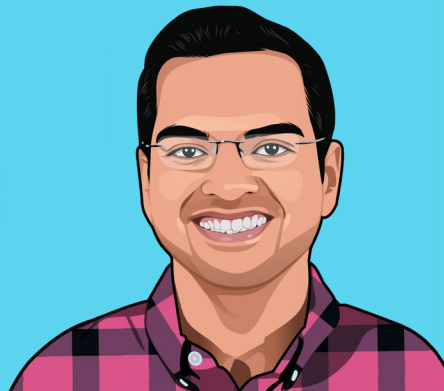
.NET Core



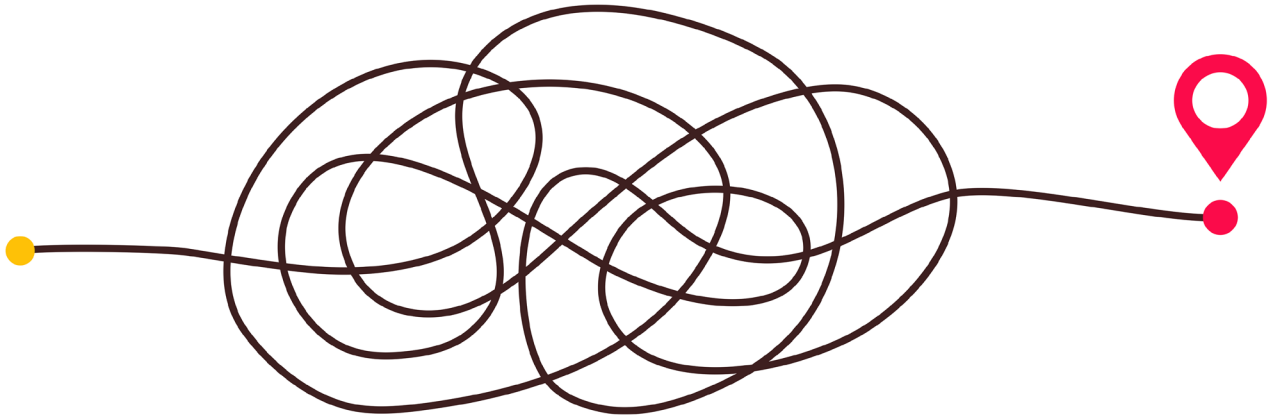
DOTNETMASTERY

The ultimate learning recourse for .NET developers.

- What is this course about?

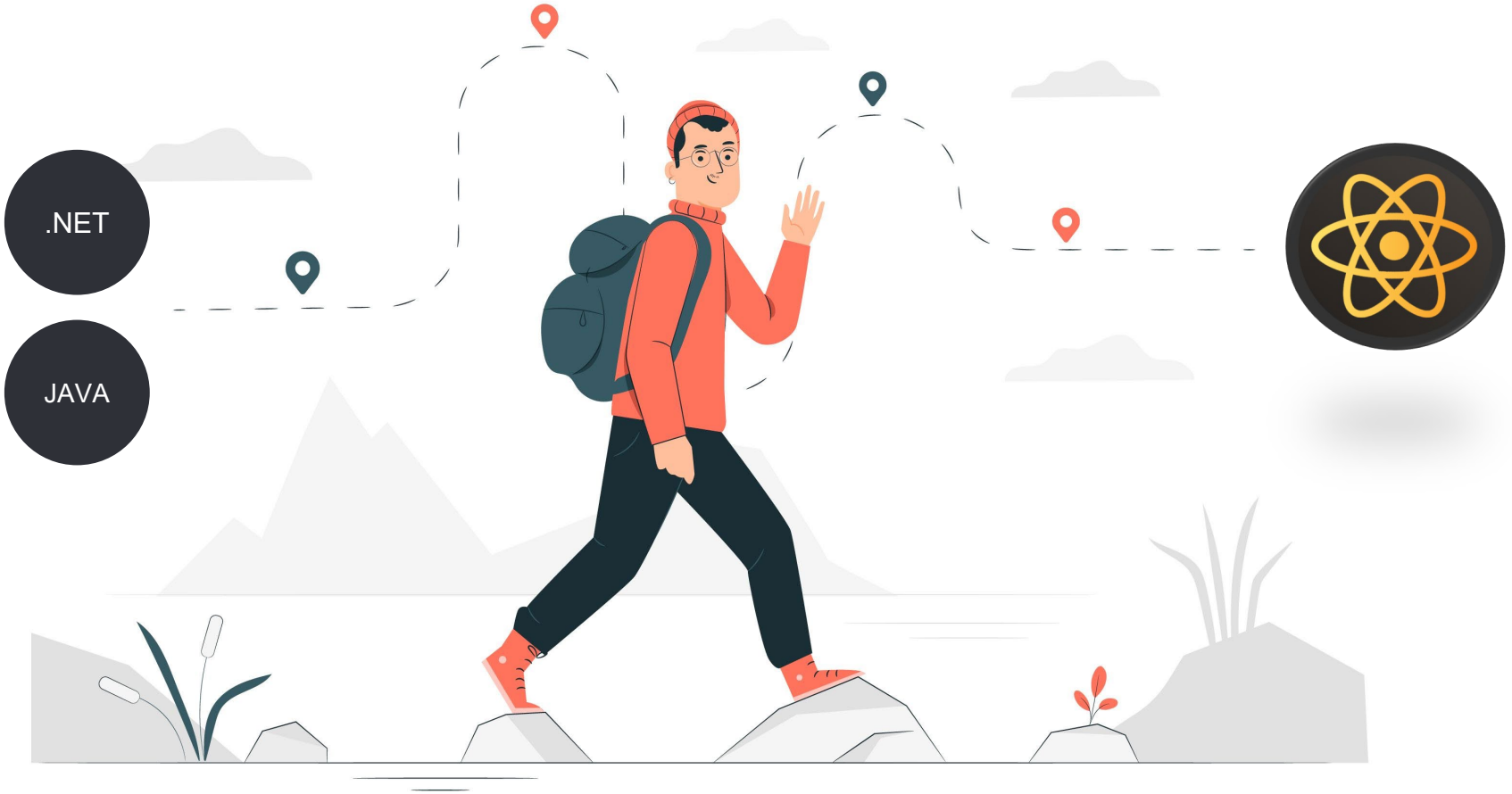
- Why this course?

My learning curve!



NOT! Your Journey

Your Journey!


.NET
JAVA

Best Practices

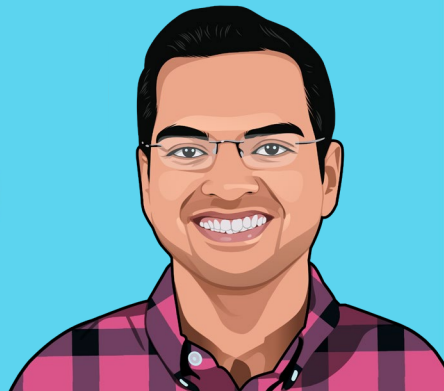
Real World Examples

.NET Core

9 Projects

25 Assignments

.NET Core



DOTNETMASTERY

The ultimate learning recourse for .NET developers.

