

Microsoft .NET Bootcamp

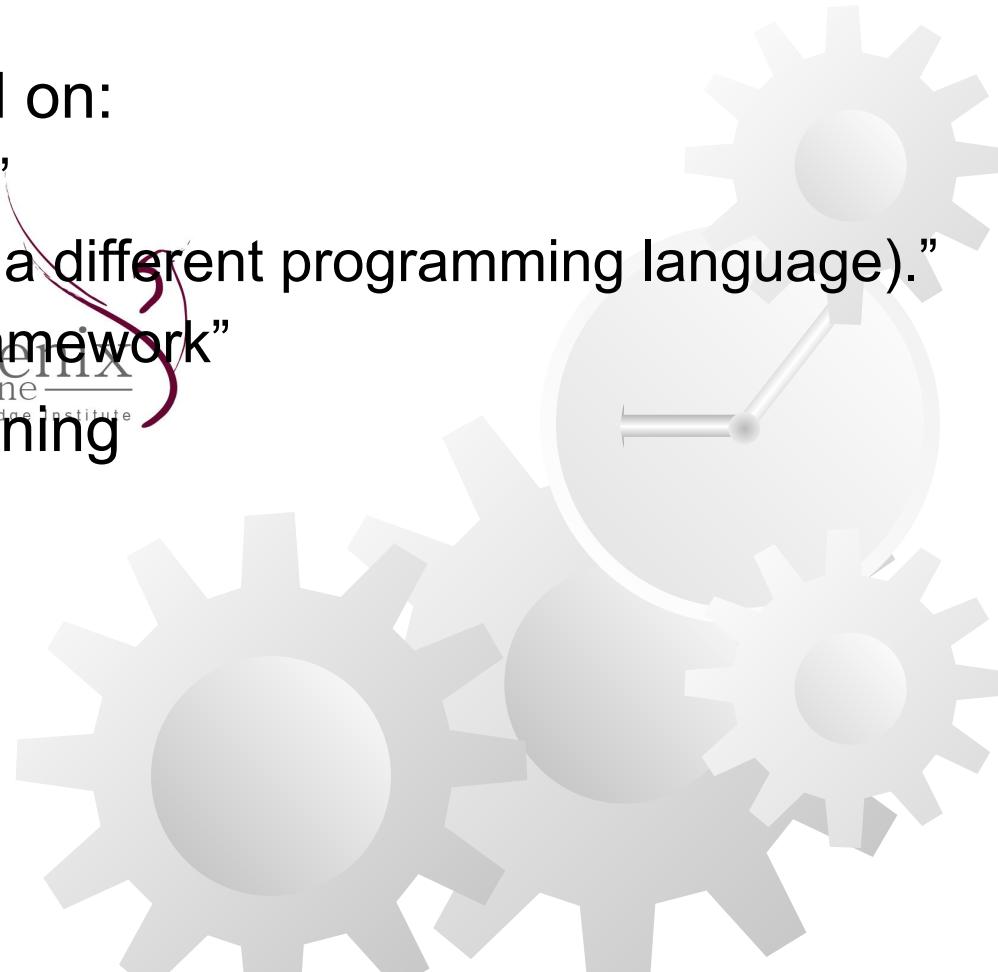


Vener Guevarra



Introduction

- Name
- Level and tenure
- Your self assessment based on:
 - “I don’t know why I’m here.”
 - “I am a programmer (using a different programming language).”
 - “I am familiar with .NET Framework”
- Your expectations of the training



Bootcamp

- Attendance (15%)
- Lab Exercises (40%)
- Checkpoint Exam (15 %)
- Final Exam (30%)



Bootcamp Requirements

- Visual Studio 2019 or 2022
- SQL Server
- Visual Studio Code
- NodeJS
- GIT
- Github Account
- Google Drive Access



Schedule

- 9am - 6pm
- 1 hour Lunch break
- 15 minutes AM break
- 15 minutes PM break
- Lab Exercise 1.5/2 hours



.NET Core



.NET Core/ASP.NET Core/EF Core

.NET Core 3.1

+

ASP.NET Core 3.1

+

EF Core 5

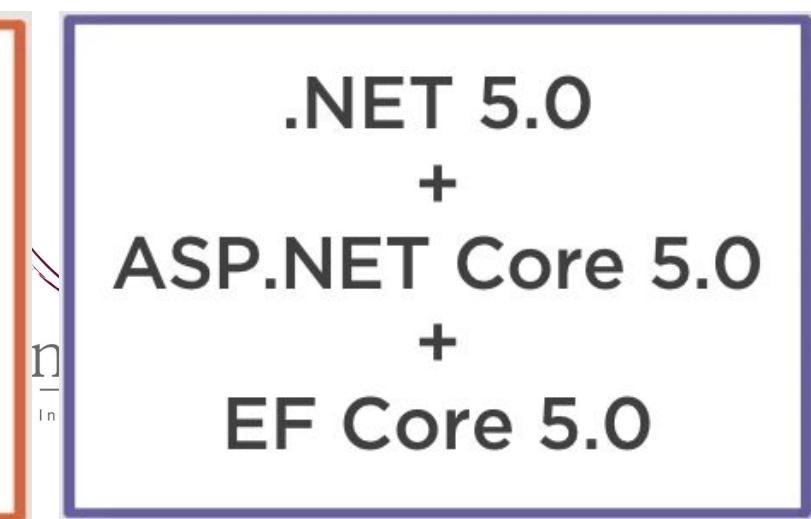
.NET 5.0

+

ASP.NET Core 5.0

+

EF Core 5.0



Introduction to ORM

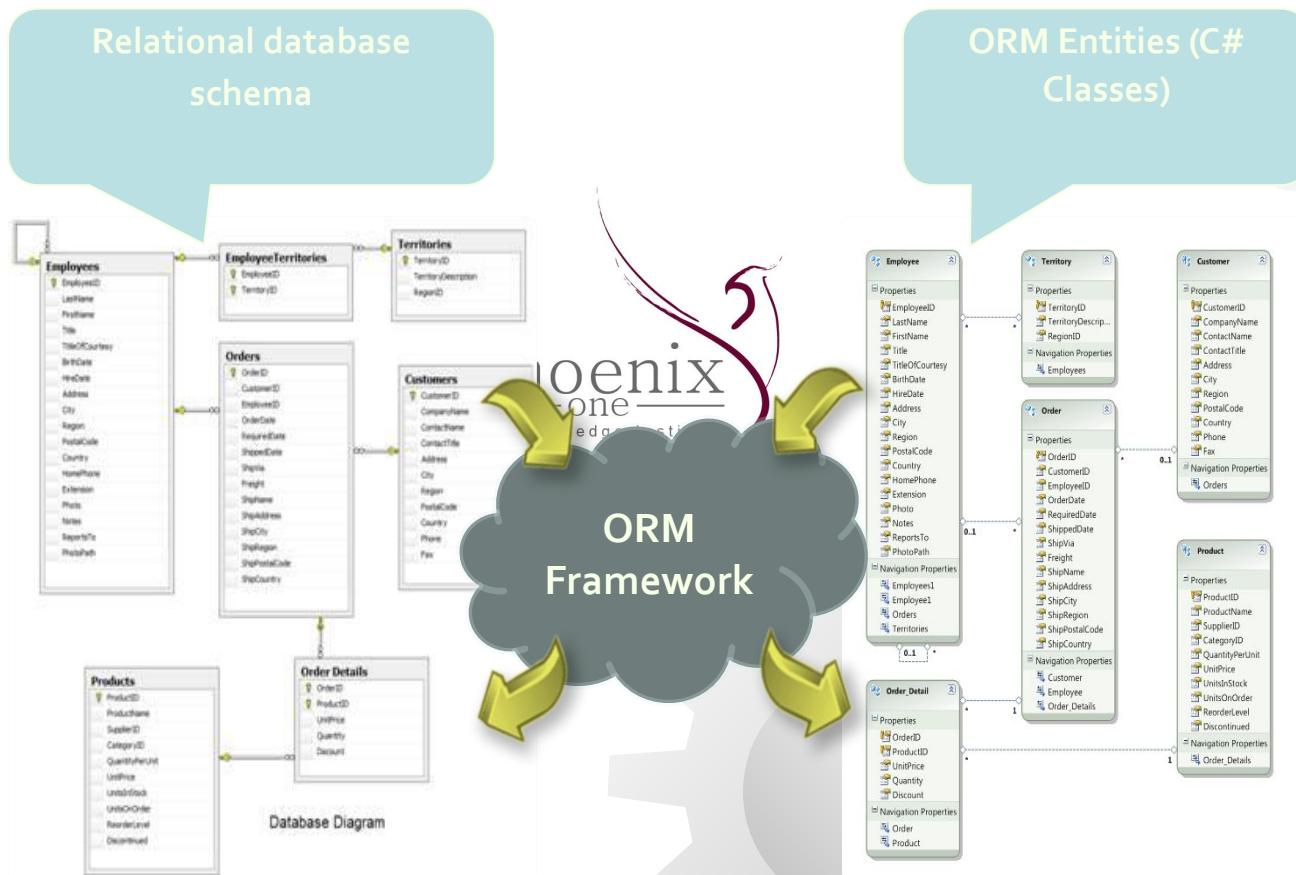
- Object-Relational Mapping (ORM) is a programming technique for automatic mapping and converting data
- Between relational database tables and object-oriented classes and objects
- ORM creates a "virtual object database"
- ORM frameworks automate the ORM process



ORM Frameworks

- ORM frameworks typically provide the following functionality:
 - Creating object model by database schema (Database First)
 - Creating database schema by object model (Code First)
 - Querying data by object-oriented API
- Data manipulation operations
 - CRUD – create, retrieve, update, delete
- ORM frameworks automatically generate SQL to perform the requested data operations

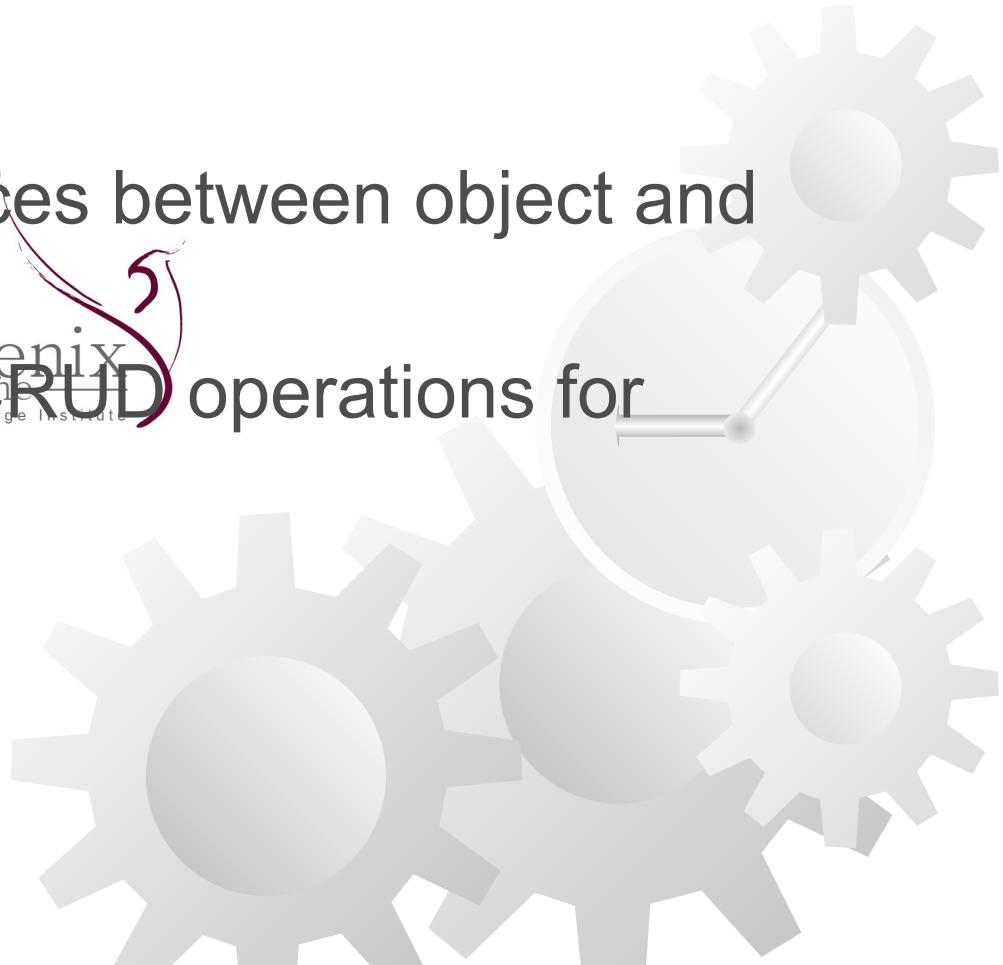
ORM Mapping Example



ORM Advantages

- Developer productivity
 - Writing less code
- Abstract from differences between object and relational world
- Manageability of the **CRUD** operations for complex relationships
- Easier maintainability

Phoenix
One
The Knowledge Institute



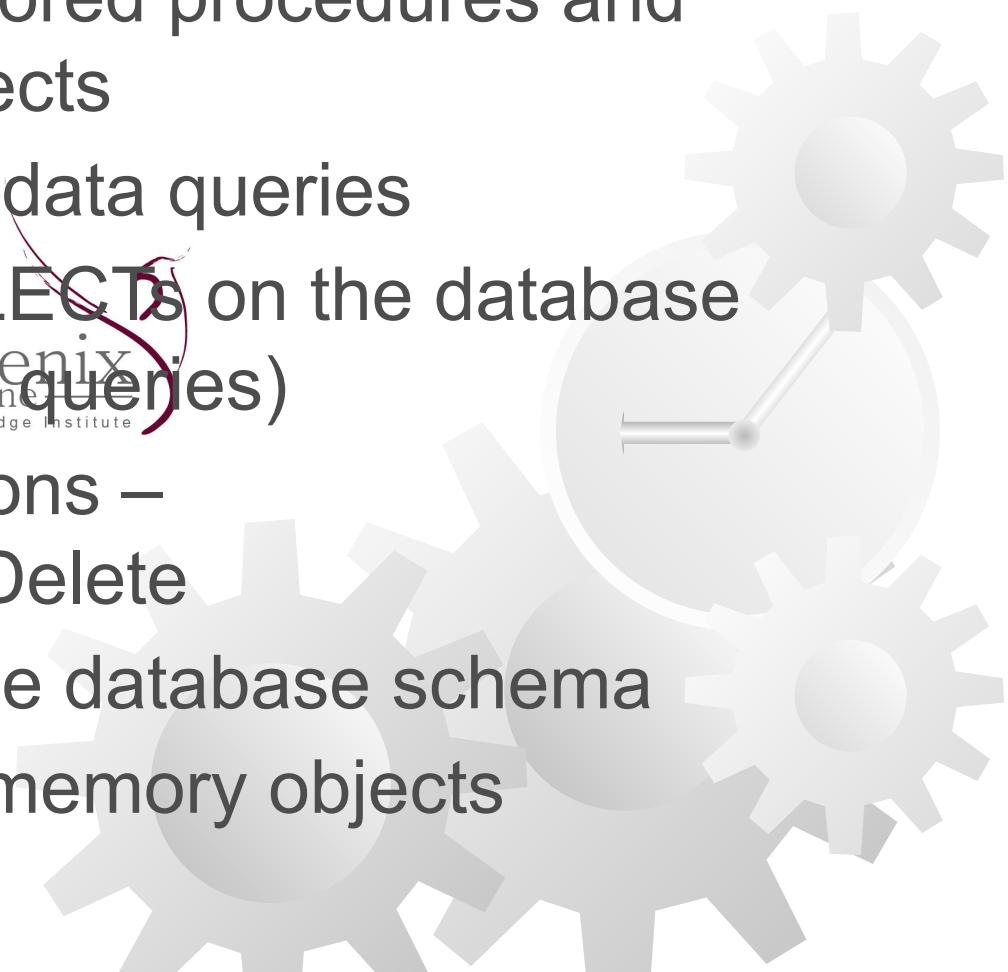
Entity Framework (EF) Overview

- Provides a runtime infrastructure for managing SQL-based database data as .NET objects
- The relational database schema is mapped to an object model (classes and associations)
- Visual Studio has built-in tools for generating Entity Framework SQL data mappings

Phoenix
The Knowledge Institute

Entity Framework Features

- Maps tables, views, stored procedures and functions as .NET objects
- Provides LINQ-based data queries
- Executed as SQL SELECTs on the database server (parameterized queries)
- Built-in CRUD operations – Create/Read/Update/Delete
- Creating or deleting the database schema
- Tracks changes to in-memory objects



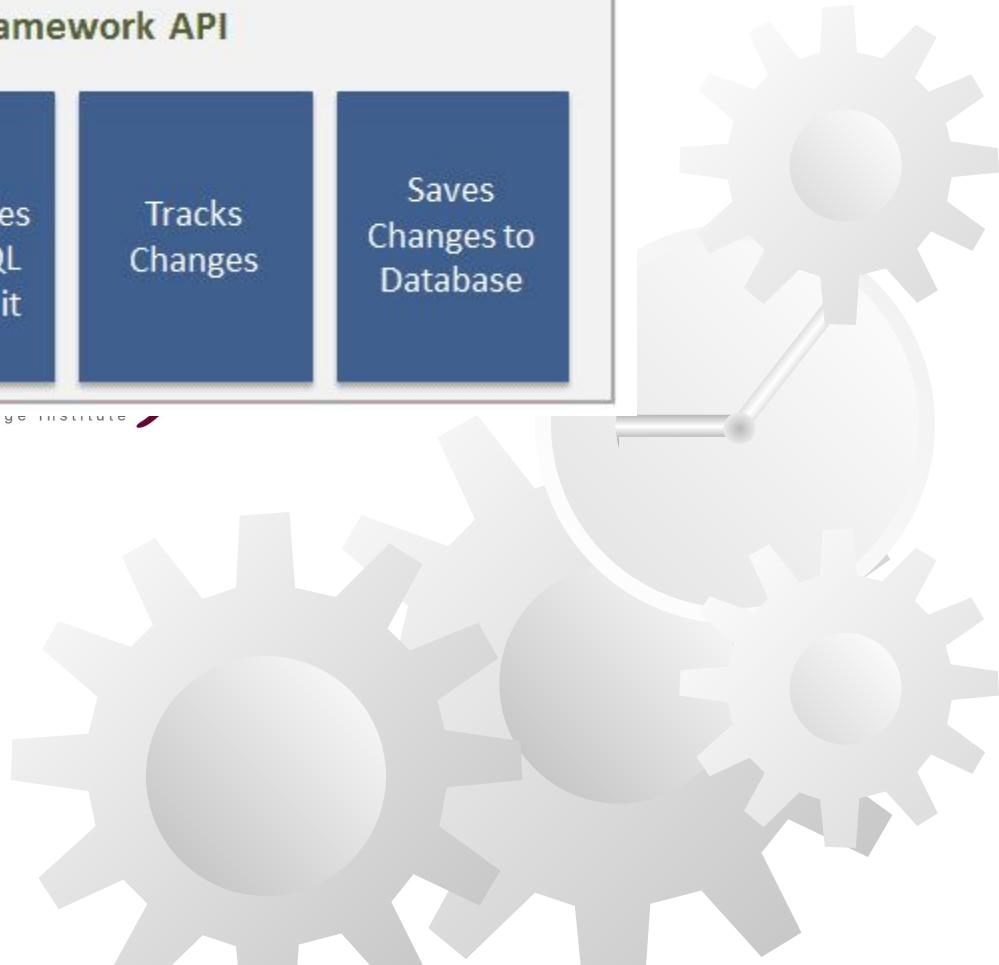
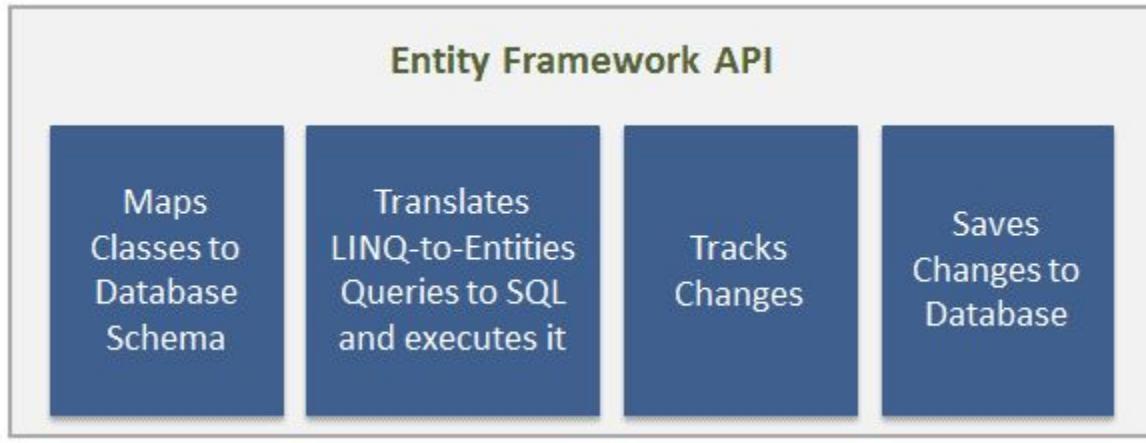
Entity Framework Components (1 of 2)

- The DbContext class
 - DbContext holds the database connection and the entity classes
 - Provides LINQ-based data access
 - Implements identity tracking, change tracking, and API for CRUD operations
- Entity classes
 - Each database table is typically mapped to a single entity class (C# class)

Entity Framework Components (2 of 2)

- Associations (Relationship Management)
 - An association is a primary key / foreign key based relationship between two entity classes
 - Allows navigation from one entity to another, e.g. Student.Courses
- Concurrency control
 - Entity Framework uses optimistic concurrency control (no locking by default)
 - Provides automatic concurrency conflict detection and means for conflicts resolution

Entity Framework API



C# Programming



Module 1 : Introduction to Object Orientation



Module Objectives

Upon completion of this module, participants should be able to explain the following object oriented concepts:

- Definition of Classes and Instances
- Decomposition
- Collaboration
- Class Design Practices and Techniques
- Inheritance, Encapsulation, and Polymorphism
- Introduction to UML: Class and Collaboration Diagrams



Module objectives

Module Agenda

Object Orientation
Concepts

Class Design Practices
and Techniques

Introduction to UML



Object Orientation Concepts



Object Technology

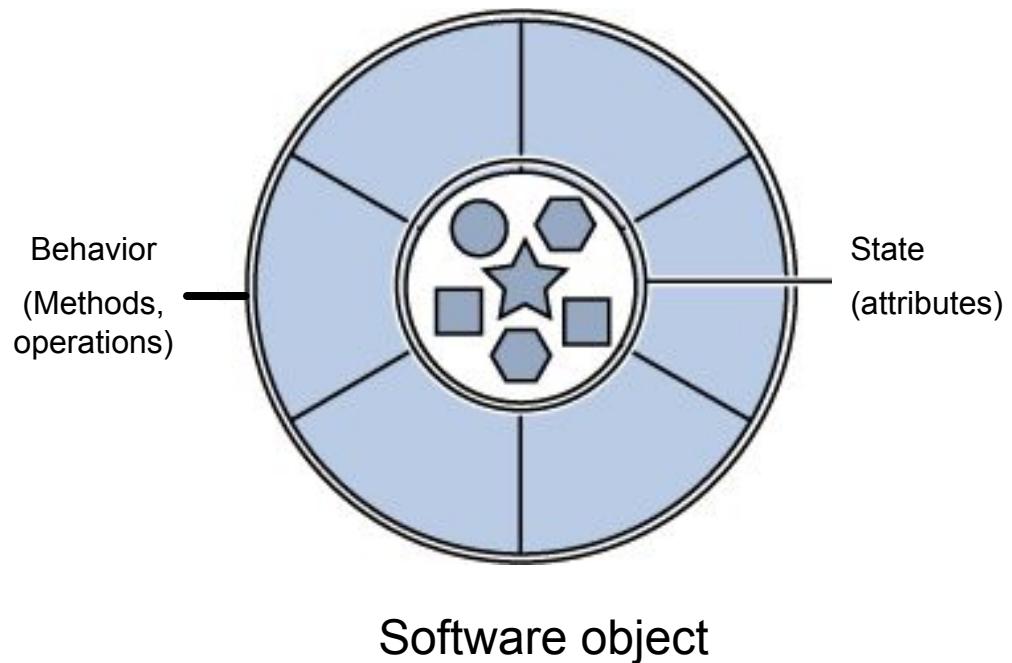
- Object Technology
 - A set of principles guiding software construction together with languages, databases, and other tools that support those principles.
 - . (*Object Technology - A Manager's Guide*, Taylor, 1997)



What is an Object?

■ An Object is an entity with

- A unique identity
- A State
- A Behavior



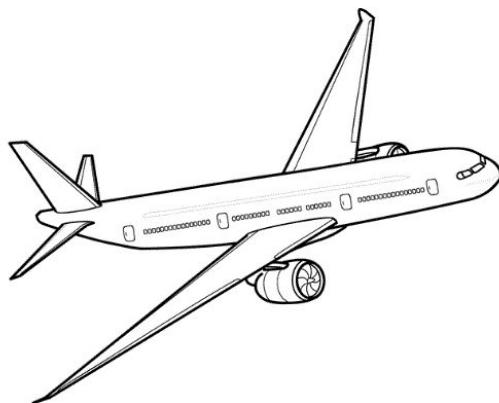
What is an object?

- An object can be
 - A person
 - A thing in the “real world”
 - A model of reality
 - A tangible or visible thing
 - A thing to which action or thought can be directed



What is a Class?

- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- An object is an instance of a class.



Class: Airplane



Representing Classes in the UML

- A class is represented using a rectangle with compartments.

Employee	
- name	
- employeeID:UniqueID	
- hireDate	
- status	
- discipline	
+ submitReport()	
+ setMaxLoad()	
+takeVacationLeave()	
+programmingLoad()	



Employee Liza

Relationship between Classes and Objects

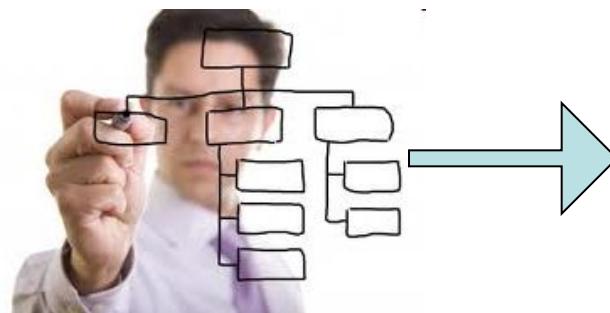
- A class is an abstract definition of an object.
 - It defines the structure and behavior of each object in the class.
 - It serves as a template for creating objects.
- Classes are not collections of objects.



Liza



Ed

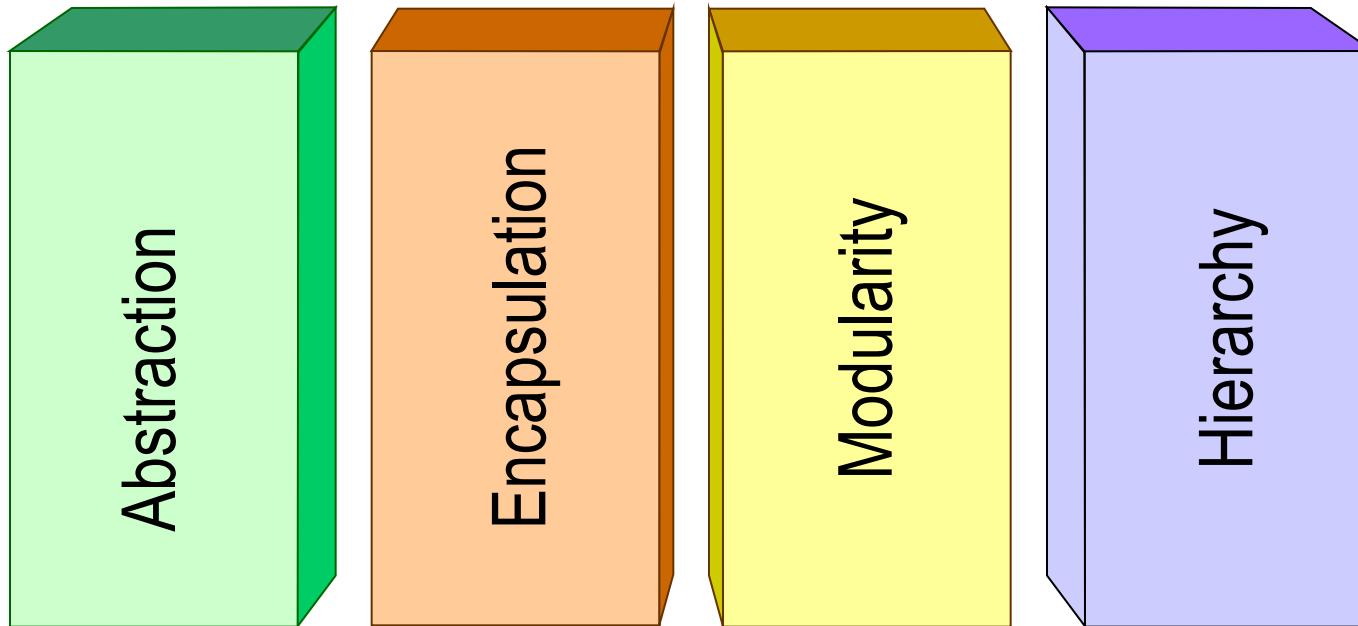


John

Employee
<ul style="list-style-type: none"> - name - employeeID:UniqueID - hireDate - status - discipline
<ul style="list-style-type: none"> + submitReport() + setMaxLoad() +takeVacationLeave() +programmingLoad()

Basic Principles of Object Orientation

Object Orientation

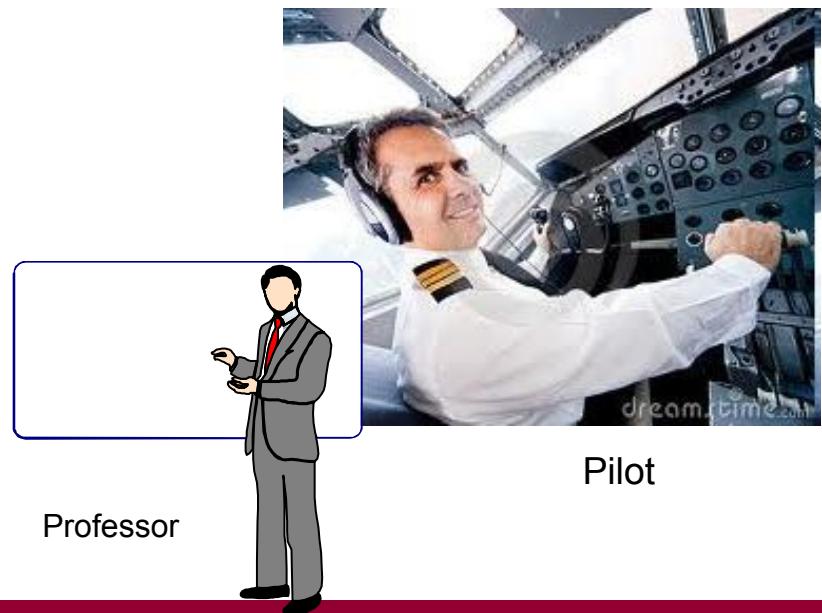


Abstraction

- **Abstraction** can be defined as:

- Any model that includes the most important, essential, or distinguishing aspects of something while suppressing or ignoring less important, immaterial, or diversionary details.
- The result of removing distinctions so as to emphasize commonalities.
(Dictionary of Object Technology, Firesmith, Eykholt, 1995)

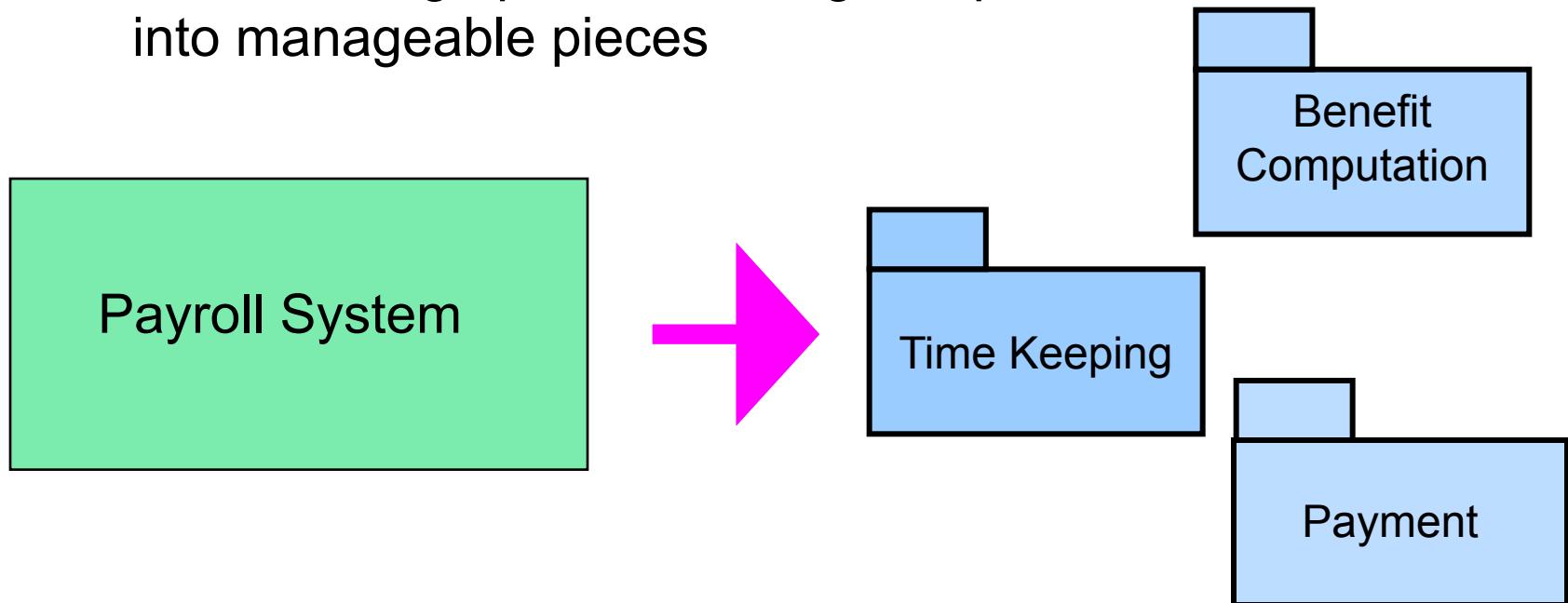
- Defines a boundary relative to the perspective of the viewer.



Modularity

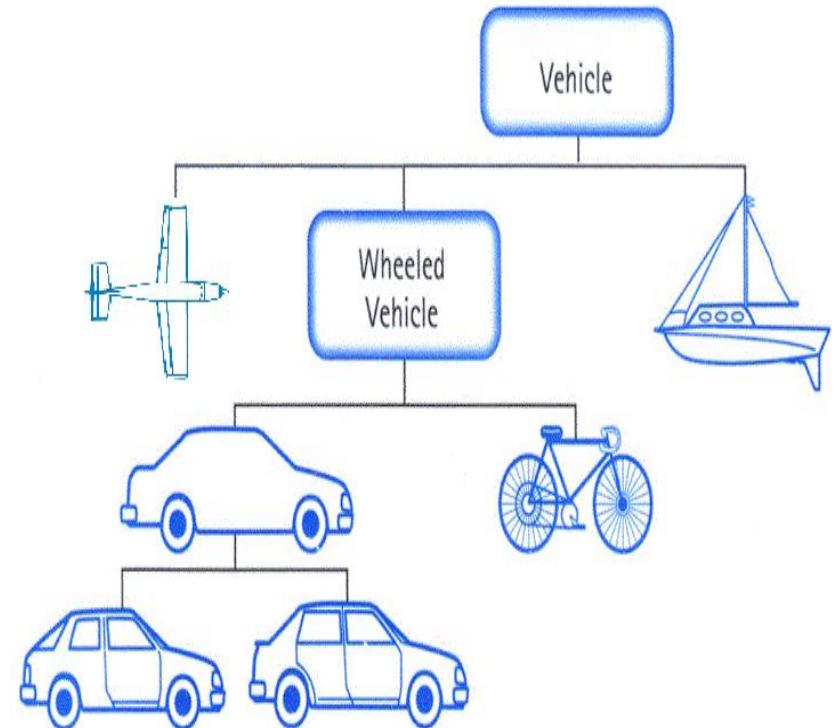
- Modularity is the logical and physical decomposition of things into small, simple groupings which increase the achievement of software-engineering goals. (Dictionary of Object Technology (Firesmith, Eykholt, 1995)

The breaking up of something complex
into manageable pieces

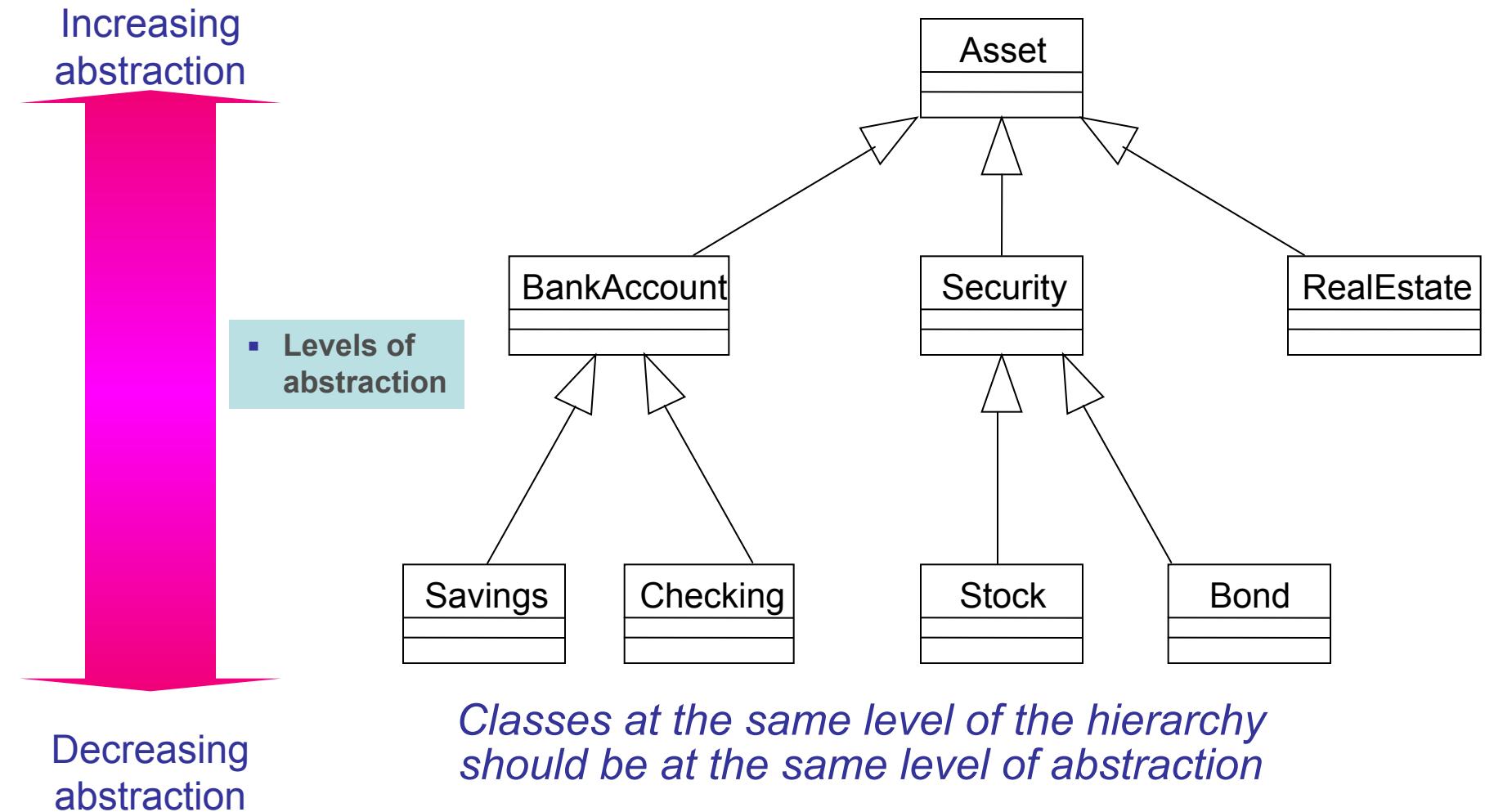


What is Hierarchy?

