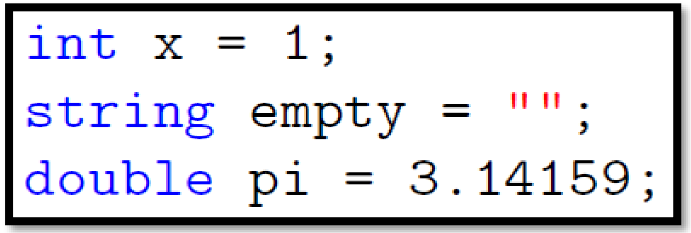
CHAPTER 3: VOID METHODS

Fall 2019 – CSC 180 – Introduction to Programming

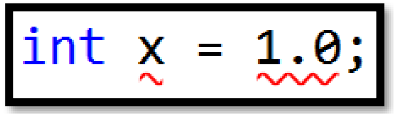
3.1 FLOATING-POINT

- a combined **declaration** and **assignment** is sometimes called an **initialization**.

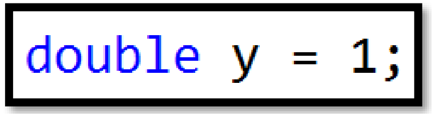


```
int x = 1;
string empty = "";
double pi = 3.14159;
```

- C# distinguishes the integer value 1 from the floating-point value 1.0, even though they seem to be the same number.
  - They belong to different types, and strictly speaking, you are not allowed to make assignments between types.
  - You cannot assign a **double** value to an **int** variable
  - You should not be able to assign an **int** to a double **value** but C# will automatically convert it for you …
    - **int**s are converted into **double**s automatically if necessary, because no information is lost in the translation.

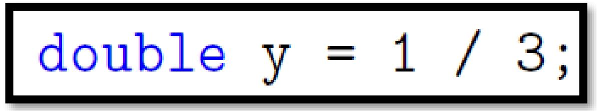


```
int x = 1.0;
```

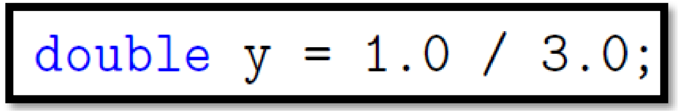


```
double y = 1;
```

- Discussion on division (important!):

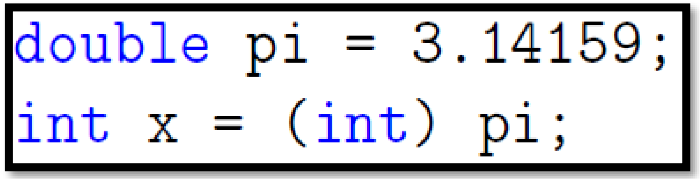


```
double y = 1 / 3;
```



```
double y = 1.0 / 3.0;
```

- To convert from **double** to **int** you can use type **casting**:
  - Beware of the lost information!
  - Type casting takes precedence over arithmetic operations



```
double pi = 3.14159;
int x = (int) pi;
```

CASTING

- The process of converting a value of one data type to another type is called type conversion or casting.

- Implicit conversion (when a value is converted automatically):

```
int a = 4;
long b = 5;
b = a;              //implicit conversion – in here it is widening
```

- Explicit conversion:

```
int a = (int) b;    //explicit conversion
```

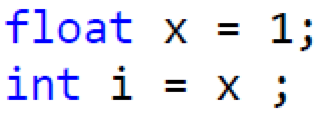
- System.Convert conversion:

```
string possibleInt = "1234";
int count = Convert.ToInt32(possibleInt);
```

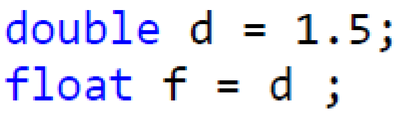
WIDENING AND NARROWING – SKIP

- Whenever we move a value from one type to another the C# compiler will check their types. It considers every operation in terms of "widening and narrowing" values.

- The general principle which C# uses is that:
  - if you are "narrowing" a value it will always ask you to explicitly tell it that this is what you want to do.
    - The following could case an error when compiled:

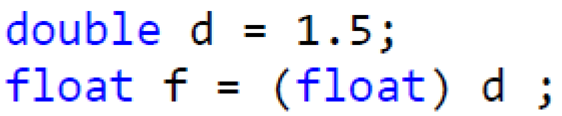


```
float x = 1;
int i = x ;
```



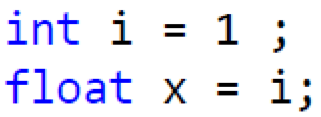
```
double d = 1.5;
float f = d ;
```

    - We could force C# to regard a value as being of a certain type by the use of *casting*



```
double d = 1.5;
float f = (float) d ;
```

