Introduction to C#

Mark Sapossnek

CS 594
Computer Science Department
Metropolitan College
Boston University

Prerequisites

- This module assumes that you understand the fundamentals of
  - Programming
    - Variables, statements, functions, loops, etc.
  - Object-oriented programming
    - Classes, inheritance, polymorphism, members, etc.
    - C++ or Java

Learning Objectives

- C# design goals
- Fundamentals of the C# language
  - Types, program structure, statements, operators
- Be able to begin writing and debugging C# programs
  - Using the .NET Framework SDK
  - Using Visual Studio.NET
- Be able to write individual C# methods

Agenda

- **Hello World**
- Design Goals of C#
- Types
- Program Structure
- Statements
- Operators
- Using Visual Studio.NET
- Using the .NET Framework SDK

Hello World

```
using System;

class Hello {
   static void Main( ) {
      Console.WriteLine("Hello world");
      Console.ReadLine();  // Hit enter to finish
   }
}
```

Agenda

- Hello World
- **Design Goals of C#**
- Types
- Program Structure
- Statements
- Operators
- Using Visual Studio.NET
- Using the .NET Framework SDK

Design Goals of C#
The Big Ideas

- Component-orientation
- Everything is an object
- Robust and durable software
- Preserving your investment

Design Goals of C#
Component-Orientation

- C# is the first “Component-Oriented” language in the C/C++ family
- What is a component?
  - An independent module of reuse and deployment
  - Coarser-grained than objects (objects are language-level constructs)
  - Includes multiple classes
  - Often language-independent
  - In general, component writer and user don’t know each other, don’t work for the same company, and don’t use the same language

Design Goals of C#
Component-Orientation

- Component concepts are first class
  - Properties, methods, events
  - Design-time and run-time attributes
  - Integrated documentation using XML
- Enables “one-stop programming”
  - No header files, IDL, etc.
  - Can be embedded in ASP pages

Design Goals of C#
Everything is an Object

- Traditional views
  - C++, Java™: Primitive types are “magic” and do not interoperate with objects
  - Smalltalk, Lisp: Primitive types are objects, but at some performance cost
- C# unifies with no performance cost
  - Deep simplicity throughout system
- Improved extensibility and reusability
  - New primitive types: Decimal, SQL…
  - Collections, etc., work for all types

Design Goals of C#
Robust and Durable Software

- Garbage collection
  - No memory leaks and stray pointers
- Exceptions
- Type-safety
  - No uninitialized variables, no unsafe casts
- Versioning
- Avoid common errors
  - E.g. if (x = y) ...
- One-stop programming
  - Fewer moving parts

Design Goals of C#
Preserving Your Investment

- C++ Heritage
  - Namespaces, pointers (in unsafe code), unsigned types, etc.
  - Some changes, but no unnecessary sacrifices
- Interoperability
  - What software is increasingly about
  - C# talks to XML, SOAP, COM, DLLs, and any .NET Framework language
- Increased productivity
  - Short learning curve
  - Millions of lines of C# code in .NET

Agenda

- Hello World
- Design Goals of C#
- **Types**
- Program Structure
- Statements
- Operators
- Using Visual Studio.NET
- Using the .NET Framework SDK

Types
Overview

- A C# program is a collection of types
  - Classes, structs, enums, interfaces, delegates
- C# provides a set of predefined types
  - E.g. `int`, `byte`, `char`, `string`, `object`, …
- You can create your own types
- All data and code is defined within a type
  - No global variables, no global functions

Types
Overview

- Types contain:
  - Data members
    - Fields, constants, arrays
    - Events
  - Function members
    - Methods, operators, constructors, destructors
    - Properties, indexers
  - Other types
    - Classes, structs, enums, interfaces, delegates

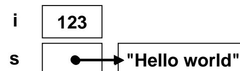
Types
Overview

- Types can be instantiated…
  - …and then used: call methods, get and set properties, etc.
- Can convert from one type to another
  - Implicitly and explicitly
- Types are organized
  - Namespaces, files, assemblies
- There are two categories of types: value and reference
- Types are arranged in a hierarchy

