLine("Rect Draw");

}

}

static void Main(string[] args)

{

Shape c = new Circle();

c.Draw();

Shape r = new Rectangle();

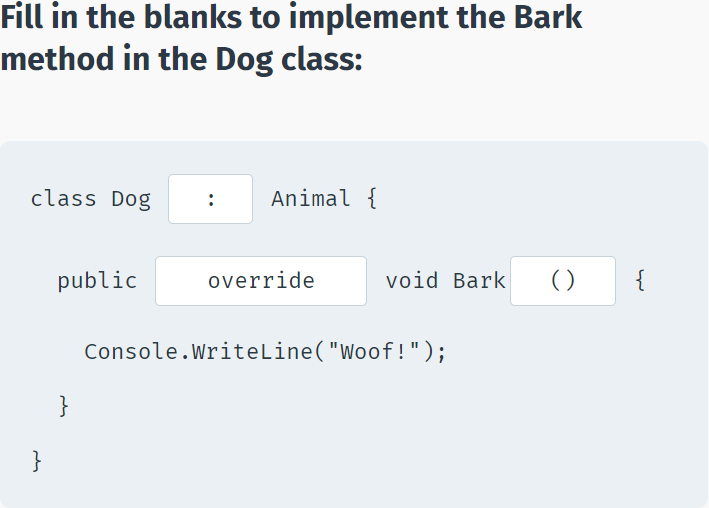
r.Draw();

}

}

}

As you can see, each object invoked its own **Draw** method, thanks to polymorphism.



To summarize, **polymorphism** is a way to call the same method for different objects and generate different results based on the object type. This behavior is achieved through virtual methods in the base class.

To implement this, we create objects of the base type, but instantiate them as the derived type:

Shape c = new Circle();

**Shape** is the base class. **Circle** is the derived class.

So why use polymorphism? We could just instantiate each object of its type and call its method, as in:

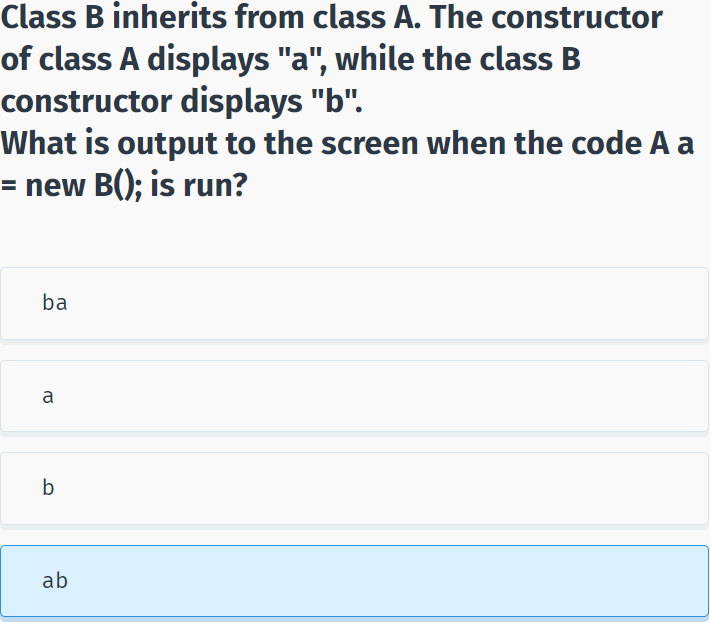
Circle c = new Circle();

c.Draw();

The polymorphic approach allows us to treat each object the same way. As all objects are of type Shape, it is easier to maintain and work with them. You could, for example, have a list (or array) of objects of that type and work with them dynamically, without knowing the actual derived type of each object.

Polymorphism can be useful in many cases. For example, we could create a game where we would have different Player types with each Player having a separate behavior for the Attack method.

In this case, Attack would be a virtual method of the base class Player and each derived class would override it.



## Abstract Classes

**Abstract Classes**

As described in the previous example, polymorphism is used when you have different derived classes with the same method, which has different implementations in each class. This behavior is achieved through **virtual** methods that are **overridden** in the derived classes.

In some situations there is no meaningful need for the virtual method to have a separate definition in the base class.

These methods are defined using the **abstract** keyword and specify that the derived classes must define that method on their own.

You cannot create objects of a class containing an abstract method, which is why the class itself should be abstract.

We could use an abstract method in the Shape class:

abstract class Shape {

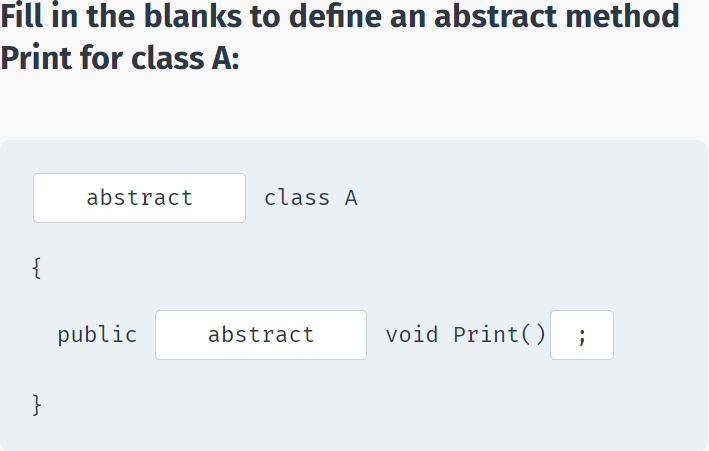
public abstract void Draw();

}

As you can see, the **Draw** method is abstract and thus has no body. You do not even need the curly brackets; just end the statement with a semicolon.

The Shape class itself must be declared **abstract** because it contains an **abstract** method. Abstract method declarations are only permitted in **abstract** classes.

Remember, **abstract** method declarations are only permitted in **abstract** classes. Members marked as abstract, or included in an **abstract** class, must be implemented by classes that derive from the **abstract** class. An abstract class can have multiple abstract members.



An abstract class is intended to be a base class of other classes. It acts like a template for its derived classes.

Now, having the abstract class, we can derive the other classes and define their own **Draw**() methods:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

abstract class Shape {

public abstract void Draw();

}

class Circle : Shape {

public override void Draw() {

Console.WriteLine("Circle Draw");

}

}

class Rectangle : Shape {

public override void Draw() {

Console.WriteLine("Rect Draw");

}

}

static void Main(string[] args)

{

Shape c = new Circle();

c.Draw();

}

}

}

Abstract classes have the following features:

- An abstract class cannot be instantiated.

- An abstract class may contain abstract methods and accessors.

- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

It is not possible to modify an **abstract** class with the **sealed** modifier because the two modifiers have opposite meanings. The **sealed** modifier prevents a class from being inherited and the **abstract** modifier requires a class to be inherited.



## Interfaces

**Interfaces**

An **interface** is a completely abstract class, which contains **only** abstract members.

It is declared using the **interface** keyword:

public interface IShape

{

void Draw();

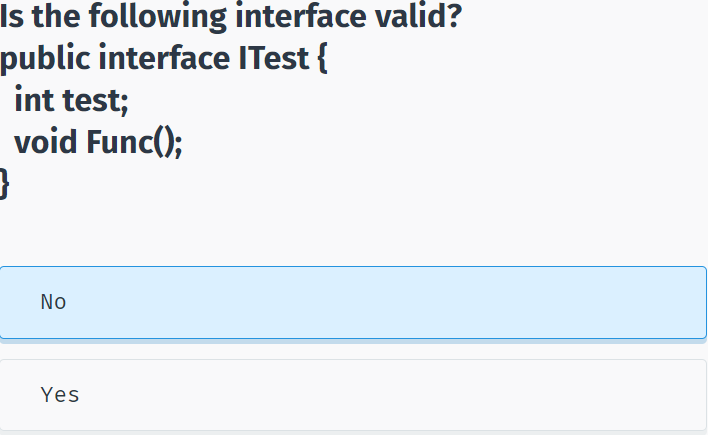
}

All members of the interface are by default abstract, so no need to use the abstract keyword.

Also, all members of an interface are always public, and no access modifiers can be applied to them.

It is common to use the capital letter **I** as the starting letter for an interface name.

Interfaces can contain properties, methods, etc. but **cannot** contain fields (variables).



When a class **implements** an interface, it must also implement, or define, all of its methods.

The term **implementing an interface** is used (opposed to the term "inheriting from") to describe the process of creating a class based on an interface. The interface simply describes what a class should do. The class implementing the interface must define how to accomplish the behaviors.

The syntax to implement an interface is the same as that to derive a class:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

public interface IShape {

void Draw();

}

class Circle : IShape {

public void Draw() {

Console.WriteLine("Circle Draw");

}

}

static void Main(string[] args)

{

IShape c = new Circle();

c.Draw();

}

}

}

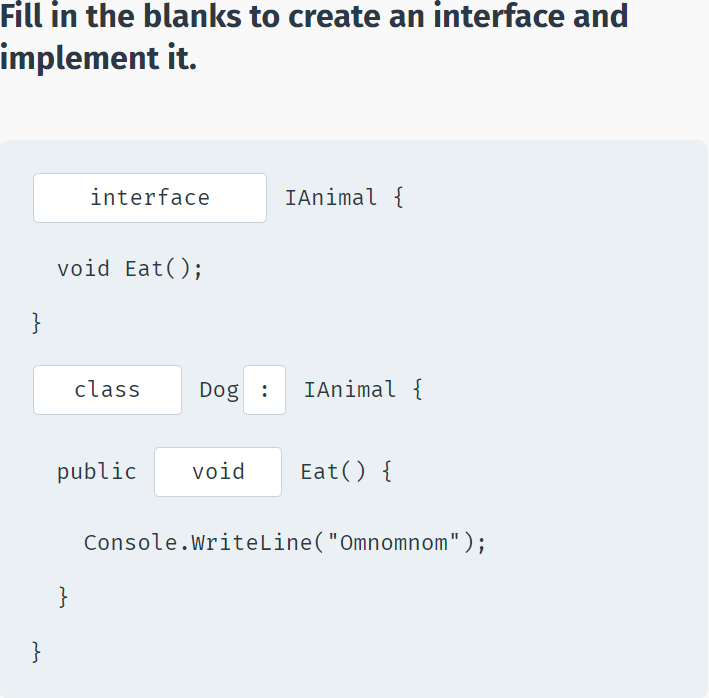
Note, that the **override** keyword is not needed when you implement an interface.

But why use interfaces rather than abstract classes?

A class can inherit from just one base class, but it can implement **multiple** interfaces!

Therefore, by using interfaces you can include behavior from multiple sources in a class.

To implement multiple interfaces, use a comma separated list of interfaces when creating the class: **class A: IShape, IAnimal, etc**.



## Nested Classes

**Nested Classes**

C# supports **nested** classes: a class that is a member of another class.

**For example:**

class Car {

string name;

public Car(string nm) {

name = nm;

Motor m = new Motor();

}

public class Motor {

// some code

}

}

The **Motor** class is nested in the **Car** class and can be used similar to other members of the class.

A nested class acts as a member of the class, so it can have the same access modifiers as other members (public, private, protected).

Just as in real life, objects can contain other objects. For example, a car, which has its own attributes (color, brand, etc.) contains a motor, which as a separate object, has its own attributes (volume, horsepower, etc.). Here, the Car class can have a nested Motor class as one of its members.



