Fundamentals of C#

Introduction to programming

After this lesson you will know:

- What is computer program
- What is a programming language
- Basics of object oriented programming

What is a computer program?

What is a computer program?

A List of Instructions for the computer to execute

What is a programming language?

What is a programming language?

A programming language is a set of commands, instructions, and other syntax use to create a software program. Languages that programmers use to write code are called "high-level languages." This code can be compiled into a "low-level language," which is recognized directly by the computer hardware.

Change tire program:

- Pull over the vehicle as safely as possible
- Open the car's trunk
- Locate the wrench in tools compartment and the jack
- Head over to the flat tire and loosen the bolts of the tire
- Then place the jack underneath the vehicle
- Crank the jack to begin raising the vehicle
- Keep cranking until the tire raised 2 inches above ground
- Remove the loosened bolts and take off the tire
- Go to trunk and pull spare tire from compartment
- Bring tire and attach it to the car's leg
- Bolt the tire on as tight as you can
- Crank the jack to lower the vehicle until tire touches ground
- Tighten the bolts completely after the tire is attached
- Remove the jack and store away the tools and flat tire

Methods

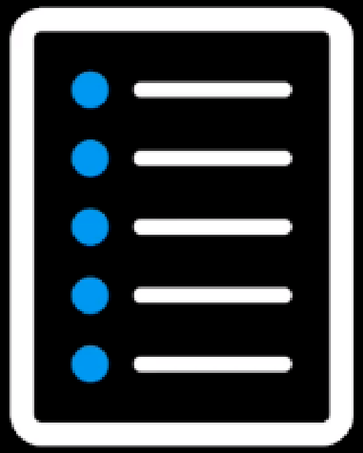
Methods

Named block of instructions

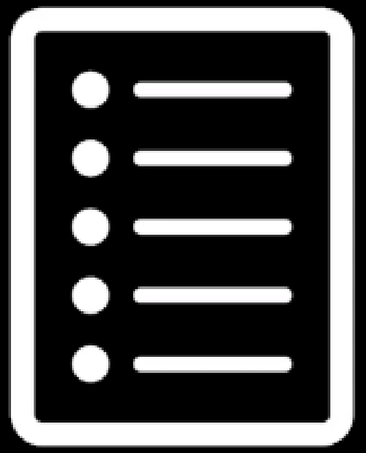
Park car:
Get tools:
Tire change:
Save tools:

- Pull over the vehicle as safely as possible
- Open the car's trunk
- Locate the wrench in tools compartment and the jack
- Head over to the flat tire and loosen the bolts of the tire
- Then place the jack underneath the vehicle
- Crank the jack to begin raising the vehicle
- Keep cranking until the tire raised 2 inches above ground
- Remove the loosened bolts and take off the tire
- Go to trunk and pull spare tire from compartment
- Bring tire and attach it to the car's leg
- Bolt the tire on as tight as you can
- Crank the jack to lower the vehicle until tire touches ground
- Tighten the bolts completely after the tire is attached
- Remove the jack and store away the tools and flat tire