Types
Unified Type System

- Value types
  - Directly contain data
  - Cannot be null
- Reference types
  - Contain references to objects
  - May be null

```
int i = 123;
string s = "Hello world";
```



Types
Unified Type System

- Value types
  - Primitives `int i; float x;`
  - Enums `enum State { Off, On }`
  - Structs `struct Point {int x,y;}`
- Reference types
  - Root `object`
  - String `string`
  - Classes `class Foo: Bar, IFoo {...}`
  - Interfaces `interface IFoo: IBar {...}`
  - Arrays `string[] a = new string[10];`
  - Delegates `delegate void Empty();`

Types
Unified Type System

	Value (Struct)	Reference (Class)
Variable holds	Actual value	Memory location
Allocated on	Stack, member	Heap
Nullability	Always has value	May be null
Default value	0	null
Aliasing (in a scope)	No	Yes
Assignment means	Copy data	Copy reference

Types
Unified Type System

- Benefits of value types
  - No heap allocation, less GC pressure
  - More efficient use of memory
  - Less reference indirection
  - Unified type system
    - No primitive/object dichotomy

Types
Conversions

- Implicit conversions
  - Occur automatically
  - Guaranteed to succeed
  - No information (precision) loss
- Explicit conversions
  - Require a cast
  - May not succeed
  - Information (precision) might be lost
- Both implicit and explicit conversions can be user-defined

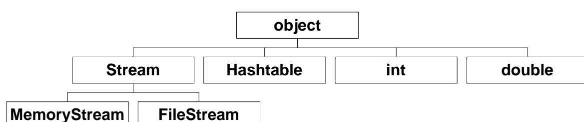
Types
Conversions

```
int x = 123456;
long y = x;                  // implicit
short z = (short)x;          // explicit

double d = 1.2345678901234;
float f = (float)d;          // explicit
long l = (long)d;            // explicit
```

Types
Unified Type System

- Everything is an object
  - All types ultimately inherit from object
  - Any piece of data can be stored, transported, and manipulated with no extra work



Types
Unified Type System

- Polymorphism
  - The ability to perform an operation on an object without knowing the precise type of the object

```
void Poly(object o) {
  Console.WriteLine(o.ToString());
}
```

```
Poly(42);
Poly("abcd");
Poly(12.345678901234m);
Poly(new Point(23,45));
```

Types
Unified Type System

- Question: How can we treat value and reference types polymorphically?
  - How does an int (value type) get converted into an object (reference type)?
- Answer: Boxing!
  - Only value types get boxed
  - Reference types do not get boxed

Types
Unified Type System

- Boxing
  - Copies a value type into a reference type (`object`)
  - Each value type has corresponding “hidden” reference type
  - Note that a reference-type copy is made of the value type
    - Value types are never aliased
  - Value type is converted implicitly to `object`, a reference type
    - Essentially an “up cast”

Types
Unified Type System

- Unboxing
  - Inverse operation of boxing
  - Copies the value out of the box
    - Copies from reference type to value type
  - Requires an explicit conversion
    - May not succeed (like all explicit conversions)
    - Essentially a “down cast”

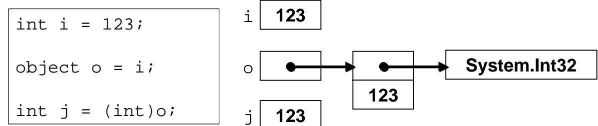
Types
Unified Type System

- Boxing and unboxing

```
int i = 123;

object o = i;

int j = (int)o;
```



Types
Unified Type System

- Benefits of boxing
  - Enables polymorphism across all types
  - Collection classes work with all types
  - Eliminates need for wrapper classes
  - Replaces OLE Automation's Variant
- Lots of examples in .NET Framework

```
Hashtable t = new Hashtable();
t.Add(0, "zero");
t.Add(1, "one");
t.Add(2, "two");
```

```
string s = string.Format(
  "Your total was {0} on {1}",
  total, date);
```

