

2015



C# programming basics

The logo features a stylized profile of a human head in grey, with a yellow brain inside. The brain is depicted with white lines representing neural connections.

BASIC PRINCIPLES OF C #, CLR

BRAIN
ACADEMY

Training program

- **Block 1.** C# programming fundamentals
- **Block 2.** Windows application development
- **Block 3.** Service-oriented and web application development
- **Block 4.** Application architecture and design patterns
- **Block 5.** Certification

Block 1 content

1. Basic principles of C#, CLR
2. Object oriented fundamentals
3. Exception handling
4. Advanced programming (Delegates, events, lambdas. Generics. Collections)
5. Assembly management and application debug
6. Multithreading and asynchronous processing
7. Data access
8. Unsafe code and pointers. .NET Framework security

Module contents

- Basic principles of C#, CLR
 - C# & CLR basics
 - Data types
 - Operators
 - Array, Structure, Enum
 - System.Console

Lecture contents

- Basic principles of C #, CLR. Data types
 - Principles of data storage and general information about data types
 - Data types (the primitive, reference and value types)
 - Literals and variables, constants
 - Using types
 - Scope of the variables

Lecture contents

- Basic principles of C #, CLR. Data types
 - Principles of data storage and general information about data types
 - Data types (the primitive, reference and value types)
 - Literals and variables, constants
 - Using types
 - Scope of the variables

Principles of data storage (1/4)

 What is a variable and what is it for?

- Where should data be stored which can be entered by the user or rigidly prescribed in the code of the program?
- What are the temporary data?

Principles of data storage (2/4)

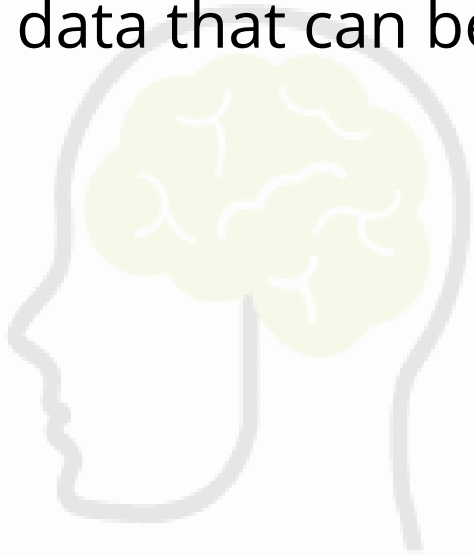
- Persistent data are stored on the hard drive and temporary data are in the computer's memory
- The temporary data are all that is necessary to the program for the calculations
- The processor is able to perform mathematical operations only on the processor registers or memory. Therefore, to make calculations on persistent data, they must be loaded into memory

Principles of data storage (3/4)

- The assembler uses addresses for the work with the parts of memory (slots) that stores data
- In the high-level programming languages (including C #), memory locations storing data (storage locations) are named
- **The variable is the name, which is used for storage location**
- The variable name is the usual way to reference the stored value

Principles of data storage (4/4)

 **Variable** - a named storage location capable of containing data that can be modified during program execution



BRAIN
ACADEMY

Lecture contents

- Basic principles of C #, CLR. Data types
 - Principles of data storage and general information about data types
 - Data types (the primitive, reference and value types)
 - Literals and variables, constants
 - Using types
 - Scope of the variables

General information about data types (1/2)



Type - in programming, the nature of a variable

- For example, integer, real number, text character, or floating point number
- Data types in programs are declared by the programmer and determine the range of values a variable can take as well as the operations that can be performed on it

General information about data types (2/2)

- **Data Types Are Important**

- C# is a strongly typed language. As a general rule, all operations are type-checked by the compiler for type compatibility.

- **Exception**

- Dynamic data type checking to be deferred until runtime, rather than occurring at compile time

Declaring variables

- All variables must be declared **prior** to their use
- Variables are declared using this form of statement:

`type var_name;`

- where type is the data type of the variable and var_name is its name

or

`type var_name_1, var_name_2, ... , var_name_n ;`

- where var_name_m, $m = 1$ to n , are their names

Naming of variables

- C# allows very flexible naming of variables and other symbols - the use of alphanumeric characters, a mixture of upper and lower case letters and the underscore symbol
- The only limitations when combining underscores, letters and numeric digits are that a name cannot start with a number and the name must not be a reserved word

Initializing a variable (1/2)

- giving it an initial value when it is declared
`type var_name = value;`
 - where the value must be compatible with the specified
- When you declare a variable in a program, you must either specify its type or use the **var** keyword to let the compiler infer the type (dynamic initialization later)



Initializing a variable (2/2)


Code example #1:


```
1. int cnt = 200;  
2. char ch_ltr = 'C';
```

Also it is possible use a comma-separated list


```
1. int n, m = 1, k = 10, j;
```

