CSc 346
Object Oriented Programming

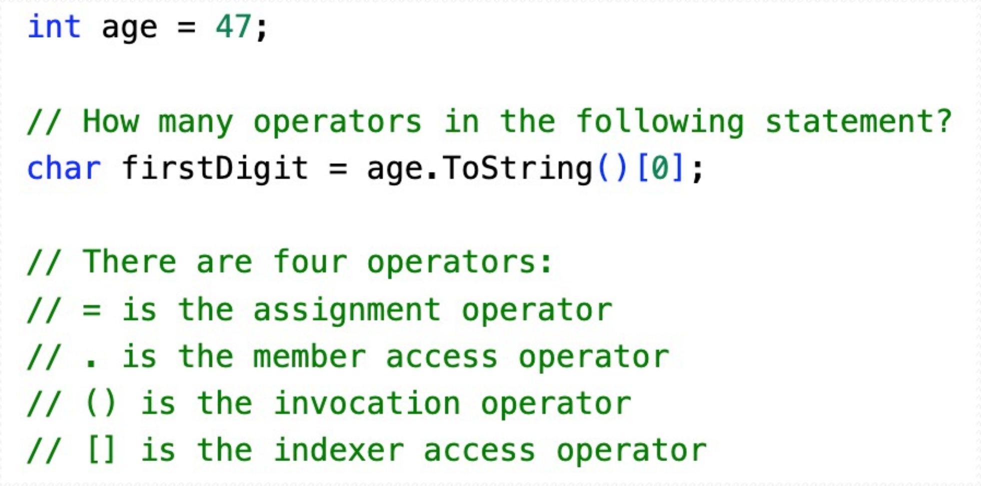
Ken Gamradt

Spring 2024

Chapter 3

Miscellaneous operators

- **nameof** operator returns the short name (without the namespace) of a variable, type, or member as a string value
  - Useful when outputting exception messages
- **sizeof** operator returns the size in bytes of simple types
  - Useful for determining the efficiency of data storage



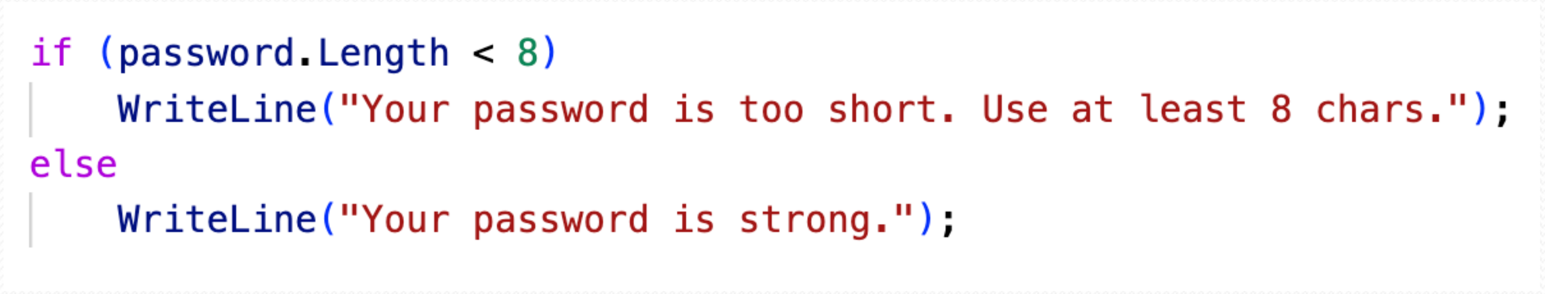
```
int age = 47;

// How many operators in the following statement?
char firstDigit = age.ToString()[0];

// There are four operators:
// = is the assignment operator
// . is the member access operator
// () is the invocation operator
// [] is the indexer access operator
```

Why you should always use braces with if statements

- As there is only a single statement inside each block, the preceding code could be written without the curly braces



```
if (password.Length < 8)
    WriteLine("Your password is too short. Use at least 8 chars.");
else
    WriteLine("Your password is strong.");
```

- This style of if statement should be avoided because it can introduce serious bugs