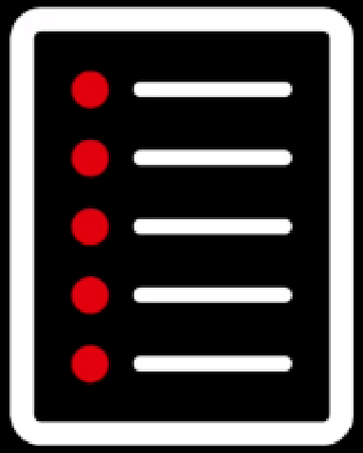
Park Car



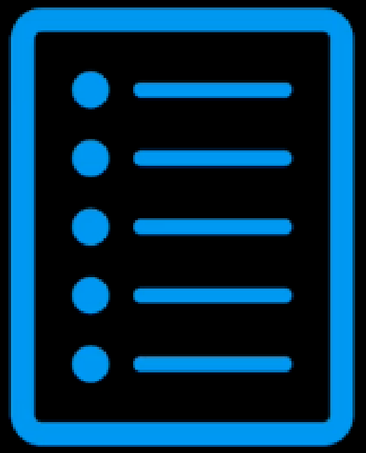
Tire Change

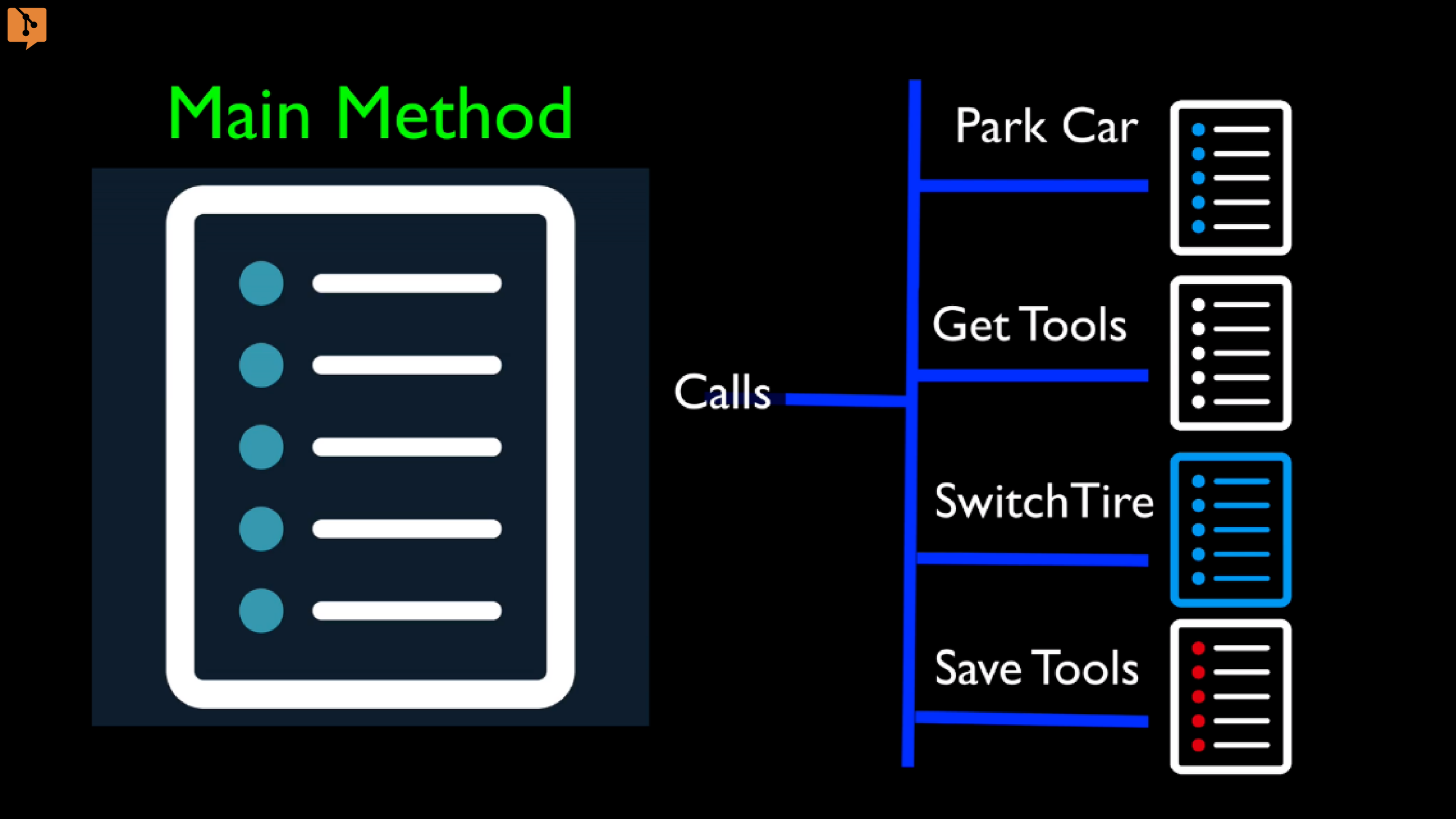


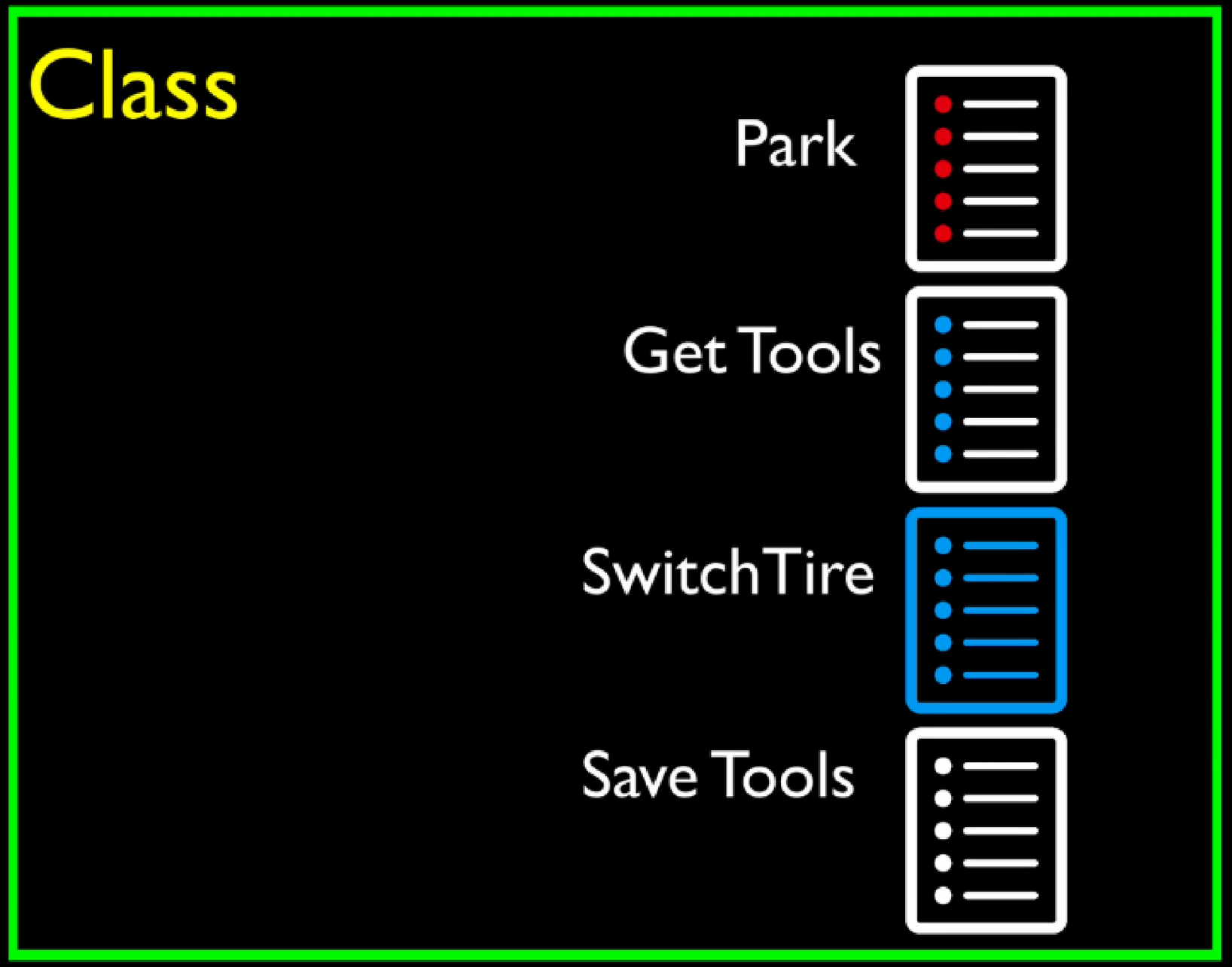
Get Tools



Save Tools




Main Method
Calls
Park Car
Get Tools
SwitchTire
Save Tools


Class
Park
Get Tools
SwitchTire
Save Tools

What is C#?

After this lesson you will know:

- Key features on why c# is so popular

C# is a general object-oriented programming (OOP) language for networking and Web development. C# is specified as a common language infrastructure (CLI) language. In January 1999, Dutch software engineer Anders Hejlsberg formed a team to develop C# as a complement to Microsoft's NET framework. Initially, C# was developed as C-Like Object Oriented Language (Cool). The actual name was changed to avert potential trademark issues. In January 2000, NET was released as C#. Its NET framework promotes multiple Web technologies.

The following reasons make C# a widely used professional language –

- It is a modern, general-purpose programming language
- It is object oriented.
- It is component oriented.
- It is easy to learn.
- It is a structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- It is a part of .Net Framework.

Platform independence

- With .NET core C# could be run on any machine.

Robustness

- C# is strongly typed.
- C# supports pointers in a limited extent.
- Its automatic memory management (garbage collection) eliminates memory leaks and other problems associated with dynamic memory allocation/de-allocation.

Performance

Maturity

- Strong community.
- Plenty of tools available.
- Reliable.

Version	Release date
C# 1.0	2000
…	…
C# 1.2	2003
C# 2.0	2005
C# 3.0	2007
C# 4.0	2010
C# 5.0	2013
C# 6.0	2015
C# 7.0	March 2017

Other .NET languages

- F#
- Visual Basic

Types & Variables

After this lesson you will know:

- Why are types needed
- Primitive data types
- How to work with *String*
- What is a variable and scope
- How to name variables
- How variables are stored in the memory
- What is type casting
- What is **null**

Static typing

```
Int32 d = 5;
String hello = "Hello";
Console.WriteLine(d);
Console.WriteLine(hello)
```

Dynamic typing

```
x=10
print x
x="Hello world!"
print x
```

Static typing

More errors detected earlier in development.

```
Int32 d = 5;
String hello = "Hello";
Console.WriteLine(d);
Console.WriteLine(hello)
```

Dynamic typing

```
x=10
print x
x="Hello world!"
print x
```

Static typing

More errors detected earlier in development.

Allows for compiler optimisation which yields faster code.

Dynamic typing

```
x=10
print x
x="Hello world!"
print x
```

Static typing

More errors detected earlier in development.

Allows for compiler optimisation which yields faster code.

Can lead to boilerplate.

Dynamic typing

```
x=10
print x
x="Hello world!"
print x
```

Static typing

More errors detected earlier in development.

Allows for compiler optimisation which yields faster code.

Can lead to boilerplate.

Dynamic typing

Less boilerplate for a self describing data.

```
x=10
print x
x="Hello world!"
print x
```

Static typing

More errors detected earlier in development.

Allows for compiler optimisation which yields faster code.

Can lead to boilerplate.

Dynamic typing

Less boilerplate for a self describing data.

More errors detected later in development and in maintenance.

Static typing

More errors detected earlier in development.

Allows for compiler optimisation which yields faster code.