- Hierarchy is: “Any ranking or ordering of abstractions into a tree-like structure.
- Kinds:
 - Aggregation hierarchy
 - Class hierarchy
 - Containment hierarchy
 - Generalization hierarchy
 - Specialization hierarchy
 - Inheritance hierarchy
 - Partition hierarchy
 - Type hierarchy”
- Dictionary of Object Technology (Firesmith, Eykholt,, 1995),



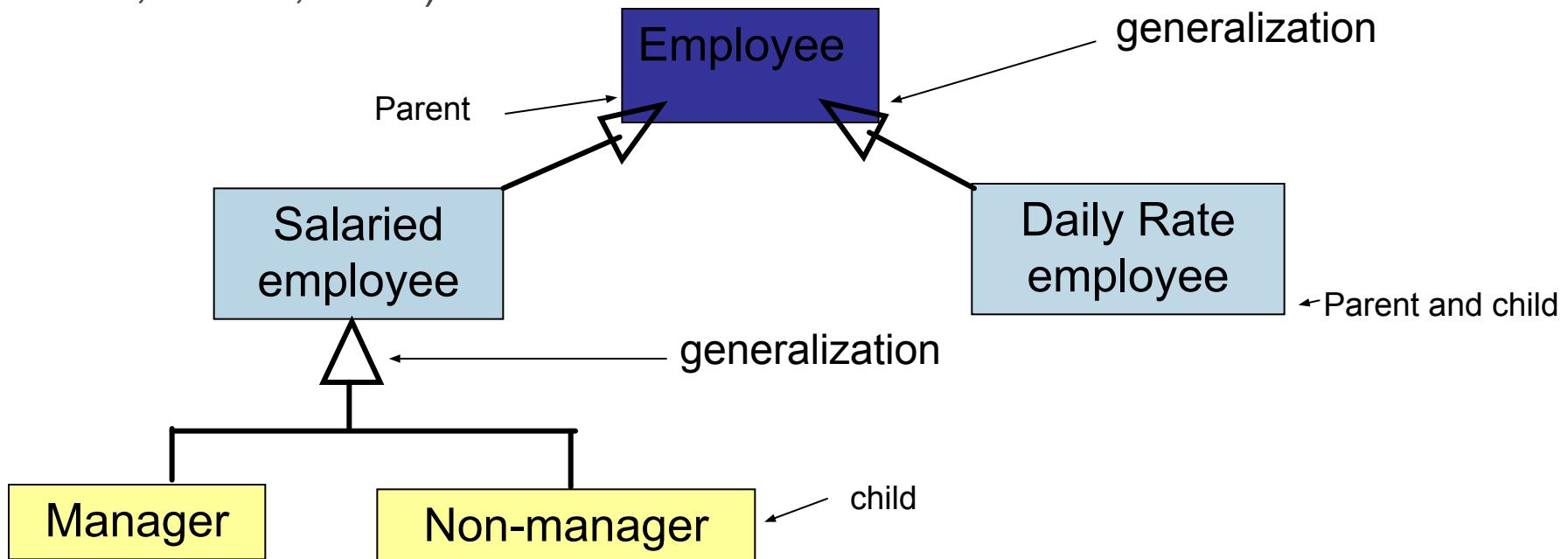
Hierarchy



Generalization

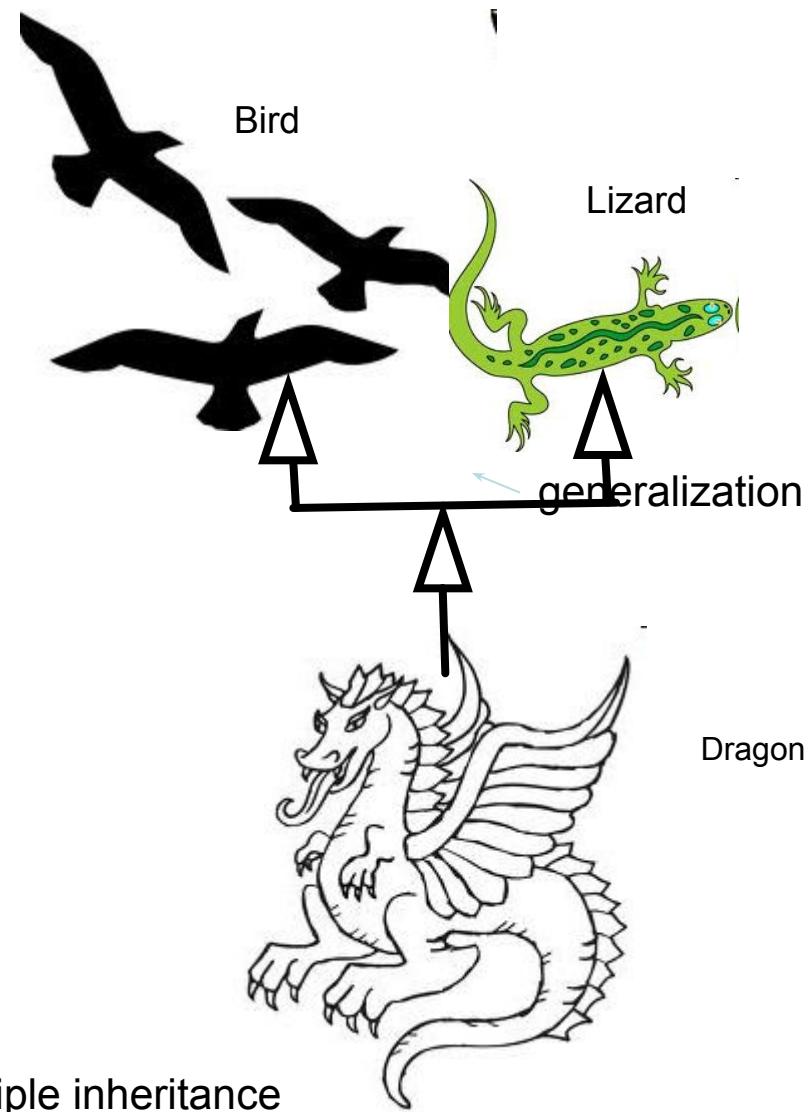
- **Generalization** can be defined as:

A specialization/generalization relationship, in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). (*The Unified Modeling Language User Guide*, Booch, 1999)



What Is Generalization?

- A relationship among classes where one class shares the structure and/or behavior of one or more classes
- Is an “is a kind of” relationship



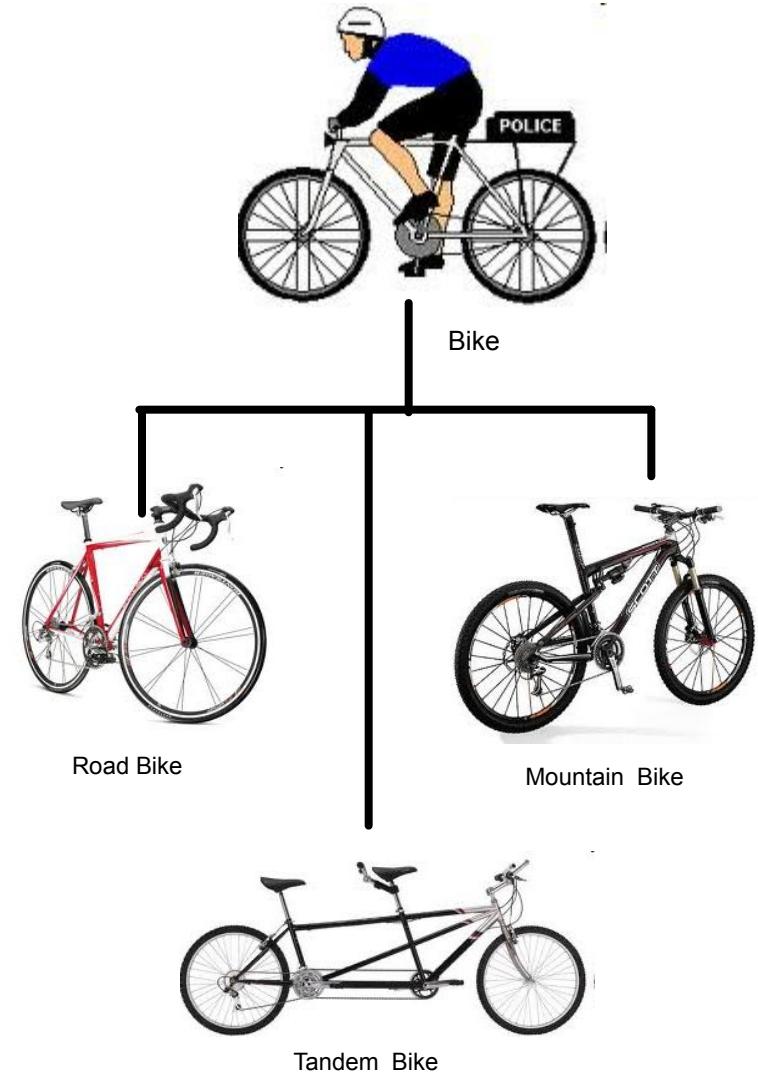
Generalization vs Inheritance

- The terms “generalization” and “inheritance” are generally interchangeable.
- Generalization is a relationship.
- Inheritance is a mechanism that generalization relationship represents.

Inheritance Definition

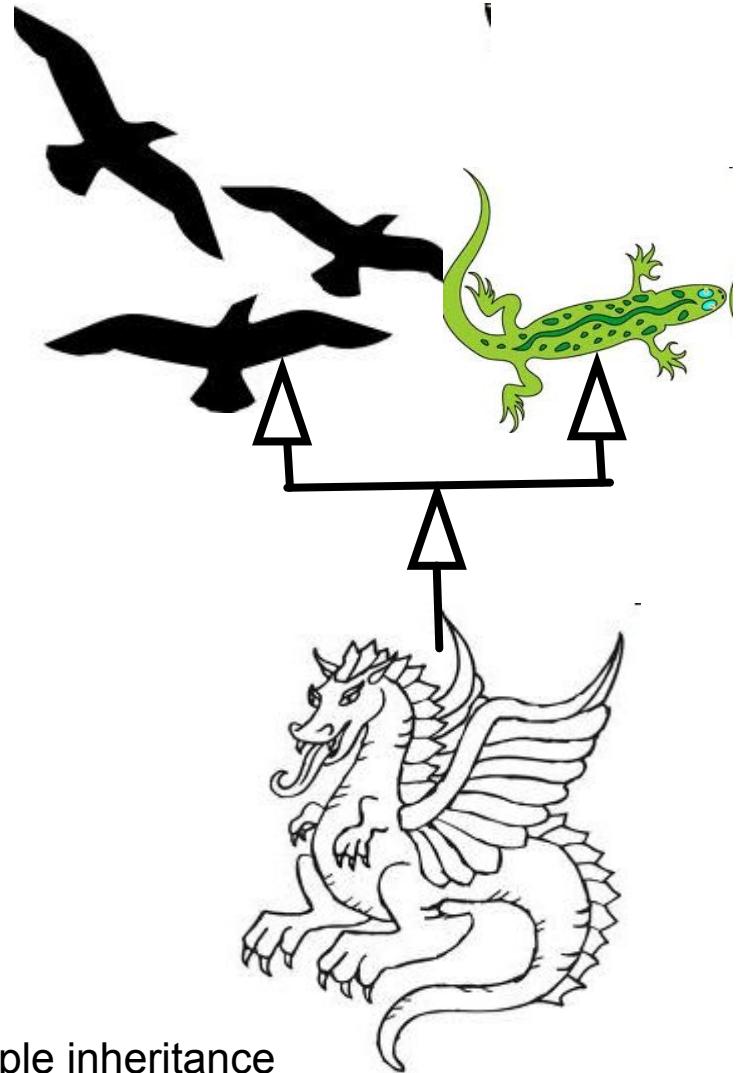
- Inheritance is the mechanism by which more-specific elements incorporate the structure and behavior of more-general elements.

(The Unified Modeling Language User Guide, Booch, 1999)



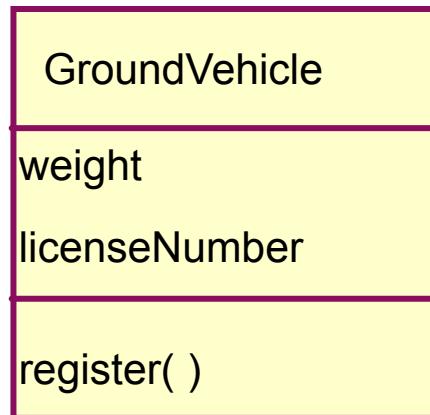
Inheritance

- Defines a hierarchy of abstractions in which a subclass inherits from one or more super classes
 - Single inheritance
 - Multiple inheritance

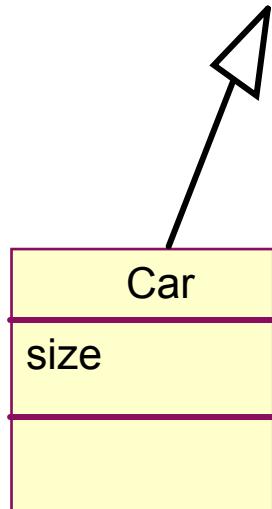


Example: What Gets Inherited

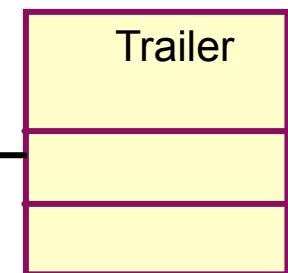
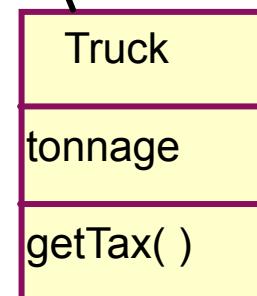
*Superclass
(parent)*



Subclass



generalization



Encapsulation and Information Hiding

- Encapsulation is the ‘bundling together’ of data and behavior so that they are inseparable
- Information Hiding is the process of hiding all internal details of an object except those strictly necessary to the outside



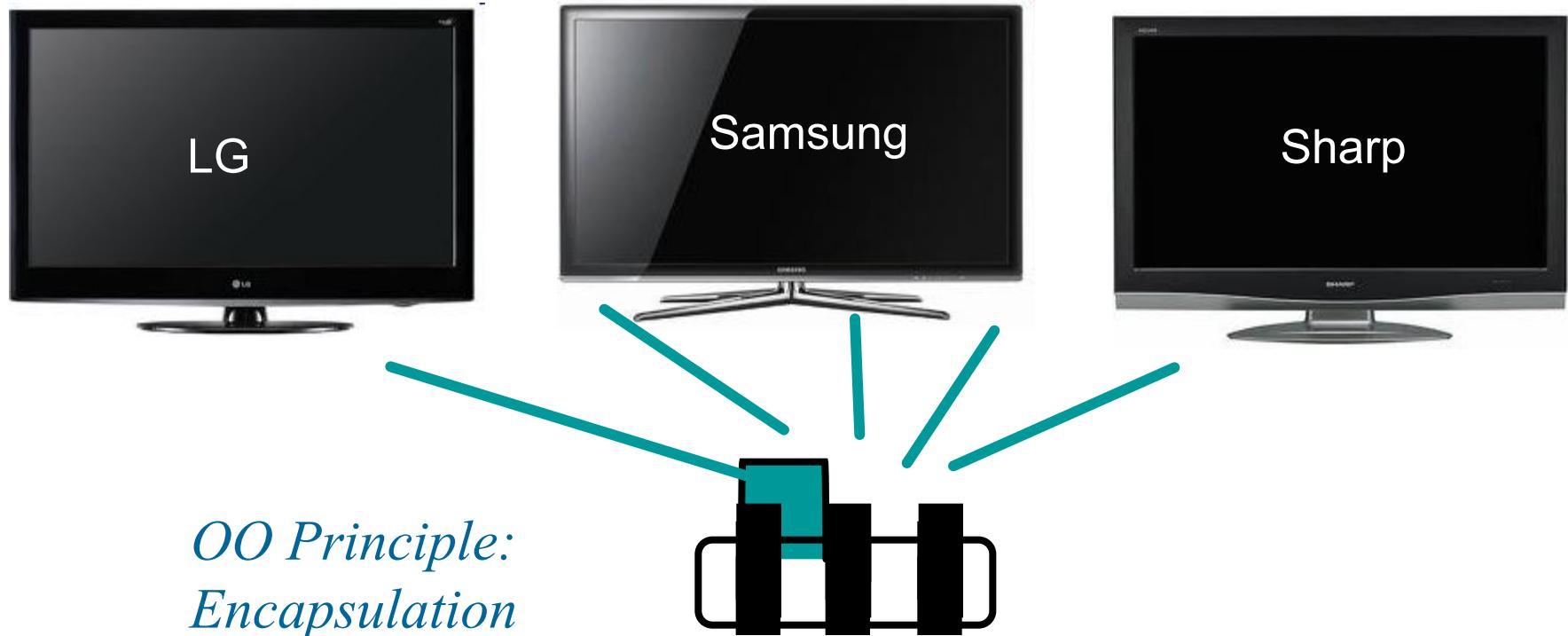
Encapsulation and Information Hiding

- A Pig shows:
 - Encapsulation
 - By holding *both* its *methods* (e.g. how to eat)
 - And its data (e.g. how hungry it is)
 - Information hiding
 - There is no *direct* way to find out how hungry a pig is (nor how it digests its food)
- A *String* shows:
 - Encapsulation
 - By holding *both* its *methods* (e.g. reverseString)
 - And its data (i.e. the characters in the String)
 - Information hiding
 - The only way to find out the characters at certain position is to send the String a message (e.g. at: or printString)



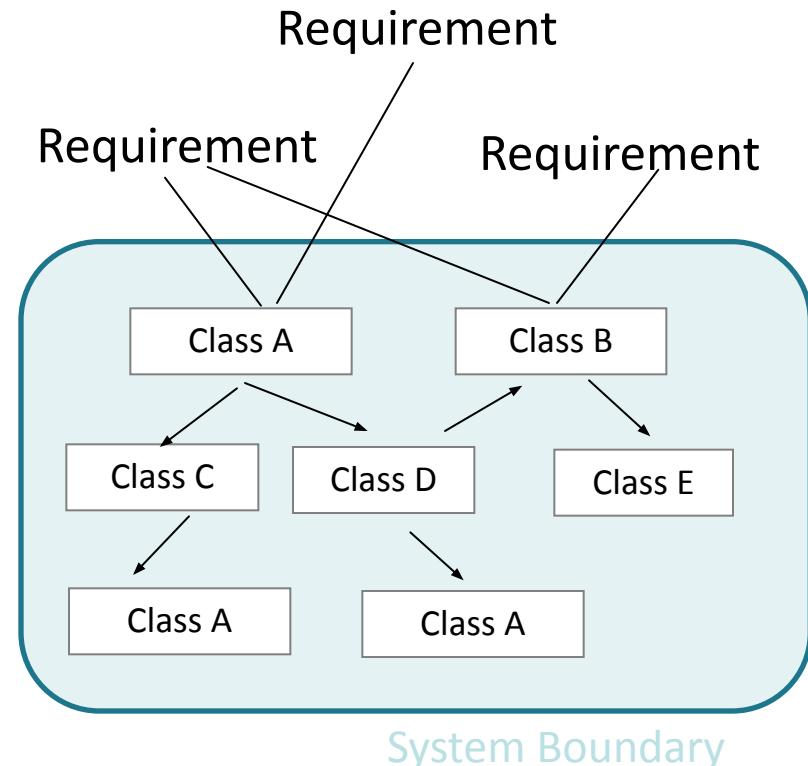
Polymorphism

- It is derived from a *Greek word* 'polumorphos' with two roots: Polus (many) + Morphe (shape/form).
- The ability to hide many different implementations behind a single interface.



Decomposition

- Object-oriented *decomposition*
breaks a large system down into progressively smaller classes or objects that are responsible for some part of the problem domain.
- According to Booch, algorithmic decomposition is a necessary part of object-oriented analysis and design, but object-oriented systems start with and emphasize decomposition into classes.



Collaboration

- In object-oriented systems, Collaboration occurs among objects that mimic real-world objects inside an application.
- Collaboration is a set of objects and a protocol (i.e. a set of allowed behaviors) that determines how these objects interact.



Collaboration

Introduction to UML: Class and Collaboration Diagrams



What is UML?

- UML stands for “Unified Modelling Language”
- It is an industry-standard graphical language for



Specifying



Visualizing



Constructing



Documenting



Business Modeling



Communicating

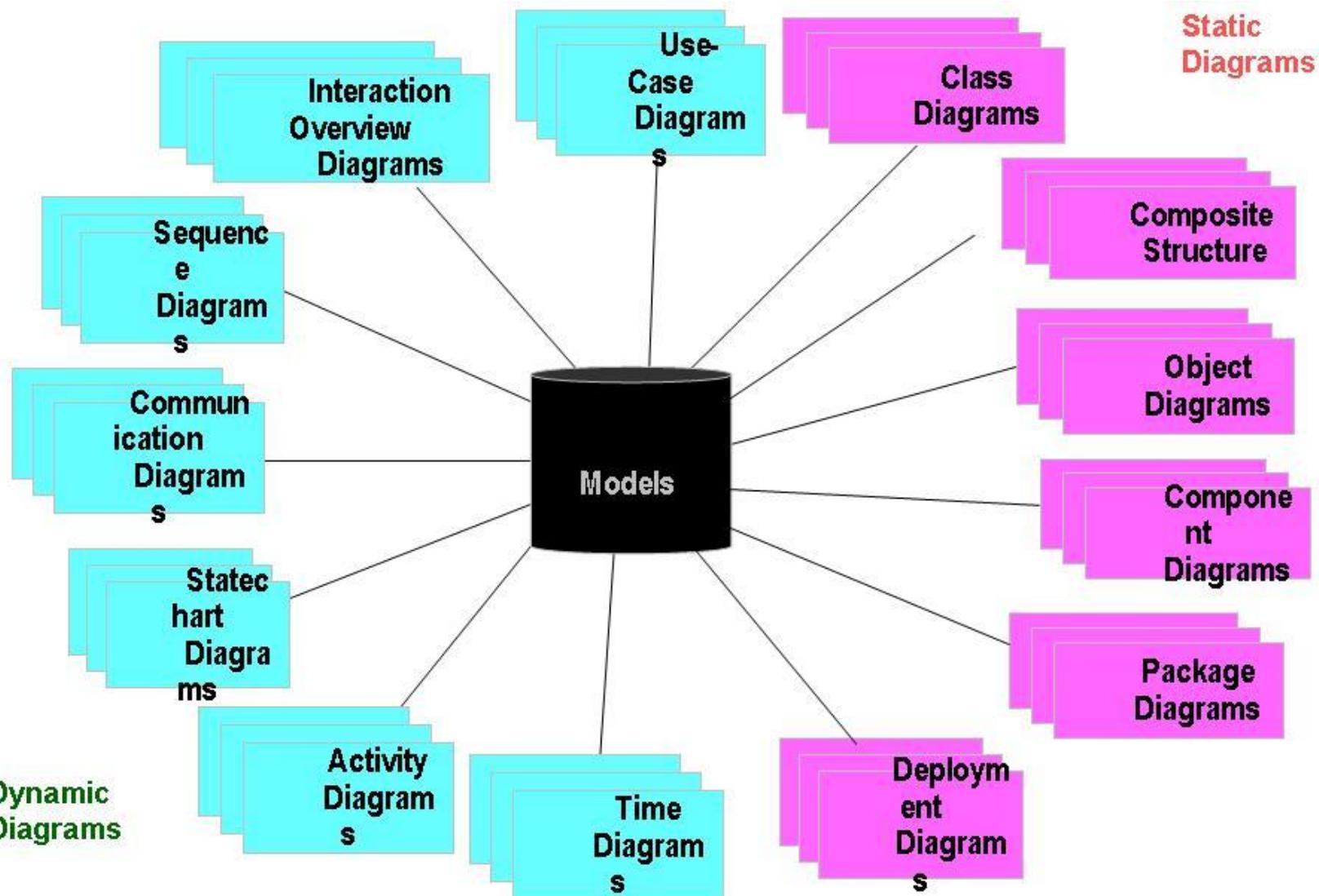
- the artifacts of software intensive systems

Goals of UML

- Provide users with ready-to-use, expressive visual modeling language to develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular languages and processes.
- Provide formal basis for understanding the modeling language.
- Support higher-level development concepts such as collaborations, frameworks, patterns and components.
- Integrate best practices.

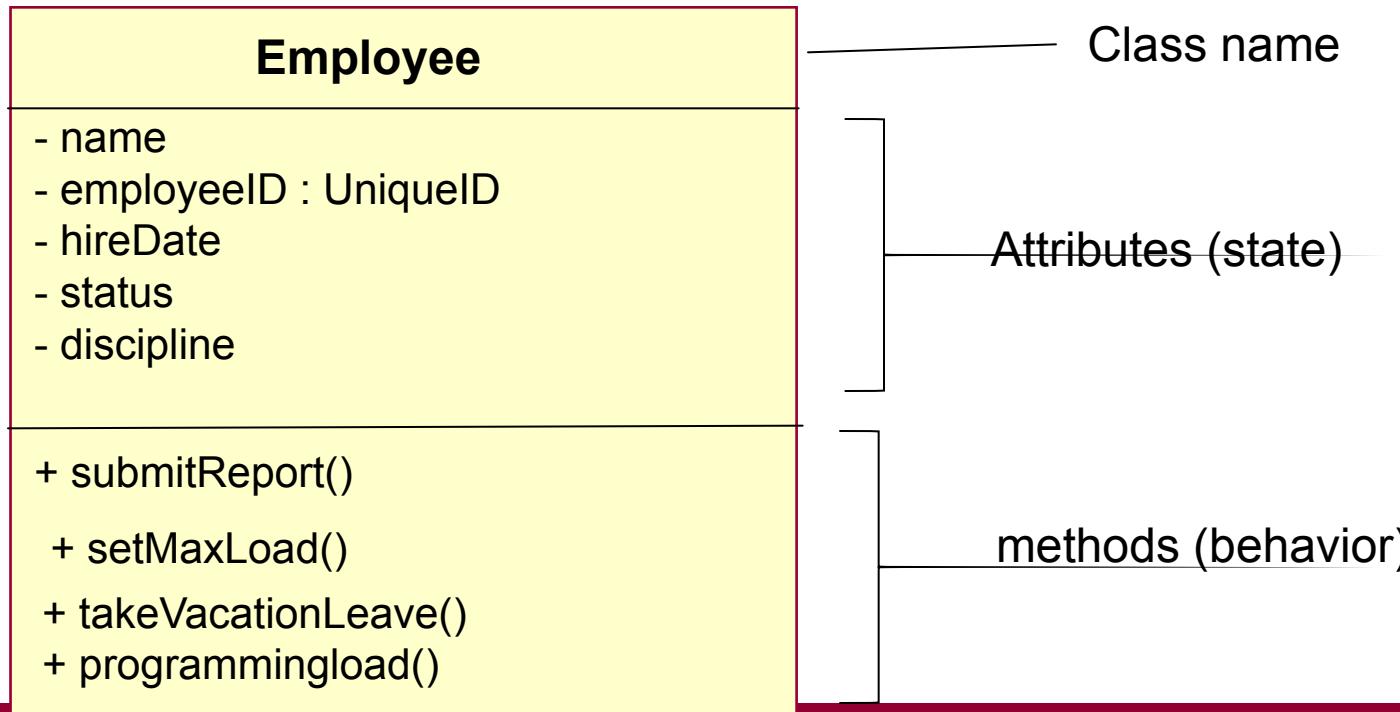


Unified Modeling Language Diagrams



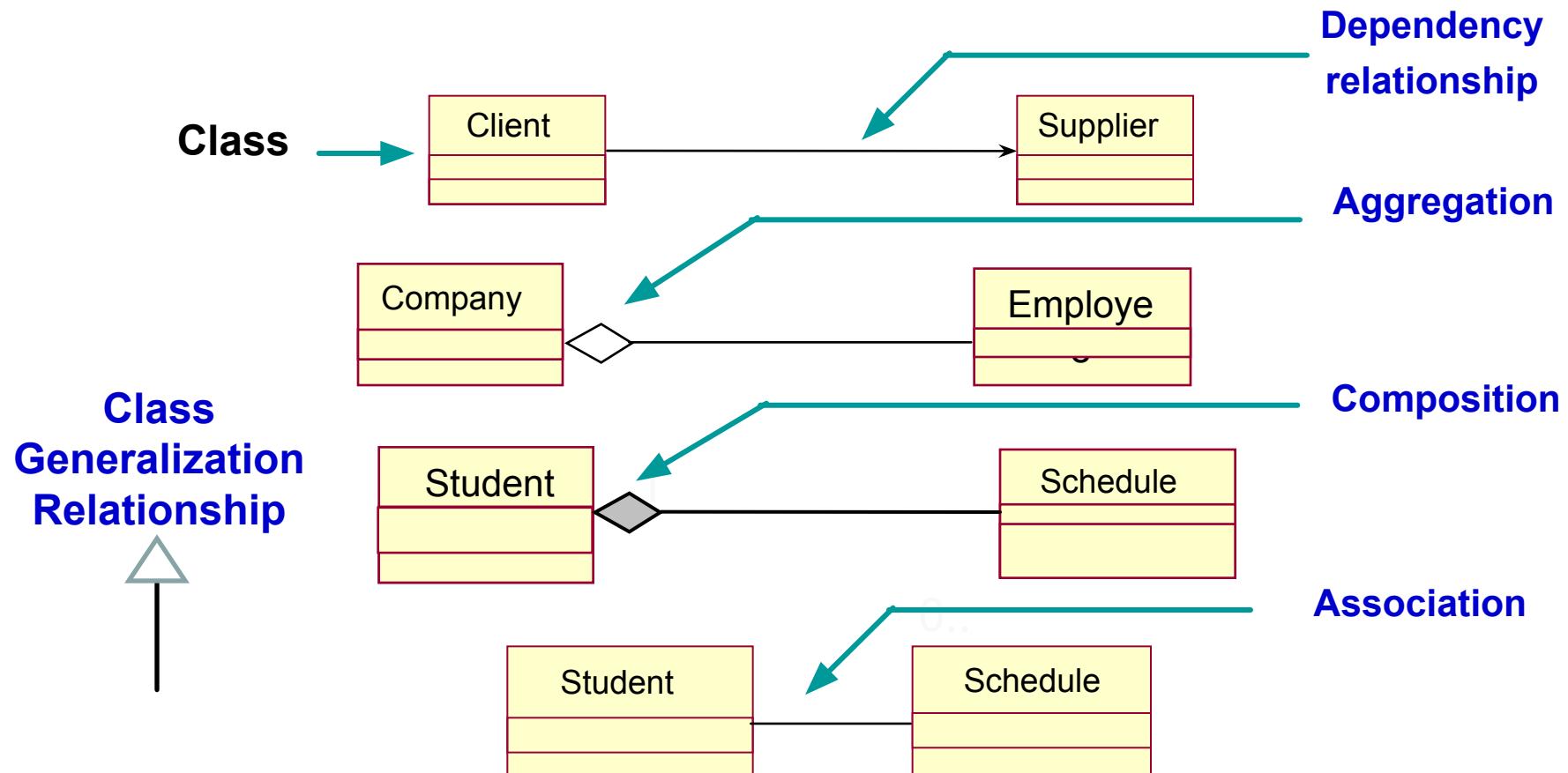
UML : Class

- A class is an UML construct used to detail the pattern from which objects will be produced at run-time.
- An object is an instance of a class.
- It expresses both the persistent state of the system and the behavior of the system.



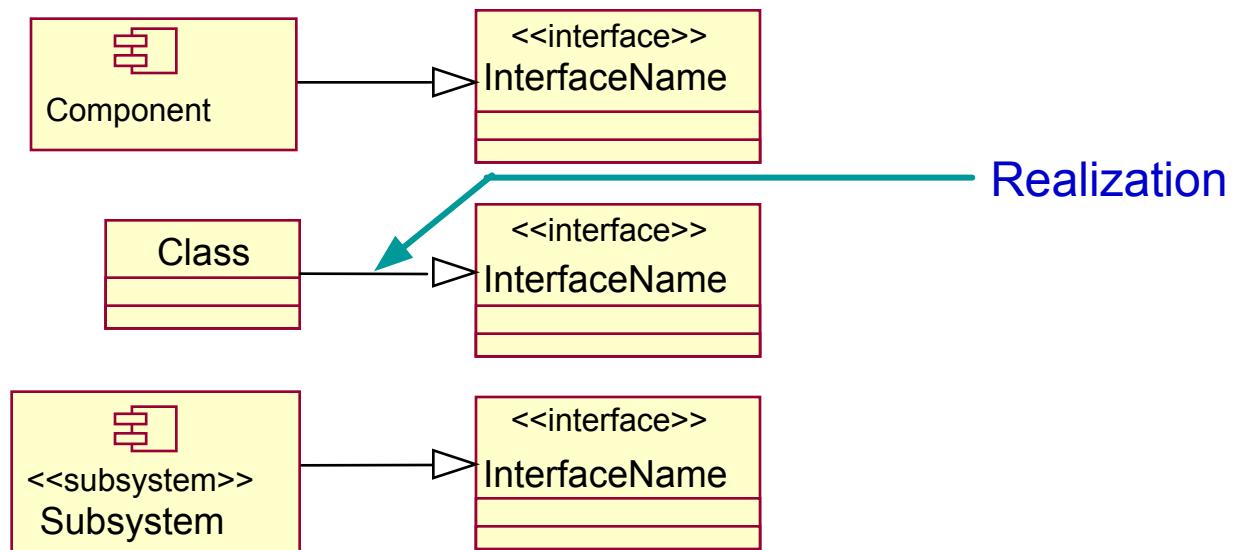
UML : Class Relationships

A RELATIONSHIP is what a class or an object knows about another class or object.



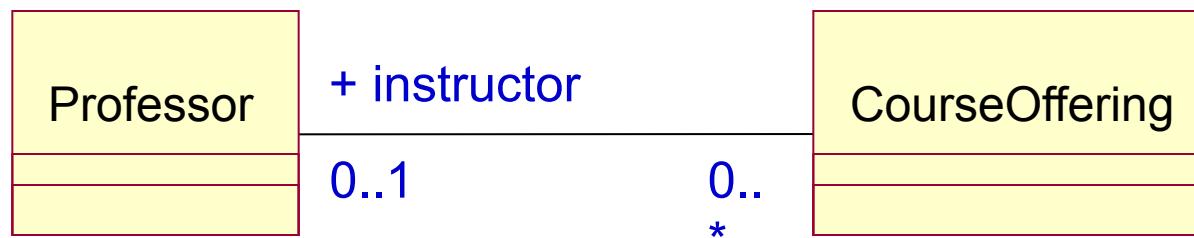
Relationships: Realization

- One classifier serves as the contract that the other classifier agrees to carry out, found between:
 - Interfaces and the classifiers that realize them



What Is Multiplicity?

- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
 - For each instance of Professor, many Course Offerings may be taught.
 - For each instance of Course Offering, there may be either one or zero Professor as the instructor.

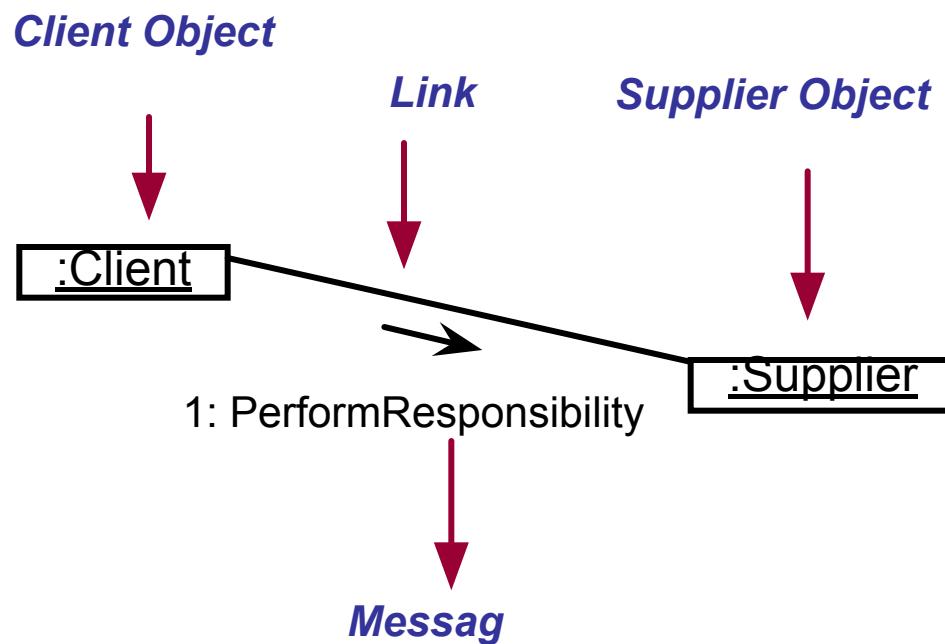


Multiplicity Indicators

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional scalar role)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

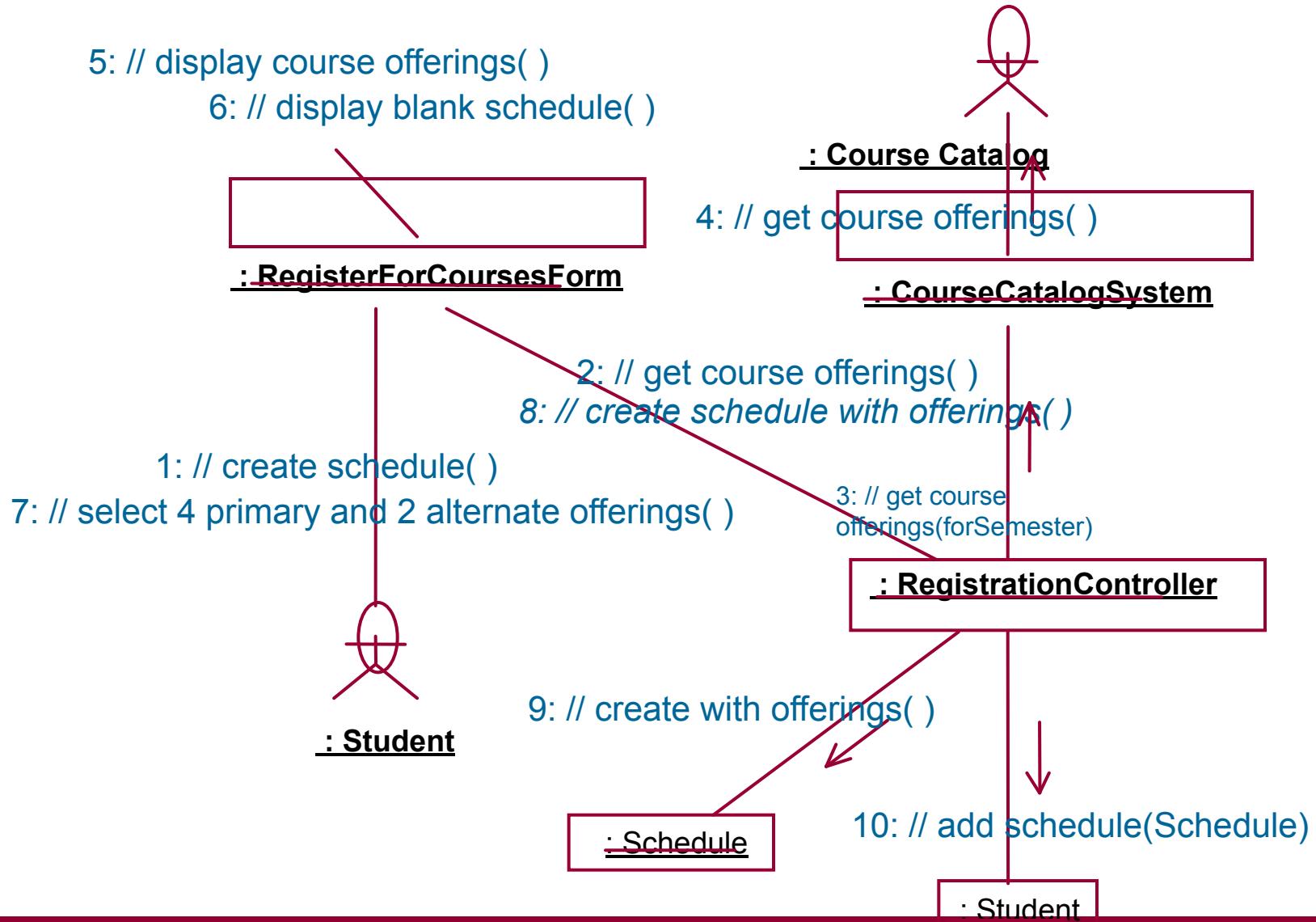
UML : Communication Diagram

- A communication diagram, formerly called a collaboration diagram, is an interaction diagram that shows similar information to sequence diagrams but its primary focus is on object relationships.



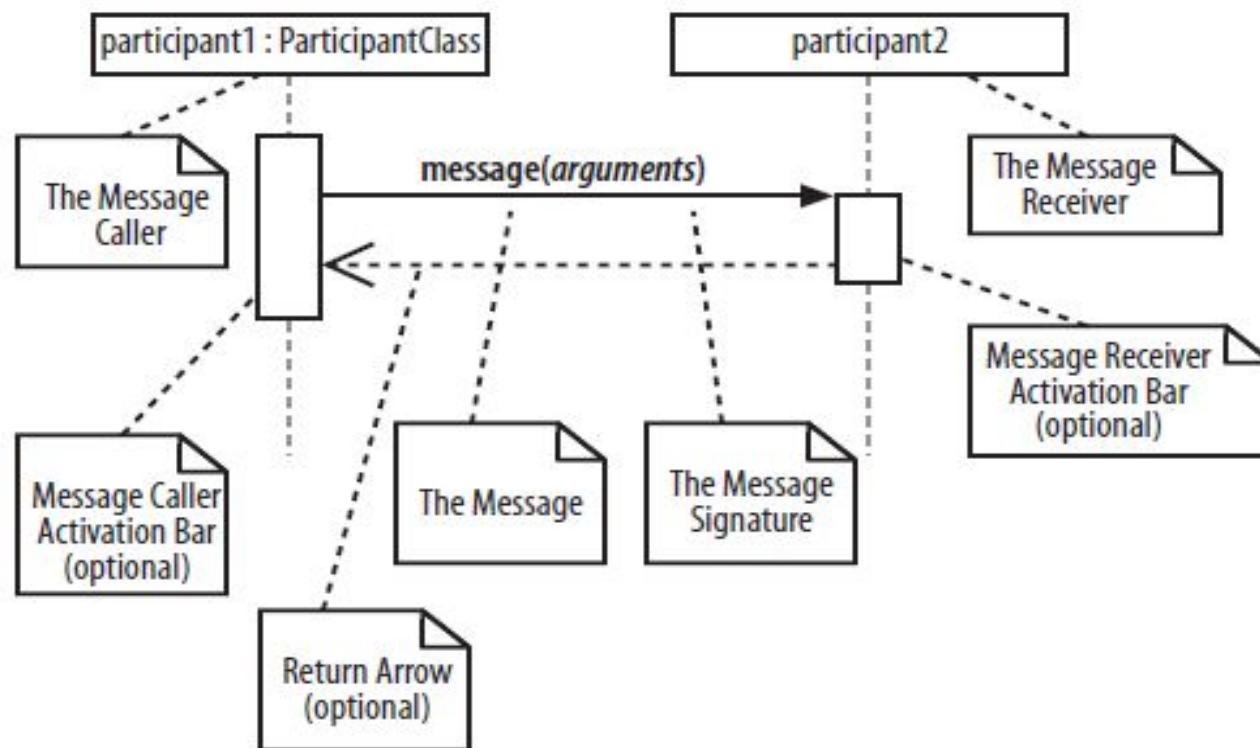
Anatomy of a Communication Diagram

Communication Diagram Example



UML : Sequence Diagram

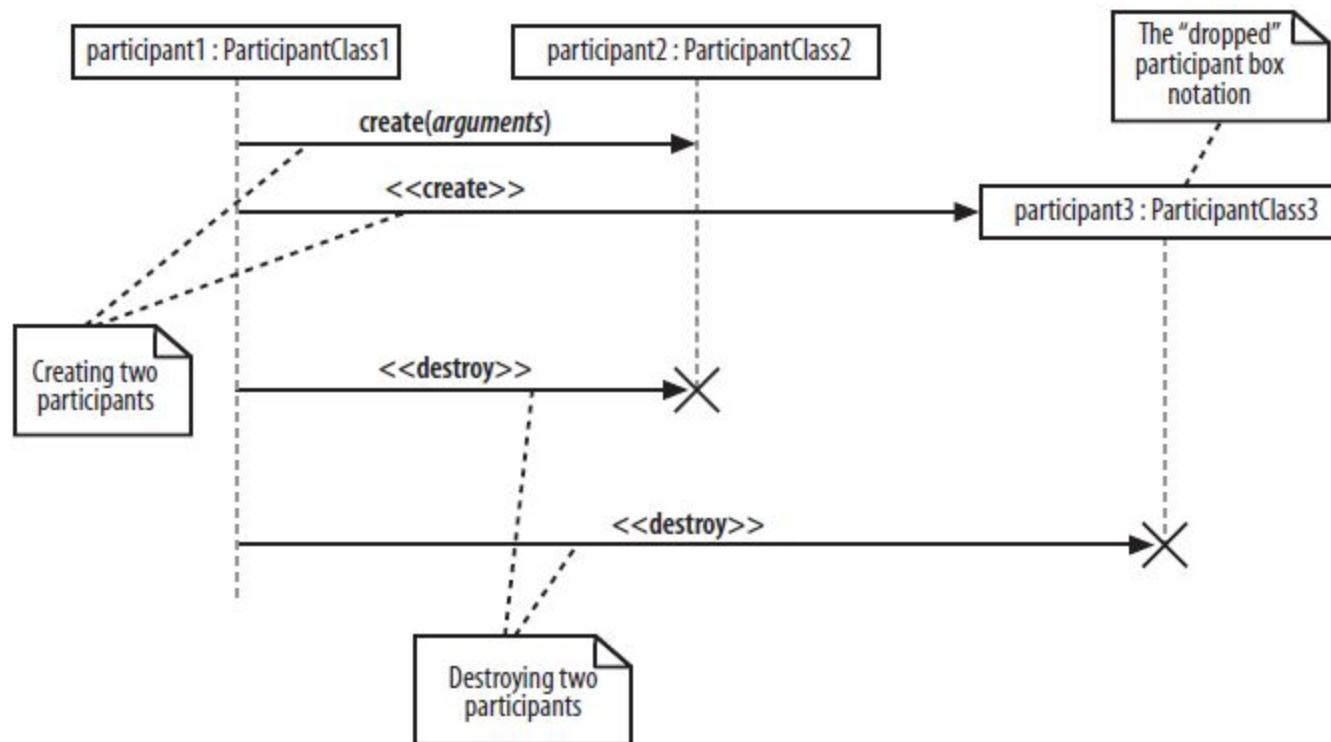
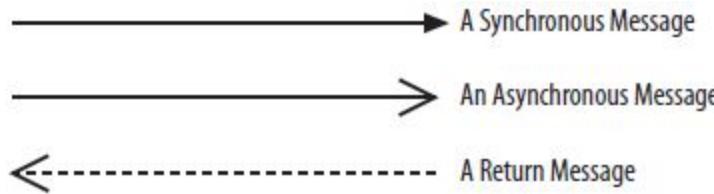
- A sequence diagram is another type of interaction diagram that emphasize on time-based ordering of the activity that takes place among a set of objects in a use case.