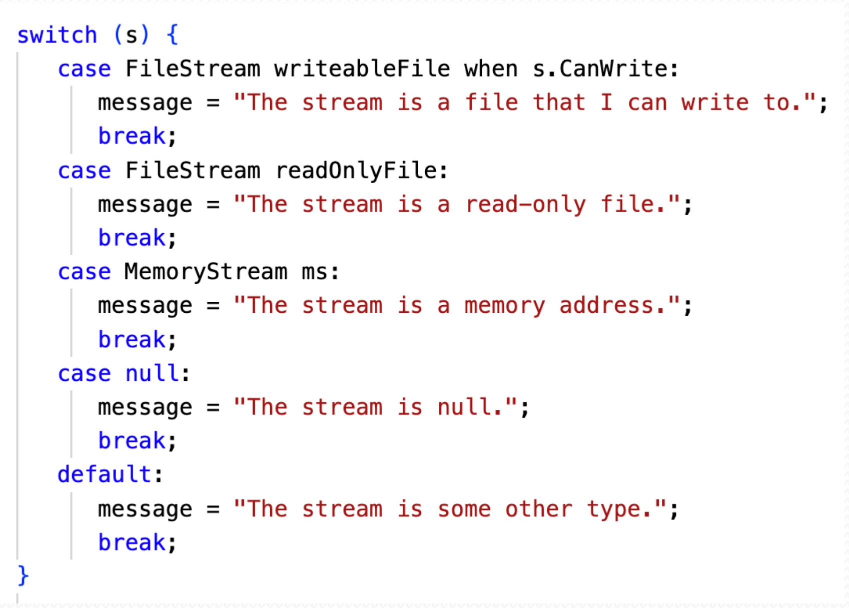
Why you should always use braces with if statements

- An infamous example is the **#gotofail bug** in Apple's iPhone iOS operating system
- For 18 months after Apple's iOS 6 was released it had a bug due to an if statement without braces in its Secure Sockets Layer (SSL) encryption code
- Any user running Safari who tried to connect to secure websites was not properly secure because an important check was being accidentally skipped
- Just because you can leave out the curly braces, doesn't mean you should
- Your code is not "more efficient" without them, it is
  - **harder to read**
  - **less maintainable**
  - **potentially more dangerous**

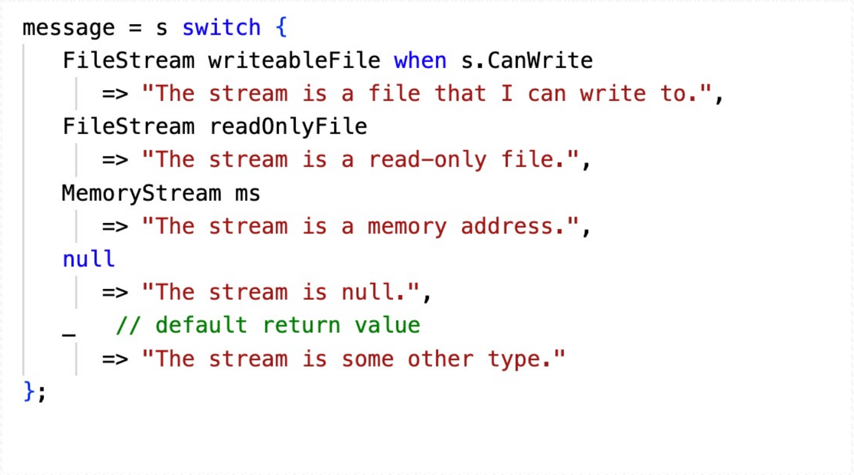
Simplifying switch statements with switch expressions

- In C# 8.0 or later, you can simplify switch statements using **switch expressions**
- Most switch statements are very simple, yet they require a lot of typing
- switch expressions are designed to simplify the code you need to type while still expressing the same intent in scenarios where all cases return a value to set a single variable
- switch expressions use a lambda, =>, to indicate a return value