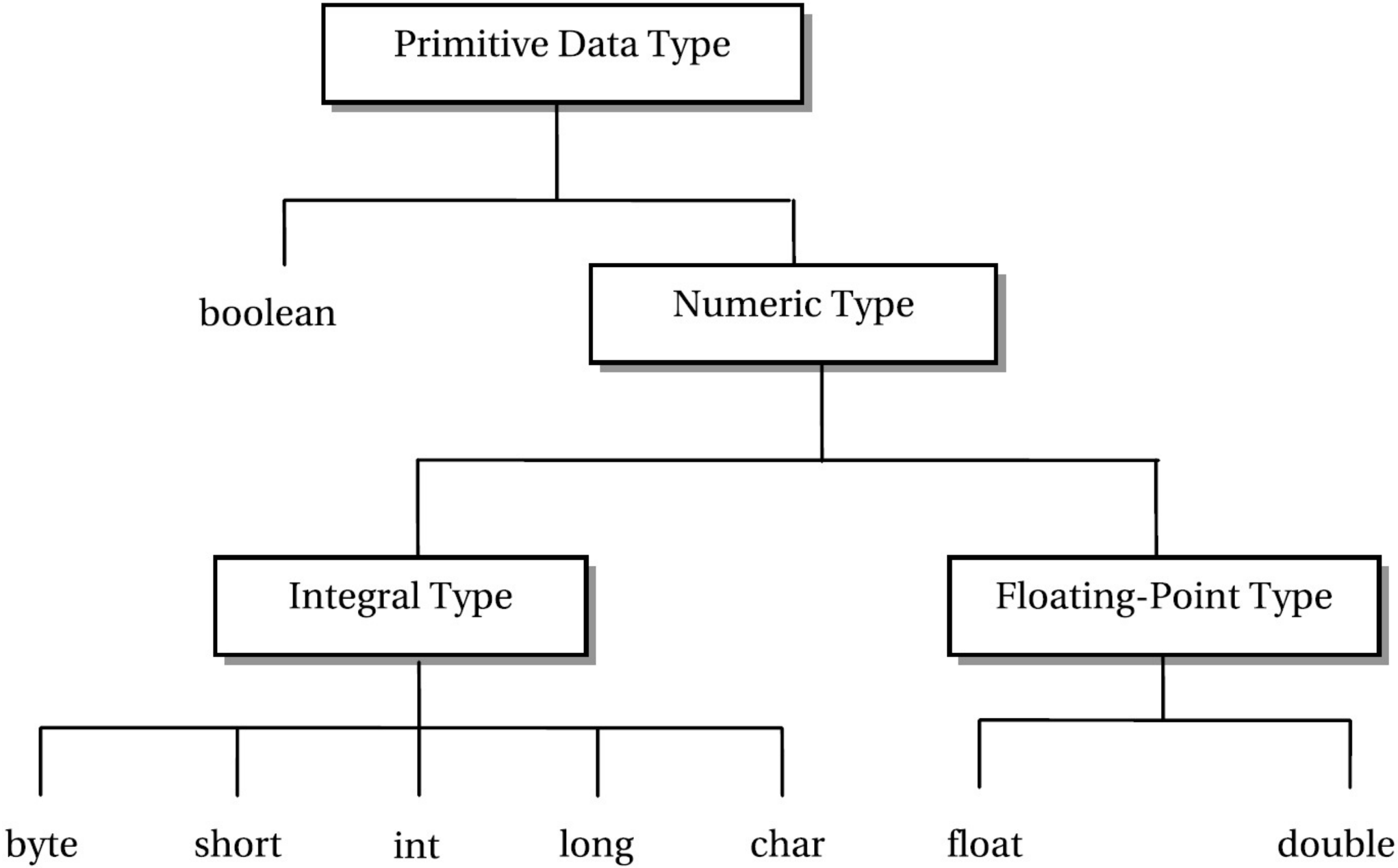
Can lead to boilerplate.

Dynamic typing

Less boilerplate for a self describing data.

More errors detected later in development and in maintenance.

Tends to prohibit compilation and yields poor performing code.


Primitive Data Type
boolean
Numeric Type
Integral Type
Floating-Point Type
byte
short
int
long
char
float
double

Integers

Name	Width	Range
long	64	-9 223 372 036 854 775 808 to +9 223 372 036 854 775 807
int	32	-2 147 483 648 to +2 147 483 647
short	16	-32 768 to +32 767
byte	8	-128 to +127

- The most commonly used integer type is *int*.
- If the integer values are larger than its feasible range, then an overflow occurs.

Floating point numbers

Name	Width	Approximate
double	64	4.9e-324 to 1.8e+308
float	32	1.4e-045 to 3.4e+038

- Floats are used when evaluating expressions that require fractional precision.
- Be aware that floating-point arithmetic can only approximate real arithmetic. See 0.30000000000000004.com.

Boolean

- The program is supposed to do decision making by itself.
- To do this, C# has the boolean-type flow controls (selections and iterations).
- This type has only two possible values, *true* and *false*.

Char

- A character stored by the machine is represented by a sequence of 0’s and 1’s.
- The char type is a 16-bit unsigned primitive data type.
- C# uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.

String [documentation]

```
String example1 = "CODELEX";
char[] chars = { 'C', 'O', 'D', 'E', 'L', 'E', 'X' }
String example2 = new String(chars);
```

String concatenation

```
String stringType = "String";
int otherType = 5;
String example = "A " + stringType
         + " can be concatenated"
         + " with other types also"
         + otherType;
Console.WriteLine(example);
```

String immutability

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

String immutability

Once created *String* cannot be changed

```
// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

String immutability

Once created *String* cannot be changed

```
// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
```

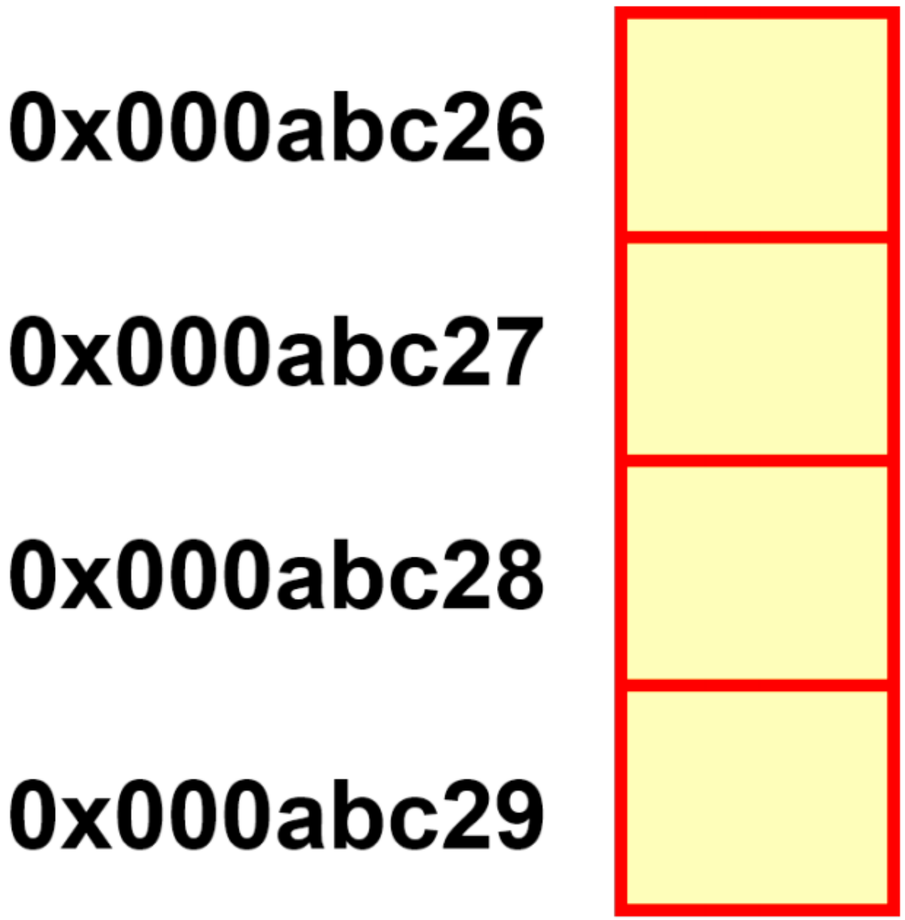
If String method returns String - although it may look similar but it is a new object

Variable ≈ Box

Variable Declaration

- Give it a name, for example *x*.
- Assign a type.
- For example `int x = 10;`
- Variable declaration tells the compiler to allocate appropriate memory space for the variable based on its data type.

How variables are stored in memory?



How variables are stored in memory?

- The number *0x000abc26* stands for one memory address in hexadecimal (0-9, and a-f).
- The variable *x* itself refers to *0x000abc26* in the program after compilation.

Two "Before" rules

- A declared variable must have a value assigned before it can be used.
- A variable must be declared before it can have a value assigned.
  - In practice, do not declare the variable until you need it.

What if there is no value?

- If there is no memory allocated for the variable it is **null**.
- Primitive types cannot be **null**.