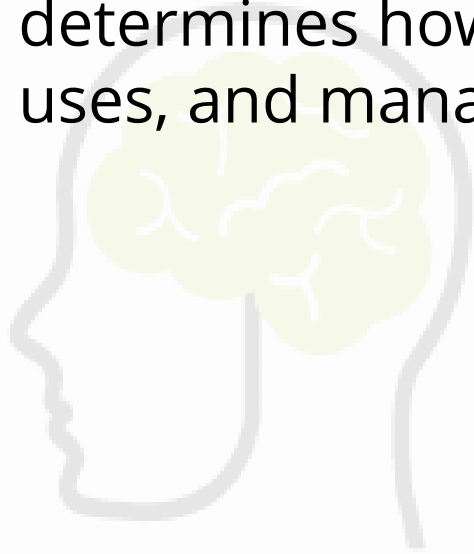
General information about data types (1/2)

 **Common Language Specification (CLS)** - a subset of language features supported by the common language runtime, including features common to several object-oriented programming languages

 **CLS-compliant** - pertaining to code that publicly exposes only language features that are in the Common Language Specification. CLS compliance can apply to classes, interfaces, components, and tools

General information about data types (2/2)

 **Common Type System (CTS)** - the specification that determines how the common language runtime defines, uses, and manages types



BRAIN
ACADEMY

CTS functions

- Establishes a framework that helps enable cross-language integration, type safety, and high-performance code execution
- Provides an **object-oriented model** that supports the complete implementation of many programming languages
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other
- Provides a **library** that contains the primitive data types used in application development


CTS – General Information (1/2)

- CTS supports the principle of **inheritance**
Types can derive from other types, called **base types**
- All types, including built-in numeric types such as `System.Int32` (C# keyword: `int`),
derive ultimately from a **single base type**
`System.Object` (C# keyword: `object`)

CTS – General Information (2/2)

- Each type in the CTS is defined as either a **value type** or a **reference type**. This includes all custom types in the .NET Framework class library and also your own user-defined types
- Types that you define by using the **struct** keyword are value types; all the built-in numeric types are **structs**
- Types that you define by using the **class** keyword are reference types

FCL General information

 **.NET Framework class library** - a library of classes, interfaces, and value types that are included in .NET Framework

- This library provides access to system functionality and is designed to be the foundation on which .NET Framework applications, components, and controls are built