  - if you are widening there is no problem.



```
int i = 1 ;
float x = i;
```

CONSOLE INPUT/OUTPUT

- To display to the console use Console.Write and Console.WriteLine methods
  - Console.Write(“Hello World!"); //displays a string literal
  - Console.WriteLine("Your input: {0}",a); //displays a string literal and a variable
  - Console.WriteLine("The two numbers were: {0}, {1}",a, b); //displays a string literal and two variables
  - Console.WriteLine("The two numbers were: {a}, {b}"); //displays a string literal and two variables
    - Console.WriteLine("The two numbers were: " + a + ", " + b); //same as above, but less optimal

- To read input from the console (from the user) we can use the following:
  - val = Console.ReadLine(); //reads an entire line from the console and saves it into **val**
  - int a = Convert.ToInt32(val); //converts a value (from variable **val**) to an integer and saves it into **a**
  - double b = Convert.ToDouble(val); //converts a value (from variable **val**) to a double and saves it into **b**

- See also this web resouce: https://www.programiz.com/csharp-programming/basic-input-output

3.3 MATH METHODS

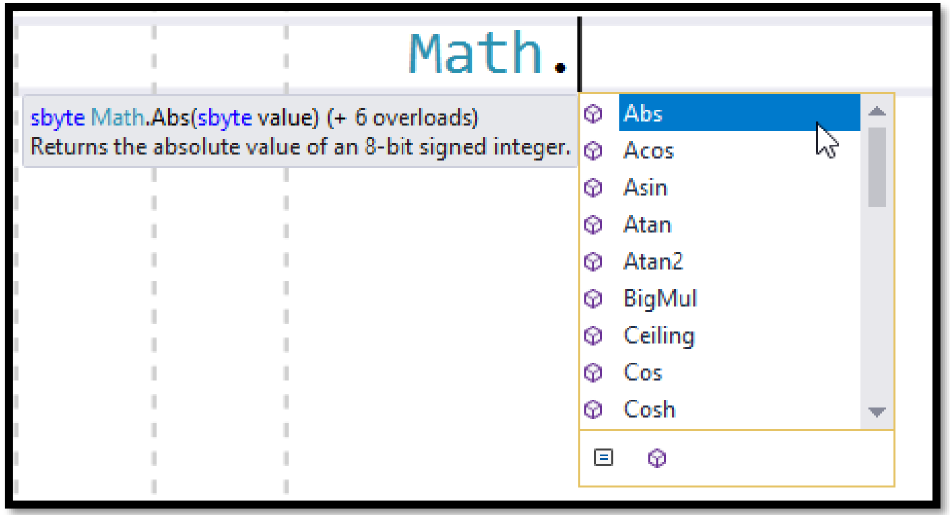
- C# provides functions that perform the most common mathematical operations.
- These functions are called methods.
- The math methods are invoked using a syntax that is similar to the **Write** statements
  - the expression in parentheses is called the **argument** of the method

```
double root = Math.Sqrt(17.0);
double angle = 1.5;
double height = Math.Sin(angle);
```

- C# assumes that the values you use with **sin** and the other trigonometric functions (cos, tan) are in **radians**.
  - To convert from degrees to radians, you can divide by 360 and multiply by 2π.

```
double degrees = 90;
double angle = degrees * 2 * Math.PI / 360.0;
```

- Use IntelliSense to find other **Math** methods (such as **Abs**, **Round**, …)



```
Math.
```

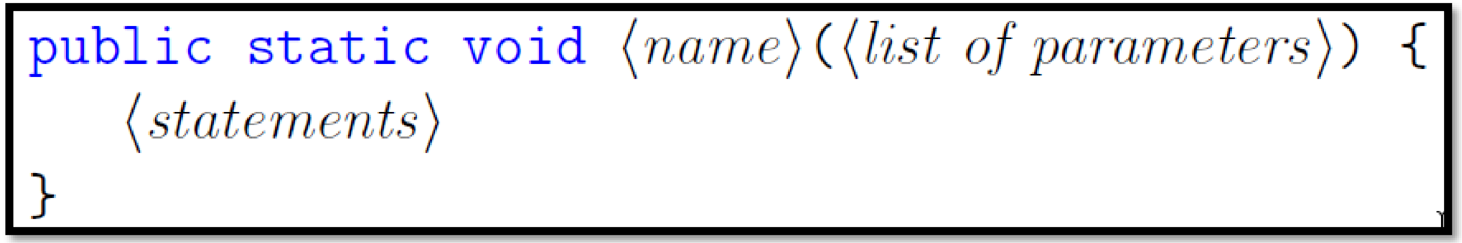
sbyte Math.Abs(sbyte value) (+ 6 overloads)
Returns the absolute value of an 8-bit signed integer.

