https://www.codecademy.com/learn/learn-c-sharp/modules/csharp-hello-world/cheatsheet

Table of Contents

Hello World

Comments

```
// This is a single line comment

/* This is a multi-line comment
   and continues until the end
   of comment symbol is reached */
```

Console.WriteLine()

```
Console.WriteLine("Hello, world!");

// Prints: Hello, world!
```

Data Types and Variables

.toUpper() in C#

```
string str2 = "This is C# Program xsdd_$#%";

// string converted to Upper case
string upperstr2 = str2.ToUpper();

//upperstr2 contains "THIS IS C# PROGRAM XSDD_$#%"
```

IndexOf() in C#

```
string str = "Divyesh";

// Finding the index of character
// which is present in string and
// this will show the value 5
int index1 = str.IndexOf('s');

Console.WriteLine("The Index Value of character 's' is " + index1);
//The Index Value of character 's' is 5
```

Bracket Notation

```
// Get values from this string.
string value = "Dot Net Perls";

//variable first contains letter D
char first = value[0];

//Second contains letter o
char second = value[1];

//last contains letter s
char last = value[value.Length - 1];
```

Substring() in C#

```
string myString = "Divyesh";
string test1 = myString.Substring(2);
```

String Concatenation in C#

```
// Declare strings
string firstName = "Divyesh";
string lastName = "Goardnan";
```

```
// Concatenate two string variables
string name = firstName + " " + lastName;
Console.WriteLine(name);
//Ths code will output Divyesh Goardnan
```

.ToLower() in C#

```
string mixedCase = "This is a MIXED case string.";

// Call ToLower instance method, which returns a new copy.
string lower = mixedCase.ToLower();

//variable lower contains "this is a mixed case string."
```

String Length in C#

```
string a = "One example";
Console.WriteLine("LENGTH: " + a.Length);
// This code outputs 11
```

String Interpolation in C#

```
int id = 100

// We can use an expression with a string interpolation.
string multipliedNumber = $"The multiplied ID is {id * 10}.";

Console.WriteLine(multipliedNumber);
// This code would output "The multiplied ID is 1000."
```

String New-Line

```
Console.WriteLine("Hello\nWorld");

// The console output will look like:
// Hello
// World
```

Variables and Types

```
string foo = "Hello";
string bar = "How are you?";
int x = 5;

Console.WriteLine(foo);
// Prints: Hello
```

Console.ReadLine()

```
Console.WriteLine("Enter your name: ");

name = Console.ReadLine();
```

Arithmetic Operators

```
int result;

result = 10 + 5;  // 15

result = 10 - 5;  // 5

result = 10 * 5;  // 50

result = 10 / 5;  // 2

result = 10 % 5;  // 0
```

Unary Operator

```
int a = 10;
a++;

Console.WriteLine(a);
// Prints: 11
```

Logic and Conditionals

If Statements

```

```

```
if (true) {
  // This code is executed.
  Console.WriteLine("Hello User!");
}

if (false) {
  // This code is skipped.
  Console.WriteLine("This won't be seen :(");
}
```

Break Keyword

```
string color = "blue";

switch (color) {
  case "red":
    Console.WriteLine("I don't like that color.");
    break;
  case "blue":
    Console.WriteLine("I like that color.");
    break;
  default:
    Console.WriteLine("I feel ambivalent about that color.");
    break;
}
// Regardless of where the break statement is in the above switch
statement,
// breaking will resume the program execution here.
Console.WriteLine("- Steve");
```

Comparison Operators

```
int x = 5;
Console.WriteLine(x < 6); // Prints "True" because 5 is less than 6.
Console.WriteLine(x > 8); // Prints "False" because 5 is not greater than
8.

string foo = "foo";
Console.WriteLine(foo == "bar"); // Prints "False" because "foo" does not
equal "bar".
```

Switch Statements

```
// The expression to match goes in parentheses.
switch (fruit) {
  case "Banana":
    // If fruit == "Banana", this block will run.
    Console.WriteLine("Peel first.");
    break;
  case "Durian":
    Console.WriteLine("Strong smell.");
    break;
  default:
    // The default block will catch expressions that did not match any
above.
    Console.WriteLine("Nothing to say.");
    break;
}

// The switch statement above is equivalent to this:
if (fruit == "Banana") {
  Console.WriteLine("Peel first.");
} else if (fruit == "Durian") {
  Console.WriteLine("Strong smell.");
} else {
  Console.WriteLine("Nothing to say.");
}
```

Boolean Expressions

```
// These expressions all evaluate to a boolean value.
// Therefore their values can be stored in boolean variables.
bool a = (2 > 1);
bool b = a && true;
bool c = !false || (7 < 8);
```

