2D ARRAYS, CLASSES AND OBJECTS, PROCEDURAL DESIGN

BME 121 2016

Jeff Luo

Topics

- Review: If else, switch case, inline if statements
- Review: While and For loops
- Review: Random Number Generator
- 2D Rectangular Arrays
- Advanced String Formatting
- Classes and Objects
- Reference Types and Reference Parameters
- Procedural Design
- WA4
- PA2

If else, switch case, inline if statements

Final value must be a string or int

```
if (  // test1  )
{
    // test1 == true
}
else if ( // test2 )
{
    // test1 == false && test2 == true
}
else
{
    // otherwise
}
```

Subsequent tests only run
if previous test == false

```
switch (  // expression )
{
    case value1:
        // expression == value1
        break;
    case value2:
        // expression == value2
        break;
    default:
        // otherwise
        break;
}
```

Inline If Statement:
```
// test ? // true value : // false value
```
Example:
```
int x = 5 > 10 ? 15 : 30   ;
```

While and For Loops

- For Loops: repeat for some countable number of times

```
// As long as n < 10, keep
repeating:
for(int n = 0; n < 10; n++)
{
   WriteLine("Counter: {0}", n);
}
```

- While Loops: repeat while some condition holds

```
int age = 0;
// As long as age <= 0 or age >=
200, keep repeating:
while(age <= 0 || age >= 200)
{
   age = int.Parse(ReadLine());
}
```

While and For Loops

The compiler translates for loops into while loops, then into CPU code:

```
// As long as n < 10, keep repeating:
for(int n = 0; n < 10; n++)
{
   WriteLine("Counter: {0}", n);
}
```

```
int n = 0;
// As long as n < 10, keep repeating:
while(n < 10)
{
   WriteLine("Counter: {0}", n);
   n++;
}
```

The counter modification is always put into the while loop as the last line(s) of code.

same

Infinite Loops

- These Loops will never stop

```
while(true)
{
  // Code
}

for( ; true; )
{
  // Code
}
```
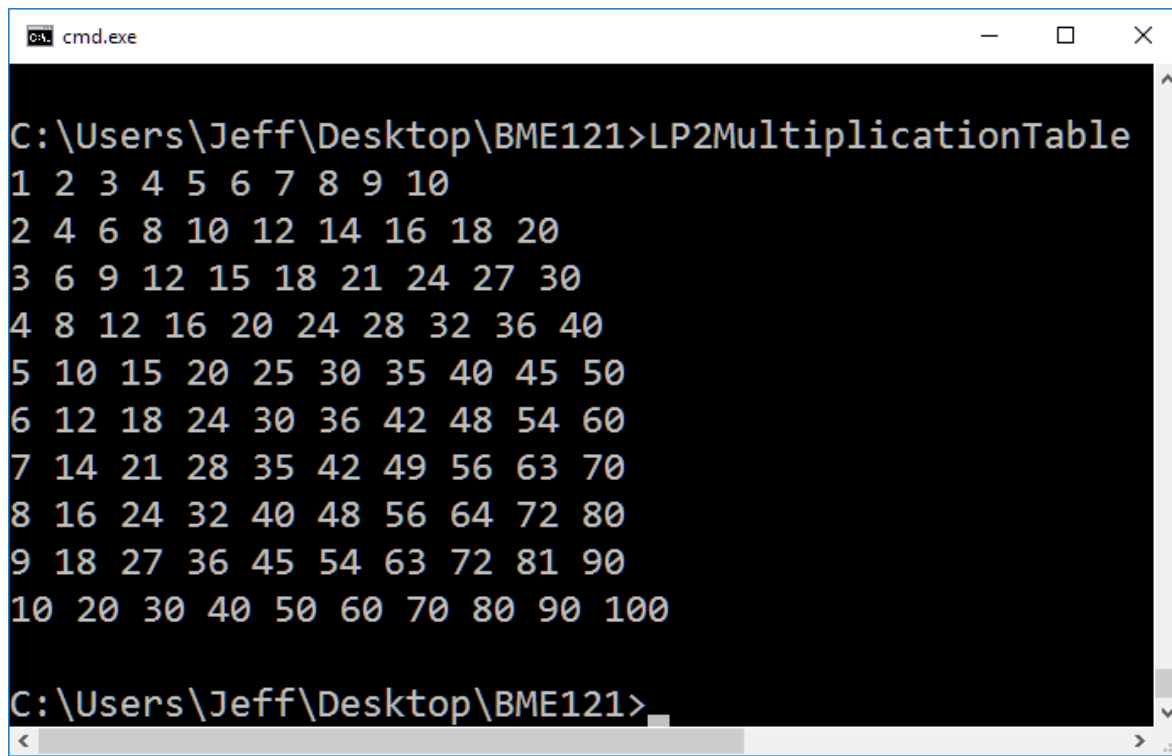

- Infinite Loops are sometimes useful
  - Operating System Software
  - Clocks, Pacemakers, …
  - Declare your amazingness
- In Command Prompt, you can forcefully terminate any program by the key combination Control + C (Mac: Command + C), or by pressing the red X.
- This is one way to stop an infinite loop

Practice 1: Display the 10 x 10 Multiplication Table (print 1 space after each number)

Random Number Generator

- Class Random gives us an artificially random number generator
  - To have a true random number generator, we must make observations of a real world random phenomenon: eg weather, quantum effects, etc.
- Creating a copy of the generator:

Random rand = new Random();

// rand will be our generator

- Rule: each program should only have 1 copy of class Random.