## Namespaces

**Namespaces**

When you create a blank project, it has the following structure:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn {

class Program {

static void Main(string[] args) {

}

}

}

Note, that our whole program is inside a **namespace**. So, what are namespaces?

Namespaces declare a scope that contains a set of related objects. You can use a namespace to organize code elements. You can define your own namespaces and use them in your program.

The **using** keyword states that the program is using a given namespace.

For example, we are using the **System** namespace in our programs, which is where the class **Console** is defined:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hi");

}

}

}

Without the **using** statement, we would have to specify the namespace wherever it is used:

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

System.Console.WriteLine("Hi");

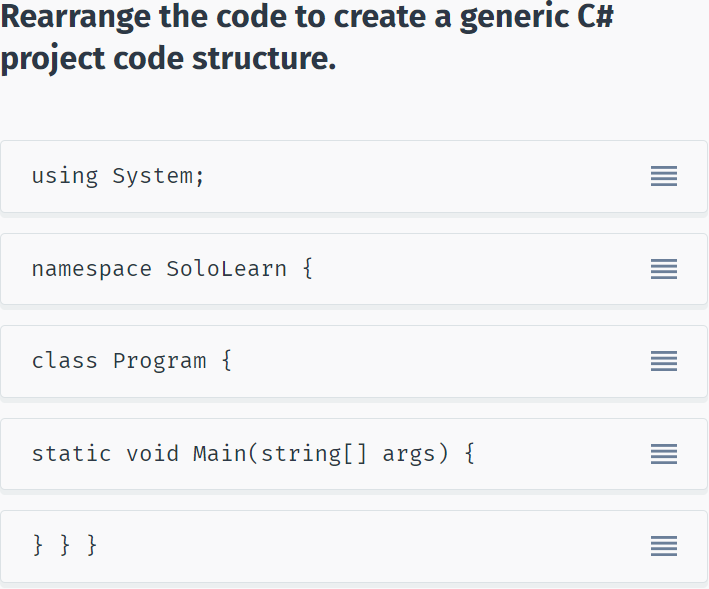
}

}

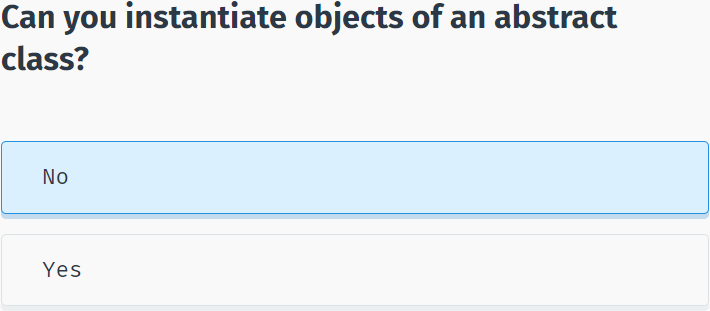
}

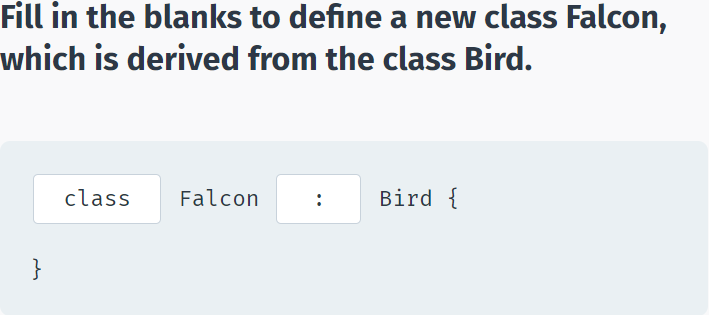
The .NET Framework uses namespaces to organize its many classes. **System** is one example of a .NET Framework namespace.

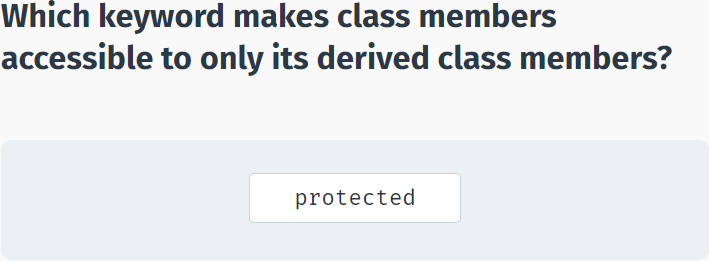
Declaring your own namespaces can help you group your class and method names in larger programming projects.

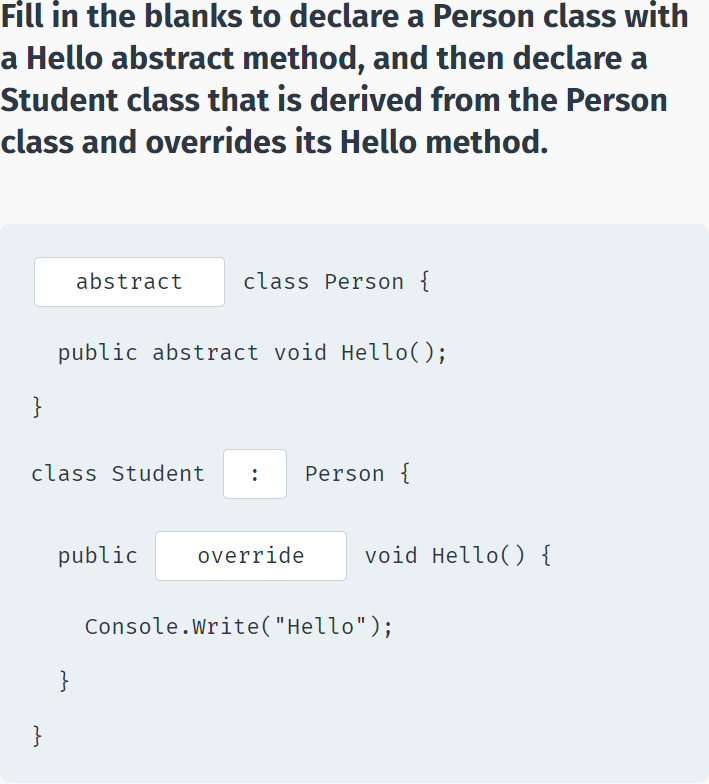


## Module 7 Quiz

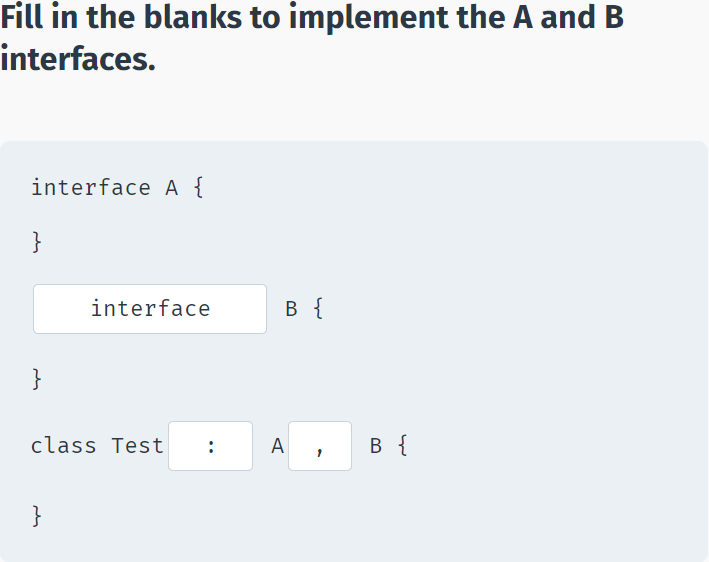














# Structs, Enums, Exceptions & Files

## Structs

**Structs**

A **struct** type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory. The following example shows a simple struct declaration:

struct Book {

public string title;

public double price;

public string author;

}

Structs share most of the same syntax as classes, but are more limited than classes.

Unlike classes, structs can be instantiated without using a new operator.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

struct Book {

public string title;

public double price;

public string author;

}

static void Main(string[] args)

{

Book b;

b.title = "Test";

b.price = 5.99;

b.author = "David";

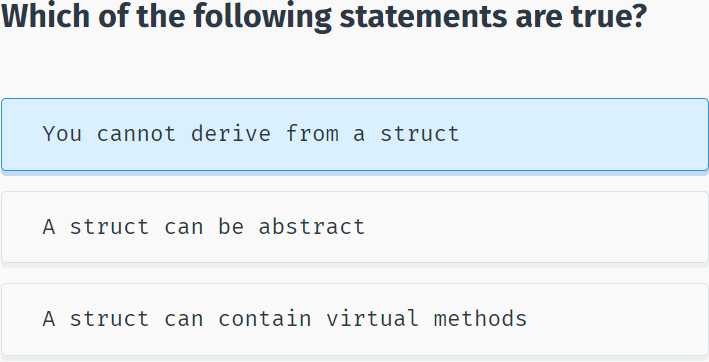
Console.WriteLine(b.title);

}

}

}

Structs do **not** support inheritance and cannot contain virtual methods.



Structs can contain methods, properties, indexers, and so on. Structs cannot contain default constructors (a constructor without parameters), but they can have constructors that take parameters. In that case the new keyword is used to instantiate a struct object, similar to class objects.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

struct Point {

public int x;

public int y;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

}

static void Main(string[] args)

{

Point p = new Point(10, 15);

Console.WriteLine(p.x);

}

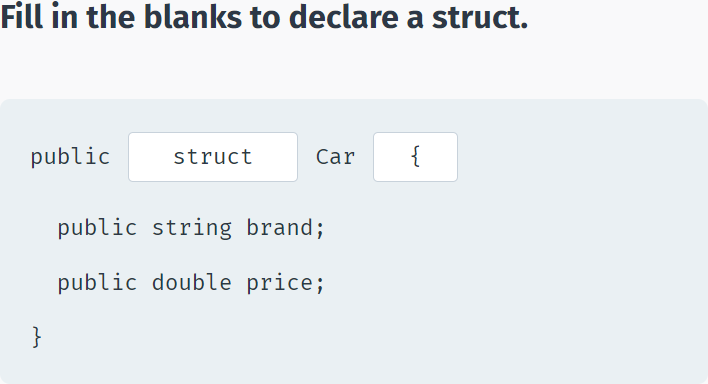
}

}

**Structs vs Classes**

In general, classes are used to model more complex behavior, or data, that is intended to be modified after a class object is created. Structs are best suited for small data structures that contain primarily data that is not intended to be modified after the struct is created. Consider defining a struct instead of a class if you are trying to represent a simple set of data.

All standard C# types (int, double, bool, char, etc.) are actually structs.



## Enums

**Enums**

The **enum** keyword is used to declare an enumeration: a type that consists of a set of named constants called the enumerator list.

By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.

For example, in the following enumeration, Sun is 0, Mon is 1, Tue is 2, and so on:

enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

You can also assign your own enumerator values:

enum Days {Sun, Mon, Tue=4, Wed, Thu, Fri, Sat};

In the example above, the enumeration will start from 0, then Mon is 1, Tue is 4, Wed is 5, and so on. The value of the next item in an Enum is one increment of the previous value.

Note that the values are comma separated.

You can refer to the values in the Enum with the dot syntax.

