The Learning House

427 S 4th Street #300

Louisville KY 40202

Types and Variables in C#

.NET Cohort

Coding Bootcamp

SOFTWARE GUILD

It's All About the Types

A C# program is, conceptually, a set of type declarations for objects and workflows.

- Every project type (program or DLL) is a set of type declarations.
- Namespaces are used to organize your type declarations for uniqueness.

Everything is an object and all objects have a type.

A Type is a Template

- Think of a type as a template for creating objects. Types specify the characteristics of whatever you are modelling in code.
  - For example: a object type BankAccount might have a balance, account number, and have methods of Deposit() and Withdraw().

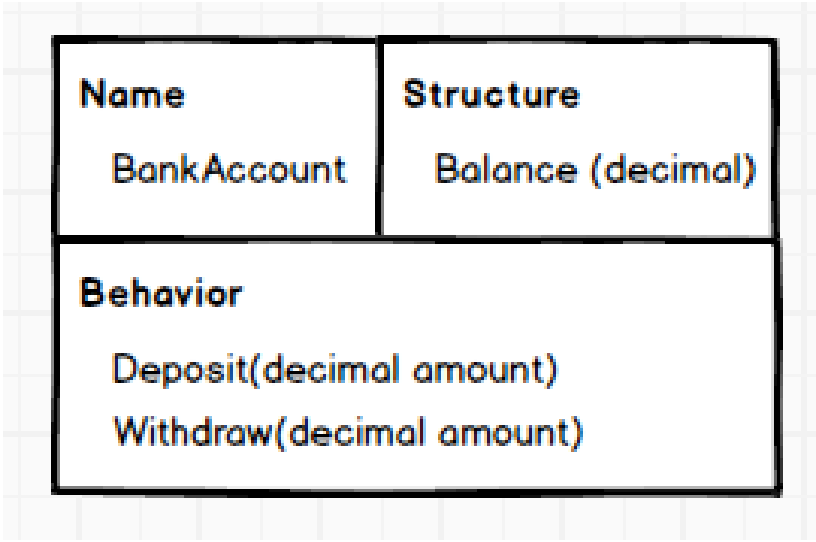
Type Definitions in C#

A type is defined by the elements:

- Name
- Data structure
- Behaviors and constraints

As developers, we define our own custom types as classes and structs so that we can organize common code together.

A bank account structure is a good example. Other structures you could create include: customer information, addresses, movie information, etc.

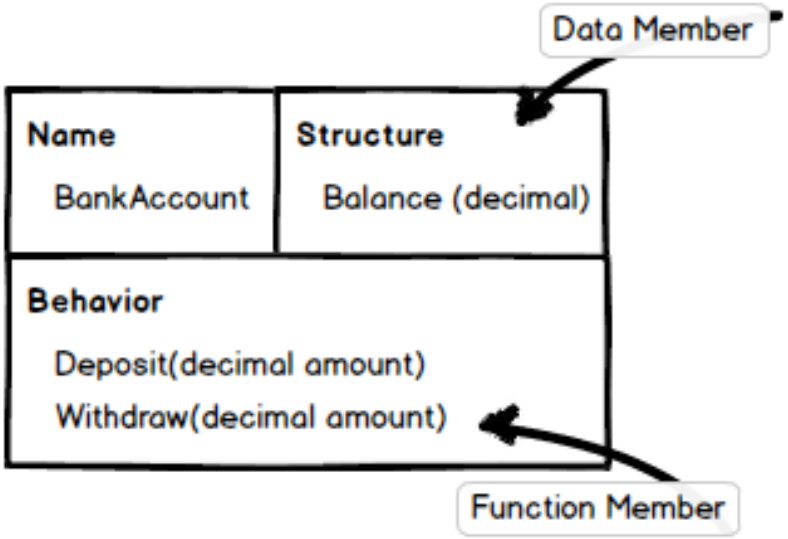


Data Members vs Function Members

Data Members store data that is relevant to the instance of the class or to the class as a whole. (Think adjectives.)