- Artificial RNG:

- first random number = f(seed number)
  - seed number is usually a function of the present computer time in milliseconds

- next number = f(previous number)
  - we can find out the next random number in any situation if we know the previous one, or the seed

Random Number Generator

- Creating a generator with current time as the seed:

```
Random rand1 = new Random();
```

- Creating a generator with a specific seed:

```
int seed = 12345;
Random rand2 = new Random(seed);
```

- Generating random integers between two bounds:
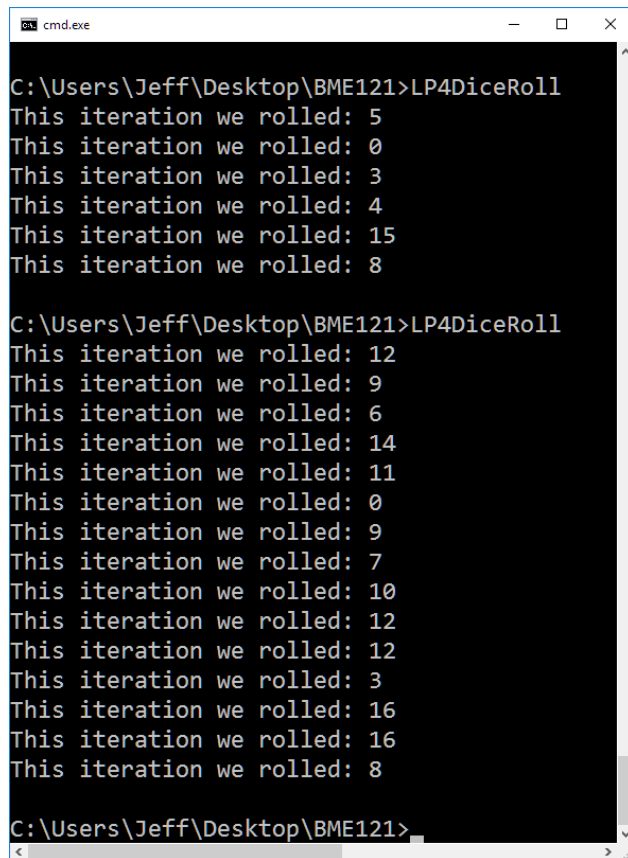
```
// range: [low, high)
rand1.Next(low, high);
```

- Generating random doubles:

```
// range: [0.0, 1.0)
rand1.NextDouble();
```

Practice 2: Lucky Dice Roll

- Write a program which will continuously roll a 20-face dice until we get the number 8, then exit the program.

- Display each roll along the way

2D Arrays

• Helpful to draw boxes to visualize:

  string[,] array = new string[3,4];

Columns

	0	1	2	3
0	"hi"			
Rows 1				
2				

Array Initializer

array[0]   array[1]

int[] array = new int[5] { 10, 20, 30, 40, 50 };

int[,] array2 = new int[5,2] {
        {10, 20},
        {30, 40},
        {50, 60},
        {70, 80},
        {90, 100}
};

10	20
30	40
50	60
70	80
90	100

Outer pair of curly brackets enclose the rows,
Inner pairs of curly brackets enclose the columns of a row

Practice 3: Array Initializer

- Using Array Initializer Syntax, create a 2D int array which stores the 6 x 3 multiplication table.
- Then, using two for loops, display the 6 x 3 table (formatting is flexible).

Advanced Printing and Formatting Part 1

- `Write()` only prints stuff to screen at the current "cursor" position
  - If the contents of the current line exceed 80 characters, it will move onto the next line and continue where it left off
- `WriteLine()` will print stuff to screen and insert a "new line" character at the end
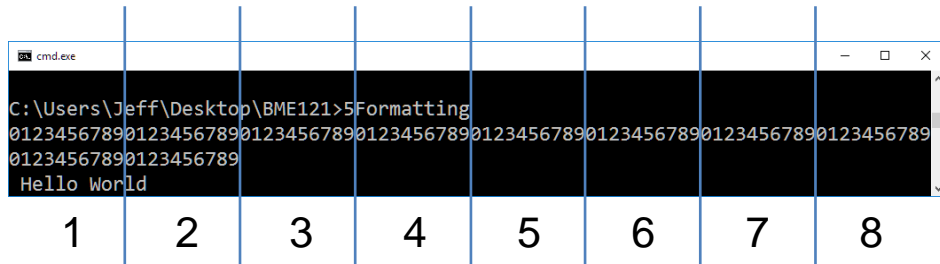 - New Line (`\n`): advances the cursor by 1 line, and shifts it all the way to the left
  - Using `Write`("\n") accomplishes the same thing as `WriteLine`("").

```
static void Main( )
{
    for(int i = 0; i < 10; i++)
    {
        Console.Write("0123456789");
    }
    Console.Write("\n Hello World");
}
```



```
cmd.exe                                                    —  □  ×
C:\Users\Jeff\Desktop\BME121>5Formatting
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789
 Hello World
```

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Advanced Printing and Formatting Part 2
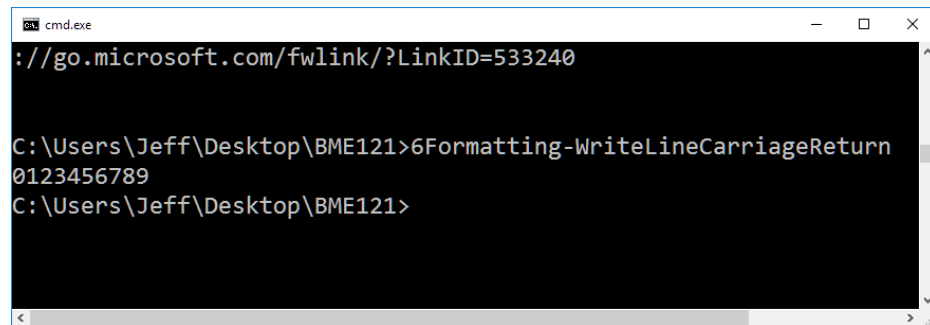
- You can write over the current line by adding the "Carriage return" character (\r) at the end of line:
Console.Write("\r")