Types
Unified Type System

- Disadvantages of boxing
  - Performance cost
- The need for boxing will decrease when the CLR supports generics (similar to C++ templates)

Types
Predefined Types

- Value
  - Integral types
  - Floating point types
  - `decimal`
  - `bool`
  - `char`
- Reference
  - `object`
  - `string`

Predefined Types
Value Types

- All are predefined structs

Signed	sbyte, short, int, long
Unsigned	byte, ushort, uint, ulong
Character	char
Floating point	float, double, decimal
Logical	bool

Predefined Types
Integral Types

C# Type	System Type	Size (bytes)	Signed?
sbyte	System.Sbyte	1	Yes
short	System.Int16	2	Yes
int	System.Int32	4	Yes
long	System.Int64	8	Yes
byte	System.Byte	1	No
ushort	System.UInt16	2	No
uint	System.UInt32	4	No
ulong	System.UInt64	8	No

Predefined Types
Floating Point Types

- Follows IEEE 754 specification
- Supports ± 0, ± Infinity, NaN

C# Type	System Type	Size (bytes)
float	System.Single	4
double	System.Double	8

Predefined Types
`decimal`

- 128 bits
- Essentially a 96 bit value scaled by a power of 10
- Decimal values represented precisely
- Doesn’t support signed zeros, infinities or NaN

C# Type	System Type	Size (bytes)
decimal	System.Decimal	16

Predefined Types
`decimal`

- All integer types can be implicitly converted to a decimal type
- Conversions between decimal and floating types require explicit conversion due to possible loss of precision
- s * m * 10e
  - s = 1 or –1
  - $0 \leq m \leq 296$
  - $-28 \leq e \leq 0$

Predefined Types
Integral Literals

- Integer literals can be expressed as decimal or hexadecimal
- U or u: uint or ulong
- L or l: long or ulong
- UL or ul: ulong

```
123           // Decimal
0x7B          // Hexadecimal
123U          // Unsigned
123ul         // Unsigned long
123L          // Long
```

Predefined Types
Real Literals

- F or f: float
- D or d: double
- M or m: decimal

```
123f              // Float
123D              // Double
123.456m          // Decimal
1.23e2f           // Float
12.3E1M           // Decimal
```

Predefined Types
`bool`

- Represents logical values
- Literal values are true and false
- Cannot use 1 and 0 as boolean values
  - No standard conversion between other types and bool

C# Type	System Type	Size (bytes)
`bool`	`System.Boolean`	1 (2 for arrays)

Predefined Types
`char`

- Represents a Unicode character
- Literals
  - 'A'   // Simple character
  - '\u0041'   // Unicode
  - '\x0041'   // Unsigned short hexadecimal
  - '\n'   // Escape sequence character

C# Type	System Type	Size (bytes)
`Char`	`System.Char`	2

Predefined Types
`char`

- Escape sequence characters (partial list)

Char	Meaning	Value
`\'`	Single quote	0x0027
`\"`	Double quote	0x0022
`\\`	Backslash	0x005C
`\0`	Null	0x0000
`\n`	New line	0x000A
`\r`	Carriage return	0x000D
`\t`	Tab	0x0009

Predefined Types
Reference Types

Root type	`object`
Character string	`string`

Predefined Types
object

- Root of object hierarchy
- Storage (book keeping) overhead
  - 0 bytes for value types
  - 8 bytes for reference types
- An actual reference (not the object) uses 4 bytes

C# Type	System Type	Size (bytes)
`object`	`System.Object`	0/8 overhead

Predefined Types
object Public Methods

- `public bool Equals(object)`
- `protected void Finalize()`
- `public int GetHashCode()`
- `public System.Type GetType()`
- `protected object MemberwiseClone()`
- `public void Object()`
- `public string ToString()`