What is React?

What is React?

- Create React App

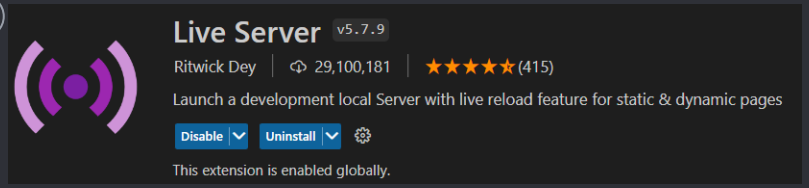
What is React?

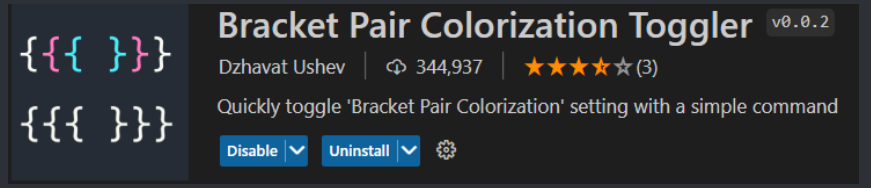
- React is Front End Javascript Framework
- React is Declarative
- React is Composable / Component based
- React is FAST!
- Maintained by META
- React is responsible for maintaining/managing the UI Aspect.

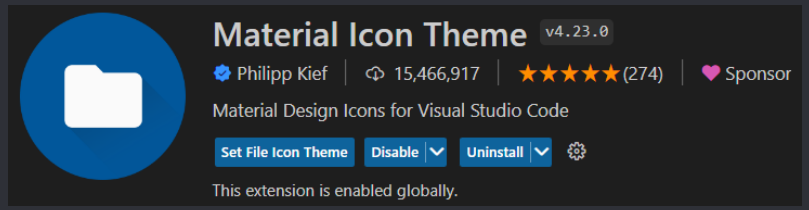
Tools Needed

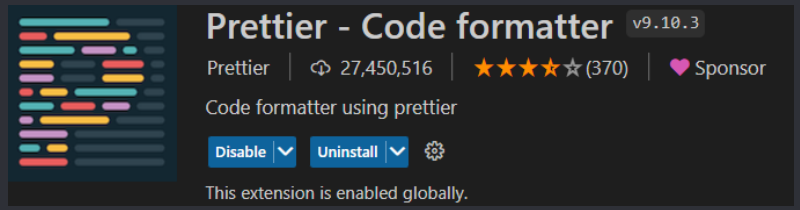
- Visual Studio Code
- Node JS
- Visual Studio 2022
- SQL Server Management Studio
- Azure Subscription

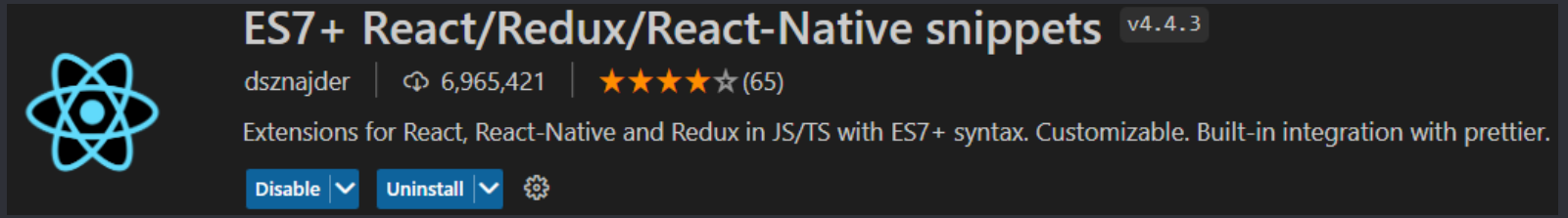
Visual Studio Code Extensions











- How to get Help?

Section 2

Section 3

Section 4

Section 5

Section 6

Section 7

Section 2

Section 2

Start json-server

- npx json-server --watch data/traveldb.json --port 5001

Section 2

Class vs Functional Components

Class Components

- Less common
- render() method is required, which returns an HTML element
- Stateful components
- Class components have lifecycle methods

Functional Components

- More common
- render() method is not present, it directly returns HTML element or nothing.
- Stateless components
- Class components do have lifecycle methods
- Hooks!

With React version 16.8, **class components have taken a back seat to functional components**. Functional components are more concise, leading to cleaner, less complex code. They don't include lifecycle methods or inherited members that may or may not be required for code functionality.

Anything that can be done using class components can also be done using functional components. The only exception is that React supports a special class component called **Error Boundaries** that can't be duplicated as a function component.

Because class components extend from React.Component, they have state and lifecycle methods associated with them. Their presence requires you to better understand when lifecycle events will occur and how to respond to them to manage state. Classes also require additional setup for the component or to make API calls for data, primarily implemented via the constructor. Sharing logic between multiple class objects without implementing design patterns is more difficult, leading to code that is more complex and difficult to maintain.

Deciding which component to use always devolves into a discussion of **inheritance vs composition**. Using functional components encourages composition, while class components lend themselves to inheritance design patterns. Currently, composition is considered a best practice in programming, which is why most new React code uses functional components instead of class components. That said, React still supports class components for legacy purposes.

JSX Components

- Component Names must be Pascal Casing

```
function Footer() {
```

```
<Footer />
```

- Props must be camel casing

```
<input maxLength={5} readOnly="false">
```

- Inline style is an object so it must be enclosed in double curly brackets

```
<p style={{ color: "gray", backgroundColor: "black" }}>Happy Coding!</p>
```

- Integer in props will go in {} and “” will be string , and bool goes in {}

```
<input maxLength={5} readOnly={false} placeholder="Ben"></input>
```

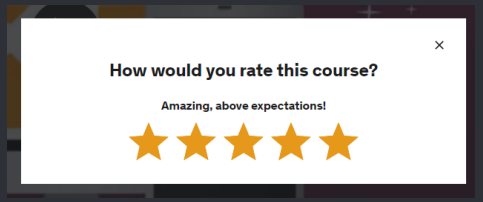
- ‘class’ needs to be replaced with ‘className’

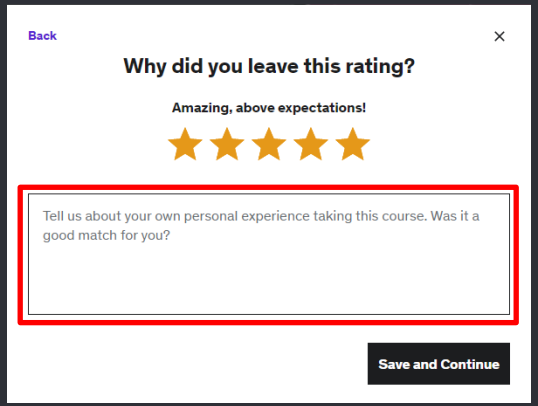
```
<h1 className="heading1">
```

RTK Query

- Queries – used to fetch data (Get/ GetALL)
- Mutation – used to update data (Create/ Update/ Delete)

