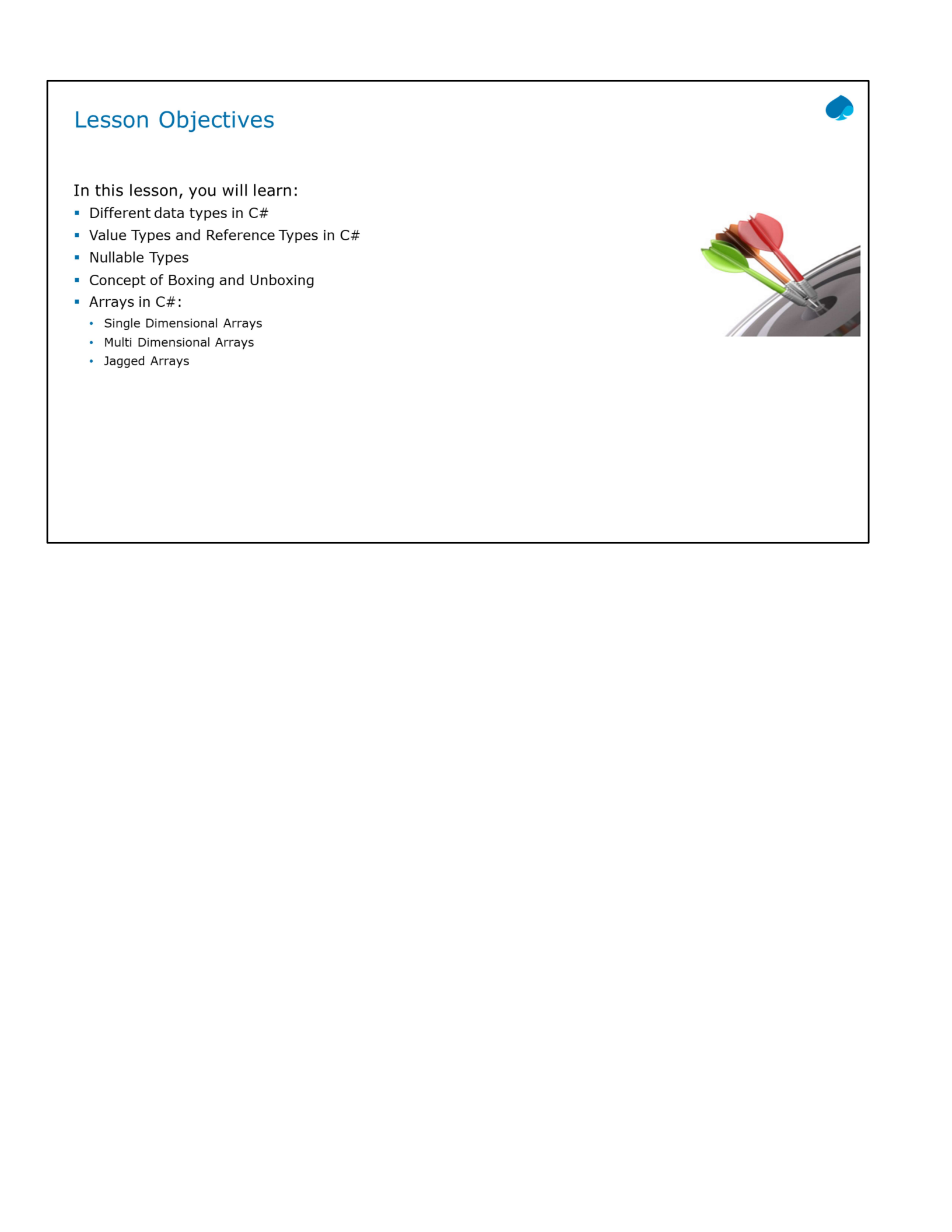
.NET Framework 4.6 and C # 6.0

Lesson 3

Data Types and Arrays in C#

Lesson Objectives



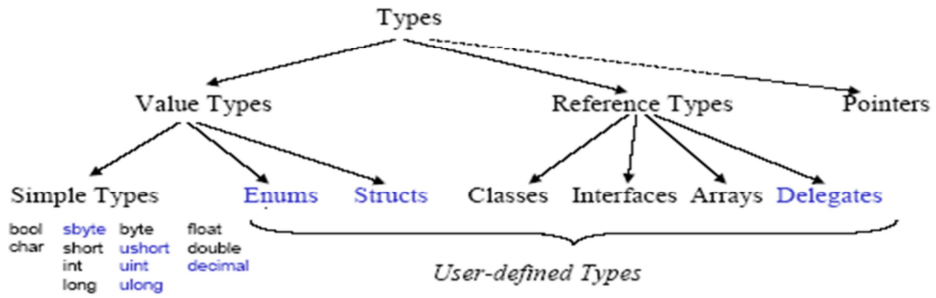
In this lesson, you will learn:

- Different data types in C#
- Value Types and Reference Types in C#
- Nullable Types
- Concept of Boxing and Unboxing
- Arrays in C#:
  - Single Dimensional Arrays
  - Multi Dimensional Arrays
  - Jagged Arrays

3.1: Data Types in C#

Concept of Data Types in C#

- *Unified Type System*



All types are compatible with *object*
- can be assigned to variables of type *object*
- all operations of type *object* are applicable to them

Datatypes in C#:

Every entity in a C# program is an object that lives on either the stack or the managed heap. Every method is defined in a class or struct declaration. There are no such things as free functions, defined outside the scope of class or struct declarations, as there are in C++. Even the built-in value types, such as int, long, double, and so on, have methods associated with them implicitly.

C# Type	.Net Framework Type
Bool	System.Boolean
Byte	System.Byte
Spite	System.SByte
Char	System.Char
Decimal	System.Decimal
Double	System.Double
Float	System.Single
Int	System.Int32
Uint	System.UInt32
Long	System.Int64
Ulong	System.UInt64
Object	System.Object
Short	System.Int16
Ushort	System.UInt16
String	System.String

Concept of Data Types in C#

Storage of Basic Types:

Type	Storage
char, unsigned char, signed char	**2 byte**
short, unsigned short	**2 bytes**
int, unsigned int	**4 bytes**
long, unsigned long	**8 bytes**
float	**4 bytes**
double	**8 bytes**

Concept of Data Types in C#

The Object Type

- C# predefines a reference type named object.
- Every reference and value type is a kind of object. This means that any type we work with can be assigned to an instance of type object.