- methods can be **composed**

```
double x = Math.Cos(angle + Math.PI/2);
```

3.5 ADDING NEW METHODS

- The method named **Main** is special, but the syntax is the same for other methods:
  - By convention, methods start with an upper case letter and use "Pascal case" such as: *JammingWordsTogetherLikeThis*

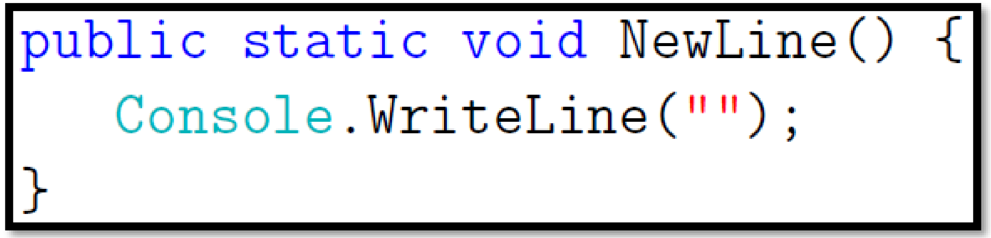


```
public static void ⟨name⟩(⟨list of parameters⟩) {
    ⟨statements⟩
}
```

- The list of **parameters** specifies what information, if any, you have to provide to use (or invoke) the new method.
  - The list of parameters can be empty (our first example has no parameters) or can have one or more parameters
  - The parameter for Main is ***string[] args***, which means that whoever invokes Main has to provide an array of strings (we'll get to arrays in Chapter 10).

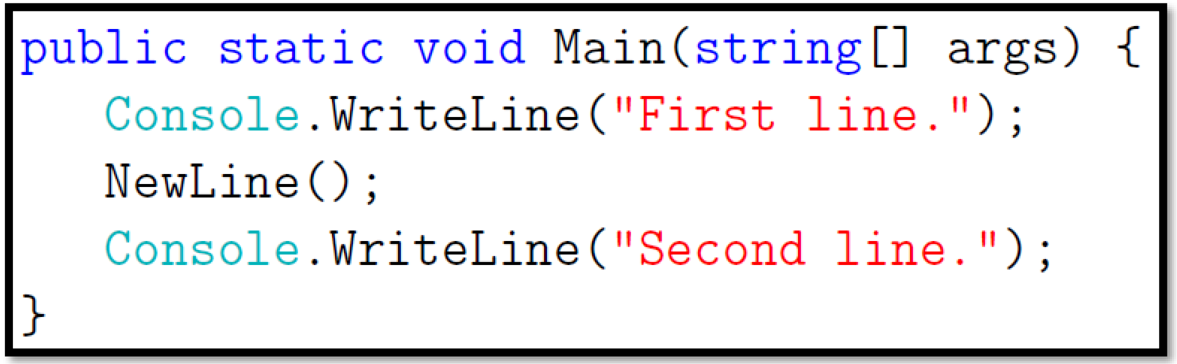
3.5 ADDING NEW METHODS (2)

- Example of a **Method definition**:
  - Note: this method has no parameters
  - The method is called NewLine



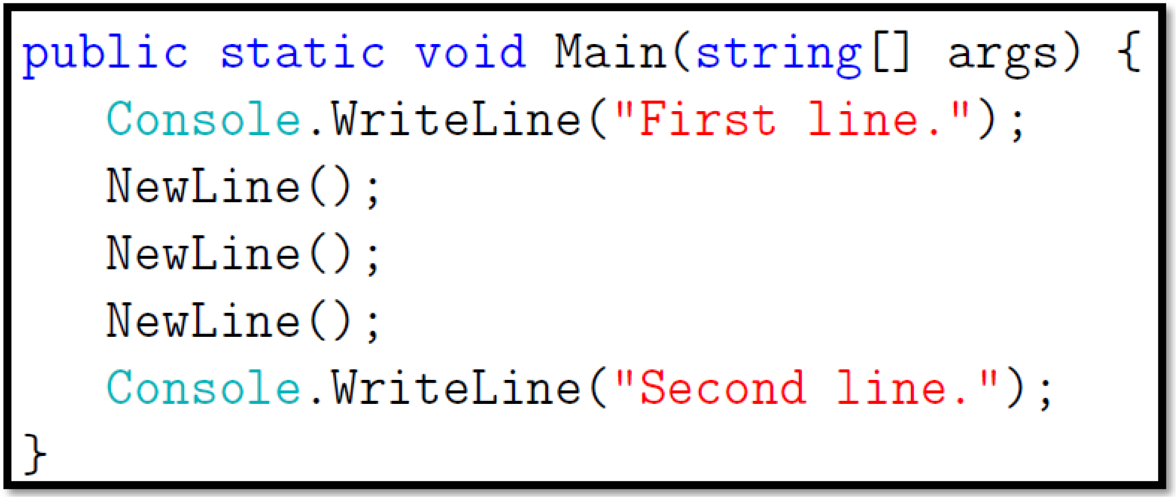
```
public static void NewLine() {
    Console.WriteLine("");
}
```