Reviews



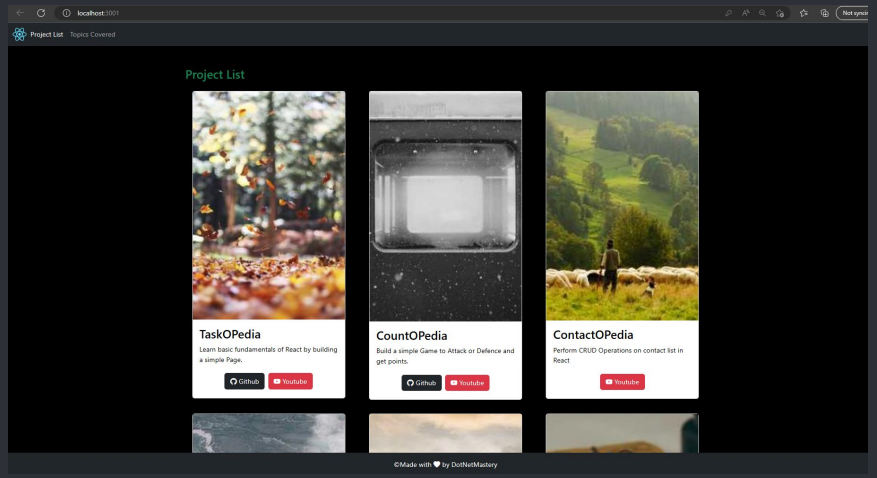


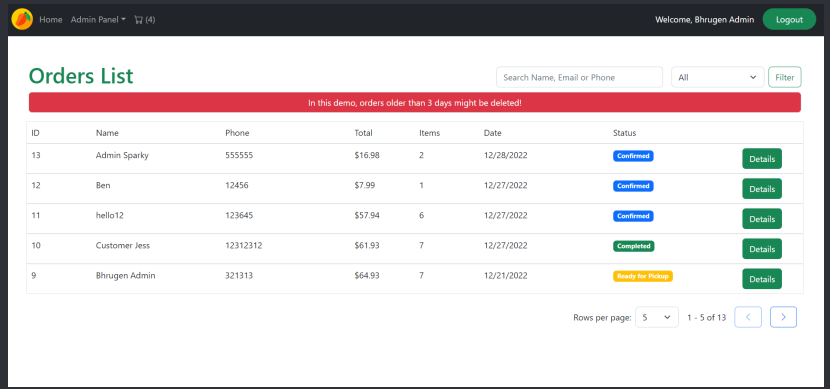
dotnetmastery@gmail.com

Feedback

dotnetmastery@gmail.com

Bonus Content





My code doesn't work!

- Google it!
- Check Q&A Section
- POST a queston (with Git branch)

Code works, but I am confused

- Google it!
- Message me on udemy!

My code doesn't work!

- Google it!
- Check Q&A Section
- POST a queston (with Git branch)

Code works, but I am confused

- Google it!
- Message me on udemy!

Prerequisites

- Basic programming experience
- HTML / CSS / JS Familiarity
- .NET API CRUD Operations (.NET Core & EF Core)

- Reviews

- Azure Deployment

Project Resources

- JSX

Projects Built

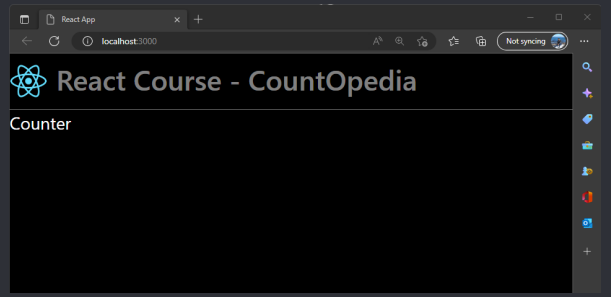
1. TaskOPedia -
2. CountOPedia
3. ContactOPedia
4. LifeCyclOPedia
5. WatchOPedia
6. RouteOPedia
7. ReduxOPedia
8. TravelOPedia
9. Red Mango

What we will learn

1. React Fundamentals
2. TypeScript
3. Redux Toolkit & RTK Query
4. Routing in Redux
5. Axios
6. Stripe Payment in React
7. Authentication and Authorization in React
8. File Management in React

Assignment 7 – CountOPedia Setup

1. Remove all files other than index.html, index.js and react Image
2. Move react image to src/images folder. (Create images folder)
3. Clean index.js and index.html (Remove anything that is not used)
4. Add Bootstrap js/css CDN inside index.html
5. Create 2 components (Design to look as shown in image)
    - Header.jsx (Functional Component)
    - Counter.jsx (Class Component)
6. Copy the images provided in course snippet section 4 to the images folder