Predefined Types
string

- An immutable sequence of Unicode characters
- Reference type
- Special syntax for literals
  - `string s = "I am a string";`

C# Type	System Type	Size (bytes)
`String`	`System.String`	20 minimum

Predefined Types
string

- Normally have to use escape characters

```
string s1= "\\\\server\\fileshare\\filename.cs";
```

- Verbatim string literals
  - Most escape sequences ignored
    - Except for “”
  - Verbatim literals can be multi-line

```
string s2 = @"\\server\fileshare\filename.cs";
```

Types
User-defined Types

- User-defined types

Enumerations	`enum`
Arrays	`int[], string[]`
Interface	`interface`
Reference type	`class`
Value type	`struct`
Function pointer	`delegate`

Types
Enums

- An enum defines a type name for a related group of symbolic constants
- Choices must be known at compile-time
- Strongly typed
  - No implicit conversions to/from int
  - Can be explicitly converted
  - Operators: +, -, ++, --, &, |, ^, ~, …
- Can specify underlying type
  - byte, sbyte, short, ushort, int, uint, long, ulong

Types
Enums

```
enum Color: byte {
   Red   = 1,
   Green = 2,
   Blue  = 4,
   Black = 0,
   White = Red | Green | Blue
}

Color c = Color.Black;
Console.WriteLine(c);                // Black
Console.WriteLine(c.ToString());     // Black
Console.WriteLine(Enum.Format(typeof(Color),c,"F"));
```

Types
Enums

- All enums derive from `System.Enum`
  - Provides methods to
    - determine underlying type
    - test if a value is supported
    - initialize from string constant
    - retrieve all values in enum
    - …

Types
Arrays

- Arrays allow a group of elements of a specific type to be stored in a contiguous block of memory
- Arrays are reference types
- Derived from `System.Array`
- Zero-based
- Can be multidimensional
  - Arrays know their length(s) and rank
- Bounds checking

Types
Arrays

- Declare

```
int[] primes;
```

- Allocate

```
int[] primes = new int[9];
```

- Initialize

```
int[] prime = new int[] {1,2,3,5,7,11,13,17,19};
int[] prime = {1,2,3,5,7,11,13,17,19};
```

- Access and assign

```
prime2[i] = prime[i];
```

- Enumerate

```
foreach (int i in prime) Console.WriteLine(i);
```

Types
Arrays

- Multidimensional arrays
  - Rectangular
    - `int[,] matR = new int[2,3];`
    - Can initialize declaratively
    - `int[,] matR = new int[2,3] { {1,2,3}, {4,5,6} };`
  - Jagged
    - An array of arrays
    - `int[][] matJ = new int[2][];`
    - Must initialize procedurally

Types
Interfaces

- An interface defines a contract
  - Includes methods, properties, indexers, events
  - Any class or struct implementing an interface must support all parts of the contract
- Interfaces provide polymorphism
  - Many classes and structs may implement a particular interface
- Contain no implementation
  - Must be implemented by a class or struct

Types
Classes

- User-defined reference type
  - Similar to C++, Java classes
- Single class inheritance
- Multiple interface inheritance

Types
Classes

- Members
  - Constants, fields, methods, operators, constructors, destructors
  - Properties, indexers, events
  - Static and instance members
- Member access
  - `public`, `protected`, `private`, `internal`, `protected internal`
    - Default is `private`
- Instantiated with `new` operator

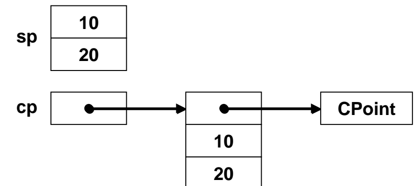
Types
Structs

- Similar to classes, but
  - User-defined value type
  - Always inherits from object
- Ideal for lightweight objects
  - `int`, `float`, `double`, etc., are all structs
  - User-defined "primitive" types
    - Complex, point, rectangle, color, rational
- Multiple interface inheritance
- Same members as class
- Member access
  - `public`, `internal`, `private`
- Instantiated with `new` operator