- **For example:** object o;
  - o = 10;
  - o = "hello, object";
  - o = 3.14159;
  - o = new int[24];
  - o = false;

Data Types in C#:

Every object in the CLR derives from System.Object.

Object is the base type of every type. In C#, the object keyword is an alias for System.Object. It can be convenient that every type in the CLR and in C# derives from Object.

Concept of Data Types in C#

Value Types:

- A variable of a value type always contains a value of that type.
- The assignment to a variable of a value type creates a copy of the assigned value.
- The two categories of value types are as follows:
  - Struct type: user-defined struct types, Numeric types, Integral types, Floating-point types, decimal, bool
  - Enumeration type

Datatypes in C#:

In the managed world of the CLR, there are two kinds of types – Value Types and Reference Types.

Value types:

Value types are defined in C# by using the struct keyword. Instances of value types are the only kind of instances that can live on the stack. They live on the heap if they are members of reference types or if they are boxed, which is discussed later.

Concept of Data Types in C#

Reference Types:

- Variables of reference types, also referred to as **objects**, store references to the actual data.
- Assignment to a variable of a reference type creates a copy of the reference but not of the referenced object.
- Following are some of the reference types:
  - class
  - interface
  - delegate

Data Types in C#:

Reference types:

Reference types are defined in C# by using the class keyword. They are called reference types because the variables you use to manipulate them are actually references to objects on the managed heap. In fact, in the CLR reference-type variables are like value types that reference an object on the heap.