  - "Carriage return" only repositions the cursor all the way to the left of the screen, on the same line

  - The name comes from old mechanical typewriters, where the carriage moved to the left as you typed each line of text, and the "return" moved the carriage all the way back to the right (left to right typewriters)

type guide (aligns type);
ribbon vibrator (raises & lowers ribbon)

carriage release
lever

platen

paper finger

ribbon spool

keytop

typebars

Carriage

shift
key

space bar

Advanced Printing and Formatting Part 2

```
static void Main( )
{

  for(int i = 0; i < 10; i++)

  {

    Console.Write("0123456789\r");

  }

}
```


What happens if we use this instead?
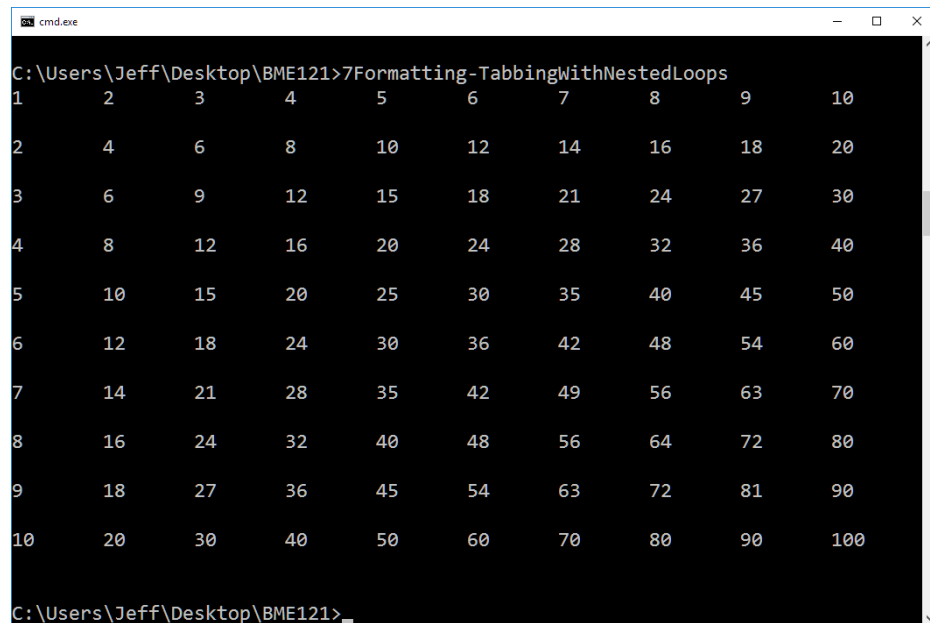
```
Console.WriteLine("0123456789\r");
```

Advanced Printing and Formatting Part 3

- To align content by Tabs, like in Microsoft Word, use \t

```csharp
// 10 x 10 Multiplication Table
static void Main( )
{
    for(int i = 1; i <= 10; i++) // outer loop
    {
        for(int j = 1; j <= 10; j++) // inner loop
        {
            Console.Write("{0}\t", i * j);
        }
        // Escape each line
        Console.Write("\n");
    }
}
```

Placeholder Field Spacing and Alignment

```
static void Main( )
{
   WriteLine("{0,10:D}|", 1234);    // default right aligned
   WriteLine("{0,10:D6}|", 1234);   // zero-padded to 6 decimal places
   WriteLine("{0,-10:D}|", 1234);   // left aligned
   WriteLine("{0,-10:D6}|", 1234);  // left aligned & zero-padded
}
```


Note: All four of these specifiers take up 10 characters of space:

{ID,space:L#} where L is the specifier, and # is the number of digits of rounding

minus sign means left aligned, by default every number is right aligned

More details and examples on this:
Leading zeros, number alignment: http://msdn.microsoft.com/en-us/library/dd260048(v=vs.110).aspx
Standard Number Format Strings: http://msdn.microsoft.com/en-us/library/dwhawy9k(v=vs.110).aspx
Custom Number Format Strings: http://msdn.microsoft.com/en-us/library/0c899ak8(v=vs.110).aspx

