

Value vs. Reference Types

The bit by bit details....

Yes, this was covered in GDAPS1. I'm covering it ALL again because it's REALLY important.

This underpins EVERYTHING else in this course!

Recap from last time...

And let's see how it all looks in the debugger as we go.

Do NOT try to type this and follow along. I'd rather you watch and ask questions.

I'll give you any new demo code later!

Value & Reference Types

Variables in C# fall into one of 2 categories:

- Value types
- Reference types

Which type we have impacts how we can use it!

Value types

The data itself lives where the variable lives

- Usually on the stack

The value is always **copied**

```
int a = 10;  
int b = a;    // The value is copied here  
b += 5;       // b is 15, a is still 10
```

Can *never* have the value null

Reference types

The data is always on the heap

- Even if the variable isn't

Multiple variables can refer to the same data

```
int[] nums = new int[5];  
int[] moreNums = nums; // moreNums now refers to the same  
                        // underlying array as nums  
  
moreNums[2] = 100; // changes the 3rd element  
                  // referred to by BOTH variables
```

May have a null value

- Called a null reference

I don't usually like end of line comments. In this case, you really need to read the code before the comment makes sense so EOL is appropriate.

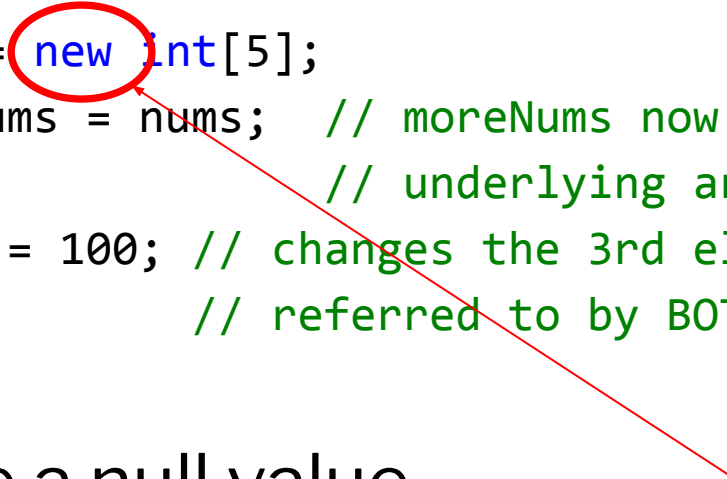
Reference types

The data is always on the heap

- Even if the variable isn't

Multiple variables can refer to the same data

```
int[] nums = new int[5];  
int[] moreNums = nums; // moreNums now refers to the same  
                        // underlying array as nums  
moreNums[2] = 100; // changes the 3rd element  
                  // referred to by BOTH variables
```



New keyword was only called once!

Only 1 object was created.

May have a null value

- Called a null reference
- The variable exists on the stack, but points to nothing. That's acceptable.