Boolean Type

```
bool skyIsBlue = true;
bool penguinsCanFly = false;
Console.WriteLine($"True or false, is the sky blue? {skyIsBlue}.");
// This simple program illustrates how booleans are declared. However, the
real power of booleans requires additional programming constructs such as
conditionals.
```

Logical Operators

```
// These variables equal true.
bool a = true && true;
bool b = false || true;
bool c = !false;

// These variables equal false.
bool d = true && false;
bool e = false || false;
bool f = !true;
```

Else Clause

```
if (true) {
  // This block will run.
  Console.WriteLine("Seen!");
} else {
  // This will not run.
  Console.WriteLine("Not seen!");
}

if (false) {
  // Conversely, this will not run.
  Console.WriteLine("Not seen!");
} else {
  // Instead, this will run.
  Console.WriteLine("Seen!");
}
```

If and Else If

```
int x = 100, y = 80;

if (x > y)
{
  Console.WriteLine("x is greater than y");
}
else if (x < y)
{
  Console.WriteLine("x is less than y");
}
else
{
  Console.WriteLine("x is equal to y");
}
```

Ternary Operator

```
bool isRaining = true;
// This sets umbrellaOrNot to "Umbrella" if isRaining is true,
// and "No Umbrella" if isRaining is false.
string umbrellaOrNot = isRaining ? "Umbrella" : "No Umbrella";

// "Umbrella"
Console.WriteLine(umbrellaOrNot);
```

Methods

Optional Parameters

```
// y and z are optional parameters.
static int AddSomeNumbers(int x, int y = 3, int z = 2)
{
  return x + y + z;
}

// Any of the following are valid method calls.
AddSomeNumbers(1); // Returns 6.
AddSomeNumbers(1, 1); // Returns 4.
AddSomeNumbers(3, 3, 3); // Returns 9.
```

Variables Inside Methods

```
static void DeclareAndPrintVars(int x)
{
  int y = 3;
  // Using x and y inside the method is fine.
  Console.WriteLine(x + y);
}

static void Main()
{
  DeclareAndPrintVars(5);

  // x and y only exist inside the body of DeclareAndPrintVars, so we
cannot use them here.
  Console.WriteLine(x * y);
}
```

Return Keyword

```
static int ReturnAValue(int x)
{
  // We return the result of computing x * 10 back to the caller.
  // Notice how we are returning an int, which matches the method's return
type.
  return x * 10;
}

static void Main()
{
  // We can use the returned value any way we want, such as storing it in
a variable.
  int num = ReturnAValue(5);
  // Prints 50 to the console.
  Console.WriteLine(num);
}
```

Expression-Bodied Methods

```
static int Add(int x, int y)
{
  return x + y;
}

static void PrintUpper(string str)
{
  Console.WriteLine(str.ToUpper());
}

// The same methods written in expression-body form.
static int Add(int x, int y) => x + y;

static void PrintUpper(string str) => Console.WriteLine(str.ToUpper());
```

Lambda Expressions

```
int[] numbers = { 3, 10, 4, 6, 8 };
static bool isTen(int n) {
  return n == 10;
}

// `Array.Exists` calls the method passed in for every value in `numbers`
and returns true if any call returns true.
Array.Exists(numbers, isTen);

Array.Exists(numbers, (int n) => {
  return n == 10;
});
```

```
// Typical syntax
// (input-parameters) => { <statements> }
```

Shorter Lambda Expressions

```
int[] numbers = { 7, 7, 7, 4, 7 };

Array.Find(numbers, (int n) => { return n != 7; });

// The type specifier on `n` can be inferred based on the array being
passed in and the method body.
Array.Find(numbers, (n) => { return n != 7; });

// The parentheses can be removed since there is only one parameter.
Array.Find(numbers, n => { return n != 7; });

// Finally, we can apply the rules for expression-bodied methods.
Array.Find(numbers, n => n != 7);
```

Void Return Type

```
// This method has no return value
static void DoesNotReturn()
{
  Console.WriteLine("Hi, I don't return like a bad library borrower.");
}
// This method returns an int
static int ReturnsAnInt()
{
  return 2 + 3;
}
```

Method Declaration

```
// This is an example of a method header.
static int MyMethodName(int parameter1, string parameter2) {
  // Method body goes here...
}
```

Out Parameters

return can only return one value. When multiple values are needed, out parameters can be used.

out parameters are prefixed with out in the method header. When called, the argument for each out parameter must be a variable prefixed with out.