Number Format Specifiers

Specifier	Letter	Supported Numbers	Example	Number	Sample Output
General	G	All	{0,8:G}	1234.5F	**1234.5
			{0,8:G}	1234	****1234
Fixed Point	F	All	{0,8:F2}	1234.5F	*1234.50
Round Trip	R	All	{0,8:R2}	1234.5F	**1234.5
Number	N	All	{0,8:N1}	1234.5F	*1,234.5
Exponential	E	All	{0,14:E6}	1234.5F	*1.234500E+003
Decimal	D	Integers Only	{0,8:D}	1234	****1234
Currency	C	All	{0,10:C}	1234.5	*$1,234.50
Percentage	P	All	{0,8:P}	0.89	*89.00*%
Hexadecimal	X	Integers Only	{0,8:X}	1234	*****4D2


```
cmd.exe

C:\Users\Jeff\Des
    1234.5
      1234
1234.50
    1234.5
1,234.5
1.234500E+003
      1234
$1,234.50
89.00 %
      4D2
```
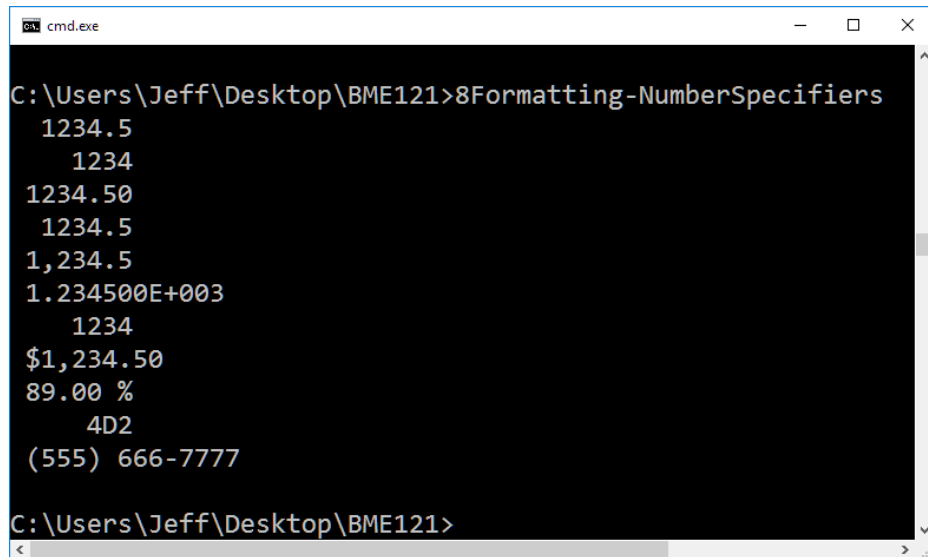
NOTE: * means space. By default, right alignment is often used for formatted numbers with extra space. A minus sign can be used for left alignment.

Number Format Specifiers

```
static void Main( )
{
    Console.WriteLine("{0,8:G}", 1234.5F);  // General
    Console.WriteLine("{0,8:G}", 1234);      // General2
    Console.WriteLine("{0,8:F2}", 1234.5F); // Fixed Point
    Console.WriteLine("{0,8:R2}", 1234.5F); // Round Trip
    Console.WriteLine("{0,8:N1}", 1234.5F); // Number
    Console.WriteLine("{0,14:E6}", 1234.5F);// Exponential
    Console.WriteLine("{0,8:D}", 1234);      // Decimal
    Console.WriteLine("{0,10:C}", 1234.5);   // Currency
    Console.WriteLine("{0,8:P}", 0.89);      // Percentage
    Console.WriteLine("{0,8:X}", 1234);      // Hexadecimal

    // Custom Formatting:
    // Phone Number
    // # is a placeholder for a single digit
    Console.WriteLine("{0,15:(###) ###-####}",
        5556667777D);
}
```
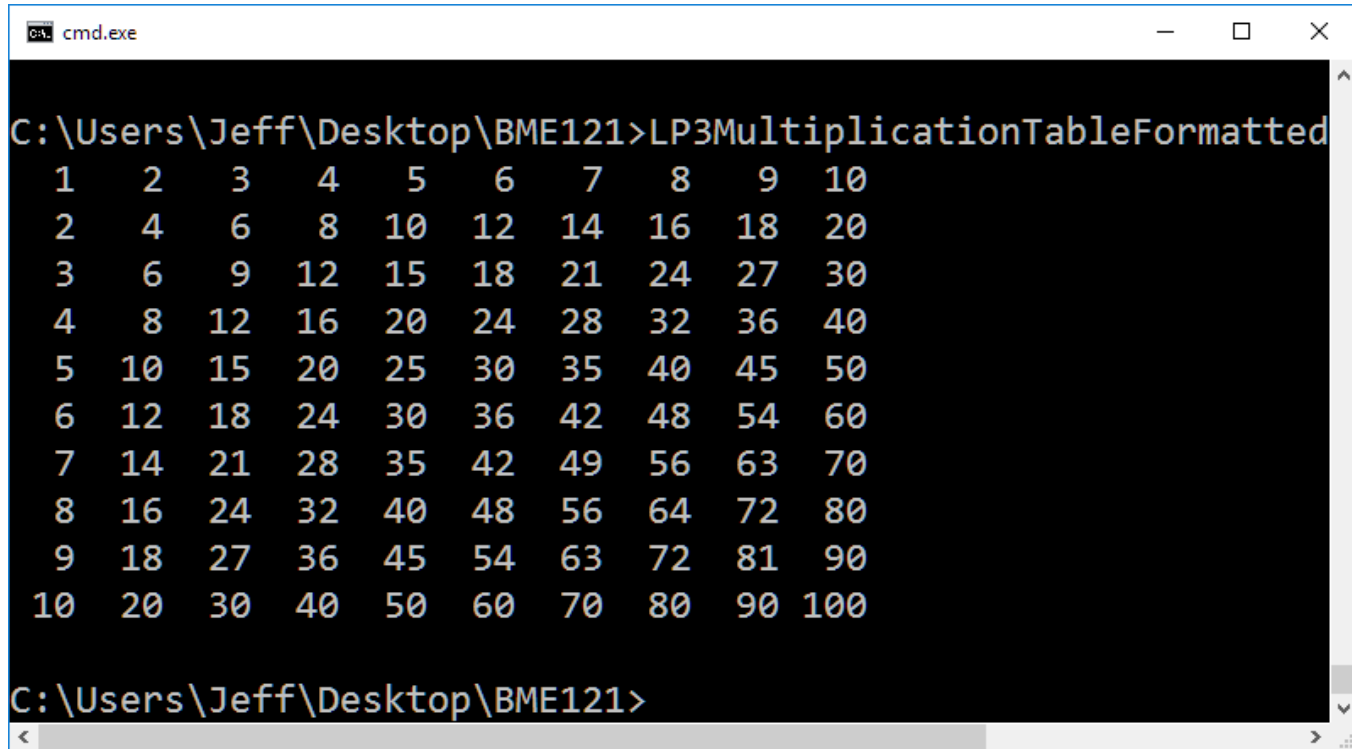
```
C:\Users\Jeff\Desktop\BME121>8Formatting-NumberSpecifiers
  1234.5
    1234
1234.50
  1234.5
1,234.5
1.234500E+003
    1234
$1,234.50
89.00 %
    4D2
(555) 666-7777

C:\Users\Jeff\Desktop\BME121>
```