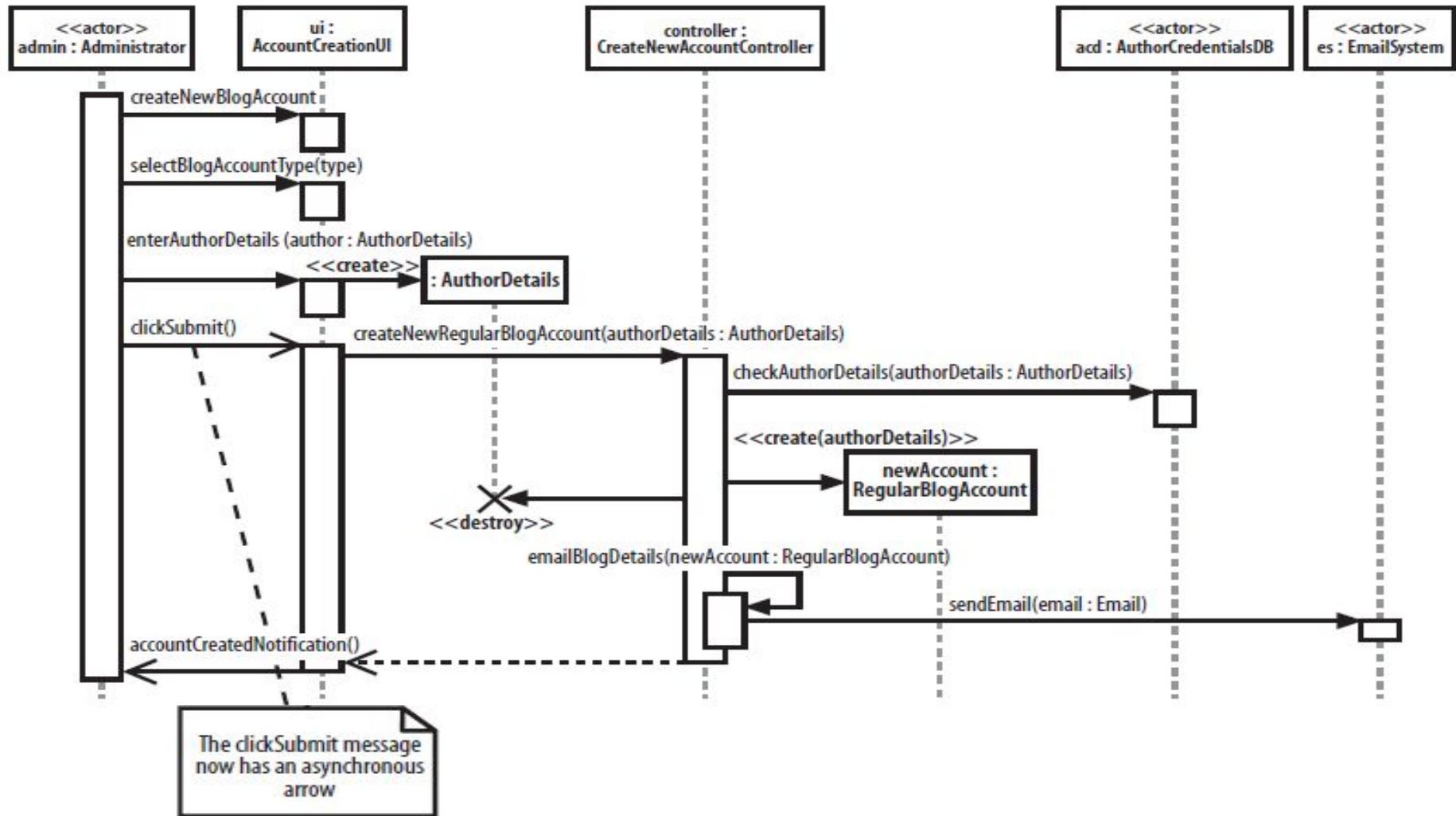
Anatomy of a Sequence Diagram

Sequence Diagram

■ Message Arrows:



Sequence Diagram Example

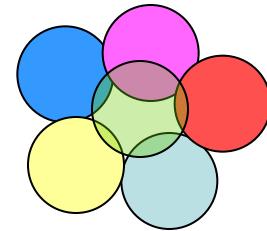


Class Design Practices and Techniques

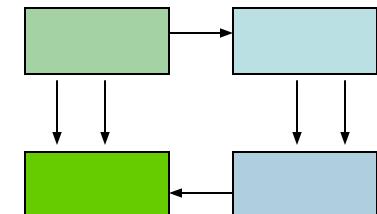


Cohesion and Coupling

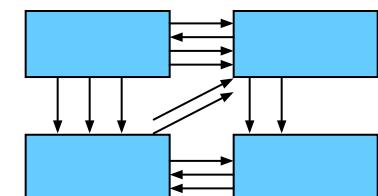
- Cohesion measures how strongly related and focused the responsibilities of a class are.
- Coupling measures how strongly one class is connected to, or relies on other objects.
- Goal is low coupling and high cohesion.



High cohesion



Low coupling



High coupling

S.O.L.I.D Principles of Object-Oriented Class Design



Single Responsibility Principle



Open/Closed Principle



Liskov Substitution Principle



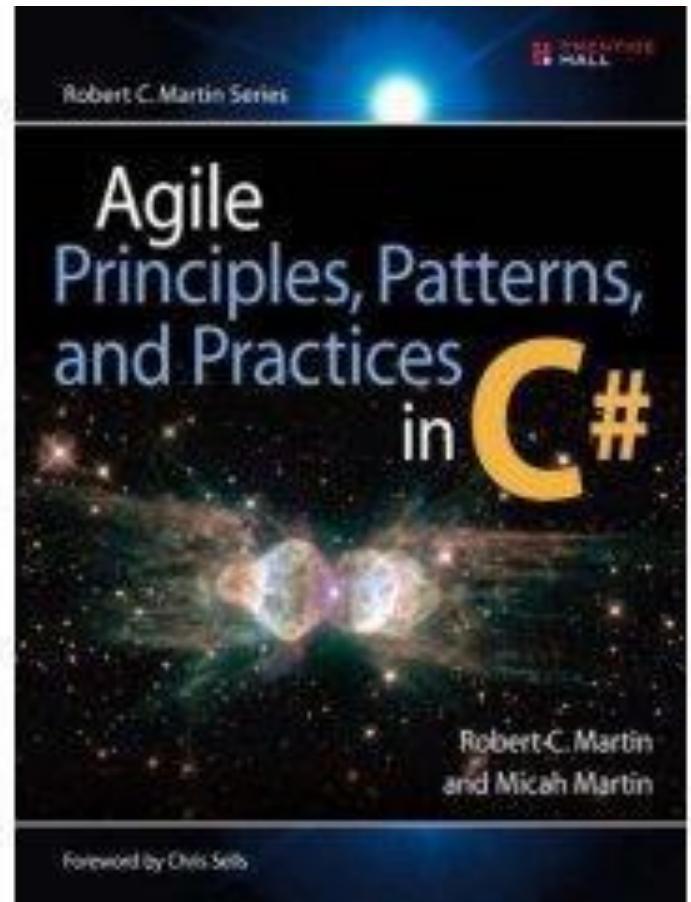
Interface Segregation Principle



Dependency Inversion Principle

Where Does SOLID Come From?

- SOLID was introduced by Robert C. Martin in his article Design Principles and Design Patterns c2000.
- In 2006, Robert C. Martin and Micah Martin wrote the book Agile Principles, Patterns and Practices in C#



SRP : Single Responsibility Principle

“A class should have only one reason to change.”



Each responsibility should be a separate class.

Robots on an assembly line are streamlined for the individual tasks they perform. This makes maintaining, upgrading, and replacing them easier and less expensive.

OCP: Open/Close Principle

“Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.” - *Bertrand Meyer*



Objects should be designed in such a way that it is easy to extend them without modifying them.

- Meyer's Open/Closed Principle
- Polymorphic Open/Closed Principle

LSP: Liskov Substitution Principle

- All derived classes must be substitutable for their base class. (Barbara Liskov)

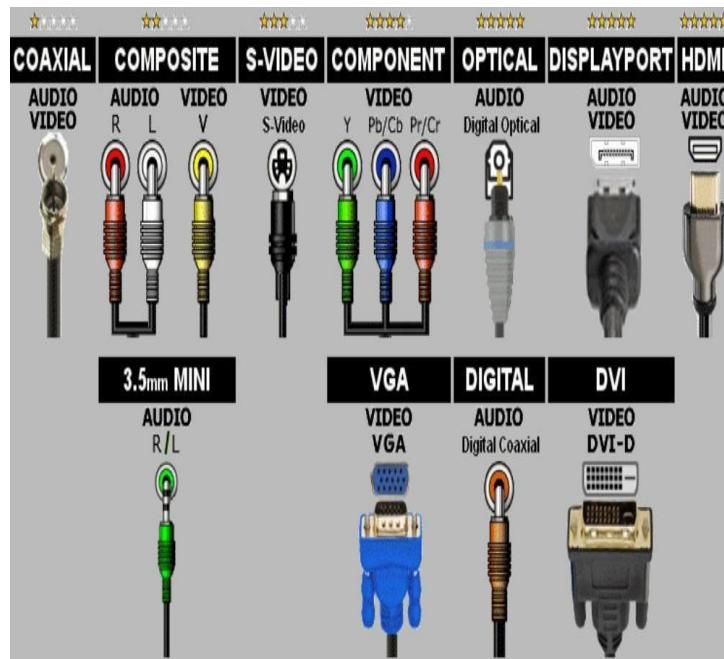


“Subtypes must be substitutable for their base types.”

As you extend objects, the original functionality of the elements that makeup the object should not change.

ISP: Interface Segregation Principle

“Clients should not be forced to depend on methods they do not use.”



- “Many client specific interfaces are better than one general purpose interface”
- Each interface should specifically describe only what is needed and nothing more.

DIP: Dependency Inversion Principle

- “High-level modules should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend on details. Details should depend upon abstractions”



Checkpoint

- What are the similarities and differences between a class and object?



Checkpoint

Checkpoint

- What are the four basic principles of object orientation? Provide a brief description of each.



Checkpoint

Checkpoint

- What is a class relationship?

Give some examples.

- What are the two types of Interaction Diagram?



Checkpoint

Checkpoint

- What are the 5 Class design principles?
- What is the acronym of this class design principles?



Checkpoint

Module 2: Introduction to .NET 4.0 Framework



Module Agenda

.NET 4.0 Framework

Framework Class Library

Common Language
Runtime



Module Objectives

Upon completion of this module,
you should be able to:

- Define the components of .NET Framework
- Define an assembly
- Explain the .NET compilation and execution
- Describe the tools that .NET Framework provides



Module objectives

.NET 4.0 Framework



What is the .NET Framework 4?

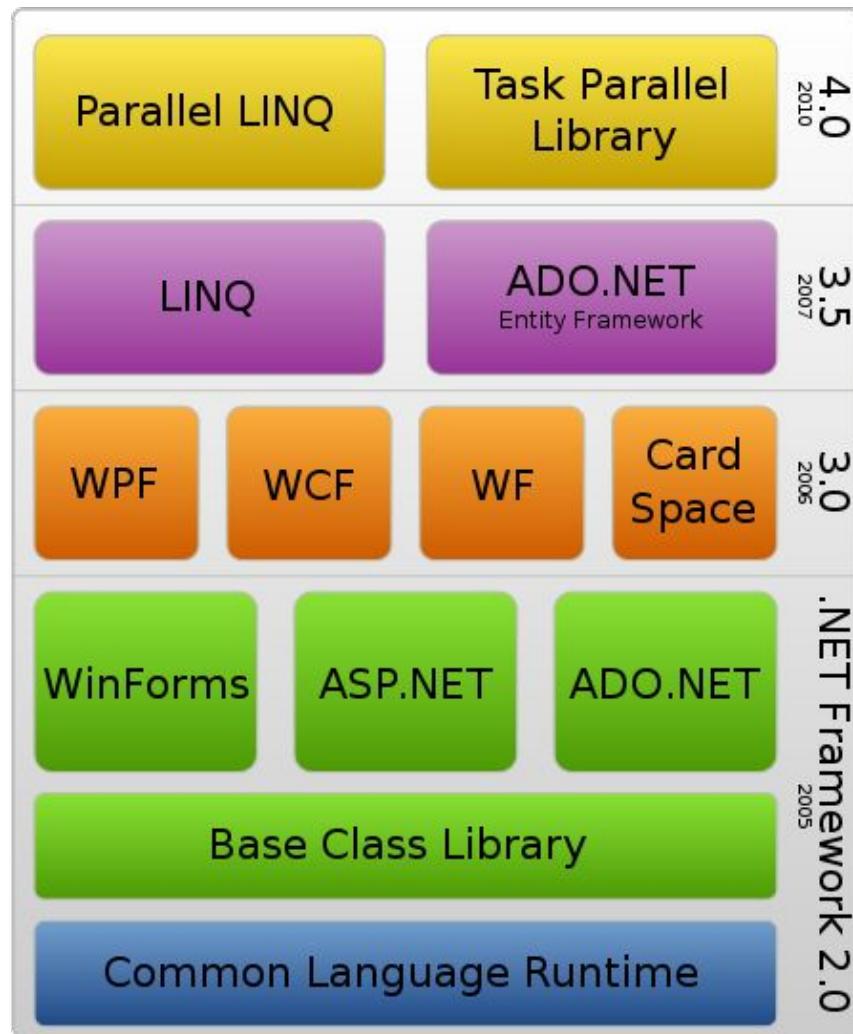


Common Language Runtime

Class Library

Development Framework

.NET Framework Evolution

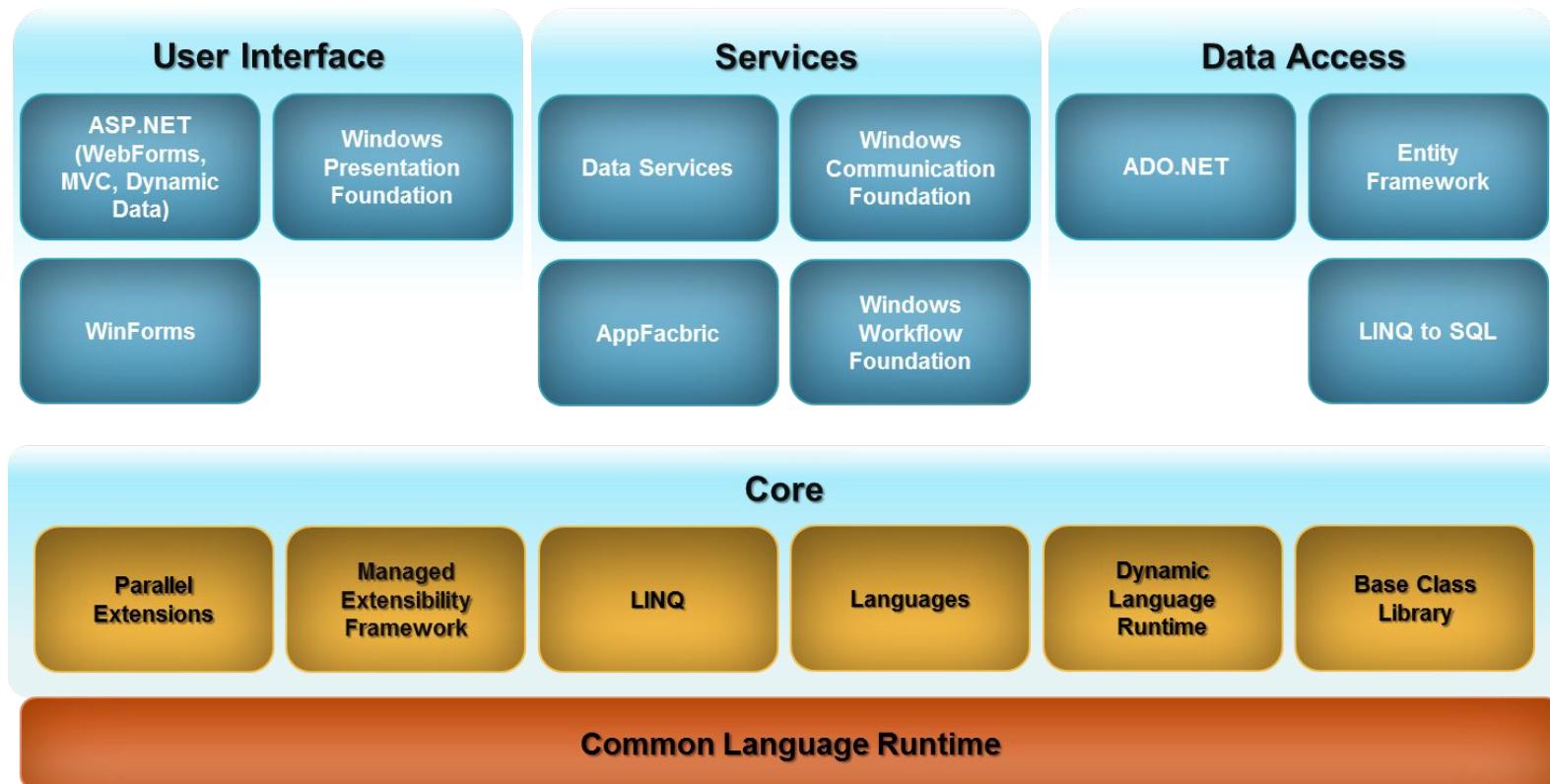


.NET 4.0 Framework Features

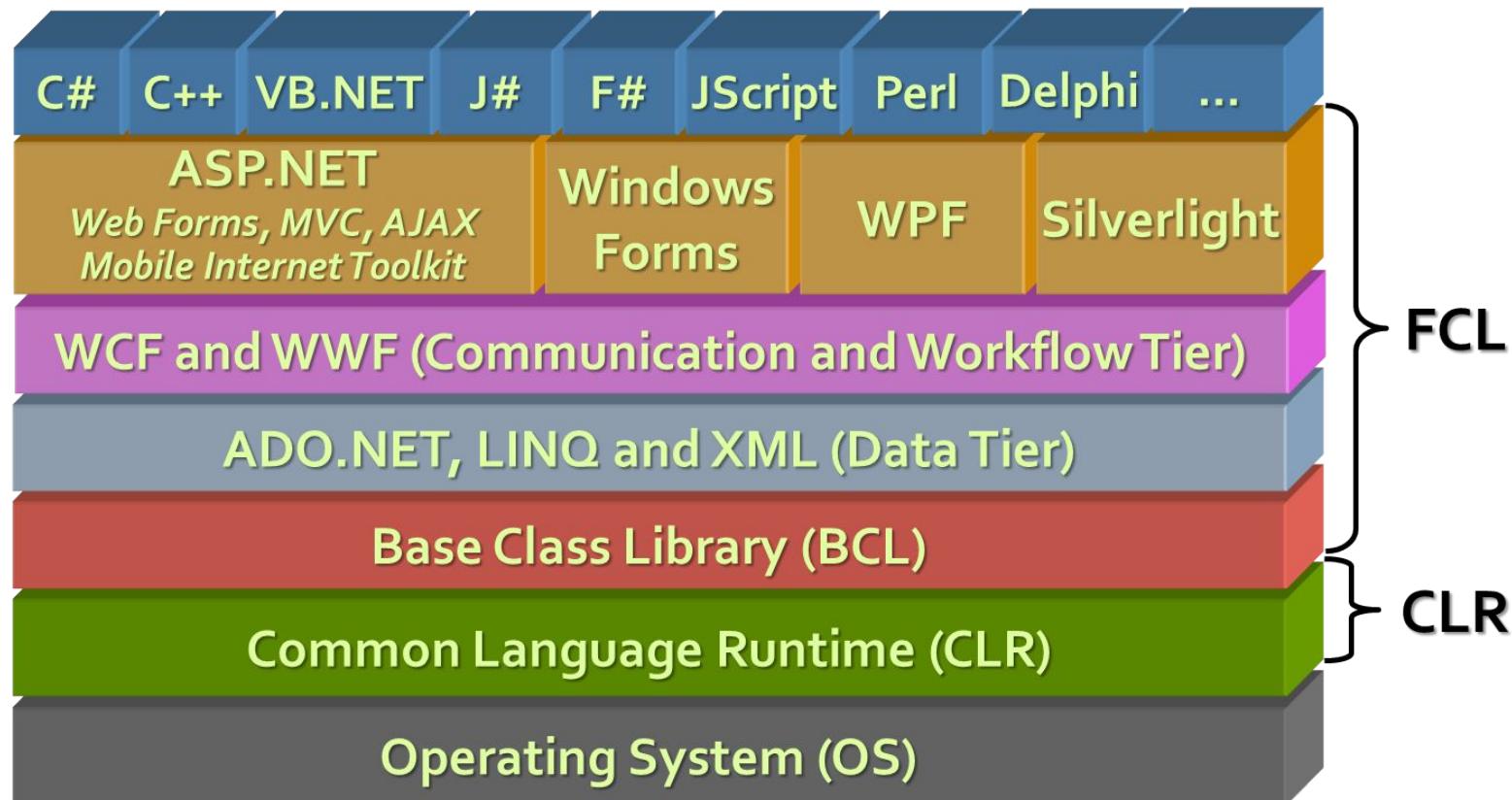
- Application Compatibility and Deployment
- Core New Features and Improvements
- Managed Extensibility Framework
- Parallel Computing
- Networking
- Web
- Client
- Data
- Windows Communication Foundation
- Windows Workflow Foundation



Microsoft .NET Framework 4.0



Components of .NET Framework



Framework Class Library (FCL)

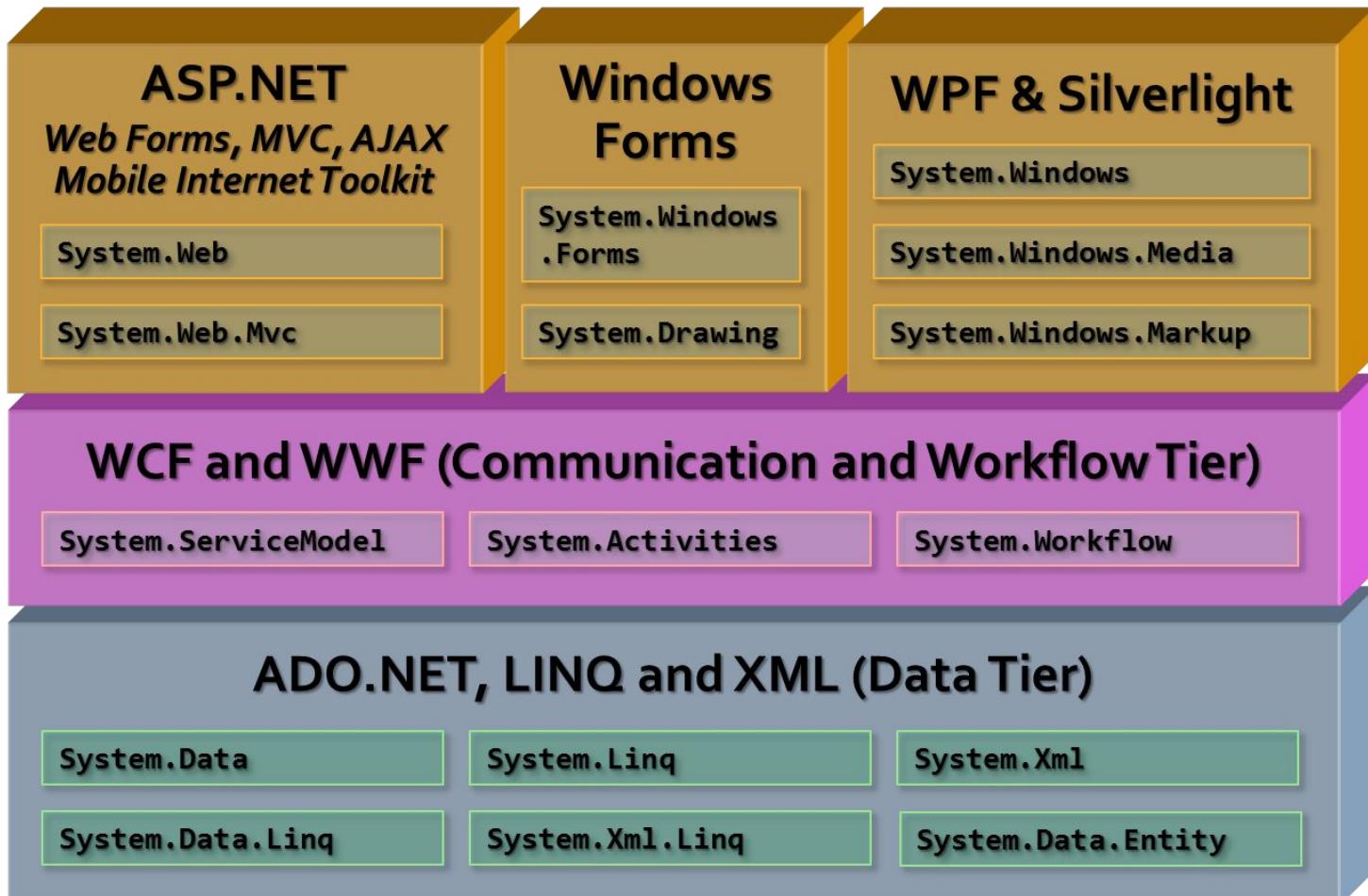


Framework Class Library

- Provides basic functionality to developers:
 - Console applications
 - WPF and Silverlight rich-media applications
 - Windows Forms GUI applications
 - Web applications – Dynamic Web Sites
 - Web Services, Communication and Workflow
 - Server and Desktop applications
 - Applications for mobile devices



FCL Namespaces

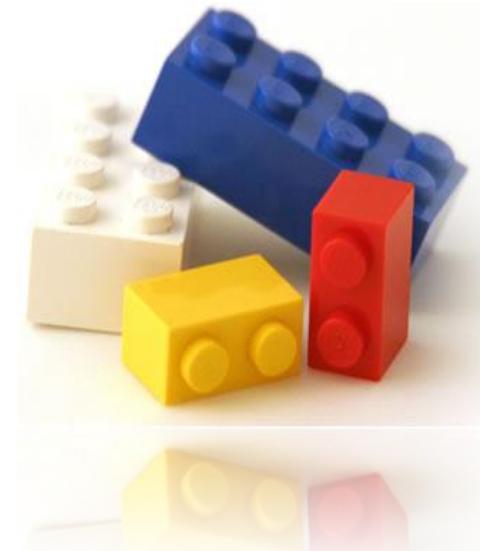


Common Language Runtime (CLR)



Common Language Runtime

- Managed execution environment
 - Executes .NET applications
 - Controls the execution process
- Automatic memory management – Garbage Collection
- Programming languages integration
- Multiple versions support for assemblies
- Integrated type safety and security

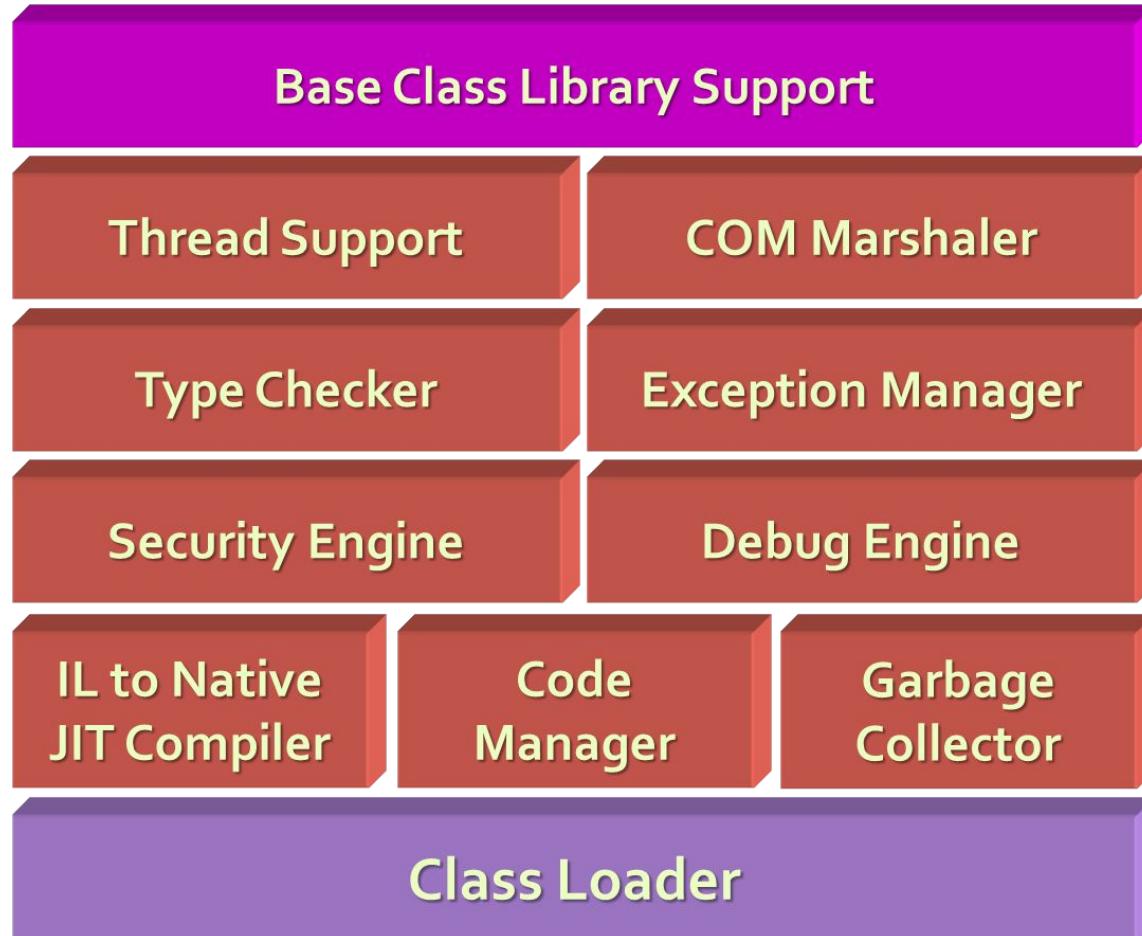


Responsibilities of CLR

- Execution of the IL code and the JIT compilation
- Managing memory and application resources
- Ensuring type safety
- Interaction with the OS
- Managing Security
 - Code access security
 - Role-based security



CLR Architecture



Memory Management

- CLR manages memory automatically
 - Dynamically loaded objects are stored in the managed heap
 - Unusable objects are automatically cleaned up by the Garbage Collector
- Some of the big problems are solved
 - Memory leaks
 - Access to freed or unallocated memory
- Objects are accessed through a reference



Garbage Collection (GC)

- A process of *destroying object and reclaiming memory* occupied by unused objects back to the heap automatically done by .NET Framework.
- **Automatic memory management policies** are implemented by a *garbage collector*.
- **System.GC.Collect();** - method that enforces the Garbage Collector to start.
- **Important:** The object won't be destroyed until the garbage collector is triggered to go looking for unused objects.



Finalizers vs Destructors

- Finalizer - refers to a non-deterministic clean-up. A finalizer is executed when the internal garbage collection system frees the object.
- Destructor – refers to a deterministic clean-up. A destructor is run when the program explicitly frees an object.
- C# implements finalization using destructors, i.e., finalize is automatically called when destructor is compiled.
- C# objects are not allowed to override the Finalize method and must implement a destructor instead.

IDisposable

- To avoid some uncertainty of finalization, use Disposable.
- It requires a class that must implement System.IDisposable interface.
- Example:

```
class MyClass : IDisposable {
    public MyClass() {
        // constructor statements
        Console.WriteLine("Constructor called");
    }
    public void DoSomeWork() {
        Console.WriteLine("Doing some work...");
    }
    public void Dispose() {
        // disposal statements
        Console.WriteLine("Dispose method called");
    }
}
```

Using Statement

- The using statement provides a clean mechanism for controlling the lifetime of resources.
- You can create an object, and this object will be destroyed when the using statement block finishes.
- Example:

```
public void DoSomeWork() {  
    Console.WriteLine("Doing some work...");  
    using (TextReader reader = new StreamReader(filename)) {  
        string line;  
        while ((line = reader.ReadLine()) != null) {  
            Console.WriteLine(line);  
        }  
    }  
}
```

Managed Code

- CLR executed code is called managed code
- Represents programming code in the low level language MSIL (MS Intermediate Language)
- Contains Metadata
- Applications written in any .NET language are
 - Compiled to managed code (MSIL)
 - Packaged as assemblies (.exe or .dll files)



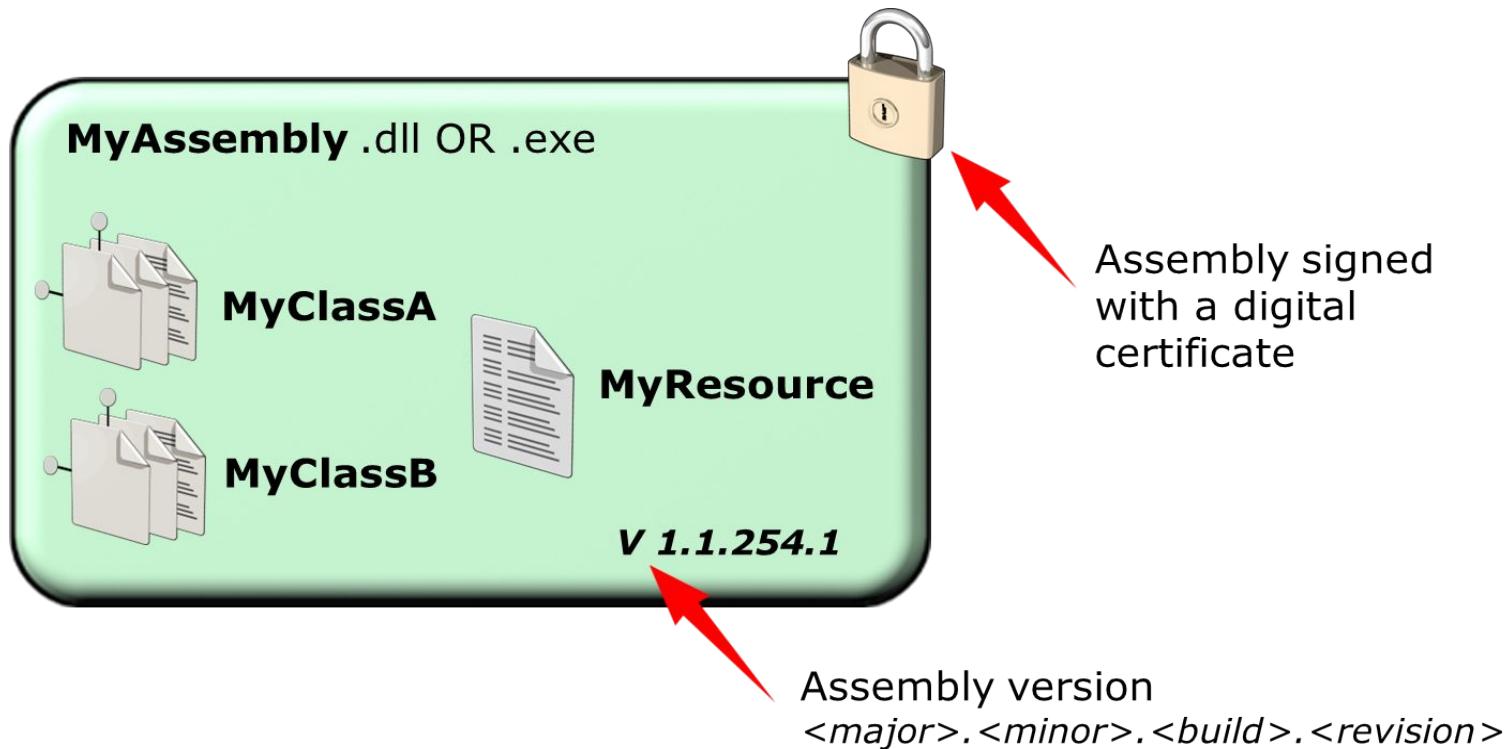
Unmanaged Code (Win32)

- No protection of memory and type-safety
 - Reliability problems
 - Safety problems
- Doesn't contain metadata
 - Needs additional overhead like (e.g. use COM,
- Compiled to machine-dependent code
 - Need of different versions for different platforms
 - Hard to be ported to other platforms



What is an Assembly?