The out parameters become aliases for the variables that were passed in. So, we can assign values to the parameters, and they will persist on the variables we passed in after the method terminates.

```
// f1, f2, and f3 are out parameters, so they must be prefixed with `out`.
static void GetFavoriteFoods(out string f1, out string f2, out string f3)
{
  // Notice how we are assigning values to the parameters instead of using
`return`.
  f1 = "Sushi";
  f2 = "Pizza";
  f3 = "Hamburgers";
}

static void Main()
{
  string food1;
  string food2;
  string food3;
  // Variables passed to out parameters must also be prefixed with `out`.
  GetFavoriteFoods(out food1, out food2, out food3);
  // After the method call, food1 = "Sushi", food2 = "Pizza", and food3 =
"Hamburgers".
  Console.WriteLine($"My top 3 favorite foods are {food1}, {food2}, and
{food3}");
}
```

Arrays and Loops

C# Arrays

```
// `numbers` array that stores integers
int[] numbers = { 3, 14, 59 };

// 'characters' array that stores strings
string[] characters = new string[] { "Huey", "Dewey", "Louie" };
```

Declaring Arrays

```
// Declare an array of length 8 without setting the values.
string[] stringArray = new string[8];
```

```
// Declare array and set its values to 3, 4, 5.
int[] intArray = new int[] { 3, 4, 5 };
```

Declare and Initialize array

```
// `numbers` and `animals` are both declared and initialized with values.
int[] numbers = { 1, 3, -10, 5, 8 };
string[] animals = { "shark", "bear", "dog", "raccoon" };
```

Array Element Access

```
// Initialize an array with 6 values.
int[] numbers = { 3, 14, 59, 26, 53, 0 };

// Assign the last element, the 6th number in the array (currently 0), to
58.
numbers[5] = 58;

// Store the first element, 3, in the variable `first`.
int first = numbers[0];
```

C# Array Length

```
int[] someArray = { 3, 4, 1, 6 };
Console.WriteLine(someArray.Length); // Prints 4

string[] otherArray = { "foo", "bar", "baz" };
Console.WriteLine(otherArray.Length); // Prints 3
```

C# For Loops

```
// This loop initializes i to 1, stops looping once i is greater than 10,
and increases i by 1 after each loop.
for (int i = 1; i <= 10; i++) {
  Console.WriteLine(i);
}

Console.WriteLine("Ready or not, here I come!");
```

C# For Each Loop

```
string[] states = { "Alabama", "Alaska", "Arizona", "Arkansas",
"California", "Colorado" };

foreach (string state in states) {
  // The `state` variable takes on the value of an element in `states` and
updates every iteration.
  Console.WriteLine(state);
}
// Will print each element of `states` in the order they appear in the
array.
```

C# While Loop

```
string guess = "";
Console.WriteLine("What animal am I thinking of?");

// This loop will keep prompting the user, until they type in "dog".
while (guess != "dog") {
  Console.WriteLine("Make a guess:");
  guess = Console.ReadLine();
}
Console.WriteLine("That's right!");
```

C# Do While Loop

```
do {
  DoStuff();
} while(boolCondition);

// This do-while is equivalent to the following while loop.

DoStuff();
while (boolCondition) {
  DoStuff();
}
```

C# Infinite Loop

```
while (true) {
  // This will loop forever unless it contains some terminating statement
such as `break`.
}
```

C# Jump Statements

```
while (true) {
  Console.WriteLine("This prints once.");
  // A `break` statement immediately terminates the loop that contains it.
  break;
}

for (int i = 1; i <= 10; i++) {
  // This prints every number from 1 to 10 except for 7.
  if (i == 7) {
    // A `continue` statement skips the rest of the loop and starts
another iteration from the start.
    continue;
  }
  Console.WriteLine(i);
}

static int WeirdReturnOne() {
  while (true) {
    // Since `return` exits the method, the loop is also terminated.
Control returns to the method's caller.
    return 1;
  }
}
```

Classes and Objects

C# Classes

In C#, classes are used to create custom types. The class defines the kinds of information and methods included in a custom type.

```
using System;

namespace BasicClasses
{
    class Forest {
    public string name;
    public int trees;
    }
}

// Here we have the Forest class which has two pieces of data, called
fields. They are the "name" and "trees" fields.
```

C# Constructor

```
// Takes two arguments
public Forest(int area, string country)
{
  this.Area = area;
  this.Country = country;
}

// Takes one argument
public Forest(int area)
{
  this.Area = area;
  this.Country = "Unknown";
}


// Typically, a constructor is used to set initial values and run any code
needed to “set up” an instance.

// A constructor looks li
```

C# Parameterless Constructor

In C#, if no constructors are specified in a class, the compiler automatically creates a parameterless constructor.

```
public class Freshman
{
  public string FirstName
  { get; set; }
}

public static void Main (string[] args)
{
  Freshman f = new Freshman();
  // name is null
  string name = f.FirstName;
}

// In this example, no constructor is defined in Freshman, but a
parameterless constructor is still available for use in Main().
```

C# Access Modifiers

In C#, members of a class can be marked with access modifiers, including public and private. A public member can be accessed by other classes. A private member can only be accessed by code in the same class.

