Introduction to Arrays

Principles of Computer Programming I

Spring/Fall 20XX

AUGUSTA UNIVERSITY

Outline

- Array basics and syntax
- Array initialization
- Loops and arrays
- Array resizing

Collections of Values

- Sometimes you need lots of variables of the same type
- Example: Computing average grade

```
int homework1Grade = 89;
int homework2Grade = 72;
int homework3Grade = 88;
int homework4Grade = 80;
int homework5Grade = 91;
double averageGrade = (homework1Grade + homework2Grade +
  homework3Grade + homework4Grade + homework5Grade) / 5.0;
```

- This is tedious. What if you add another homework?

Arrays: Groups of Variables

- Instead of 5 separate variables, use an **array** of 5 variables

Type of variable: "int array"

Instantiation – arrays are objects

```
int[] homeworkGrades = new int[5];
homeworkGrades[0] = 89;
homeworkGrades[1] = 72;
homeworkGrades[2] = 88;
homeworkGrades[3] = 80;
homeworkGrades[4] = 91;
```

Size of array

Each "entry" is an `int` variable

Brackets, **not** parentheses

First index is `0`

"subscript" or "index" operator

Basic Array Syntax

- Declaring an array:

```
type[] arrayName;
```

Any data type

- Assigning to an array:

```
arrayName = new type[<size>];
```

Must match type of array variable

- One-line initialization:

```
type[] arrayName = new type[<size>];
```

- Accessing array elements:

Between `0` and `size - 1`

Must match type of array

Write (assignment):

```
arrayName[index] = <value>;
```

Read:

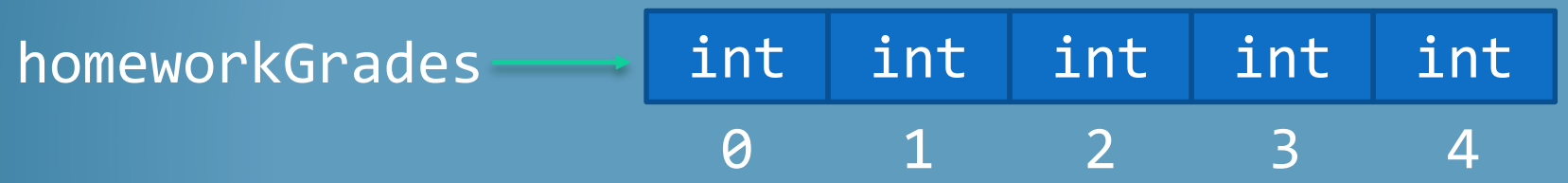
```
Console.WriteLine(arrayName[index]);
```

Calls `ToString` like any other variable

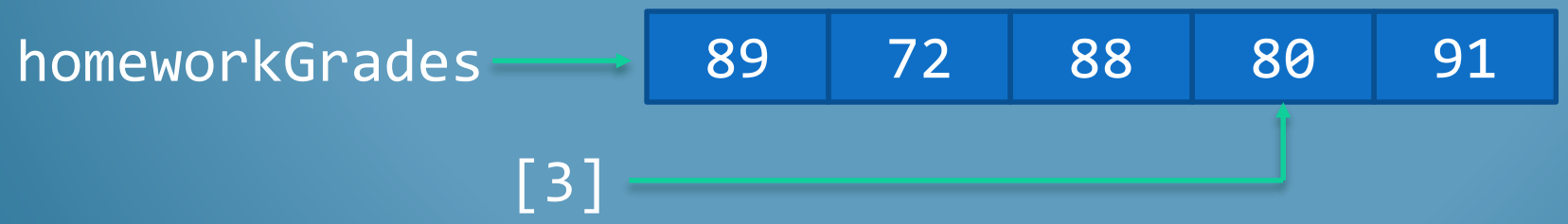
Visualizing Arrays

- Array instantiation creates a set of "adjacent" variables

```
int[] homeworkGrades = new int[5];
```



- Index operator determines which one to access
  - Index = number of variables to skip



Array Bounds Checking

- Valid index values: 0 to (size of array) – 1
- What happens if we use an invalid index?

```
double[] myArray = new double[3];
myArray[0] = 6.88;
myArray[1] = -2.5;
myArray[2] = 100.3;
myArray[3] = -0.05;
```

Index = array size, invalid

- Run-time error, not compile-time

`System.IndexOutOfRangeException`

Outline

- Array basics and syntax
- **Array initialization**
- Loops and arrays
- Array resizing

Array Initialization Shortcuts

- Initializing and filling an array:

```
int[] myArray = new int[3];
myArray[0] = 10;
myArray[1] = 20;
myArray[2] = 30;
```

- Initialization with values:

```
int[] myArray = new int[] {10, 20, 30};
```

Size not needed

Comma between each value

Syntax:

```
type[] arrayName = new type[] {<value1>, <value2>, ...};
```

Initialization Shortcuts

- Even shorter initialization with values:

```
type[] arrayName = {<value1>, <value2>, ...};
```

Must be values of this type

```
int[] homeworkGrades = {89, 72, 88, 80, 91};
```

- Assignment with shortcuts: new is required

```
int[] homeworkGrades;
homeworkGrades = {89, 72, 88, 80, 91};
homeworkGrades = new int[] {89, 72, 88, 80, 91};
```

Error!