Function Members execute code. They define behavior. (Think verbs.)

Function members may return data or not (if not, they are *void*).



Predefined Simple Types

- Eleven numeric types:
  - byte/sbyte, ushort/short, uint/int, ulong/long, float, double, decimal
- Unicode character: char
- Boolean type: bool (True/False)
  - Pro tip: Some languages convert numbers to Booleans (0=false, 1/-1 = true). C# does not do this.

Simple Type Bounds

Name	Meaning	Range	.NET Framework Type	Default Value
sbyte	8-bit signed integer	-128–127	System.SByte	0
byte	8-bit unsigned integer	0–255	System.Byte	0
short	16-bit signed integer	-32,768–32,767	System.Int16	0
ushort	16-bit unsigned integer	0–65,535	System.UInt16	0
int	32-bit signed integer	-2,147,483,648–2,147,483,647	System.Int32	0
uint	32-bit unsigned integer	0–4,294,967,295	System.UInt32	0
long	64-bit signed integer	-9,223,372,036,854,775,808–9,223,372,036,854,775,807	System.Int64	0
ulong	64-bit unsigned integer	0–18,446,744,073,709,551,615	System.UInt64	0
float	Single-precision float	1.5×10-45–3.4×1038	System.Single	0.0f
double	Double-precision float	5×10-324–1.7×10308	System.Double	0.0d
bool	Boolean	true, false	System.Boolean	false
char	Unicode character	U+0000–U+ffff	System.Char	\x0000
decimal	Decimal value with 28-significant-digit precision	±1.0×1028–±7.9×1028	System.Decimal	0m

Predefined Non-Simple Types

- string – a sequence of Unicode characters
- object – the type on which all other types are based. (Everything in C# is an object.)
- dynamic – used primarily for dynamic languages (IronPython, IronRuby, etc.)

User-Defined Types

- These types are the heart of your program. There are 6 types you can declare:
  - class
  - struct
  - array
  - enum
  - delegate
  - interface

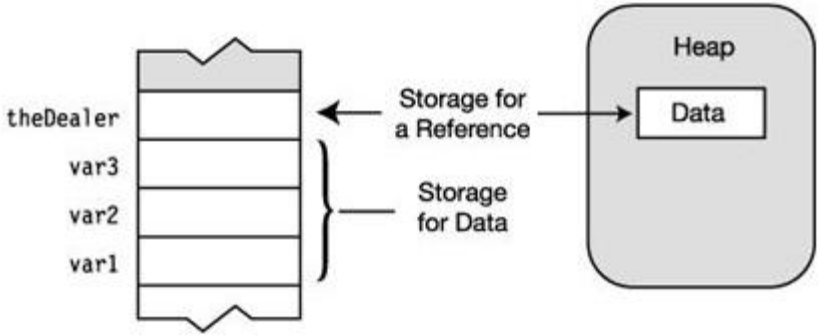
Don’t worry, we will cover these in detail.

Variable Declarations

- Variables must be *declared* before they can be used. The declaration does two things:
  1. Gives the variable a name and type
  2. Tells the compiler to allocate memory space

Declaring Example

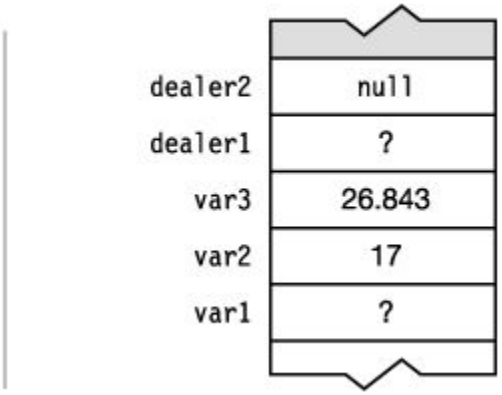
```
int     var1;       // Value Type
int     var2;       // Value Type
float   var3;       // Value Type
Dealer  theDealer;  // Reference Type
```



Initializing Variables

- A local variable without an initializer is undefined and can't be used until it has been assigned a value. (That will cause a compiler error.)

```
int     var1;             // Value Type
int     var2 = 17;        // Value Type
float   var3 = 26.843F;   // Value Type
Dealer  dealer1;          // Reference Type
Dealer  dealer2 = null;   // Reference Type
```



Automatic Initializers

- Some variables are automatically set to default values depending on the circumstances.