By default, fields, properties, and methods are private, and classes are public.

```
public class Speech
{
  private string greeting = "Greetings";

  private string FormalGreeting()
  {
    return $"{greeting} and salutations";
  }

  public string Scream()
  {
    return FormalGreeting().ToUpper();
  }

}

public static void Main (string[] args)
{
  Speech s = new Speech();
  //string sfg = s.FormalGreeting(); // Error!
  //string sg = s.greeting; // Error!
  Console.WriteLine(s.Scream());
}

// In this example, greeting and FormalGreeting() are private. They cannot
be called from the Main() method, which belongs to a different class.
However the code within Scream() can access those members because Scream()
is part of the same class.
```

C# Field

In C#, a field stores a piece of data within an object. It acts like a variable and may have a different value for each instance of a type.

A field can have a number of modifiers, including: public, private, static, and readonly. If no access modifier is provided, a field is private by default.

```
public class Person
{
   private string firstName;
   private string lastName;
}

// In this example, firstName and lastName are private fields of the
Person class.

// For effective encapsulation, a field is typically set to private, then
accessed using a property. This ensures that values passed to an instance
are validated (assuming the property implements some kind of validation
for its field).
```

C# this Keyword

```
// We can use the this keyword to refer to the current class's members
hidden by similar names:
public NationalPark(int area, string state)
{
  this.area = area;
  this.state = state;
}

// The code below requires duplicate code, which can lead to extra work
and errors when changes are needed:
public NationalPark(int area, string state)
{
  area = area;
  state = state;
}
public NationalPark(int area)
{
  area = area;
  state = "Unknown";
}

// Use this to have one constructor call another:
public NationalPark(int area) : this (state, "Unknown")
{ }
```

C# Members

In C#, a class contains members, which define the kind of data stored in a class and the behaviors a class can perform.

```
class Forest
{
  public string name;
  public string Name
  {
    get { return name; }
    set { name = value; }
  }
}

// A member of a class can be a field (like name), a property (like Name)
or a method (like get()/set()). It can also be any of the following:
// Constants
// Constructors
// Events
```

```
// Finalizers
// Indexers
// Operators
// Nested Types
```

C# Dot Notation

```
string greeting = "hello";

// Prints 5
Console.WriteLine(greeting.Length);

// Returns 8
Math.Min(8, 920);
```

C# Class Instance

In C#, an object is an instance of a class. An object can be created from a class using the new keyword.

```
Burger cheeseburger = new Burger();
// If a class is a recipe, then an object is a single meal made from that
recipe.

House tudor = new House();
// If a class is a blueprint, then an object is a house made from that
blueprint.
```

C# Property

In C#, a property is a member of an object that controls how one field may be accessed and/or modified. A property defines two methods: a get() method that describes how a field can be accessed, and a set() method that describes how a field can be modified.

One use case for properties is to control access to a field. Another is to validate values for a field.

```
public class Freshman
{
  private string firstName;

  public string FirstName
  {
    get { return firstName; }
    set { firstName = value; }
  }
}
```

```
public static void Main (string[] args) {
  Freshman f = new Freshman();
  f.FirstName = "Louie";

  // Prints "Louie"
  Console.WriteLine(f.FirstName);
}

// In this example, FirstName is a property
```

C# Auto-Implemented Property

In C#, an auto-implemented property reads and writes to a private field, like other properties, but it does not require explicit definitions for the accessor methods nor the field. It is used with the { get; set; } syntax. This helps your code become more concise.

```
public class HotSauce
{
  public string Title
  { get; set; }

  public string Origin
  { get; set; }
}

// In this example, Title and Origin are auto-implemented properties.
Notice that a definition for each field (like private string title) is no
longer necessary. A hidden, private field is created for each property
during runtime.
```

C# Static Constructor

In C#, a static constructor is run once per type, not per instance. It must be parameterless. It is invoked before the type is instantiated or a static member is accessed.

```
class Forest
{
  static Forest()
  {
    Console.WriteLine("Type Initialized");
  }
}
// In this class, either of the following two lines would trigger the
static constructor (but it would not be triggered twice if these two lines
followed each other in succession):
```

```
Forest f  = new Forest();
Forest.Define();
```

C# Static Class

In C#, a static class cannot be instantiated. Its members are accessed by the class name.

This is useful when you want a class that provides a set of tools, but doesn't need to maintain any internal data.

Math is a commonly-used static class.

```
//Two examples of static classes calling static methods:

Math.Min(23, 97);
Console.WriteLine("Let's Go!");
```

Interfaces and Inheritance

C# Inheritance

In C#, inheritance is the process by which one class inherits the members of another class. The class that inherits is called a subclass or derived class. The other class is called a superclass, or a base class.