Concept of Data Types in C#

Reference Types (contd.):

- The following are built-in reference types:
  - object
  - string

Comparison between Data Types in C#

Let us differentiate between value and reference types:

Value Types	Reference Types
The variable contains the value directly	The variable contains a reference to the data (data is stored in separate memory area)
Allocated on stack	Allocated on heap using the new keyword
Assigned as copies	Assigned as references
Default behavior is pass by value	Passed by reference
== and != compare values	== and != compare the references, not the values
simple types, structs, enums	classes

3.2: Boxing and Unboxing

Concept of Boxing and Unboxing

Boxing and unboxing enable value types to be treated as objects.

Boxing:

- int number = 2000;
- object obj = number;

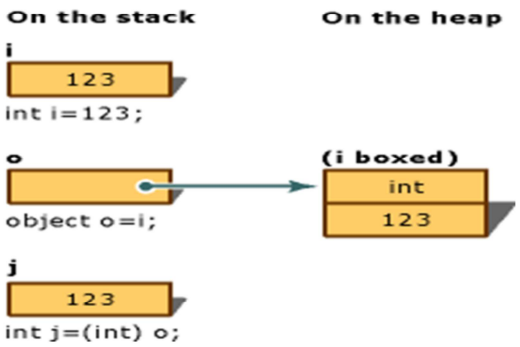
Now both the integer variable and the object variable exist on the stack. However, the value of the object resides on the heap.

Note: Value types, including both struct types and built-in types, such as int, can be converted to and from the type object.

Concept of Boxing and Unboxing

Unboxing Conversions

- int number = 2000;
- object obj = number;
- int anothernumber = (int)obj;



Concept of Nullable Types

- Nullable types represent value-type variables that can be assigned the value of null.
- You cannot create a nullable type based on a reference type: Reference types already support the null value.
- A nullable type can represent the normal range of values for its underlying value type, plus an additional null value.

Concept of Nullable Types

Example:

- A Nullable<Int32>, pronounced “Nullable of Int32”, can be assigned any value from -2147483648 to 2147483647, or it can be assigned the null value.
- A Nullable<bool> can be assigned the values true or false, or null.

Concept of Nullable Types

The ability to assign null to numeric and Boolean types is particularly useful when dealing with databases and other data types containing elements that may not be assigned a value.

Example:

- A Boolean field in a database can store the values true or false, or it may be undefined.

Concept of Nullable Types

The syntax T? is shorthand for System.Nullable<T>, where T is a value type

```
static void Main()
{
int? num = null;
if (num.HasValue == true)
    {
    Console.WriteLine("num = " + num.Value);
    }
else
    {
    Console.WriteLine("num = Null");
    }
}
```

Demo

Demo on Data Types, Boxing and Unboxing, and Nullable Types

3.4: Implicitly Typed Local Variables

Concept of Implicitly Typed Local Variables

- Type of the local variable being declared is inferred from the expression used to initialize the variable.
- When a local variable declaration specifies var as the type and no type named var is in scope, the declaration is an implicitly typed local variable declaration.

Example:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

Implicitly Typed Local Variables:
In an implicitly typed local variable declaration, the type of the local variable being declared is inferred from the expression used to initialize the variable. When a local variable declaration specifies var as the type and no type named var is in scope, the declaration is an implicitly typed local variable declaration.
Example:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

The implicitly typed local variable declarations shown above are precisely equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

Concept of Implicitly Typed Local Variables

- The above declarations are equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

Concept of Implicitly Typed Local Variables

Restrictions:

- Declarator must include an initializer.
- Initializer must be an expression - cannot be an object or collection initializer.
- Compile-time type of the initializer expression cannot be null type.
- If the local variable declaration includes multiple declarators, the initializers must all have the same compile-time type.

Implicitly Typed Local Variables:

A local variable declarator in an implicitly typed local variable declaration is subject to the following restrictions:

The declarator must include an initializer.

The initializer must be an expression. The initializer cannot be an object or collection initializer by itself, but it can be a new expression that includes an object or collection initializer.

The compile-time type of the initializer expression cannot be the null type.

If the local variable declaration includes multiple declarators, the initializers must all have the same compile-time type.

