List<T>

Dynamic Arrays

Topics

- Arrays
- Arrays Pros-Cons
- List<T>
- Looping
- Removing
- Cloning
- Intermediate Level

Arrays

https://www.tutorialspoint.com/csharp/csharp_arrays.htm

The Basics: Arrays

What are arrays?

An int variable will use 4 bytes of memory.
An array of ints will be a block of ints in memory.

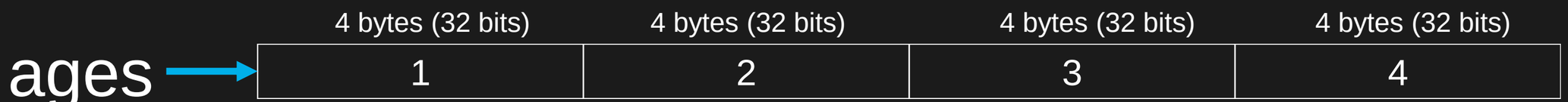
All of the ints are contiguous in memory.
In other words, they are right next to each other in memory.

The Basics: Arrays

EXAMPLE:

```
int[] ages = new int[4] {1,2,3,4};
```

The computer will set aside 4 bytes for each int (4 ints in the example)



Arrays How To

- Create an array

```
int[] nums = new int[5] { 1, 2, 4, 5, 6 };
string[] names = new string[] { "Bats", "Batman", "The Dark Knight" };
```

- Loop over an array
  remember, array indexes are 0 based meaning they start at 0.

```
for (int j = 0; j < nums.Length; j++)
{
    Console.WriteLine(nums[j]); //j is the index
}
```

- Assign a value to a spot in the array

```
names[0] = "The Caped Crusader";
```

Array Challenge

1. Create a method called ArrayChallenge.
2. In the method, create an array of 10 ints
3. Fill the array with 10 random ints.
4. Loop over the array and print the numbers.
5. Call ArrayChallenge from Main.

LINKS

Arrays

Slides

Arrays: Pros & Cons

Arrays: Pros

- Arrays are great for dealing with a **fixed number of elements**.
  If I know I only need 7 cars, then use an array.

- Vehicle[ ] cars = new Vehicle[7];

- This will allocate 7 spots in memory all at once so the memory will be contiguous

Arrays: Pros

Accessing an element in an array is *super fast!*

The ***worst-case performance*** is **O(1) constant time**

In other words, it takes the same amount of time to access the first element as it does the 1,000,000 element.

This is the major advantage to using arrays.

Arrays: Pros

EX:

int[] nums = new int[1000001];

nums[0] takes the *same amount of time* as nums[1000000].

How?

To look up the int at an index is just a calculation:

(start of array in memory) + index * (size of type)

The size of array does not affect the lookup performance – it is constant time O(1).

Arrays: Cons

- Arrays are **bad** if you need to resize the array.

- Why?
  All of the memory is allocated at the same time for an array, so if I want to grow or shrink the array, you have to re-allocate all of the memory and copy items from the old array to the new array.
  - This creation of the new array, copy from the old to the new, then deleting the old array uses more memory and takes time.

Arrays: Cons

EXAMPLE:

- Vehicle[] cars = new Vehicle[7];
- What happens if I want to add an 8th car to my array?
  You would have to *manually* resize the array.

1. Create the new array
   Vehicle[ ] newcars = new Vehicle[8];

2. Copy from the old array to the new array
   ```
   for(int i=0; i < 7; i++)
         newcars[i] = cars[i]; //copy the cars to the new array
   newcars[7] = new Vehicle(); //add the new car to the array
   ```

3. Delete the old array.
   cars = newcars; //the garbage collector will eventually delete the old array

Arrays: Cons

- What do we have to do to remove the 3rd item and shrink the array?
- What would we have to do if we wanted to insert an item in the middle of the array?

- All cases would require custom code to handle and would potentially be a source of bugs in your code.