```
static void Main(string[] args)
{
    //wont even compile
    int integer = null;
    Console.WriteLine(integer.toString());
}
```

Variable scope

- Class Level Scope
- Method Level Scope
- Block Level Scope

Class Level Scope

- Declaring the variables in a class but outside any method can be directly accessed anywhere in the class.
- These variables are also termed as the fields or class members.
- Class level scoped variable can be accessed by the non-static methods of the class in which it is declared.
- Access modifier of class level variables doesn’t affect their scope within a class.
- Member variables can also be accessed outside the class by using the access modifiers.

```
// C# program to illustrate the
// Class Level Scope of variables
using System;

// declaring a Class
class GFG { // from here class level scope starts

    // this is a class level variable
    // having class level scope
    int a = 10;

    // declaring a method
    public void display()
    {
        // accessing class level variable
        Console.WriteLine(a);
```

Method Level Scope

- Variables that are declared inside a method have method level scope. These are not accessible outside the method.
- However, these variables can be accessed by the nested code blocks inside a method.
- These variables are termed as the local variables.
- There will be a compile-time error if these variables are declared twice with the same name in the same scope.
- These variables don’t exist after method’s execution is over.

```
// C# program to illustrate the
// Method Level Scope of variables
using System;

// declaring a Class
class GFG { // from here class level scope starts

    // declaring a method
    public void display()

    { // from here method level scope starts

        // this variable has
        // method level scope
        int m = 47;

        // accessing method level variable
        Console.WriteLine(m);

    } // here method level scope ends

    // declaring a method
    public void display1()

    { // from here method level scope starts

        // it will give compile time error as
```

Block Level Scope

- These variables are generally declared inside the for, while statement etc.
- These variables are also termed as the loop variables or statements variable as they have limited their scope up to the body of the statement in which it declared.
- Generally, a loop inside a method has three level of nested code blocks(i.e. class level, method level, loop level).
- The variable which is declared outside the loop is also accessible within the nested loops. It means a class level variable will be accessible to the methods and all loops. Method level variable will be accessible to loop and method inside that method.
- A variable which is declared inside a loop body will not be visible to the outside of loop body.

```
// C# code to illustrate the Block
// Level scope of variables
using System;

// declaring a Class
class GFG

{ // from here class level scope starts

    // declaring a method
    public void display()

    { // from here method level scope starts

        // this variable has
        // method level scope
        int i = 0;

        for (i = 0; i < 4; i++) {

            // accessing method level variable
            Console.WriteLine(i);
        }

        // here j is block level variable
        // it is only accessible inside
        // this for loop
        for (int j = 0; j < 5; j++) {
            // accessing block level variable
            Console.WriteLine(j);
        }

        // this will give error as block level
        // variable can't be accessed outside
```

Naming rules

- Identifiers are the names that identify the elements such as **variables**, **methods**, and **classes** in the program.
- The naming rule excludes the following situations:
  - cannot start with a digit
  - cannot be any reserved word
  - cannot include any blank between letters
  - cannot contain +, -, ☒, / and %
  - C# is case sensitive.

Type Conversion and Compatibility

- Type conversion happens when we assign the value of one data type to another.
- If the data types are compatible, then C# does Automatic Type Conversion.
- If not comparable, then they need to be converted explicitly which is known as Explicit Type conversion.
- For example, assigning an int value to a long variable.
- For example, the integer 1 is compatible to a double value 1.0.
- However, there is no automatic conversion from double to int. (Why?)

Lossy Conversion

- if we want to assign a value of larger data type to a smaller data type we perform explicit type casting.
- This is useful for incompatible data types where automatic conversion cannot be done.(double to int)
- Here, target-type specifies the desired type to convert the specified value to.
- Sometimes, it may result into the lossy conversion.

Lossy Conversion

```
// C# program to demonstrate the
// Explicit Type Conversion
using System;
namespace Casting{

    class GFG {

            // Main Method
            public static void Main(String []args)
            {
                double d = 765.12;

                // Explicit Type Casting
                int i = (int)d;

                // Display Result
                Console.WriteLine("Value of i is " +i); //???
            }
    }
```

Explanation: Here due to lossy conversion, the value of i becomes 765 and there is a loss of 0.12 value.

Casting

C# provides **built-in** methods for Type-Conversions.

Built-in casting

- **ToBoolean**
- **ToChar**
- **ToByte**
- **ToDecimal**
- **ToDouble**
- **ToInt16**
- **ToInt32**
- **ToInt64**
- **ToString**
- **ToUInt16**
- **ToUInt32**
- **ToUInt64**

```
// C# program to demonstrate the
// Built- In Type Conversion Methods
using System;
namespace Casting{

class GFG {

        // Main Method
        public static void Main(String []args)
        {
            int i = 12;
            double d = 765.12;
            float f = 56.123F;

            // Using Built- In Type Conversion
            // Methods & Displaying Result
            Console.WriteLine(Convert.ToString(f));
```

Type Checking

- Note that the C# compiler does only type-checking but no real execution before compilation.
- In other words, the values of variables are unknown until they are really executed.

QUIZ

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(0.5 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1)
        //what will be the output?
    }
}
```

In which of the following answers does the number of bits increase from fewest to most?

- byte < long < short < int
- int < byte < short < long
- byte < short < int < long
- short < byte < long < int

```
static void Main(string[] args)
{
    int t;
    Console.WriteLine(t);
    // What will be the output?
    // 1. 0
    // 2. runtime error
    // 3. compilation error
    // 4. null
}
```

```
static void Main(string[] args)
{
    {
        String name = "CODELEX";
        Console.WriteLine("programing @" + name);
    }
    name = "codelex";
    Console.WriteLine("programing @" + name);
    // What will be the output?
    // 1. programming @CODELEX 
    //  programming @codelex
    // 2. programming @CODELEX 
    //  programming @CODELEX
    // 3. runtime error
    // 4. compilation error
}
```

Which of these statements are true?

- null can be any type, including primitive types
- primitive - *int*, object - Integer
- null can be any type, excluding primitive types
- primitive - *Integer*, object - int

C# Programming Yellow Book

- Variables and Scope [Chapter 3.2]

C# Notes for Professionals

- Literals [Chapter 18]
- Built-in Types [Chapter 32]
- Type Conversion [Chapter 36]
- Casting [Chapter 37]

Further reading and tutorials:

- Data types
- Static vs dynamic typing
- Why String is Immutable in C#?
- Primitives vs objects

Operators

After this lesson you will know:

- What are and how to apply:
  - arithmetic operators
  - comparison operators
  - boolean operators
  - arithmetic compound operators
- In which order arithmetic operations are being executed

Arithmetic operators

Name	Meaning	Example	Result
+	Addition	`4 + 3`	7
–	Subtraction	`9 – 4`	5
*	Multiplication	`2 * 6`	12
/	Division	`9 / 3`	3
%	Remainder	`20 % 3`	2

Integer division throws away reminder

Rational operators

C# operator	Mathematics symbol
<	<
<=	≤
>	>
>=	≥
==	=
!=	≠

Logical operators

Operator	Name	Description
!	not	logical negation
&&	and	logical conjunction
||	or	logical disjunction
^	exclusive or	logical exclusion