When you define a class that inherits from another class, the derived class implicitly gains all the members of the base class, except for its constructors and finalizers. The derived class can thereby reuse the code in the base class without having to re-implement it. In the derived class, you can add more members. In this manner, the derived class extends the functionality of the base class.

```
public class Honeymoon : TripPlanner
{ }

// Similar to an interface, inheritance also uses the colon syntax to
denote a class inherited super class. In this case, Honeymoon class
inherits from TripPlanner class.

// A derived class can only inherit from one base class, but inheritance
is transitive. That base class may inherit from another class, and so on,
which creates a class hierarchy.
```

C# override/virtual Keywords

In C#, a derived class (subclass) can modify the behavior of an inherited method. The method in the derived class must be labeled override and the method in the base class (superclass) must be labeled virtual.

The virtual and override keywords are useful for two reasons:

Since the compiler treats “regular” and virtual methods differently, they must be marked as such. This avoids the “hiding” of inherited methods, which helps developers understand the intention of the code.

```
class BaseClass
{
  public virtual void Method1()
  {
    Console.WriteLine("Base - Method1");
  }
}

class DerivedClass : BaseClass
{
  public override void Method1()
  {
    Console.WriteLine("Derived - Method1");   }
}
```

C# protected Keyword

In C#, a protected member can be accessed by the current class and any class that inherits from it. This is designated by the protected access modifier.

```


public class BankAccount
{
  protected decimal balance = 0;
}

public class StudentAccount : BankAccount
{
}

// In this example, the BankAccount (superclass) and StudentAccount
(subclass) have access to the balance field. Any other class does not.
```

C# abstract Keyword

In C#, the abstract modifier indicates that the thing being modified has a missing or incomplete implementation. It must be implemented completely by a derived, non-abstract class.

The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own.

If at least one member of a class is abstract, the containing class must also be marked abstract.

The complete implementation of an abstract member must be marked with override.

```
abstract class Shape
{
  public abstract int GetArea();
}

class Square : Shape
{
  int side;
  public Square(int n) => side = n;
  // GetArea method is required to avoid a compile-time error.
  public override int GetArea() => side * side;
}

// In this example, GetArea() is an abstract method within the abstract
Shape class. It is implemented by the derived class Square.
```

C# Interface

In C#, an interface contains definitions for a group of related functionalities that a class can implement.

Interfaces are useful because they guarantee how a class behaves. This, along with the fact that a class can implement multiple interfaces, helps organize and modularize components of software.

It is best practice to start the name of an interface with “I”.

```
interface IAutomobile
{
  string LicensePlate { get; }
  double Speed { get; }
  int Wheels { get; }
}

// The IAutomobile interface has three properties. Any class that
implements this interface must have these three properties.

public interface IAccount
{
  void PayInFunds ( decimal amount );
  bool WithdrawFunds ( decimal amount );
  decimal GetBalance ();
}

// The IAccount interface has three methods to implement.

public class CustomerAccount : IAccount
{ }

// This CustomerAccount class is labeled with : IAccount, which means that
```

```
it will implement that interface.
```

References

C# Reference Types

In C#, classes and interfaces are reference types. Variables of reference types store references to their data (objects) in memory, and they do not contain the data itself.

An object of type Object, string, or dynamic is also a reference type.

```
SportsCar sc = new SportsCar(100);
SportsCar sc2 = sc;
sc.SpeedUp(); // Method adds 20
Console.WriteLine(sc.Speed); // 120
Console.WriteLine(sc2.Speed); // 120

// In this code, sc and sc2 refer to the same object. The last two lines
will print the same value to the console.
```

C# Object Reference

In C#, an object may be referenced by any type in its inheritance hierarchy or by any of the interfaces it implements.

```
// Woman inherits from Human, which inherits from Animal, and it
implements IPerson:
class Human : Animal
class Woman : Human, IPerson

// All of these references are valid:
Woman eve = new Woman();
Human h = eve;
Animal a = eve;
IPerson p = eve;
```

C# Object Reference Functionality

In C#, the functionality available to an object reference is determined by the reference's type, not the object's type.

```
Player p = new Player();
Fan f = p;
p.SignContract();
```

```
f.SignContract();
// Error! 'SignContract()` is not defined for the type 'Fan'
```

C# Polymorphism

Polymorphism is the ability in programming to present the same interface for different underlying forms (data types).