- Method **invocation** (**method call**)
  - we invoke this new method the same way we invoke other methods:
  - what is the output?



```
public static void Main(string[] args) {
    Console.WriteLine("First line.");
    NewLine();
    Console.WriteLine("Second line.");
}
```

  - we can invoke a method as many times as we want:
  - what is the output?
  - note that we can invoke a method inside another method



```
public static void Main(string[] args) {
    Console.WriteLine("First line.");
    NewLine();
    NewLine();
    NewLine();
    Console.WriteLine("Second line.");
}
```

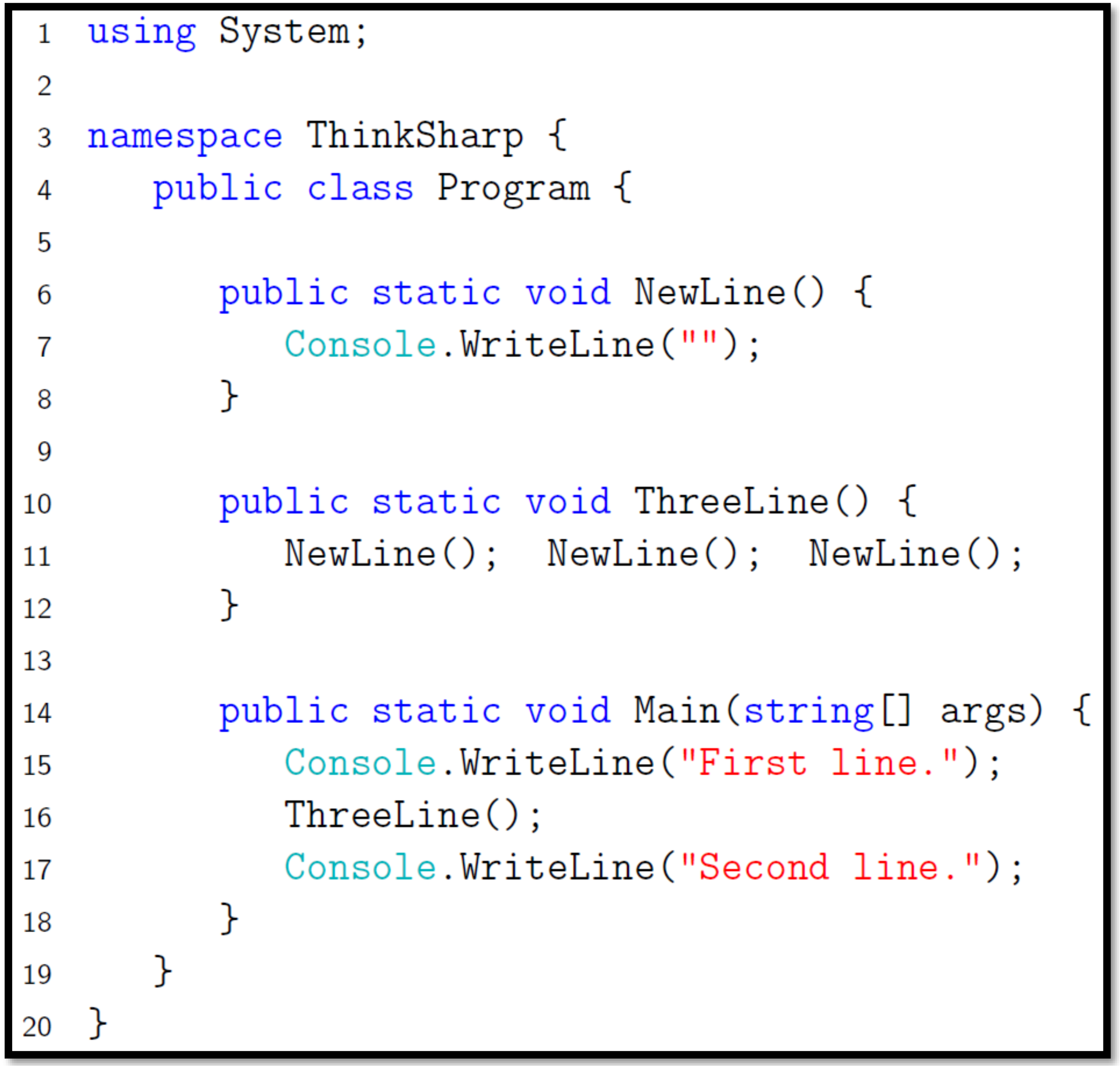
3.5 ADDING NEW METHODS (3)

- Pulling together the code fragments from the previous section, the class definition looks like this:

- Line 4 indicates that this is the class definition for a new class called Program.