Types
Classes and Structs

```
struct SPoint { int x, y; ... }
 class CPoint { int x, y; ... }

SPoint sp = new SPoint(10, 20);
CPoint cp = new CPoint(10, 20);
```



Types
Delegates

- A delegate is a reference type that defines a method signature
- When instantiated, a delegate holds one or more methods
  - Essentially an object-oriented function pointer
- Foundation for framework events

Agenda

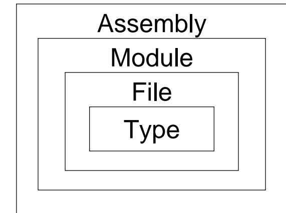
- Hello World
- Design Goals of C#
- Types
- **Program Structure**
- Statements
- Operators
- Using Visual Studio.NET
- Using the .NET Framework SDK

Program Structure
Overview

- Organizing Types
- Namespaces
- References
- Main Method
- Syntax

Program Structure
Organizing Types

- Physical organization
  - Types are defined in files
  - Files are compiled into modules
  - Modules are grouped into assemblies



Program Structure
Organizing Types

- Types are defined in files
  - A file can contain multiple types
  - Each type is defined in a single file
- Files are compiled into modules
  - Module is a DLL or EXE
  - A module can contain multiple files
- Modules are grouped into assemblies
  - Assembly can contain multiple modules
  - Assemblies and modules are often 1:1

Program Structure
Organizing Types

- Types are defined in ONE place
  - "One-stop programming"
  - No header and source files to synchronize
  - Code is written "in-line"
  - Declaration and definition are one and the same
  - A type must be fully defined in one file
    - Can't put individual methods in different files
- No declaration order dependence
  - No forward references required

Program Structure
Namespaces

- Namespaces provide a way to uniquely identify a type
- Provides logical organization of types
- Namespaces can span assemblies
- Can nest namespaces
- There is no relationship between namespaces and file structure (unlike Java)
- The fully qualified name of a type includes all namespaces

Program Structure
Namespaces

```
namespace N1 {           // N1
  class C1 {             // N1.C1
    class C2 {           // N1.C1.C2
    }
  }
  namespace N2 {         // N1.N2
    class C2 {           // N1.N2.C2
    }
  }
}
```

Program Structure
Namespaces

- The using statement lets you use types without typing the fully qualified name
- Can always use a fully qualified name

```
using N1;

C1 a;               // The N1. is implicit
N1.C1 b;            // Fully qualified name

C2 c;               // Error! C2 is undefined
N1.N2.C2 d;         // One of the C2 classes
C1.C2 e;            // The other one
```

Program Structure
Namespaces

- The `using` statement also lets you create aliases

```
using C1 = N1.N2.C1;
using N2 = N1.N2;

C1 a;               // Refers to N1.N2.C1
N2.C1 b;            // Refers to N1.N2.C1
```

Program Structure
Namespaces

- Best practice: Put all of your types in a unique namespace
- Have a namespace for your company, project, product, etc.
- Look at how the .NET Framework classes are organized

Program Structure
References

- In Visual Studio you specify references for a project
- Each reference identifies a specific assembly
- Passed as reference (`/r` or `/reference`) to the C# compiler

```
csc HelloWorld.cs /reference:System.WinForms.dll
```

Program Structure
Namespaces vs. References

- Namespaces provide language-level naming shortcuts
  - Don't have to type a long fully qualified name over and over
- References specify which assembly to use

Program Structure
Main Method

- Execution begins at the static `Main()` method
- Can have only one method with one of the following signatures in an assembly
  - `static void Main()`
  - `static int Main()`
  - `static void Main(string[] args)`
  - `static int Main(string[] args)`