- Building blocks of .NET Framework applications
- Collection of types and resources that form a logical unit of functionality



Intermediate Language (MSIL, IL, CIL)

- Lower level language (machine language) for the .NET CLR
- Has independent set of CPU instructions
 - Loading and storing data, calling methods
 - Arithmetic and logical operations
 - Exception handling
 - Etc.
- MSIL is converted to instructions for the current physical CPU by the JIT compiler

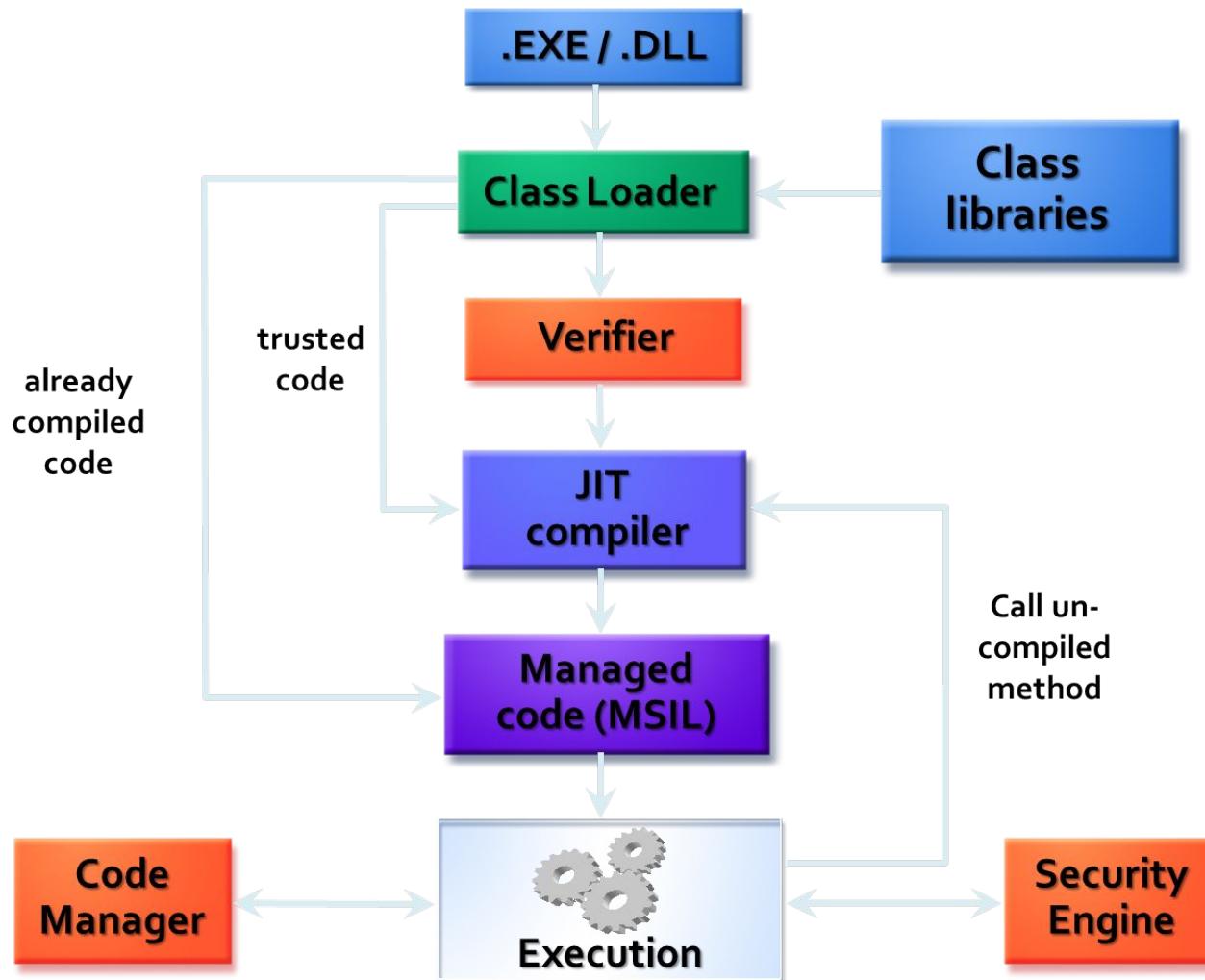


Sample MSIL Program

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size          11 (0xb)
    .maxstack  8
    ldstr      "Hello, world!"
    call       void
        [mscorlib]System.Console::WriteLine(string)
    ret
} // end of method HelloWorld::Main
```



How CLR executes MSIL?

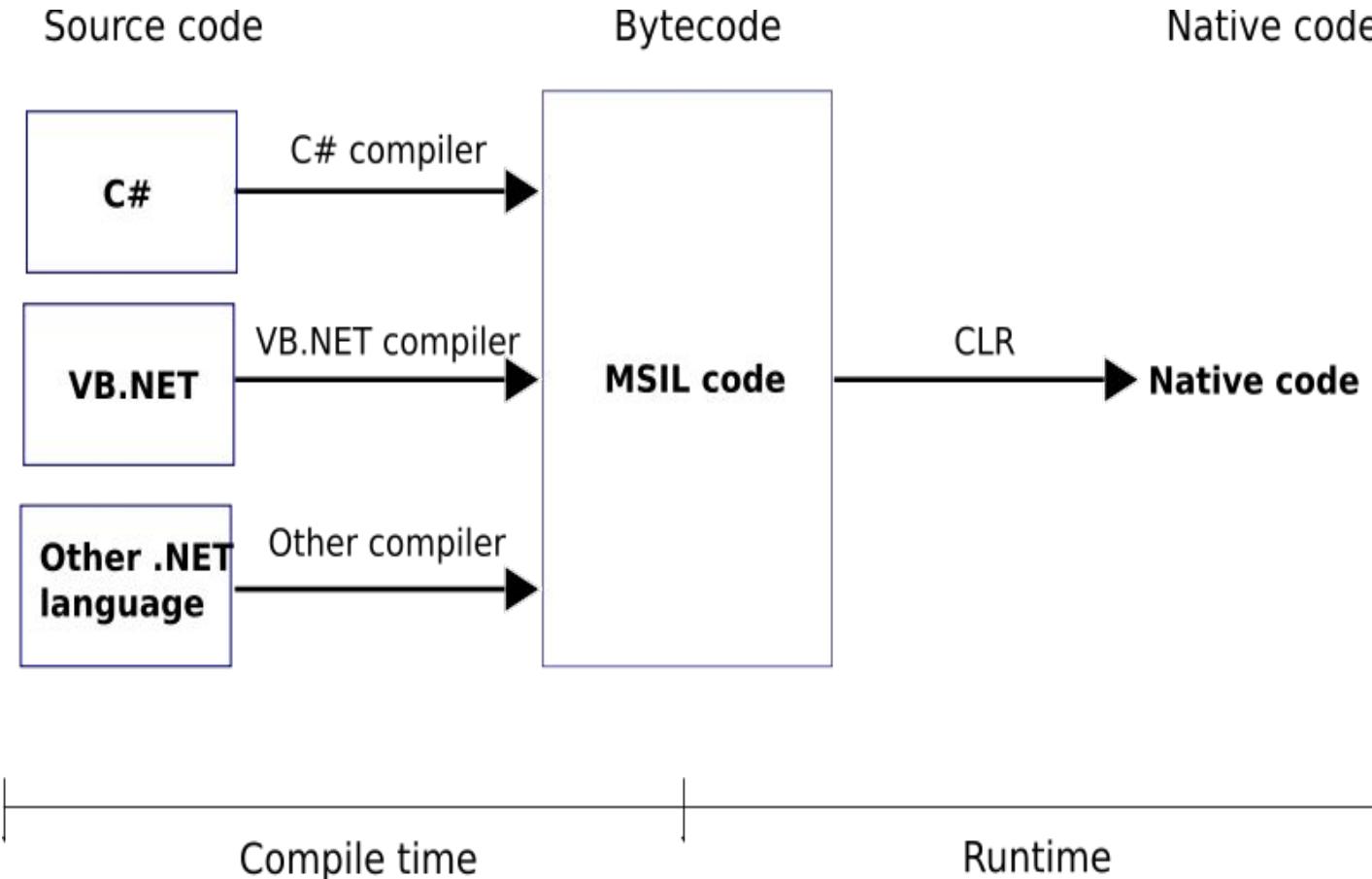


How .NET Supports Multiple Languages?

- Common Language Infrastructure (CLI)
 - Open specification developed by Microsoft (ECMA-335)
 - Multiple high-level languages run on different platforms without changes in the source code or pre-compilation
 - Standardized part of CLR
 - .NET Framework is CLI implementation for Windows
 - Mono is CLI implementation for Linux

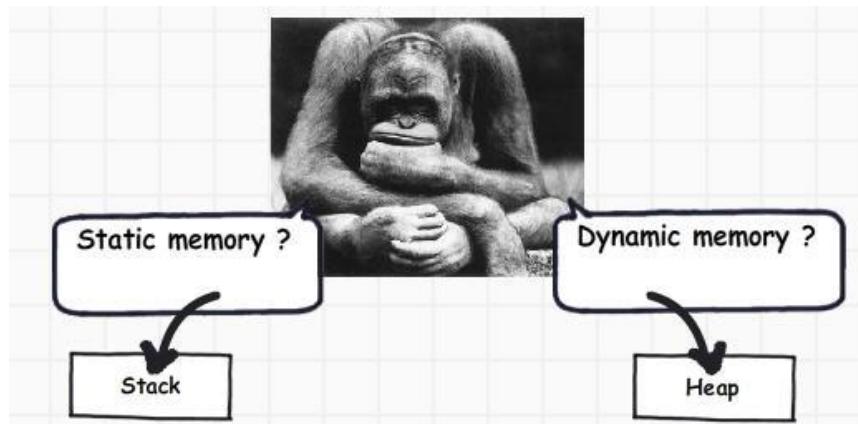


.NET Code Compilation and Execution

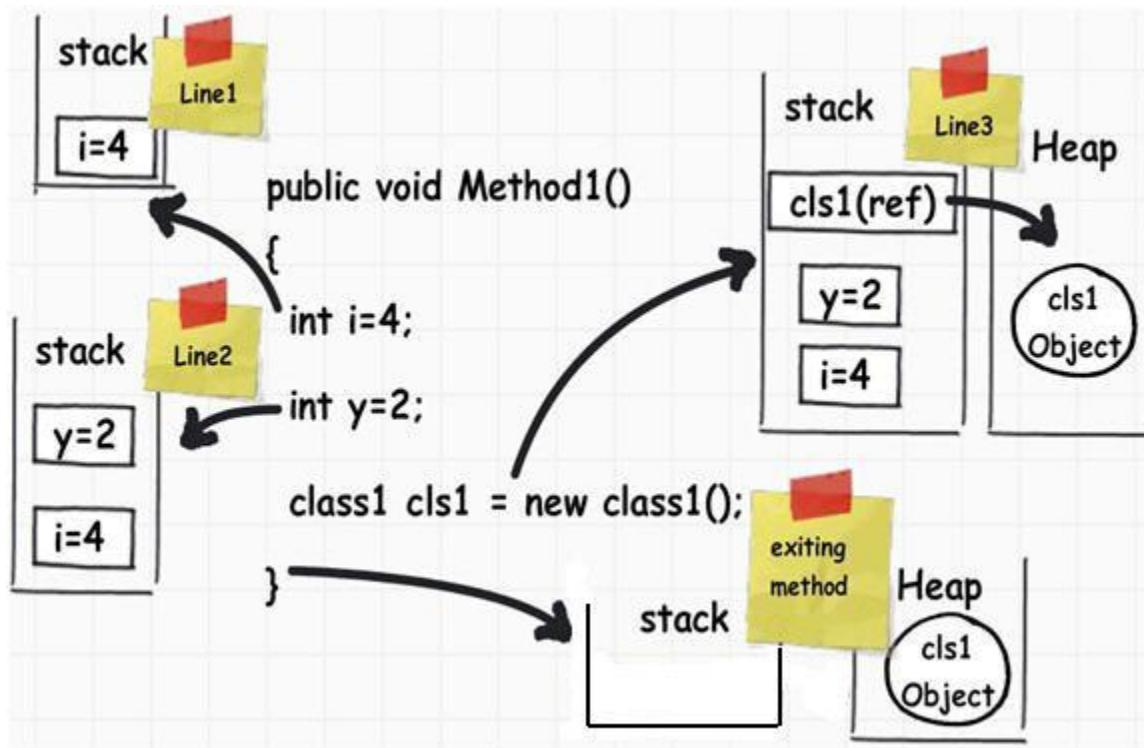


Stack, Heap and Large Object Heap (LOH)

- Stack – value types are stored
- Heap – reference types are stored
- Large Object Heap – used for objects > 85K bytes



Stack, Heap and Large Object Heap (LOH)



.NET Framework Tools

- Caspol.exe
- Gacutil.exe
- Ildasm.exe
- Makecert.exe
- Ngen.exe
- Sn.exe



Checkpoint

- A component of .NET Framework responsible for managing memory and application resources?
 - a. FCL
 - b. MSIL
 - c. CLR
 - d. Garbage Collector

- A collection of types and resources that form a logical unit of functionality.
 - a. FCL
 - b. Assembly
 - c. Unmanaged Code



Checkpoint

Checkpoint

- What is the correct sequence of a .NET compilation and execution?
 - a. Native Code MSIL Code Source Code
 - b. Source code MSIL code Native Code
 - c. MSIL Code native Code Source Code
 - d. MSIL Code Source Code Native Code
- A .NET framework tool that enables users to manipulate the assemblies in the Global Assembly Cache
 - a. GACUTIL
 - b. CASPOL
 - c. ILDASM



Checkpoint

Knowledge Check



Knowledge Check

1. Which of the following statements are TRUE about the .NET CLR?
 1. It provides a language-neutral development & execution environment.
 2. It ensures that an application would not be able to access memory that it is not authorized to access.
 3. It provides services to run "managed" applications.
 4. The resources are garbage collected.
 5. It provides services to run "unmanaged" applications.

- b. Only 1 and 5
- c. Only 1, 2 and 4
- d. 1, 2, 3, 4
- e. Only 4 and 5

Knowledge Check

2. Which of the following statements is correct about Managed Code?
 - a. Managed code is the code that is compiled by the JIT compilers.
 - b. Managed code is the code where resources are Garbage Collected.
 - c. Managed code is the code that runs on top of Windows.
 - d. Managed code is the code that is written to target the services of the CLR.
3. Which of the following utilities can be used to compile managed assemblies into processor-specific native code?
 - a. Gacutil
 - b. Ngen
 - c. Sn
 - d. Ilasm

Knowledge Check

4. Which of the following components of the .NET framework provide an extensible set of classes that can be used by any .NET compliant programming language?
 - a. NET class libraries
 - b. Common Language Runtime
 - c. Common Language Infrastructure
 - d. Common Type System
5. Which of the following assemblies can be stored in Global Assembly Cache?
 - a. Private Assemblies
 - b. Shared Assemblies
 - c. Public Assemblies
 - d. Protected Assemblies

Knowledge Check

6. Which of the following statements correctly define .NET Framework?
 - a. It is an environment for developing, building, deploying and executing Desktop Applications, Web Applications and Web Services.
 - b. It is an environment for developing, building, deploying and executing only Web Applications.
 - c. It is an environment for developing, building, deploying and executing Distributed Applications.
 - d. It is an environment for developing, building, deploying and executing Web Services.

Knowledge Check

7. Which of the following constitutes the .NET Framework?
1. ASP.NET Applications
 2. CLR
 3. Framework Class Library
 4. WinForm Applications
 5. Windows Services
- b. 1, 2
- c. 2, 3
- d. 3, 4
- e. 2, 5

Knowledge Check

8. Which of the following benefits do we get on running managed code under CLR?
 1. Type safety of the code running under CLR is assured.
 2. It is ensured that an application would not access the memory that it is not authorized to access.
 3. It launches separate process for every application running under it.
 4. The resources are Garbage collected.
 - b. Only 2, 3 and 4
 - c. Only 1, 2 and 4
 - d. Only 4
 - e. All of the above

Knowledge Check

9. Which of the following statements are correct about JIT?
1. JIT compiler compiles instructions into machine code at run time.
 2. The code compiler by the JIT compiler runs under CLR.
 3. The instructions compiled by JIT compilers are written in native code.
 4. The instructions compiled by JIT compilers are written in Intermediate Language (IL) code.
- b. 1, 2, 3
- c. 2, 4
- d. 3, 4, 5
- e. 1, 2

Knowledge Check

10. Code that targets the Common Language Runtime is known as
 - a. Unmanaged
 - b. Distributed
 - c. Managed Code
 - d. Native Code

Module 3: Working with Visual Studio .NET 2010



Module Objectives

Upon completion of this module, you should be able to:

- Describe the various project types that Visual Studio supports and when to use them
- Know the Coding and Debugging Techniques available in Visual Studio.
- Describe the features available in Visual Studio 2010 that aids in programming productivity



Module objectives

Module Agenda

Microsoft Visual Studio

Coding Techniques

Productivity Tools



Microsoft Visual Studio



Microsoft Visual Studio (VS)

- Intuitive IDE that enables developers to quickly build applications in their chosen programming language
- Development tool helps developers to:
 - Write code
 - Design user interface
 - Compile code
 - Execute, Test, Debug applications
 - Browse the help and documentation
 - Manage project's files

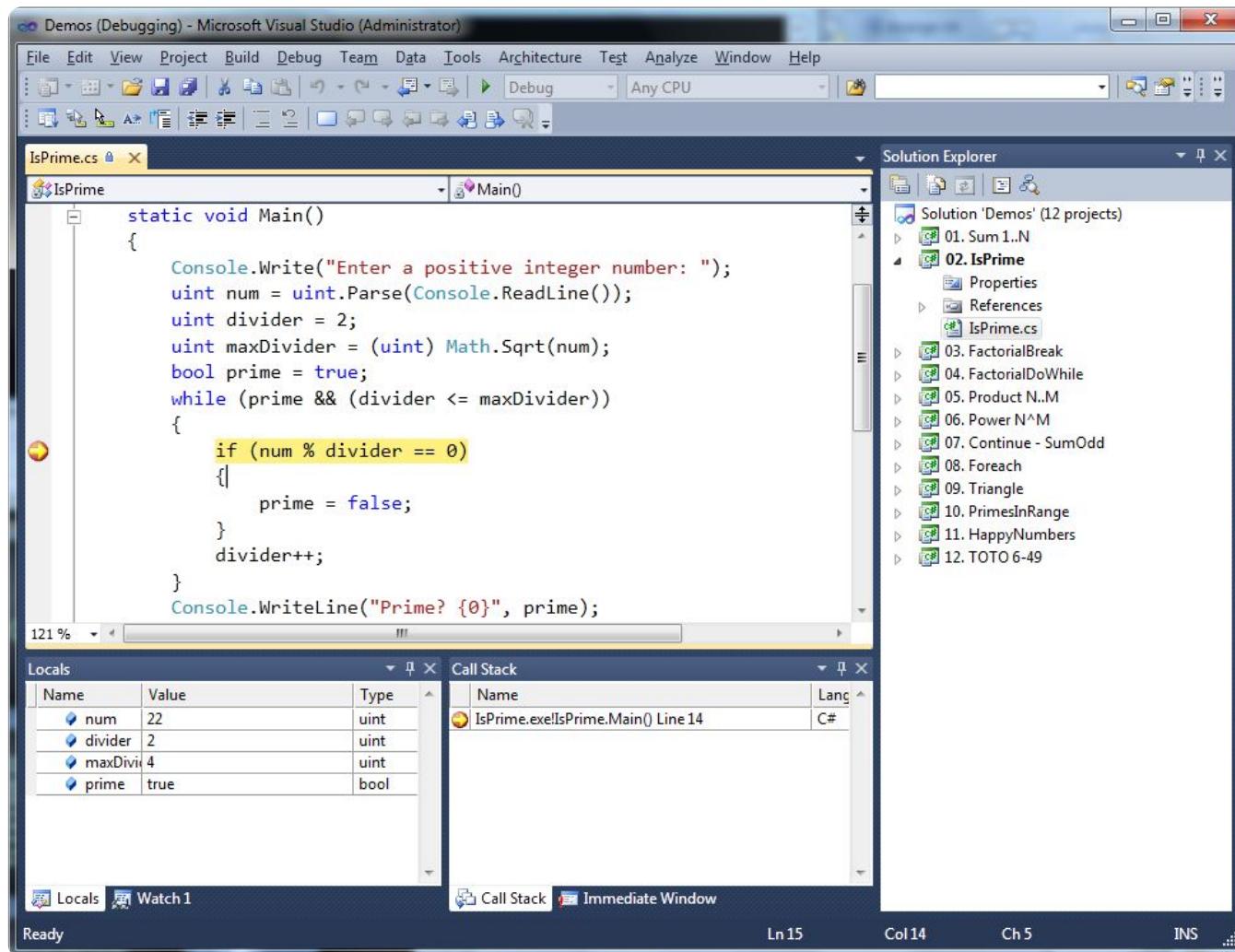


Benefits of Visual Studio

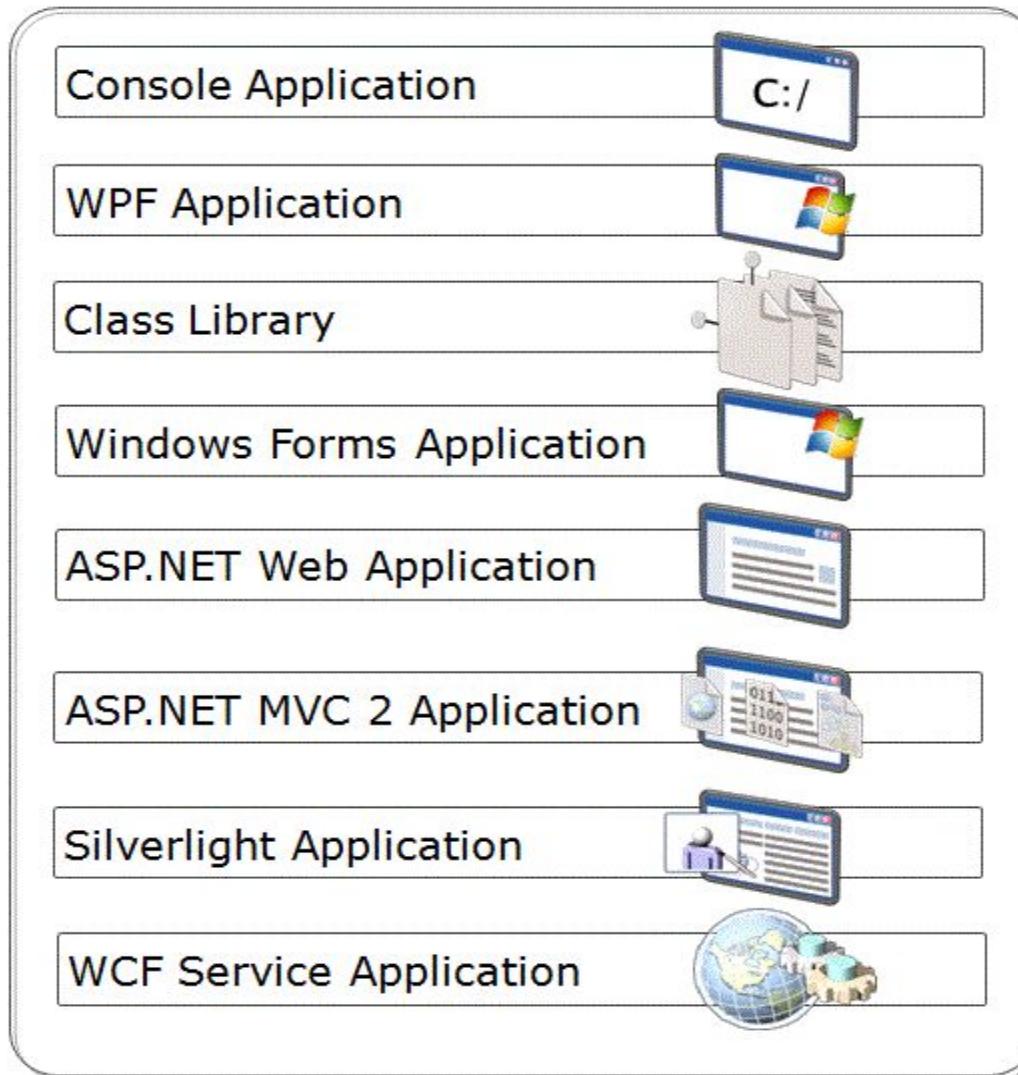
- Single tool for:
 - Writing code in many languages
 - Using different technologies
 - For different platforms
- Full integration of most development activities
- Very easy to use



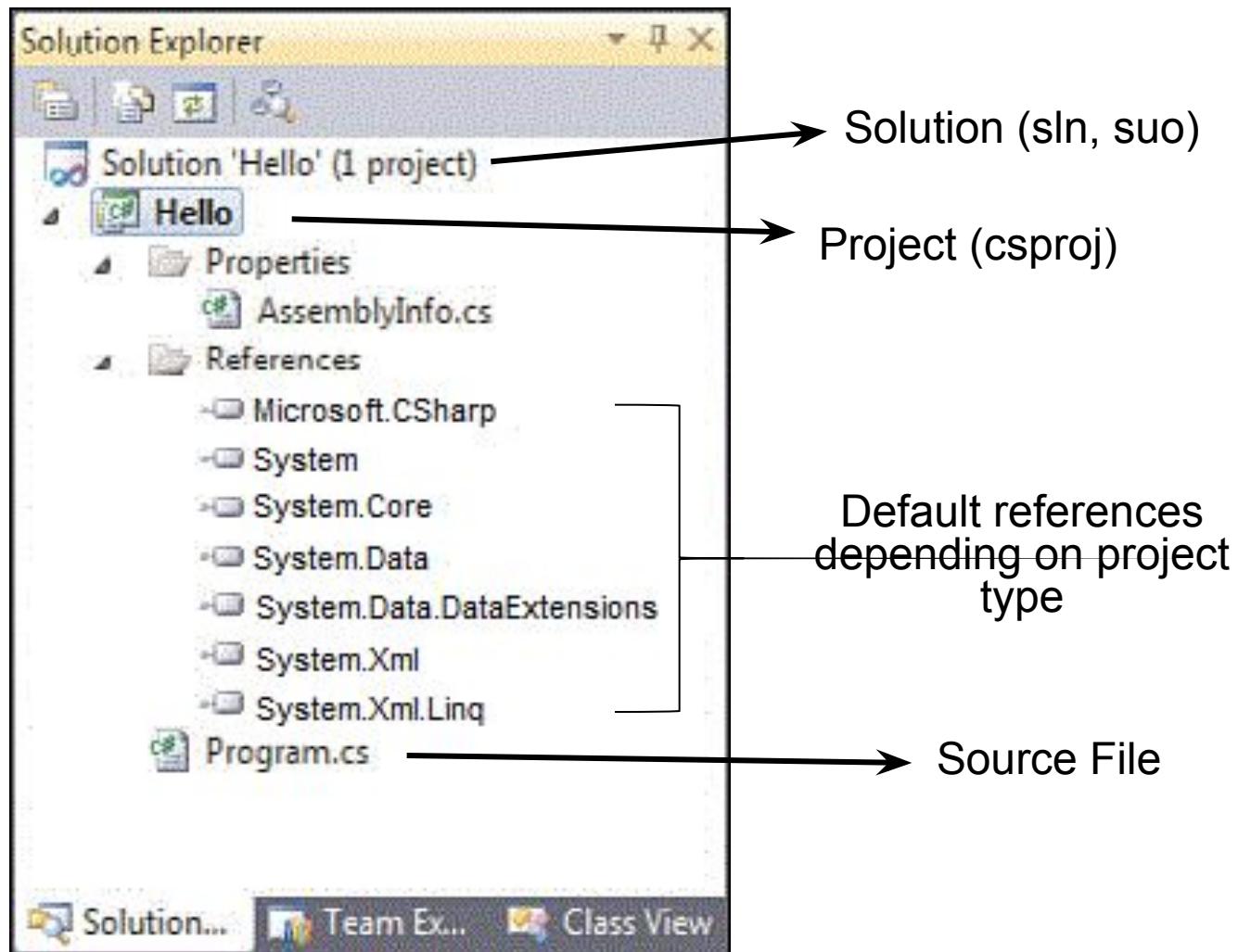
Visual Studio IDE



Project Templates in VS 2010



Visual Studio Solution



Builds and Builds Configuration

- A solution and its individual projects are typically built and tested in a Debug build.
- Use the Configuration Manager Dialog Box to define project configurations, which are sets of properties for each supported combination of build and platform.
- Build configurations provide a way to store multiple versions of solution and project properties.
- The active configuration can be quickly accessed and changed, allowing you to easily build multiple configurations of the same project.



Coding and Debugging Techniques



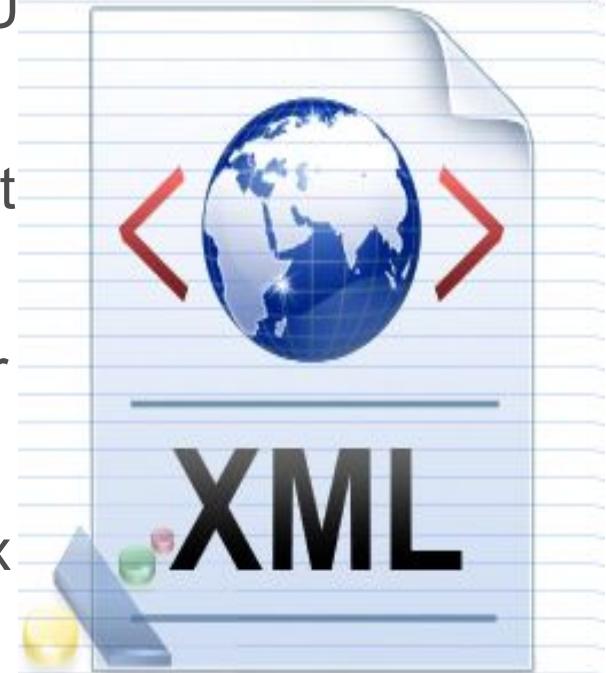
Coding Techniques

- Code Outlining and Regions
 - Add an empty line around the #region keyword
 - Place members in a well-defined order
 - Using #region and #endregion statements, to expand and collapse a block of code.
 - use #regions only for:
 - Private fields and constants (preferably in a Private Definitions region).
 - Nested classes
 - Interface implementations (only if the interface is not the main purpose of that class)



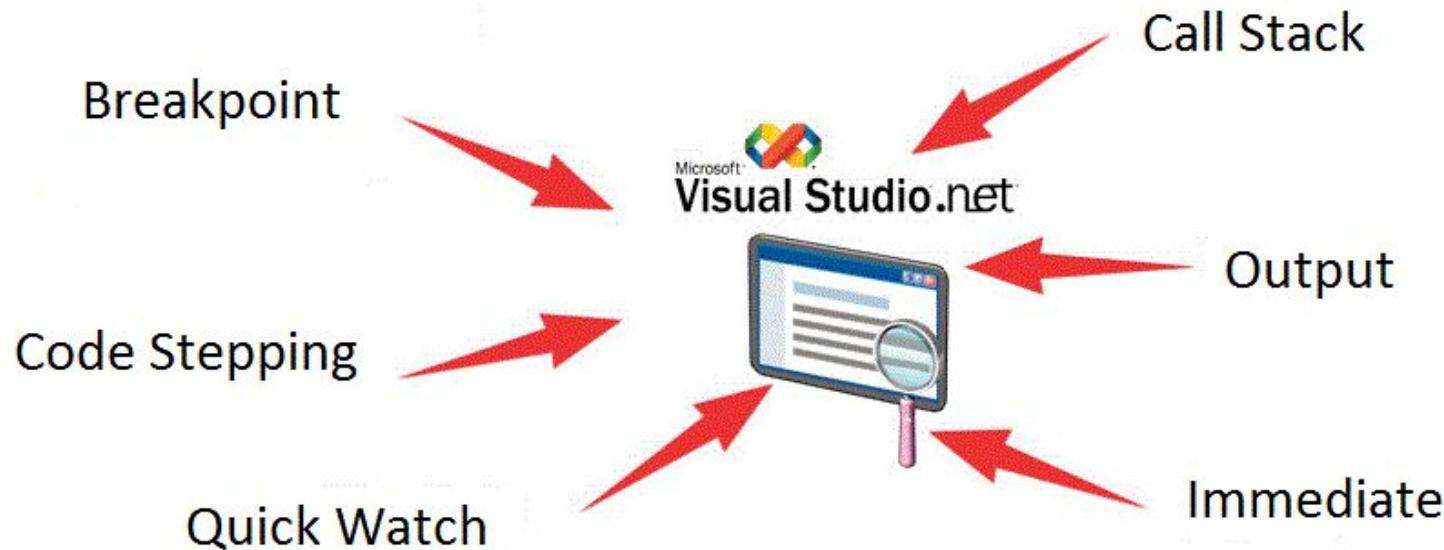
Coding Techniques

- **Inline/XML**
 - Write comments and documentation in ~~UML~~ English
 - Use XML comments to document almost all definitions except namespace.
 - Write XML documentation with the caller in mind
 - Only write comments to explain complex algorithms or decisions
 - Avoid inline comments
 - Don't use /* */ for comments

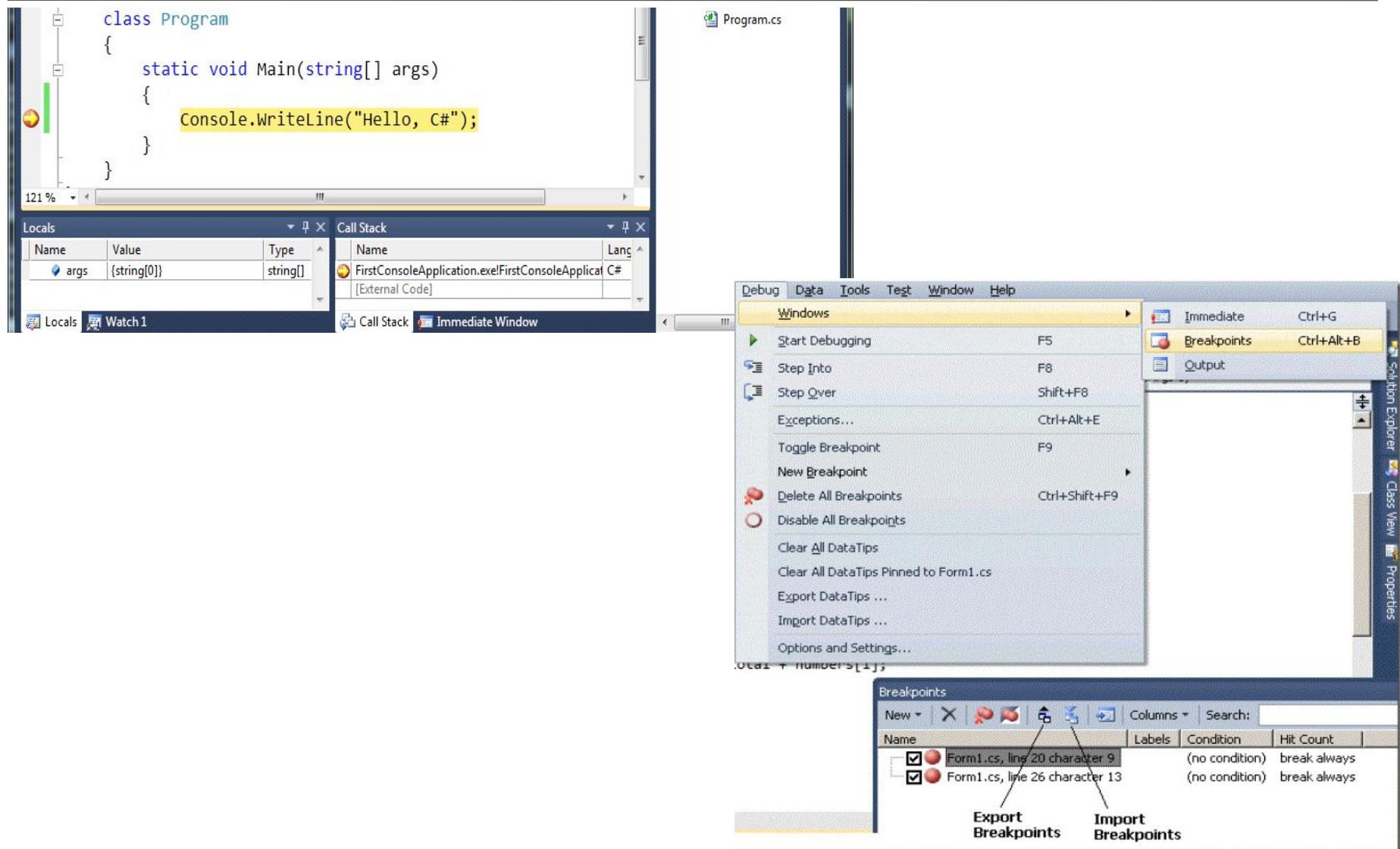


Debugging Techniques

Visual Studio 2010 provides several tools to help you debug code



Debugging in Visual Studio



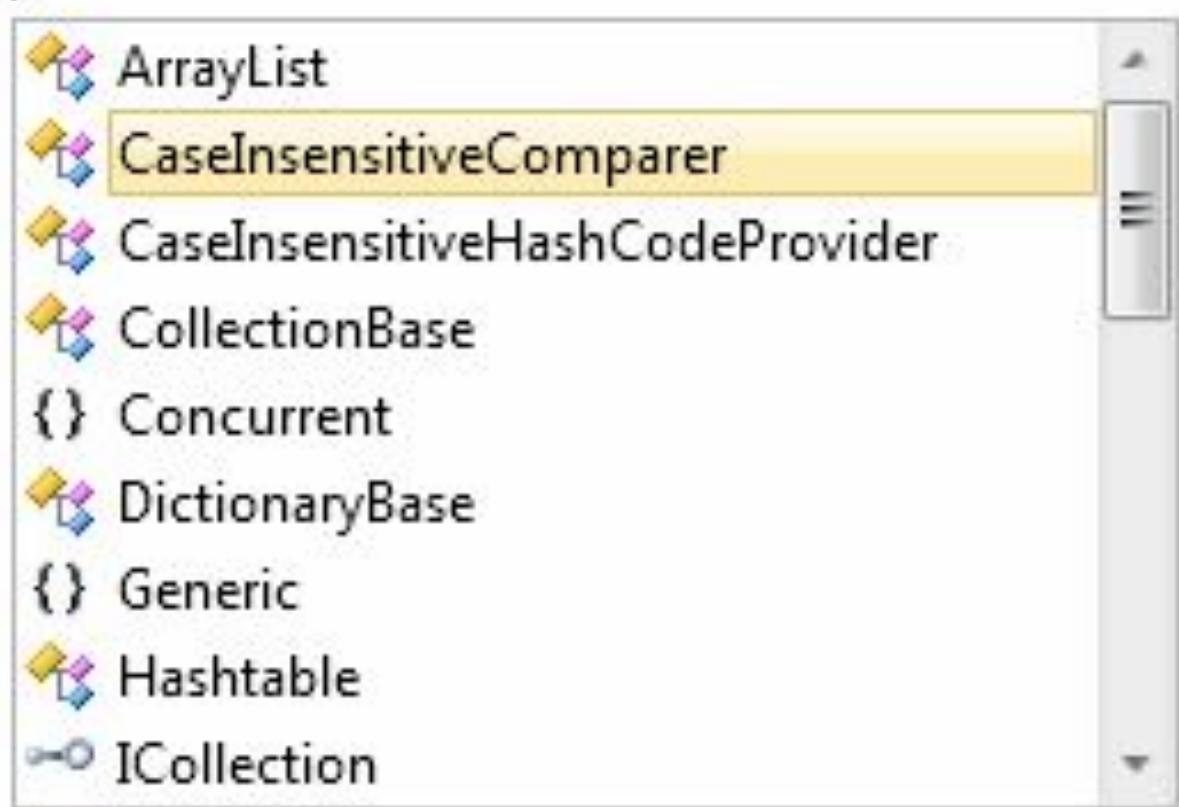
Productivity Tools



Intellisense

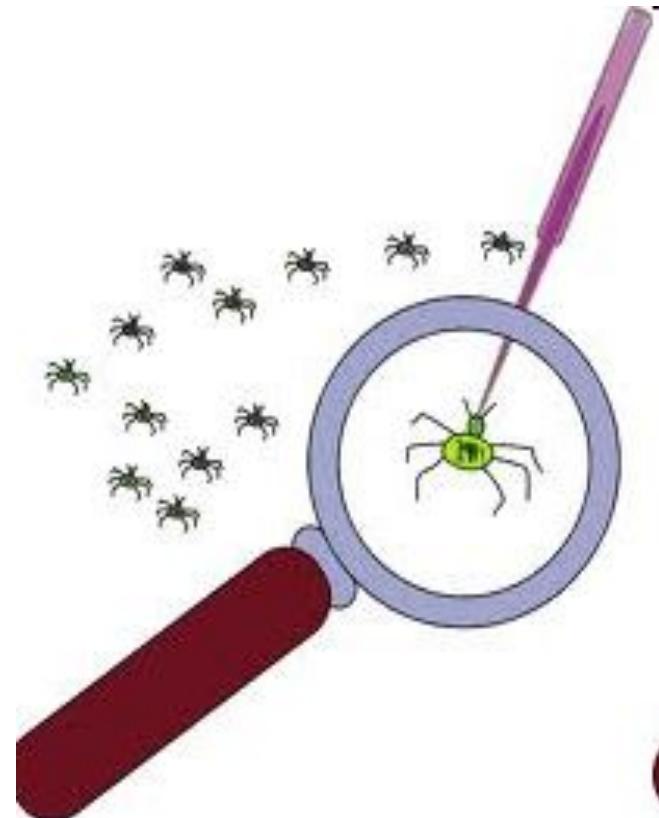
- Offers Quick Info and Complete Word

System.Collections.|

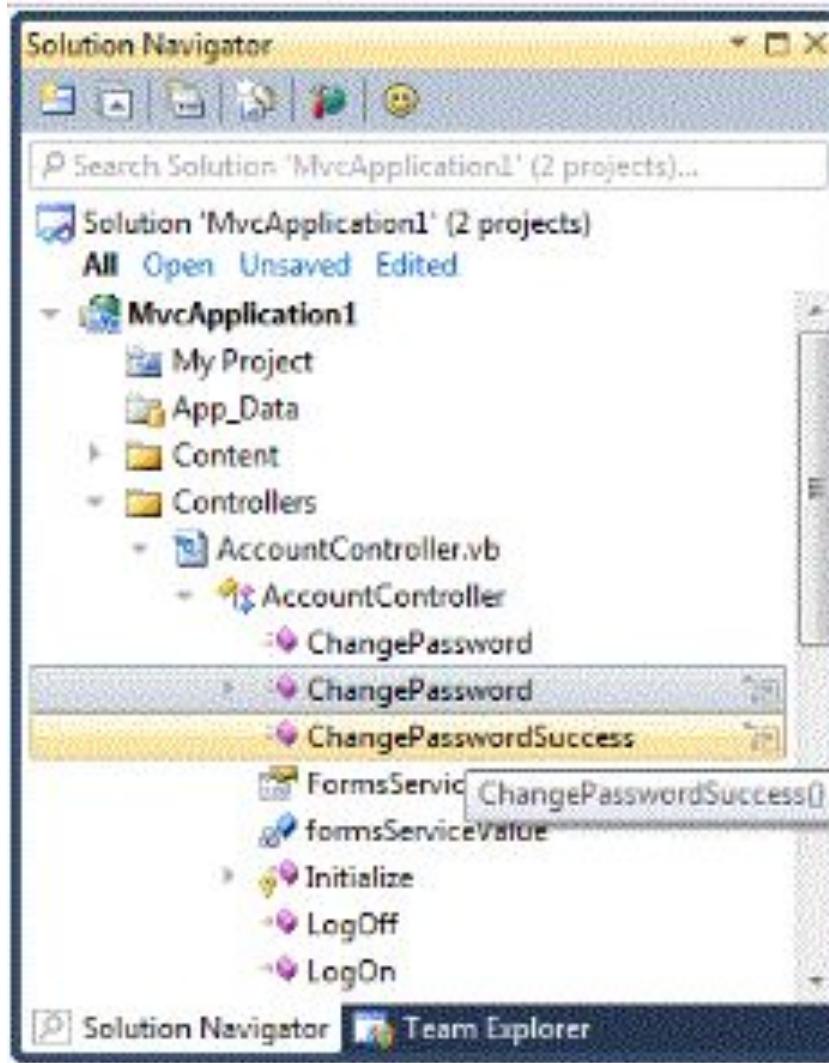


Debugging the Code

- The process of debugging the code application includes:
 - Spotting an error
 - Finding the lines of code that cause the error
 - Fixing the code
 - Testing to check if the error is gone and no errors are introduced
- Iterative and continuous process



Solution Navigator



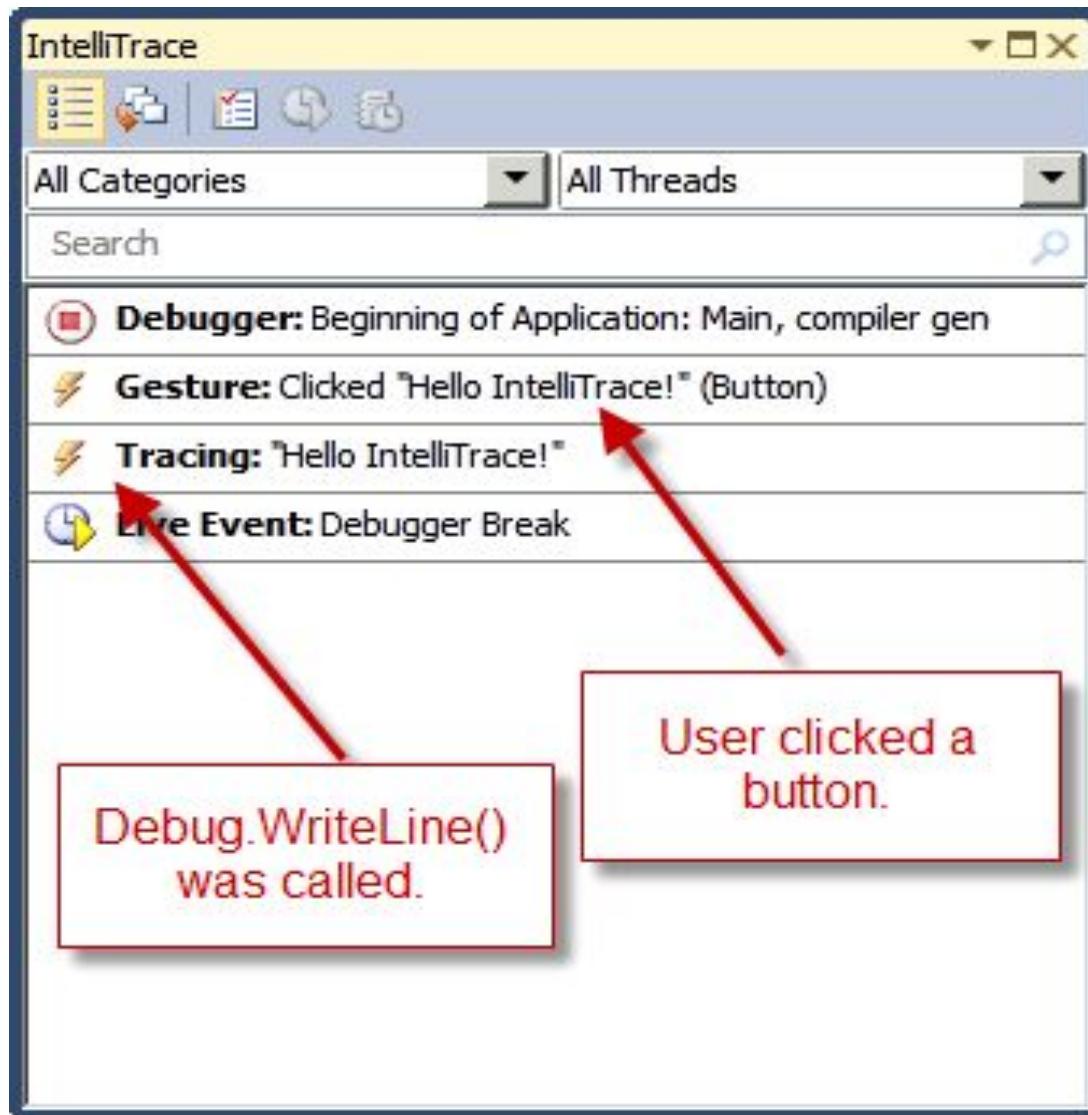
Intellitrace

- It is available in Microsoft Visual Studio 2010 Ultimate edition
- It operates invisibly in the background, recording debug information.
- In IntelliTrace mode, you can navigate to various points in time where events of interest have been recorded.
- Advantages:
 - Debug faster
 - Debug non-reproducible errors
 - Debug applications launched from Visual Studio and IntelliTrace files that were created by IntelliTrace or Test Manager.

[Click here to start your trace!](#)

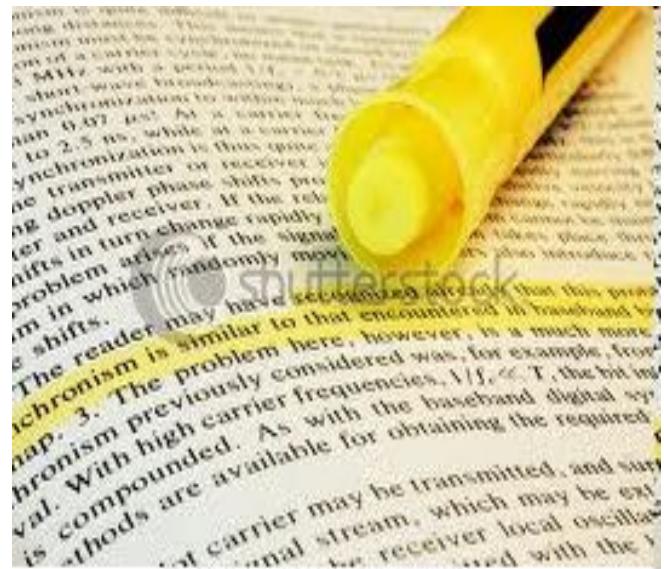


IntelliTrace



Reference Highlighting

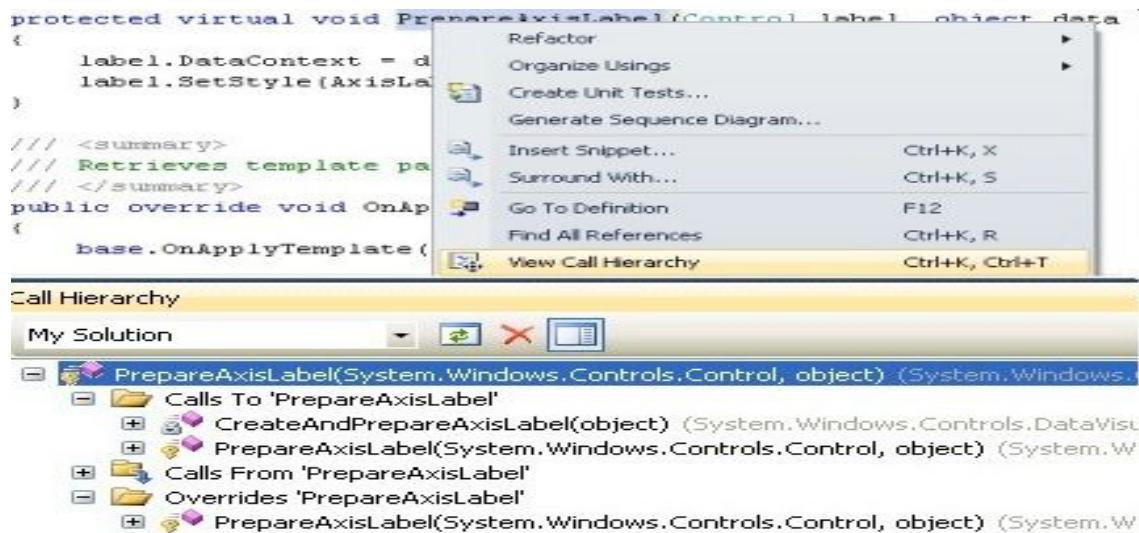
- In the Code Editor, all instances of the symbol are highlighted in the document.
- Highlighted symbols may include declarations and references, names of classes, objects, variables, methods, and properties.
- This makes it very easy to locate where a symbol has been used in that document especially for control structures.



www.shutterstock.com • 2830006

Call Hierarchy

- Helps better understand how code flows and to evaluate the effects of changes to code
- It displays all calls TO and FROM a method, property, or constructor.



Call Hierarchy

The screenshot shows a code editor with a C# class definition:

```
protected ExpressionVisitor()
{
}

protected virtual Expression Visit(Expression exp)
{
    if (exp == null)
        return exp;
    ...
}
```

Below the code editor is a "Call Hierarchy" window. The window title is "Call Hierarchy". The main pane displays the call hierarchy for the "Visit" method:

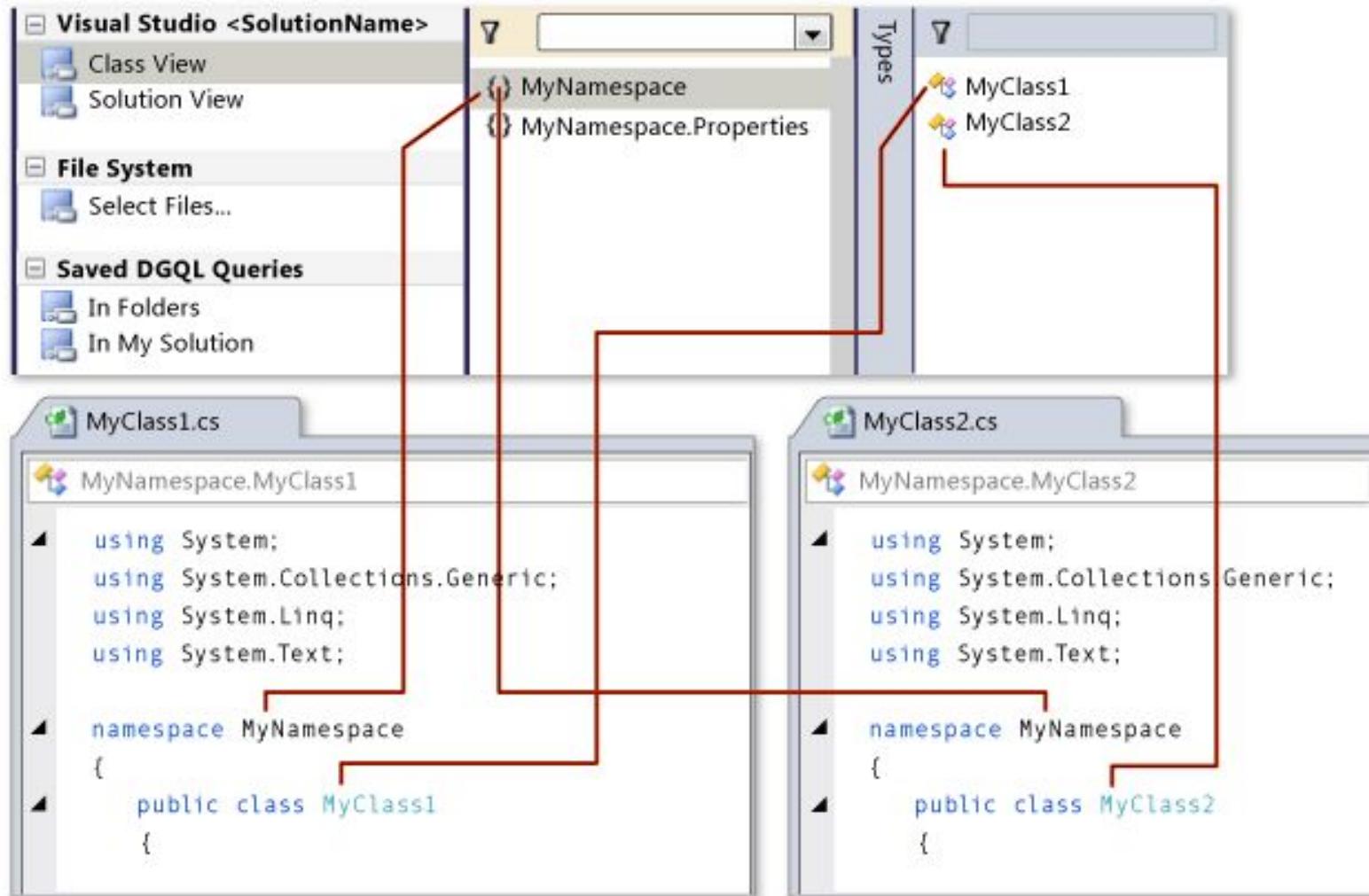
- Visit(System.Linq.Expressions.Expression) (LinqToTerraServiceProvider.ExpressionVisitor)
 - Calls To 'Visit'
 - Calls From 'Visit'
 - Overrides 'Visit'
 - Visit(System.Linq.Expressions.Expression) (LinqToTerraServiceProvider.Evaluator.No...
 - Visit(System.Linq.Expressions.Expression) (LinqToTerraServiceProvider.Evaluator.Su...

Architecture Explorer

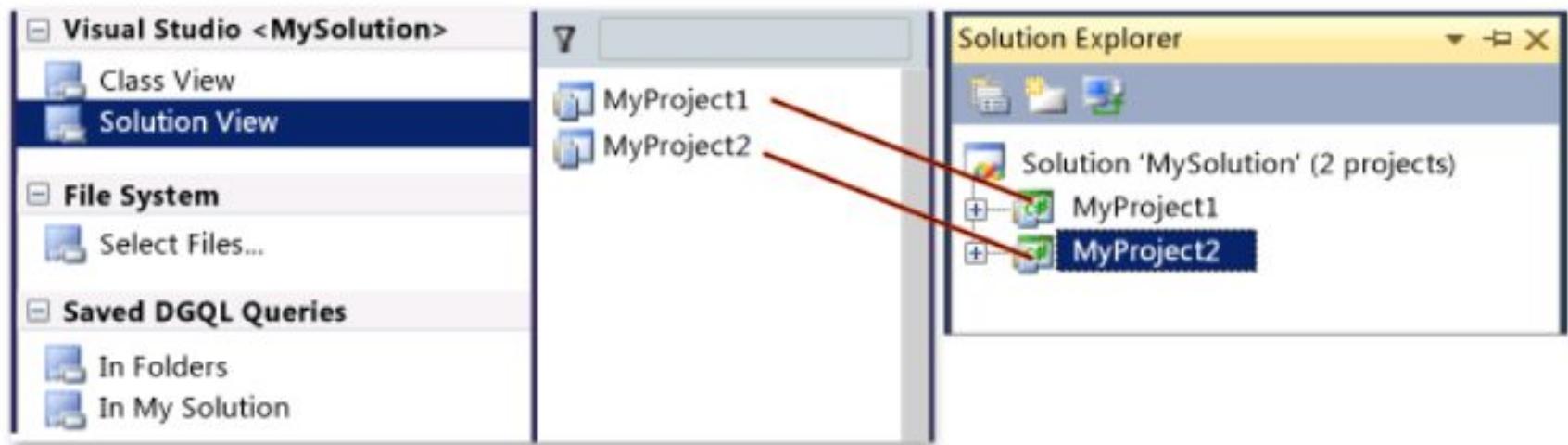
- On the Architecture menu, point to Windows, and then click Architecture Explorer.
- It represents structures as nodes and relationships as links.
- Use to find source code in Visual Studio Solution
- Use to find compiled code in Managed Assembly or Executable Files



Architecture View – Class View

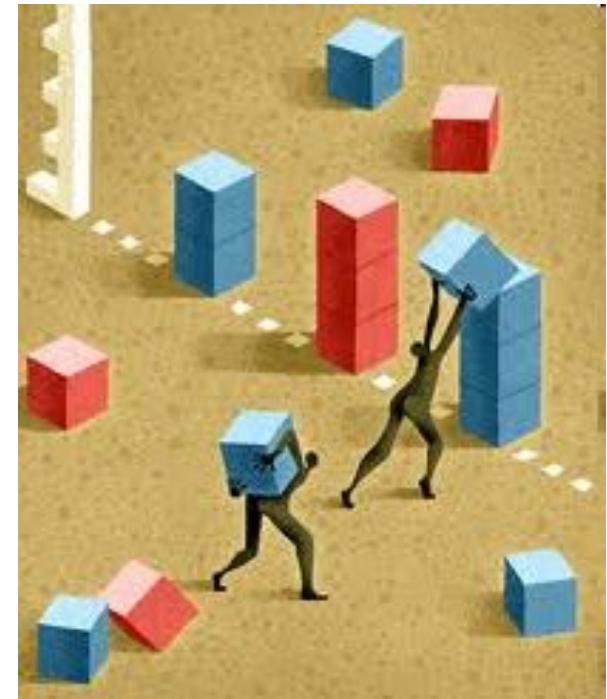


Architecture View – Solution View



Code Metrics

- It is a set of software measures that provide developers better insight into the code they are developing.
- Benefits:
 - Developers can understand which types and/or methods should be reworked or more thoroughly tested.
 - Development teams can identify potential risks, understand the current state of a project, and track progress during software development.

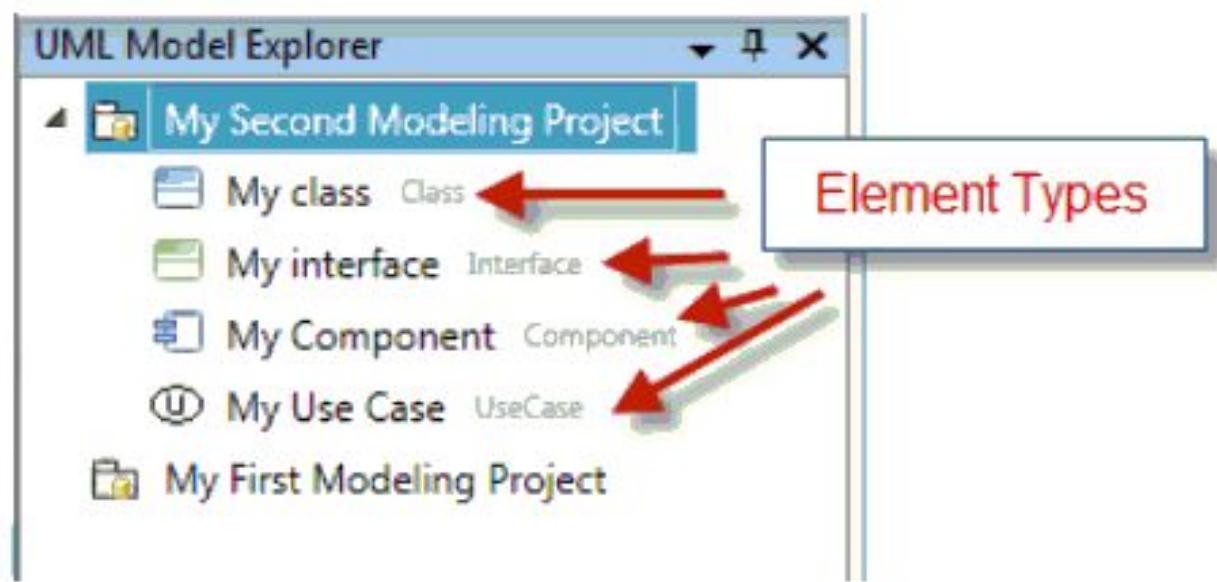


Code Metrics

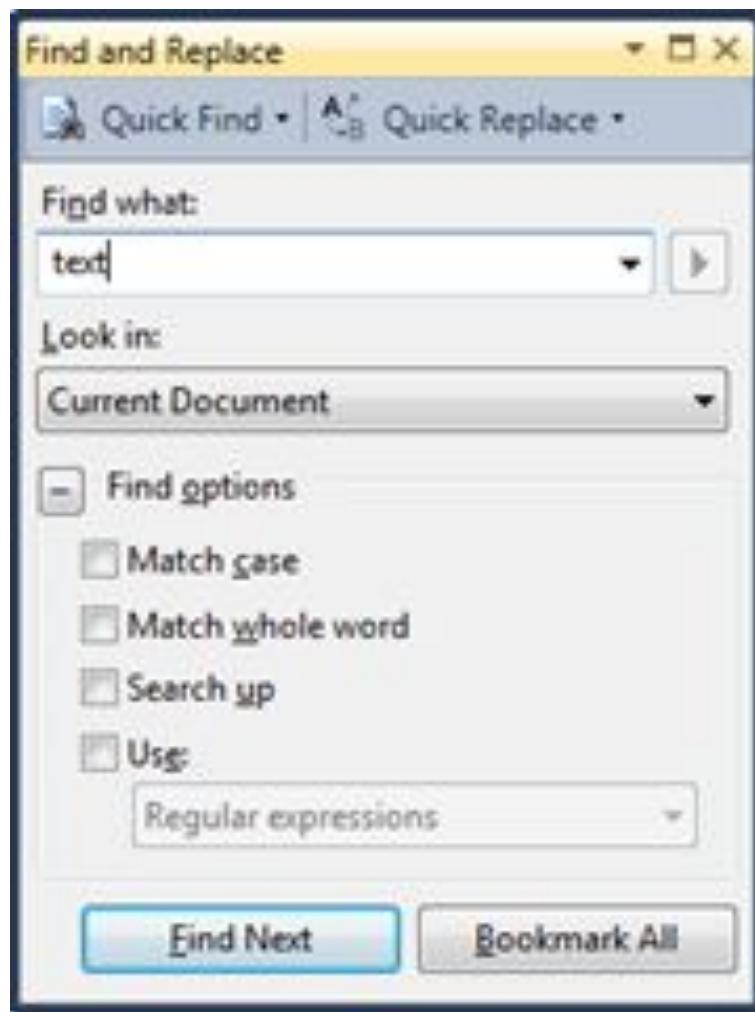
Code Metrics Viewer		Maintainability Index	Cyclomatic Complexity
Hierarchy		0 Max: 45 ➡ Goto Next	
System.Json.mrx			
JsonViewer.WinForms.exe	72		
System.Json.UnitTesting.dll	78		
System.Json.dll	83		
System.Json	88		
System.Json.Contracts	100		
System.Json.Lexer	76		
System.Json.Parser	85		
System.Json.Reflection	84		
System.Json.Serialization	84		
JsonMemberAttribute	92		
JsonRootAttribute	100		
JsonSerializer	64		
Convert(object, Type, Type) : object	68	68	
Deserialize(JsonObject) : object	86	86	
Deserialize(Stream) : IEnumerable<object>	60	60	
Deserialize<T>(JsonObject) : T	85	85	

UML Model Explorer

- A toolwindow in Visual Studio Ultimate designed to help understand and manipulate the UML models that will be building.
- A model can help you visualize the world in which your system works, clarify users' needs, define the architecture of your system, analyze the code, and ensure that your code meets the requirements.



Code Navigation – Find and Replace



Code navigation – Incremental Search

```
14     public class ScarletCharacter
15     {
16         IAdornmentLayer _layer;
17         IWpfTextView _view;
18         Brush _brush;
19         Pen _pen;
20
21         public ScarletCharacter(IWpfTextView view)
22         {
23             _view = view;
24             IncrementalSearch: tex
```

Incremental Search: tex

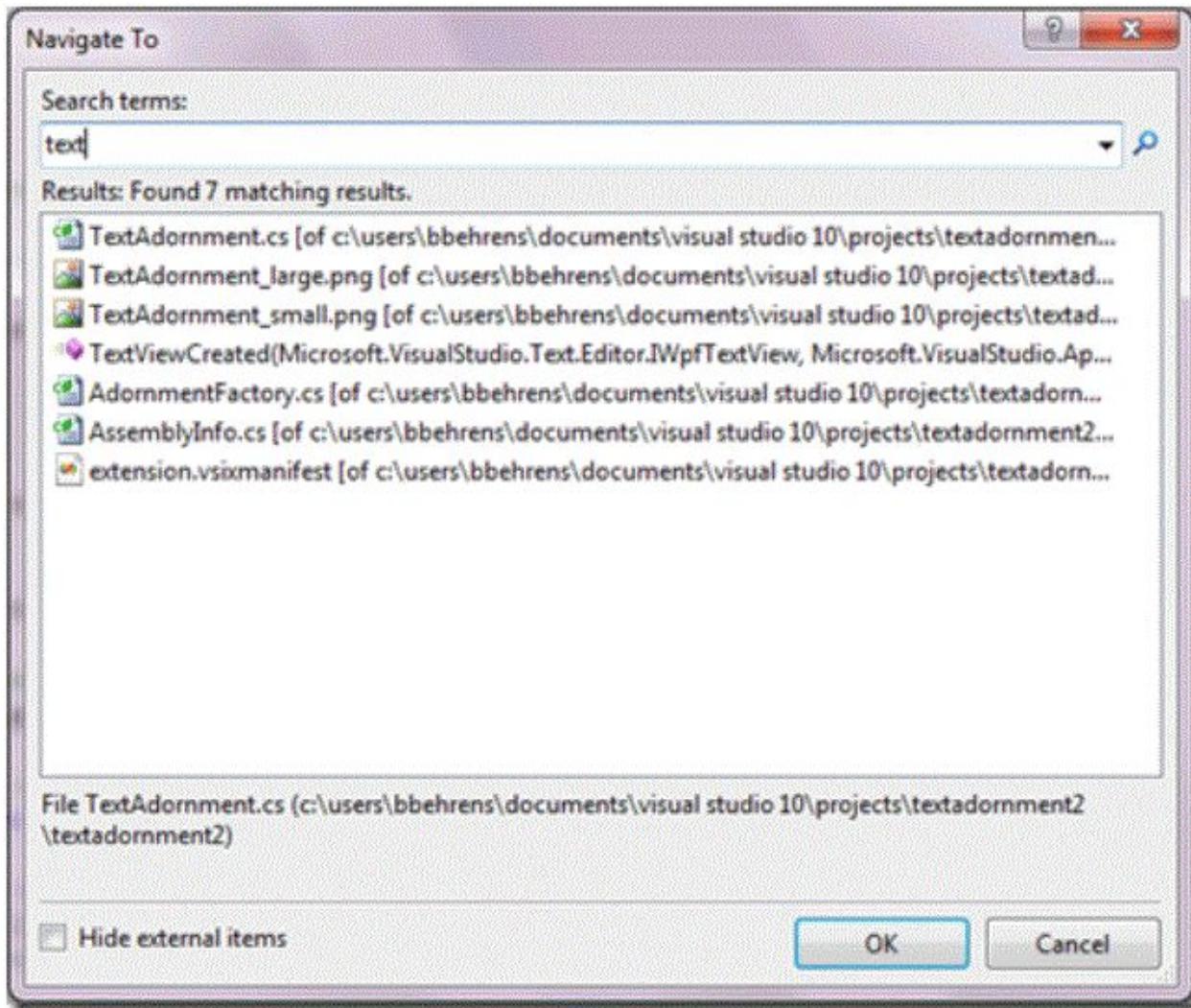
Ln 17

Col 16

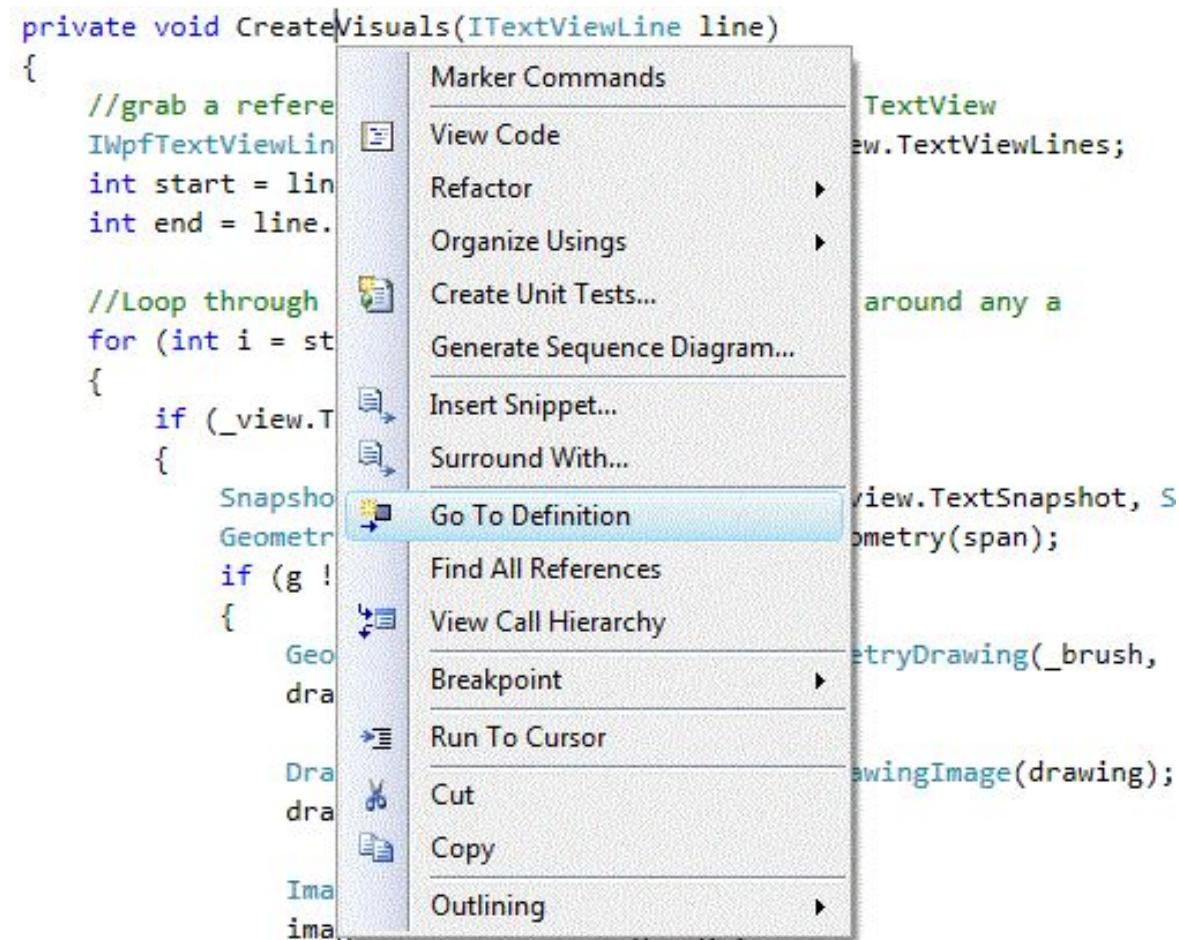
Ch 16

INS

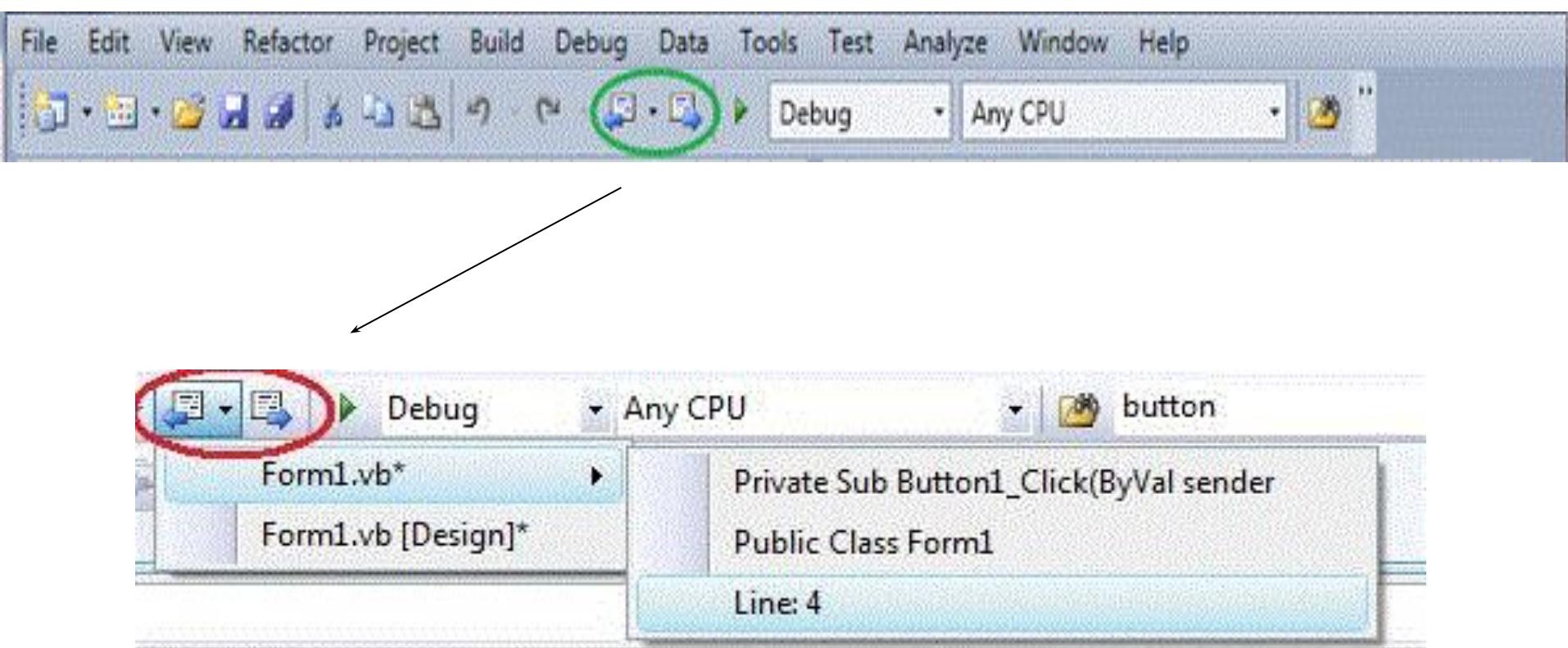
Code Navigation – Navigate To



Code Navigation – Go To Definition

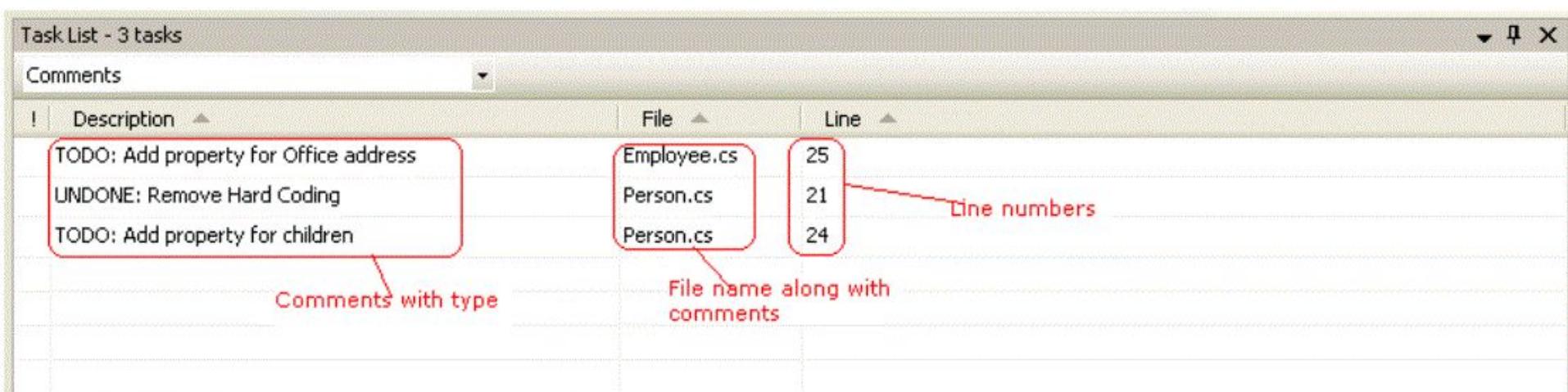


Code Navigation –Navigate Backward/Forward



Task List

- Two types of tasks can be captured in the Task List pane: **User Tasks** and **Code Comments**.
- Comment token: TODO, HACK, or UNDONE, or a custom comment token.



The screenshot shows the 'Task List - 3 tasks' pane with the following details:

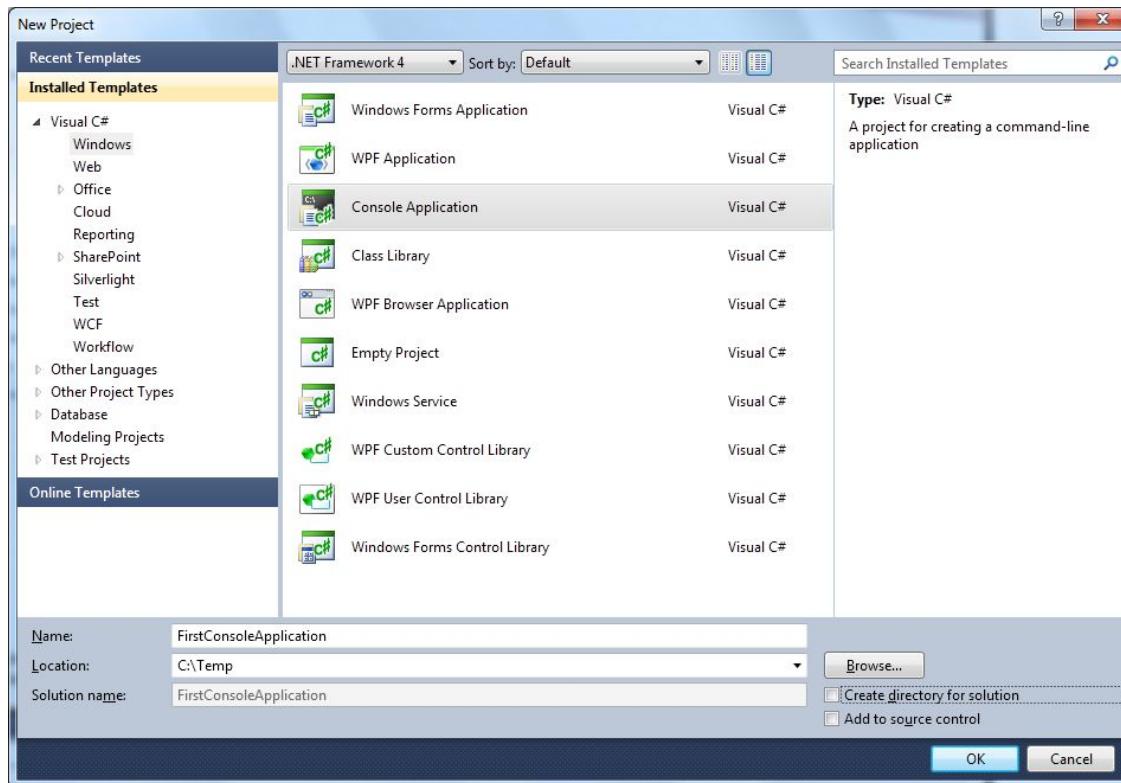
- Comments:** The pane is set to 'Comments' view.
- Columns:** The columns are 'Description', 'File', and 'Line'.
- Tasks:** There are three tasks listed:
 - Description:** TODO: Add property for Office address
 - Type:** UNDONE: Remove Hard Coding
 - Description:** TODO: Add property for children
- File Column:** Shows file names: Employee.cs, Person.cs, Person.cs.
- Line Column:** Shows line numbers: 25, 21, 24.

Annotations with red boxes and arrows point to specific areas:

- A red box encloses the first three items in the 'Description' column, with a red arrow pointing to it labeled "Comments with type".
- A red box encloses the file names in the 'File' column, with a red arrow pointing to it labeled "File name along with comments".
- A red box encloses the line numbers in the 'Line' column, with a red arrow pointing to it labeled "Line numbers".

Creating New Console Application

- File □ New □ Project
- Choose C# console application
- Choose project directory and name



Creating New Console Application

- Visual Studio creates some source code for you

The screenshot shows the Microsoft Visual Studio interface for a new console application named "FirstConsoleApplication". The code editor displays the following C# code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

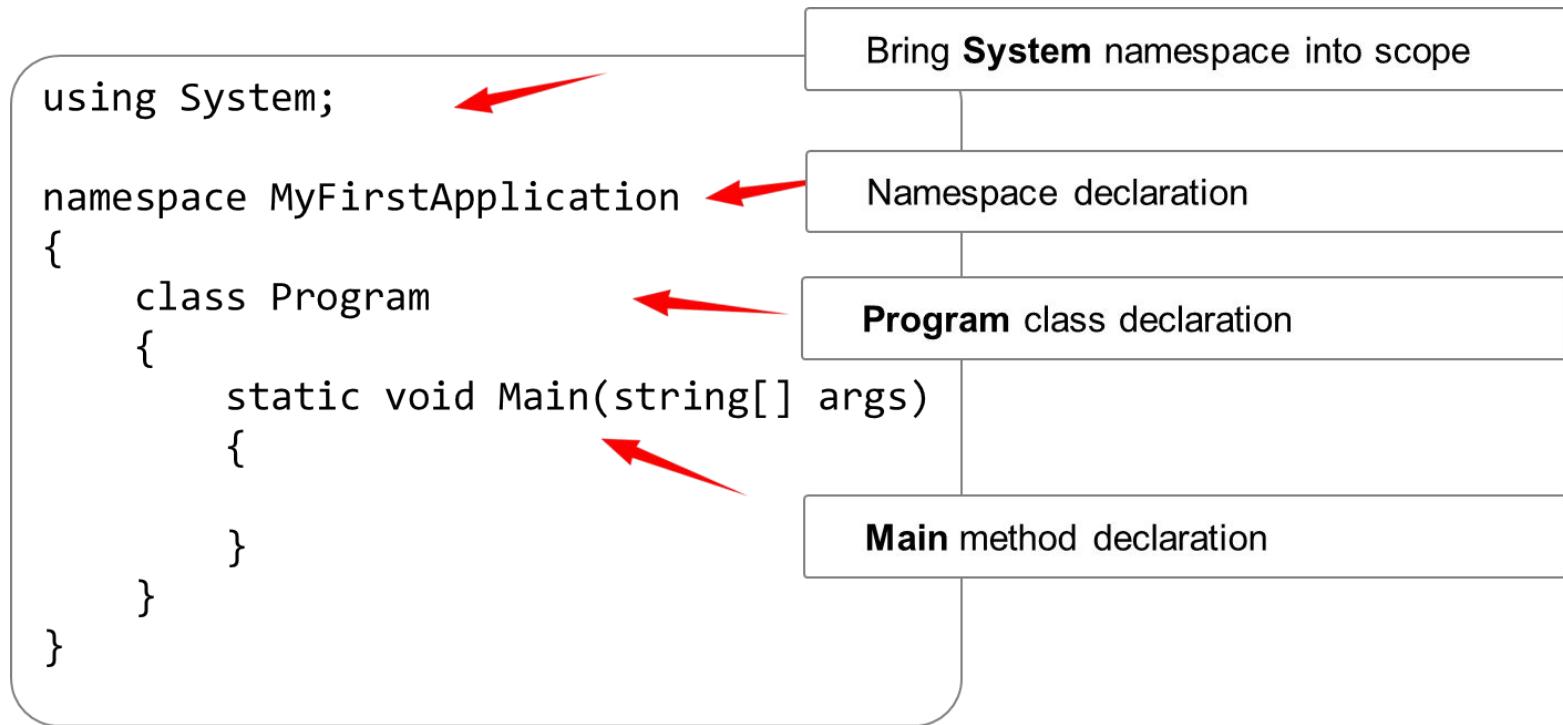
namespace FirstConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Three callout bubbles highlight specific aspects of the generated code:

- A bubble pointing to the namespace declaration (`namespace FirstConsoleApplication`) contains the text: "Namespace not required".
- A bubble pointing to the imports (`using` statements) contains the text: "Some imports are not required".
- A bubble pointing to the class definition (`class Program`) contains the text: "Class name should be changed".

Creating New Console Application

- Structure of a Console Application



System.Console

System.Console method includes:

Clear()

Read()

ReadKey()

ReadLine()

Write()

WriteLine()