- A **class** is a collection of related methods (and other members we'll see later)

- The other class we've seen is the **Math** class. It contains methods named **Sqrt**, **Sin** and others.



```
1  using System;
2
3  namespace ThinkSharp {
4     public class Program {
5
6        public static void NewLine() {
7           Console.WriteLine("");
8        }
9
10       public static void ThreeLine() {
11          NewLine();  NewLine();  NewLine();
12       }
13
14       public static void Main(string[] args) {
15          Console.WriteLine("First line.");
16          ThreeLine();
17          Console.WriteLine("Second line.");
18       }
19    }
20 }
```

3.5 ADDING NEW METHODS (3)

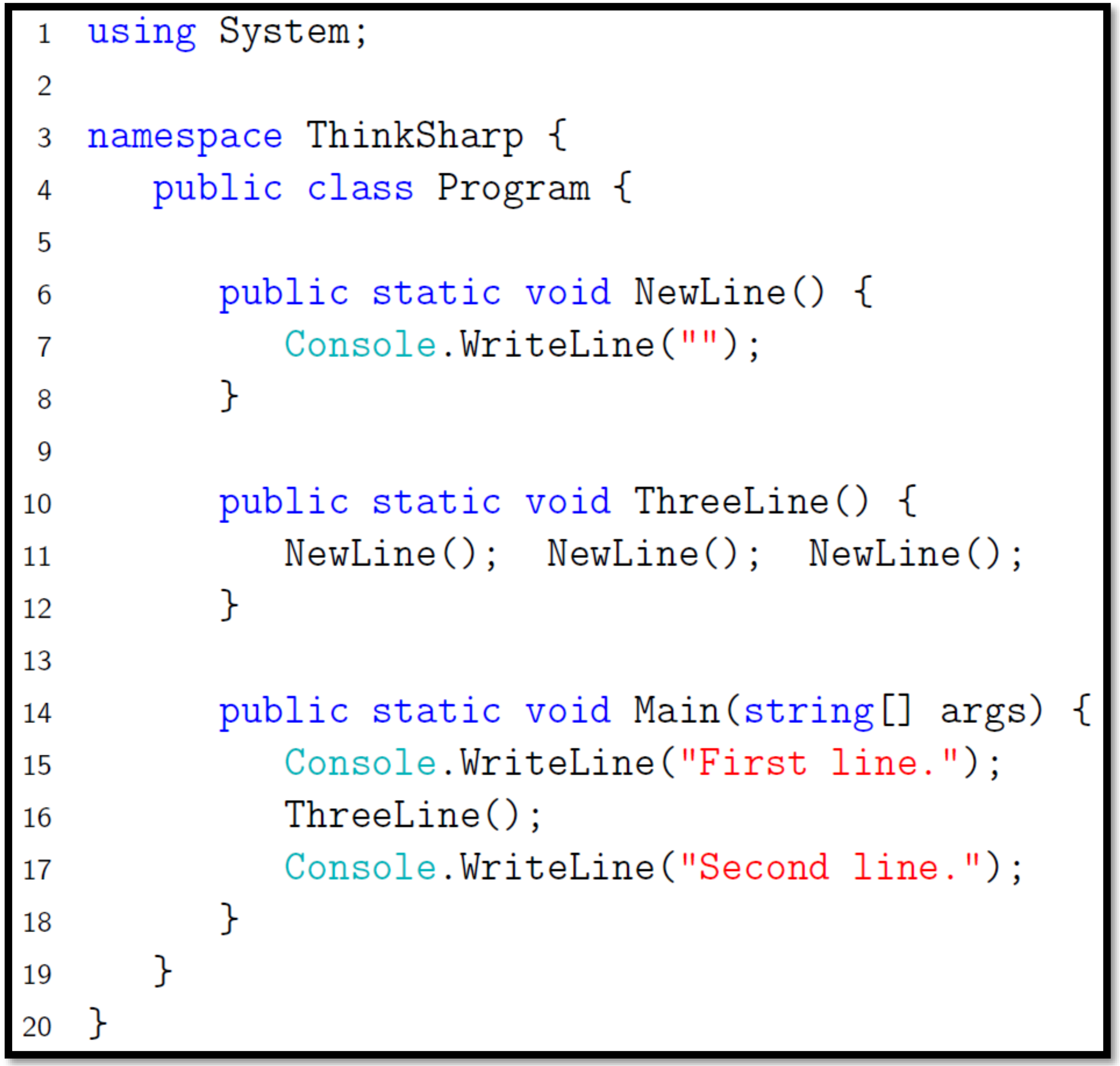
- Pulling together the code fragments from the previous section, the class definition looks like this:

- Line 4 indicates that this is the class definition for a new class called **Program**.

- A **class** is a collection of related methods (and other members we'll see later)

- The other class we've seen is the **Math** class. It contains methods named **Sqrt**, **Sin** and others.