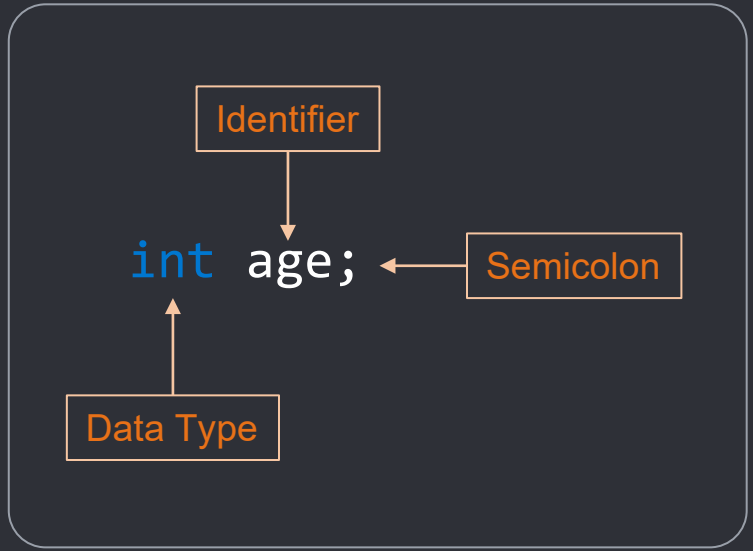


Class Component Lifecycle Methods

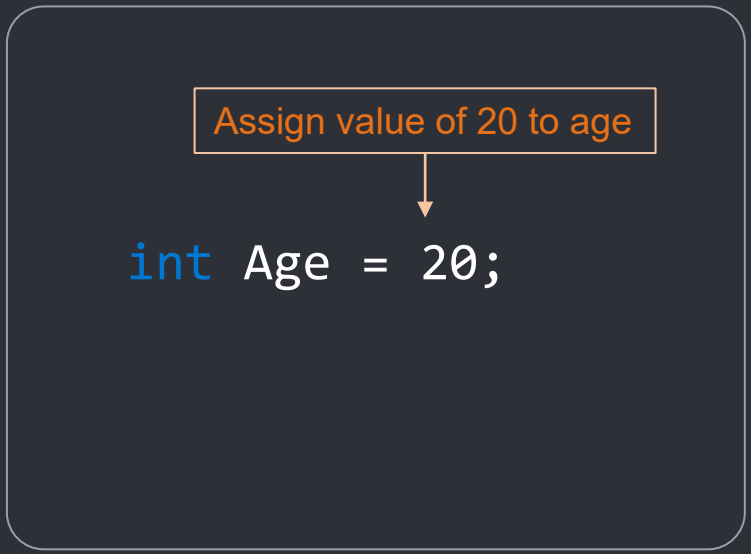
1. ComponentDidMount
   - Runs after the component has been rendered to DOM
2. ComponentDidUpdate
   - Runs after a component is updated.
   - It is not executed when component is rendered for the first time.
3. ComponentWillUnmount
   - Runs after a component is unmounted.
   - Ideally used for cleanup operations

DataTypes

Declare Variable



Declare & Assign Variable



Data Types

Byte

- 8-bit unsigned integer
- 0 to 255 (whole numbers)

```
byte num=100;
```

Short

- 16-bit unsigned integer
- -32,767 to 32,767 (whole numbers)

```
short num=10000;
```

Integer

- 32-bit unsigned integer (whole number)
- -2,147,483,648 to 2,147,483,647

```
int num=100000000;
```

Long

- 64-bit unsigned integer (whole number)
- -9,223,372,036,854,775,808
  to
  9,223,372,036,854,775,807

```
long num=10000000000000000;
```

Data Types

Float

- ±3.4 x 10^{38}
- ~ 6-9 digits precision

```
Float num=100.01f;
```

Double

- ±1.7 × 10^{308}
- ~ 15-17 digits precision

```
double num=1.999980099;
```

Decimal

- ±7.9228 x 10^{28}
- 28-29 digits precision

```
decimal num=1.00920382;
```

```
Max Decimal : 79228162514264337593543950335
Min Decimal : -79228162514264337593543950335
Max Float : 3.4028235E+38
Min Float : -3.4028235E+38
Max Double : 1.7976931348623157E+308
Min Double : -1.7976931348623157E+308
```

```
Result of 1/3:

float: 0.3333333
double: 0.3333333333333333
decimal: 0.3333333333333333333333333333
```

Float vs Double vs Decimal

- Float 32 bit (7 digits) It is used mostly in graphic libraries because very high demands for processing powers, also used situations that can tolerate rounding errors.
- Double 64 bit (15-16 digits) Double Types are probably the most commonly used data type for real values. Use double for non-integer math where the most precise answer isn't necessary.
- Decimal 128 bit (28-29 significant digits) It gives you a high level of accuracy because of which it is used in financial application.

```
float flt = 1F/3;
double dbl = 1D/3;
decimal dcm = 1M/3;
Console.WriteLine("float: {0} double: {1} decimal: {2}", flt, dbl, dcm);
```

output

```
float: 0.3333333
double: 0.3333333333333333
decimal: 0.3333333333333333333333333333
```

Data Types

Character

- 16-bit
- Single character literal or Unicode.

```
char temp'A';
```

String

- Multiple characters and numbers

```
string temp="Hello";
```

DataType Range

Type	Description	Range	Byte
byte	8-bit unsigned integer	0 to 255	1
short	16-bit signed integer	-32,768 to 32,767	2
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647	4
long	64-bit signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8
float	32-bit Single-precision floating point type	$\pm 3.4 \times 10^{38}$ ~ 6-9 digits precision	4
double	64-bit double-precision floating point 15-digit precision	$\pm 1.7 \times 10^{308}$ ~ 15-17 digits precision	8
decimal	128-bit decimal type	$\pm 7.9228 \times 10^{28}$ 28-29 digits precision	16
char	16-bit single Unicode character	Any valid character	2
bool	8-bit logical true/false value	True or False	1

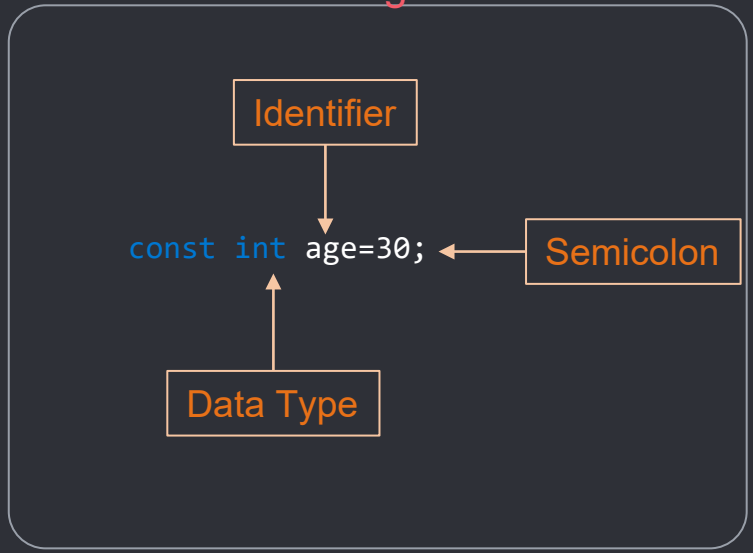
Constants

Constants are immutable values which are known at compile time and do not change for the life of the program

```
const <data_type> <constant_name> = value;
```

Constants

Declare & Assign Constant



Naming Convention

- Give your variables meaningful names!
- The name can contain letters, digits, and the underscore character (_)
- The first character of the name must be a letter. The underscore is also a legal first character, but its use is not recommended at the beginning of a name.
- Case matters (that is, upper- and lowercase letters). C# is case-sensitive; thus, the names "age" and "Age" refer to two different variables.
- C# keywords can't be used as variable names.
- Camel Casing is the recommended approach for naming variables.
  e.g. thisIsTheVariableName, lowerCase

Type Conversion

- Implicit type conversion
  No special syntax is required because the conversion always succeeds, and no data will be lost

- Explicit type conversion
  Explicit conversions require a cast expression.
  There is change for data loss!