For reasons of backward compatibility, when a local variable declaration specifies var as the type and a type named var is in scope, the declaration refers to that type. However, a warning is generated to call attention to the ambiguity. Since a type named var violates the established convention of starting type names with an upper case letter, this situation is unlikely to occur.

Concept of Implicitly Typed Local Variables

Following are examples of incorrect implicitly typed local variable declarations:

- `var x;` // Error, no initializer to infer type from
- `var y = {1, 2, 3};` // Error, collection initializer not permitted
- `var z = null;` // Error, null type not permitted

Concept of Implicitly Typed Local Variables

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers)
    Console.WriteLine(n);
```

the type of n is inferred to be int

Implicitly Type Local Variables:
The for-initializer of a for statement and the resource-acquisition of a using statement can be an implicitly typed local variable declaration.
Likewise, the iteration variable of a foreach statement may be declared as an implicitly typed local variable, in which case the type of the iteration variable is inferred to be the element type of the collection being enumerated.
In the following example, the type of n is inferred to be int, the element type of numbers.

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers)
        Console.WriteLine(n);
```

3.5: Arrays

Concept of Arrays

An array is a data structure that contains a number of variables called the elements of the array.

- All of the array elements must be of the same type, which is called the element type of the array.
- An array can be a single-dimensional array, a multidimensional array, or a jagged array (Array of Arrays).
- Array types are reference type derived from the abstract base type System.Array.

Arrays:
C# arrays are zero indexed. That is, the array indexes start at zero.
Array elements can be of any type, including an array type.

Salient Features

Single Dimensional Arrays exhibit the following features:

- They declare a single-dimensional array of five integers.
  - **Example:** int[] array1 = new int[5];
- It is an array that stores string elements.
  - **Example:** string[] stringArray = new string[6];
- It declares and sets array element values.
  - **Example:** int[] array2 = new int[] { 1, 3, 5, 7, 9 };
- The alternative syntax is as follows:
  - **Example:** int[] array3 = { 1, 2, 3, 4, 5, 6 };

3.7: Multi Dimensional Array

Illustrations

Multi-Dimensional Arrays can be elucidated with the following examples:

- **Example 1:** // Declare a two dimensional array
  - int[,] multiDimensionalArray1 = new int[2, 3];
- **Example 2:** // Declare and set array element values
  - int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

Arrays:
Examples of defining Arrays:
Single-Dimensional Array:

Example 1: int[] numbers = new int[5] {1, 2, 3, 4, 5};
Example 2: string[] names = new string[3] {"Matt", "Joanne", "Robert"};
Example 3: int[] numbers = {1, 2, 3, 4, 5};
Example 4: string[] names = {"Matt", "Joanne", "Robert"};

Multidimensional Array:

Example 1: int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
Example 2: string[,] siblings = new string[2, 2] { {"Mike","Amy"}, {"Mary","Albert"} };
Example 3: int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };
Example 4: string[,] siblings = new string[,] { {"Mike","Amy"}, {"Mary","Ray"} };
Example 5: int[,] numbers = { {1, 2}, {3, 4}, {5, 6} };
Example 6: string[,] siblings = { {"Mike", "Amy"}, {"Mary", "Albert"} };

Salient Features

- Jagged array is an array whose elements are arrays.
- The elements of a jagged array can be of different dimensions and sizes.
- A jagged array is sometimes called an "array of arrays".
- Following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