In order to assign Enum values to int variables, you have to specify the type in parentheses:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

static void Main(string[] args)

{

int x = (int)Days.Tue;

Console.WriteLine(x);

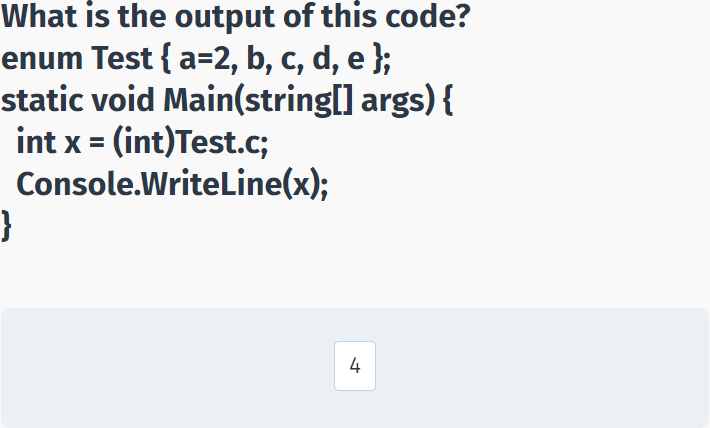
}

}

}

Basically, Enums define variables that represent members of a fixed set.

Some sample Enum uses include month names, days of the week, cards in a deck, etc.



Enums are often used with **switch** statements.

**For example:**

enum TrafficLights { Green, Red, Yellow };

static void Main(string[] args)

{

TrafficLights x = TrafficLights.Red;

switch (x) {

case TrafficLights.Green:

Console.WriteLine("Go!");

break;

case TrafficLights.Red:

Console.WriteLine("Stop!");

break;

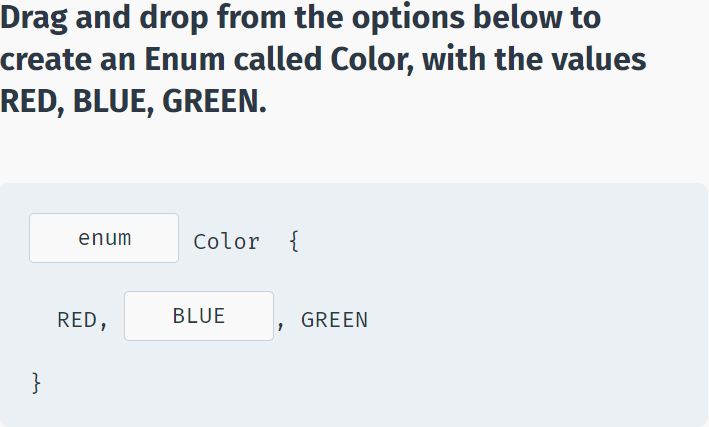
case TrafficLights.Yellow:

Console.WriteLine("Caution!");

break;

}

}



## Exception Handling

**Exceptions**

An **exception** is a problem that occurs during program execution. Exceptions cause abnormal termination of the program.

An exception can occur for many different reasons. Some examples:

- A user has entered invalid data.

- A file that needs to be opened cannot be found.

- A network connection has been lost in the middle of communications.

- Insufficient memory and other issues related to physical resources.

For example, the following code will produce an exception when run because we request an index which does not exist:

static void Main(string[] args)

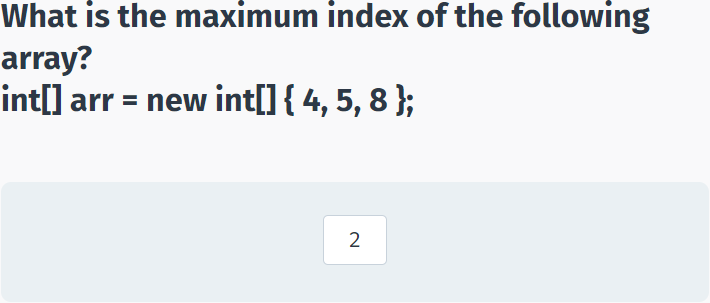
{

int[] arr = new int[] { 4, 5, 8 };

Console.Write(arr[8]);

}

As you can see, exceptions are caused by user error, programmer error, or physical resource issues. However, a well-written program should handle all possible exceptions.



**Handling Exceptions**

C# provides a flexible mechanism called the **try-catch** statement to handle exceptions so that a program won't crash when an error occurs.

The try and catch blocks are used similar to:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

try {

int[] arr = new int[] { 4, 5, 8 };

Console.Write(arr[8]);

}

catch(Exception e) {

Console.WriteLine("An error occurred");

}

}

}

}

The code that might generate an exception is placed in the **try** block. If an exception occurs, the catch blocks is executed without stopping the program.

The type of exception you want to catch appears in parentheses following the keyword **catch**.

We use the general **Exception** type to handle all kinds of exceptions. We can also use the exception object e to access the exception details, such as the original error message (**e.Message**):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

try {

int[] arr = new int[] { 4, 5, 8 };

Console.Write(arr[8]);

}

catch(Exception e) {

Console.WriteLine(e.Message);

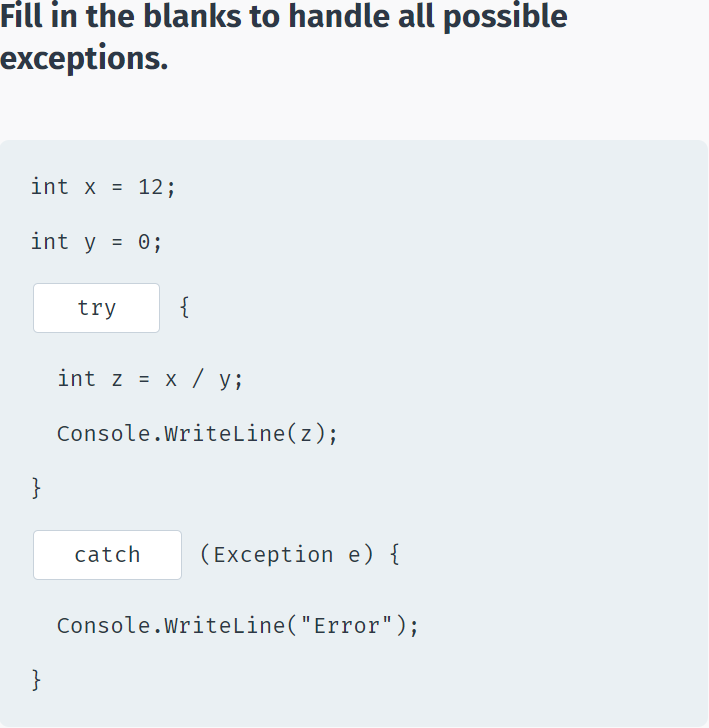
}

}

}

}

You can also catch and handle different exceptions separately. Tap next to learn more!



**Handling Multiple Exceptions**

A single **try** block can contain multiple **catch** blocks that handle different exceptions separately.

Exception handling is particularly useful when dealing with user input.

For example, for a program that requests user input of two numbers and then outputs their quotient, be sure that you handle division by zero, in case your user enters 0 as the second number.

int x, y;

try {

x = Convert.ToInt32(Console.Read());

y = Convert.ToInt32(Console.Read());

Console.WriteLine(x / y);

}

catch (DivideByZeroException e) {

Console.WriteLine("Cannot divide by 0");

}

catch(Exception e) {

Console.WriteLine("An error occurred");

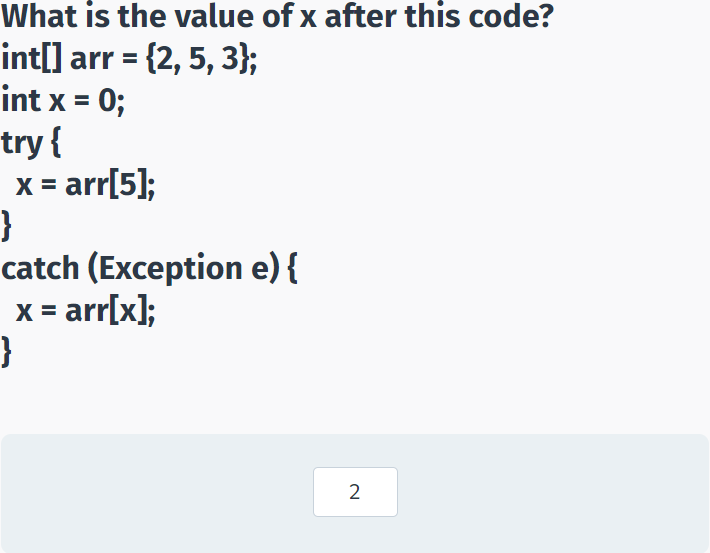
}

The above code handles the **DivideByZeroException** separately. The last **catch** handles all the other exceptions that might occur. If multiple exceptions are handled, the **Exception** type must be defined last.

Now, if the user enters 0 for the second number, "Cannot divide by 0" will be displayed.

If, for example, the user enters non-integer values, "An error occurred" will be displayed.

The following exception types are some of the most commonly used: FileNotFoundException, FormatException, IndexOutOfRangeException, InvalidOperationException, OutOfMemoryException.



**finally**

An optional finally block can be used after the catch blocks. The finally block is used to execute a given set of statements, whether an exception is thrown or not.

**For example:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

int result=0;

int num1 = 8;

int num2 = 4;

try {

result = num1 / num2;

}

catch (DivideByZeroException e) {

Console.WriteLine("Error");

}

finally {

Console.WriteLine(result);

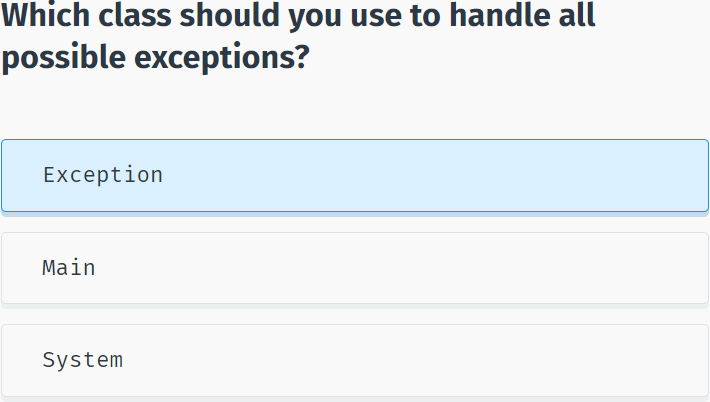
}

}

}

}

The **finally** block can be used, for example, when you work with files or other resources. These should be closed or released in the **finally** block, whether an exception is raised or not.



## Working with Files

**Writing to Files**

The **System.IO** namespace has various classes that are used for performing numerous operations with files, such as creating and deleting files, reading from or writing to a file, closing a file, and more.

The **File** class is one of them.

**For example:**

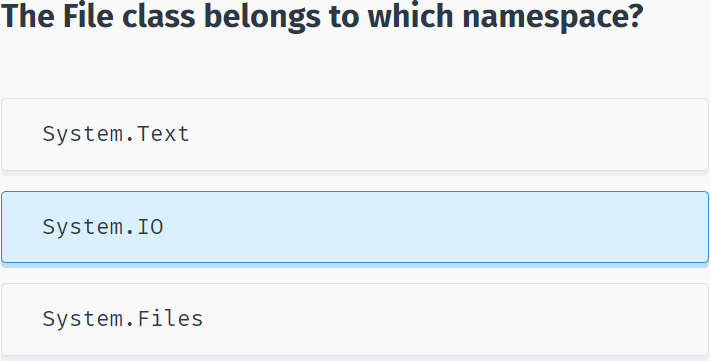
string str = "Some text";

File.WriteAllText("test.txt", str);

The **WriteAllText()** method creates a file with the specified path and writes the content to it. If the file already exists, it is overwritten.