Simplifying switch statements with switch expressions



```
switch (s) {
    case FileStream writeableFile when s.CanWrite:
        message = "The stream is a file that I can write to.";
        break;
    case FileStream readOnlyFile:
        message = "The stream is a read-only file.";
        break;
    case MemoryStream ms:
        message = "The stream is a memory address.";
        break;
    case null:
        message = "The stream is null.";
        break;
    default:
        message = "The stream is some other type.";
        break;
}
```



```
message = s switch {
    FileStream writeableFile when s.CanWrite
        => "The stream is a file that I can write to.",
    FileStream readOnlyFile
        => "The stream is a read-only file.",
    MemoryStream ms
        => "The stream is a memory address.",
    null
        => "The stream is null.",
    _   // default return value
        => "The stream is some other type."
};
```

Looping with the foreach statement

- The **foreach** statement is a bit different from other iteration statements
- It is used to perform a block of statements on each item in a sequence
  - E.g., an array or collection
- Each item is usually read-only
- If the sequence structure is modified during iteration, for example, adding or removing an item, then an exception will be thrown

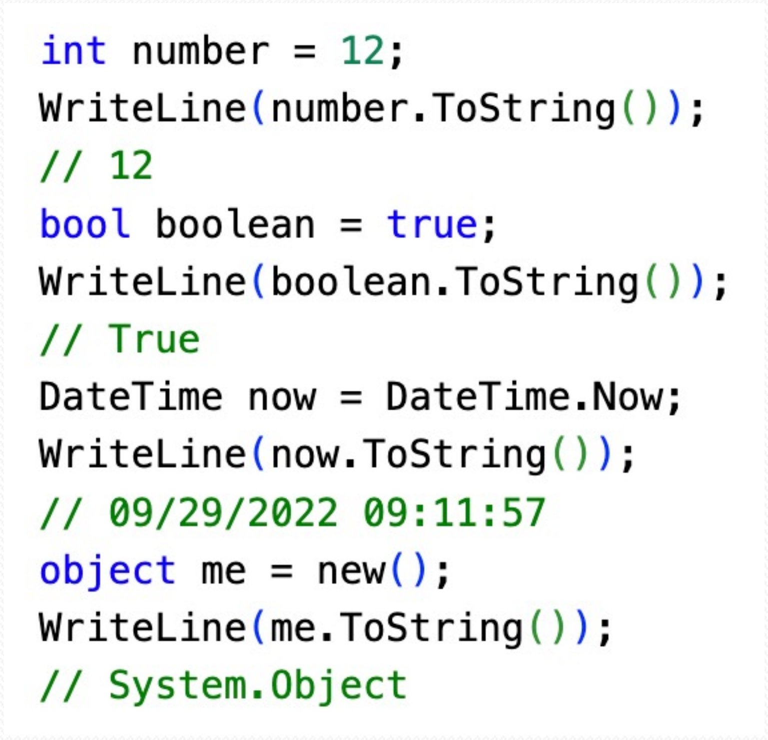
```
string[] names = { "Adam", "Barry", "Charlie" };

// name is equivalent to name[i] as i counts from 0..size-1
// no access to i is available with this statement
foreach (string name in names) {
  WriteLine($"{name} has {name.Length} characters.");
}
```

```
Adam has 4 characters.
Barry has 5 characters.
Charlie has 7 characters.
```

Converting from any type to a string

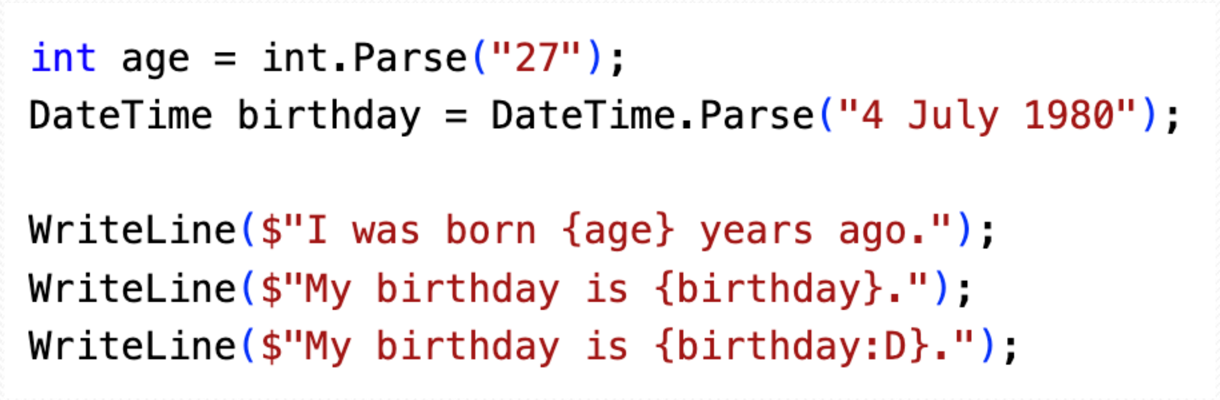
- The most common conversion is from any type into a string variable for outputting as human-readable text, so all types have a method named **ToString** that they inherit from the System.Object class
- The ToString method converts the current value of any variable into a textual representation
- Some types can't be sensibly represented as text, so they return their namespace and type name instead



```
int number = 12;
WriteLine(number.ToString());
// 12
bool boolean = true;
WriteLine(boolean.ToString());
// True
DateTime now = DateTime.Now;
WriteLine(now.ToString());
// 09/29/2022 09:11:57
object me = new();
WriteLine(me.ToString());
// System.Object
```