Practice 4: Display the 10 x 10 Multiplication Table, Formatted Nicely

WASHROOM BREAK

Classes & Objects

- A Class is a template which defines a group of variables (called fields), and methods that work on those variables
- Each copy of this template is an Object, which stores some unique set of data in the fields

- Closest Non-programming Analogy:
  - Class: a registration form (template, & rules)
  - Object: an individual filled registration form (which has a specific unique set of data)

Simple Event Registration
Fill out the form to register to our event

Name
First Name   Last Name

Title

Company

E-mail   ex: myname@example.com

Phone Number   -
Area Code Phone Number

Are you an existing customer?
○ Yes   ○ No

Code Structure of a Class

```
class ClassName // always Capitalized and Camel Case
{
  // fields

  // constructor method

  // other methods
}
```

Class Definition Example

```
class RegistrationForm
{
  // fields
  string firstName;
  string lastName;
  string title;
  string company;
  string email;
  int areaCode;
  int phoneNumber;
  bool existingCustomer;
  // …
}
```

Simple Event Registration
Fill out the form to register to our event

Name
First Name   Last Name

Title

Company

E-mail
ex: myname@example.com

Phone Number
-
Area Code Phone Number

Are you an existing customer?
○ Yes   ○ No

Defining a Class

```
class Grumpy
{
  private string name;
  private string favQuote;

  public Grumpy() {
    this.name = "GrumpyCat";
    this.favQuote = "No";
  }

  public string GetQuote() {
    return this.favQuote;
  }
}
```

Methods in a Class

- So far, methods:
  - Behave like math functions (takes some values as input, computes then returns some value as output, concept of substitution)
  - Group up repeated code (allows easy code reuse)
  - Assist Main() by breaking down a big problem into steps
- Methods for a Class have additional purposes:
  - Constructor Method: a special method that should ensure every field has a (default) value when an object is created
    - Has no return type in the method, because it creates the object
  - Setter Methods: methods whose purpose is to update a field with a new value
  - Getter Methods: retrieves a value stored in a field and gives it to the caller
  - Other: can also be helper methods to any of the 3 above purposes

Constructor, Setter, and Getter Methods

```
// constructor
public Grumpy() {
    this.name = "GrumpyCat";
    this.favQuote = "No";
}


// setter
public void SetQuote(string quote) {
    this.favQuote = quote;
}


// getter
public string GetQuote() {
    return this.favQuote;
}
```

Constructor Method's name must always be the same as the name of the class

```
// fields
private string name;
private string favQuote;
```

Methods in a Class

```
Grumpy cat;   // creates a variable to store a Grumpy object
cat = new Grumpy(); // runs the constructor method to create
the Grumpy object and then stores it in variable cat


cat.GetQuote(); // runs the GetQuote method on the cat object


cat.SetQuote("Nope"); // runs the SetQuote method on the cat
object, updating the field favQuote of this object to "Nope"
```

"this" Keyword

- In our programs, only 1 method is active at any given moment
- The methods defined for a class works for every object of that class, but only 1 object at a time
- "this" means the current object we are working on

```
Grumpy cat1 = new Grumpy();
Grumpy cat2 = new Grumpy();
cat1.SetQuote("Nope"); // "this" refers to cat1
cat2.SetQuote("Go away"); // "this" refers to cat2 at this point in
the program
```

Practice 5: Defining a Class

- Define a class called Sphere which stores the radius of a perfect sphere as a double field
- Create the following methods:
  - A constructor method which sets the initial radius of a sphere
  - A method which updates the stored radius to a new value we pass in
  - A method which retrieves the stored radius
  - A method which updates the stored radius by doubling it
  - A method which computes and returns the volume of the sphere
  - A method which computes and returns the surface area of the sphere
- Feel free to Google the math formulas
- Come up with good method names!

CPU Architecture
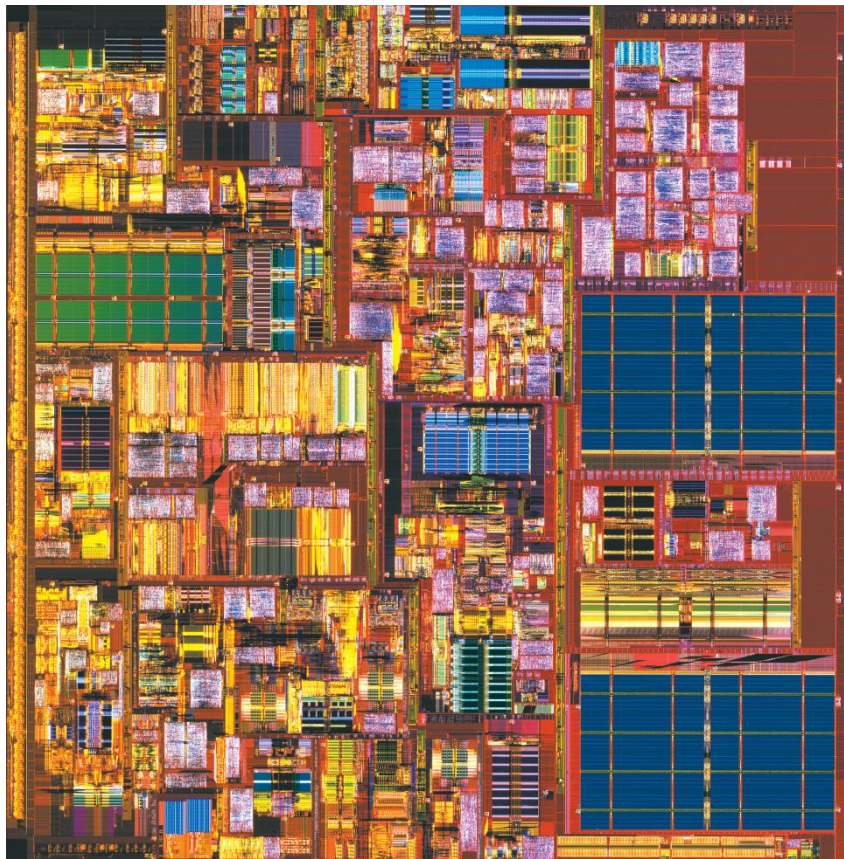
- 95%+ of the surface area of a CPU is computation circuitry
- CPU has some memory, divided into 4 tiers:
  - Registers: 16 memory cells, 64 bits each
  - L1 Cache: 256 KB
  - L2 Cache: 1MB
  - L3 Cache: 8MB
- L1 to L3 Cache used to store very frequently used code or data
- All of the data is moved back and forth between CPU and RAM