```
using System;  
...  
Console.WriteLine("Hello there!");
```

Compiling Source Code

- The process of compiling includes:
 - Syntactic checks
 - Type safety checks
 - Translation of the source code to lower level language (MSIL)
 - Creating of executable files (assembly)
- You can start compilation by
 - Using Build->Build Solution/Project
 - Pressing [F6] or [Shift+Ctrl+B]



© 3poD - www.ClipartOf.com/46011

Running Programs

- The process of running application includes:
 - Compiling (if project not compiled)
 - Starting the application
- You can run application by:
 - Using Debug->Start menu
 - By pressing [F5] or [Ctrl+F5]



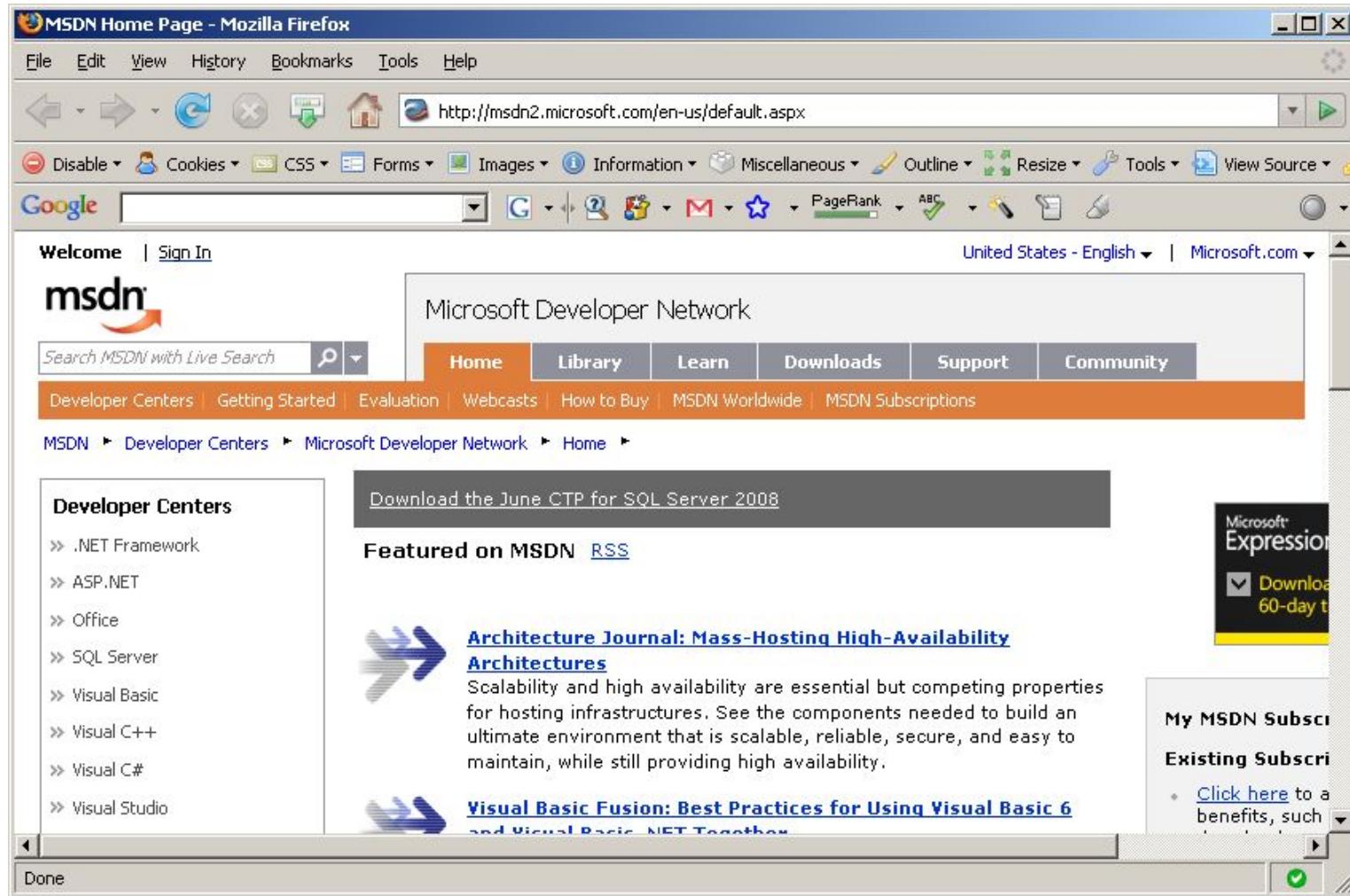
NOTE: Not all types of projects are able to be started!

What is MSDN Library?

- Complete documentation of all classes and their functionality
 - With descriptions of all methods, properties, events, etc.
 - With code examples
- Related articles
- Library of samples
- Use Local copy of the Web version at
<http://msdn.microsoft.com/>



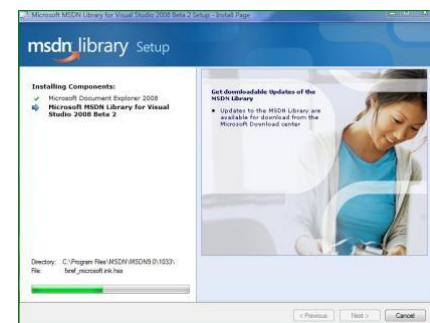
MSDN Library



This screenshot shows the Microsoft Developer Network (MSDN) homepage as it appeared in Mozilla Firefox. The browser window title is "MSDN Home Page - Mozilla Firefox". The address bar shows the URL <http://msdn2.microsoft.com/en-us/default.aspx>. The page itself features the MSDN logo and navigation links for Home, Library, Learn, Downloads, Support, and Community. A sidebar on the left lists developer centers for .NET Framework, ASP.NET, Office, SQL Server, Visual Basic, Visual C++, Visual C#, and Visual Studio. The main content area highlights the "Architecture Journal: Mass-Hosting High-Availability Architectures" and "Visual Basic Fusion: Best Practices for Using Visual Basic 6 and Visual Basic .NET Together". A sidebar on the right promotes Microsoft Expression and provides links for existing subscriptions.

How to use MSDN Library?

- Offline version
 - Use the table of contents
 - Use the alphabetical index
 - Search for phrase or keyword
 - Filter by technology
 - Browse your favorite articles
- Online version
 - Use the built-in search



Checkpoint

- All are new productivity tools in Visual Studio 2010 except:
 - a. Code Navigation – Navigate To
 - b. Solution Navigator
 - c. Solution Explorer
- A project template that is considered lightweight compared to the Windows Forms application template because there is no graphical user interface.
 - a. Console Application
 - b. ASP.Net Web Application
 - c. Class Library



Checkpoint

Checkpoint

- Which is not a debugging tool in Visual Studio?
 - a. BreakPoint
 - b. Immediate
 - c. Code Stepping
 - d. Stack Watch



Checkpoint

Knowledge Check



Knowledge Check

1. The integrated development environment (IDE) is used for creating applications written in .NET programming languages such as C#.
 - a. Solution Explorer
 - b. Eclipse
 - c. Visual Studio .NET
 - d. Microsoft
2. The.cs filename extension indicates a

 - a. C# file
 - b. dynamic help file
 - c. help file
 - d. cool solution file

Knowledge Check

3. A _____ appears when the mouse pointer is positioned over an IDE toolbar icon for a few seconds.
 - a. drop-down list
 - b. menu
 - c. tool tip
 - d. down arrow

4. The Visual Studio .NET IDE provides:
 - a. help documentation
 - b. a toolbar
 - c. windows for accessing project files
 - d. All of the above

Knowledge Check

5. The _____ allows programmers to modify controls visually, without writing code.
 - a. Properties window
 - b. Solution Explorer
 - c. Menu bar
 - d. Toolbox

6. A Visual Studio IDE Productivity Tool that makes it very easy to locate where a symbol has been used in that document especially for control structures. :
 - a. IntelliTrace
 - b. IntelliSence
 - c. Call Hierarchy
 - d. Reference Highlighting

Knowledge Check

7. What is the process of debugging the code application ?
 1. Testing to check if the error is gone and no errors are introduced
 2. Finding the lines of code that cause the error
 3. Spotting an error
 4. Fixing the Code
- b. 3 – 2 – 4 – 1
- c. 2 – 4 – 3 – 1
- d. 1 – 2 – 3 – 4
- e. 2 – 3 – 1 - 4

Knowledge Check

8. Solution Navigator merges the following functionalities except which A Visual Studio IDE Productivity Tool?
 - a. Class View
 - b. Code Metrics
 - c. Find Symbol References
 - d. Object Browser

9. Below is a list of what can be done in Code Navigation Productivity Tool, except which functionality?
 - a. Incremental Search
 - b. Go to Definition
 - c. Find and Replace
 - d. None of the above

Knowledge Check

10. A Visual Studio IDE Productivity Tool that helps better understand how code flows and to evaluate the effects of changes to code
 - a. IntelliTrace
 - b. Architecture Explorer
 - c. Call Hierarchy
 - d. Reference Highlighting

Module 4: Using C# Programming Constructs



Module Agenda

Identifiers and Variables

Data Types

Type Conversion



Module Objectives

Upon completion of this module,
participants should be able to:

- Differentiate identifiers from variables
- Recognize the different data types in C#
- Differentiate value types vs reference types
- Know data type conversion



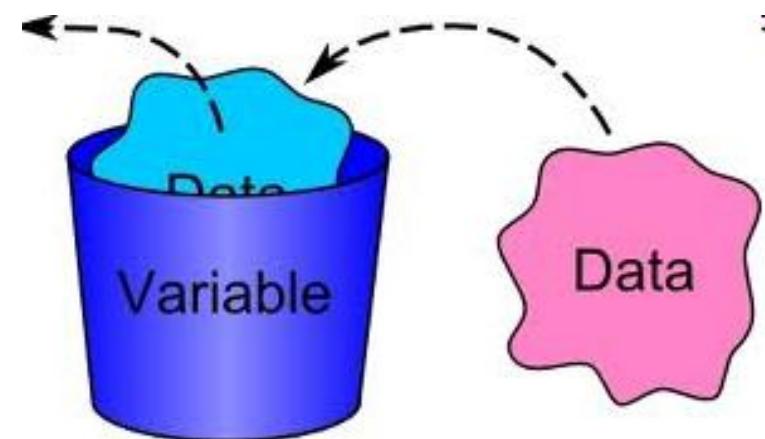
Module objectives

Variables and Identifiers



What is a Variable?

- A variable is a:
 - Placeholder of information that can usually be changed at run-time
- Variables allow you to:
 - Store information
 - Retrieve the stored information
 - Manipulate the stored information

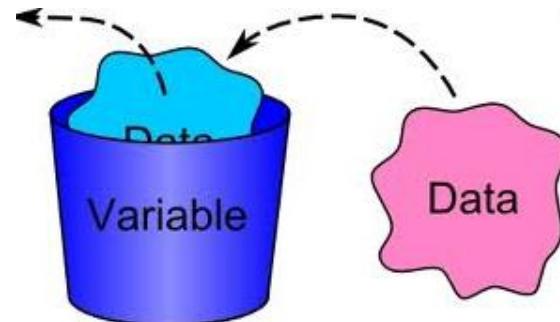


Variable Characteristics

- A variable has:
 - Name
 - Type (of stored data)
 - Value
- Example

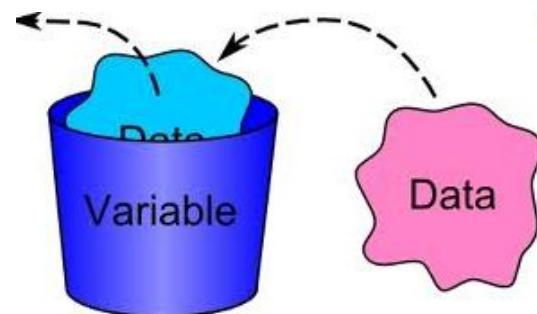
```
int counter = 5;
```

- Name: counter
- Type: int
- Value: 5



Declaring and Using Variables

- When declaring a variable:
 - Specify its type
 - Specify its name (called identifier)
 - May give it an initial value
- Syntax:



```
<data_type> <identifier> [= <initialization>];
```

- Example:

```
int height = 200;
```

Identifiers (1 of 2)

- Identifiers are names used to identify the elements in programs, such as namespaces, classes, methods, and variables.
- Identifiers may consist of:
 - Letters (Unicode)
 - Digits [0-9]
 - Underscore “_”



Identifiers (2 of 2)

- Identifiers

- C# identifiers are case-sensitive
- Cannot be a C# keyword
- Should have a descriptive name
- Should be neither too long nor too short



Identifiers Examples

- Example of correct identifiers

```
int New = 2; // Here N is capital
int _2Pac; // This identifiers begins with _

string поздрав = "Hello"; // Unicode symbols used
// The following is more appropriate:
string greeting = "Hello";

int n = 100; // Undescriptive
int numberOfClients = 100; // Descriptive

// Overdescriptive identifier:
int numberOfPrivateClientOfTheFirm = 100;
```

- Example of incorrect identifiers

```
int new; // new is a keyword
int 2Pac; // Cannot begin with a digit
```

Do not use these C# keywords as Identifiers

C# keywords						
abstract	add*	alias*	as	ascending*	base	bool
break	by*	byte	case	catch	char	checked
class	const	continue	decimal	default	delegate	descending*
do	double	dynamic*	else	enum	equals*	event
explicit	extern	false	finally	fixed	float	for
foreach	from*	get*	global*	goto	group*	if
implicit	in	int	into*	interface	internal	is
join*	let*	lock	long	namespace	new	null
object	on*	operator	out	orderby*	override	params
partial*	private	public	ref	protected	readonly	remove*
return	sbyte	sealed	select*	set*	short	stackalloc
sizeof	static	string	struct	switch	this	throw
true	try	typeof	uint	unchecked	ulong	unsafe
ushort	using	value*	var*	virtual	void	
volatile	where*	while	yield*	* Contextual keyword		

PascalCasing and camelCasing

Pascal Casing		Camel Casing	
Class, Struct	AppDomain	Private field	listItem
Interface	IBusinessService	Variable	listOfValues
Enumeration type	ErrorLevel	Const variable	maximumItems
Enumeration values	FatalError	Parameter	typeName
Event	Click		
Protected field	MainPanel		
Const field	MaximumItems		
Read-only static field	RedValue		
Method	ToString		
Namespace	System.Drawing		
Property	BackColor		
Type Parameter	TEntity		

Data Type



What is a Data Type?

- A Data Type:
 - Is a domain of values of similar characteristics
 - Defines the type of information stored in the computer memory (variable)
- Examples
 - Positive integers: 1, 2, 3, ...
 - Alphabetical characters: a, b, c, ...
 - Days of week: Monday, Tuesday, ...



Data Types in .NET Framework

All types in the .NET Framework are either:

- Value type
 - primitive data type
 - enumeration
 - structures
- Reference type
 - classes
 - string
 - interface
 - array
 - delegate



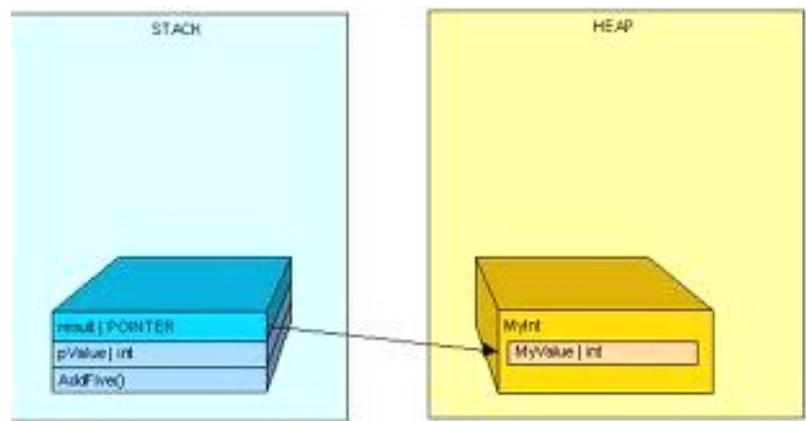
Value Types

- When value type is copied, the variables do not refer to the same object
- When data is changed in either variable, the changes will not be reflected in other copies
- Value on stack



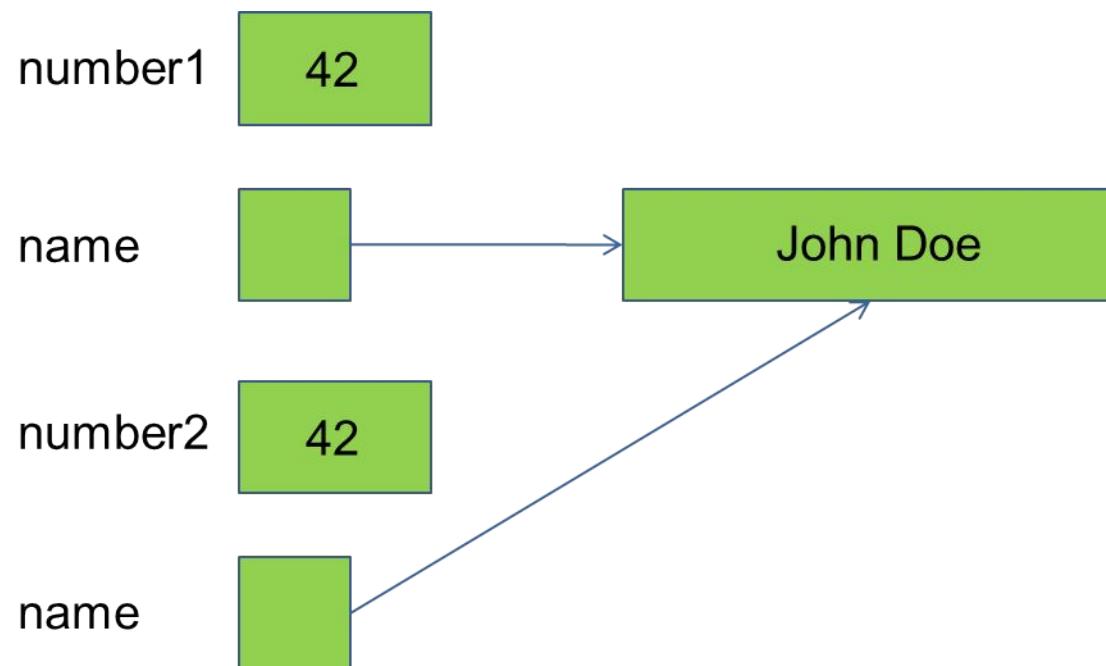
Reference Types

- Reference types - value on heap; use 'ref' keyword
 - When assigning a reference, it refers to an object in memory
 - When the same reference is assigned to two different variables, both variables refer to the same object
 - When the data in the object is changed, the changes will be reflected in all variables that reference that object



Value vs. Reference Types

```
int      number1 = 42;  
string   name = "John Doe";  
int      number2 = number1;  
string   text = name;
```



Passing a Value Type by Reference into a Method

```
static void Main(string[] args)
{
    int myValue = 1005;
    ChangeInput(myValue);
}
```



myValue still equals 1005

```
static void ChangeValue(int input)
{
    input = 29910;
}
```

```
static void Main(string[] args)
{
    int myValue = 1005;
    ChangeInput(ref myValue);
}
```



myValue now equals 29910

```
static void ChangeValue(ref int input)
{
    input = 29910;
}
```

Value Types

Type	System namespace	Definition	Uninitialized value
short	System.Int16	16 bit signed integer	0
int	System.Int32	32 bit signed integer	0
long	System.Int64	64 bit signed integer	0
ushort	System.UInt16	16 bit unsigned integer	0
uint	System.UInt32	32 bit unsigned integer	0
ulong	System.UInt64	64 bit unsigned integer	0
float	System.Single	32 bit real number	0.0
double	System.Double	64 bit real number	0.0
decimal	System.Decimal	128 bit real number	0
bool	System.Boolean	true or false	false
char	System.Char	16 bit Unicode character	'\0'
byte	System.Byte	8 bit unsigned integer	0
sbyte	System.SByte	8 bit signed integer	0
enum	user-defined	defines a type for a closed set	0 index value
struct	user-defined	defines a compound type that consists of other types	assumed value types, null reference types

Primitive Data Types

- Primitive Data Types
 - Integer
 - Floating-Point / Decimal Floating-Point
 - Boolean
 - Character
 - String
 - Object



What are Integer Types?

- Integer types:
 - Represent whole numbers
 - Maybe signed or unsigned
 - Have range of values, depending on the size of memory used
- The default value of integer type is:
 - **0** – for integer types, except
 - **0L** – for the **long** type

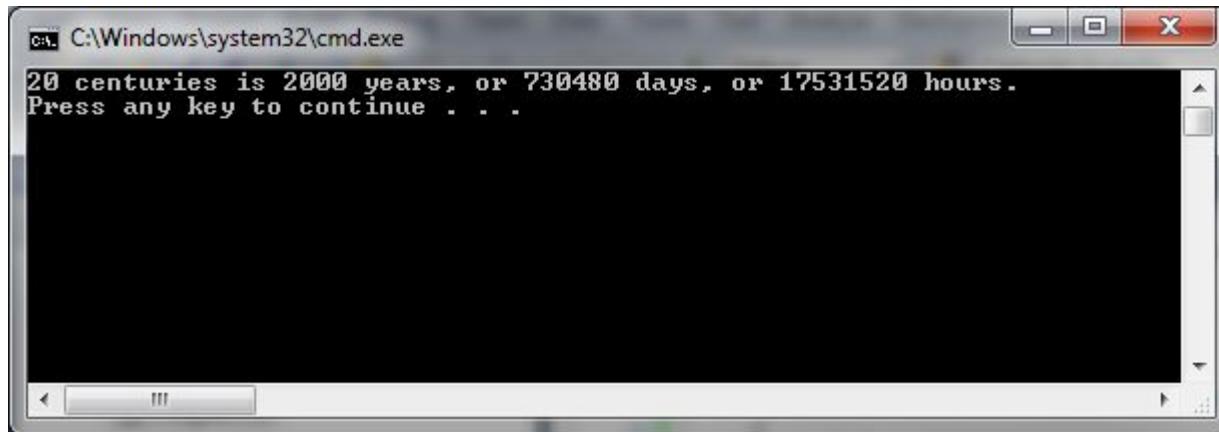


Integer Types

Integer Type	Size	Minimum	Maximum
sbyte	8 bits	-128	127
byte	8 bits	0	255
short	16 bits	-32,768	32767
ushort	16 bits	0	65,535
int	32 bits	-2,147,483,648	2,147,483,647
uint	32 bits	0	4,294,967,295
long	64 bits	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
ulong	64 bits	0	18,446,744,073,709,551,615

Integer Type Example

```
byte centuries = 20;      // Usually a small number
ushort years = 2000;
uint days = 730480;
ulong hours = 17531520; // May be a very big number
Console.WriteLine("{0} centuries is {1} years, or {2}
days, or {3} hours.", centuries, years, days, hours);
```



What are Floating-Point Types?

- Floating-point types:
 - Represent real numbers
 - Maybe signed or unsigned
 - Have range of values and different precision depending on the used memory
 - Can behave abnormally in the calculations



Floating-Point Types

Floating-Point Type	Size	Minimum	Maximum
float	32	$\pm 1.5 \times 10^{-45}$	$\pm 3.4 \times 10^{38}$
double	64	$\pm 5.0 \times 10^{-324}$	$\pm 1.7 \times 10^{308}$

- The default value of floating-point types:
 - **0.0F** for the **float** type
 - **0.0D** for the **double** type

Floating-Point Type Example

```
float floatPI = 3.141592653589793238f;  
double doublePI = 3.141592653589793238;  
Console.WriteLine("Float PI is: {0}", floatPI);  
Console.WriteLine("Double PI is: {0}", doublePI);
```



- NOTE: The “f” suffix in the first statement!
 - Real numbers are by default interpreted as double!
 - One should explicitly convert them to float

Decimal Floating-Point Types

- There is a special decimal floating-point real number type in C#:
 - **decimal** ($\pm 1,0 \times 10^{-28}$ to $\pm 7,9 \times 10^{28}$): 128-bits, precision of 28-29 digits
 - Used for financial calculations
 - No round-off errors
 - Almost no loss of precision
- The default value of **decimal** type is:
 - **0.0M** (M is the suffix for decimal numbers)

$$\frac{53}{100} = 0.53$$

Boolean Data Type

- The Boolean data type:
 - Is declared by the `bool` keyword
 - Has two possible values: true or false
 - Is useful in logical expressions
- The default value is false



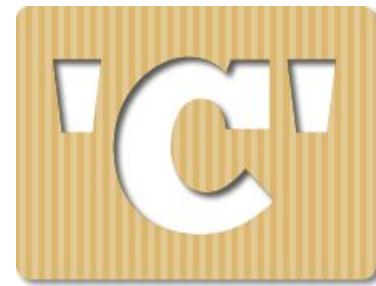
Boolean Example

- Example of boolean variables taking values of true or false:

```
int a = 1;  
int b = 2;  
  
bool greaterAB = (a > b);  
  
Console.WriteLine(greaterAB); // False  
  
bool equalA1 = (a == 1);  
  
Console.WriteLine(equalA1); // True
```

Character Type

- The Character data type:
 - Represents symbolic information
 - Is declared by the char keyword
 - Gives each symbol a corresponding integer code
 - Has a '\0' default value
 - Takes 16 bits of memory (from u+0000 to U=FFFF)



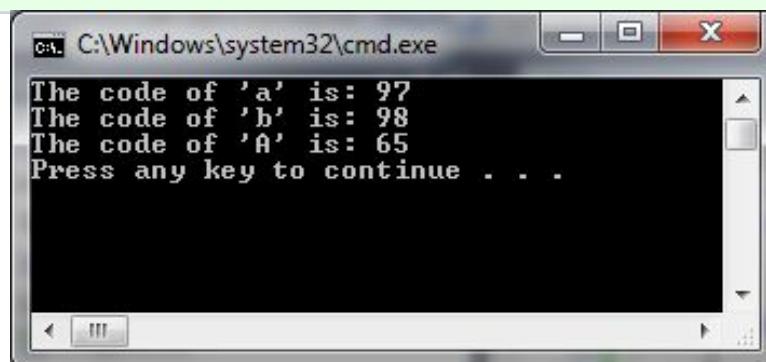
Character and Codes

- The example below shows that every symbol has an its unique Unicode code:

```
char symbol = 'a';
Console.WriteLine("The code of '{0}' is: {1}",
    symbol, (int) symbol);

symbol = 'b';
Console.WriteLine("The code of '{0}' is: {1}",
    symbol, (int) symbol);

symbol = 'A';
Console.WriteLine("The code of '{0}' is: {1}",
    symbol, (int) symbol);
```



String Type

- The string data type:
 - Represents a sequence of characters
 - Is declared by the string keyword
 - Has a default value null
- Strings are enclosed in quotes:



```
string s = "Microsoft .NET Framework";
```

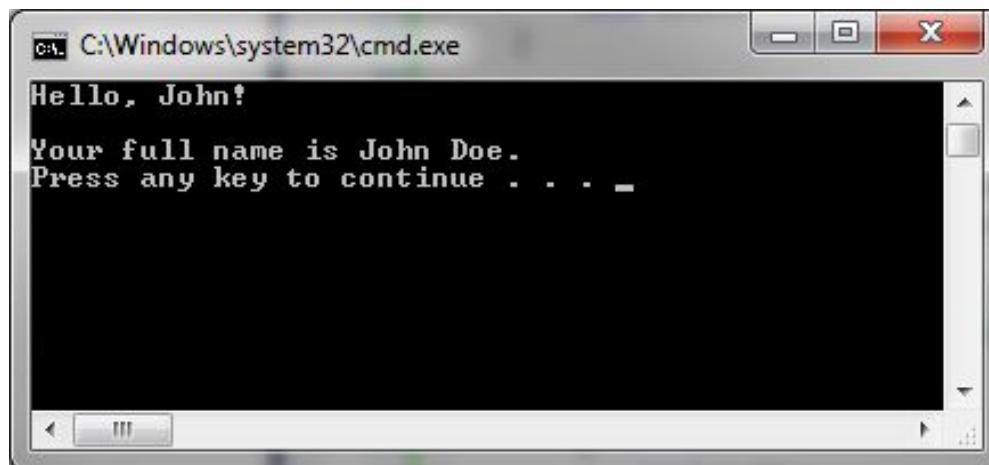
- Strings can be concatenated using the + operator

String Example

- Concatenating the two names of a person to obtain his full name:

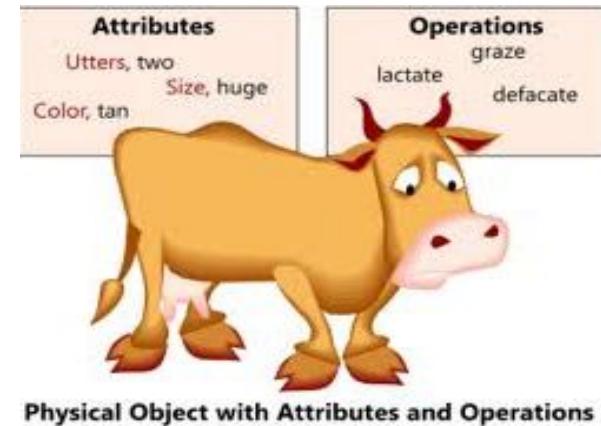
```
string firstName = "John";
string lastName = "Doe";
Console.WriteLine("Hello, {0}!\n", firstName);

string fullName = firstName + " " + lastName;
Console.WriteLine("Your full name is {0}.",
    fullName);
```



Object Type

- The object type:
 - Is declared by the object keyword
 - Is the base type of all other types
 - Can hold values of any type



Using Objects Example

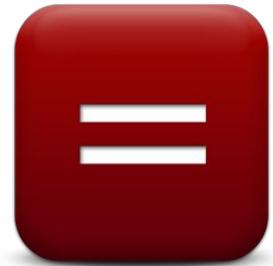
- Example of an object variable taking different types of data:

```
object dataContainer = 5;  
Console.WriteLine("The value of dataContainer is: ");  
Console.WriteLine(dataContainer);  
  
dataContainer = "Five";  
Console.WriteLine("The value of dataContainer is: ");  
Console.WriteLine(dataContainer);
```



Assigning Values to Variables

- Assigning of values to variables
 - Is achieved by the = operator
- The = operator has
 - Variable identifier on the left
 - Value of the corresponding data type on the right
 - Could be used in a cascade calling, where assigning is done from right to left



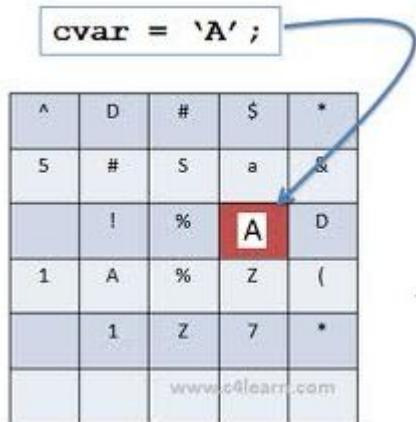
Assigning Values Example

```
int firstValue = 5;  
int secondValue;  
int thirdValue;  
  
// Using an already declared variable:  
secondValue = firstValue;  
  
// The following cascade calling assigns  
// 3 to firstValue and then firstValue  
// to thirdValue, so both variables have  
// the value 3 as a result:  
  
thirdValue = firstValue = 3; // Avoid this!
```

Initializing Variables

- Initializing
 - Is assigning of initial value
 - Must be done before the variable is used
- Several ways of initializing:
 - By using the new keyword
 - By using a literal expression
 - By referring to an already initialized variable

cvar = 'A' ;



^	D	#	\$	*
S	#	S	a	&
!	%	A	D	
1	A	%	Z	(
1	Z	7	*	www.c4learn.com

Initialization Example

```
// The following would assign the default
// value of the int type to num:
int num = new int(); // num = 0

// This is how we use a literal expression:
float heightInMeters = 1.74f;

// Here we use an already initialized variable:
string greeting = "Hello World!";
string message = greeting;
```

Explicit and Implicit Variable Declaration

Explicit	Implicit
<pre>int price = 10;</pre>	<pre>var price = 20;</pre>
<pre>string numberOfEmployees; numberOfEmployees = "Hello";</pre>	<pre>var day = 'Monday';</pre>

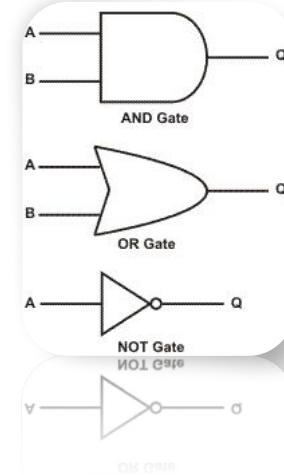
What are Literals

- Literals are:
 - Representations of values in the source code
- There are six types of literals
 - Boolean
 - Integer
 - Real
 - Character
 - String
 - The null literal



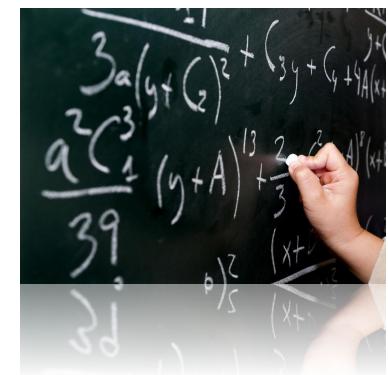
Boolean and Integer Literals

- The boolean literals are:
 - True
 - False
- The integer literals:
 - Are used for variables of type int, uint, long, and ulong
 - Consists of digits
 - May have a sign (+, -)
 - Maybe in a hexadecimal format



Integer Literal Example

- Examples of integer literals
 - The '0x' and '0X' prefixes mean a hexadecimal value,
e.g. 0xA8F1
 - The 'u' and 'U' suffixes mean a ulong or uint type,
e.g. 12345678U
 - The 'l' and 'L' suffixes mean a long or ulong type,
e.g. 9876543L



Integer Literal Example

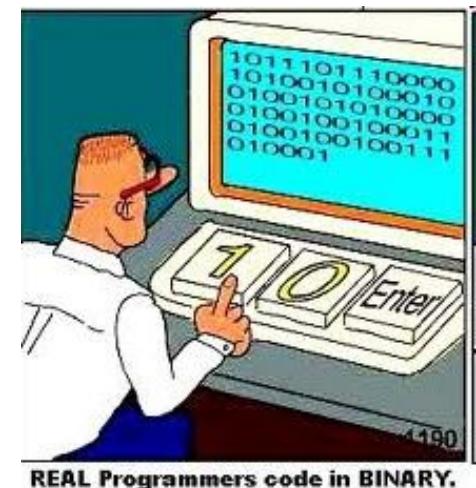
```
// The following variables are
// initialized with the same value:
int numberInHex = -0x10;
int numberInDec = -16;

// The following causes an error,
// because 234u is of type uint
int unsignedInt = 234u;

// The following causes an error,
// because 234L is of type long
int longInt = 234L;
```

Real Literals

- The real literals:
 - Are used for values of type float, double and decimal
 - May consist of digits, a sign and “.”
 - Maybe in exponential notation: 6.02e+23
- The “f” and “F” suffixes mean float
- The “d” and “D” suffixes mean double
- The “m” and “M” suffixes mean decimal
- The default interpretation is double



REAL Programmers code in BINARY.

Real Literals Example

- Example of incorrect float literal:

```
// The following causes an error
// because 12.5 is double by default
float realNumber = 12.5;
```

- A correct way to assign floating-point value (using also the exponential format):

```
// The following is the correct
// way of assigning the value:
float realNumber = 12.5f;

// This is the same value in exponential format:
realNumber = 1.25e+7f;
```

Character Literals

- The character literals:
 - Are used for values of the char type
 - Consist of two single quotes surrounding the character value: ‘<value>’
- The value maybe:
 - Symbol
 - The code of the symbol
 - Escaping sequence



Escaping Sequences

- Escaping sequences are:
 - Means of presenting a symbol that is usually interpreted otherwise
 - Means of presenting system symbols
- Common escaping sequences are:
 - \' for single quote
 - \\" for double quote
 - \\ for backslash
 - \n for new line
 - \uXXXX for denoting any other Unicode symbol



Character Literals Example

```
char symbol = 'a'; // An ordinary symbol

symbol = '\u006F'; // Unicode symbol code in a
// hexadecimal format (letter 'o')

symbol = '\u8449'; // 葉 (Leaf in Traditional Chinese)

symbol = '\''; // Assigning the single quote symbol

symbol = '\\'; // Assigning the backslash symbol

symbol = '\n'; // Assigning new line symbol

symbol = '\t'; // Assigning TAB symbol

symbol = "a"; // Incorrect: use single quotes
```

String Literals

- String literals:
 - Are used for values of the string type
 - Consist of two double quotes surrounding the value: “<value>”
 - May have a @ prefix which ignores the used of escape sequences @”value”
- The value is a sequence of character literals

```
string s = "I am a sting literal";
```



String Literal Example

- Benefits of quoted strings (the @ prefix)

```
// Here is a string literal using escape sequences
string quotation = "\"Hello, Jude\"", he said.;
string path = "C:\\WINNT\\Darts\\Darts.exe";

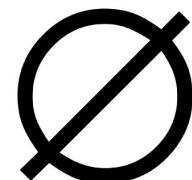
// Here is an example of the usage of @
quotation = @"""Hello, Jimmy!""", she answered.;
path = @"C:\\WINNT\\Darts\\Darts.exe";

string str = @"some
text";
```

- In quoted strings \" is used instead of ""

Nullable Types

- Nullable types are instances of the System.Nullable struct
 - Wrapper over the primitive types
 - E.g. int?, double?, etc.
- Nullable type can represent the normal range of values for its underlying value type, plus an additional null value
- Useful when dealing with Databases or other structures that have default value null



Nullable Type Example

- Example with Integer

```
int? someInteger = null;
Console.WriteLine(
    "This is the integer with Null value -> " + someInteger);
someInteger = 5;
Console.WriteLine(
    "This is the integer with value 5 -> " + someInteger);
```

- Example with Double

```
double? someDouble = null;
Console.WriteLine(
    "This is the real number with Null value -> "
    + someDouble);
someDouble = 2.5;
Console.WriteLine(
    "This is the real number with value 5 -> " +
    someDouble);
```

Read-Only and Constants

- Read-only variable
 - Use `readonly` keyword
 - Initialized at run time

```
readonly string currentDateTime = DateTime.Now.ToString();
```

- Constant
 - Use **const** keyword
 - Initialized at compile time

```
const double PI = 3.14159;
int radius = 5;
double area = PI * radius * radius;
double circumference = 2 * PI * radius;
```

Variable Scope

- **Block scope**

```
if (length > 10)
{
    int area = length * length;
}
```

- **Procedure scope**

```
void ShowName()
{
    string name = "Bob";
}
```

- **Class scope**

```
private string message;

void SetString()
{
    message = "Hello World!";
}
```

- **Namespace scope**

```
public class CreateMessage
{
    public string message
        = "Hello";
}

public class DisplayMessage
{
    public void ShowMessage()
    {
        CreateMessage newMessage
            = new CreateMessage();
        MessageBox.Show(
            newMessage.message);
    }
}
```

Type Conversion



Type Conversion

- Implicit Conversion
 - Automatically performed by the .NET Framework

```
int a = 4;  
long b;  
b = a; //Implicit conversion of int to long
```

- Explicit Conversion
 - Requires writing of code to perform the conversion

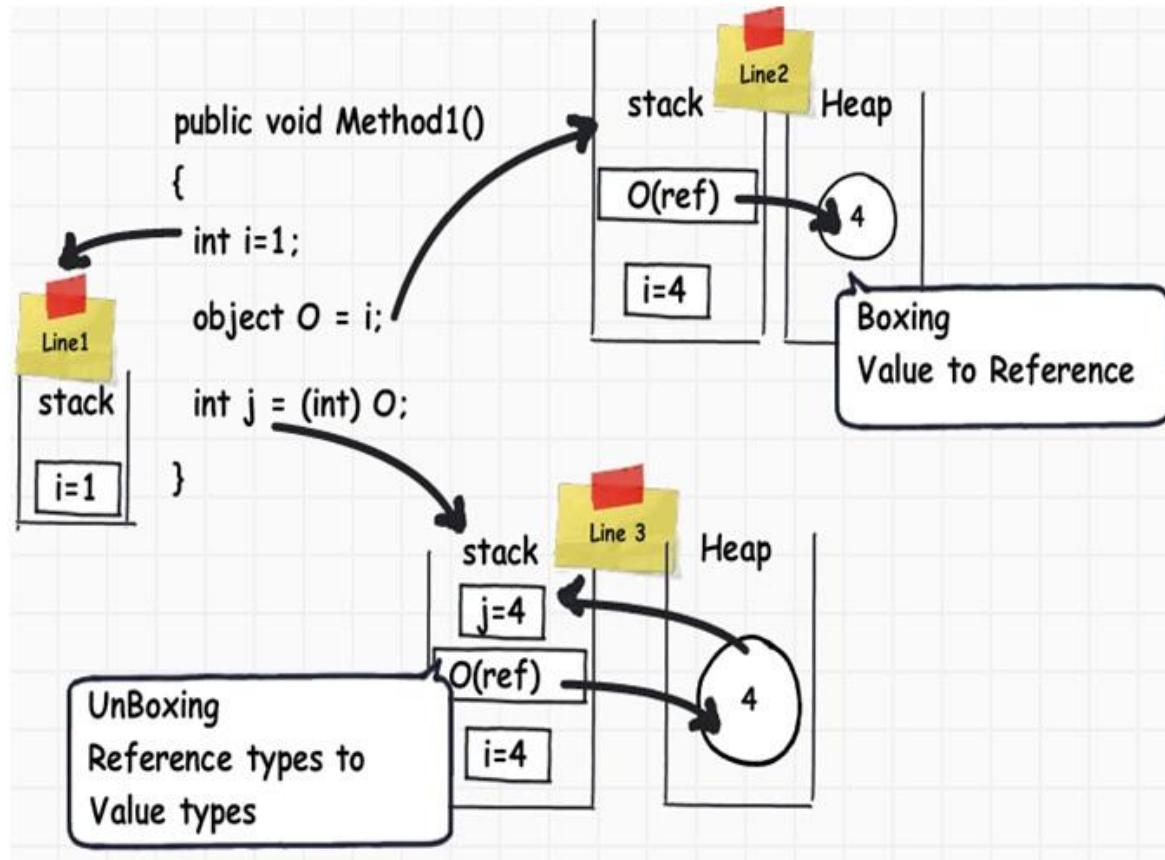
```
DataType variableName = (castDataType) variableName;  
. . .  
int count = Convert.ToInt32("1234");  
. . .  
int number = 0;  
  
if (int.TryParse("1234", out number) ) {
```

Boxing and Unboxing

- Boxing
 - Value type to Reference type
 - The CLR allocates a portion of memory on the heap
 - The CLR copies the value from the stack to a new portion of memory
- Unboxing
 - Reference type to Value type
 - If the types match, the CLR copies the value from the heap



Boxing and Unboxing



Checkpoint

- Which is the correct variable declaration and assignment?
 - a. a int = 5;
 - b. boolean b = true;
 - c. string s = “Incorrect”;

- Which is not a valid data type?
 - a. array
 - b. floating
 - c. delegate



Checkpoint

Checkpoint

- If **Residence** is a class (a reference type), what message does the following code example display?