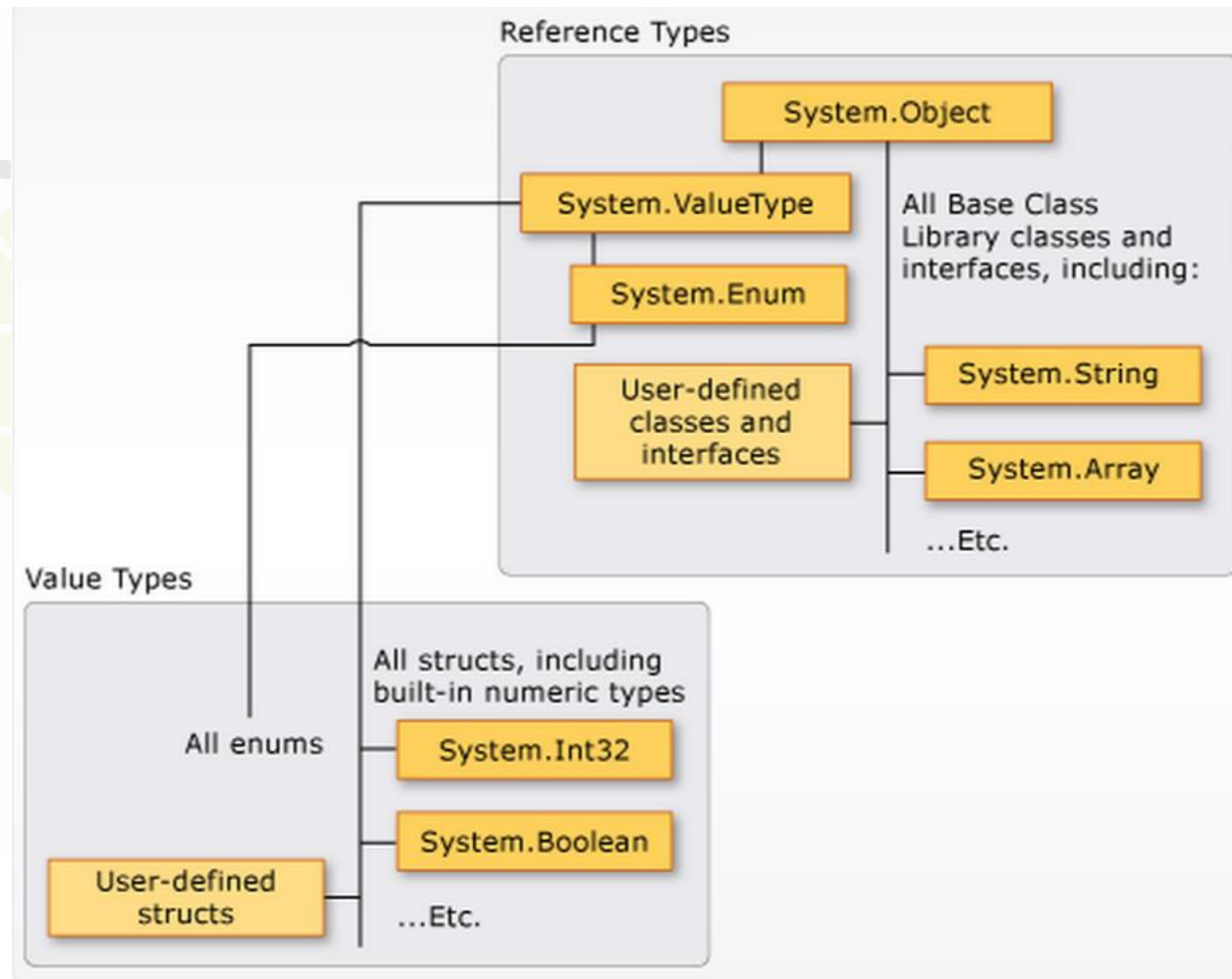
Variable vs. object



Are the variables the objects?

- If we take variables as **slots**, which either temporarily or permanently exist in certain areas of **memory**: heap, stack, CPU registers:
- Variable is an object if and only if the language provides the programmer with an object interface consisting of management operations of the variable slot: creating a new slot space of the slot values, removal slot, extract current values of the slot, etc.
- None of existing object-oriented programming language does **not provide** explicitly this **object interface** for its variables

Value and reference types relationship (1/2)

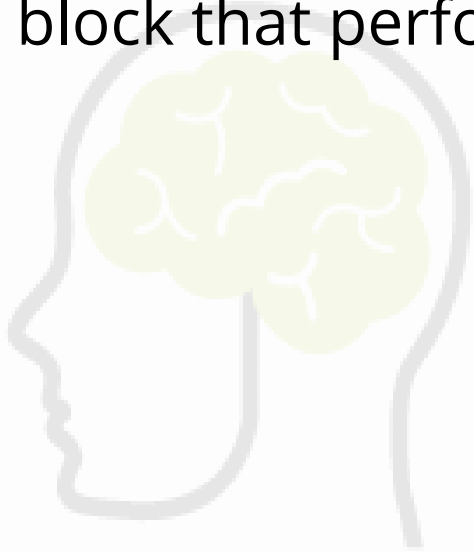


Value and reference types relationship (2/2)

Notice that each of these types ultimately derive from `System.Object`, which defines a set of methods (e.g., `ToString()`, `Equals()`, `GetHashCode()`) **common** to all types in the .NET base class libraries

Method

 **Method** - in object-oriented programming, a named code block that performs a task when called



BRAIN
ACADEMY

General information about data types



Value type - a data type that is represented by the type's actual value. If a value type is assigned to a variable, that variable is given a fresh copy of the value

- This is in contrast to a reference type, where assignment does not create a copy
- Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared

Simple types (1/5)

- Certain data types are so commonly used that many compilers allow code to manipulate them using simplified syntax
- There are 13 value types at the core of C# referred to as the **simple types** (or **primitive** types). They consist of a **single value**
- Primitive types **map directly to types** existing in the Framework Class Library (**FCL**)

Simple types (2/5)

- For the types that are compliant with the Common Language Specification (CLS), other languages will offer similar primitive types
- All value types are **sealed**, which prevents a value type from being used as a base type for any other reference type or value type
- It's not possible to define any new types using Boolean, Char, Int32, UInt64, Single, Double, Decimal, and so on as base types

Simple types (3/5)



Value type	Category	.NET Framework Type	Meaning	CLS-Compliant	Description	Range
bool	Boolean	System.Boolean	Represents true/false values	Yes	A true/false value	true, false
byte	Unsigned, numeric, integral	System.Byte	8-bit unsigned integer	Yes	Unsigned 8-bit value	0–255
char	Unsigned, numeric, integral	System.Char	Character	Yes	16-bit Unicode character (char never represents an 8-bit value as it would in unmanaged C++.)	0000–FFFF Unicode
decimal	Numeric, decimal	System.Decimal	Numeric type for financial calculations	Yes	A 128-bit high-precision floating-point value commonly used for financial calculations in which rounding errors can't be tolerated. Of the 128 bits, 1 bit represents the sign of the value, 96 bits represent the value itself, and 8 bits represent the power of 10 to divide the 96-bit value by (can be anywhere from 0 to 28). The remaining bits are unused.	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$

Simple types (4/5)



Value type	Category	.NET Framework Type	Meaning	CLS-Compliant	Description	Range
double	Numeric, floating-point	System.Double	Double-precision floating point	Yes	IEEE 64-bit floating point value	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
float	Numeric, floating-point	System.Single	Single-precision floating point	Yes	IEEE 32-bit floating point value	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
int	Signed, numeric, integral	System.Int32	Integer	Yes	Signed 32-bit value	-2,147,483,648 to 2,147,483,647
long	Signed, numeric, integral	System.Int64	Long integer	Yes	Signed 64-bit value	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Simple types (5/5)