Value Types vs Reference Types

- CPU can't fit a large amount of data in its 16 registers
- Value Types:
  - designed so that it's value can fit within a register cell
  - int, double, bool, char, *string**
- Reference Types:
  - designed to fit it's value in a sequence of memory cells (in L1 to L3 Cache, or RAM), the CPU examines one part of this sequence at a time
  - arrays, classes

* a string is technically a char[], ie a reference type, but C# does fancy magic to make it work like a value type

Value Types vs Reference Types

- Value Types:
  - Value variables store values
  - `int x = 20;`

- Reference Types:
  - Reference variables store the memory address of an array or object
  - Always created using new keyword
  - New tells the computer to allocate a sequence of memory cells
  - `int[] y = new int[5];`
  - `SimpleName a = new SimpleName();`

x
20

y
&320012030

Value 0
Value 1
...

a
&320023400
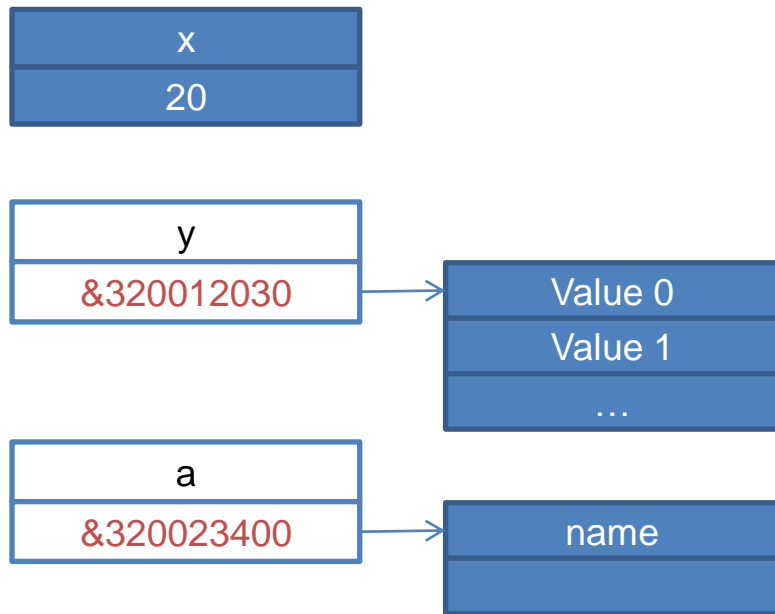
name

Value Types vs Reference Types

- Value Types:
  - `int x = 20;`
  - `int y = x;`   `// copies the value in x to y`

x
20

y
20
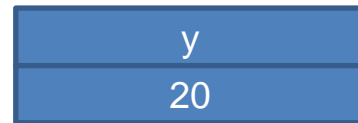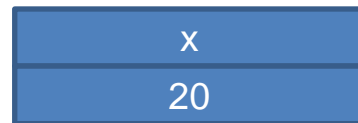
- Reference Types:
  - `SimpleName a = new SimpleName("Grumpy");`
  - `// copies the memory address`
  - `SimpleName b = a;`
  - `// we call this "b refers to the same object as a"`

a
&320012030

b
&320012030
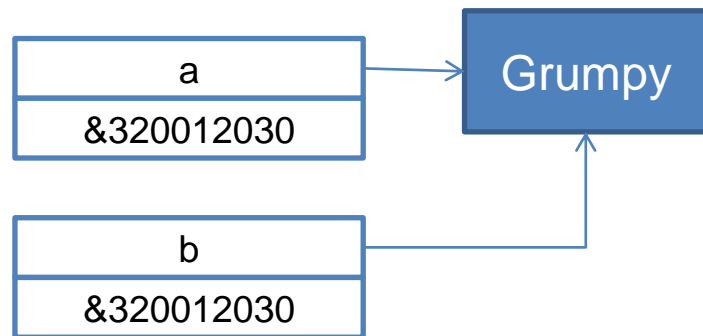
Grumpy

Reference Types and Objects

- What happens to the memory that held the SimpleName object with the value of "Grumpy" after we execute the 3rd line of code?

```
SimpleName x;
x = new SimpleName("Grumpy");
x = new SimpleName("Cat");
```
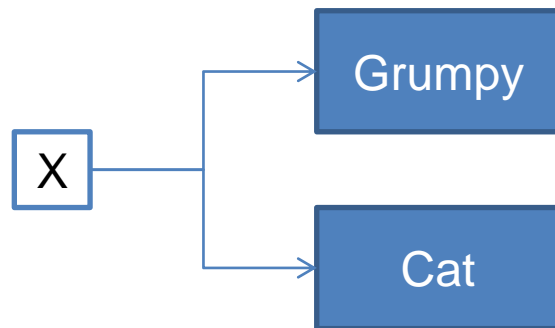
Pass by Value vs Pass by Reference