Arrays of Objects

- Array can be any data type, including objects like Rectangle

```
Rectangle[] shapes = new Rectangle[3];
```

- But this does **not** create 4 Rectangle objects

```
shapes[0].SetLength(3);
```

✗ NullReferenceException

- Initialization:

```
shapes[0] = new Rectangle(3, 5);
shapes[1] = new Rectangle(14, 10);
shapes[2] = new Rectangle(19, 22);
```

or

```
Rectangle[] shapes = {new Rectangle(3, 5),
  new Rectangle(14, 10), new Rectangle(19, 22)};
```

Default Values of Arrays

- Value type: Variable stores the data directly (numeric types)
- Reference type: Variable stores a reference to the data (objects)
- For array of **value** types, default values are `0`

```
int[] homeworkGrades = new int[5];
```

This works:

```
homeworkGrades[0]++;
homeworkGrades[3] += 50;
```

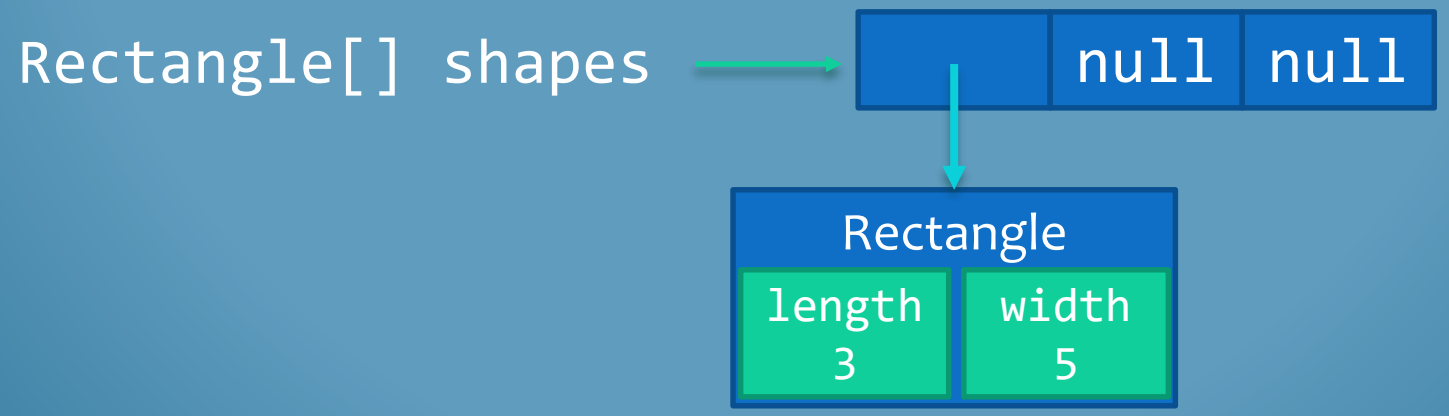
Default Values of Arrays

- For array of **reference** types, default values are `null`

```
Rectangle[] shapes = new Rectangle[3];
```

- Must initialize each element by **instantiating** an object

```
shapes[0] = new Rectangle(3, 5);
```



Outline

- Array basics and syntax
- Array initialization
- **Loops and arrays**
- Array resizing

Using Arrays

- Accessing array elements individually:

```
int[] homeworkGrades = {89, 72, 88, 80, 91};
double average = (homeworkGrades[0] + homeworkGrades[1]
  homeworkGrades[2] + homeworkGrades[3] + homeworkGrades[4]) / 5.0;
```

- Can we write a loop that does this with less repetition?

Use <, stop when counter == 5

(there is no homeworkGrades[5])

```
int counter = 0, sum = 0;
while(counter < 5)
{
  sum += homeworkGrades[counter];
  counter++;
}
double average = sum / 5.0;
```

Custom-Sized Arrays

- Array size can be any `int`, including a variable:

```
int numGrades = 10;
int[] homeworkGrades = new int[numGrades];
```

Creates an array of 10 `ints`

- Array size can be user-provided:

```
Console.WriteLine("How many grades are there?");
int numGrades = int.Parse(Console.ReadLine());
int[] homeworkGrades = new int[numGrades];
```

- How do we fill this array when we don't know its size?