Truth table

x	y	!x	x && y	x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Arithmetic compound operators

Operator	Description
++	Increment
+=	Addition assignment
--	Decrement
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

Example

```
int x = 1;
Console.WriteLine(x);
// output 1
x = x + 1;
Console.WriteLine(x);
// output 2
x += 2;
Console.WriteLine(x);
// output 4
x++; // same as x += 1
     // and x = x + 1
Console.WriteLine(x);
// output 5
```

++x vs x++

```
//preincrement
int x = 1;
int y = ++x;
Console.WriteLine(y);
// output 2
Console.WriteLine(x);
// output 2

//postincrement
int w = 1;
int z = w++;
Console.WriteLine(z);
// output 1
Console.WriteLine(w);
```

Operator precedence

Precedence	*Operator*
↓	`var++` and `var--` (Postfix)
	`+`, `-` (Unary plus and minus), `++var` and `--var` (Prefix)
	(type) (Casting)
	`!` (Not)
	`*`, `/`, `%` (Multiplication, division, and remainder)
	`+`, `-` (Binary addition and subtraction)
	`<`, `<=`, `>`, `>=` (Comparison)
	`==`, `!=` (Equality)
	`^` (Exclusive OR)
	`&&` (AND)
	`||` (OR)
	`=`, `+=`, `-=`, `*=`, `/=`, `%=` (Assignment operator)

Math methods

Method	Description
Math.Abs()	Absolute value
Math.Round()	Rounding
...	...

System.Math provides methods for mathematical operations

QUIZ

```
double x = 1 / 2;
Console.WriteLine(x);
// output?
```

```
Console.WriteLine(
                    1/2
                    - 1/10
                    - 1/10
                    - 1/10
                    - 1/10
                    - 1/10
                    );
// output?
```

```
boolean result = 1 < x < 3;
Console.WriteLine(result);
// output?
```

```
public class PrePostIncrement {
    public static void Main(string[] args) {
        int a = 21;
        int b = 35;
        int sum = a++ + ++b;
        Console.WriteLine("sum = " + sum);
        // 1. 56
        // 2. 57
        // 3. 58
        // 4. compilation error
    }
```

C# Programming Yellow Book

- Giving Values to Variables [Chapter 2.2.6]
- Controlling Program Flow [Chapter 2.3.2]

C# Notes for Professionals

- Operators [Chapter 3]

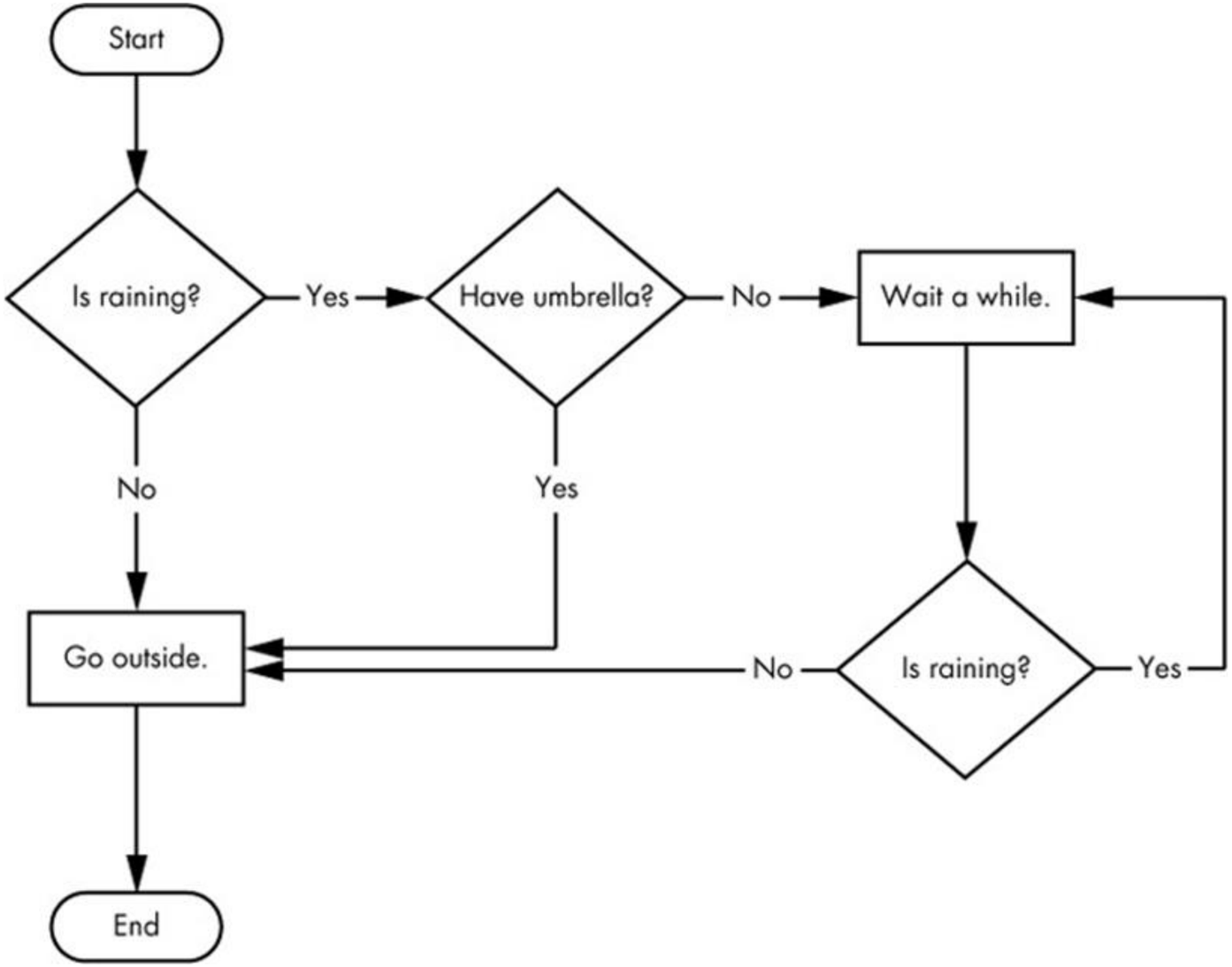
Flow of Control

After this lesson you will know:

- What is flow of control
- How to change flow
- How to use **if**, **else** and **switch** instructions

What is flow of control?

In computer science, flow of control (or control flow) is the order in which individual instructions are executed


Start
Is raining?
Yes
Have umbrella?
No
Wait a while.
No
Yes
Go outside.
No
Is raining?
Yes
End

```
int number = 10;

if (number > 0)
{
    Console.WriteLine("Number is positive.");
}
else
{
    Console.WriteLine("Number is not positive.");
}

Console.WriteLine("This statement is always executed.")
```

```
switch (number)
{
    case 1:
        Console.WriteLine("Number is one!");
        break;
    case 2:
        Console.WriteLine("Number is two!");
        break;
    case 10:
        Console.WriteLine("Number is ten!");
        break;
    default:
        Console.WriteLine("I don't know this number.")
        break;
```

```
int i = x > 5 ? 0 : 1;
// i == 0 if condition was true, i == 1 otherwise
```

QUIZ

How many choices are possible when using a single if-else statement?