Implicit Type Conversion

```
int num1 = 100;

double dou1 = num1;

long lng1=num1;
```

Explicit Type Conversion

```
double dou1 = 100;

int num1 = dou1;      // gives error
```

Explicit Type Conversion

```
double dou1 = 100;

int num1 = (int)dou1;      // num 1 = 100
```

Explicit Type Conversion – Data Loss!

```
double doub1 = 900800700600500400;

// there will be data loss here!
int intNum = (int) doub1;
```

Explicit Type Conversion (For Non -compatible Types)

```
string age = "10";

int ageValue = (int)age;     // gives error
```

Explicit Type Conversion (For Non -compatible Types)

```
string age = "10";

int ageValue = Convert.ToInt32(age);
int ageVal = int.Parse(age);
```

Explicit Type Conversion (For Non -compatible Types)

```
Convert.ToInt16();
Convert.ToInt32();
Convert.ToInt64();
Convert.ToDouble();
Convert.ToFloat();
Convert.ToChar();
Convert.ToBoolean();
```

C# Operators

- Assignment Operator
- Arithmetic Operators
- Comparison Operators
- Logical Operators

Assignment Operator

Basic assignment operator (=) is used to assign values to variables

```
int num;
num = 100;

int num1 = 1000;
```

Arithmetic Operator

Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, multiplication, division, etc.

```
int num1 = 10;
int num2 = 20;
int num3 = num1 + num2 // num3 = 30
```

Operators

Arithmetic Operator

Operator	Operator Name	Example
+	Addition Operator	20 + 10 evaluates to 30
-	Subtraction Operator	20 - 10 evaluates to 10
*	Multiplication Operator	20 * 10 evaluates to 200
/	Division Operator	20 / 10 evaluates to 2
%	Modulo Operator (Remainder)	17 % 5 evaluates to 2

Arithmetic Operator (Increment/Decrement)

Operator	Operator Name	Example
++	Increment Operator	x++ (x=x+1)
--	Increment Operator	x-- (x=x-1)

Arithmetic Operators

Postfix Increment

- Increments value by 1
- ++ (x++)
- The value returned by x++ is the value of (x) before the increment takes place

```
int x = 10;
int y = x++;
```

x = 11, y = 10

Prefix Increment

- Increments value by 1
- ++ (++x)
- The value returned by ++x is the value of (x) after the increment takes place

```
int x = 10;
int y = ++x;
```

x = 11, y = 11

Postfix Decrement

- Decrements value by 1
- -- (x--)
- The value returned by x-- is the value of (x) before the decrement takes place

```
int x = 10;
int y = x--;
```

x = 9, y = 10

Prefix Decrement

- Decrements value by 1
- -- (--x)
- The value returned by --x is the value of (x) after the decrement takes place

```
int x = 10;
int y = --x;
```

x = 9, y = 9

Assignment Operator

Operator	Operator Name	Example	
=	Assignment	x=100	
++	Increment Operator	x++	x=x+1
--	Decrement Operator	x--	x=x-1
+=	Addition Assignment	x+=10	x=x+10
-=	Subtraction Assignment	x-=10	x=x-10
*=	Multiplication Assignment	x*/10	x=x*10
/=	Division Assignment	x/=10	x=x/10

Comparision Operator

Operator	Operator Name	Example
==	Equal	x==y
!=	Not Equal	x!=y
>	Greater than	x>y
<	Less than	x<y
>=	Greater than or equal to	x>=y
<=	Less than or equal to	x<=y

Comparision Operator

X	Y	Operation	Result
10	10	x==y	True
10	10	x!=y	False
10	10	x>y	False
10	10	x<y	False
10	10	x>=y	True
10	10	x<=y	True

X	Y	Operation	Result
5	10	x==y	False
5	10	x!=y	True
5	10	x>y	False
5	10	x<y	True
5	10	x>=y	False
5	10	x<=y	True

Logical Operator

Operator	Operator Name	Example
&&	And	x && y
||	Or	x || y
!	Not	!x

Logical Operator

Operator	Example	x=3, y=4	x=10,y=1	x=1,y=10
&&	x>5 && y<5	False && True False	True && True True	False && False False
||	x>5 || y<5	False || True True	True || True True	False || False False
!	!(x>5)	!(False) True	!(True) False	!(False) True

AND and OR Operators

x	y	x&&y	x||y	!x
true	true	true	true	false
true	false	false	true	false
False	true	false	true	true
False	false	false	false	true

Strings

A *string* is a series of characters that is used to represent text.

String's in C#

```
string firstName;
firstName = "Ben";
string lastName = "Spark";
string fullName =  firstName + lastName;
string greetingMessage = "Welcome" + fullName;
```

String's in C#

```
string firstName = "Ben";
string lastName = "Spark";

//String Concatination
string greetingMessage = "Welcome, " + firstName + lastName + ".";

//String Interpolation
string interpolMessage = $"Welcome, {firstName} {lastName}.";

//String Formatting (Composite)
string formatMessage = string.Format("Welcome, {0} {1}.", firstName, lastName);
```

Output : *Welcome, Ben Spark.*

String’s Methods in C#

Functions/methods of the *String* class are used to manipulate and perform different actions on a given string.

Length

```
//index             0123456
//length            1234567
string greeting = "Welcome";

//Gets the length of the string (It is not a method!)
int greetingLength = greeting.Length;  // Output -> 7
```

IndexOf

```
                              11111111
                    012345678901234567
string greeting = "Welcome, Ben Spark";

//Get the index of ‘ ’ (space)
// Output -> 8
// Gets the start index of the specified char
int indexOfSpace = greeting.IndexOf(' ');

//Get the index of ?
// Output -> -1
// Returns -1 if the char is not found.
int indexOfQue = greeting.IndexOf('?');
```

IndexOf returns the first index of the string/char.

```
// Output -> 9
greeting.IndexOf('B');
```

LastIndexOf

```
                          11111111
                01234567890123456
string greeting = "Welcome, Ben Spark";

//Get the last index of ' ' (space)
// Output -> 12
// Gets the start index of the specified char
int lastIndexOfSpace = greeting.LastIndexOf(' ');

//Get the last index of ?
// Output -> -1
// Returns -1 if the char is not found.
int lastIndexOfQue = greeting.LastIndexOf('?');
```

IndexOf returns the first index of the string/char.

IndexOf

```
                              11111111
                    012345678901234567
string greeting = "Welcome, Ben Spark";

//Get the index of Ben
// Output -> 9
// Gets the start index of the specified string
int indexOfBen = greeting.IndexOf("Ben");

//Get the index of Bill
// Output -> -1
// Returns -1 if the string is not found.
int indexOfBill = greeting.IndexOf("Bill");
```

IndexOf returns the first index of the string/char.