The code above creates a file **test.txt** and writes the contents of the **str** string into it.

To use the **File** class you need to use the System.IO namespace: **using System.IO**;



**Reading from Files**

You can read the content of a file using the ReadAllText method of the File class:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.IO;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

string str = "Some text";

File.WriteAllText("test.txt", str);

string txt = File.ReadAllText("test.txt");

Console.WriteLine(txt);

}

}

}

This will output the content of the test.txt file.

The following methods are available in the File class:

**AppendAllText()** - appends text to the end of the file.

**Create()** - creates a file in the specified location.

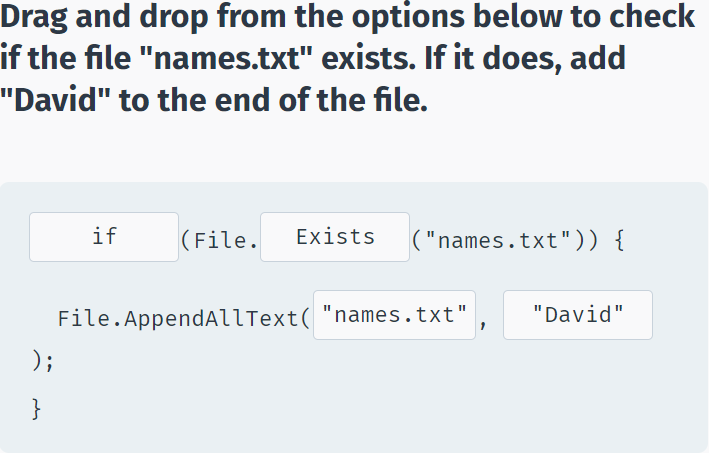
**Delete()** - deletes the specified file.

**Exists()** - determines whether the specified file exists.

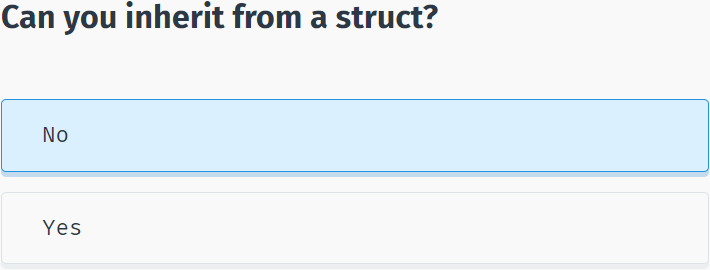
**Copy()** - copies a file to a new location.

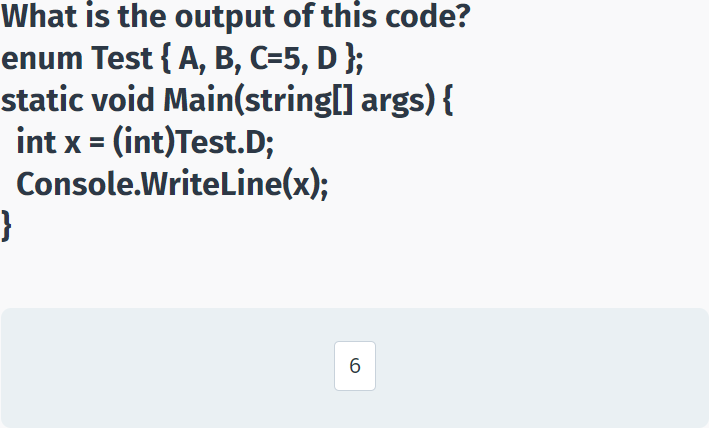
**Move()** - moves a specified file to a new location

All methods automatically close the file after performing the operation.

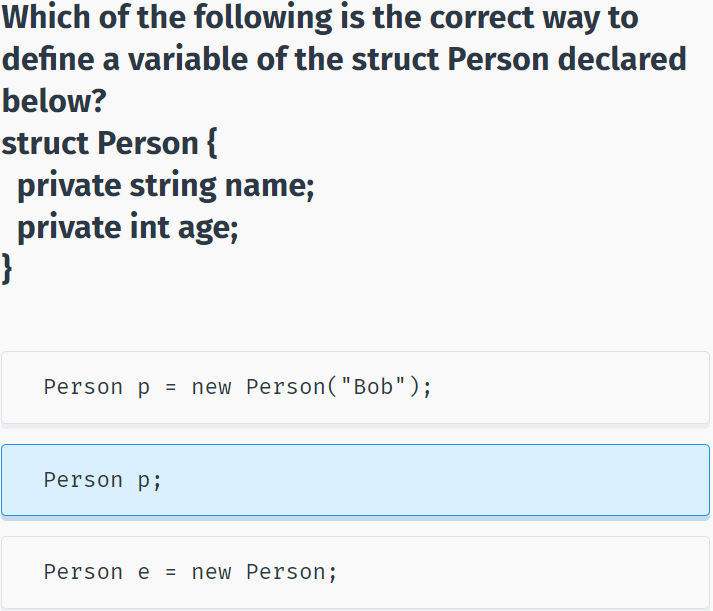


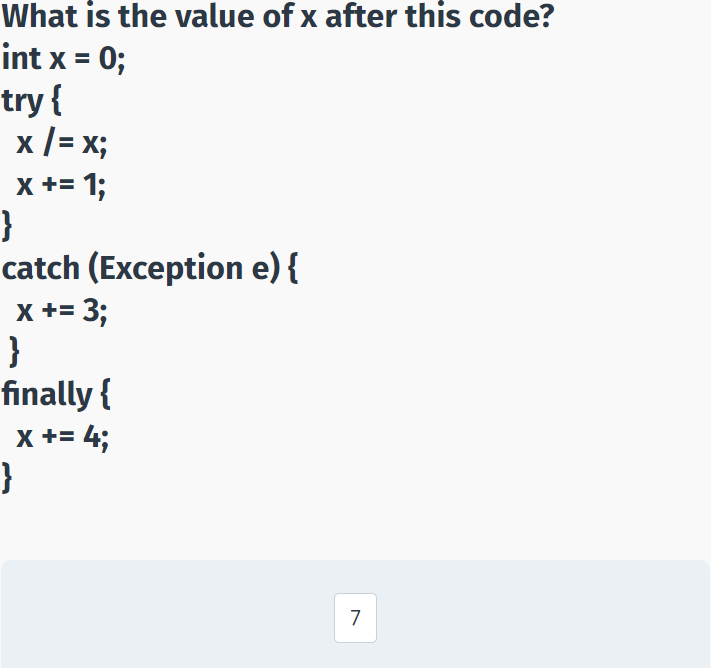
## Module 8 Quiz













# Generics

## Generic Methods

**Generics**

**Generics** allow the reuse of code across different types.

For example, let's declare a method that swaps the values of its two parameters:

static void Swap(ref int a, ref int b) {

int temp = a;

a = b;

b = temp;

}

Our **Swap** method will work only for integer parameters. If we want to use it for other types, for example, doubles or strings, we have to overload it for all the types we want to use it with. Besides a lot of code repetition, it becomes harder to manage the code because changes in one method mean changes to all of the overloaded methods.

Generics provide a flexible mechanism to define a generic type.

static void Swap<T>(ref T a, ref T b) {

T temp = a;

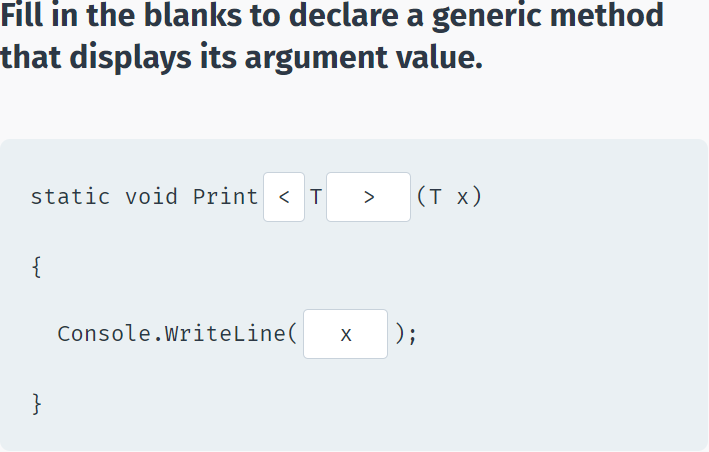
a = b;

b = temp;

}

In the code above, **T** is the name of our generic type. We can name it anything we want, but **T** is a commonly used name. Our Swap method now takes two parameters of type **T**. We also use the **T** type for our **temp** variable that is used to swap the values.

Note the brackets in the syntax **<T>**, which are used to define a generic type.



Now, we can use our Swap method with different types, as in:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

static void Swap<T>(ref T a, ref T b) {

T temp = a;

a = b;

b = temp;

}

static void Main(string[] args)

{

int a = 4, b = 9;

Swap<int>(ref a, ref b);

Console.WriteLine(a+" "+b);

string x = "Hello";

string y = "World";

Swap<string>(ref x, ref y);

Console.WriteLine(x+" "+y);

}

}

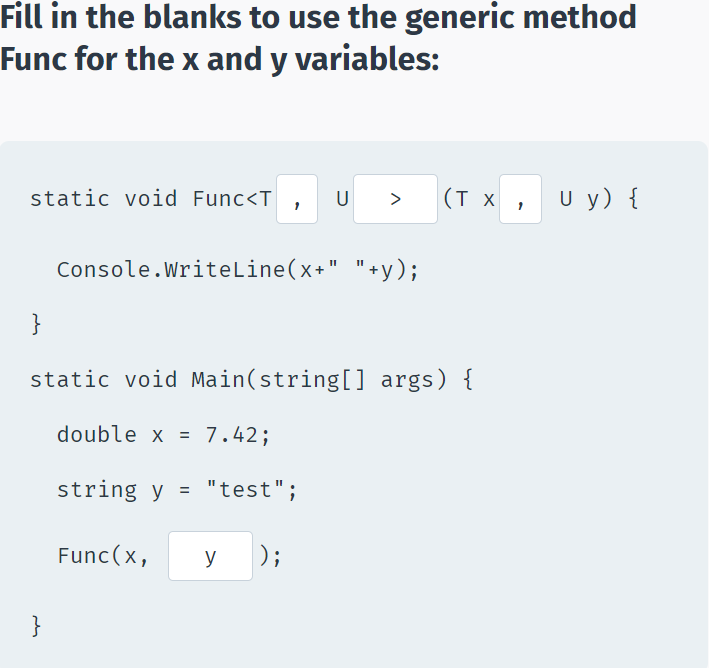
}

When calling a generic method, we need to specify the type it will work with by using brackets. So, when **Swap<int>** is called, the **T** type is replaced by **int**. For **Swap<string>**, **T** is replaced by **string**.

If you omit specifying the type when calling a generic method, the compiler will use the type based on the arguments passed to the method.

Multiple generic parameters can be used with a single method.

For example: **Func<T, U>** takes two different generic types.



## Generic Classes

**Generic Classes**

Generic types can also be used with classes.

The most common use for generic classes is with collections of items, where operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored. One type of collection is called a stack. Items are "pushed", or added to the collection, and "popped", or removed from the collection. A stack is sometimes called a Last In First Out (LIFO) data structure.