Custom-Sized Arrays

- `while` loops make it easy to process "the whole array"
  - Counter-controlled loop with a variable: the size of the array

```
Console.WriteLine("How many grades are there?");
int numGrades = int.Parse(Console.ReadLine());
int[] homeworkGrades = new int[numGrades];
int i = 0;
while(i < numGrades)
{
  Console.WriteLine($"Enter grade for homework {i+1}");
  homeworkGrades[i] = int.Parse(Console.ReadLine());
  i++;
}
```

Size of array

Homework 1 is in `homeworkGrades[0]`

No input validation (for now)

Looping with the Length Property

- Arrays are objects with instance variables
- `int length` contains the length (size) of the array, can be accessed with property `Length`

```
class Array
{
  private int length;
  public int Length
  {
    get
    {
      return length;
    }
  }
}
```

```
int i = 0, sum = 0;
while(i < homeworkGrades.Length)
{
  sum += homeworkGrades[i];
  i++;
}
double average = (double) sum /
  homeworkGrades.Length;
```

Loops through all `int`s in `homeworkGrades`, however many

Could use `i`, but no need to

AUGUSTA UNIVERSITY

Loops and Arrays with Objects

"Ask the user how many `Items` they want to create, then use user input to initialize each `Item` with a price and description"

```
Console.WriteLine("How many items would you like to stock?");
Item[] items = new Item[int.Parse(Console.ReadLine())];
int i = 0;
while(i < items.Length)
{
  Console.WriteLine($"Enter description of item {i+1}:");
  string description = Console.ReadLine();
  Console.WriteLine($"Enter price of item {i+1}:");
  decimal price = decimal.Parse(Console.ReadLine());
  items[i] = new Item(description, price);
  i++;
}
```

Arrays of Objects

"Assume we have an array of `Item` objects named `myItems`. Find and display the lowest price in the array."

```
decimal lowestPrice = decimal.MaxValue;   OR myItems[0].GetPrice()
int i = 0;
while(i < myItems.Length)
{
  if(myItems[i].GetPrice() < lowestPrice)
  {
    lowestPrice = myItems[i].GetPrice();
  }
  i++;
}
Console.WriteLine($"The lowest-priced item costs {lowestPrice:C}");
```

Arrays of Objects

"Assume we have an array of `Item` objects named `myItems`. Find and display the lowest-priced `Item` *object* in the array."

```
Item lowestItem
int i = 1;
while(i < myItems.Length)
{
  if(myItems[i].GetPrice() < lowestItem.GetPrice())
  {
    lowestItem = myItems[i];
  }
  i++;
}
Console.WriteLine($"The lowest-priced item is {lowestItem}");
```

Loops with Multiple Arrays

"Assume `myNums` and `yourNums` are arrays of `int` of equal size. Display "Match" if at least 1 element of `myNums` matches the element in the corresponding position in `yourNums`, and "No match" if all elements of the two arrays are different."

Match

myNums	9	-2	18	0	99
yourNums	14	-2	6	110	9

No match

12	0	8	-2	40
8	-2	6	110	9

Loops with Multiple Arrays

```
int i = 0;
bool match = false;
while(i < myNums.Length)
{
  if(myNums[i] == yourNums[i])
  {
    match = true;
  }
  i++;
}
if(match)
  Console.WriteLine("Match");
else
  Console.WriteLine("No match");
```

Can also use `yourNums.Length`

Same counter variable used as index to both arrays

Outline

- Array basics and syntax
- Array initialization
- Loops and arrays
- **Array resizing**

Adding Values to an Array

- What if you want to add more to an existing array?

```
int[] homeworkGrades = {89, 72, 88, 80, 91};
homeworkGrades[5] = 89;
homeworkGrades = new int[6];
```

✗ IndexOutOfRangeException

✗ homeworkGrades refers to a new empty array, existing data is lost

- **Copy** to a new array:

```
int[] newHomeworkGrades = new int[6];
int i = 0;
while(i < homeworkGrades.Length)
{
  newHomeworkGrades[i] = homeworkGrades[i];
  i++;
}
newHomeworkGrades[5] = 89;
```

Array Resizing

- `Array.Resize` is a shortcut for this loop:

```
Array.Resize(ref homeworkGrades, 6);
homeworkGrades[5] = 89;
```

Copies homeworkGrades to a new array with size 6

Now this variable refers to the new array: `{89, 72, 88, 80, 91, 89}`

- `ref` parameter: similar to `out`, means "the method will change this variable"

```
int[] myArray = {1, 2, 3};
Array.Resize(ref myArray, 5);
```

Array to resize

New size

Array.Resize Method

- If new size is larger: new elements will have default value

```
int[] myArray = {1, 2, 3};
Array.Resize(ref myArray, 5);
```

→ myArray is now {1, 2, 3, 0, 0}

```
Rectangle[] shapes = {new Rectangle(14, 10),
  new Rectangle(19, 22)};
Array.Resize(ref shapes, 3);
```

→ shapes[2] is now null

- If new size is smaller: last elements in array are lost

```
int[] myArray = {1, 2, 3};
Array.Resize(ref myArray, 2);
```

→ myArray is now {1, 2}