Parsing from strings to numbers or dates and times

- The second most common conversion is from strings to numbers or date and time values
- The opposite of ToString is Parse
- Only a few types have a Parse method
  - Including all the number types and DateTime

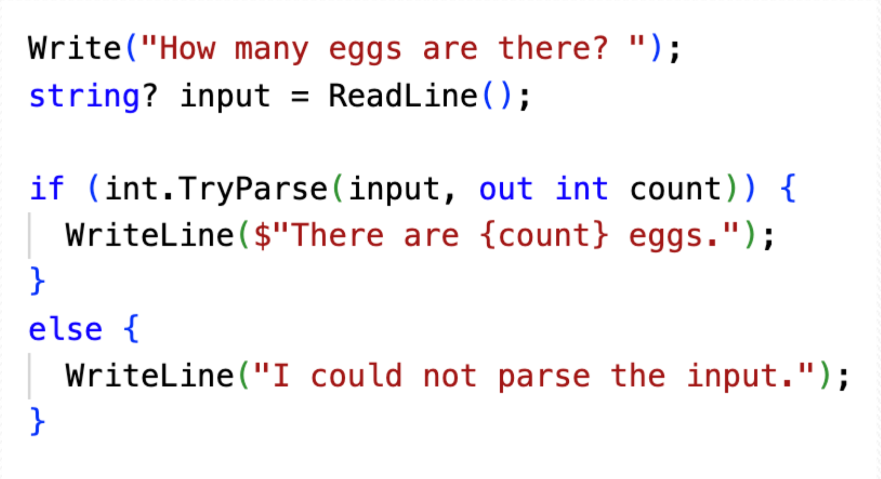


```
int age = int.Parse("27");
DateTime birthday = DateTime.Parse("4 July 1980");

WriteLine($"I was born {age} years ago.");
WriteLine($"My birthday is {birthday}.");
WriteLine($"My birthday is {birthday:D}.");
```

Avoiding exceptions using the TryParse method

- To avoid errors, you can use the **TryParse** method instead
- TryParse attempts to convert the input string
  - returns **true** if it can convert it
  - returns **false** if it cannot
- The out keyword is required to allow the TryParse method to set the count variable when the conversion works



```
Write("How many eggs are there? ");
string? input = ReadLine();

if (int.TryParse(input, out int count)) {
  WriteLine($"There are {count} eggs.");
}
else {
  WriteLine("I could not parse the input.");
}
```

Handling exceptions

- Some languages return error codes when something goes wrong
- .NET uses exceptions that are richer and designed only for failure reporting compared to return values that have multiple uses
- When this happens, we say a **runtime exception has been thrown**
- When an exception is thrown, the thread is suspended and if the calling code has defined a **try-catch** statement, then it is given a chance to handle the exception
- If the current method does not handle it, then its calling method is given a chance, and so on up the call stack

Handling exceptions

- The default behavior of a console application or a .NET is to output a message about the exception, including a stack trace, and then stop running the code
- The application is terminated
- This is better than allowing the code to continue executing in a potentially corrupt state
- Your code should only catch and handle exceptions that it understands and can properly fix