**For example:**

class Stack<T> {

int index=0;

T[] innerArray = new T[100];

public void Push(T item) {

innerArray[index++] = item;

}

public T Pop() {

return innerArray[--index];

}

public T Get(int k) { return innerArray[k]; }

}

The generic class stores elements in an array. As you can see, the generic type **T** is used as the type of the array, the parameter type for the **Push** method, and the return type for the **Pop** and **Get** methods.

**Now we can create objects of our generic class:**

Stack<int> intStack = new Stack<int>();

Stack<string> strStack = new Stack<string>();

Stack<Person> PersonStack = new Stack<Person>();

We can also use the generic class with custom types, such as the custom defined **Person** type.

In a generic class we do not need to define the generic type for its methods, because the generic type is already defined on the class level.



Generic class methods are called the same as for any other object:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

class Stack<T> {

int index=0;

T[] innerArray = new T[100];

public void Push(T item) {

innerArray[index++] = item;

}

public T Pop() {

return innerArray[--index];

}

public T Get(int k) { return innerArray[k]; }

}

static void Main(string[] args)

{

Stack<int> intStack = new Stack<int>();

intStack.Push(3);

intStack.Push(6);

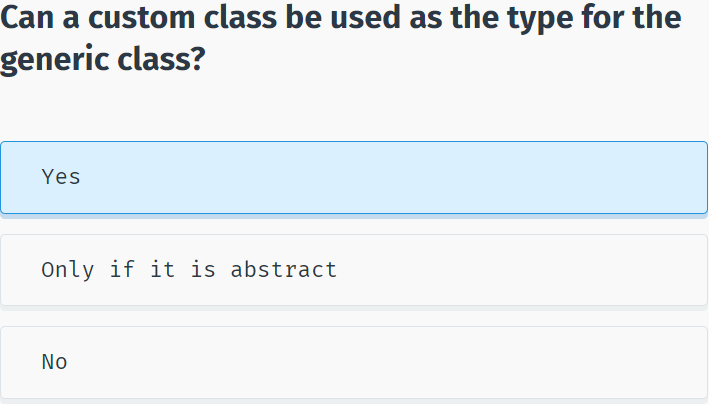
intStack.Push(7);

Console.WriteLine(intStack.Get(1));

}

}

}



## Collections

**C# Collections**

A **collection** is used to group related objects. Unlike an array, it is **dynamic** and can also group objects. A collection can grow and shrink to accommodate any number of objects. Collection classes are organized into **namespaces** and contain built in methods for processing elements within the collection.

A collection **organizes** related data in a computer so that it can be used efficiently.

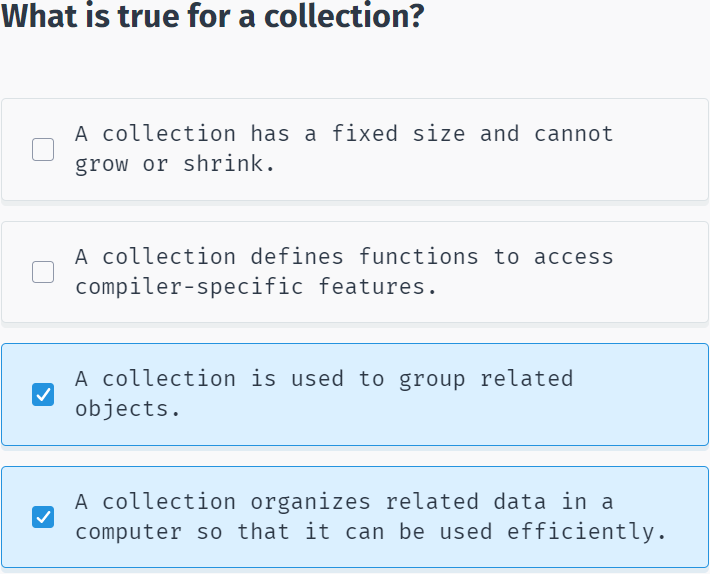
Different kinds of collections are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, **Dictionaries** are used to represent connections on social websites (such as Twitter, Facebook), queues can be used to create task schedulers, **HashSets** are used in searching algorithms, etc.

A collection typically includes methods to **add, remove**, and **count** objects. The **for** statement and the **foreach** statement are used to **iterate** through collections. Since a collection is a class you must first declare an instance of the class before you can add elements to that collection.

**For example:**

List<int> li = new List<int>();

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change.



**Generic Collections**

Generic collections are the preferred type to use as long as every element in the collection is of the same data type. Only desired data types can be added to a generic collection and this is enforced by using strong typing which reduces the possibility of errors.

The .NET Framework provides a number of generic collection classes, useful for storing and manipulating data.

The **System.Collections.Generic** namespace includes the following generic collections:

**- List<T>**

**- Dictionary<TKey, TValue>**

**- SortedList<TKey, TValue>**

**- Stack<T>**

**- Queue<T>**

**- Hashset<T>**

To access a generic collection in your code, you will need to include the statement: **using Systems.Collections.Generic**;

**Non-Generic Collections**

Non-generic collections can store items that are of type Object. Since an Object data type can refer to any data type, you run the risk of unexpected outcomes. Non-generic collections may also be slower to access as well as execute.

The **System.Collections** namespace includes the following non-generic collections:

**- ArrayList**

**- SortedList**

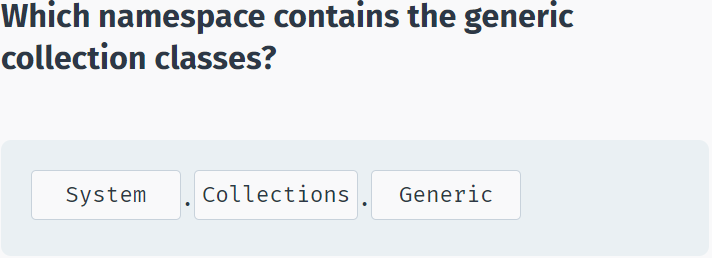
**- Stack**

**- Queue**

**- Hashtable**

**- BitArray**

Because non-generic collections are error prone and less performant, it is recommended to always use generic collections from the System.Collections.Generic namespace if available and to avoid using legacy collections from the System.Collections namespace.



## Lists and BitArray

**List<T>**

A **list** is similar to an array, but the elements in a list can be inserted and removed **dynamically**.

The C# generic collection **List<T>** class requires all elements be of the same type **T**.

**List<T>** properties and methods include:

**Count** A property that gets the number of elements contained in the list.

**Item[int i]** Gets or sets the element in the list at the index i. Item is the indexer and is not required when accessing an element. You only need to use the brackets [] and the index value inside the brackets.

**Add(T t)** Adds an element t to the end of the list.

**RemoveAt(int index)** Removes the element at the specified position (index) from the list.

**Sort()** Sorts elements in the list.

**Now let's try List<T>:**

using System;

using System.Collections.Generic;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

List<int> li = new List<int>();

li.Add(59);

li.Add(72);

li.Add(95);

li.Add(5);

li.Add(9);

li.RemoveAt(1); // remove 72

Console.Write("\nList: ");

for (int x = 0; x < li.Count; x++)

Console.Write(li[x] + " "); // 59 95 5 9

li.Sort();

Console.Write("\nSorted: ");

for (int x = 0; x < li.Count; x++)

Console.Write(li[x] + " "); // 5 9 59 95

}

}

}

Additional List<T> properties and methods are listed below. Try them out by adding them to the List<T> example code above.

Capacity - A property that gets the number of elements the list can hold before needing to be resized.

**Clear()** - Removes all the elements from the list.

**TrimExcess()** - Sets the capacity to the actual number of elements in the list. This is useful when trying to reduce memory overhead.

**AddRange(IEnumerable coll)** - Adds the elements of collection coll with elements of the same type as List<T> to the end of the list. IEnumerable is the collections interface that supports simple iteration over the collection.

**Insert(int i, T t)** - Inserts an element t at a specific index i in the list.

**InsertRange(int i, IEnumerable coll)** - Inserts the elements of a collection coll at a specified index i in the list. IEnumerable is the collections interface that supports simple iteration over the collection.

**Remove(T t)** - Removes the first occurrence of the object t from the list.

**RemoveRange(int i, int count)** - Removes a specified number of elements count from the list starting at a specified index i.

**Contains(T t)** - Returns true if the specified element t is present in the list.

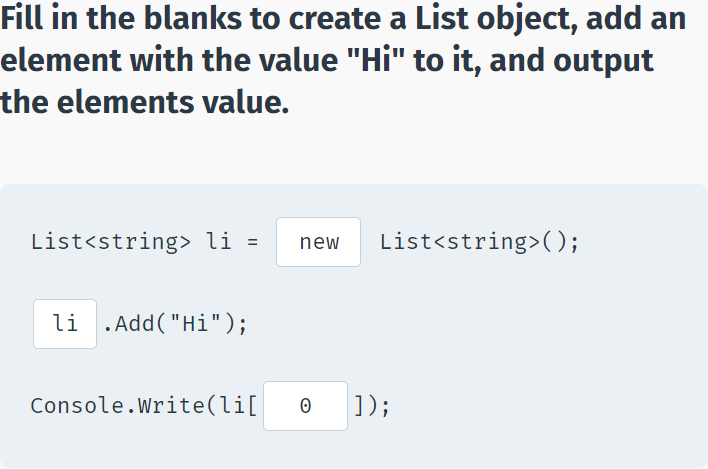
**IndexOf(T t)** - Returns the index of the first occurrence of the element t in the list.

**Reverse()** - Reverses the order of the elements in the list.

**ToArray()** - Copies the elements of the list into a new array.

Tap **Try It Yourself** and modify the code to insert elements using the for loop.

Remember, you need to include the statement: **using Systems.Collections.Generic**; to use List<T>.



**SortedList<K, V>**

A **sorted list** is a collection of **key/value pairs** that are sorted by key. A key can be used to access its corresponding value in the sorted list.

The C# generic collection **SortedList<K, V>** class requires all element key/value pairs to be of the same type **K, V**. Duplicate keys are **not permitted**, which ensures that every key/value pair is unique.

**SortedList<K, V>** properties include:

**Count** - Gets the number of key/value pairs contained in the sorted list.

**Item[K key]** - Gets or sets the value associated the specified key contained in the sorted list. Item is the indexer and is not required when accessing an element. You only need to use the brackets [] and the key, value.

**Keys** - Gets a sorted and indexed collection containing only the keys in the sorted list.

**SortedList<K, V>** methods include:

**Add(K key, V value)** - Adds an element with a specific key, value pair into the sorted list.

**Remove(K key)** - Removes the element with the specific key, value pair associated with the specified key from the sorted list.