- 1
- 2
- 3
- 4

```
//What is the output?

int sum = 14;

if (sum < 20)
{
    Console.WriteLine("Under");
}
else
{
    Console.WriteLine("Over");
}

Console.WriteLine("The limit.");
```

```
//What is the output?

int sum = 21;

if (sum != 20)
{
    Console.WriteLine("You win.");
    return;
}
else
{
    Console.WriteLine("You lose.");
    return;
}

Console.WriteLine("The prize.");
```

C# Programming Yellow Book

- Controlling Program Flow [Chapter 2.3.2]
- The Switch Construction [Chapter 3.5]

C# Notes for Professionals

- Conditional Statements [Chapter 4]

Arrays

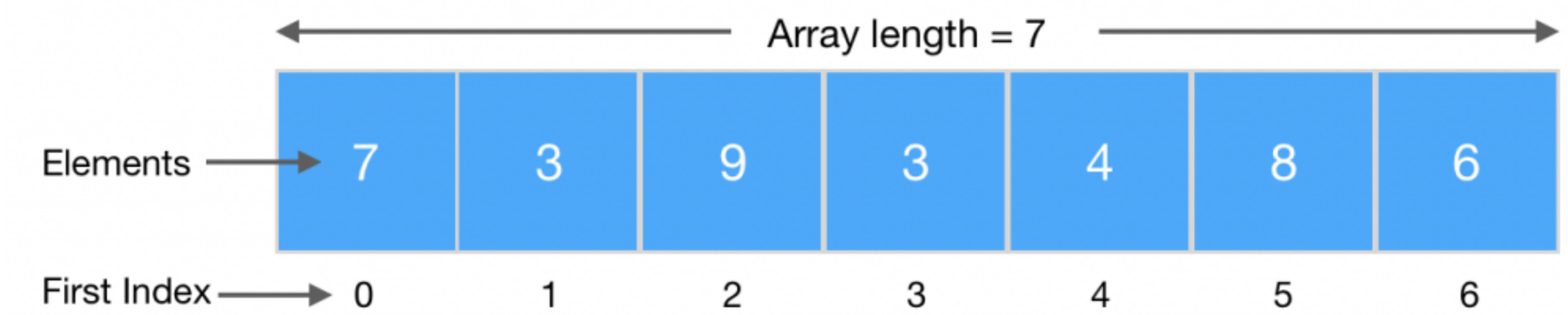
After this lesson you will know:

- What is an *array*
- What is a *multidimensional array*

What is an array?

The array is a data structure, which stores a fixed-size sequential collection of elements of the same type.

Array



```
//must have an initial size defined
int[] myArray = new int[7];

myArray[0] = 7;
myArray[1] = 3;
myArray[2] = 9;
myArray[3] = 8;
myArray[4] = 4;
myArray[5] = 8;
myArray[6] = 6;

Console.WriteLine(myArray[0]); //7
Console.WriteLine(myArray[3]); //8
```

```
int[] myArray = {
    1789, 2035, 1899, 1456, 2013,
    1458, 2458, 1254, 1472, 2365,
    1456, 2165, 1457, 2456, 1923
};

Console.WriteLine(myArray);
```

Two dimensional array

	Column 1	Column 2	Column 3	Column 4
Row 1	`a[0][0]`	`a[0][1]`	`a[0][2]`	`a[0][3]`
Row 2	`a[1][0]`	`a[1][1]`	`a[1][2]`	`a[1][3]`
Row 3	`a[2][0]`	`a[2][1]`	`a[2][2]`	`a[2][3]`

```
int t, i;
int[,] table = new int[3, 4];
for (t = 0; t< 3; ++t) {
    for (i = 0; i< 4; ++i) {
        table[t, i] = (t* 4) + i + 1;
        Console.Write(table[t, i] + " ");
    }
    Console.WriteLine();
}
/*
    1 2 3 4
    5 6 7 8
    9 10 11 12
*/
```

QUIZ

```
byte[] array = { 12, 34, 9, 0, -62, 88 };
Console.WriteLine(array.Length);
// What will be the output?
// 1. 7
// 2. 6
// 3. 5
// 4. compilation error
// 5. runtime error
```

```
int[] ar = { 2, 4, 6, 8 };

ar[0] = 23;
ar[3] = ar[0];

Console.WriteLine(ar[0] + " " + ar[3]);
// What will be the output?
// 1. 23 2
// 2. 23
// 3. 23 23
// 4. compilation error
// 5. runtime error
```

C# Programming Yellow Book

- Arrays [Chapter 3.3]

C# Notes for Professionals

- Arrays [Chapter 20]

Further reading and tutorials:

- Arrays @docs.microsoft.com

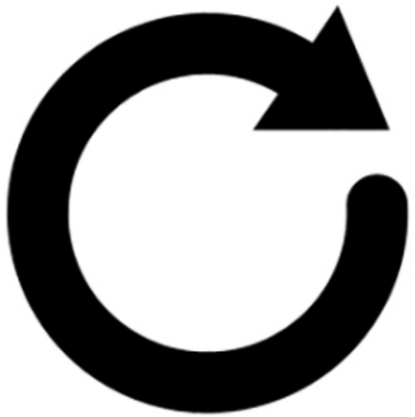
Loops

After this lesson you will know:

- What is a loop and infinite loop
- How to use "while", "do...while", "for" and "foreach" instructions
- How to **break** & **continue** the loop

What is loop?

Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.



LOOPS REPEAT ACTIONS...

SO YOU DON'T HAVE TO

```
int x = 1;

while (x <= 4)
{
    Console.WriteLine("Value of x: " + x);
    x++;
}
```

```
int x = 21;

do
{
    Console.WriteLine("Value of x: " + x);
    x++;
} while (x < 20);
```

```
for (int i = 1; i <= 4; i++)
{
    Console.WriteLine("Value of i: " + i);
}
```

```
string[] array = { "Ron", "Harry", "Hermione" };

foreach (var x in array)
{
    Console.WriteLine(x);
}
```

infinite loop - **bug**

```
int x = 1;

while (x <= 4)
{
    Console.WriteLine("Value of x: " + x);
}
```

```
using System;

namespace BreakExample
{
    class Program
    {
        private static void Main(string[] args)
        {
            int[] arrayOfInts = {32, 87, 3, 589, 12};
            const int searchFor = 12;

            int i;
            var foundIt = false;

            for (i = 0; i < arrayOfInts.Length; i++) {
                if (arrayOfInts[i] == searchFor) {
                    foundIt = true;
                    break;
                }
            }

            if (foundIt) {
                Console.WriteLine("Found " + searchFor + " at index " + i);
            } else {
                Console.WriteLine(searchFor + " not in the array");
            }
        }
```

```
using System;

namespace ContinueExample
{
    class Program
    {
        private static void Main(string[] args)
        {
            const string searchMe = "peter piper picked a " + "peck of pickled pe
            var max = searchMe.Length;
            var numPs = 0;

            for (var i = 0; i < max; i++)
            {
                var searchMeArray = searchMe.ToCharArray();
                if (searchMeArray[i] != 'p') {
                    continue;
                }
                numPs++;
```

QUIZ

```
using System;

namespace DoWhileExample
{
    class Program
    {
        private static void Main(string[] args)
        {
            var i = 0;
            do {
                Console.WriteLine("CODELEX");
            } while (i == 1);
            // What will be the output?
            // 1. CODELEX
            //    CODELEX
            // 2. CODELEX
            // 3.
```

```
using System;

namespace WhileQuiz
{
    class Program
    {
        private static void Main(string[] args)
        {
            var count = 0;
            while (count < 100) {
                // Point A
                Console.WriteLine("Welcome to Java!");
                count++;
                // Point B
            }
            // Point C


            //Which statement is true?
            // 1. count < 100 is always true at Point B
            // 2. count < 100 is always false at Point B
            // 3. count < 100 is always true at Point A
            //    and count < 100 is always false at Point C
```

```
using System;

namespace ReturnQuiz
{
    class Program
    {
        private static void Main(string[] args)
        {
            for (var i = 0; i < 3; i++) {
                if (i == 1) {
                    Console.Write("Bye!");
                    return;
                }
                Console.Write("Currently @" + i + ", ");
            }
            // What will be the output?
            // 1. Currently @0, Currently @1, Currently @2, Bye!
            // 2. Currently @0, Currently @1, Bye!
            // 3. Currently @0, Bye!
```

C# Programming Yellow Book

- Loops [Chapter 2.3.3]

C# Notes for Professionals

- Looping [Chapter 28]

Classes and Objects

Lesson Objective

- *Class*
- *Object* and *instance*
- *Method*
- **static**
- **constructor**
- **this**

What is a Class?