Reference types

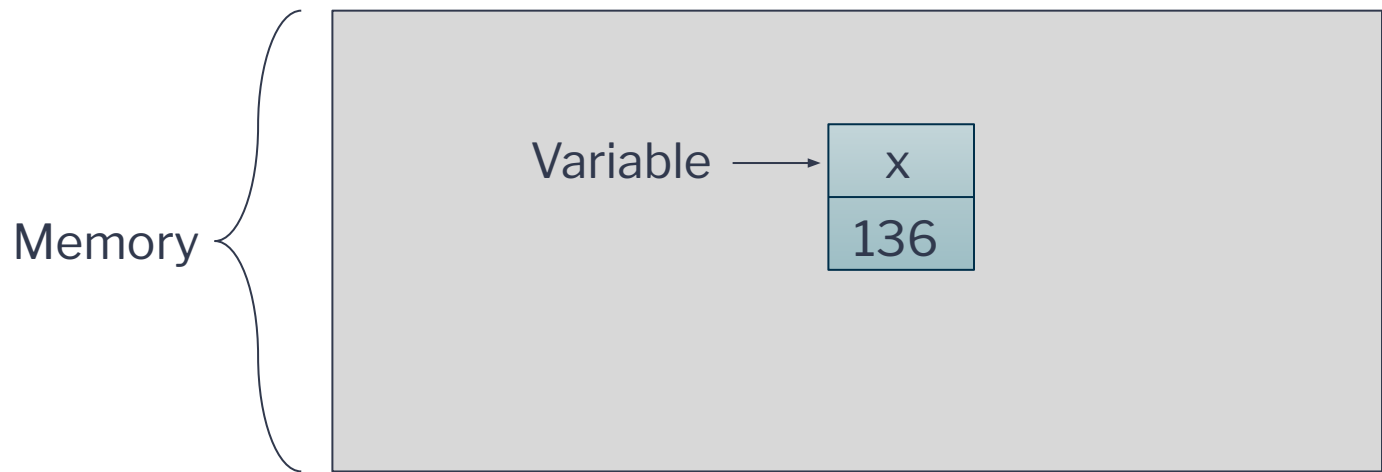
Objects (instances of classes) behave the same way!
(Pet class in the speaker notes)

```
Pet pax = new Pet("Pax", "dog", new DateTime(2018,11,5));  
Pet myDog = pax;  
  
pax.Name = "DOG!!!";  
Console.WriteLine(myDog);
```

Okay...but WHY do they behave so differently?

Conceptualizing memory

We often draw memory as a big box or grid



And variables as smaller boxes

- *With values in them*

A little more complicated

As you might expect...

- Memory is a little more complex than that
- As are variables

But we can conceptualize it similarly

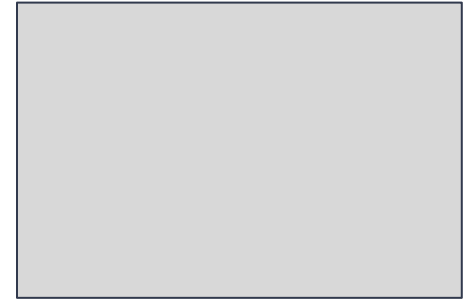
- With boxes and arrows

So it's time to delve a little deeper

Memory

When a program starts up...

- It's given a section of memory by the operating system



It then further divides up this memory:

- **Stack** memory
- **Heap** memory



The stack is usually visualized
taller than it is wide

The stack & the heap

The 2 main areas of any program's memory

Each is used for different things

- Certain variables are stored on the stack
- While others are stored on the heap

And each is organized differently

The stack

May have heard of StackTrace or Stack Overflow Exceptions

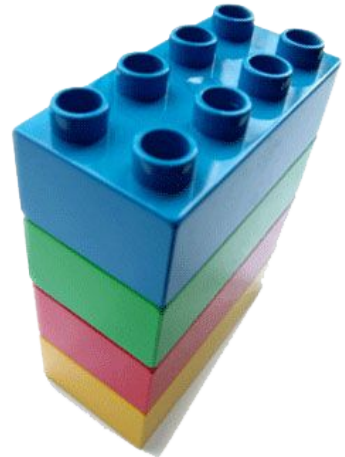
Tracks the current state of the program:

- Which methods are currently running
- And the local variables of those methods

Much smaller than the heap

Each time a new method is called

- A ***stack frame*** is created
- Placed on “top” of the stack



Stack frame

Stack frame: A section of the stack representing a method

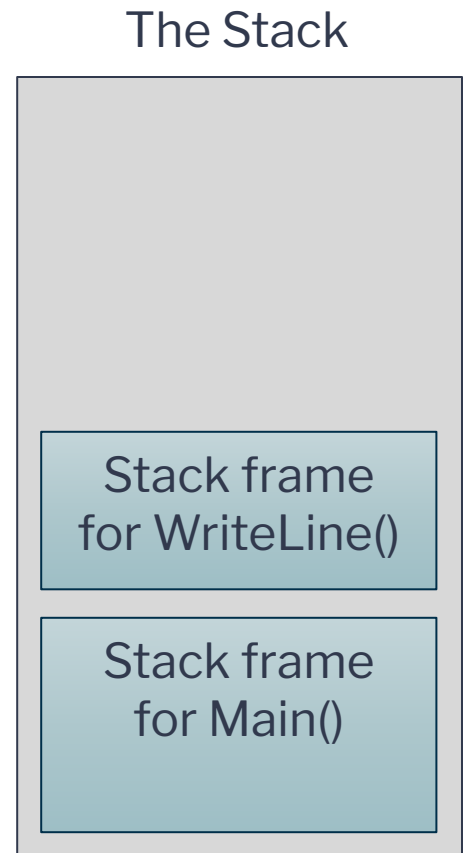
Each time a method is called:

- A new stack frame is created
- It's large enough to hold all variables

Each time a method ends:

- Its stack frame is removed

Frames are essentially “stacked”
on top of each other



What's in a stack frame?

All local variables & parameters for current method

Also tracks which line of code was running when method was called

Given the stack on the right, we know...

- The program is still inside Main()
- There are 5 local variables
- Plus the “args” string array parameter



Entering a method

When entering a new method...

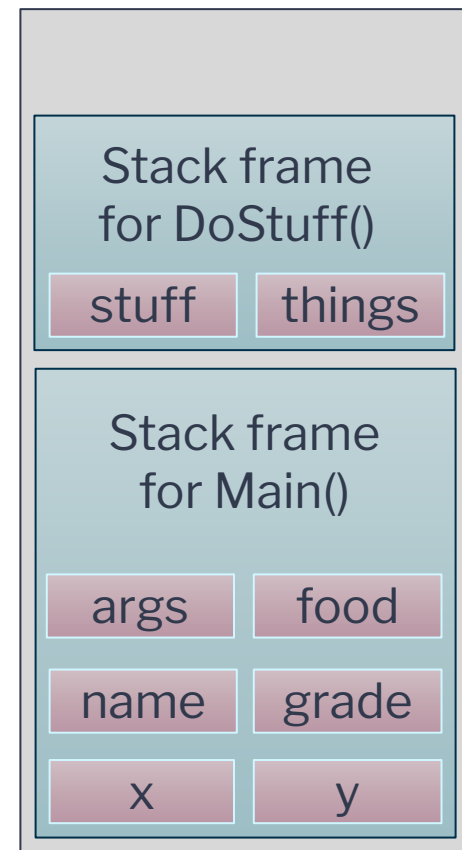
- Another stack frame is created
- With room for all of its variables & parameters

Example:

- Main() calls DoStuff()
- DoStuff() has two local variables:
 - stuff
 - things

```
StackFramesExample demoHelper = new StackFramesExample();  
demoHelper.RunDemo();
```

The Stack



Exiting a method

When exiting a method...

- Current stack frame is removed
- All local variables/parameters from that method are gone

Example:

- DoStuff() has ended
- Main() resumes where it left off



Why do we have a stack?

Reserve / allocate enough memory for all of the variables and params we might need

Stores all local variables & parameters

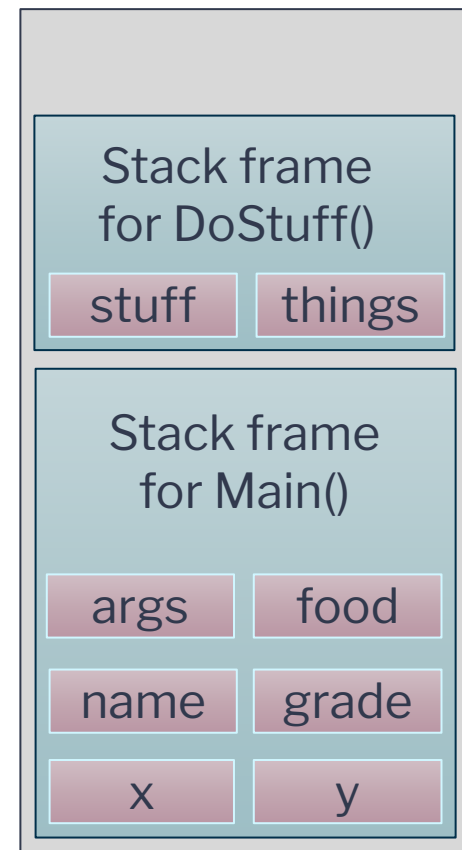
Tracks the current execution of the program

