Data Types

Identifiers and Keywords

An **identifier** is a name of a program component programmers use to uniquely identify namespaces, classes, methods, variables, constants, etc. Identifiers are user-defined words. For example, in the program shown in *Code Listing 1*, the identifiers are ConsoleApp, ComputeRectangleArea, length, width, area, WriteLine, and ReadKey.

Code Listing 1. Sample class with identifiers

```
using System;
namespace ConsoleApp
{
	class ComputeRectangleArea
	{
		static void Main()
		{
			int length, width, area;
			length = 50;
			width = 8;
			area = length * width;
			Console.WriteLine("The area of the rectangle is " + area);
			Console.ReadKey();
		}
	}
}
```

The following are the syntax rules when naming an identifier in C#:

- It must start with a letter of the English alphabet or an underscore character.
- The identifier's name can only have any combination of letters, digits, and underscores. White spaces are not allowed.
- Identifiers are case sensitive. For example, the identifier Area is not the same with identifiers area or AREA.
- It cannot be a reserved keyword.
- The classes and methods in C# must always begin with a capital letter.

The following are the examples of valid and invalid identifiers:

Valid Identifiers	
_score	score_
first_Name	ScoreClass
grade1	ComputeScore

Table 1. Example valid identifiers

Invalid Identifiers	
1score	*1score* is invalid because it begins with number.
first Name	*first Name* is invalid because it contains white space.
class	*class* is an invalid identifier because it is a reserved keyword.

Table 2. Example invalid identifiers

Keywords are **reserved words** a programming language uses for its own use, and they have a special predefined meaning to the compiler. These cannot be used as identifiers. If you do use a keyword in a program, the compiler will throw an error message. *Table 3* shows the list of keywords in C#.

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	int	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void	volatile	while

Table 3. C# keywords (Harwani, 2015)

Variables

A **variable** is an identifier and a memory location that stores a specific value. Variables hold a value that can be changed during program execution. For example, a variable named `score` assigned an initial value of 25. When the program starts, the value of variable `score` will change to 85. The basic syntax of declaring a variable is as follows:

```
data_type identifier;
```

The `data_type` is one (1) of C#'s data types, and the `identifier` is the name of the variable. For example, `int score;`

Variables are initialized, or assigned a value, with an equal sign followed by the value. The following are some valid examples of declaring and initializing variables:

- You can initialize a variable at the time of declaration. For example:

  ```
  int score = 25;
  ```

- You can declare and initialize more than one (1) variable of the same time data type using a comma in a single statement:

  ```
  int score, age;
  score = 85; age = 24;
  int length = 8, width = 5;
  ```

When creating a program, you must define a variable with a meaningful name that is easy to understand what value must store on it. For example, the variables `gameScore` and `age` must store integer type values. Use *camelCase* notation that starts with a lowercase letter for naming local variables. For example, `numberOfStudents`, `age`, `totalPrice`, etc.

Constants

A **constant** is an identifier and a memory location whose value cannot be changed during program execution. Constants must initialize a value at the time of declaration. The basic syntax of initializing a constant in C# is as follows:

```
const data_type IDENTIFIER = value;
```

Constants in C# are defined using the **`const`** keyword. For example: `const double PI = 3.14159;`. In the example, the constant `PI` with the value of `3.14159` cannot be changed during program execution. The name of the constants must be in all capital to make it easy to identify that its value must not change.

Data Types

Data types are used to specify a set of values and their operations associated with variables or constants. The values have a particular data type because they are stored in memory in the same format and have the same operations defined for them. A variable or constant stores data of a specific type.

When declaring a variable or constant to store a data, the appropriate data type for that data must be identified. The data type will instruct the compiler what kind of value a variable or constant can hold and its operations. There are two (2) types of primitive data types in C#:

- *Value types*. These data types store the value directly. The data type `int` is a value type that stores its value in a memory location directly.
- *Reference types*. These data types do not store the actual value in a variable or constant. Instead, they store the reference (or address) where the value is stored. The *class objects*, *strings,* and *arrays* are reference types because they hold references to blocks of memory and are managed on the heap.