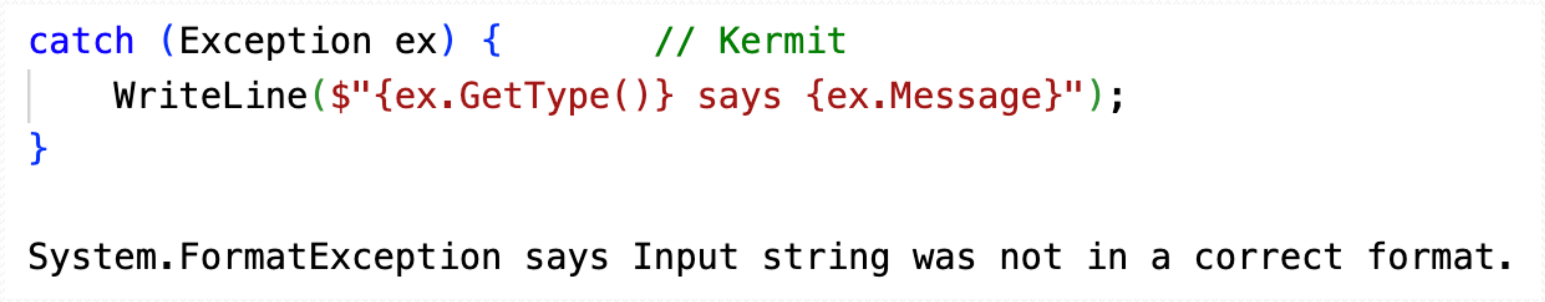
Wrapping error-prone code in a try block

- When you know that a statement may cause an error, you should wrap that statement in a try block
- Any statements in the **catch block** will be executed only if an exception is thrown by a statement in the **try block**
- When the code was executed, the error exception was caught, and console application continued running
- This is better than the default behavior
- It might also be useful to see the type of error that occurred

```
WriteLine("Before parsing");  // alwyas
Write("What is your age? ");  // always
string? input = ReadLine();   // nullable
try {
    int age = int.Parse(input);
    // valid parse only
    WriteLine($"You are {age} years old.");
}
catch {
    // invalid parse only
}
WriteLine("After parsing");  // always
```

Catching all exceptions

- To get information about any type of exception that might occur, you can declare a variable of type System.Exception to the catch block

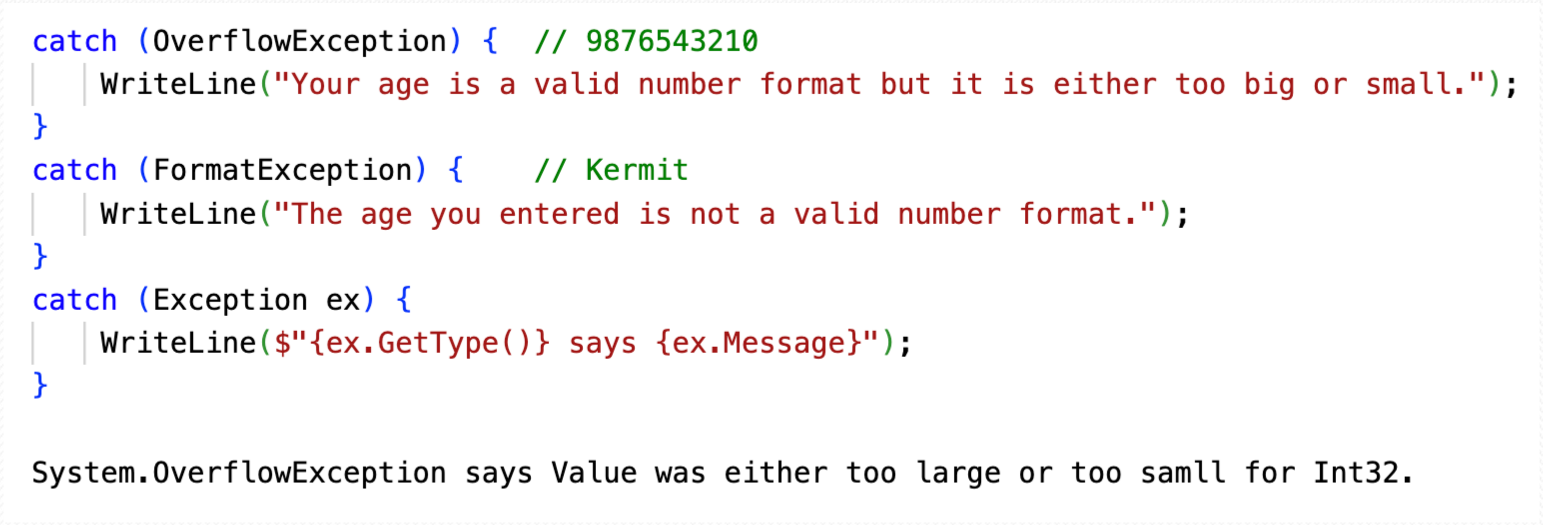


```
catch (Exception ex) {        // Kermit
    WriteLine($"{ex.GetType()} says {ex.Message}");
}

System.FormatException says Input string was not in a correct format.
```