LastIndexOf

```
                              111111111122
                    012345678901234567890
string greeting = "Welcome, Ben Spark Ben";

//Get the last index of Ben
// Output -> 19
// Gets the start index of the specified string
int indexOfBen = greeting.LastIndexOf("Ben");

//Get the last index of Bill
// Output -> -1
// Returns -1 if the string is not found.
int indexOfBill = greeting.LastIndexOf("Bill");
```

SubString

```
                          11111111
                01234567890123456
string greeting = “Welcome, Ben Spark";

//Get all string after B
// Output -> Ben Spark
// Gets all the value from the specified index
string fullName = greeting.Substring(9);

//Get first 5 characters of the string
// Output -> Welco
//Specify the start index and number of characters we want from the start index
string firstFive = greeting.Substring(0,5);
```

String index starts from 0

greeting.Substring(9,3) -> “Ben”

ToUpper / ToLower

```
string firstName = "Ben";
string lastName = "Spark";

//Convert string to Upper Case
// Output -> Welcome, BEN.
string greetingMessageUpper = "Welcome, " + firstName.ToUpper() + ".";

//Convert string to Lower Case
// Output -> Welcome, ben.
string greetingMessageLower = "Welcome, " + firstName.ToLower() + ".";
```

ToUpper / ToLower

```
string firstName = "Ben";
string lastName = "Spark";

//Convert string to Upper Case
// Output -> Welcome, BEN.
string greetingMessageUpper = "Welcome, " + firstName.ToUpper() + ".";

//Convert string to Lower Case
// Output -> Welcome, ben.
string greetingMessageLower = "Welcome, " + firstName.ToLower() + ".";
```

Trim

```
string greeting = "   Welcome   ";

//Trim spaces from start and end of the string
greeting.Trim();  // Output -> “Welcome”

//Trim spaces from start of the string
greeting.TrimStart();  // Output -> “Welcome   ”

//Trim spaces from end of the string
greeting.TrimEnd();  // Output -> “   Welcome”
```

IsNullOrEmpty / IsNullOrWhiteSpace

`IsNullOrEmpty` - checks if a string is null or empty (contains no data).

`IsNullOrWhiteSpace` - checks if a string is null or empty or if a string has only whitespaces.

```
string greeting = "Welcome";
string greetingEmpty = "";
string greetingWhiteSpace = "    ";
string greetingNull = null;

Console.WriteLine(string.IsNullOrEmpty(greeting));  // Output -> False
Console.WriteLine(string.IsNullOrEmpty(greetingEmpty));  // Output -> True
Console.WriteLine(string.IsNullOrEmpty(greetingWhiteSpace));  // Output -> False
Console.WriteLine(string.IsNullOrEmpty(greetingNull));  // Output -> True
Console.WriteLine(string.IsNullOrWhiteSpace(greeting));  // Output -> False
Console.WriteLine(string.IsNullOrWhiteSpace(greetingEmpty));  // Output -> True
Console.WriteLine(string.IsNullOrWhiteSpace(greetingWhiteSpace));  // Output -> True
Console.WriteLine(string.IsNullOrWhiteSpace(greetingNull));  // Output -> True
```

Compare Strings

```
string firstName = "Bhrugen";
string lastName = "";

Console.WriteLine(firstName == lastName);  // Output -> False
Console.WriteLine(firstName.Equals(lastName));  // Output -> False

lastName = "Bhrugen";
Console.WriteLine(firstName == lastName);  // Output -> True
Console.WriteLine(firstName.Equals(lastName));  // Output -> True
```

String and Int Conversion

```
string age = "20";
string ageTemp = "Test";

int balance = (int)age;

int age1 = Convert.ToInt32(age);
int age2 = int.Parse(age);
int age3 = int.Parse(ageTemp);   //Throws Error



int age4;
int.TryParse(ageTemp,out age4);      //Does not throw error, returns 0
```

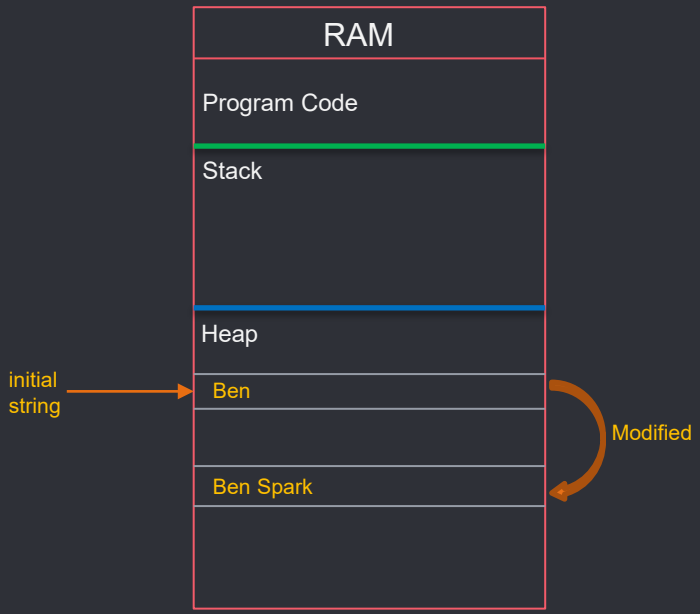
String and Int Conversion

```
string num = "20000";


string tempString1 = num.ToString();          // "20000"

string tempString2 = num.ToString("C");       // "$20,000.00"
```

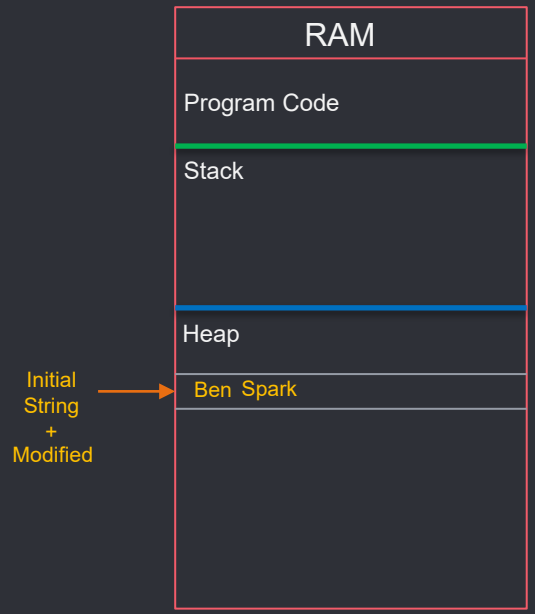
StringBuilder

String vs StringBuilder

```
string name = "Ben";      // “Ben”
name = name + " Spark";   // “Ben Spark”
//Strings are immutable
```




```
StringBuilder builder = new StringBuilder();
builder.Append("Ben");       // “Ben”
builder.Append(" Spark");    // “Ben Spark”
//StringBuilder are mutable
```



StringBuilder

- Defined in System.Text
- StringBuilder is Mutable.
- Recommended if you are modifying string too many times.
- StringBuilder does not contain methods available in string like
  - IndexOf()
  - Contains()
  - LastIndexOf()
- New methods in StringBuilder for manipulation
  - Append()
  - Remove()
  - Clear()

Method / Function

A *method/function* is a block of code that performs certain actions and only runs when it is called.