Value Types

A data type is a value type if it holds the data within its own memory allocation. Value types directly store the values within the variable. For example, consider the following figure:

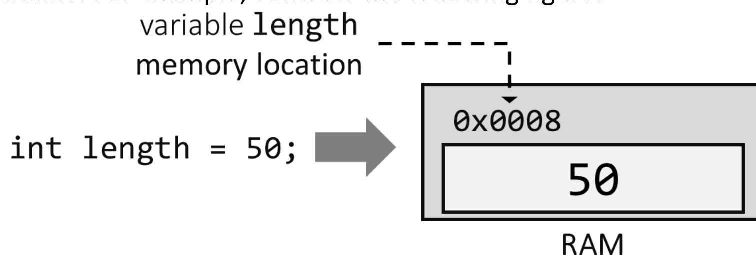

Figure 1. Memory allocation for value type

In *Figure 1*, the variable `length` of `int` type is assigned a value of `50`. When this statement is executed, the compiler will instruct the computer system to directly store `50` in the memory space allocated for the variable `length`.

The table below lists the available value types in C#.

Data Type	Description	Range of Values	Default Value	Example
sbyte	8-bit signed integer	-128 to 127	0	sbyte size = 100;
short	16-bit signed integer	-32,768 to 32,767	0	short score = 1400
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647	0	int num = 12400;
long	64-bit signed integer	-2^{63} to $2^{63}-1$	0L	long length = 124000L;
byte	8-bit unsigned integer	0 to 255	0	byte age = 100;
ushort	16-bit unsigned integer	0 to 65,535	0	ushort uScore = 1400;
uint	32-bit unsigned integer	0 to 4,294,967,295	0	uint uNum = 12400;
ulong	64-bit unsigned integer	0 to $2^{64}-1$	0	ulong uLength = 124000L;
float	Single-precision 32-bit floating-point number	1.5E-45 to 3.4E+38	0.0F	float rate = 235.25F;
double	Double-precision 64-bit floating-point number	5E-324 to 1.7E+308	0.0D	double price = 235.25;
bool	8-bit Logical Boolean type	true or false	false	bool isCorrect = true;
char	Unicode 16-bit character	'\u0000' to '\uffff'	'\0'	char firstLetter = 'C';

Table 4. Value types in C# (Harwani, 2015)

Reference Types

A reference type does not store an actual value, but it stores the address where the value is stored. It means that the reference types contain the memory locations of where the value is stored. For example, consider the figure below:

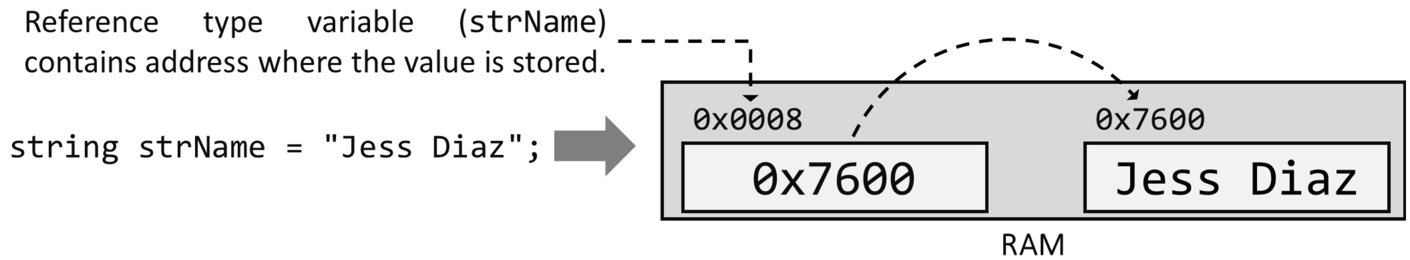

Figure 2. Memory allocation for reference type

In *Figure 2*, the compiler will instruct the computer system to select a different memory location for the variable `strName`. The content of the variable `strName` is "`0x7600`", which is the address or memory location of the actual string value "`Jess Diaz`".

The reference types in C# are Strings, Objects, Arrays, and Delegates.

Type Conversion

Type conversion or **type casting** is the process of converting a value of one (1) type of data to another data type. This is used to ensure that a function correctly processes the variables. Converting a string to an integer is an example of type conversion.

The following are the two (2) forms of type casting in C#:

- *Implicit conversion*. This is the conversion of a lower precision data type to a value of higher precision data type. A compiler automatically performs implicit conversion if the precision of data type to convert is lower than precision of preferred data type. For example, a variable of `short` data type is capable of storing values up to `32,767`, and the maximum value allowed into a `byte` is `255`; so, there will be no problem when you convert the value of `byte` type into a value of `short` data type. An example of implicit conversion includes converting the value of `int` data type to a value of `double` data type, because `int` data type has a lower precision than `double` data type. *Table 5* shows the list of all the valid implicit numeric conversions in C#.