Catching specific exceptions

- Now that we know which specific type of exception occurred, we can improve our code by catching just that type of exception and customizing the message that we display to the user

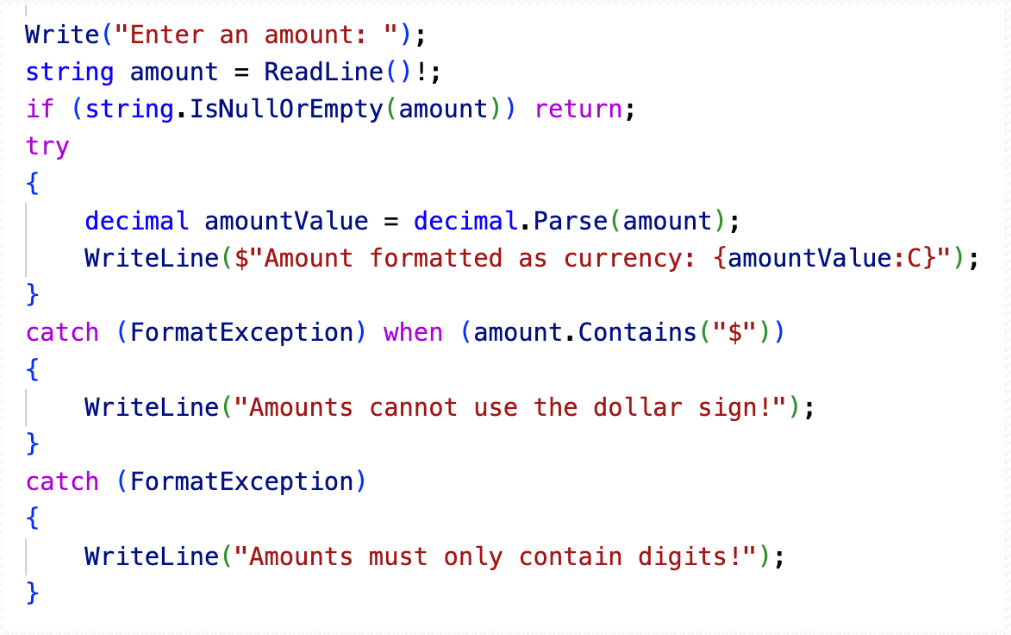


```
catch (OverflowException) {  // 9876543210
    WriteLine("Your age is a valid number format but it is either too big or small.");
}
catch (FormatException) {    // Kermit
    WriteLine("The age you entered is not a valid number format.");
}
catch (Exception ex) {
    WriteLine($"{ex.GetType()} says {ex.Message}");
}

System.OverflowException says Value was either too large or too samll for Int32.
```

Catching with filters

- You can also add filters to a catch statement using the when keyword



```
Write("Enter an amount: ");
string amount = ReadLine()!;
if (string.IsNullOrEmpty(amount)) return;
try
{
    decimal amountValue = decimal.Parse(amount);
    WriteLine($"Amount formatted as currency: {amountValue:C}");
}
catch (FormatException) when (amount.Contains("$"))
{
    WriteLine("Amounts cannot use the dollar sign!");
}
catch (FormatException)
{
    WriteLine("Amounts must only contain digits!");
}
```