- Which methods are currently in progress
- Makes it easy to return once a method is over

Efficient to allocate new stack frames

- Each compiled method knows how “large” it is
- Data is placed starting at the bottom of the stack – we know how much empty space is “above” the current method
- Variables on the stack are very organized

The Stack



More about the stack

A program can only access data from the current stack frame (and the heap)

- This is why you can't access a variable from the Main method inside the DoStuff() method (i.e. this is how "scope" is enforced)

How big is the stack?

- 1 megabyte (by default)
- Usually enough for almost all programs

What happens if you run out of room?

- You have overflowed the stack!
- You get a Stack Overflow exception!
- Often from methods or properties calling themselves infinitely

By doing this:

```
public void Search()  
{  
    Search();  
}
```

The heap

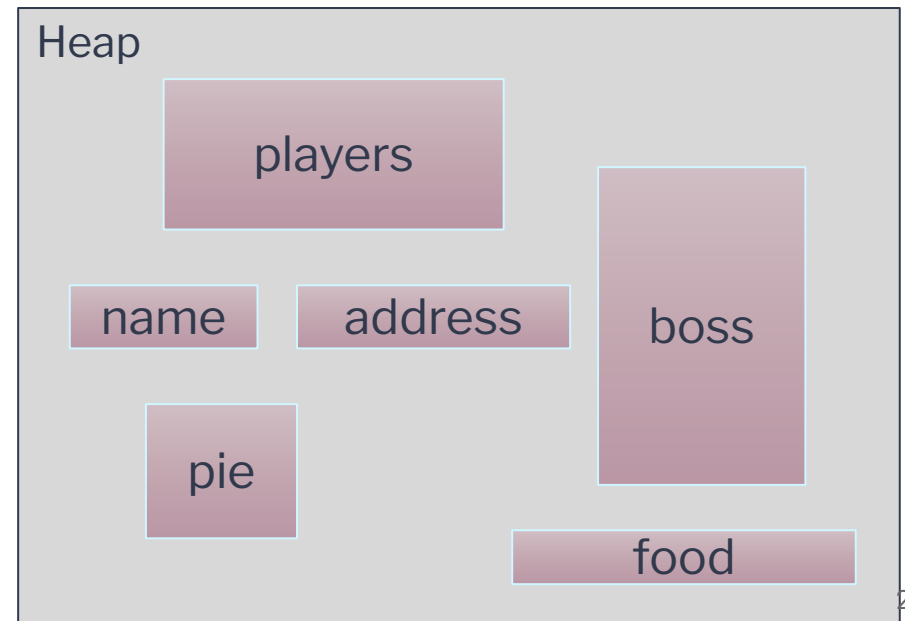
Only holds data (not program state)

Not much organization here

- Wherever it fits!
- Not sequential!
- Searches through the heap looking for the wanted data

A much larger area of memory than the stack

- Enough memory to hold all of your stuff



Managing memory

C# doesn't let us delete our own variables

Stack only data is automatically removed when leaving a method

Heap data is managed for you

- C# has a *Garbage Collector*
- Runs in the background of all C# programs
- Removes unused variables as necessary
 - “Oh, this hasn't been used? Cool. Let's get rid of it.”
- Sometimes rearranges memory, too
 - “Easier” to find

References

Most variables exist on the stack

- Sometimes the variable's data is also on the stack

But sometimes the data is actually stored on the heap

How are variables on the stack connected to data on the heap?