From	To	Example
sbyte	short, int, long, float, or double	sbyte a = 25; short b = a;
byte	short, ushort, int, uint, long, ulong, float, or double	byte a = 25; int b = a;
short	int, long, float, or double	short a = 25; int b = a;
ushort	int, uint, long, ulong, float, or double	ushort a = 25; long b = a;

From	To	Example
int	long, float, or double	int a = 25; double b = a; //the value of b is 25.0
uint	long, ulong, float, or double	uint a = 25; long b = a;
long	float or double	long a = 25; double b = a;
ulong	float or double	ulong a = 25; double b = a;
float	double	float a = 25.0F; double b = a;
char	ushort, int, uint, long, ulong, float, or double	char a = 'a'; int b = a; //the value of b is 97 because the decimal value of character 'a' in ASCII table is 97

Table 5. Implicit numeric conversions (Deitel, P. and Deitel, H., 2015)

Note: There is no implicit conversion of any data type to char type, so the values of the other integral types do not automatically convert to the char type. The bool and double data types have no possible implicit conversions to the other data types.

- *Explicit conversion*. Converting a higher precision data type to a lower precision data type is invalid, and the compiler will return an error. For example, the statements `double num1 = 25.0; int num2 = a;` will return an error because the precision of variable num1 of double data type is higher than the variable num2 of int type. However, explicit conversions allow the users to convert a value of higher precision data type into a value of lower precision data type by using a **cast operator**. The following is the general syntax of performing explicit conversion in C#:

```
(data_type) data_to_convert;
```

The (data_type) is the cast operator that will create a converted copy of the value in data_to_convert. For example, the following statements uses a cast operator to explicitly convert data types:

```
//first example
int num = 25;
byte b = (byte) num;
//second example
double price = 75.5;
int varA = (int) price; //the value of variable a will be 75
```

In the first example, the value 25 is assigned to the variable num of int type, then the converted copy of value from variable num is assigned to the variable b. The value stored in variable num is still an integer type because the cast operator created a converted copy of its value as byte data type.

Explicit conversion involves the risk of losing information, i.e., from double to int data type. For example, in the second example, the value of the variable price of double data type is copied and converted into int data type then stored in the variable varA of int data type. The cast operator dropped the decimal part of the floating-point number 75.5 to convert it to 75 as int data type.

The following statements shows some of the ways on how to use the cast operator:

```
int a = (int) 7.9;              //the return value of a is 7
double b = (double) (5 + 3);    //the value of b is 8.0
Console.WriteLine((int) 2.5);   //the output is 2
int c = 64; char d = (char) c;  //the return value of d is the character '@'
```

The following table shows the list of all valid explicit conversions in C#.

From	To
sbyte	byte, ushort, uint, ulong, or char
byte	sbyte or char
short	sbyte, byte, ushort, uint, ulong, or char
ushort	sbyte, byte, short, or char
int	sbyte, byte, short, ushort, uint, ulong, or char
uint	sbyte, byte, short, ushort, int, uint, long, ulong, or char
long	sbyte, byte, short, ushort, int, uint, ulong, or char

From	To
ulong	sbyte, byte, short, ushort, int, uint, long, or char
char	sybte, byte, or short
float	sbyte, byte, short, ushort, int, uint, long, ulong, or char
double	sbyte, byte, short, ushort, int, uint, long, ulong, char, or double

Table 6. Explicit conversions (Deitel, P. and Deitel, H., 2015)

The explicit conversion of a data type to `char` data type will return the corresponding character of the value from the ASCII table.

```
int a = 64; char b = (char) a;
//the value of b is the character '@' because it is the corresponding character of decimal value 64
in ASCII table
```

REFERENCES:

Deitel, P. and Deitel, H. (2015). *Visual c# 2012 how to program, 5th edition*. USA: Pearson Education, Inc.

Gaddis, T. (2016). *Starting out with visual c#, 4th edition*. USA: Pearson Education, Inc.

Harwani, B. (2015). *Learning object-oriented programming in c# 5.0*. USA: Cengage Learning PTR

.