- When you look at a class definition that contains several methods, it is **tempting to read it from top to bottom**, but that **is not the order of execution of the program**.
  - Execution always begins at the first statement of **Main**

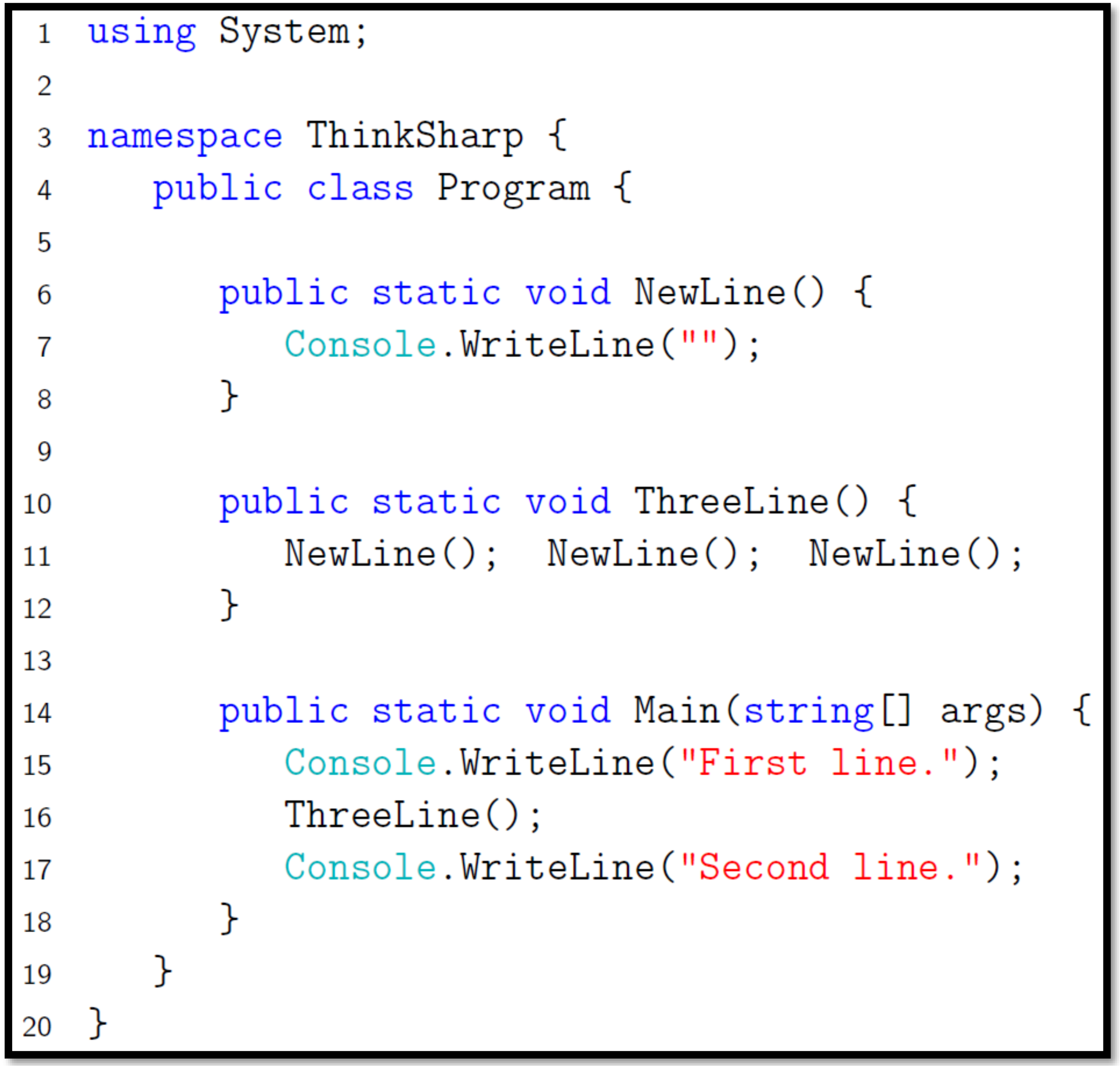


```
 1  using System;
 2
 3  namespace ThinkSharp {
 4      public class Program {
 5
 6          public static void NewLine() {
 7              Console.WriteLine("");
 8          }
 9
10          public static void ThreeLine() {
11              NewLine();  NewLine();  NewLine();
12          }
13
14          public static void Main(string[] args) {
15              Console.WriteLine("First line.");
16              ThreeLine();
17              Console.WriteLine("Second line.");
18          }
19      }
20  }
```

3.7 PROGRAMS WITH MULTIPLE METHODS

- Execution always begins at the first statement of **Main**, regardless of where it is in the program
  - Statements are executed one at a time, in order, until you reach a method invocation.