**Now let's try SortedList<K, V>:**

using System;

using System.Collections.Generic;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

SortedList<string, int> sl = new SortedList<string, int>();

sl.Add("Solo", 59);

sl.Add("A", 95);

sl.Add("Learn", 72);

sl.Remove("A");

Console.WriteLine("Sorted List: ");

foreach (string s in sl.Keys)

Console.WriteLine(s + ": " + sl[s]); // Learn: 72 Solo: 59

Console.WriteLine("\nCount: " + sl.Count); // 2

}

}

}

Here are additional **SortedList<K, V>** properties and methods:

**Values** - Gets a sorted and indexed collection of the values in the sorted list.

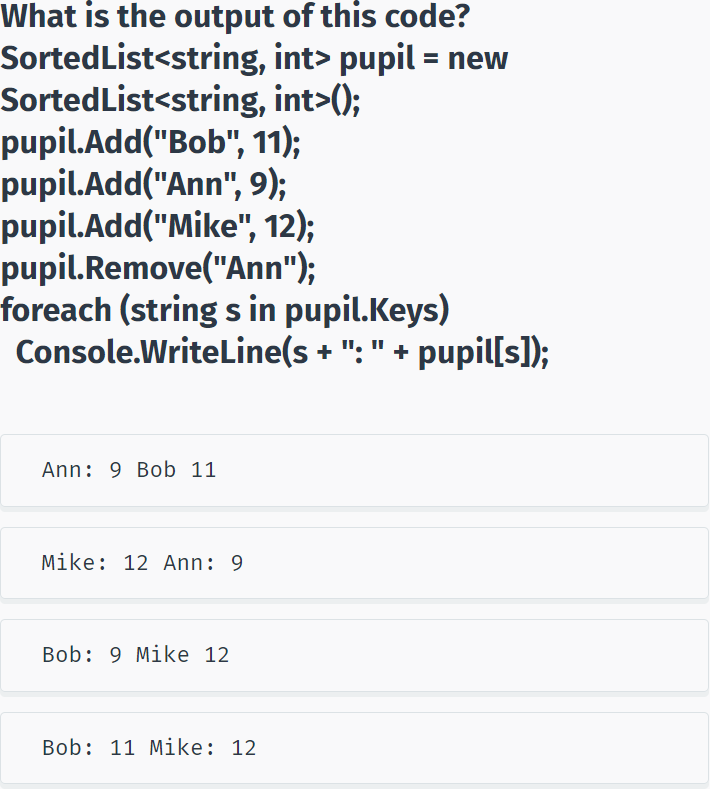
**Clear()** - Removes all the elements from the sorted list.

**ContainsKey(K key)** - Returns true when the specified key is present in the sorted list.

**ContainsValue(V value)** - Returns true when a specified value is present in the sorted list.

**IndexOfKey(K key)** - Returns the index of the specified key within the sorted list.

**IndexOfValue(V value)** - Returns the index of the specified value within the sorted list.



**BitArray**

A bit array is a **collection of bits**. The value of a bit can be either **0** (off/false) or **1** (on/true).

Bit arrays compactly store bits. Most commonly, they are used to represent a simple group of boolean flags or an ordered sequence of boolean values.

**BitArray** properties include:

**Count** - Gets the number of bits in the bit array.

**IsReadOnly** - Gets a value indicating if the bit array is read only or not.

**BitArray** methods include:

**Get(int i)** - Gets the value of the bit at a specified position i in the bit array.

**Set(int i, bool value)** - Sets the bit at a specified position i to a specified value in the bit array.

**SetAll(bool value)** - Sets all the bits to a specified value in the bit array.

**And(BitArray ba)** - Performs the bitwise AND operation on the elements of the bit array object with a specified bit array ba.

**Or(BitArray ba)** - Performs the bitwise OR operation on the elements of the bit array and the specified bit array ba.

**Not()** - Inverts the bit values of the bit array.

**Xor(BitArray ba)** - Performs the bitwise XOR operation on the elements of the current bit array object and the elements in the specified bit array ba.

**This example demonstrates some properties and methods of the BitArray class:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SoloLearn

{

class Program

{

// Printing BitArray

public static void PrintBarr(string name, BitArray ba)

{

Console.Write(name + " : ");

for (int x = 0; x < ba.Length; x++)

Console.Write(ba.Get(x) + " ");

Console.WriteLine();

}

public static void Main(string[] args)

{

BitArray ba1 = new BitArray(4);

BitArray ba2 = new BitArray(4);

ba1.SetAll(true);

ba2.SetAll(false);

ba1.Set(2, false);

ba2.Set(3, true);

PrintBarr("ba1", ba1);

PrintBarr("ba2", ba2);

Console.WriteLine();

PrintBarr("ba1 AND ba2", ba1.And(ba2));

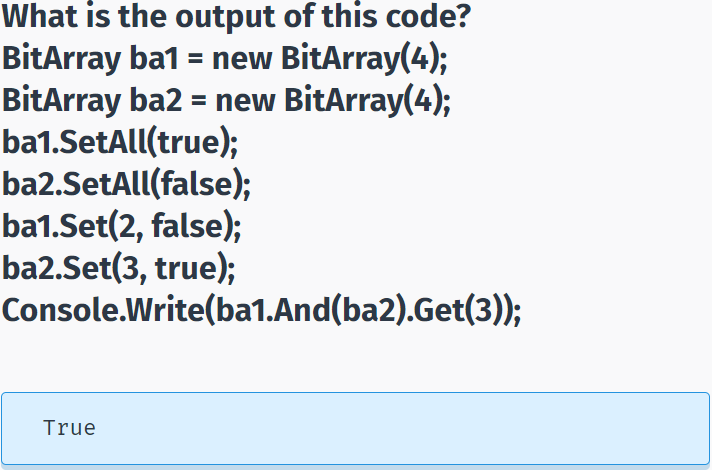
PrintBarr(" NOT ba2", ba2.Not());

}

}

}

For example, BitArrays can be used in image processing to store the individual bits of a gray-scale image.



## Stack & Queue

**Stack<T>**

A **stack** is a **Last In, First Out (LIFO)** collection of elements where the last element that goes into the stack will be the first element that comes out.

Inserting an element onto a stack is called **pushing**. Deleting an element from a stack is called **popping**. Pushing and popping can be performed only at the **top** of the stack.

Stacks can be used to create undo-redo functionalities, parsing expressions (infix to postfix/prefix conversion), and much more.

The C# generic collection **Stack<T>** class requires all elements to be of the same type **T**.

**Stack<T>** properties include:

**Count** - Returns the number of elements in the stack.

**Stack<T>** methods include:

**Peek()** - Returns the element at the top of the stack without removing it.

**Pop()** - Returns the element at the top of the stack and removes it from the stack.

**Push(T t)** - Inserts an element t at the top of the stack.

**Now let's try Stack<T>:**

using System;

using System.Collections.Generic;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

Stack<int> s = new Stack<int>();

s.Push(59);

s.Push(72);

s.Push(65);

Console.Write("Stack: ");

foreach (int i in s)

Console.Write(i + " "); // 65 72 59

Console.Write("\nCount: " + s.Count); // 3

Console.Write("\nTop: " + s.Peek()); // 65

Console.Write("\nPop: " + s.Pop()); // 65

Console.Write("\nStack: ");

foreach (int i in s)

Console.Write(i + " "); // 72 59

Console.Write("\nCount: " + s.Count); // 2

}

}

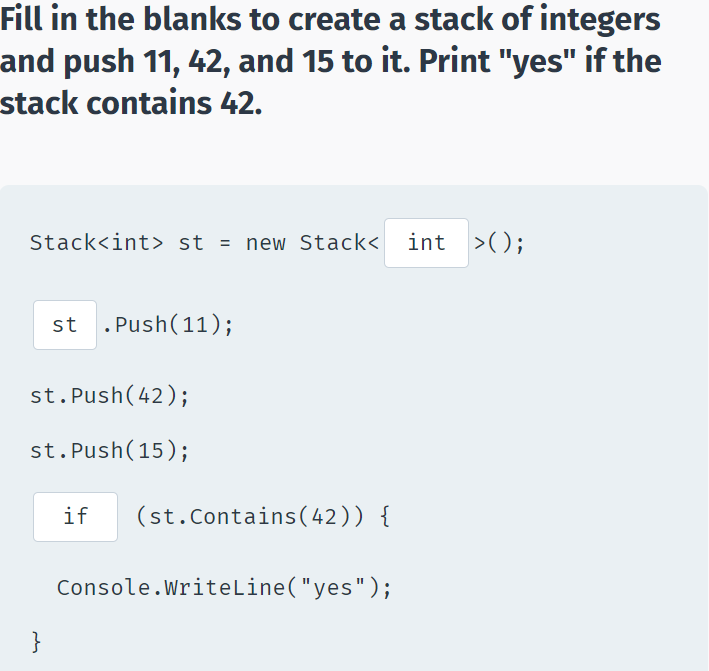
}

Here are additional **Stack<T>** methods:

**Clear()** - Removes all the elements from the stack.

**Contains(T t)** - Returns true when the element t is present in the stack.

**ToArray()** - Copies the stack into a new array.



**Queue<T>**

A **queue** is a **First In, First Out (FIFO)** collection of elements where the first element that goes into a queue is also the first element that comes out.

Inserting an element into a queue is referred to as **Enqueue**. Deleting an element from a queue is referred to as **Dequeue**.

Queues are used whenever we need to manage objects in order starting with the first one in.

Scenarios include printing documents on a printer, call center systems answering people on hold people, and so on.

The C# generic collection **Queue<T>** class requires that all elements be of the same type **T**.

**Queue<T>** properties include:

**Count** - Gets the number of elements in the queue.

And methods include:

**Dequeue()** - Returns the object at the beginning of the queue and also removes it.

**Enqueue(T t)** - Adds the object t to the end of the queue.

**Now let's try Queue<T>:**

using System;

using System.Collections.Generic;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

Queue<int> q = new Queue<int>();

q.Enqueue(5);

q.Enqueue(10);

q.Enqueue(15);

Console.Write("Queue: ");

foreach (int i in q)

Console.Write(i + " "); // 5 10 15

Console.Write("\nCount: " + q.Count); // 3

Console.Write("\nDequeue: " + q.Dequeue()); // 5

Console.Write("\nQueue: ");

foreach (int i in q)

Console.Write(i + " "); // 10 15

Console.Write("\nCount: " + q.Count); // 2

}

}

}

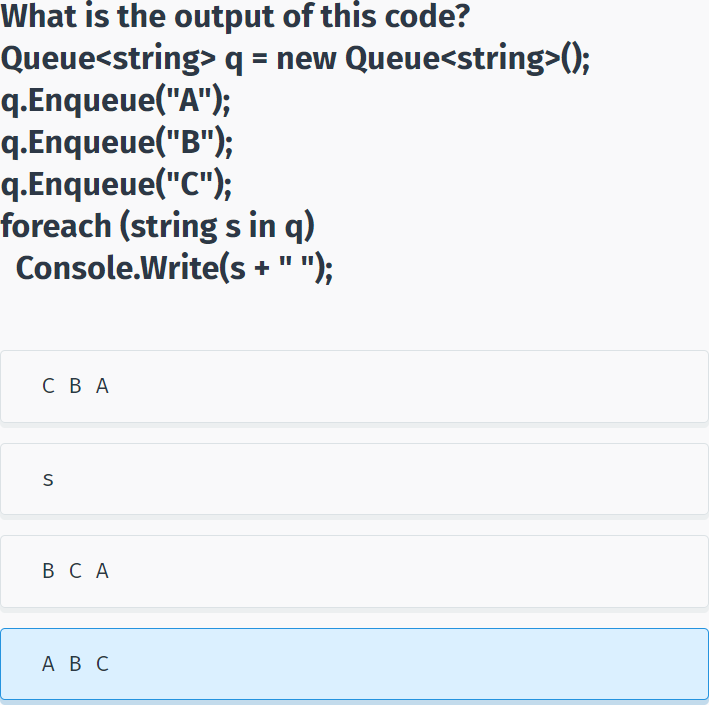
Here are additional **Queue<T>** methods:

**Clear()** - Removes all objects from the queue.

**Contains(T t)** - Returns true when the element t is present in the queue.

**Peek()** - Returns the object at the beginning of the queue without removing it.

**ToArray()** - Copies the queue into a new array.



## Dictionary & HashSet

**Dictionary**<U, V>

A **dictionary** is a collection of unique key/value pairs where a key is used to access the corresponding value. Dictionaries are used in database indexing, cache implementations, and so on.