A class, in the context of C#, are **prototypes** / **templates** that are used to create objects.

```
class Car
{
    string brand;
    public Car(string brand)
    {
        this.brand = brand;
    }
}

Car car = new Car("Audi");
```

What is a Method?

A C# method is a collection of instructions that are grouped together to perform an operation.

```
class Car
{
    private string _brand;
    private bool _driving;
    public Car(string brand)
    {
        this._brand = brand;
        this._driving = false;
    }

    public bool driving
    {
        get => this._driving;
    }

    public void drive()
    {
        this._driving = true;
    }

    public void stop()
    {
        this._driving = false;
    }
}

var car = new Car("Audi");
```

What is Constructor?

Constructor is a special method which is invoked when we initialize object instance. If no constructor is defined default no arg constructor is generated.

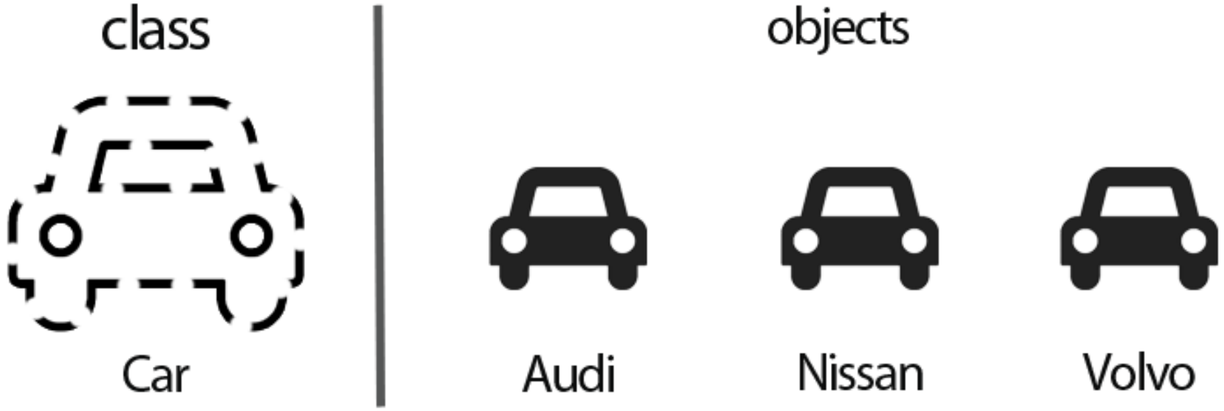
C# allows two types of constructors

- No argument Constructors
- Parameterized Constructors

```
class Car
{
    string brand;
    public Car(string brand)
    {
        this.brand = brand;
    }

    void drive()
    {
        //do stuff
    }
}

class Plane
{
    void fly()
    {
        //do stuff
    }
}

Car car = new Car("car");
car.drive();

Plane plane = new Plane();
plane.fly();
```

What is an Object?

Object refers to a particular instance of a class, where the object can be a combination of variables, functions, and data structures. In C# object is a direct or indirect base of every data type.


class
objects
Car
Audi
Nissan
Volvo

```
class Car
{
    private string _brand;

    public Car(string brand)
    {
        this._brand = brand;
    }

    public void drive()
    {
        //do stuff
    }
}

var cars = new Car[] {new Car("Audi"), new Car("Nissan"), new Car("Volvo")}
foreach (var car in cars)
```

static

- means that field/method
  - is defined for the class declaration
  - is not unique

```
class Car
{
    public static int Counter = 0;
    private string _brand;

    public Car(string brand)
    {
        this._brand = brand;
        Car.Counter++;
    }

    public static int countOfCarsCreated()
    {
        return Car.Counter;
    }
}

new Car("Audi");
new Car("Nissan");
```

Static variables are initialized

- When class is loaded
- Before any object of that class is created
- Before any static method of the class executes

this

- can be used inside a constructor or instance method
- **this** works as a reference to the current Object, whose Method or constructor is being invoked

```
class Car
{
    private int _mileage;

    public void drive()
    {
        this._mileage++;
    }

    public void report()
    {
        Console.WriteLine($"Current mileage is {_mileage}km")
    }
}

var audi = new Car("Audi");
for (int i = 0; i < 100; i++)
{
    audi.drive();
```

this & static

this & **static**

Remember: there is no **this** available in a **static** context, because there is no instance present

When to use **static**?

When to use **static**?

One rule-of-thumb: ask yourself **does it make sense to call this method, even if no Object has been constructed yet?** If so, it should definitely be static.

QUIZ

```
class Car
{
    private string _brand;

    public Car(string brand)
    {
        _brand = brand;
    }

    public static void stop()
    {
        Console.WriteLine($"{this._brand} is being stopped");
    }
}

Car audi = new Car("Audi");
audi.stop();
// What will be the output?
// 1. "Audi is being stopped..."
```

Which statements are false?

- class is like a blueprint, but instance is an object based on this blueprint
- instance is like a blueprint, but class is an object based on this blueprint
- instance variables can be accessed from a static context
- class variables can be accessed from an instance context

Which of these applies to static?

- is unique for each instance
- it not unique for each instance
- this is accessible in a static context
- this is not accessible in a static context
- once static field is changed all instances see the change
- once static field is changed only current instance see the change

C# Programming Yellow Book

- Designing With Objects [Chapter 4.5]
- Static Items [Chapter 4.6]
- The Construction of Objects [Chapter 4.7]

C# Notes for Professionals

- Constructors and Finalizers [Chapter 39]

Documentation

- Classes & Structs

Collections

After this lesson you will know:

- Collection types and how to operate with them:
  - ArrayList
  - List
  - SortedList
  - Stack
  - HashSet
    - How to choose appropriate collection

Why Array is not enough?

```
//create array bigger than needed
Book[] books = new Book[10];
int nextIndex = 0;
books[nextIndex] = b;

//track indexes
nextIndex = nextIndex + 1;
```

What if there will be more books?

List

- Modifiable list
- Internally implemented with arrays
- Features
  - Get/Insert items by index
  - Add/Remove items
  - Delete items
  - Loop over all items
  - GetRange/RemoveRange

Array -> List

```
Book[] books = new Book[10];
int nextIndex = 0;
books[nextIndex] = b;
nextIndex += 1;

List<Book> books
        = new List<Book>();
books.add(b);
```

ArrayList

- Modifiable list
- Features
  - Get/Insert items by index
  - Add/Remove items
  - AddRange/RemoveRange items
  - Loop over all items
  - Sort
  - Reverse

ArrayList

```
var myArryList = new ArrayList()
                            {
                                1,
                                "Two",
                                3,
                                4.5F
                            };

foreach (var val in myArryList)
    Console.WriteLine(val);
```

points to remember

- ArrayList can store items(elements) of any datatype.
- ArrayList resizes automatically as you add the elements.
- ArrayList values must be cast to appropriate data types before using it.
- ArrayList can contain multiple null and duplicate items.
- ArrayList can be accessed using foreach or for loop or indexer.
- Use Add(), AddRange(), Remove(), RemoveRange(), Insert(), InsertRange(), Sort(), Reverse() methods.