- Data is passed into a method either by value or by reference, depending on whether it is a value or reference type:

```
void MyMethod(int x, int[] y) {
    // x is passed by value, y is passed by reference
    x = 50;
    y[0] = 20;
}
void Main() {
    int a = 30;
    int[] b = new int[1] {60} ;
    MyMethod(a, b);

}
```

After MyMethod is executed, a stays the same, but b now has the value of 20 in the single cell of the array.

Global Variables, Local Variables, and Fields

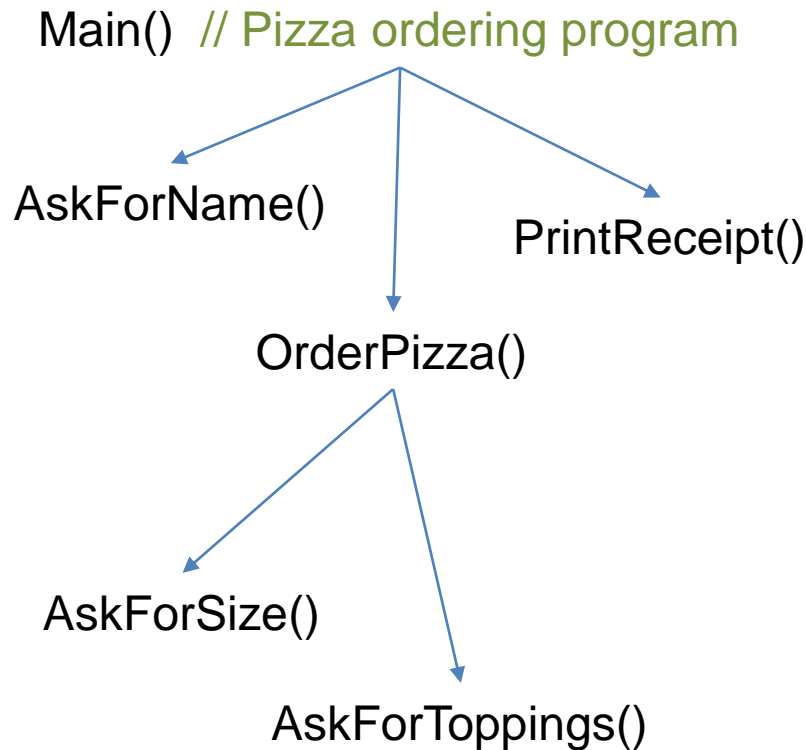- Fields of a class are also global variables for the methods of that class
  - But not global variables for other classes
- Variables inside a method are local variables (including parameter variables)

```
class Cat {
    // Fields, and Global Variables
    // for this class
    int height;

    // otherCat and x are local
    // variables to the Fight method
    void Fight(Cat otherCat) {
        int x = 20;
    }
}
```

WA4

Procedural Software Design

- Procedure: a series of actions conducted in a certain order or manner

- Procedural Design:
  - Breaking down a large problem into a set of (sub)procedures, each solving a smaller piece of the problem, and when executed together in the correct order, can solve the large problem

- Each procedure is a method
- Main() → main procedure

Main() // Pizza ordering program

AskForName()

PrintReceipt()

OrderPizza()

AskForSize()

AskForToppings()

Procedural Design

- Divide and conquer a complex problem

- Allows us to focus on one part of the problem at a time

- 3 Major components:

  - Data Structures: The variables you choose to model and store the data of the problem

  - Procedures: The methods in the program whom each solve a portion of the larger problem

  - Main Procedure: The overall procedure, which will call each of the (sub)procedures to solve the larger problem

Procedural Design

- The goal is to break down the problem into as many smaller problems as possible

- The more independent and reusable methods we can write, the more portable and adaptable our code will be to future projects

- EG: Our own custom MyMath library with:
  - bool IsEven(int x)
  - double CubicRoot(double x)
  - double ShortPI = 3.14d;
  - …

- One big benefit: easier to debug!

Practice 6: Procedural Design

- Break down each of these problems into a main procedure, and a set of smaller procedures:
  - Making Bacon Pancakes
  - Playing Tic Tac Toe
- For each procedure, write a description of what it does and how it should be used (no need to code this)

- Checklist:
  - has your main procedure utilized every smaller procedure you've come up with?
  - will the main procedure solve the problem?
  - are there any procedures which can be reused?

Video Games as a Procedure

- A video game generally has these procedures:

1. Set up the game
2. Get a player's input
3. Update the board
4. Check for win or tie
5. Switch players
6. Repeat 2-5 until someone wins or ties

PA2 – Connect 4

- Classically played on a 6 x 7 board, where players alternatively take turns to drop coins vertically from the top until one player gets 4 in a row, column, or diagonal:

PA2 – Connect 4

- We've broken down the Connect 4 game into a set of procedures, each which accomplishes a part of the tasks required to run a game

- This version includes an AI opponent that will randomly select 1 of the 7 columns to drop a coin, which isn't a very clever one