- Method invocations are like a detour in the flow of execution.
  - Instead of going to the next statement, you go to the first line of the invoked method, execute all the statements there, and then come back and pick up again where you left off.



```
using System;

namespace ThinkSharp {
    public class Program {

        public static void NewLine() {
            Console.WriteLine("");
        }

        public static void ThreeLine() {
            NewLine();  NewLine();  NewLine();
        }

        public static void Main(string[] args) {
            Console.WriteLine("First line.");
            ThreeLine();
            Console.WriteLine("Second line.");
        }
    }
}
```

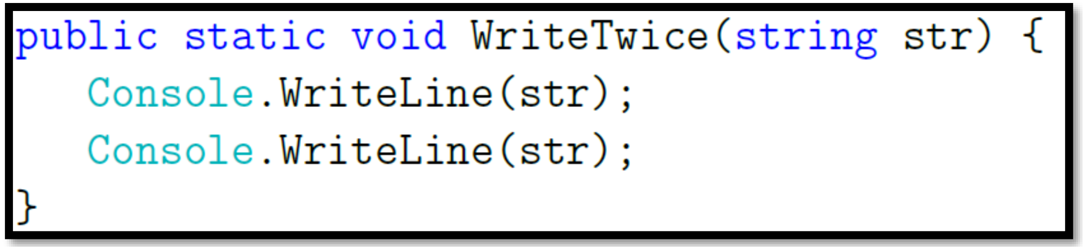
3.8 PARAMETERS AND ARGUMENTS

- Some of the methods we have used require arguments
(which are values that you provide when you invoke the method)
  - For example, to find the sine of a number, you have to provide the number.
  - So Sin takes a double as an argument.

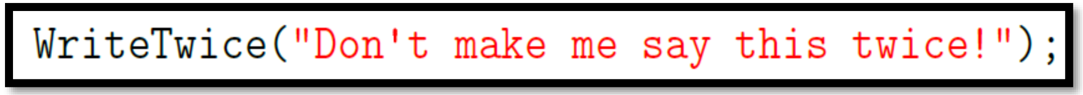
- Some methods take more than one argument;
  - for example, Pow takes two doubles, the base and the exponent.

- When you **use** a method, you provide arguments.
When you **write** a method, you specify a list of parameters.
  - A parameter is a variable that stores an argument. The parameter list indicates what arguments are required.

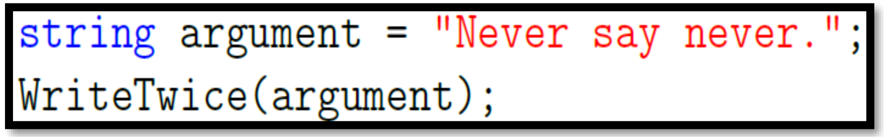
- Example:



```
public static void WriteTwice(string str) {
    Console.WriteLine(str);
    Console.WriteLine(str);
}
```



```
WriteTwice("Don't make me say this twice!");
```

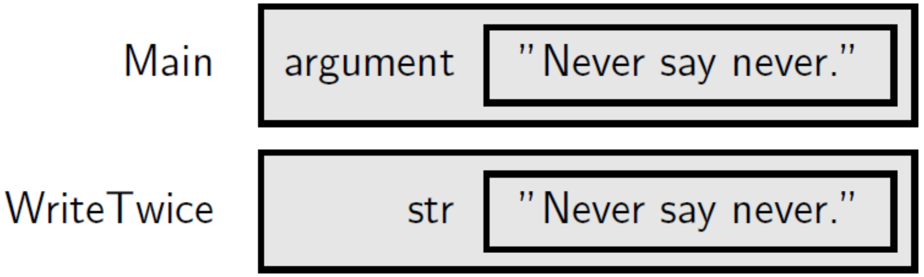


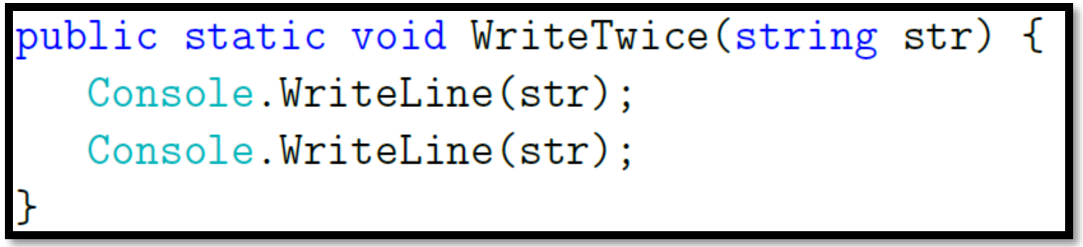
```
string argument = "Never say never.";
WriteTwice(argument);
```

3.9 STACK DIAGRAMS

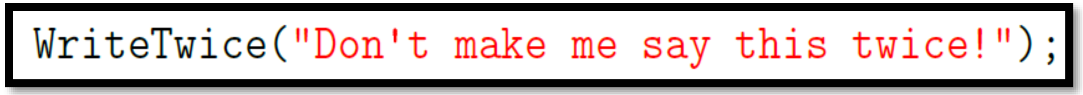
- Parameters and other variables only exist inside their own methods.
  - Within the confines of **Main**, there is no such thing as **str**.
  - Similarly, inside **WriteTwice** there is no such thing as **argument**.