Program Structure
Syntax

- Identifiers
  - Names for types, methods, fields, etc.
  - Must be whole word – no white space
  - Unicode characters
  - Begins with letter or underscore
  - Case sensitive
  - Must not clash with keyword
    - Unless prefixed with @

Agenda

- Hello World
- Design Goals of C#
- Types
- Program Structure
- **Statements**
- Operators
- Using Visual Studio.NET
- Using the .NET Framework SDK

Statements
Overview

- High C++ fidelity
- `if`, `while`, `do` require `bool` condition
- `goto` can't jump into blocks
- `switch` statement
  - No fall-through
- `foreach` statement
- `checked` and `unchecked` statements
- Expression statements must do work

```
void Foo() {
   i == 1;    // error
}
```

Statements
Overview

- Statement lists
- Block statements
- Labeled statements
- Declarations
  - Constants
  - Variables
- Expression statements
  - `checked, unchecked`
  - `lock`
  - `using`
- Conditionals
  - `if`
  - `switch`
- Loop Statements
  - `while`
  - `do`
  - `for`
  - `foreach`
- Jump Statements
  - `break`
  - `continue`
  - `goto`
  - `return`
  - `throw`
- Exception handling
  - `try`
  - `throw`

Statements
Syntax

- Statements are terminated with a semicolon (`;`)
- Just like C, C++ and Java
- Block statements { ... } don't need a semicolon

Statements
Syntax

- Comments
  - `// Comment a single line, C++ style`
  - ```
    /* Comment multiple
                        lines,
          C style
    */
    ```
    ```

Statements
Statement Lists & Block Statements

- Statement list: one or more statements in sequence
- Block statement: a statement list delimited by braces { ... }

```
static void Main() {
  F();
  G();
  {         // Start block
    H();
    ;       // Empty statement
    I();
  }         // End block
}
```

Statements
Variables and Constants

```
static void Main() {
  const float pi = 3.14f;
  const int r = 123;
  Console.WriteLine(pi * r * r);

  int a;
  int b = 2, c = 3;
  a = 1;
  Console.WriteLine(a + b + c);
}
```

Statements
Variables and Constants

- The scope of a variable or constant runs from the point of declaration to the end of the enclosing block

Statements
Variables and Constants

- Within the scope of a variable or constant it is an error to declare another variable or constant with the same name

```
{
  int x;
  {
    int x;  // Error: can't hide variable x
  }
}
```

Statements
Variables

- Variables must be assigned a value before they can be used
  - Explicitly or automatically
  - Called definite assignment
- Automatic assignment occurs for static fields, class instance fields and array elements

```
void Foo() {
  string s;
  Console.WriteLine(s);      // Error
}
```

Statements
Labeled Statements & `goto`

- `goto` can be used to transfer control within or out of a block, but not into a nested block

```
static void Find(int value, int[,] values,
                 out int row, out int col) {
   int i, j;
   for (i = 0; i < values.GetLength(0); i++)
      for (j = 0; j < values.GetLength(1); j++)
         if (values[i, j] == value) goto found;
   throw new InvalidOperationException("Not found");
found:
   row = i; col = j;
}
```

Statements
Expression Statements

- Statements must do work
  - Assignment, method call, ++, --, new

```
static void Main() {
  int a, b = 2, c = 3;
  a = b + c;
  a++;
  MyClass.Foo(a,b,c);
  Console.WriteLine(a + b + c);
  a == 2;                      // ERROR!
}
```

Statements
if Statement

- Requires bool expression

```
int Test(int a, int b) {
  if (a > b)
    return 1;
  else if (a < b)
    return -1;
  else
    return 0;
}
```

Statements
switch Statement

- Can branch on any predefined type (including string) or enum
  - User-defined types can provide implicit conversion to these types
- Must explicitly state how to end case
  - With break, goto case, goto label, return, throw or continue
  - Eliminates fall-through bugs
  - Not needed if no code supplied after the label

Statements
switch Statement