The C# generic collection **Dictionary<K, V>** class requires all key/value pairs be of the same type **K, V**. Duplicate keys are **not permitted** to ensure that every key/value pair is unique.

**Dictionary<K, V>** properties include:

**Count** - Gets the number of key/value pairs contained in the dictionary.

**Item[K key]** - Gets the value associated with the specified key in the dictionary. Item is the indexer and is not required when accessing an element. You only need to use the brackets [] and key value.

**Keys** - Gets an indexed collection containing only the keys contained in the dictionary.

**Dictionary<K, V>** methods include:

**Add(K key, V value)** - Adds the key, value pair to the dictionary.

**Remove(K key)** - Removes the key/value pair related to the specified key from the dictionary.

**Now let's try Dictionary<K, V>:**

using System;

using System.Collections.Generic;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

Dictionary<string, int> d = new Dictionary<string, int>();

d.Add("Uno", 1);

d.Add("One", 1);

d.Add("Dos", 2);

d.Add("Deux", 2);

d.Remove("One"); // Remove key-value pair One, 1

d.Remove("Dos"); // Remove key-value pair Dos, 2

Console.WriteLine("Dictionary: ");

foreach (string s in d.Keys)

Console.WriteLine(s + ": " + d[s]); // Uno: 1 Deux: 2

Console.WriteLine("\nCount: {0}", d.Count); // 2

}

}

}

In the above example, the dictionary **d** uses strings as it's keys and integers as the values.

Here are the additional **Dictionary<K, V>** properties and methods:

**Values** - Gets an indexed collection containing only the values in the dictionary.

**Clear()** - Removes all the key/value pairs from the dictionary.

**ContainsKey(K key)** - Returns true if the specified key is present in the dictionary.

**ContainsValue(V value)** - Returns true if the specified value is present in the dictionary.



**HashSet<T>**

A **hash set** is a set of unique values where duplicates are not allowed.

C# includes the **HashSet<T>** class in the generic collections namespace. All **HashSet<T>** elements are required to be of the same type **T**.

Hash sets are different from other collections because they are simply a set of values. They do not have index positions and elements cannot be ordered.

The HashSet<T> class provides high-performance set operations. HashSets allow fast lookup, addition, and removal of items, and can be used to implement either dynamic sets of items or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by the last name).

HashSet<T> **properties** include:

**Count** Returns the number of values in the hash set.

And **methods** include:

**Add(T t)** Adds a value (t) to the hash set.

**IsSubsetOf(ICollection c)** Returns true if the hash set is a subset of the specified collection (c).

**Now let's try HashSet<T>:**

using System;

using System.Collections.Generic;

namespace SoloLearn

{

class Program

{

static void Main(string[] args)

{

HashSet<int> hs = new HashSet<int>();

hs.Add(5);

hs.Add(10);

hs.Add(15);

hs.Add(20);

Console.Write("\nHashSet: ");

foreach (int i in hs)

Console.Write(i + " "); // 5 10 15 20 \*elements may be in any order

Console.Write("\nCount: " + hs.Count); // 4

HashSet<int> hs2 = new HashSet<int>();

hs2.Add(15);

hs2.Add(20);

Console.Write("\n{15, 20} is a subset of {5, 10, 15, 20}: " + hs2.IsSubsetOf(hs)); // True

}

}

}

Here are additional HashSet<T> methods:

**Remove(T t)** Removes the value (t) from the hash set.

**Clear()** Removes all the elements form the hash set.

**Contains(T t)** Returns true when a value (t) is present in the hash set.

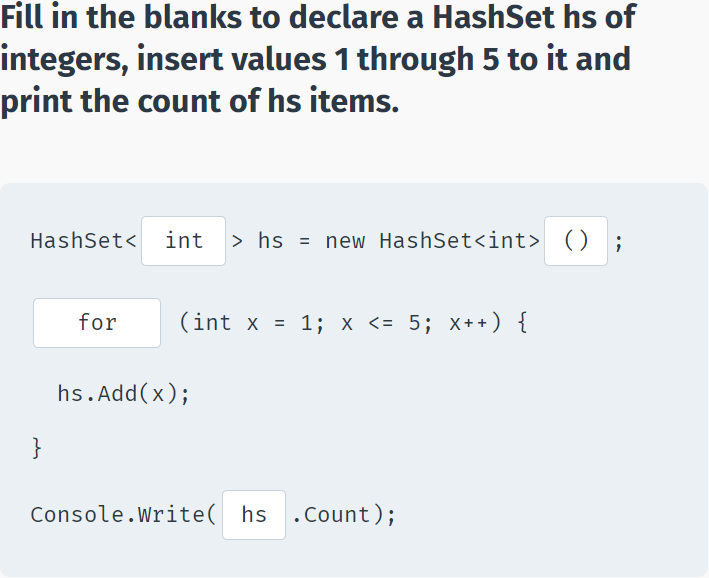
**ToString()** Creates a string from the hash set.

**IsSupersetOf(**ICollection c**)** Returns true if the hash set is a superset of the specified collection.

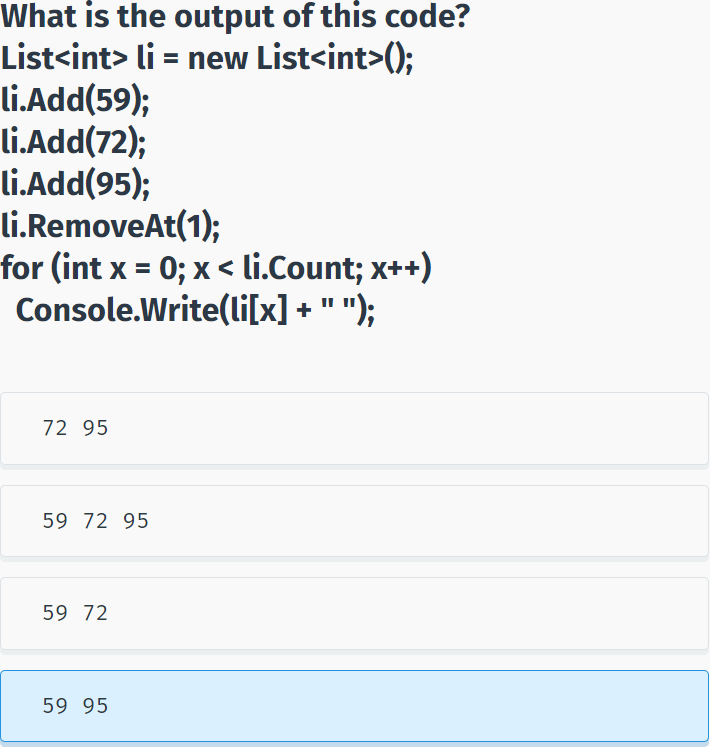
**UnionWith(**ICollection c**)** Applies set union operation on the hash set and the specified collection (c).

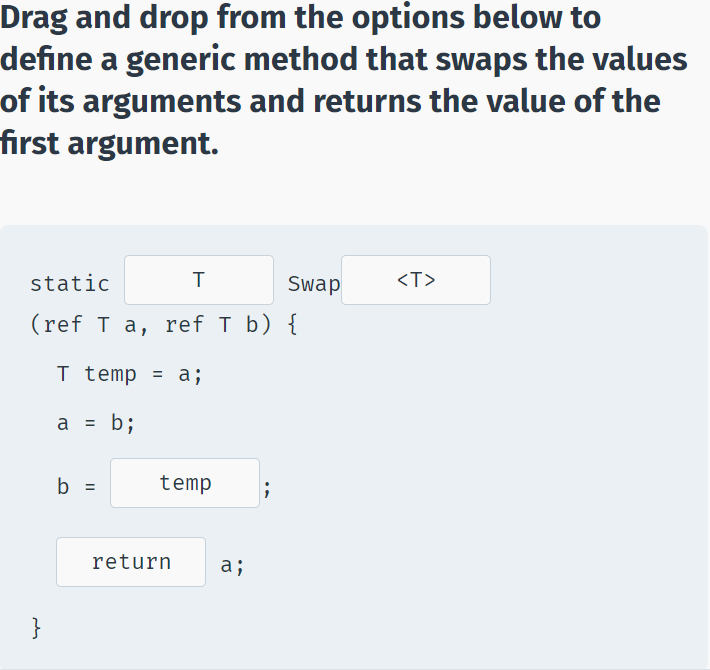
**IntersectWith(**ICollection c**)** Applies set intersection operation on the hash set and the specified collection (c).

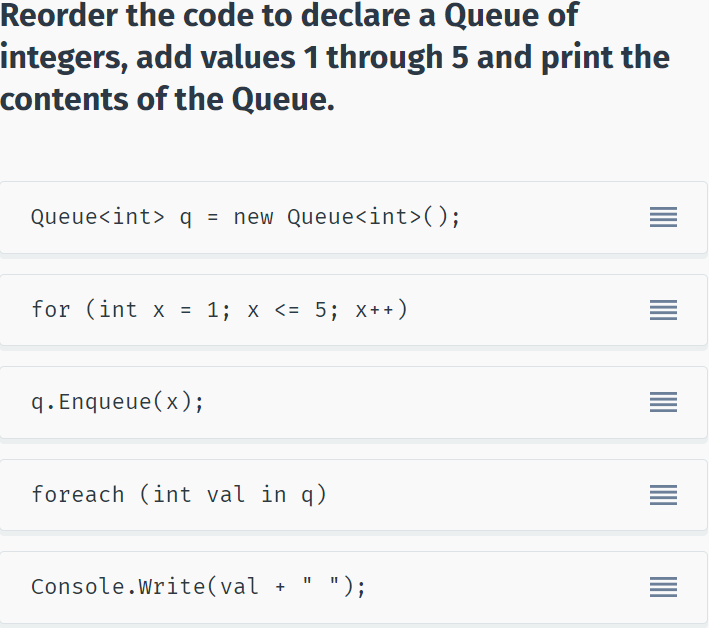
**ExceptWith(**ICollection c**)** Applies set difference operation on the hash set and the specified collection (c).