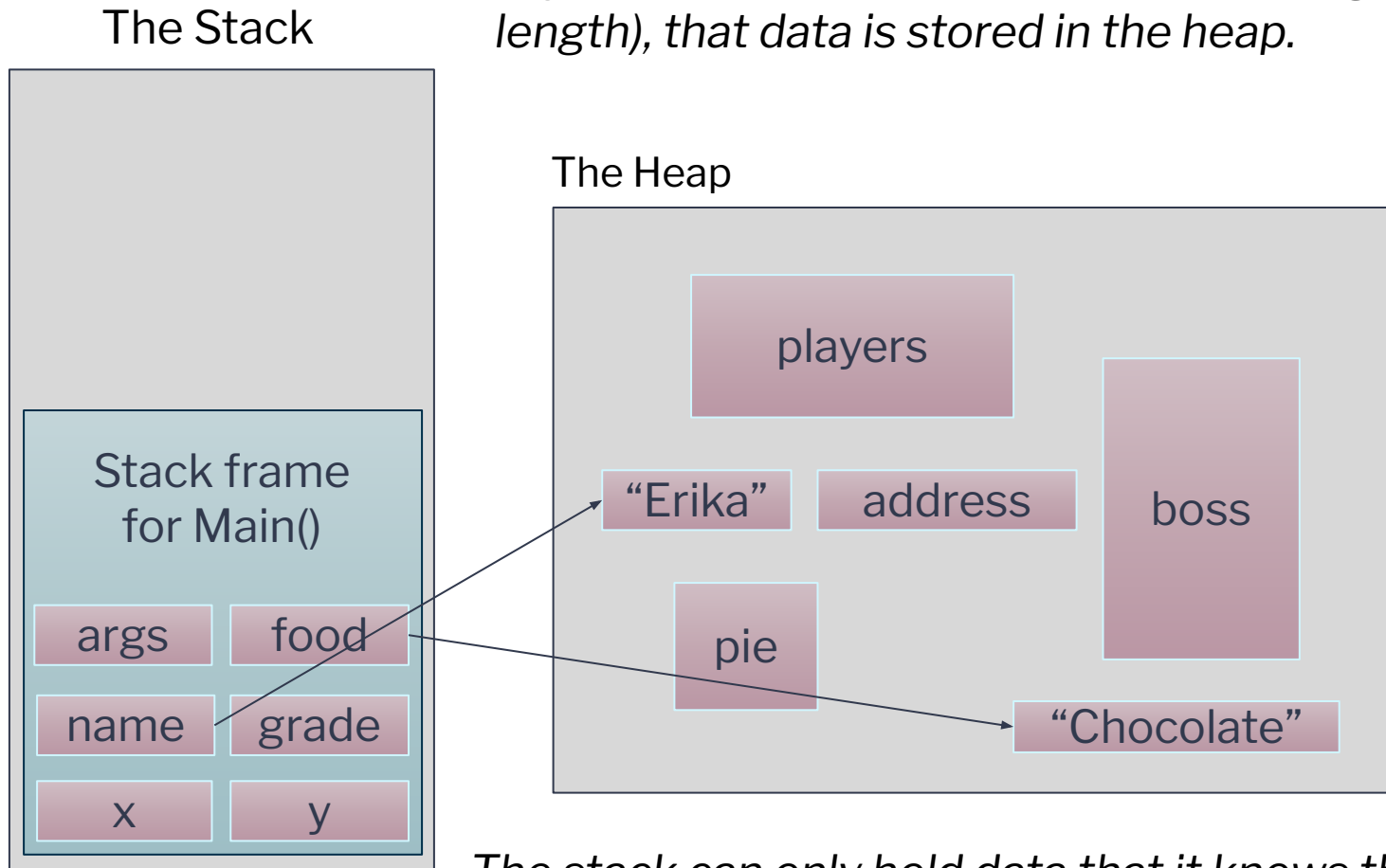
- Why does this matter?

A reference

- The variable *refers* to the data on the heap
- (This is similar to a pointer in other languages)

Reference example

If name is a String (which is actually an object, an array of characters with varying length), that data is stored in the heap.



The stack can only hold data that it knows the exact size of – like integers.

Why cover all of this?

Understanding how our data works is imperative!

Otherwise working with collections of data will be much harder!

- Such as arrays of objects!
- And, eventually, trees & graphs!