Example:

```
int[][] jaggedArray = new int[3][];
```

Salient Features

Let us see an example on jagged arrays:

```
int[][] jaggedArray = new int[3][];

jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];

jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Array of Elements:
Each of the elements is a single-dimensional array of integers
The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.
It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size.

Demo

Demo with Arrays in C#

Summary

In this lesson, you have learnt:

- Different Data Types in C#
- Difference between Value Types and Reference Types
- Concept of Boxing and Unboxing
- Different types of Arrays in C#

Review Questions

Question 1: How are Value Types different from Reference Types?

Question 2: What is Boxing and Unboxing in C#?

Question 3: What are Jagged Arrays?

Answers for the Review Questions:

Answer 1:
a) The variable of Value
Type contains the value directly where as the variable of reference type
contains reference to the data (data is stored in a separate memory area).

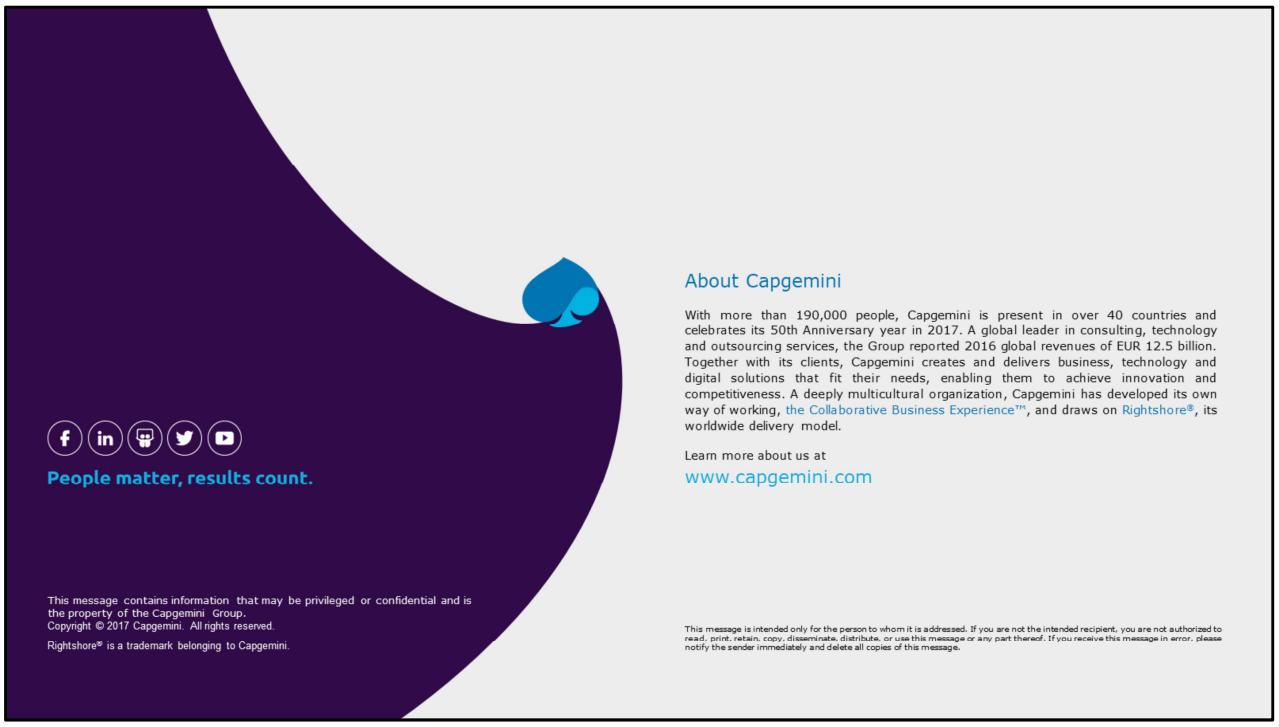
b) Value Types are
allocated on stack and assigned as copies where
as Reference Types are allocated on heap using
the new keyword and are assigned as references.

c) Examples of Value
Types: simple types,
structs, enums.

Examples of Reference
Types: classes, interfaces,
String, object etc.

Answer 2:
Boxing is conversion of value type to reference type. Unboxing is conversion of reference type back to value type.

Answer 3:
Jagged Array is array of arrays.


People matter, results count.
This message contains information that may be privileged or confidential and is the property of the Capgemini Group.
Copyright © 2017 Capgemini. All rights reserved.
Rightshore® is a trademark belonging to Capgemini.
About Capgemini
With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.
Learn more about us at
www.capgemini.com
This message is intended only for the person to whom it is addressed. If you are not the intended recipient, you are not authorized to read, print, retain, copy, disseminate, distribute, or use this message or any part thereof. If you receive this message in error, please notify the sender immediately and delete all copies of this message.