Value type	Category	.NET Framework Type	Meaning	CLS-Compliant	Description	Range
sbyte	Signed, numeric, integral	System.SByte	8-bit signed integer	No	Signed 8-bit value	-128 to 127
short	Signed, numeric, integral	System.Int16	Short integer	Yes	Signed 16-bit value	-32,768 to 32,767
uint	Unsigned, numeric, integral	System.UInt32	An unsigned integer	No	Unsigned 32-bit value	0 to 4,294,967,295
ulong	Unsigned, numeric, integral	System.UInt64	An unsigned long integer	No	Unsigned 64-bit value	0 to 18,446,744,073,709,551,615
ushort	Unsigned, numeric, integral	System.UInt16	An unsigned short integer	No	Unsigned 16-bit value	0 to 65,535



Simple types code examples

Code example #2:

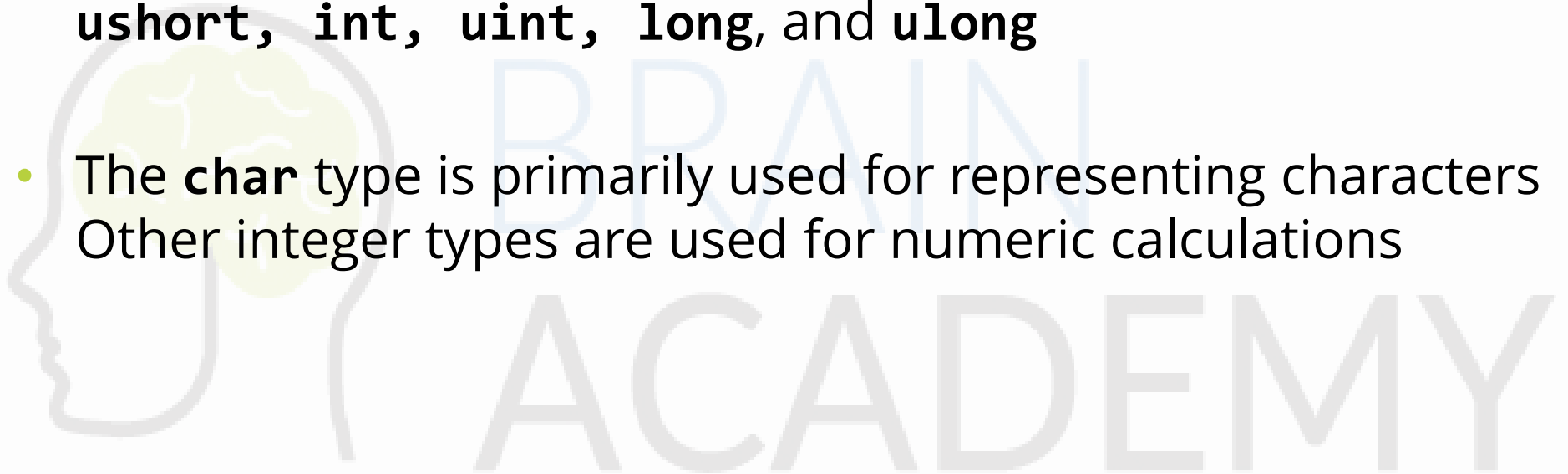
1. `int b = 0;`
2. `System.Int32 b = 0;`

Code example #3:

1. `//Using new to create variables`
2. `Console.WriteLine("Using new to create variables");`
3. `bool f = new bool();`
4. `int i = new int();`
5. `double d = new double();`
6. `Console.WriteLine("new bool, new int, new double");`
7. `Console.WriteLine("{0}, {1}, {2}", f, i, d);`
8. `Console.WriteLine();`

Integers

- There are nine integer types: **char**, **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, and **ulong**
- The **char** type is primarily used for representing characters
Other integer types are used for numeric calculations



FCL. System.Int32 Structure

- Some methods
 - **ToString()**
 - Converts the numeric value of this instance to its equivalent string representation
 - **Parse(String)**
 - Converts the string representation of a number to its 32-bit signed integer equivalent



Integers. Code example (1/2)

Code example #4:

```
1. // Use long
2. Console.WriteLine("Use integers");
3. long minutes;
4. long hours;
5. minutes = 5300;
6. //Our course in minutes
7. hours = minutes / 45;
8. Console.WriteLine("Our course in academic hours: " +
9. hours + " hours.");
10. Console.WriteLine("Our course in academic hours: " +
11. hours.ToString() + " hours (inherited).");
12. Console.WriteLine();
```



Integers. Code example (2/2)

Code example #5:

```
1. // Use byte, int
2. Console.WriteLine("Use byte, int");
3. byte y;
4. int sum;
5. sum = 0;
6. // No need control for y <=50
7. for (y = 1; y <= 50; y++)
8.     sum = sum + y;
9. Console.WriteLine("Summa from 1 to 50 " + sum);
10. Console.WriteLine();
```

Floating-point types (1/2)

- There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers
- Many of the math functions in C#'s class library use double values



Floating-point types (2/2)

Code example #6:

```
1. //Using floating-point types
2. Console.WriteLine("Use floating-point types");
3. Double r;
4. Double area;
5. area = 100.0;
6. r = Math.Sqrt(area / 3.1416);
7. Console.WriteLine("Radius is " + r);
8. Console.WriteLine();
```