We can break the idea into two related concepts. A programming language supports polymorphism if:

Objects of different types have a common interface (interface in the general meaning, not just a C# interface), and The objects can maintain functionality unique to their data type

```
class Novel : Book
{
  public override string Stringify()
  {
    return "This is a Novel!";
  }
}

class Book
{
  public virtual string Stringify()
  {
    return "This is a Book!";
  }
}

// In the below code, you'll see that a Novel and Book object can both be
referred to as Books. This is one of their shared interfaces. At the same
time, they are different data types with unique functionality.

Book bk = new Book();
Book warAndPeace = new Novel();
Console.WriteLine(bk.Stringify());
Console.WriteLine(warAndPeace.Stringify());

// This is a Book!
// This is a Novel

// Even though bk and warAndPeace are the same type of reference, their
behavior is different. Novel overrides the Stringify() method, so all
Novel objects (regardless of reference type) will use that method.
```

C# Upcasting

In C#, upcasting is creating an inherited superclass or implemented interface reference from a subclass reference.

```
// In this case, string inherits from Object:

string s = "Hi";
Object o = s;

// In this case, Laptop implements the IPortable interface:

Laptop lap = new Laptop();
IPortable portable = lap;
```

C# Downcasting

In C#, downcasting is creating a subclass reference from a superclass or interface reference.

Downcasting can lead to runtime errors if the superclass cannot be cast to the specified subclass.

```
Account a = new Account();
CustomerAccount ca = a;
// error CS0266: Cannot implicitly convert type `Account` to
`CustomerAccount`. An explicit conversion exists (are you missing a cast?)
```

```
// Dog inherits from Pet. An implicit downcast throws a compile-time
error:
Pet pet = new Pet();
Dog dog = pet;
// error CS0266: Cannot implicitly convert type `Pet` to `Dog`. An
explicit conversion exists (are you missing a cast?)

// Every downcast must be explicit, using the cast operator, like (TYPE).
This fixes the compile-time error but raises a new runtime error.
Pet pet = new Pet();
Dog dog = (Pet)pet;
// runtime error: System.InvalidCastException: Specified cast is not
valid.

//The explicit downcast would only work if the underlying object is of
type Dog:
Dog dog = new Dog();
Pet pet = dog;
Dog puppy = (Dog)pet;
```

C# Null Reference

In C#, an undefined reference is either a null reference or unassigned. A null reference is represented by the keyword null.

Be careful when checking for null and unassigned references. We can only compare a null reference if it is explicitly labeled null.

```
MyClass mc; //unassigned

Console.WriteLine (mc == null);
// error CS0165: Use of unassigned local variable 'mc'

MyClass mc = null; //explicitly 'null'

Console.WriteLine(mc == null);
// Prints true.

// Array of unassigned references
MyClass[] objects = new MyClass[5];
// objects[0] is unassigned, objects[1] is unassigned, etc...
```

C# Value Types

In C#, value types contain the data itself. They include int, bool, char, and double.

Here's the entire list of value types:

char, bool, DateTime All numeric data types Structures (struct) Enumerations (enum)

C# Comparison Type

In C#, the type of comparison performed with the equality operator (==), differs with reference and value types.

When two value types are compared, they are compared for value equality. They are equal if they hold the same value.

When two reference types are compared, they are compared for referential equality. They are equal if they refer to the same location in memory.

```
// int is a value type, so == uses value equality:
int num1 = 9;
int num2 = 9;
Console.WriteLine(num1 == num2);
// Prints true

// All classes are reference types, so == uses reference equality:
WorldCupTeam japan = new WorldCupTeam(2018);
```

```
WorldCupTeam brazil = new WorldCupTeam(2018);
Console.WriteLine(japan == brazil);
// Prints false
// This is because japan and brazil refer to two different locations in
memory (even though they contain objects with the same values):
```

C# Override

In C#, the override modifier allows base class references to a derived object to access derived methods.

In other words: If a derived class overrides a member of its base class, then the overridden version can be accessed by derived references AND base references.

```
// In the below example, DerivedClass.Method1() overrides
BaseClass.Method1(). bcdc is a BaseClass-type reference to a DerivedClass
value. Calling bcdc.Method1() invokes DerivedClass.Method1().

class MainClass {
  public static void Main (string[] args) {
    BaseClass bc = new BaseClass();
    DerivedClass dc = new DerivedClass();
    BaseClass bcdc = new DerivedClass();

    bc.Method1();
    dc.Method1();
    bcdc.Method1();
  }
}

class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived - Method1");
    }
}

// The above code produces this result:
// Base - Method1
// Derived - Method1
// Derived - Method1

// If we wanted bcdc.Method1() to invoked BaseClass.Method1(), then we
```

```
would label DerivedClass.Method1() as new, not override.
```

C# String Comparison

In C#, string is a reference type but it can be compared by value using ==.

```
//In this example, even if s and t are not referentially equal, they are
equal by value:
string s = "hello";
string t = "hello";

// b is true
bool b = (s == t);
```

C# String Types Immutable

In C#, string types are immutable, which means they cannot be changed after they are created.

```
// Two examples demonstrating how immutablility determines string
behavior. In both examples, changing one string variable will not affect
other variables that originally shared that value.

//EXAMPLE 1
string a = "Hello?";
string b = a;
b = "HELLLLLLLO!!!!";

Console.WriteLine(b);
// Prints "HELLLLLLLO!!!!"

Console.WriteLine(a);
// Prints "Hello?"


//EXAMPLE 2
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
// Prints "Hello "
```

C# Empty String

In C#, a string reference can refer to an empty string with "" and String.Empty.