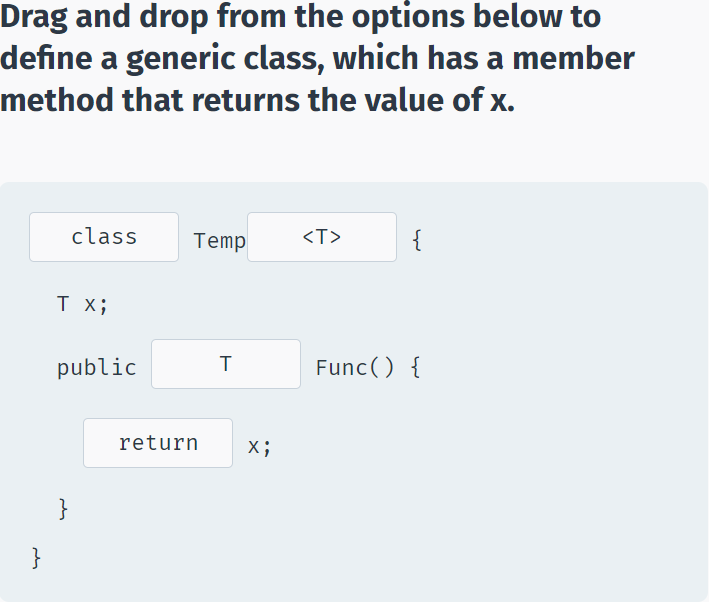


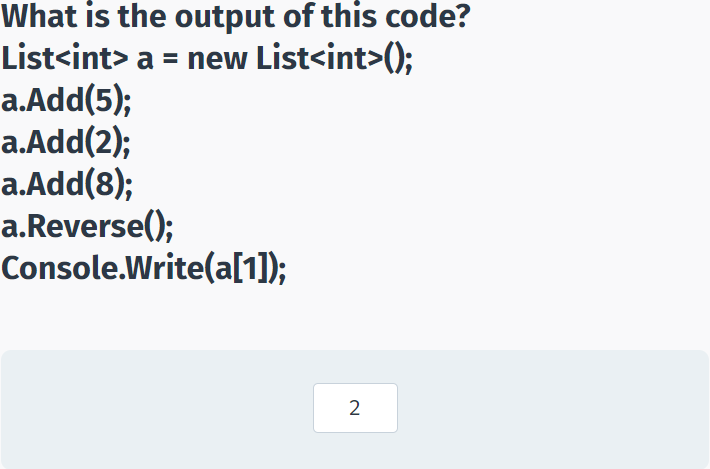
## Module 9 Quiz

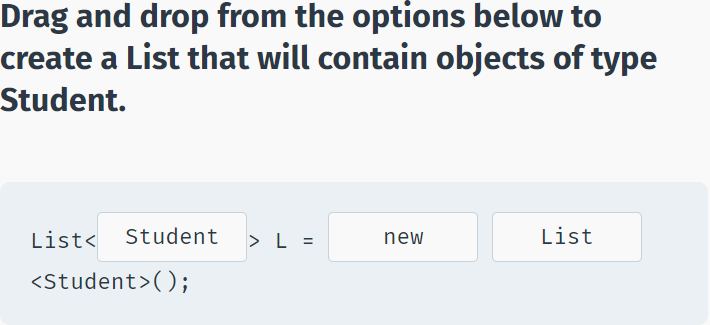


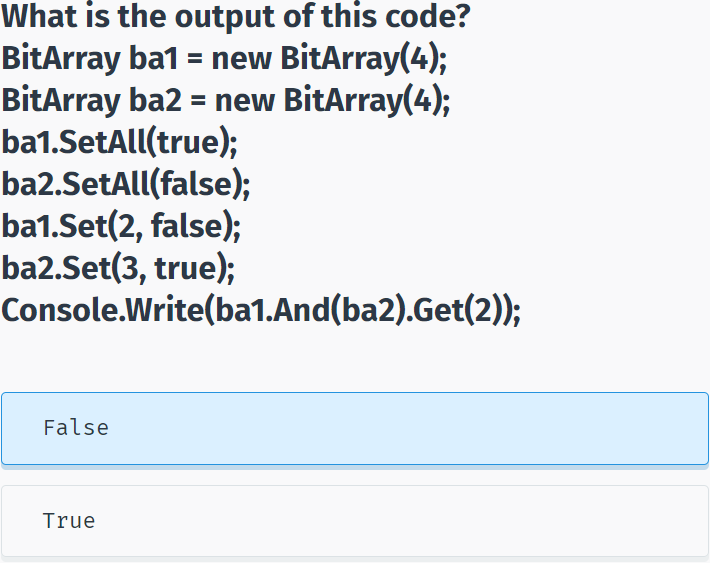


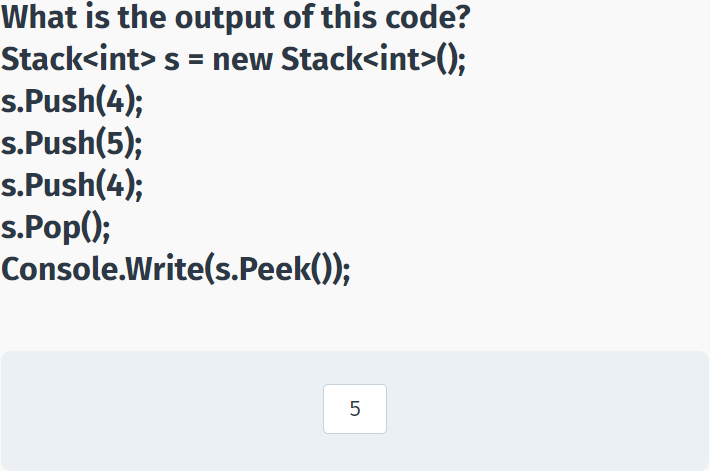


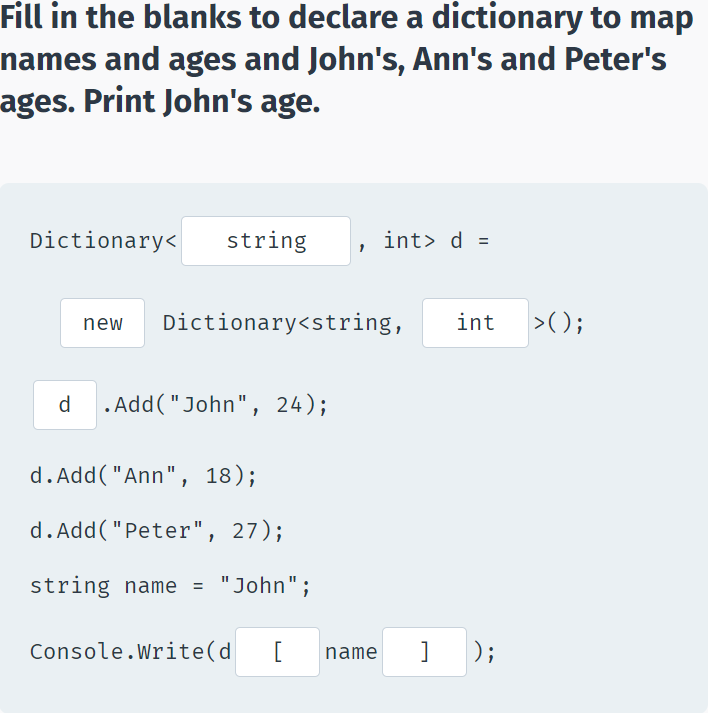












# Dictionary

## Noun

basic concepts : những khái niệm cơ bản

language : ngôn ngữ

applications : những ứng dụng

Web services : dịch vụ web

client-server : máy khách-máy chủ

database : cơ sở dữ liệu

.NET Framework : là một nền tảng lập trình

Common Language Runtime : ngôn ngữ thực thi tổng quát

class library : lớp thư viện

foundation : nền tảng

memory management : quản lý bộ nhớ

collection : bộ sưu tập

aspects of code : các khía cạnh của code

execution time : thời gian thực hiện

accuracy : sự chính xác

core services : những dịch vụ cốt lỗi

task : nhiệm vụ

various functionalities : các chức năng khác nhau

components : các thành phần

variables : biến (ví dụ f(x) = x2 ->x chính là biến trong hàm)

memory location : vị trí bộ nhớ

name : tên

data type : kiểu dữ liệu

underscore character : ‘\_’ dấu gạch dưới

semicolon : ‘;’ dấu chấm phẩy

commas : ‘,’ dấu phẩy

single quotes : “ ‘ “ dấu nháy đơn

parentheses : ‘(‘ dấu ngoặc đơn

curly braces : ‘{‘ dấu ngoặc nhọn

quotation marks : ‘ “ ‘ dấu ngoặc kép

two slashes : ‘//’ 2 dấu xẹt

tilde : dấu ngã ‘~’

information : thông tin

identifier : sự định danh

statement : câu lệnh

operation : sự điều hành

sequence : sự liên tục

character : chữ cái

additional software : phần mềm bổ sung

classes : các lớp

methods : các phương thức

arguments : các đối số

namespaces : không gian tên

general message : thông báo chung

text-only interface : giao diện thuần văn bản

a line terminator : dấu xuống dòng

console : bàn điều khiển

cursor : con trỏ

formatted string : chuỗi định dạng

syntax : cú pháp

conversion : sự chuyển đổi

alternatives : lựa chọn thay thế

default : mặc định

precedence : quyền ưu tiên

Operator : hệ điều hành

compound assignment operators : toán tử gán ghép

Prefix : tiền tố

Postfix : hậu tố

nested if Statements : lệnh if lồng nhau

equality : sự bằng nhau

increment : sự gia tăng

decrement : sự giảm dần

iteration : sự lặp lại

loop : vòng lặp

expression : biểu thức

reusable code : tái sử dụng code

parameter : tham số

rectangle : hình chữ nhật

Recursion : đệ quy

Pyramid : kim tự tháp

algorithm : thuật toán

properties : tính chất

instance : ví dụ

instantiation : sự tức thời

characteristic : nét đặc trưng

Stack : ngăn xếp - bộ nhớ tĩnh lưu các biến biết trước dung lượng

Heap : chất đống - bộ nhớ động lưu các dữ liệu ko biết trước dung lượng

encapsulation : sự đóng gói

access modifier : công cụ sửa đổi truy cập

constructors : hàm khởi tạo

convention : quy ước

accessors : người truy cập

Arrays : mảng

Strings : chuỗi

elements : phần tử

situation : tình huống

index : chỉ mục

Jagged Arrays : mảng răng cưa

Inheritance : sự kế thừa

Polymorphism : đa hình

## Verb

enable : cho phép, kích hoạt

build : xây dựng

create : chế tạo

run : chạy

consists of : bao gồm

collect : sưu tầm

accomplish : đạt được

file access : truy cập file

reserves : dự trữ

store : lưu trữ

declare : khai báo

specify : xác định

complete : hoàn thành

descriptive of : mô tả về

contain : lưu trữ

perform : biểu diễn

install : cài đặt

Pressing any key : nhấn phím bất kỳ

pass data : truyền dữ liệu

display : trưng bày

include : bao gồm

prompt : nhắc nhở

assign : chỉ định

convert : chuyển đổi

expect : mong đợi

follow : theo

facilitate : tạo điều kiện

alter : thay đổi

evaluate : đánh giá

continue : tiếp tục

break : phá vỡ

indicate : biểu thị

define : định nghĩa

input : đầu vào

output : đầu ra

value : theo giá trị

reference : theo tham chiếu

concatenate values : nối các giá trị

determine : xác định

represent : đại diện

omit : bỏ sót

## Adjective

elegant : thanh lịch

variety of : đa dạng

secure : an toàn

robust : mạnh mẽ

upcoming : sắp tới

complex : phức tạp

important : quan trọng

understandable : có thể hiểu được

readable : có thể đọc được

True : đúng

False : sai

different : khác nhau

empty : bỏ trống

explanatory : được giải thích

conditional : có điều kiện

entire : toàn bộ

distinguished : được phân biệt

dynamic : năng động

private : riêng tư

public : cộng đồng

protected : được bảo vệ

flexible : linh hoạt

identical : giống hệt

## Other

such as : như là

extensively : một cách chuyên sâu

automatically : một cách tự động

implicitly : ngầm hiểu

explicitly : rõ ràng

indefinitely : vô thời hạn

foreach : cho mỗi

# End – 30 / 12 / 2020!