```
Residence myHouse = new Residence(ResidenceType.House,  
2);  
Residence anotherHouse = new  
Residence(ResidenceType.House, 2);  
if (myHouse == anotherHouse){  
    Console.WriteLine("They're the same house");  
}  
else {  
    Console.WriteLine("They're different houses");  
}  
a. They're the same house.  
b. They're different houses
```



Checkpoint

Checkpoint

- What is boxing?
 - a. Encapsulating an object in a value type.
 - b. Encapsulating a copy of an object in a value type.
 - c. Encapsulating a value type in an object.
 - d. Encapsulating a copy of a value type in an object



Checkpoint

Knowledge Check



Knowledge Check

1. The operator performs division.
 - a. /
 - b. +
 - c. \
 - d. ^

2. Every variable has a
 - a. name
 - b. value
 - c. type
 - d. All of the above

Knowledge Check

3. Arithmetic expressions are evaluated
 - a. from right to left
 - b. from left to right
 - c. according to the rules of operator precedence
 - d. from the lowest level of precedence to the highest level of precedence
4. The operator makes an explicit conversion from one type to another.
 - a. cast
 - b. changetype
 - c. casting
 - d. conversion

Knowledge Check

5. Variables used to store integer values should be declared with the _____ keyword
 - a. integer
 - b. int
 - c. intvariable
 - d. None of the above
6. Which of the following does not store a sign?
 - a. short
 - b. integer
 - c. long
 - d. byte
 - e. single

Knowledge Check

7. Which of the following statements are correct about data types?
1. If the integer literal exceeds the range of *byte*, a compilation error will occur.
 2. We cannot implicitly convert non-literal numeric types of larger storage size to *byte*.
 3. *Byte* cannot be implicitly converted to *float*.
 4. A *char* can be implicitly converted to only *int* data type.
 5. We can cast the integral character codes.
- b. 1, 3, 5
- c. 2, 4
- d. 3, 5
- e. 1, 2, 5

Knowledge Check

8. Which of the following are value types?
- 1. Integer
 - 2. Array
 - 3. Single
 - 4. String
 - 5. Long
- b. 1, 2, 5
 - c. 1, 3, 5
 - d. 2, 4
 - e. 3, 5

Knowledge Check

9. What will be the output of the following code snippet when it is executed?

```
int x = 1;
```

```
float y = 1.1f;
```

```
short z = 1;
```

```
Console.WriteLine((float) x + y * z - (x += (short) y));
```

- a. 0.1
- b. 1.0
- c. 1.1
- d. 11

Knowledge Check

10. Which of the following is an 8-byte integer?
 - a. Char
 - b. Long
 - c. Integer
 - d. Short

11. Which of the following statements is correct?
 - a. Information is never lost during narrowing conversions.
 - b. The *CInteger()* function can be used to convert a Single to an Integer.
 - c. Widening conversions take place automatically.
 - d. Assigning an Integer to an Object type is known as Unboxing.
 - e. 3.14 can be treated as Decimal by using it in the form *3.14F*.

Knowledge Check

12. Which of the following are value types?
1. We can assign values of any type to variables of type object.
 2. When a variable of a value type is converted to object, it is said to be unboxed.
 3. When a variable of type object is converted to a value type, it is said to be boxed.
 4. Boolean variable cannot have a value of *null*.
 5. When a value type is boxed, an entirely new object must be allocated and constructed.
- b. 2, 5
- c. 1, 5
- d. 3, 4
- e. 2, 3

Knowledge Check

13. Which of the following is the correct ways to set a value 3.14 in a variable *pi* such that it cannot be modified?
 - a. *const float pi = 3.14F;*
 - b. *float pi = 3.14F;*
 - c. *const float pi; pi = 3.14F;*
 - d. *pi = 3.14F;*
14. Which of the following is the correct default value of a *Boolean* type?
 - a. 0
 - b. 1
 - c. True
 - d. False

Knowledge Check

15. Which of the following statements are correct about data types?
1. Each value type has an implicit default constructor that initializes the default value of that type.
 2. It is possible for a value type to contain the *null* value.
 3. All value types are derived implicitly from *System.ValueType* class.
 4. It is not essential that local variables in C# must be initialized before being used.
 5. Variables of reference types referred to as objects and store references to the actual data.
- b. 1, 3, 5
- c. 2, 4
- d. 3, 5
- e. 2, 3, 4

Knowledge Check

16. Which of the following are the correct way to initialize the variables *i* and *j* to a value 10 each?
1. int i = 10; int j = 10;
 2. int i, j;
 i = 10 : j = 10;
 3. int i = 10, j = 10;
 4. int i, j = 10;
 5. int i = j = 10;
- b. 2, 4
- c. 1, 3
- d. 3, 5
- e. 4, 5

Knowledge Check

17. Which of the following statement correctly assigns a value 33 to a variable *c*?

byte a = 11, b = 22, c;

- a. *c* = (byte) (*a* + *b*);
- b. *c* = (byte) *a* + (byte) *b*;
- c. *c* = (int) *a* + (int) *b*;
- d. *c* = *a* + *b*;

Knowledge Check

18. You need to identify a type that meets the following criteria:

Is always a number and Is not greater than 65,535.

Which type should you choose?

- a. System.UInt16
- b. int
- c. System.String
- d. System.IntPtr

Knowledge Check

19. Which statement is incorrect when choosing an identifier?
 - a. In C#, an identifier must begin with an underscore, the at sign (@), or an uppercase letter.
 - b. An identifier can contain only letters, digits, underscores, and the “at” sign, not special characters such as #, \$, or &.
 - c. An identifier cannot be a C# reserved keyword.
 - d. None of the above
20. The simple types such as int, bool, char, double etc. are implemented as:
 - a. classes in the .NET Framework class library
 - b. structs in the .NET Framework class library
 - c. all of the above
 - d. None of the above

Module 5: Using C# Programming Constructs



Module Objectives

Upon completion of this module, you should be able to:

- Use operators to construct expressions.
- Use decision statements.
- Use iteration statements.



Module objectives

Module Agenda

Expressions

Operators

Statements

Preprocessor Directives



Expressions



Expressions (1 of 2)

- Expressions are sequences of operators, literals and variables that are evaluated to some value
- Examples:

```
int r = (150-20) / 2 + 5; // r=70
// Expression for calculation of circle area
double surface = Math.PI * r * r;
// Expression for calculation of circle perimeter
double perimeter = 2 * Math.PI * r;
```



Expressions (2 of 2)

- Expressions has:
 - Type (integer, real, boolean ...)
 - Value
- Examples:

Expression of type
int. Calculated at
compile time.

Expression of
type int.
Calculated at
runtime.

```
int a = 2 + 3; // a = 5
int b = (a+3) * (a-4) + (2*a + 7) / 4; // b = 12
bool greater = (a > b) || ((a == 0) && (b == 0));
```

Expression of type bool
Calculated at runtime.

Operators



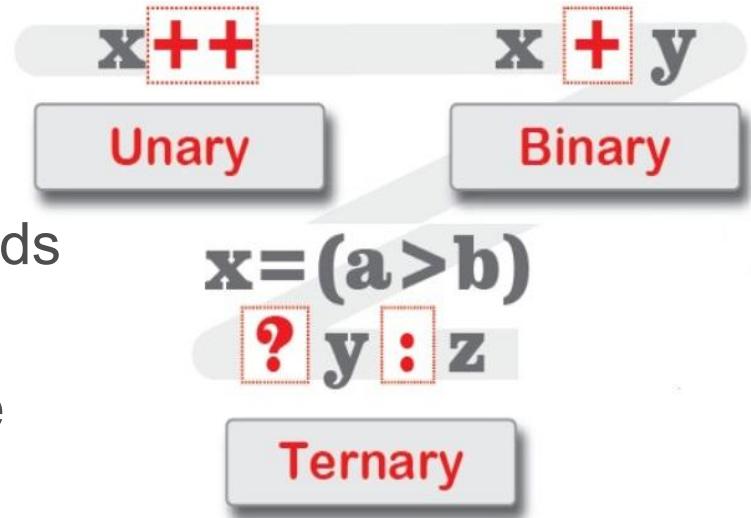
What is an Operator?

- Operator is an operation performed over data at runtime
 - Takes one or more arguments(operands)
 - Produces a new value
- Operators have precedence
 - Precedence defines which will be evaluated first
- Expressions are sequences of operators and operands that are evaluated to a single value



Operators in C#

- Operators in C#:
 - Unary – take one operand
 - Binary – take two operands
 - Ternary (?) – takes three operands
- Excerpts for the assignment operators, all binary operators are left-associative
- The assignment operators and the conditional operator(?) are right-associative
- Parenthesis operator always has highest precedence



Categories of Operators in C#

Operator category	Operators
Arithmetic	+ - * / %
Logical (boolean and bitwise)	& ^ ! ~ && true false
String concatenation	+
Increment, decrement	++ --
Shift	<< >>
Relational	== != < > <= >=
Assignment	= += -= *= /= %= &= = ^= <<= >>=
Member access	.
Indexing	[]
Cast	()
Conditional	? :
Delegate concatenation and removal	+ -
Object creation	new
Type information	as is sizeof typeof
Overflow exception control	checked unchecked
Indirection and Address	* -> [] &

Operators Precedence

Category	Operators
Primary	<code>x.y f(x) a[x] x++ x-- new typeof checked unchecked</code>
Unary	<code>+ - ! ~ ++x --x (T)x</code>
Multiplicative	<code>* / %</code>
Additive	<code>+ -</code>
Shift	<code><< >></code>
Relational and type testing	<code>< > <= >= is as</code>
Equality	<code>== !=</code>
Logical AND	<code>&</code>
Logical XOR	<code>^</code>
Logical OR	<code> </code>
Conditional AND	<code>&&</code>
Conditional OR	<code> </code>
Conditional	<code>?:</code>
Assignment	<code>= *= /= %= += -= <<= >>= &= ^= =</code>

Arithmetic Operators – Example (1 of 2)

```
int squarePerimeter = 17;  
  
double squareSide = squarePerimeter / 4.0;  
  
double squareArea = squareSide * squareSide;  
  
Console.WriteLine(squareSide); // 4.25  
  
Console.WriteLine(squareArea); // 18.0625
```

```
int a = 5;  
  
int b = 4;  
  
Console.WriteLine( a + b ); // 9  
  
Console.WriteLine( a + b++ ); // 9  
  
Console.WriteLine( a + b ); // 10  
  
Console.WriteLine( a + (++b) ); // 11  
  
Console.WriteLine( a + b ); // 11
```



Arithmetic Operators – Example (2 of 2)

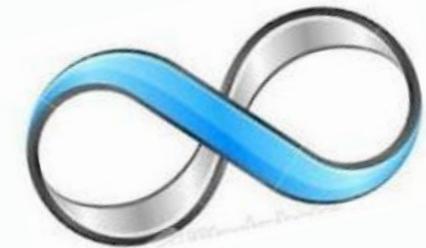
```
Console.WriteLine(11.0 / 3); // 3.666666667
```

```
Console.WriteLine(11 / 3.0); // 3.666666667
```

```
Console.WriteLine(11 % 3); // 2
```

```
Console.WriteLine(11 % -3); // 2
```

```
Console.WriteLine(-11 % 3); // -2
```



```
Console.WriteLine(1.5 / 0.0); // Infinity
```

```
Console.WriteLine(-1.5 / 0.0); // -Infinity
```

```
Console.WriteLine(0.0 / 0.0); // NaN
```

```
int x = 0;
```

```
Console.WriteLine(5 / x); // DivideByZeroException
```

Arithmetic Operators – Overflow Example

```
int bigNum = 2000000000;  
  
int bigSum = 2 * bigNum; // Integer overflow!  
  
Console.WriteLine(bigSum); // -294967296
```

```
bigNum = Int32.MaxValue;  
  
bigNum = bigNum + 1;  
  
Console.WriteLine(bigNum); // -2147483648
```

```
checked
```

```
{
```

```
// This will cause OverflowException
```

```
bigSum = bigNum * 2;
```



Logical Operators

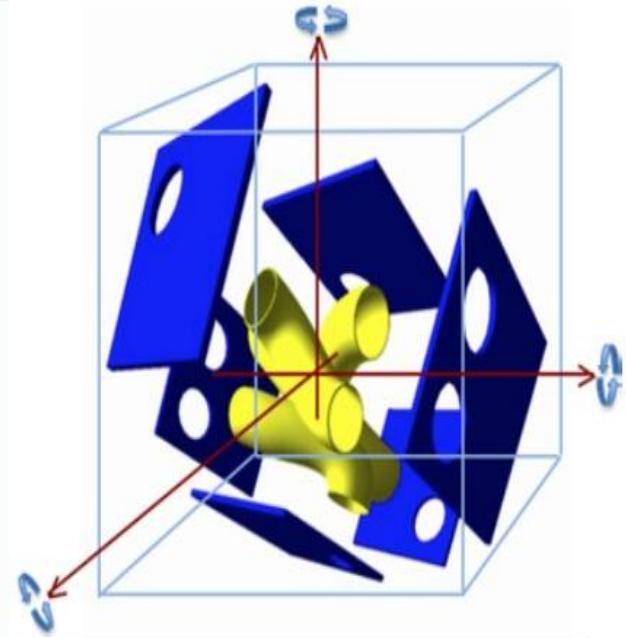
- Logical operators take boolean operands and return boolean result
- Operator ! Turns true to false and false to true
- Behavior of the operators &&, || and ^
- (1 == true, 0 == false) :



Operation					&&	&&	&&	&&	^	^	^	^
Operand1	0	0	1	1	0	0	1	1	0	0	1	1
Operand2	0	1	0	1	0	1	0	1	0	1	0	1
Result	0	1	1	1	0	0	0	1	0	1	1	0

Logical Operators - Example

```
bool a = true;  
  
bool b = false;  
  
Console.WriteLine(a && b); // False  
  
Console.WriteLine(a || b); // True  
  
Console.WriteLine(a ^ b); // True  
  
Console.WriteLine(!b); // True  
  
Console.WriteLine(b || true); // True  
  
Console.WriteLine(b && true); // False  
  
Console.WriteLine(a || true); // True  
  
Console.WriteLine(a && true); // True  
  
Console.WriteLine(!a); // False  
  
Console.WriteLine((5>7) ^ (a==b)); // False
```



Bitwise Operator - Example

- Bitwise operators are used on integer numbers

```
ushort a = 3;           // 00000000 00000011
ushort b = 5;           // 00000000 00000101
Console.WriteLine( a | b); // 00000000 00000111
Console.WriteLine( a & b); // 00000000 00000001
Console.WriteLine( a ^ b); // 00000000 00000110
Console.WriteLine(~a & b); // 00000000 00000100
Console.WriteLine( a << 1); // 00000000 00000110
Console.WriteLine( a >> 1); // 00000000 00000001
```



Bitwise Operators

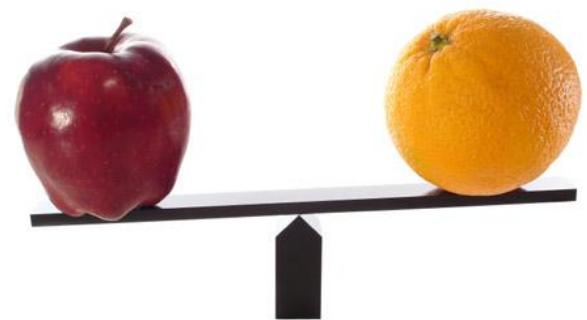
- Bitwise operator \sim turns all 0 to 1 and all 1 to 0
 - Like $!$ for boolean expressions but bit by bit
- The operators $|$, $&$ and \wedge behave like \mid , $\&\&$ and $\wedge\wedge$ for boolean expressions but bit by bit
- The $<<$ and $>>$ move bits (left to right)

Operation	$ $	$ $	$ $	$ $	$\&$	$\&$	$\&$	$\&$	\wedge	\wedge	\wedge	\wedge
Operand1	0	0	1	1	0	0	1	1	0	0	1	1
Operand2	0	1	0	1	0	1	0	1	0	1	0	1
Result	0	1	1	1	0	0	0	1	0	1	1	0

Comparison Operators

- Comparison operators are used to compare variables
 - `==`, `<`, `>`, `>=`, `<=`, `!=`
- Example

```
int a = 5;  
  
int b = 4;  
  
Console.WriteLine(a >= b); // True  
  
Console.WriteLine(a != b); // True  
  
Console.WriteLine(a == b); // False  
  
Console.WriteLine(a == a); // True  
  
Console.WriteLine(a != ++b); // False  
  
Console.WriteLine(a > b); // False
```



Assignment Operators

- Assignment operators are used to assign a value to a variable ,
— =, +=, -=, |=, ...
- Example

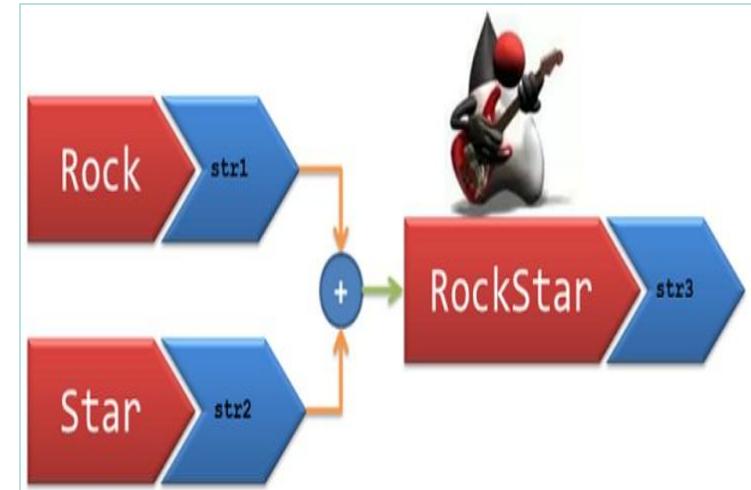
```
int x = 6;  
int y = 4;  
Console.WriteLine(y *= 2); // 8  
int z = y = 3; // y=3 and z=3  
Console.WriteLine(z); // 3  
Console.WriteLine(x |= 1); // 7  
Console.WriteLine(x += 3); // 10  
Console.WriteLine(x /= 2); // 5
```



String Concatenation Operator

- String concatenation operator + is used to concatenate strings
- If the second operand is not a string, it is converted to string automatically
- Example

```
string first = "First";  
  
string second = "Second";  
  
Console.WriteLine(first + second);  
// FirstSecond  
  
string output = "The number is : ";  
  
int number = 5;  
  
Console.WriteLine(output + number);  
  
// The number is : 5
```



Conditional Operator

- Conditional operator ?: has the form

```
b ? x : y
```

- Example

```
int input = Convert.ToInt32(Console.ReadLine());  
string classify;  
  
// ?: conditional operator.  
classify = (input < 0) ? "negative" : "positive";
```



Null-coalescing operator ??

- Null-coalescing operator ?? is used to define a default value for both nullable value types and reference types
 - It returns the left-hand operand if it is not null
 - Otherwise it returns the right operand



```
int? x = null;  
int y = x ?? -1;
```

Here the value of y is -1

```
int? x = 1;  
int y = x ?? -1;
```

Here the value of y is 1

Statements



Control Statements

- Refers to the control of the execution flow from one part of the application to another.

Selection
<ul style="list-style-type: none">• if• switch• ?:
Jump
<ul style="list-style-type: none">• break• continue• goto• return
Iteration
<ul style="list-style-type: none">• conditional (do, while)• explicit (for)• collections (foreach)

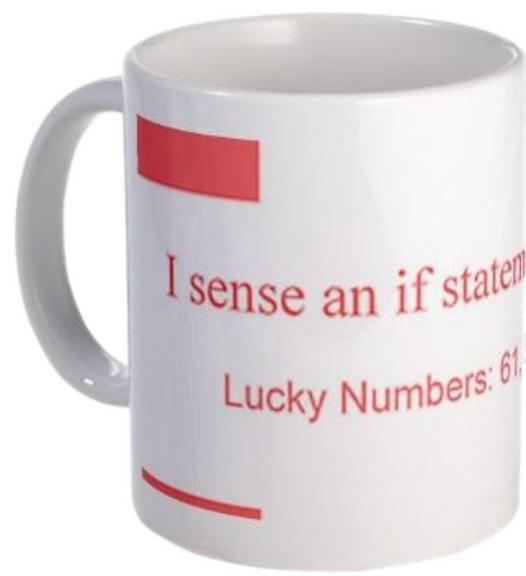


*I hit the Control key...
so why am I not in control?*

The if-else Statement

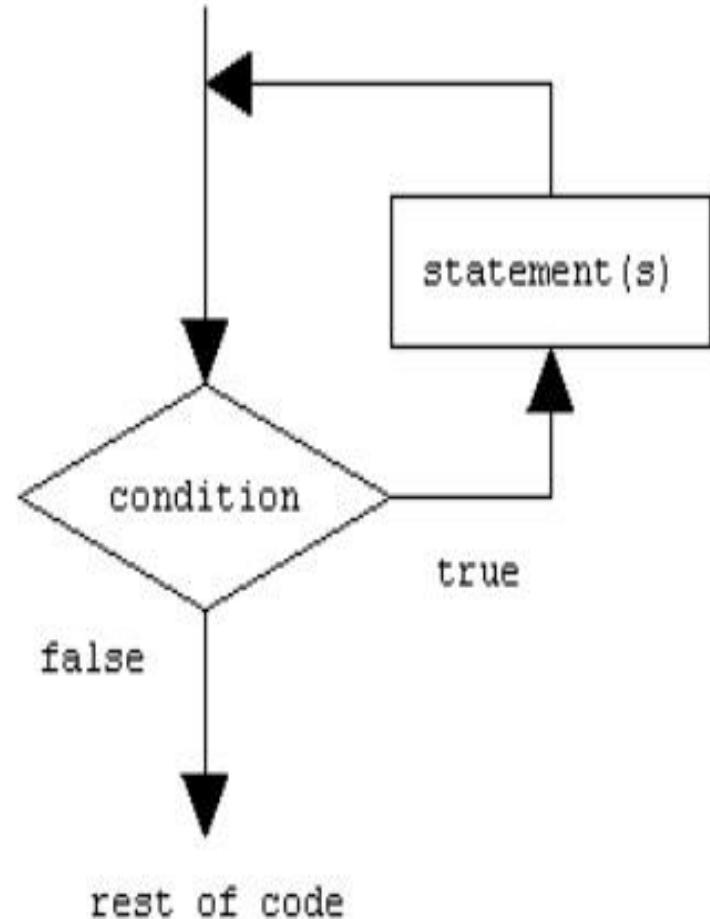
- Enables to test for a condition
- Branch to different parts of the code depending on the result
- if and if-else statements can be nested, i.e. used inside another if or else statement
- Syntax

```
if (expression)
{
    statement1;
}
else
{
    statement2;
}
```



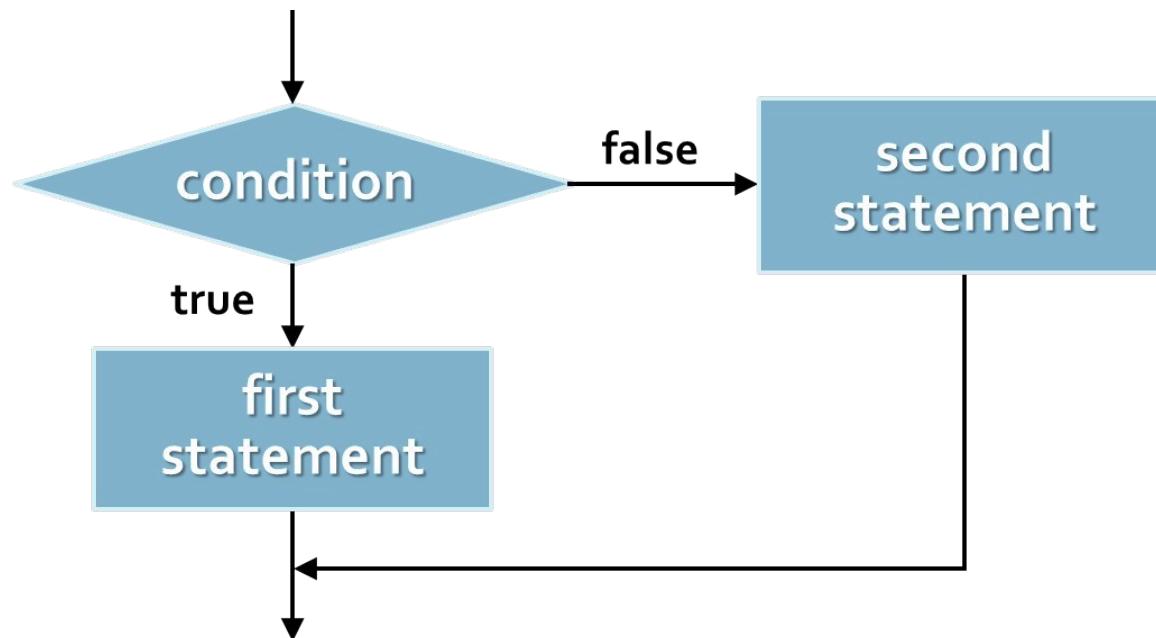
Condition and Statement

- The condition can be:
 - Boolean variable
 - Boolean logical expression
 - Comparison expression
- The condition cannot be integer variable
- The statement can be:
 - Single statement ending with a semicolon
 - Block enclosed in braces



How it works?

- The condition is evaluated
 - If it is true, the first statement is executed
 - If it is false, the second statement is executed



The if Statement – Example (1 of 3)

```
static void Main()
{
    Console.WriteLine("Enter two numbers.");

    int biggerNumber = int.Parse(Console.ReadLine());
    int smallerNumber = int.Parse(Console.ReadLine());

    if (smallerNumber > biggerNumber)
    {
        biggerNumber = smallerNumber;
    }

    Console.WriteLine("The greater number is: {0}",
                      biggerNumber);
}
```



The if Statement – Example (2 of 3)

- Checking a number if it is odd or even

```
string s = Console.ReadLine();
int number = int.Parse(s);

if (number % 2 == 0)
{
    Console.WriteLine("This number is even.");
}
else
{
    Console.WriteLine("This number is odd.");
}
```



The if Statement – Example (3 of 3)

```
if (first == second)
{
    Console.WriteLine(
        "These two numbers are equal.");
}
else
{
    if (first > second)
    {
        Console.WriteLine(
            "The first number is bigger.");
    }
    else
    {
        Console.WriteLine("The second is bigger.");
    }
}
```



The switch-case statement

- Selects for execution a statement from a list depending on the value of the switch expression
- Example

```
switch (day)
{
    case 1: Console.WriteLine("Monday"); break;
    case 2: Console.WriteLine("Tuesday"); break;
    case 3: Console.WriteLine("Wednesday");
    break;
    case 4: Console.WriteLine("Thursday"); break;
    case 5: Console.WriteLine("Friday"); break;
    case 6: Console.WriteLine("Saturday"); break;
    case 7: Console.WriteLine("Sunday"); break;
    default: Console.WriteLine("Error!"); break;
}
```



The switch-case Example

- Using multiple labels to execute the same statement in more than one case

```
switch (animal)
{
    case "dog" :
        Console.WriteLine ("MAMMAL");
        break;
    case "crocodile" :
    case "tortoise" :
    case "snake" :
        Console.WriteLine ("REPTILE");
        break;
    default :
        Console.WriteLine ("There is no such animal!");
        break;
}
```



How switch-case Works?

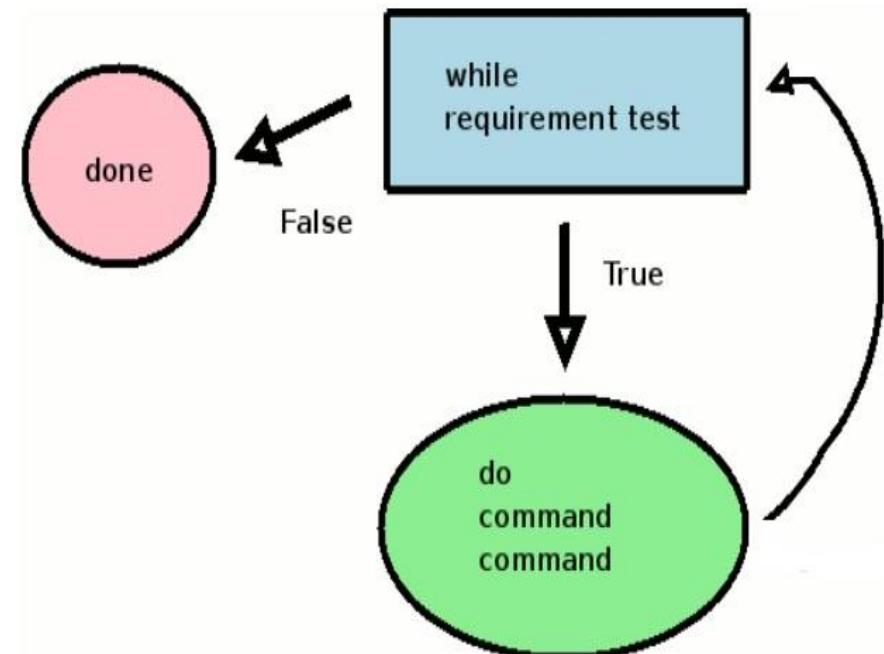
1. The expression is evaluated
2. When one of the constants specified in a case label is equal to the expression
 - The statement that corresponds to that case is executed
3. If no case is equal to the expression
 - If there is default case, it is executed
 - Otherwise the control is transferred to the end point of the switch statement



The while Loop Statement

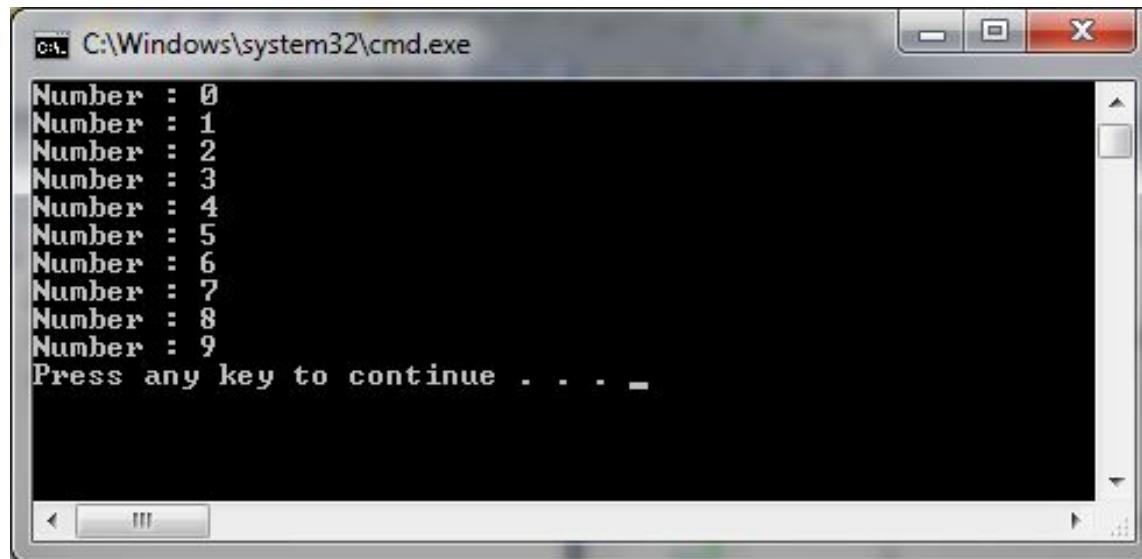
- The simplest and most frequently used loop
- The repeat condition
 - Returns a boolean result of true or false
 - Also called loop condition
- Syntax

```
while (condition)
{
    statements;
}
```

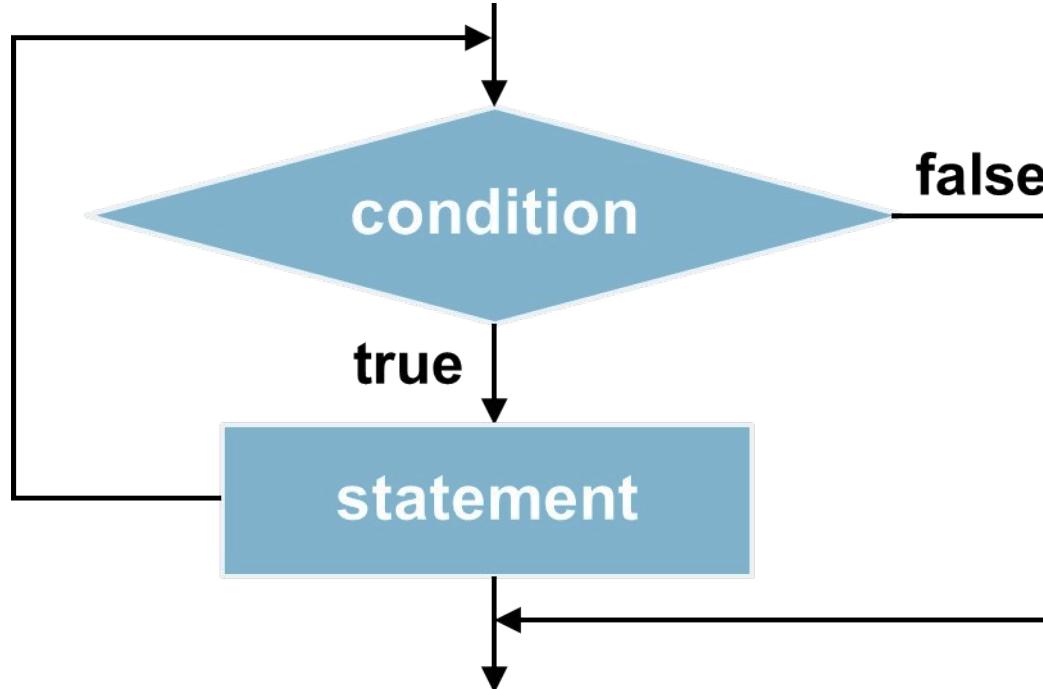


The while Loop - Example

```
int counter = 0;  
while (counter < 10)  
{  
    Console.WriteLine("Number : {0}", counter);  
    counter++;  
}
```



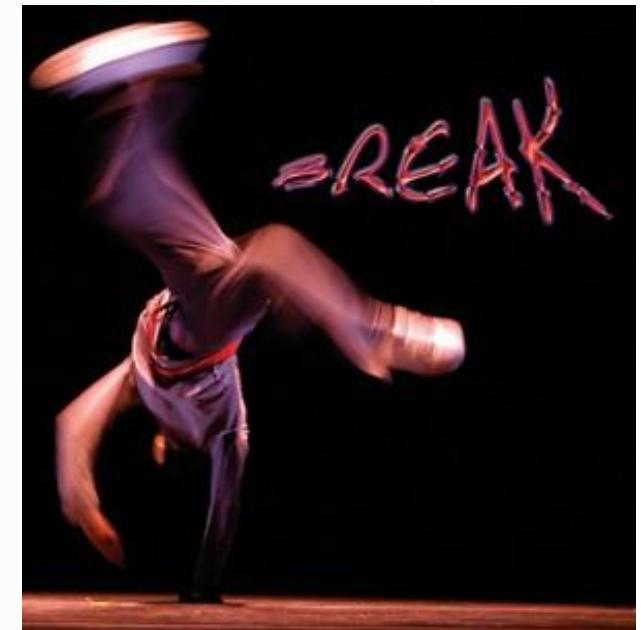
How while loop Works?



Using break Operator

- **break** operator exits the inner-most loop

```
static void Main()
{
    int n = Convert.ToInt32(Console.ReadLine());
    // Calculate n! = 1 * 2 * ... * n
    int result = 1;
    while (true)
    {
        if(n == 1)
            break;
        result *= n;
        n--;
    }
    Console.WriteLine("n! = " + result);
}
```

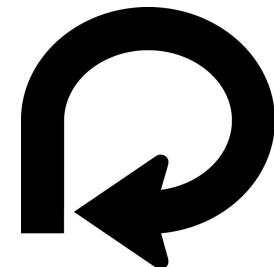


Using do-while Loop

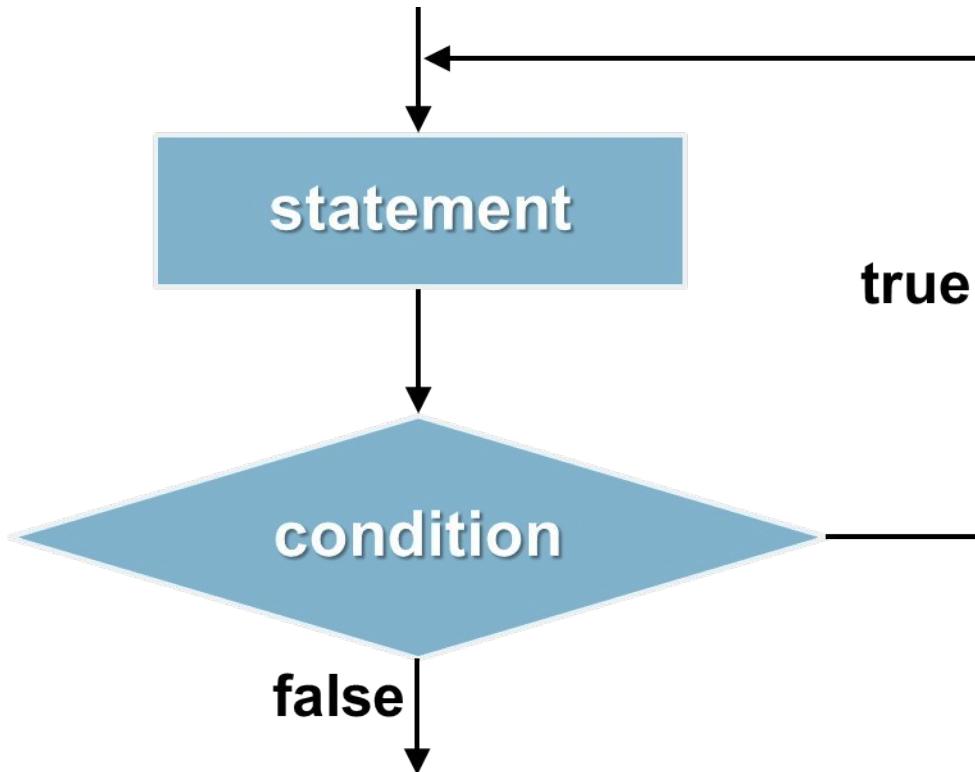
- Another loop structure is:

```
do
{
    statements;
}
while (condition);
```

- The block of statements is repeated
 - While the boolean loop condition holds
- The loop is executed at least once



How do-while statement Works?



The do-while statement - Example

- Calculating N factorial

```

static void Main()
{
    int n = Convert.ToInt32(Console.ReadLine());
    int factorial = 1;

    do
    {
        factorial *= n;
        n--;
    }
    while (n > 0);

    Console.WriteLine("n! = " + factorial);
}

```

factorial(4)=24

4*factorial(3)=24

3 *factorial(2)=6

2*factorial(1)=2

1*factorial(0)=1

The for Loop Statement

- The typical for loop syntax is:

```
for (initialization; test; update)
{
    statements;
}
```

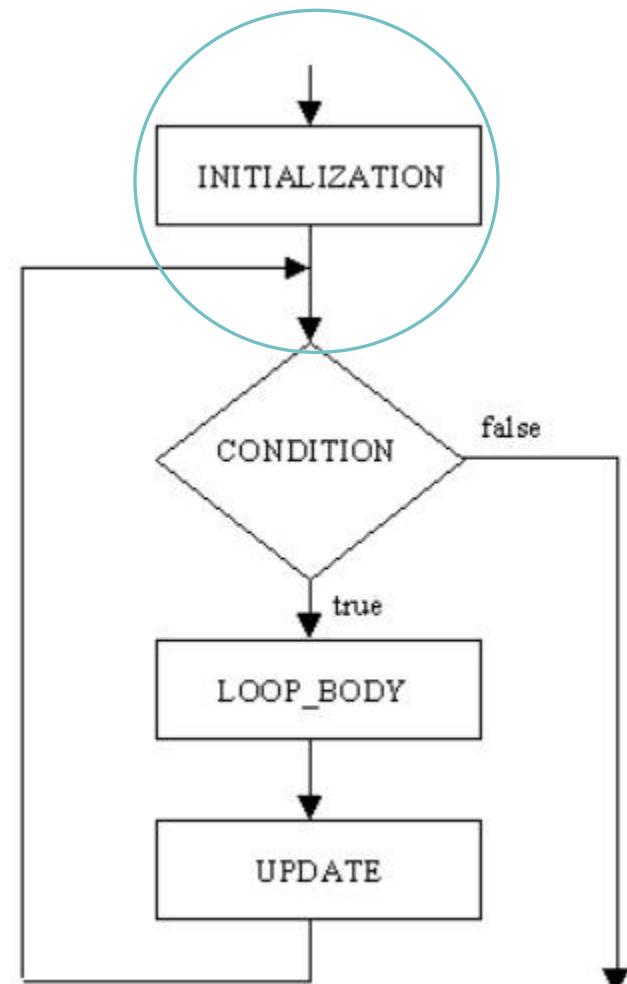
- Consists of:
 - Initialization statement
 - Boolean test expression
 - Update statement
 - Loop body block



for Loop – Initialization Expression

```
for (int number = 0; ...;  
...)  
{  
    // Can use number here  
}  
// Cannot use number here
```

- Executed once, just before the loop is entered
 - Like it is out of the loop, before it
- Usually used to declare a counter vari



for Loop – Test Expression

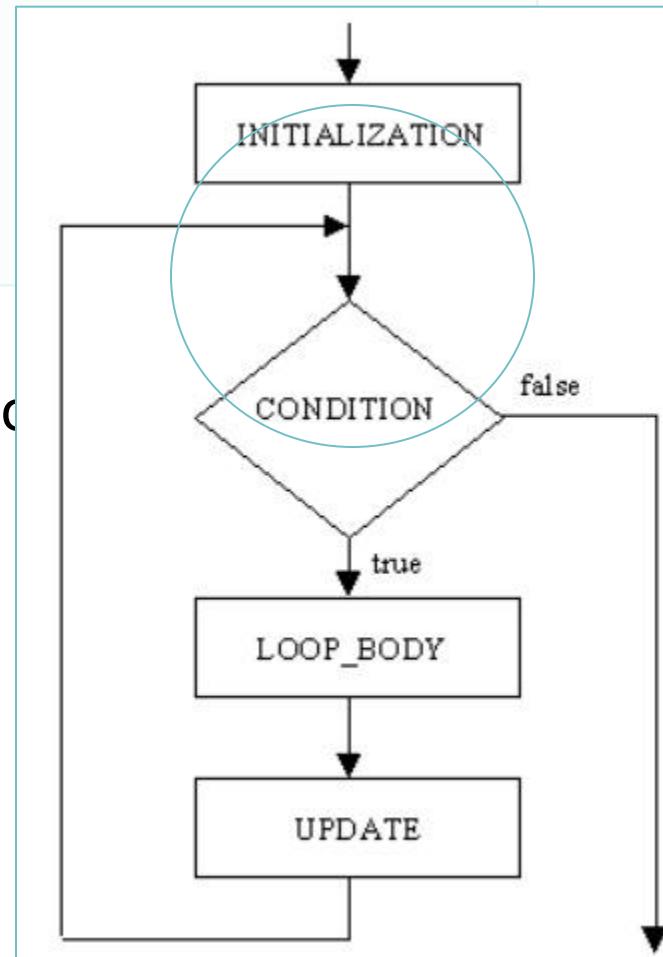
```
for (int number = 0; number < 10; ...)