Numerical Types. Code example

Code example #7:

```
1. //Additional properties
2. Console.WriteLine("Numericals max and min ");
3. Console.WriteLine("Max of int: {0}", int.MaxValue);
4. Console.WriteLine("Min of int: {0}", int.MinValue);
5. Console.WriteLine("Max of double: {0}", double.MaxValue);
6. Console.WriteLine("Min of double: {0}", double.MinValue);
7. Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
8. Console.WriteLine("double.PositiveInfinity: {0}",
    double.PositiveInfinity);
9. Console.WriteLine("double.NegativeInfinity: {0}",
    double.NegativeInfinity);
10. Console.WriteLine();
```

Decimal type (1/2)

- **Decimal** is the most interesting C# numeric type, which is intended for use in monetary calculations
- Normal floating-point arithmetic is subject to a variety of rounding errors when it is applied to decimal values
- The **decimal** type eliminates these errors and can accurately represent up to 28 decimal places (29 sometimes)



Decimal type (2/2)

Code example #8:

```
1. // Use decimal
2. Console.WriteLine("Use decimal");
3. decimal pay, credit;
4. pay = 110.1M;
5. credit = pay * 12.0M;
6. //Notice that the result is accurate to several decimal places
7. Console.WriteLine("Credit is " + credit);
8. Console.WriteLine();
```

Char type

- C# textual data which is represented by char keywords, is simple shorthand notations for System.Char (Unicode under the hood). The char can represent a single slot in a string (e.g., 'C')
- Using the methods of System.Char, you are able to determine whether a given character is numerical, alphabetical, a point of punctuation, etc.

General information. Character



Character:

- single-character
- simple-escape-sequence
- hexadecimal-escape-sequence
- unicode-escape-sequence

single-character:

Any character except ' (U+0027), \ (U+005C), and new-line-character

simple-escape-sequence: one of

`' \" \\ \0 \a \b \f \n \r \t \v`

(Please go to Notes for the table)

hexadecimal-escape-sequence:

- `\x hex-digit hex-digitopt hex-digitopt hex-digitopt`

FCL. Char structure

- Some methods
 - **IsDigit(Char)**
 - Indicates whether the specified Unicode character is categorized as a decimal digit
 - **IsSeparator(Char)**
 - Indicates whether the specified Unicode character is categorized as a separator character
 - **Parse**
 - Converts the value of the specified string to its equivalent Unicode character



Char type. Code example

Code example #9:

```
1. // Use char
2. char myChar = 'C';
3. Console.WriteLine("char.IsDigit('C'): {0}", char.IsDigit(myChar));
4. Console.WriteLine("char.IsLetter('C'): {0}",
    char.IsLetter(myChar));
5. Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
6. char.IsWhiteSpace("Hello There", 5));
7. Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
8. char.IsWhiteSpace("Hello There", 6));
9. Console.WriteLine("char.IsPunctuation('?'): {0}",
    char.IsPunctuation('?'));
10. Console.WriteLine();
```



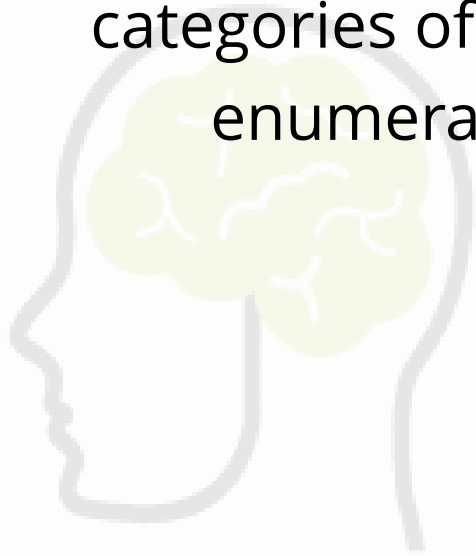

Boolean type

Code example #10:

```
1. Console.WriteLine("Use boolean"); // Use boolean
2. bool logic;
3. logic = false;
4. // Check the condition
5. if (9<10)
6. {
7.     logic = true;
8. }
9. Console.WriteLine("9<10 is " + logic);
10. Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
11. Console.WriteLine("bool.TrueString: {0}", bool.TrueString);
12. Console.WriteLine();
```

Other categories of value types

- In addition to the simple types, C# defines the other categories of value types:
 - enumerations (**enum**), structures (**struct**)



BRAIN
ACADEMY

Reference types

- Reference types are data types whose objects are represented by a **reference** (similar to a **pointer**) to the object's actual value
- If a reference type is assigned to a variable, that **variable references** (points to) **the original value**. No copy is made
- A reference type is an object allocated on the **garbage-collected managed heap**

The reference

 *Eric Lippert (C# compiler lead developer) :*

"... a reference is actually implemented as a small chunk of data which contains information used by the CLR to determine precisely which object is being referred to by the reference"

General information about GC



Garbage collector (GC) - the part of the operating system that performs garbage collection

- The garbage collector attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program

BRAIN
ACADEMY

Reference types (1/2)

- At run time, when you declare a variable of a reference type, the variable contains the value **null** until you explicitly create an instance of the object by using the new operator, or assign it an object that has been created elsewhere by using **new** :

```
Stud_Class uc = new Stud_Class();  
Stud_Class uc1 = uc;
```

Reference types (2/2)