```
int Test(string label) {
  int result;
  switch(label) {
    case null:
      goto case "runner-up";
    case "fastest":
    case "winner":
      result = 1; break;
    case "runner-up":
      result = 2; break;
    default:
      result = 0;
  }
  return result;
}
```

Statements
while Statement

- Requires bool expression

```
int i = 0;
while (i < 5) {
  ...
  i++;
}
```

```
int i = 0;
do {
  ...
  i++;
}
while (i < 5);
```

```
while (true) {
  ...
}
```

Statements
for Statement

```
for (int i=0; i < 5; i++) {
  ...
}
```

```
for (;;) {
  ...
}
```

Statements
`foreach` Statement

- Iteration of arrays

```
public static void Main(string[] args) {
   foreach (string s in args)
      Console.WriteLine(s);
}
```

Statements
`foreach` Statement

- Iteration of user-defined collections
- Created by implementing `IEnumerable`

```
foreach (Customer c in customers.OrderBy("name")) {
   if (c.Orders.Count != 0) {
      ...
   }
}
```

Statements
Jump Statements

- `break`
  - Exit inner-most loop
- `continue`
  - End iteration of inner-most loop
- `goto <label>`
  - Transfer execution to label statement
- `return [<expression>]`
  - Exit a method
- `throw`
  - See exception handling

Statements
Exception Handling

- Exceptions are the C# mechanism for handling unexpected error conditions
- Superior to returning status values
  - Can't be ignored
  - Don't have to handled at the point they occur
  - Can be used even where values are not returned (e.g. accessing a property)
  - Standard exceptions are provided

Statements
Exception Handling

- `try...catch...finally` statement
- `try` block contains code that could throw an exception
- `catch` block handles exceptions
  - Can have multiple catch blocks to handle different kinds of exceptions
- `finally` block contains code that will always be executed
  - Cannot use jump statements (e.g. `goto`) to exit a finally block

Statements
Exception Handling

- `throw` statement raises an exception
- An exception is represented as an instance of `System.Exception` or derived class
  - Contains information about the exception
  - Properties
    - `Message`
    - `StackTrace`
    - `InnerException`
- You can rethrow an exception, or catch one exception and throw another

Statements
Exception Handling

```
try {
  Console.WriteLine("try");
  throw new Exception("message");
}
catch (ArgumentNullException e) {
  Console.WriteLine("caught null argument");
}
catch {
  Console.WriteLine("catch");
}
finally {
  Console.WriteLine("finally");
}
```

Statements
Synchronization

- Multi-threaded applications have to protect against concurrent access to data
  - Must prevent data corruption
- The `lock` statement uses an instance to provide mutual exclusion
  - Only one `lock` statement can have access to the same instance
  - Actually uses the .NET Framework `System.Threading.Monitor` class to provide mutual exclusion

Statements
Synchronization

```
public class CheckingAccount {
  decimal balance;
  public void Deposit(decimal amount) {
    lock (this) {
      balance += amount;
    }
  }
  public void Withdraw(decimal amount) {
    lock (this) {
      balance -= amount;
    }
  }
}
```

Statements
`using` Statement

- C# uses automatic memory management (garbage collection)
  - Eliminates most memory management problems
- However, it results in non-deterministic finalization
  - No guarantee as to when and if object destructors are called

Statements
`using` Statement

- Objects that need to be cleaned up after use should implement the `System.IDisposable` interface
  - One method: `Dispose()`
- The `using` statement allows you to create an instance, use it, and then ensure that `Dispose` is called when done
  - `Dispose` is guaranteed to be called, as if it were in a `finally` block

Statements
`using` Statement

```
public class MyResource : IDisposable {
  public void MyResource() {
    // Acquire valuble resource
  }
  public void Dispose() {
    // Release valuble resource
  }
  public void DoSomething() {
    ...
  }
}
```