Variable	Stored In	Auto-initialized	Use
Local variables	Stack or stack and heap	No	Used for local computation inside a function member
Class fields	Heap	Yes	Members of a `class`
Struct fields	Stack or heap	Yes	Members of a `struct`
Parameters	Stack	No	Used for passing values into and out of a method
Array elements	Heap	Yes	Members of an array

Constant Variables

A constant is a value that is entered at compile time (before the program is run) that cannot change. The *const* keyword makes a variable constant.

It must be assigned a value when declared:

```
const double bodyTemperature = 98.6;
```

Using var

- *var* is a keyword that was added to the language in version 3.0. It is used as an initializer shortcut.
- Basically it means “make the type of this variable the same type as whatever I am assigning.”

```
private static void Main()
{
    var n = 1;

    Console.WriteLine(n);
}
```

(local variable) int n

SOFTWARE GUILD

Introduction to Memory Management

- When your program is running, the data is stored in memory. A running program uses two regions of memory storage:
  - The Stack – An array of memory that acts as a last-in, first-out (LIFO) structure
  - The Heap – Chunks of memory are allocated and can be added/removed in any order

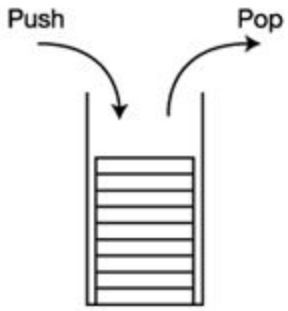
All About the Stack

The stack stores several types of data:

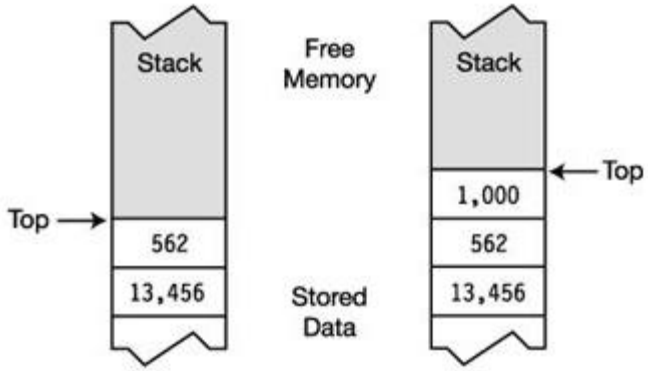
- The values of certain variables
- The current execution environment
- Parameters passed into methods

The system handles the stack manipulation, so we only need to understand conceptually how it works.

- Placing an item onto the stack is called *pushing.*
- Deleting an item from the stack is called *popping.*
- We will see more on this later in algorithms.



Data items are *pushed* onto the top of the stack and *popped* from the top of the stack.



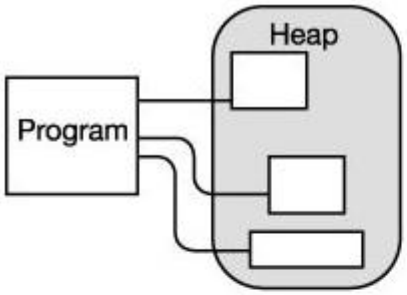
Pushing an integer (e.g., 1,000) onto the stack moves the top of the stack up.

All About the Heap

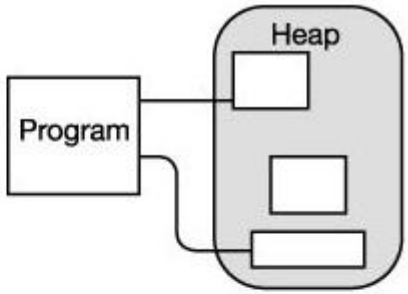
The heap is a chunk of memory space that uses pointers to reference memory locations currently in use.