Methods / Functions in C#

- Methods are certain block of code that perform some action.
- Methods are only executed when they are called.
- Methods make maintenance of application easier.
- Methods are also called as functions in C#

```
<Access Specifier> <Return Type> <Method/Function Name>(Parameters)
{
    Method Body (Code for Method Logic)
}
```

Access Specifier – This determines the visibility of a variable or a method from another class.
Return type – The return type is the data type of the value that the method returns.
If the method is not returning any values, then the return type is **void**.
Method/Function name – This is a unique identifier(name) given to the method, and it is case sensitive.
It cannot be same as any other identifier declared in the class, since the method name must be unique.
Parameters – Parameters are used to pass and receive data from a method.
Parameters always go between round brackets. Parameters are optional and are not always needed.
Method body – All the logic of what the method will do, goes inside the method body.

Methods / Functions in C#

```
<Access Specifier> <Return Type> <Method/Function Name> (Parameters)
{
    Method Body (Code for Method Logic)
}
```

```
public void HelloWorld()
{
        Console.WriteLine("Hello World");
}
```

Call Method:

```
HelloWorld();
```

Methods / Functions in C#

```
<Access Specifier> <Return Type> <Method/Function Name> (Parameters)
{
    Method Body (Code for Method Logic)
}
```

```
public double Add(double number1, double number2)
{
        double total = number1 + number2;
        return total;
}
```

Call Method:

```
Add(10,20);
```

Conditional Flow

Conditional Flow in C#

- IF/ELSE statement
- SWITCH statements
- Ternary Operator

IF/ELSE statement

```
if (condition)
{
        //some code to be executed if condition is valid.
}
```

IF/ELSE statement

```
int marks = 99;
if (marks>90)    //Condition checks if marks is greater than 90
{
        Console.WriteLine("Your Grade - A");
}
```

IF/ELSE statement

```
if (condition)
{
        //some code to be executed if condition is valid.
}
else
{
        //some code to be executed if condition is NOT valid.
}
```

IF/ELSE statement

```
int marks = 99;
if (marks>90)    //Condition checks if marks is greater than 90
{
        Console.WriteLine("Your Grade - A");
}
else   //If condition is not valid
{
        Console.WriteLine("You missed A grade!");
}
```

Nested IF/ELSE statement

```
if (condition1)
{
        //some code to be executed if condition1 is valid.
}
else
{
        //some code to be executed if condition1 is NOT valid.
        if (condition2)
        {
                //executed if condition1 is NOT valid and consition2 is valid.
        }
        else
        {
                //some code to be executed if condition 1 and 2 is NOT valid.
        }
}
```

Nested IF/ELSE statement

```
int marks = 99;
if (marks>90)    //Condition checks if marks is greater than 90
{
        Console.WriteLine("Your Grade - A");
}
else   //If condition is not valid
{
        if(marks<33)  // marks <=90 AND marks <33
        {
                Console.WriteLine("You failed.");
        }
        else  // marks <=90 AND marks >=33
        {
                Console.WriteLine("You missed A grade!");
        }
}
```

SWITCH statement

```
switch(value)
{
            case 0:
                        … //do something
                        break;
            case 1:
                        … //do something
                        break;
            default:
                        … //do something
                        break;
}
```

SWITCH statement

```
int dayOfWeek = 1;    //0 – Sunday , 1 –Monday etc.

switch(dayOfWeek)
{
            case 0:
                        Console.WriteLine(“It’s Sunday”);
                        break;
            case 1:
                        Console.WriteLine(“It’s Monday”);
                        break;
                        …
                        …
            case 7:
                        Console.WriteLine(“It’s Saturday!”);
                        break;
            default:
                        Console.WriteLine(“Invalid Input!”);
                        break;
}
```

“Its Monday!”

SWITCH statement

```
int dayOfWeek = 100;    //0 – Sunday , 1 –Monday etc.

switch(dayOfWeek)
{
            case 0:
                        Console.WriteLine(“It’s Sunday”);
                        break;
            case 1:
                        Console.WriteLine(“It’s Monday”);
                        break;
                        …
                        …
            case 7:
                        Console.WriteLine(“It’s Saturday!”);
                        break;
            default:
                        Console.WriteLine(“Invalid Input!”);
                        break;
}
```

“Invalid Input!”

Loops

Loops in C#

- `while loop`
- `do while loop`
- `for loop`
- `foreach loop`

While Loop

```
while (condition)
{
        // code to be executed if condition is true
}
```

While Loop

```
int i=0;
while (i<3)
{
        Console.WriteLine(i);
        i++;
}
```

Output

```
0
1
2
```

Do-While Loop

```
do
{
        // code to be executed unless condition is false
        // checks condition after first execution
}
while (condition)
```

Do-While Loop

```
int i=0;
do
{
        Console.WriteLine(i);
        i++;
}
while (i<3)
```

Output

```
0
1
2
```

Do-While Loop

```
int i=100;
do
{
        Console.WriteLine(i);
        i++;
}
while (i<3)
```

Output

```
100
```

For Loop

```
for(statement 1; statement 2; statement 3)
{
        // code to be executed unless condition is false
}
```

- Statement 1 – Executed only once, before the execution of code block. Contains variable for condition.
- Statement 2 – Condition for executing code block
- Statement 3 – Executed every time after the code block, contains increment/decrement.

For Loop

Executed only once, before the execution of code block.
Contains variable for condition.

Condition until which loop is executed

Executed every time after the code block, contains increment/decrement

```
for(int i=0;i<3;i++)
{
        Console.WriteLine(i);
}
```

Output

```
0
1
2
```

Foreach Loop

```
foreach(statement)
{
        // code to be executed if the statement is valid
}
```

- Statement – This is a unique statement in which we iterate through a list/collection and for each item in that list we perform some operation.

Foreach Loop

```
string name=“Ben”;
foreach(char c in name)
{
        Console.WriteLine(c);
}
```

Output

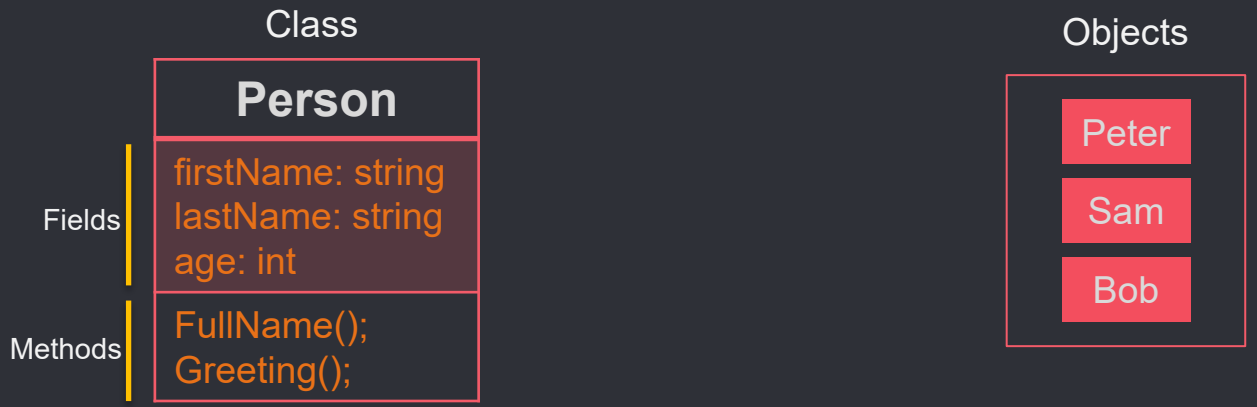
```
B
e
n
```

Classes

A class is like a blueprint of a specific object.