This is separate from null and unassigned references, which are also possible for string types.

```
// Empty string:
string s1 = "";

// Also empty string:
string s2 = String.Empty;

// This prints true:
Console.WriteLine(s1 == s2);

// Unassigned:
string s3;

// Null:
string s4 = null;
```

C# Object Class

In C#, the base class of all types is the Object class. Every class implicitly inherits this class.

When you create a class with no inheritance, C# implicitly makes it inherit from Object.

```
// When you write this code:
class Dog {}
// C# assumes you mean:
class Dog : Object {}

//Even if your class explicitly inherits from a class that is NOT an
Object, then some class in its class hierachy will inherit from Object. In
the below example, Dog inherits from Pet, which inherits from Animal,
which inherits from Object:
class Dog : Pet {}
class Pet : Animal {}
class Animal {}

//Since every class inherits from Object, any instance of a class can be
referred to as an Object.
Dog puppy = new Dog();
Object o = puppy;
```

C# Object Class Methods

In C#, the Object class includes definitions for these methods: ToString(), Equals(Object), and GetType().

```
Object obj = new Object();
Console.WriteLine(obj.ToString());
// The example displays the following output:
//       System.Object

public static void Main()
{
    MyBaseClass myBase = new MyBaseClass();
  MyDerivedClass myDerived = new MyDerivedClass();
  object o = myDerived;
  MyBaseClass b = myDerived;

    Console.WriteLine("mybase: Type is {0}", myBase.GetType());
  Console.WriteLine("myDerived: Type is {0}", myDerived.GetType());
  Console.WriteLine("object o = myDerived: Type is {0}", o.GetType());
  Console.WriteLine("MyBaseClass b = myDerived: Type is {0}",
b.GetType());
}

// The example displays the following output:
//    mybase: Type is MyBaseClass
//    myDerived: Type is MyDerivedClass
//    object o = myDerived: Type is MyDerivedClass
//    MyBaseClass b = myDerived: Type is MyDerivedClass
```

C# ToString() Method

When a non-string object is printed to the console with Console.WriteLine(), its ToString() method is called.

```
Random r = new Random();

// These two lines are equivalent:
Console.WriteLine(r);
Console.WriteLine(r.ToString());
```

Lists and LINQ

Lists in C#

In C#, a list is a generic data structure that can hold any type. Use the new operator and declare the element type in the angle brackets < >.

In the example code, names is a list containing string values. someObjects is a list containing Object instances.

```
List<string> names = new List<string>();
List<Object> someObjects = new List<Object>();
```

Generic Collections

Some collections, like lists and dictionaries, can be associated with various types. Instead of defining a unique class for each possible type, we define them with a generic type T, e.g. List.

These collections are called generic collection types. They are available in the System.Collections.Generic namespace.

The generic type T will often show up in documentation. When using a generic collection in your code, the actual type is specified when the collection is declared or instantiated.

```
using System.Collections.Generic;

List<string> names = new List<string>();
List<Object> objs = new List<Object>();
Dictionary<string,int> scores = new Dictionary<string, int>();
```

Limitless Lists

Unlike a C# array, a C# list does not have a limited number of elements. You can add as many items as you like.

```
// Initialize array with length 2
string[] citiesArray = new string[2];
citiesArray[0] = "Los Angeles";
citiesArray[1] = "New York City";
citiesArray[2] = "Dubai"; // Error!

// Initialize list; no length needed
List<string> citiesList = new List<string>();
citiesList.Add("Los Angeles");
citiesList.Add("New York City");
citiesList.Add("Dubai");
```

Count Property

```
List<string> citiesList = new List<string>();
citiesList.Add("Los Angeles");
Console.WriteLine(citiesList.Count);
// Output: 1

citiesList.Add("New York City");
```

```
Console.WriteLine(citiesList.Count);
// Output: 2

citiesList.Remove("Los Angeles");
Console.WriteLine(citiesList.Count);
// Output: 1
```

Contains()

In C#, the list method Contains() returns true if its argument exists in the list; otherwise, false.

In the example code, the first call to Contains() returns true because “New York City” is in the list. The second call returns false because “Cairo” is not in the list.

```
List<string> citiesList = new List<string> { "Los Angeles", "New York
City", "Dubai" };

result1 = citiesList.Contains("New York City");
// result1 is true

result2 = citiesList.Contains("Cairo");
// result2 is false
```

LINQ

LINQ is a set of language and framework features for writing queries on collection types. It is useful for selecting, accessing, and transforming data in a dataset.