Arrays: Pros & Cons

- PROS:
  - Super fast lookup – O(1) performance
  - Great for fixed number of items
- CONS:
  - Bad if you need to resize.
  - Duplicates data
  - Requires code to copy items to the new array

List<T>

List<T> class to the rescue!

List<T>

- **List<T>** is a .NET class that will ***auto resize*** it's collection as the collection as items are added, inserted, and removed.

  You do not need to worry about resizing the array!

- *Internally* it uses an array to store the items and does all the work for you to grow/shrink the array.

List<T>

List<T> is a .NET class that can store ***any type*** in the collection.

- **T** is a **generic type parameter** – it's the parameter to the class telling it what type to store internally.

Examples:

```
List<Vehicle> cars = new List<Vehicle>();
List<bool> flags = new List<bool>();
List<float> grades = new List< float >();
```

Creating Lists

- You'll need the using System.Collections.Generic; in the usings

- When creating a List, put the type you want to store in the < >.
  EX: If you want a list of bools…List<bool>

```
List<int> numbers = new List<int>();
List<string> supers = new List<string>();
List<double> scores = new List<double>(10);//presize to hold 10 items
```

Adding to Lists

- There are 2 ways to add items to a list
  1. Add items in the initializer

```
List<int> numbers = new List<int>() { 5, 4, 3, 2, 1 };
```

  2. Use the Add method

```
List<string> supers = new List<string>();
supers.Add("Batman");
supers.Add("Superman");
supers.Add("Wonder Woman");
```

Challenge #1: Creating and Adding

1. Create a method called ListChallenge.
2. In the method, create a List of doubles and call the variable grades.
3. Using the Random class, add 10 random grades (0-100) to the grades list.

Example:

```
List<string> supers = new List<string>();
supers.Add("Batman");
supers.Add("Superman");
supers.Add("Wonder Woman");
```

LINKS

List

SLIDES

List How-To

Add How-To

Count vs Capacity

List<T>: Capacity & Count

- **Capacity** is the total number of elements in the internal array before resizing is needed.
- **Count** is the number of actual elements that have been added to the collection.
- The initial capacity and count of a List are both 0.
  List<int> numbers = new List<int>();
  Count = 0
  Capacity = 0

List<T>: Capacity & Count

- When you add an item that is *beyond* the current capacity of the List, List will automatically resize the internal collection.
  Example:
  if you've added 4 items to a list, the Count = 4 and the Capacity = 4.
  The next Add will force a resize: the capacity is doubled and the new item is added.

- You can create a List with an initial capacity.
  Let's say you know you'll be adding 10 items to your List, you can create it like this:

```
List<int> knownSize = new List<int>(10); //the initial capacity of the list is 10
```

Looping

Looping over Lists

- for loops
  NOTE: use the Count property in the for loop condition

```
for (int i = 0; i < supers.Count; i++)
{
    Console.WriteLine($"Superhero: {supers[i]}");
}
```

- foreach loops

```
foreach (var super in supers)
{
    Console.WriteLine($"Superhero: {super}");
}
```

Challenge #2: Printing

1. Create a method called PrintGrades that takes a list as a parameter.
2. In the method, loop over the grades list and print the grades.
3. Call PrintGrades from the ListChallenge method.

Example:

```
for (int i = 0; i < supers.Count; i++)
{
    Console.WriteLine($"Superhero: {supers[i]}");
}
```

LINKS

for

foreach

Interpolated strings

Formatting

SLIDES

Looping How-To

Removing Items

Removing from Lists

- There are 2 main ways to remove items from a list

1. Use the Remove(item) method
   NOTE: this will only remove the first “Aquaman” it finds in the list.

```
bool wasRemoved = supers.Remove("Aquaman");
```

2. Use the RemoveAt(index) method
   NOTE: the index passed in must be in the 0-(Count-1) index range.

```
supers.RemoveAt(2);
```

Removing from Lists

- When removing an item, all items to the right (or after) the item will be shifted to the left in the list.
- EX: if the list has the following items – 1,5,8,10 and you remove 5, items 8 and 10 will be shifted 1 spot to the left in the list. Result: 1,8,10.