```
using (MyResource r = new MyResource()) {
  r.DoSomething();
}                    // r.Dispose() is called
```

Statements
checked and unchecked Statements

- The checked and unchecked statements allow you to control overflow checking for integral-type arithmetic operations and conversions
- checked forces checking
- unchecked forces no checking
- Can use both as block statements or as an expression
- Default is unchecked
- Use the /checked compiler option to make checked the default

Statements
Basic Input/Output Statements

- Console applications
  - System.Console.WriteLine();
  - System.Console.ReadLine();
- Windows applications
  - System.WinForms.MessageBox.Show();

```
string v1 = "some value";
MyObject v2 = new MyObject();
Console.WriteLine("First is {0}, second is {1}",
                  v1, v2);
```

Agenda

- Hello World
- Design Goals of C#
- Types
- Program Structure
- Statements
- **Operators**
- Using Visual Studio.NET
- Using the .NET Framework SDK

Operators
Overview

- C# provides a fixed set of operators, whose meaning is defined for the predefined types
- Some operators can be overloaded (e.g. +)
- The following table summarizes the C# operators by category
  - Categories are in order of decreasing precedence
  - Operators in each category have the same precedence

Operators
Precedence

Category	Operators
Primary	Grouping: (x) Member access: x.y Method call: f(x) Indexing: a[x] Post-increment: x++ Post-decrement: x— Constructor call: new Type retrieval: typeof Arithmetic check on: checked Arithmetic check off: unchecked

Operators
Precedence

Category	Operators
Unary	Positive value of: + Negative value of: - Not: ! Bitwise complement: ~ Pre-increment: ++x Post-decrement: --x Type cast: (T)x
Multiplicative	Multiply: * Divide: / Division remainder: %

Operators
Precedence

Category	Operators
Additive	Add: + Subtract: -
Shift	Shift bits left: << Shift bits right: >>
Relational	Less than: < Greater than: > Less than or equal to: <= Greater than or equal to: >= Type equality/compatibility: is Type conversion: as

Operators
Precedence

Category	Operators
Equality	Equals: == Not equals: !=
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	|
Logical AND	&&
Logical OR	||

Operators
Precedence

Category	Operators
Ternary conditional	?:
Assignment	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Operators
Associativity

- Assignment and ternary conditional operators are right-associative
  - Operations performed right to left
  - `x = y = z` evaluates as `x = (y = z)`
- All other binary operators are left-associative
  - Operations performed left to right
  - `x + y + z` evaluates as `(x + y) + z`
- Use parentheses to control order

Agenda

- Hello World
- Design Goals of C#
- Types
- Program Structure
- Statements
- Operators
- **Using Visual Studio.NET**
- Using the .NET Framework SDK

Using Visual Studio.NET

- Types of projects
  - Console Application
  - Windows Application
  - Web Application
  - Web Service
  - Windows Service
  - Class Library
  - ...

Using Visual Studio.NET

- Windows
  - Solution Explorer
  - Class View
  - Properties
  - Output
  - Task List
  - Object Browser
  - Server Explorer
  - Toolbox

Using Visual Studio.NET

- Building
- Debugging
  - Break points
- References
- Saving

Agenda

- Hello World
- Design Goals of C#
- Types
- Program Structure
- Statements
- Operators
- Using Visual Studio.NET
- **Using the .NET Framework SDK**

Using .NET Framework SDK

- Compiling from command line

```
csc /r:System.WinForms.dll class1.cs file1.cs
```

More Resources

- http://msdn.microsoft.com
- http://windows.oreilly.com/news/hejlsberg_0800.html
- http://www.csharphelp.com/
- http://www.csharp-station.com/
- http://www.csharpindex.com/
- http://msdn.microsoft.com/msdnmag/issues/0900/csharp/csharp.asp
- http://www.hitmill.com/programming/dotNET/csharp.html
- http://www.c-sharpcorner.com/
- http://msdn.microsoft.com/library/default.asp?URL=/library/dotnet/csspec/vclrfcsharpspec_Start.htm