SortedList

- Modifiable list
- Internally implemented with arrays
- Features
  - Get/Insert items by index
  - Add/Remove items
  - AddRange/RemoveRange items
  - Loop over all items
  - Sort

Internally, SortedList maintains two object[] array, one for keys and another for values. So when you add key-value pair, it runs a binary search using the key to find an appropriate index to store a key and value in respective arrays. It re-arranges the elements when you remove the elements from it.

```
SortedList sortedList = new SortedList()
                            {
                                {3, "Three"},
                                {4, "Four"},
                                {1, "One"},
                                {5, "Five"},
                                {2, "Two"}
                            };
for (int i = 0; i < sortedList.Count; i++)
{
    Console.WriteLine("key: {0}, value: {1}",
        sortedList.GetKey(i), sortedList.GetByIndex(i));
}
```

- C# has generic and non-generic SortedList.
- SortedList stores the key-value pairs in ascending order of the key. Key must be unique and cannot be null whereas value can be null or duplicate.
- Non-generic SortedList stores keys and values of any data types. So values needs to be cast to appropriate data type.
- Key-value pair can be cast to DictionaryEntry.
- Access individual value using indexer. SortedList indexer accepts key to return value associated with it.

Hashtable

- Modifiable table
- Features
  - Add/Remove items
  - Check for existence either by key or value
  - Clear all the entries in collection

```
Hashtable ht = new Hashtable();

ht.Add(1, "One");
ht.Add(2, "Two");
ht.Add(3, "Three");
ht.Add(4, "Four");
ht.Add("Fv", "Five");
ht.Add(8.5F, 8.5F);

string strValue1 = (string)ht[2];
string strValue2 = (string)ht["Fv"];
float fValue = (float) ht[8.5F];

Console.WriteLine(strValue1);
Console.WriteLine(strValue2);
Console.WriteLine(fValue);
```

- Hashtable stores key-value pairs of any datatype where the Key must be unique.
- The Hashtable key cannot be null whereas the value can be null.
- Hashtable retrieves an item by comparing the hashcode of keys. So it is slower in performance than Dictionary collection.
- Hashtable uses the default hashcode provider which is object.GetHashCode(). You can also use a custom hashcode provider.
- Use DictionaryEntry with foreach statement to iterate Hashtable.

Stack

- stores elements in LIFO style (Last In First Out).
- Stack allows null value and also duplicate values.
- It provides a Push() method to add a value and Pop() or Peek() methods to retrieve values.

```
Stack myStack = new Stack();
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);
myStack.Push(5);

Console.Write("Number of elements in Stack: {0}", myStack.Count);

while (myStack.Count > 0)
     Console.WriteLine(myStack.Pop());

Console.Write("Number of elements in Stack: {0}", myStack.Count);
```

- Stack stores the values in LIFO (Last in First out) style. The element which is added last will be the element to come out first.
- Use the Push() method to add elements into Stack.
- The Pop() method returns and removes elements from the top of the Stack. Calling the Pop() method on the empty Stack will throw an exception.
- The Peek() method always returns top most element in the Stack.

Queue

- Queue stores the elements in FIFO style (First In First Out)
- Queue collection allows multiple null and duplicate values.
- Use the Enqueue() method to add values and the Dequeue() method to retrieve the values from the Queue.

```
Queue queue = new Queue();
queue.Enqueue(3);
queue.Enqueue(2);
queue.Enqueue(1);
queue.Enqueue("Four");

Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);

while (queue.Count > 0)
    Console.WriteLine(queue.Dequeue());

Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);
```

- The Queue stores the values in FIFO (First in First out) style. The element which is added first will come out First.
- Use the Enqueue() method to add elements into Queue
- The Dequeue() method returns and removes elements from the beginning of the Queue. Calling the Dequeue() method on an empty queue will throw an exception.
- The Peek() method always returns top most element.

How to choose what's right?

QUIZ

```
Stack st = new Stack();
st.Push("Csharp");
st.Push(7.3);
st.Push(8);
st.Push('b');
st.Push(true);

// a) Unsimilar elements like “Csharp”,7.3,8 cannot be stored in the same stack collection.
// b) Boolean values can never be stored in Stack collection
// c) Perfectly workable code
// d) All of the mentioned
```

Which among the following is not the ordered collection class?

- List
- SortedList
- Queue
- Stack
- None of the mentioned

Which among the following is the correct way to find out the number of elements currently present in an ArrayListCollection called arr?

- arr.Capacity
- arr.Count
- arr.MaxIndex
- arr.UpperBound

C# Programming Yellow Book

- Generics and Collections [Chapter 5.1]

C# Notes for Professionals

- An overview of C# collections [Chapter 27]

Documentation

- Collections
- Choosing collection class

Polymorphism

After this lesson you will know:

What is an polymorphism?

What is an polymorphism?

> *The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the C#. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.*

What is an Inheritance?

It is the mechanism in c# by which one class is allow to inherit the features (fields and methods) of another class.

Super/
Parent Class

Sub/
Child
Classes

Animal

Mammals

Reptiles

Amphibians

Birds

```
public abstract class Animal {
    public abstract void makeSound();
}
```

```
public abstract class Bird : Animal {
    private string species;

    public void layEgg() {
        //complex egg laying process...
    }

    public abstract bool canFly();
}
```

```
public class Eagle : Bird {

    public override void makeSound() {
        Console.WriteLine("-- eagle specific sound --");
    }


    public override bool canFly() {
        return true;
    }

    public void attack() {
        // dangerously attack enemy
    }
```

```
public class Owl : Bird {

    public override void makeSound() {
        Console.WriteLine("-- owl specific sound --");
    }


    public override bool canFly() {
        return true;
    }

    public void lookSmart(){
        // look very smart
    }
```

```
public class CreateSomeAnimals {
    public static void main(string[] args) {
        List<Animal> animals = new List<Animal>();
            animals.Add(new Eagle());
            animals.Add(new Owl());

            animals.ForEach(animal => {
                animal.makeSound();
                // wont compile - animal.canFly()
            });
    }
}
```

Reusability

Facility to use public methods of base class without rewriting the same code over and over again.

Extensibility

Extending the base class logic as per business logic of the derived class.

Data hiding

Base class can decide to keep some data private so that it cannot be altered by the derived class.

What is an Interface?

An interface is a programming structure/syntax that allows the computer to enforce certain properties on an object (class). It is very similar to an abstract class - but interface is purely abstract.

Why to use interfaces?

An interface is a contract (or a protocol, or a common understanding) of what the classes can do. When a class implements a certain interface, it promises to provide implementation to all the methods declared in the interface.

Fields in the *interface*

```
public interface ITaxable
{
    double tax { get; }
    double calculateTax();
}
```

Interface naming

- Interface is a type, good names are:
  - *ITruck*, *ICar*, *IHuman*, *IAnimal* etc.
- Interface may describe a behaviour, good names are:
  - *ICloneable*, *IChargeable*, *IConsumable* etc.

Interfaces & abstract classes

Remember: in C# class can extend only one class, but can implement multiple interfaces

```
public class InstanceOfExample {
    public static void main(String[] args) {
        Animal animal = new Eagle();
        Console.WriteLine(animal is Animal);
        //true
        Console.WriteLine(animal is Bird);
        //true
        Console.WriteLine(animal is Eagle);
        //true
        Console.WriteLine(animal is Owl);
        //false
    }
}
```

QUIZ

Can an interface ever contain method bodies?

- No
- Yes
- Sometimes
- Always

```java
class Quiz {
    public static void main(String[] args) {
        Eagle var1 = new Animal();
        Animal var2 = new Eagle();
        Animal var3 = new Animal();
        // Which example wont compile and why?
        // 1. var1, var2
        // 2. var2
        // 3. var1, var3
        // 4. all of them
    }
}
```

Which statement is false?

- class can extend only one class, but implement more than one interface
- class can implement only one interface, but extend more than one class

Which statement is true if a child of an abstract parent class does **not** override all of the parent's abstract methods?

- Compilation error occurs in a child class
- The child class itself must be declared to be abstract
- Child classes are automatically non-abstract, so everything will work
- Compilation error occurs in a parent class