- A type that is defined as a **class**, **delegate**, **array**, or **interface** is a reference type
- Reference types are always allocated from the **managed heap (a pool of memory managed by CLR)**, and the C# `new` operator returns the memory address of the object—the memory address refers to the object's bits

General information about null

 **null** - pertaining to a value that indicates missing or unknown data



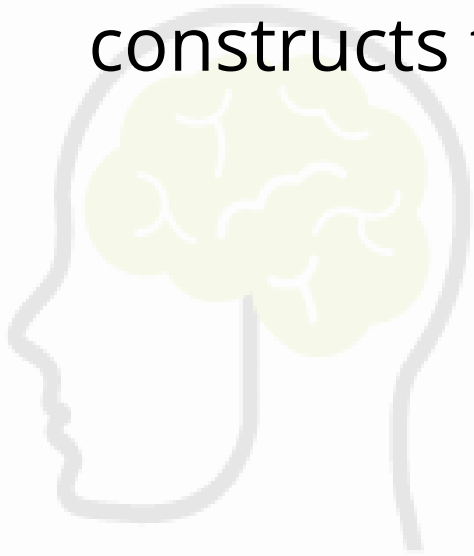
BRAIN
ACADEMY

Reference Types. Keywords

- The following keywords are used to declare reference types:
 - **class**
 - **interface**
 - **delegate**
- C# also provides the following built-in reference types:
 - **dynamic**
 - **object**
 - **string**

Custom types

- You use the **struct**, **class**, **interface** and **enum** constructs to create your own custom types




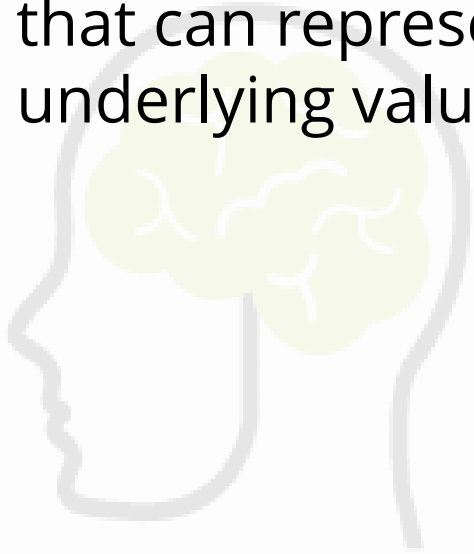
BRAIN
ACADEMY

Built-In types

- The keywords for built-in C# types, which are aliases of predefined types in the **System** namespace:
 - `bool`
 - `byte`
 - `sbyte`
 - `char`
 - `decimal`
 - `double`
 - `float`
 - `int`
 - `uint`
 - `long`
 - `ulong`
 - `object`
 - `short`
 - `ushort`
 - `string`

Nullable types (1/2)

 **Nullable type** - an instance of the System.Nullable struct that can represent the normal range of values for its underlying value type, plus an additional null value



BRAIN
ACADEMY

Nullable Types (2/2)

- Nullable types can represent all the values of an underlying type, and an **additional null value**
- Nullable types are declared like:

`System.Nullable<T> variable`

or

`T? variable`

T is the underlying type of the nullable type. T can be any value type including **struct**; it cannot be a reference type

Lecture contents

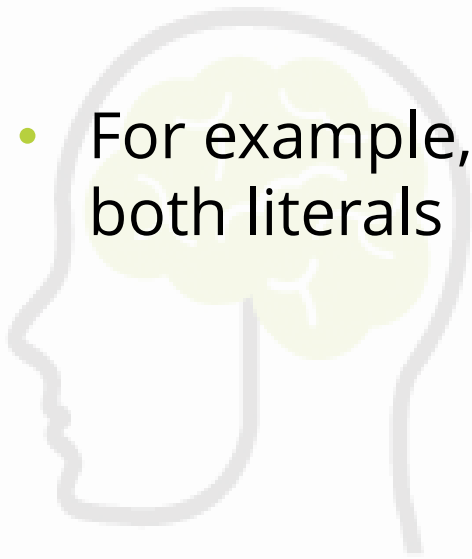
- Basic principles of C #, CLR. Data types
 - Principles of data storage and general information about data types
 - Data types (the primitive, reference and value types)
 - Literals and variables, constants
 - Using types
 - Scope of the variables

Literals (1/2)



Literals - a value used exactly as you see it

- For example, the number 25 and the string "Hello" are both literals



BRAIN
ACADEMY



Literals (2/2)

Code example #11:

```
1. // Inheriting from System.Object
2. Console.WriteLine("Inheriting from System.Object");
3. Console.WriteLine("9.GetHashCode() = {0}", 9.GetHashCode());
4. Console.WriteLine("9.Equals(23) = {0}", 9.Equals(23));
5. Console.WriteLine("9.ToString() = {0}", 9.ToString());
6. Console.WriteLine("9.GetType() = {0}", 9.GetType());
7. Console.WriteLine();
```