Classes in C#

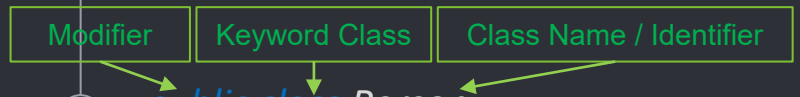
- ***Class and Object*** are the basic concepts of Object-Oriented Programming which revolve around the real-life entities.
- A class combines the fields(variables) and methods(functions) int a single unit.
- Core building block of any application.



Classes in C#

```
public class Person
{
    public string firstName;
    public string lastName;
    public int age;

    public void Greeting()
    {
        Console.WriteLine($“Welcome, {firstName} {lastName}”);
    }
    public string FullName()
    {
        return firstName + lastName;
    }
}
```

Declaring Class




```
public class Person
{
    public string firstName;
    public string lastName;
    public int age;

    public void Greeting()
    {
        Console.WriteLine($"Welcome, {firstName} {lastName}");
    }
    public string FullName()
    {
        return firstName + lastName;
    }
}
```

Class Body

Blue Print / Skeleton

Object

Basic unit of Object-Oriented Programming and represents the real-life entities.

Declaring Objects (Instantiating a class)

- In c#, **Object** is an instance of a **class** and that can be used to access the fields and functions of a **class**.
- Object is a runtime entity, it is created at runtime.

Object 1

```
Person  sam = new Person();
sam.firstName= “Sam”;
sam.lastName= “Patrick”;
sam.age= “25”;
sam.Greeting();  //Prints “Welcome, Sam Patrick”
```

instance

Class

```
public class Person
{
    public string firstName;
    public string lastName;
    public int age;

    public void Greeting()
    {
        Console.WriteLine($“Welcome, {firstName} {lastName}”);
    }
    public string FullName()
    {
        return firstName + lastName;
    }
}
```

instance

```
Person  ben = new Person();
sam.firstName= “Ben”;
sam.lastName= “Spark”;
sam.age= “20”;
sam.Greeting();  //Prints “Welcome, Ben Spark”
```

Object 2

Declaring Objects (Instantiating a class)

- In c#, **Object** is an instance of a **class** and that can be used to access the fields and functions of a **class**.
- Object is a runtime entity, it is created at runtime.

Object 1

```
Person  sam = new Person();
sam.firstName= “Sam”;
sam.lastName= “Patrick”;
sam.age= “25”;
sam.Greeting();  //Prints “Welcome, Sam Patrick”
```

instance

Class

```
public class Person
{
    public string firstName;
    public string lastName;
    public int age;

    public void Greeting()
    {
        Console.WriteLine($“Welcome, {firstName} {lastName}”);
    }
    public string FullName()
    {
        return firstName + lastName;
    }
}
```

instance

```
Person  ben = new Person();
sam.firstName= “Ben”;
sam.lastName= “Spark”;
sam.age= “20”;
sam.Greeting();  //Prints “Welcome, Ben Spark”
```

Object 2

Constructors

A constructor is a **special method** that is used to initialize objects.

Parameterless Constructor (Default)

- A constructor is a method whose name is the same as the name of its type
- If you don't provide a constructor for your class, C# creates one by default

```
public class Person
{
    public string firstName;
    public string lastName;
    public int age;

    public Person()
    {
        Console.WriteLine("Constructor Called");
    }
    public void Greeting()
    {
        Console.WriteLine($"Welcome, {firstName} {lastName}");
    }
    public string FullName()
    {
        return firstName + lastName;
    }
}
```

```
Person ben = new Person();
// outputs – "Constructor Called"
```

```
Person ben;
// No Output! Constructor is not called
```

```
Person ben;
Ben = new Person();
// outputs – "Constructor Called"
```

Parameterized Constructor

- A constructor is a method whose name is the same as the name of its type
- If you don't provide a constructor for your class, C# creates one by default

```
public class Person
{
    public string firstName;
    public string lastName;
    public int age;

    public Person(string firstNm, string lastNm)
    {
        this.firstName = firstNm;
        this.lastName=lastNm;
        Console.WriteLine("Parameter Constructor Called");
    }
    public void Greeting()
    {
        Console.WriteLine($"Welcome, {firstName} {lastName}");
    }
    public string FullName()
    {
        return firstName + lastName;
    }
}
```

```
Person ben = new Person("Ben","Spark");
// outputs – "Parameter Constructor Called"
```

```
Person ben;
// No Output! Constructor is not called
```

```
Person ben;
Ben = new Person();
// will not work! Since we do not have default constructor
```

Multiple Constructor

- A constructor is a method whose name is the same as the name of its type
- If you don't provide a constructor for your class, C# creates one by default

```
public class Person
{
    public string firstName;
    public string lastName;
    public int age;
    public Person(string firstName)
    {
        this.firstName = firstNm;
        Console.WriteLine("First Name Constructor Called");
    }
    public Person(string firstNm, string lastNm)
    {
        this.firstName = firstNm;
        this.lastName=lastNm;
        Console.WriteLine("Parameter Constructor Called");
    }
    public void Greeting()
    {
        Console.WriteLine($"Welcome, {firstName} {lastName}");
    }
    public string FullName()
    {
        return firstName + lastName;
    }
}
```

```
Person ben = new Person("Ben","Spark");
// outputs – "Parameter Constructor Called"
```

```
Person ben;
// No Output! Constructor is not called
```

```
Person ben;
Ben = new Person("Ben");
// outputs - "First Name Constructor Called"
```

Destructors / Finalizers

Finalizers (**destructors**) are used to perform any necessary final clean-up when the object is being destroyed.

Destructors / Finalizers

- Finalizers/Destructors are called at the very end when the object is being destroyed.
- Finalizers/Destructors cannot be overloaded.
- Finalizers/Destructors cannot be called. They are invoked automatically.
- Finalizers/Destructors cannot have parameters.

Destructor/Finalizer

```
public class Person
{
    public string firstName;
    public string lastName;
    public int age;
    public void Greeting()
    {
        Console.WriteLine($"Welcome, {firstName} {lastName}");
    }
    public string FullName()
    {
        return firstName + lastName;
    }
    ~Person()
    {
        Console.WriteLine("Destructor Called");
    }
}
```

```
Person ben = new Person("Ben","Spark");
// outputs – "Destructor Called"
```

```
Person ben;
// No Output!
```