Using LINQ

LINQ features can be used in a C# program by importing the System.Linq namespace.

```
using System.Linq;
```

var

Since the type of an executed LINQ query’s result is not always known, it is common to store the result in an implicitly typed variable using the keyword var.

```
var custQuery = from cust in customers
                where cust.City == "Phoenix"
                select new { cust.Name, cust.Phone };
```

Method & Query Syntax

```
// Method syntax
var custQuery2 = customers.Where(cust => cust.City == "London");

// Query syntax
var custQuery =
    from cust in customers
    where cust.City == "London"
    select cust;
```

Where

In LINQ queries, the Where operator is used to select certain elements from a sequence.

It expects an expression that evaluates to a boolean value. Every element satisfying the condition will be included in the resulting query. It can be used in both method syntax and query syntax.

```
List<Customer> customers = new List<Customer>
{
  new Customer("Bartleby", "London"),
  new Customer("Benjamin", "Philadelphia"),
  new Customer("Michelle", "Busan" )
};

// Query syntax
var custQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

// Method syntax
var custQuery2 = customers.Where(cust => cust.City == "London");

// Result: Customer("Bartleby", "London")
```

From

In LINQ queries, the from operator declares a range variable that is used to traverse the sequence. It is only used in query syntax.

In the example code, n represents each element in names. The returned query only contains those elements for which n.Contains("a") is true.

```
string[] names = { "Hansel", "Gretel", "Helga", "Gus" };

var query =
  from n in names
  where n.Contains("a")
```

```
  select n;

// Result: Hansel, Helga
```

Select

In LINQ queries, the Select operator determines what is returned for each element in the resulting query. It can be used in both method and query syntax.

```
string[] trees = { "Elm", "Banyon", "Rubber" };

// Query syntax
var treeQuery =
  from t in trees
  select t.ToUpper();

// Method syntax
var treeQuery2 = names.Select(t => t.ToUpper());

// Result: ELM, BANYON, RUBBER
```

LINQ & foreach

You can use a foreach loop to iterate over the result of an executed LINQ query.

In the example code, query is the result of a LINQ query, and it can be iterated over using foreach. name represents each element in names.

```
string[] names = { "Hansel", "Gretel", "Helga", "Gus" };

var query = names.Where(n => n.Contains("a"));

foreach (var name in query)
{
  Console.WriteLine(name);
}
```

Count()

The result of an executed LINQ query has a method Count(), which returns the number of elements it contains.

In the example code, Count() returns 2 because the resulting query contains 2 elements containing “a”.

```
string[] names = { "Hansel", "Gretel", "Helga", "Gus" };
```

```
var query = names.Where(x => x.Contains("a"));

Console.WriteLine(query.Count());
// Output: 2
```

Object Initialization

Values can be provided to a List when it is constructed in a process called object initialization.

Instead of parentheses, use curly braces after the list's type.

Note that this can ONLY be used at the time of construction.

```
List<string> cities = new List<string> { "Los Angeles", "New York City",
"Dubai" };
```

Remove()

Elements of a list can be removed with the Remove() method. The method returns true if the item is successfully removed; otherwise, false.

In the example code, attempting to remove "Cairo" returns false because that element is not in the citiesList.

```
List<string> citiesList = new List<string>();
citiesList.Add("Los Angeles");
citiesList.Add("New York City");
citiesList.Add("Dubai");

result1 = citiesList.Remove("New York City");
// result1 is true

result2 = citiesList.Remove("Cairo");
// result2 is false
```

Clear()

All elements of a list can be removed with the Clear() method. It returns nothing.

In the example code, the list is initialized with three items. After calling Clear(), there are zero items in the list.

```
List<string> citiesList = new List<string> { "Delhi", "Los Angeles",
"Kiev" };
citiesList.Clear();
```

```
Console.WriteLine(citiesList.Count);
// Output: 0
```

Clear()

```
List<string> citiesList = new List<string> { "Delhi", "Los Angeles",
"Kiev" };
citiesList.Clear();

Console.WriteLine(citiesList.Count);
// Output: 0
```

List Ranges

Unlike elements in a C# array, multiple elements of a C# list can be accessed, added, or removed simultaneously. A group of multiple, sequential elements within a list is called a range.

Some common range-related methods are:

AddRange() InsertRange() RemoveRange()

```
string[] african = new string[] { "Cairo", "Johannesburg" };
string[] asian = new string[] { "Delhi", "Seoul" };
List<string> citiesList = new List<string>();

// Add two cities to the list
citiesList.AddRange(african);
// List: "Cairo", "Johannesburg"

// Add two cities to the front of the list
citiesList.InsertRange(0, asian);
// List: "Delhi", "Seoul", "Cairo", "Johannesburg"

// Remove the second and third cities from the list
citiesList.RemoveRange(1, 2);
// List: "Delhi", "Johannesburg"
```