- One way to keep track of where each variable is defined is with a stack diagram.
  - The stack diagram for the previous example looks like this:
  - For each method there is a gray box called a frame that contains the method's parameters and variables.

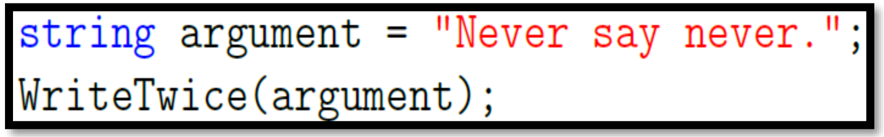




```
public static void WriteTwice(string str) {
    Console.WriteLine(str);
    Console.WriteLine(str);
}
```



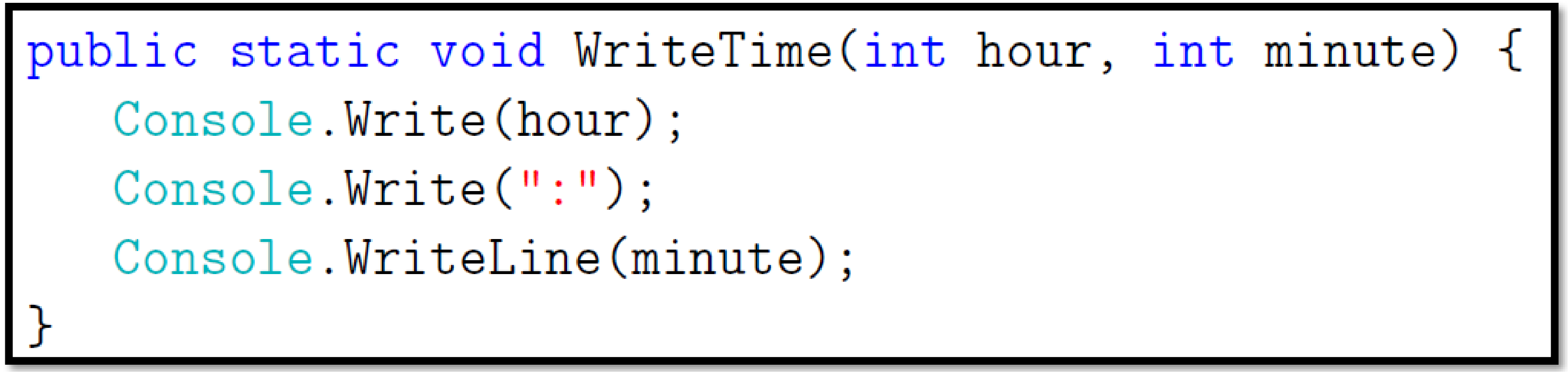
```
WriteTwice("Don't make me say this twice!");
```



```
string argument = "Never say never.";
WriteTwice(argument);
```

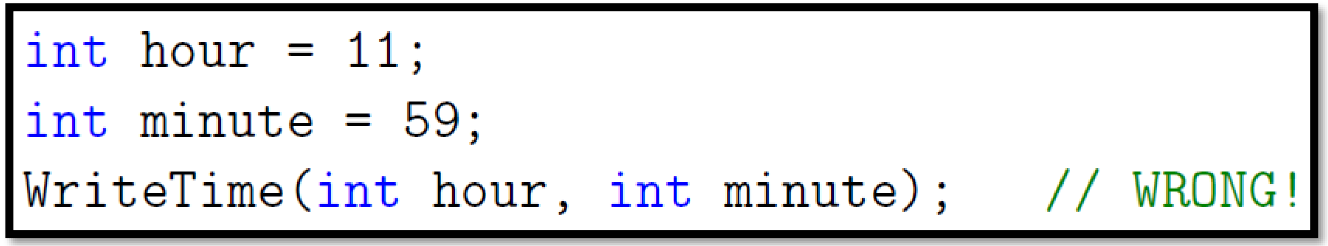
3.10 METHODS WITH MULTIPLE PARAMETERS

- you have to declare the type of every **parameter**.



```
public static void WriteTime(int hour, int minute) {
    Console.Write(hour);
    Console.Write(":");
    Console.WriteLine(minute);
}
```

- you do **not** have to declare the types of **arguments**.
  - the system can tell the type of hour and minute by looking at their declarations.



```
int hour = 11;
int minute = 59;
WriteTime(int hour, int minute);    // WRONG!
```

HOMEWORK FOR CHAPTER 3

- Requirements: see moodle for details
- Deadline: see moodle

- **Reminder: If your code does not compile, crashes at start, or contains no meaningful comments, it will automatically be graded with 0!**