The type of a numeric literal

- To specify how a numeric literal should type by appending a letter to the end of the number
 - For example, to specify that the value **6.16** should be treated as a float, append an "f" or "F" after the number: **6.16f**. If no letter is appended, the compiler will infer a type for the literal
- C# specifies some easy-to-follow rules:
 - First, for integer literals, the type of the literal is the smallest integer type that will hold it, beginning with **int**. Thus, an integer literal is either of type **int**, **uint**, **long**, or **ulong**, depending upon its value
 - Second, floating-point literals are of type **double**

Literals

Type(s)	Category	Suffix	Example/Allowed Values
bool	Boolean	None	true or false
int, uint, long, ulong	Integer	None	100
uint, ulong	Integer	u or U	100U
long, ulong	Integer	l or L	100L
ulong	Integer	ul, uL, Ul, UL, lu, lU, Lu, or LU	100UL
float	Real	f or F	1.5F
double	Real	None, d, or D	1.5
decimal	Real	m or M	1.5M
char	Character	None	'a', or escape sequence
string	String	None	"a...a", may include escape sequences

(Karli Watson. Beginning Visual C# 2010)

Some information about strings and string literals

- It is impossible to use the new operator to construct a String object from a literal string
- To concatenate several strings to form a single string use + operator
- C# offers using the verbatim string character (@) to declare a string in which all characters between quotes are considered part of the string




Literals. Code example

Code example #12:

```
1. // Use literals
2. Console.WriteLine("Use literals");
3. //u1, uL, U1, UL, lu, lU, Lu, or LU
4. ulong myint, myint1;
5. myint = 16LU;
6. Console.WriteLine(myint);
7. Console.WriteLine("Hexadecimal literals");
8. myint = 0xFF; // 255 in decimal
9. myint1 = 0x10; // 16 in decimal
10. Console.WriteLine(myint + " " + myint1);
11. Console.WriteLine();
```

Constants (1/2)

 **Constants** - a numeric or string value that is not calculated and, therefore, does not change

- Constants are immutable values which are known at compile time and do not change for the life of the program
- Constants are declared with the **const** modifier
- Only the C# built-in types (excluding **System.Object**) may be declared as **const**



Constants (2/2)

Code example #13:

```
1. //Use constans
2. Console.WriteLine("Use constants");
3. const long c_seconds = 60;
4. const long c_minutes = 60;
5. const long c_hours = 24;
6. const long secondsPerDay = c_seconds * c_minutes * c_hours;
7. Console.WriteLine("Seconds per day " + secondsPerDay);
```

Lecture contents

- Basic principles of C #, CLR. Data types
 - Principles of data storage and general information about data types
 - Data types (the primitive, reference and value types)
 - Literals and variables, constants
 - Using types
 - Scope of the variables



Dynamic initialization

- C# allows variables to be initialized dynamically, using any expression valid at the point at which the variable is declared:

1. Code example #14:

```
2. //Dynamic initialization
3. Console.WriteLine("Use dynamic initialization");
4. double side1 = 4.0;
5. double side2 = 5.0;
6. // Dynamically initialize hyp
7. double hyp = Math.Sqrt((side1 * side1) + (side2 * side2));
8. Console.Write("Hypotenuse with sides " +
9. side1 + " by " + side2 + " is ");
10. Console.WriteLine("{0:###.###}.", hyp);
11. Console.WriteLine();
```


Implicitly typed variables

 **Implicitly typed variables** - a local variable whose type is inferred from the expression that initializes the variable

- It is possible to let the compiler determine the type of a local variable based on the value used to initialize it

```
var Pi = 3.1415926;
```

Casting and type conversions (1/3)

- C# is statically-typed at compile time **and after a variable is declared, it cannot be declared again or used to store values of another type unless that type is convertible to the variable's type**
- But
 - also it is common to assign one type of variable to another
 - Sometimes, you need to copy a value into a variable or method parameter of another type. These kinds of operations are called **type conversions**

Casting and type conversions (2/3)

Kinds of conversions:



Implicit conversions:

- No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes

Explicit conversions (casts):

- Explicit conversions require a cast operator. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class

Casting and type conversions (3/3)

Kinds of conversions:


User-defined conversions:

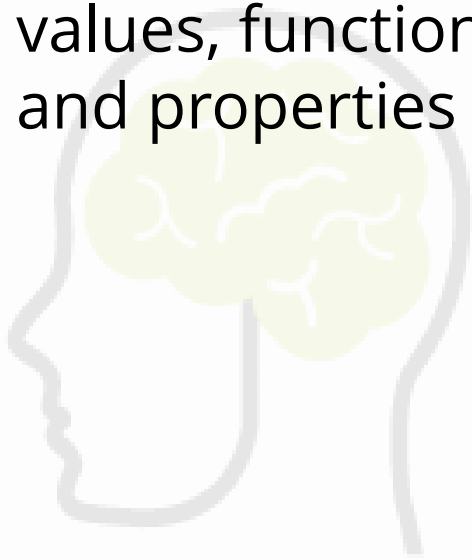
- User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship

Conversions with helper classes:

- To convert between non-compatible types

General information about expressions

 **Expression** - any combination of operators, constants, literal values, functions, and names of fields (columns), controls, and properties that evaluates to a single value