- Your main task is to write 6 of the methods
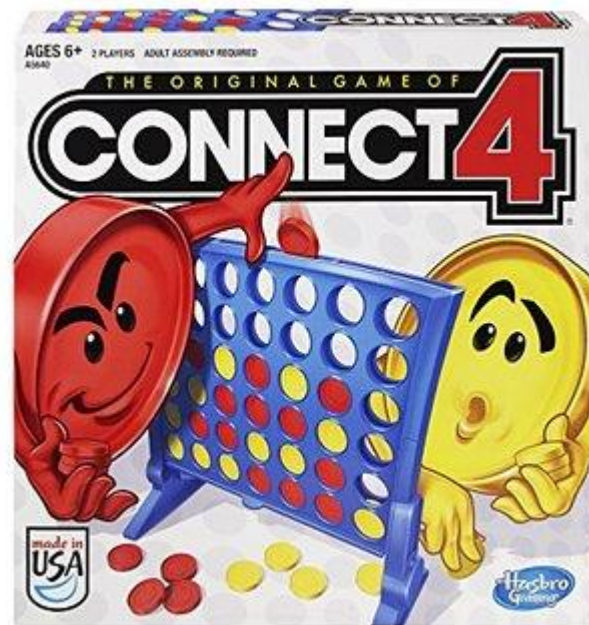
PA2 – Connect 4

- The board is declared as a field as follows:

- ```
static string[,]
gameBoard = new
string[6,7];
```

- The number of rows and columns are given to you through the fields gameRows and gameCols

- Each cell of the 2D array should have one of 3 values:
  - "X", "0", or null
  - null (no quotes) represents an empty coin slot



Coins drop in this direction

Column 0          Column 6

Row 5

Row 0
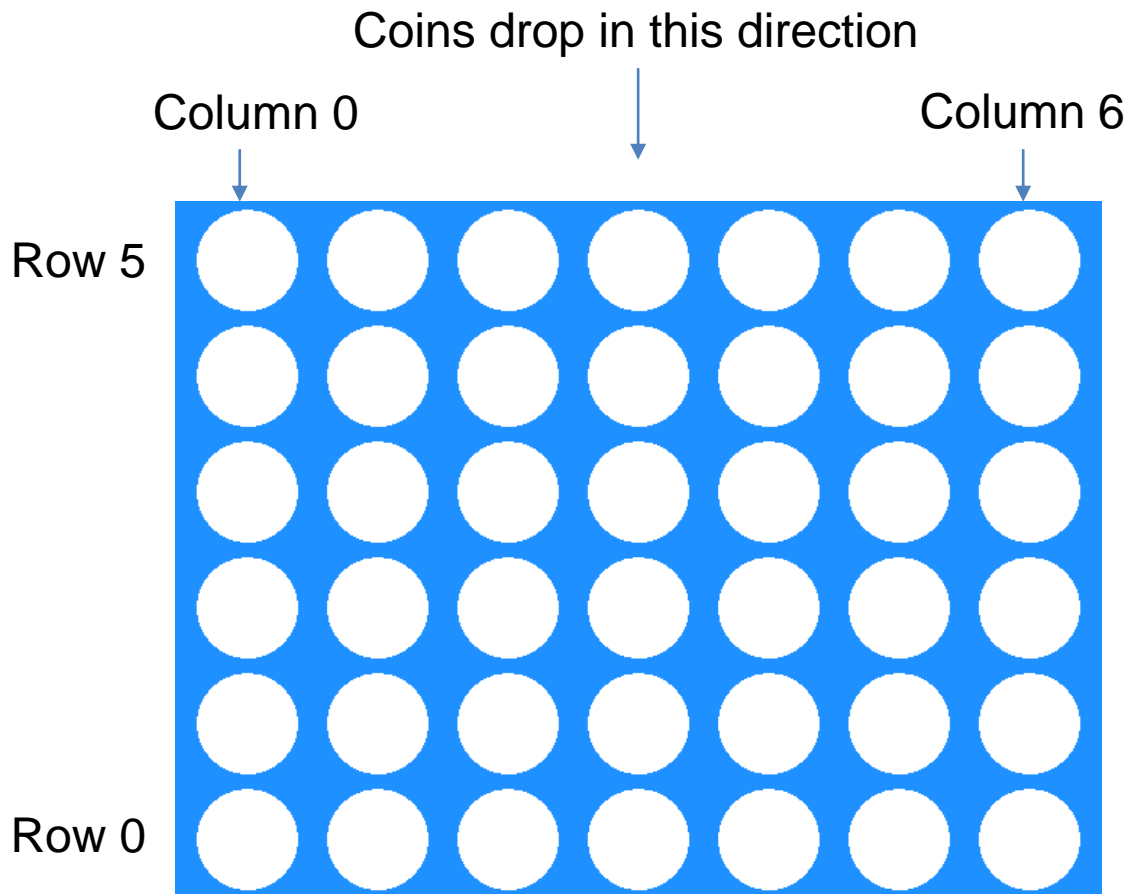
# PA2 Hints and Tips

- `SwitchPlayers():`
  - Update the variables `currentPlayer`, `currentPlayerName`, and `currentPlayerKind` using the values stored in `xName`, `xKind`, `oName`, and `oKind`
- `SelectRandomColumn():`
  - Use the `randomNumberGenerator`
- `PlayInColumn(int col):`
  - Loop through the rows from bottom row to top, when you find one empty cell at the given column, update it to store the symbol of the player
- `IsValidPlay(int col):`
  - A play in the given column is valid if the top cell is empty

- `IsBoardFull():`
  - The board is full if the top row is not empty
- `CurrentPlayerWins():`
  - It might help to break this procedure into 4 smaller procedures:
    - Check 1 Row
    - Check 1 Column
    - Check 1 Up-left diagonal
    - Check 1 Up-right diagonal
  - Each of these checking for a win in the corresponding direction, 1 row / column / diagonal at a time