Challenge #3: Removing

1. Create a method call DropFailing. You will need to pass a list as a parameter to the method.
2. Loop over the grades list and remove all failing grades. Keep track of how many grades were removed.
3. Return the # of grades that were removed.
4. Call DropFailing from the ListChallenge method.
5. Print the number of failing grades that were removed.
6. Print the grades again.

Example:

```
supers.RemoveAt(2);
```

LINKS

for

SLIDES

Removing How-To

Creating Lists from Arrays

Creating Lists from Arrays

Sometimes you need to get a List but what you have is an array. Maybe you have to pass a List as a parameter to some other method.

Thankfully, this is an easy task.

Creating Lists from Arrays

There are 3 ways you can get a List from an array:

1. Call ToList on the array.
2. Create a List and pass the array into the constructor.
3. Create a List and copy each array item into the list.

Creating Lists from Arrays

- There are 3 ways to create a list from an array

1. Use ToList on the array.
   NOTE: you'll need to add using System.Linq; to the usings at the top of the file

```
string[] names = new string[] { "Bats", "Batman", "The Dark Knight" };
List<string> theBest = names.ToList();
```

2. Pass the array to the List constructor

```
string[] names = new string[] { "Bats", "Batman", "The Dark Knight" };
List<string> batMen = new List<string>(names);
```

3. Loop over the array and copy each item to the list

```
string[] names = new string[] { "Bats", "Batman", "The Dark Knight" };
List<string> batMen = new List<string>();
for (int i = 0; i < names.Length; i++)
    batMen.Add(names[i]);
```

Cloning Lists

Cloning Lists

Since Lists are reference types, simply assigning the value of 1 list to another will not copy the list but instead point the new list variable to the same list.

EX: List<int> list2 = otherList;

list2 points to the same list of values as otherList.

Cloning Lists

If you assign one list variable to another, they will be pointing to the same list.

```
List<string> batMen = new List<string>() { "Bats", "Batman", "The Dark Knight" };
List<string> bruces = batMen;
bruces.Remove("Batman"); // will change both b/c they are the same list
```

Cloning Lists

- To properly clone a list into a new, separate list…

1. Use ToList

```
List<string> batMen = new List<string>() { "Bats", "Batman", "The Dark Knight" };
List<string> bruces = batMen.ToList();
```

2. Pass the first list to the constructor of the second list

```
List<string> batMen = new List<string>() { "Bats", "Batman", "The Dark Knight" };
List<string> bruces = new List<string>(batMen);
```

Challenge #4: Cloning

1. Create a method called CurveGrades that takes a list as a parameter.
2. In the method, clone the grades list to a list named curved
3. “Curve” the grades in the cloned list by adding +5 to each grade. NOTE: don’t go over 100.
4. Return the curved list.
5. Call CurveGrades from the ListChallenge method.
6. Print the curved grades that are returned.

Example:

```
List<string> batMen = new List<string>() { "Bats", "Batman", "The Dark Knight" };
List<string> bruces = batMen.ToList();
```

LINKS

For

Console.CursorLeft

SLIDES

Looping How-To

Cloning How-To

Intermediate Level

Challenge #5: Printing Side-by-Side

- Clone the grades list to a list named curved
- “Curve” the grades in the cloned list by adding +5 to each grade. NOTE: don’t go over 100.
- ~~Print the curved grades~~
- Print the 2 lists side-by-side using 1 loop

LINKS

For

Console.CursorLeft

SLIDES

Looping How-To

Cloning How-To

Challenge #6: Color coding

- Right-align the # and print only 2 decimal places
- Color code the grades.
  - A: Green
    B: Dark Green
    C: Yellow
    D: Dark Yellow
    F: Red

LINKS

For

Console.CursorLeft

SLIDES

Looping How-To

Cloning How-To