BRAIN
ACADEMY

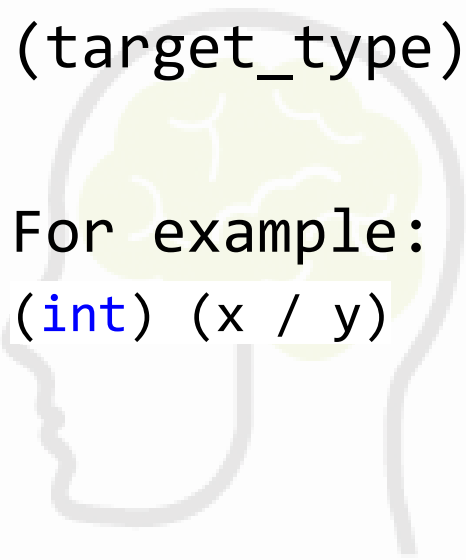
Casting incompatible types

General form:

`(target_type) expression`

For example:

`(int) (x / y)`



BRAIN
ACADEMY



Casting and type conversions (1/2)

Code example #16:



```
1. //Type conversions
2. Console.WriteLine("Type conversions ");
3. short nmb1 = 10, nmb2 = 10;
4. short nmsum;
5. // Explicitly
6. nmsum = (short)(nmb1 + nmb2);
7. Console.WriteLine("Explicitly");
8. Console.WriteLine("{0} + {1} = {2}", nmb1, nmb2, nmsum);
9. nmb1 = 30000;
10. nmb2 = 30000;
11. Console.WriteLine("Casting");
12. // An explicit conversion exists (are you missing a cast?)
13. nmsum = (short)(nmb1 + nmb2); //the result of our addition is
    //completely incorrect
14. Console.WriteLine("{0} + {1} = {2}",
15. nmb1, nmb2, nmsum);
```



Casting and type conversions (2/2)

Code example #16:

```
1. byte studByte = 0;
2. int studInt = 255;
3. // Explicitly cast -no loss
4. studByte = (byte)studInt;
5. Console.WriteLine("Value of Byte: {0}", studByte);
6. Console.WriteLine();
```


Lecture contents

- Basic principles of C #, CLR. Data types
 - Principles of data storage and general information about data types
 - Data types (the primitive, reference and value types)
 - Literals and variables, constants
 - Using types
 - Scope of the variables

Scope and lifetime of variables (1/2)

- C# allows a local variable to be declared within any **block**
- The **block begins** with an **opening curly brace** and **ends with a closing curly brace**. It defines a **scope**
- A scope determines what names are visible to other parts of your program without special qualification. It also determines the lifetime of local variables
- The most important scopes in C# are those defined by a class (later) and those defined by a method (examine now)



Scope and lifetime of variables (2/2)

Code example #15:

```
1. // Demonstrate block scope
2. Console.WriteLine("Demonstrate block scope");
3. int myx; // to all code within Main()
4. myx = 20;
5. if (myx == 20)
6. { // start new scope
7.     int myy = 10; // only to this block
8.     // myx and myy both known here.
9.     Console.WriteLine("myx and myy: " + myx + " " + myy);
10.    myx = myy * 2;
11. }
12. // myy = 1000; // Error
13. Console.WriteLine("myx is " + myx);
14. Console.WriteLine();
```



Code example output (1/2)

1. Demonstrate block scope
2. myx and myy: 20 10
3. myx is 20
- 4.
5. Inheriting from System.Object
6. 9.GetHashCode() = 9
7. 9.Equals(23) = False
8. 9.ToString() = 9
9. 9.GetType() = System.Int32
- 10.
11. Using new to create variables
12. new bool, new int, new double
13. False, 0, 0
- 14.
15. Use integers
16. Our course in academic hours: 117 hours.
17. Our course in academic hours: 117 hours (inherited).
- 18.
19. Use byte, int
20. Summa from 1 to 50 1275
- 21.
22. Use decimal
23. Credit is 1321,20
- 24.
25. Use floating-point types
26. Radius is 5,6418892388858
- 27.
28. Numericals max and min
29. Max of int: 2147483647
30. Min of int: -2147483648
31. Max of double: 1,79769313486232E+308
32. Min of double: -1,79769313486232E+308



Code example output (2/2)

```
33. double.Epsilon: 4,94065645841247E-324
34. double.PositiveInfinity: Infinity
35. double.NegativeInfinity: -Infinity
36.
37. char.IsDigit('C'): False
38. char.IsLetter('C'): True
39. char.IsWhiteSpace('Hello There', 5):
   True
40. char.IsWhiteSpace('Hello There', 6):
   False
41. char.IsPunctuation('?'): True
42.
43. Use boolean
44. 9<10 is True
45. bool.FalseString: False
46. bool.TrueString: True
47. Use literals
48. 16
49. Hexadecimal literals
50. 255 16
47.
48. Use constants
49. Seconds per day 86400
50.
51. Use dynamic initialization
52. Hypotenuse with sides 4 by 5 is 6,403.
53.
54. Type conversions
55. Explicitly
56. 10 + 10 = 20
57. Casting
58. 30000 + 30000 = -5536
59. Value of Byte: 255
60.
61. Press any key to continue . . .
```