.NET has something called the Garbage Collector which determines which heap objects are no longer being used and deletes them.

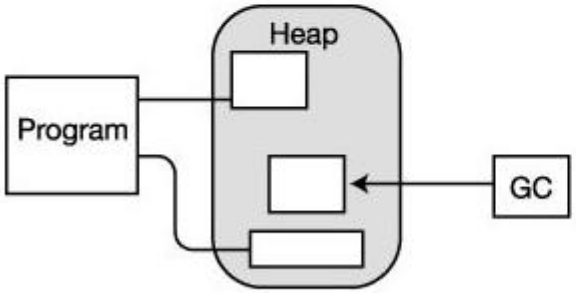
Garbage Collection is a big deal. Back in the old days, we would have to manage cleanup ourselves. If you forgot to do it, your program would eventually overflow and crash the system.



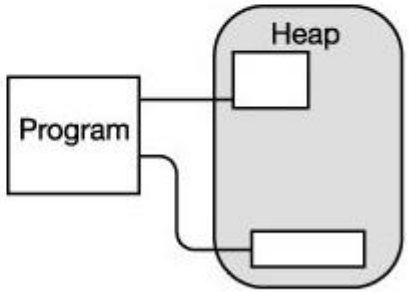
1. The program has stored three objects in the heap.



2. Later in the program, one of the objects is no longer used by the program.



3. The garbage collector finds the orphaned object and releases it.



4. After garbage collection, the released object's memory is available for reuse.

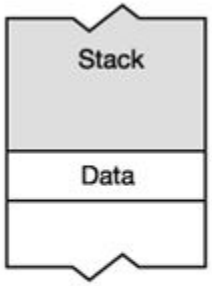
Value Types and Reference Types

The type definition of our data determines how much memory is required to store it and whether it goes on the stack or the heap.

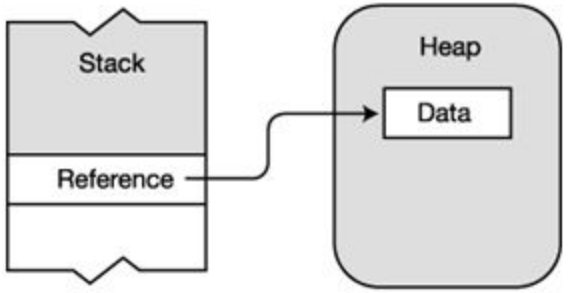
Value types only require a single segment of memory to store and go on the stack.

Reference types require two segments of memory storage:

1. The actual data, which is stored on the heap
2. A reference to the memory location on the heap, stored in the stack.



Value Type Data
– The data is stored on the stack.



Reference Type Data
– The data is stored in the heap.
– The reference is stored on the stack.

Value types vs. reference types

DEMO

Value Types and Reference Type List

	Value Types			Reference Types
Predefined types	sbyte short int long bool	byte ushort uint ulong	float double char decimal	object string dynamic
User-defined types	struct enum			class interface delegate array

Static vs. Dynamic Typing

- C# is a static typed language. This means we tell the compiler what type will be stored in a variable so it can precisely calculate the memory requirements.
- Other languages, like Python and Ruby, are dynamically typed and don’t need to know the type until runtime.
- C# will stop you from putting a string into an int, but some dynamic languages may allow this. Therefore, you have to be more careful with data types in dynamic languages.

Nullable Types

- Since many predefined types have a default (ex: bool=false, int=0) and we spend a lot of time working with databases, .NET added the concept of a nullable type.
- Null means undefined/unknown. It does not mean false, zero, or any value. Comparing null to null is always false, because we don’t know.
- We use the ? to denote a nullable type (ex: int?). We will cover this later in detail.

Conclusion

Your primary role as a developer is to take the built in types in .NET and combine them to create new types that represent your specific program and its goals.

We are only limited by the creativity of our own expression.