{
    // Can use number here

}

// Cannot use number here
```

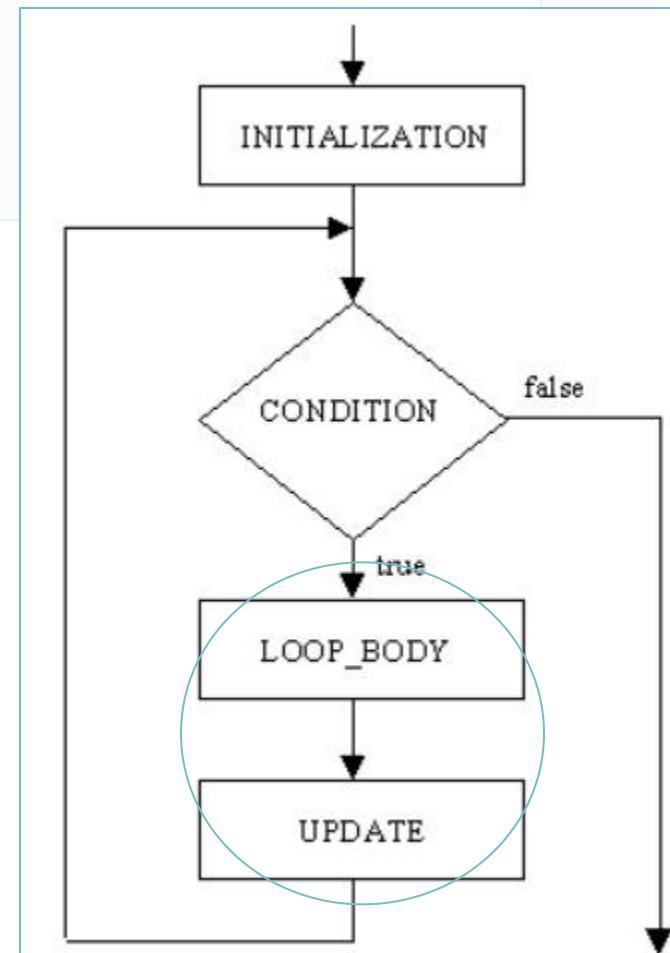
- Evaluated before each iteration of the loop
 - If true, the loop body is executed
 - If false, the loop body is skipped
- Used as a loop condition



for Loop – Update Expression

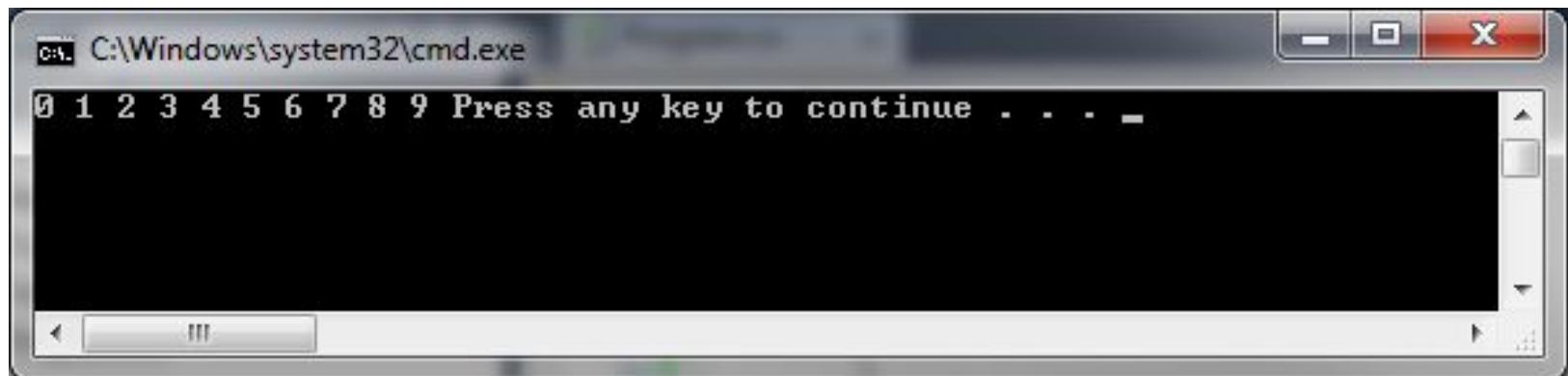
```
for (int number = 0; number < 10; number++)  
{  
    // Can use number here  
}  
// Cannot use number here
```

- Executed at each iteration after the body of the loop is finished
- Usually used to update the counter



for Loop - Example

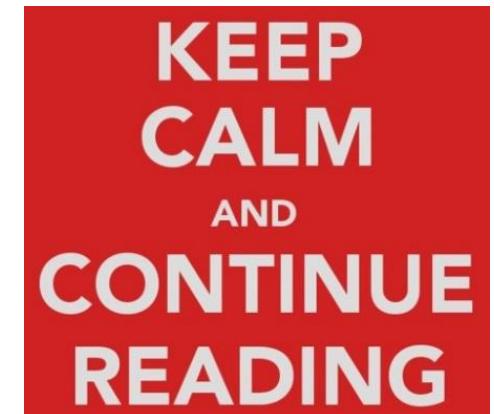
```
for (int number = 0; number < 10; number++)  
{  
    Console.WriteLine(number + " ");  
}
```



Using continue Operator

- continue operator ends the iteration of the inner-most loop
- Example

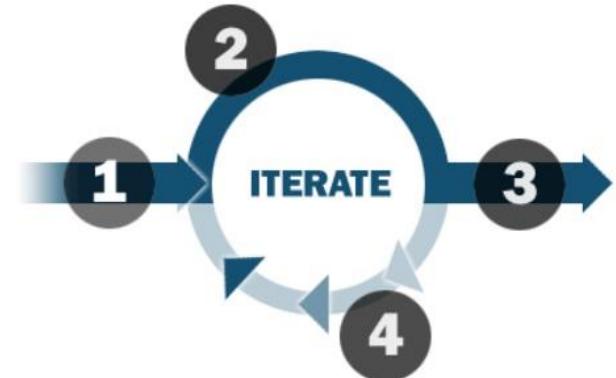
```
int n = int.Parse(Console.ReadLine());  
int sum = 0;  
for (int i = 1; i <= n; i += 2)  
{  
    if (i % 7 == 0)  
    {  
        continue;  
    }  
    sum += i;  
}  
Console.WriteLine("sum = {0}", sum);
```



The foreach Loop Statement

- The typical foreach loop syntax is:

```
foreach (Type element in collection)
{
    statements;
}
```



- Iterates over all elements of a collection
 - The element is the loop variable that takes sequentially all collection values
 - The collection can be list, array or other group of elements of the same type

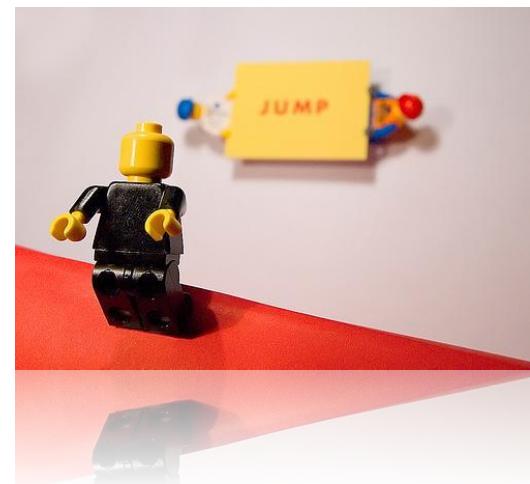
The foreach Loop Statement - Example

```
string[] days = new string[] {  
    "Monday", "Tuesday", "Wednesday", "Thursday",  
    "Friday", "Saturday", "Sunday" };  
foreach (String day in days)  
{  
    Console.WriteLine(day);  
}
```



C# Jump Statements

- Jump statements are:
 - break, continue, goto
- How continue works?
 - In while and do-while loops jumps to the test expression
 - In for loops jumps to the update expression
- To exit an inner loop use break
- To exit outer loop use goto with a label



C# Jump Statements - Example

```
int outerCounter = 0;
for (int outer = 0; outer < 10; outer++)
{
    for (int inner = 0; inner < 10;
inner++)
    {
        if (inner % 3 == 0)
            continue;
        if (outer == 7)
            break;
        if (inner + outer > 9)
            goto breakOut;
    }
    outerCounter++;
}
breakOut:
```

Label



Preprocessor Directive



Preprocessor Directives (1 of 2)

- Commands that are interpreted by the compiler and affect the output or behavior of the build process
- Do not affect your C# programs during runtime. They change how your program text is compiled
- Preprocessing directives are top lines in a C# program that start with '#'



Preprocessor Directives (2 of 2)

Directive	Description
#define, #undef	Define and undefine conditional symbols
#if, #elif, #else, #endif	Conditionally skip sections of code
#error, #warning	Issue errors and warnings
#region, #endregion	Delimit outline regions
#line, #line_default	Specify line number
#pragma	Gives the compiler special instructions for the compilation of the file in which it appears



Checkpoint

- What is the symbol used at the end of a C# expression?
 - a. Period (.)
 - b. Colon (:)
 - c. Semi-colon (;)
 - d. Slash (/)



Checkpoint

Checkpoint

- Which of the following is not a C# comparison operator?
 - =>
 - !=
 - ==
 - <
- What is the output of the following code segment?

```
int a = 3, b = 4;  
if(a > b)  
    Console.WriteLine("Black ");  
else  
    Console.WriteLine("White");
```

- Black**
- White
- Black White
- Nothing



Checkpoint

Checkpoint

- What is the output of the following code segment?

```
int j = 5;  
while(j > 0)  
{  
    Console.WriteLine("{0} ", j);  
    j--;  
}
```

- a. 0
- b. 5
- c. 5 4 3 2 1
- d. 5 4 3 2 1 0



Checkpoint

Checkpoint

- What is not a preprocessor directive?
 - a. #define
 - b. #region
 - c. #warning
 - d. #pragmatic



Checkpoint

Knowledge Check



Knowledge Check

1. What is the output of the following code segment?

```
int a = 3, b = 4;  
if(a == b)  
    Console.WriteLine("Black ");  
    Console.WriteLine("White");
```

- a. Black
- b. White
- c. Black White
- d. Nothing

Knowledge Check

2. If the following code segment compiles correctly, what do you know about the variable x? int a = 3, b = 4;
 if(x) Console.WriteLine("OK");
- a. x is an integer variable
 - b. x is a Boolean variable
 - c. x is greater than 0
 - d. none of these

Knowledge Check

3. When a loop is placed within another loop, the loops are said to be:
 - a. Infinite
 - b. Bubbled
 - c. Nested
 - d. Overlapping
4. Assume you have two variables declared as int var1 = 3; and int int var2 = 8;. Which of the following would display 838?
 - a. Console.WriteLine("{0}{1}{2}", var1, var2);
 - b. Console.WriteLine("{0}{1}{0}", var1, var2);
 - c. Console.WriteLine("{0}{1}{2}", var2, var1);
 - d. Console.WriteLine("{0}{1}{0}", var2, var1);

Knowledge Check

5. Assume you have a variable declared as int var1 = 3;. If var2 = ++var1, what is the value of var2?
 - a. 2
 - b. 3
 - c. 4
 - d. 5

6. Assume you have a variable declared as int var1 = 3;. If var2 = var1++, what is the value of var2?
 - a. 2
 - b. 3
 - c. 4
 - d. 5

Knowledge Check

7. Which of the following is not a C# comparison operator?
 - a. `=>`
 - b. `!=`
 - c. `==`
 - d. `<`
8. Assume you have declared a variable as double `salary = 45000.00`; Which of the following will display \$45,000?
 - a. `Console.WriteLine(salary.ToString("c0"));`
 - b. `Console.WriteLine(salary.ToString("c"));`
 - c. `Console.WriteLine(salary);`
 - d. None of the above

Knowledge Check

9. Which of the following compares two string variables named string1 and string2 to determine if their contents are equal?
 - a. Console.WriteLine(salary.ToString("c0"));
 - b. Console.WriteLine(salary.ToString("c"));
 - c. Console.WriteLine(salary);
 - d. Two of these
10. Falling through a switch case is most often prevented by using the:
 - a. end statement
 - b. default statement
 - c. case statement
 - d. break statement

Knowledge Check

11. Which of the following expressions is equivalent to the following code segment?

if (g > h)

if(g < k)

Console.WriteLine("Brown");

- a. if(g > h && g < k) Console.WriteLine("Brown");
- b. if(g > h && < k) Console.WriteLine("Brown");
- c. if(g > h || g < k) Console.WriteLine("Brown");
- d. two of these

Knowledge Check

12. How many case labels would a switch statement require to be equivalent to the following if statement?

```
if(v == 1)
    Console.WriteLine("one");
else
    Console.WriteLine("two");
```

- a. zero
- b. one
- c. two
- d. impossible to tell

Knowledge Check

13. If the test expression in a switch does not match any of the case values, and there is no default value, then:
 - a. a compiler error occurs
 - b. a run-time error occurs
 - c. the program continues with the next executable statement
 - d. the expression is incremented and the case values are tested again
14. Which of the following is not required of a loop control variable in a correctly working loop?
 - a. It is initialized before the loop starts.
 - b. It is tested.
 - c. It is reset to its initial value before the loop ends.
 - d. It is altered in the loop body.

Knowledge Check

15. Which of the following is equivalent to the statement `if(m == 0) d = 0 ; else d = 1;?`
- a. `? m == 0 : d = 0, d = 1;`
 - b. `m? d = 0; d = 1;`
 - c. `m == 0 ; d = 0; d = 1?`
 - d. `m == 0 ? d = 0 : d = 1;`

Module 6: Essential Data Structures



Module Objectives

At the end of this module, you will be able to:

- Create and use arrays and jagged arrays.
- Describe how to use indexers to provide access to data through an array-like syntax.
- Define and use enumerations
- Describe how to create and use structures.



Module objectives

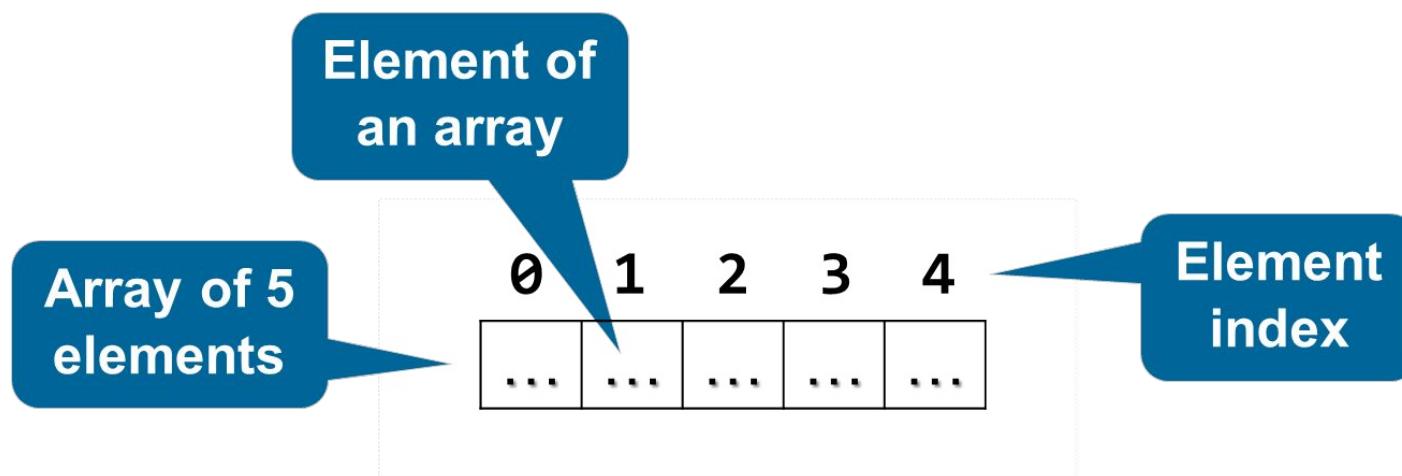
Module Agenda

- Arrays
- Indexers
- Enumerations
- Structures



What are Arrays?

- An array is a sequence of elements
 - All elements are of the same type
 - The order of the elements is fixed
 - Has fixed size (Array.Length)



Declaring Arrays

- Declaration defines the type of the elements
- Square brackets [] mean “array”
- Examples
 - Declaring array of integers:

```
int[] myIntArray;
```

- Declaring array of strings:

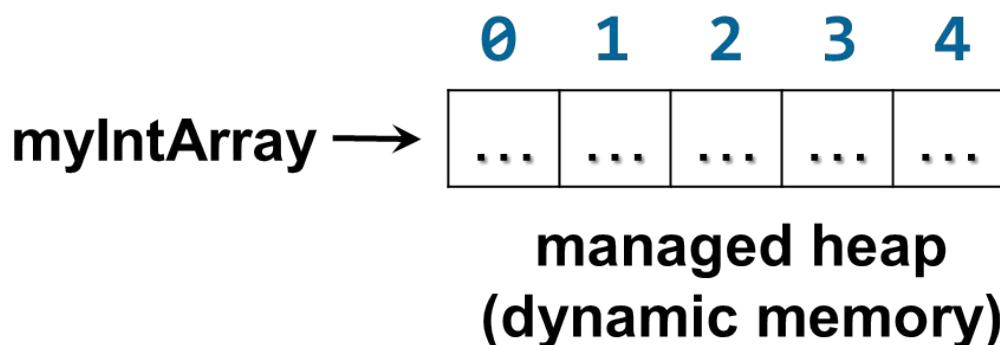
```
string[] myStringArray;
```



Creating Arrays

- Use the operator **new**
 - Specify array length
- Example of creating(allocating) array of 5 integers:

```
myIntArray = new int[5];
```



Reverse Array Example

```
int[] array = new int[] {1, 2, 3, 4, 5};  
  
// Get array size  
int length = array.Length;  
  
// Declare and create the reversed array  
int[] reversed = new int[length];  
  
// Initialize the reversed array  
for (int index = 0; index < length; index++)  
{  
    reversed[length-index-1] = array[index];  
}
```



Processing Arrays: foreach Statement

- foreach loop syntax:

```
foreach (type value in array)
```

- type – the type of the element
- value – local name of variable
- array – processing array

- Used when no indexing is needed
 - All elements are accessed one by one
 - Elements cannot be modified (read-only)



Accessing Array Elements

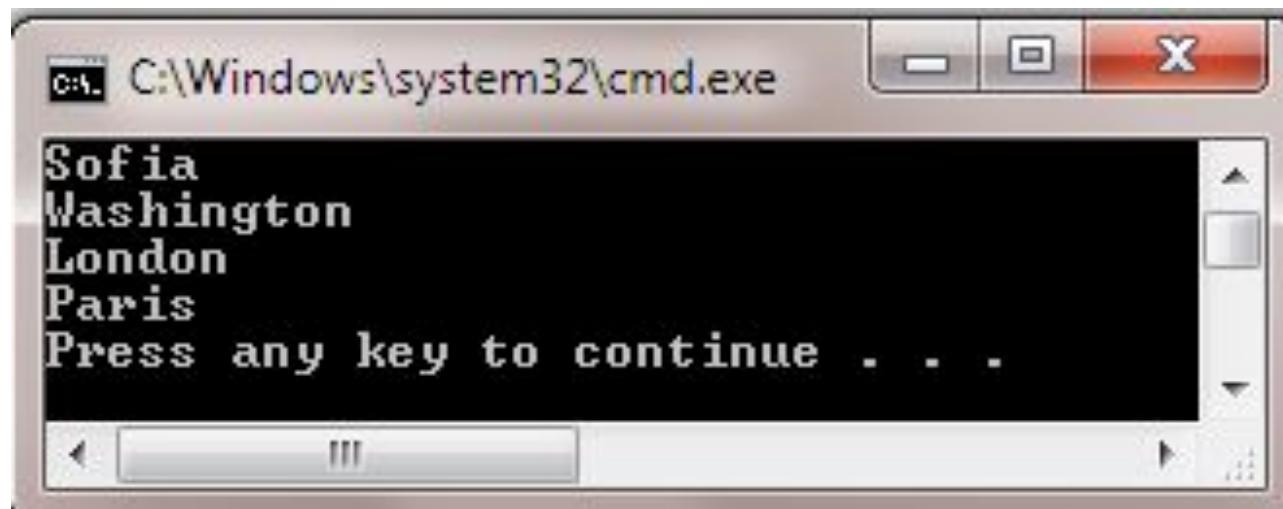
- Array elements are accessed using the square brackets operator [] (indexer)
 - Array indexer takes element's index as parameter
 - The first element has index 0
 - The last element has index Length – 1
- Array elements can be retrieved and changed by the [] operator



foreach - Example

- Print all elements of a string[] array:

```
string[] capitals =
{
    "Sofia",
    "Washington",
    "London",
    "Paris"
};
foreach (string capital in capitals)
{
    Console.WriteLine(capital);
}
```



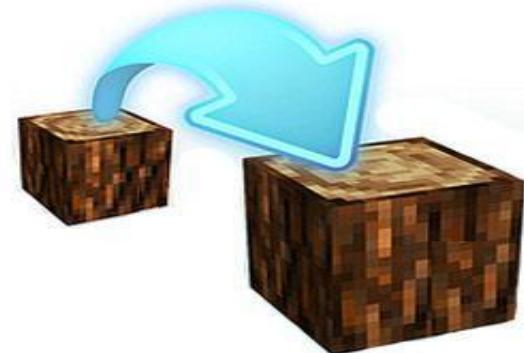
Copying Arrays

- Changing some of the values in one array will affect the other

```
int[] copyArray = array;
```

- Only the values will be copied but not the reference

```
int[] copyArray = (int[])array.Clone();
```



What is Multidimensional Array?

- Multidimensional arrays have more than one dimension also known as matrices or tables
 - Matrices are represented by a list of rows
 - Rows consist of list of values
- Example:

A three-dimensional Array (3 by 2 by 3)

Index [0] [0] [0]	Index [0] [1] [0]	Index [1] [1] [1]	Index [2] [0] [1]	Index [2] [1] [2]
12	120	-87	4	334
0	212	67	33	21
176	56	8	134	-17



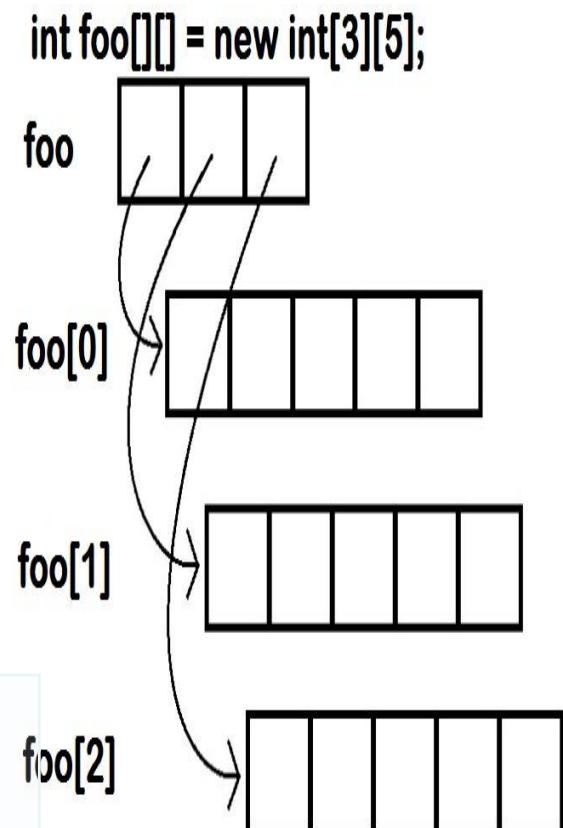
Declaring and Creating Multidimensional Arrays

- Declaring multidimensional arrays:

```
int[,] intMatrix;
float[,] floatMatrix;
string[,,] strCube;
```

- Creating a multidimensional array
 - Use new keyword
 - Must specify the size of each dimension

```
int[,] intMatrix = new int[3, 4];
float[,] floatMatrix = new float[8, 2];
string[,,] stringCube = new string[5, 5, 5];
```



Initializing Multidimensional Arrays with Values

- Creating and initializing with values multidimensional array:

```
int[,] matrix =  
{  
    {1, 2, 3, 4}, // row 0 values  
    {5, 6, 7, 8}, // row 1 values  
}; // The matrix size is 2 x 4 (2 rows, 4
```



Accessing the elements of Multidimensional Arrays

- Accessing N-dimensional array element:

```
nDimensionalArray[index1, ... , indexn]
```

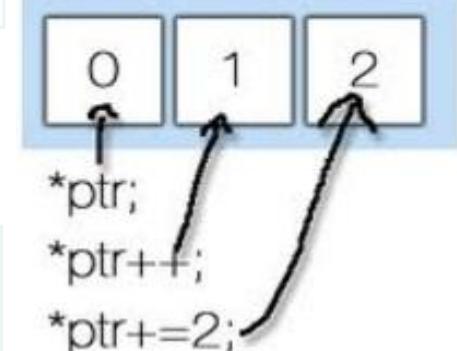
- Getting element value example:

```
int[,] array = {{1, 2}, {3, 4}}
int element11 = array[1, 1]; // element11 = 4
```

- Setting element value example:

```
int[,] array = new int[3, 4];
for (int row=0; row<array.GetLength(0); row++)
    for (int col=0; col<array.GetLength(1); col++)
        array[row, col] = row + col;
```

```
int array[3] = {0,1,2};
int *ptr = array;
```



Number
of rows

Number of
columns

Reading a Matrix - Example

- Reading a matrix from the console

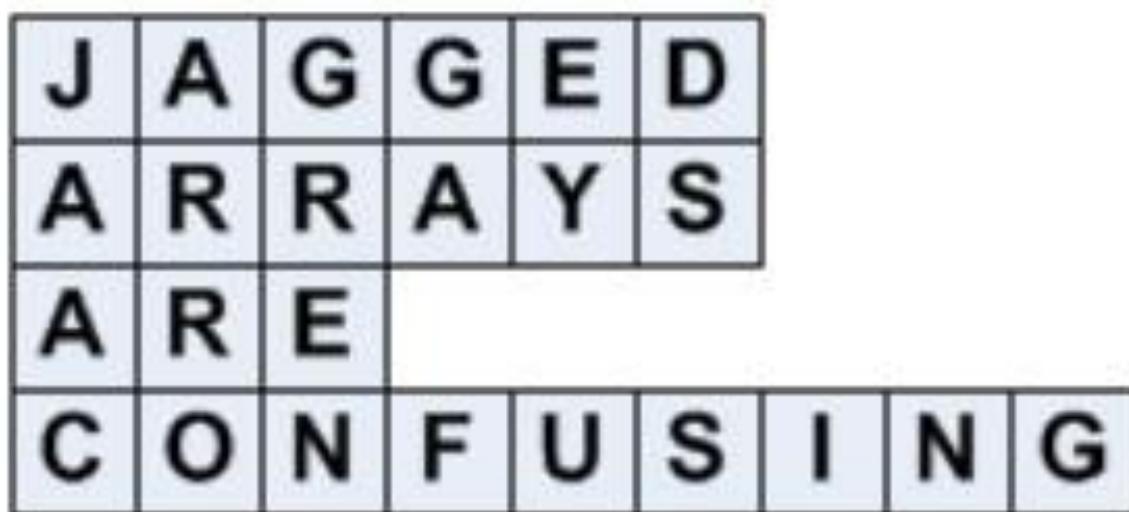
```
int rows = int.Parse(Console.ReadLine());  
int columns = int.Parse(Console.ReadLine());  
int[,] matrix = new int[rows, columns];  
String inputNumber;  
for (int row=0; row<rows; row++)  
{  
    for (int column=0; column<cols; column++)  
    {  
        Console.Write("matrix[{0},{1}] = ", row, column);  
        inputNumber = Console.ReadLine();  
        matrix[row, column] = int.Parse(inputNumber);  
    }  
}
```

A Matrix

1	2	3
4	5	6
7	8	9

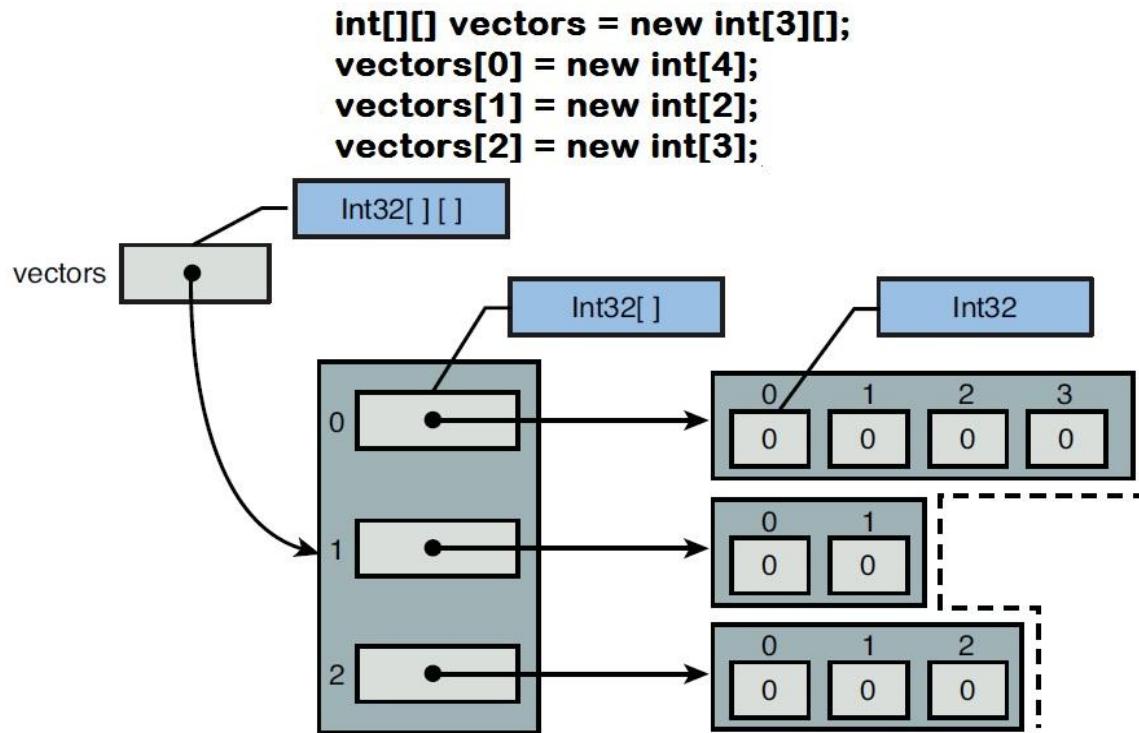
What are Jagged Arrays?

- Jagged Arrays are like multidimensional arrays
 - But each dimension has different size
 - A jagged array is arrays of arrays
 - Each of the array has different length



How to create a Jagged Array?

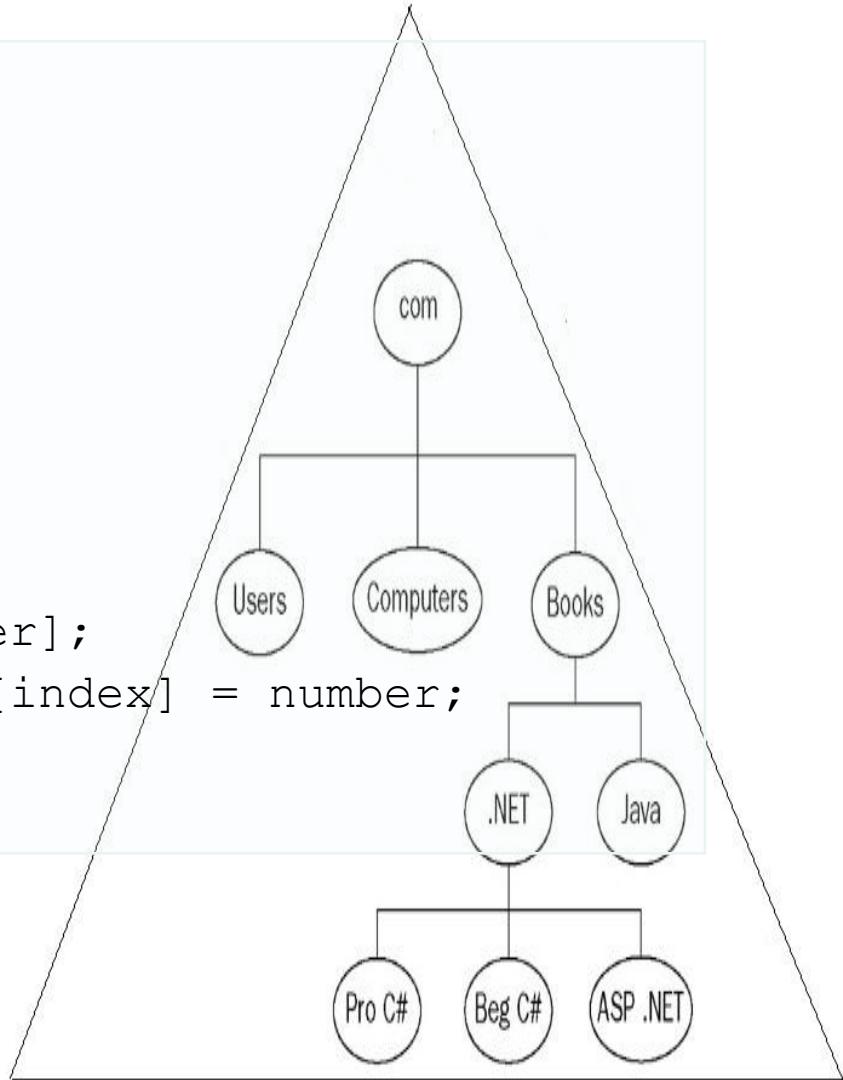
- When creating jagged arrays
 - The array is created with null arrays
 - Initialized each of the array



Jagged Array - Example

```
foreach (var number in numbers)
{
    int remainder = number % 3;
    sizes[remainder]++;
}

...
foreach (var number in numbers)
{
    int remainder = number % 3;
    int index = positions[remainder];
    numbersByRemainder[remainder][index] = number;
    positions[remainder]++;
}
```



Manipulating Arrays

- **Array Methods for manipulating the elements within the array:**
 - Sort()
 - BinarySearch()
 - Reverse()
 - Clear()
- **Use Sort() Method before using BinarySearch() method.**
- **Use Clear() Method to set each element in the array to its default value either false, 0, or null depending on data type.**
- **Retrieving a particular dimension size:**

```
bool[,,] cells;  
cells = new bool[2,3,3];  
System.Console.WriteLine(cells.GetLength(0));
```



Indexers



Indexer

- Indexers encapsulate an internal collection or array.
- The this keyword is used to define the indexers.
- The value keyword is used to define the value being assigned by the set indexer.
- A noninteger index is referred to as a key.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.



Indexer - Example

```
class MyArray
{
    public string this[int index]
    {
        get
        {
            return array[index];
        }
        set
        {
            array[index] = value;
        }
    }
}
```

```
s = myArray[i];
myArray[i] = "some string";
```

A large, stylized word "this" is composed of multiple smaller, overlapping letters "t". The letters are arranged in a staggered, overlapping fashion, creating a sense of depth and texture. The color of the letters transitions from a dark blue at the top to a light blue at the bottom.

Array vs. Indexer

	Arrays	Indexers
Subscripts	Must use a numeric subscript to access individual members	Can use nonnumeric subscripts, for example, strings to access individual members
Overloading	Cannot be overloaded in child classes	Can be overloaded in child classes
Method parameters	Can be used as a normal parameter and a ref or out parameter	Can only be used as a normal parameter; cannot be used as a ref or out parameter

Enumerations



What are Enumerations?

- Enumerations in C# are types whose values are limited to a predefined set of values
 - Declared by the keyword enum in C#
 - Hold values from a predefined set
- Example

```
public enum Color { Red, Green, Blue, Black }

...
Color color = Color.Red;
Console.WriteLine(color); // Red
color = 5; // Compilation error!
```



enums - Example

```
public class EnumTest
{
    enum Days {Sat = 1, Sun, Mon, Tue, Wed, Thu, Fri};

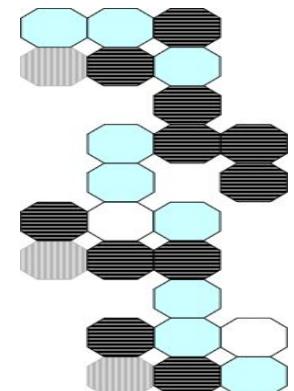
    public static void Main()
    {
        int x = (int) Days.Sun;
        int y = (int) Days.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
```

Structures



What are Structures?

- Structures in C# are similar to classes
 - Structures are value types (directly hold a value)
 - Classes are reference types (pointers)
- Structures are usually used for storing data structures, without any other functionality
- Structures can have fields, properties, etc.
 - Using methods is not recommended
- Example of structure
 - **System.DateTime** – represents a date and time



When to use Structures?

- Use structures
 - To make your type behave as a primitive type
 - If you create many instances and after that you free them –
e.g. in a cycle
- Do not use structures
 - When you often transmit your instances as method parameters
 - If you use collections without generics(too much boxing/unboxing)

Structures - Example

```
struct Point
{
    public int X, Y;
}

struct Color
{
    public byte redValue;
    public byte greenValue;
    public byte blueValue;
}

struct Square
{
    public Point location;
    public int size;
    public Color borderColor;
    public Color surfaceColor;
}
```

Checkpoint

- Which of the following is the correct output of the C#.NET code snippet given below?

```
int[ , , ] a = new int[ 3, 2, 3 ];  
Console.WriteLine(a.Length);
```

- a. 18
- b. 20
- c. 4
- d. 3



Checkpoint

Checkpoint

- Which of the following statements is correct about the array declaration given below?

```
int[][][] intMyArr = new int[2][][];
```

- a. *intMyArr* refers to a 2-D jagged array containing 2 rows.
- b. *intMyArr* refers to a 2-D jagged array containing 3 rows.
- c. *intMyArr* refers to a 3-D jagged array containing 2 2-D jagged arrays.
- d. *intMyArr* refers to a 3-D jagged array containing three 2-D jagged arrays.
- e. *intMyArr* refers to a 3-D jagged array containing 2 2-D rectangular arrays.



Checkpoint

Checkpoint

- What term is used within the implementation of an indexer's setter to access the parameter being passed to the indexer?
 - a. This
 - b. Arg
 - c. Indexer
 - d. value



Checkpoint

Checkpoint

- Which of the following will be the correct output for the C#.NET code snippet given below?

```
enum color : int { red = -3, green, blue }
Console.WriteLine( (int) color.red + ", " );
Console.WriteLine( (int) color.green + ", " );
Console.WriteLine( (int) color.blue );
```

- a. -3, -2, -1
- b. -3, 0, 1
- c. red, green, blue
- d. color.red, color.green, color.blue



Checkpoint

Checkpoint

- Which of the following is the correct way to define a variable of the type **struct Emp** declared below?

```
struct Emp {  
    private String name;  
    private int age;  
    private Single sal;  
}
```

1. Emp e(); e = new Emp();
 2. Emp e = new Emp;
 3. Emp e; e = new Emp;
 4. Emp e = new Emp();
 5. Emp e;
- b. 1, 3
- c. 2, 5
- d. 4, 5



Checkpoint

Knowledge Check



Knowledge Check

1. Arrays can be declared to hold values of _____.
 - a. type double
 - b. type int
 - c. type string
 - d. any type

2. The elements of an array are related by the fact that they have the same _____.
 - a. constant value
 - b. subscript
 - c. type
 - d. value

Knowledge Check

3. Method returns the largest index in the array.
 - a. GetUpperBound
 - b. GetUpperLimit
 - c. GetLargestIndex
 - d. GetUpperSubscript
4. The first element in every array is the _____.
 - a. subscript
 - b. zeroth element
 - c. length of the array
 - d. smallest value in the array

Knowledge Check

5. Arrays _____.
- a. are controls
 - b. always have one dimension
 - c. keep data in sorted order at all times
 - d. are objects
6. The initializer list can _____.
- a. be used to determine the size of the array
 - b. contain a comma-separated list of initial values for the array elements
 - c. be empty
 - d. All of the above

Knowledge Check

7. Which method call sorts array strWords in ascending order?
 - a. Array.Sort(strWords)
 - b. strWords.SortArray()
 - c. Array.Sort(strWords, 1)
 - d. Sort(strWords)
8. To search for a period (.) in a string called strTest, call method
 - a. String.Search(strTest, ".")
 - b. String.IndexOf(strTest, ".")
 - c. strTest.IndexOf(":")
 - d. strTest.Search(":")

Knowledge Check

9. Property _____ contains the size of an array.
- a. Elements
 - b. ArraySize
 - c. Length
 - d. Size
10. Which of the following statements is true about an *enum* used in C#.NET?
- a. An implicit cast is needed to convert from *enum* type to an integral type.
 - b. An *enum* variable cannot have a *public* access modifier.
 - c. An *enum* variable cannot have a *private* access modifier.
 - d. An *enum* variable can be defined inside a class or a namespace.
 - e. An *enum* variable cannot have a *protected* access modifier.

Knowledge Check

11. Which of the following statements are correct about an *enum* used in C#.NET?
1. By default the first enumerator has the value equal to the number of elements present in the list.
 2. The value of each successive enumerator is decreased by 1.
 3. An enumerator contains white space in its name.
 4. A variable cannot be assigned to an *enum* element.
 5. Values of *enum* elements cannot be populated from a database.
- b. 1, 2
- c. 3, 4
- d. 4, 5
- e. 1, 4

Knowledge Check

12. Which of the following statements is correct about the C#.NET code snippet given below?

```
int a = 10;  
int b = 20;  
int c = 30;  
enum color: byte {  
    red = a,  
    green = b,  
    blue = c }
```

- a. Variables cannot be assigned to *enum* elements.
- b. Variables can be assigned to any one of the *enum* elements.
- c. Variables can be assigned only to the first *enum* element.
- d. Values assigned to *enum* elements must always be successive values.

Knowledge Check

13. Which of the following statements are correct about an *enum* used in C#.NET?
1. To use the keyword *enum*, we should either use [*enum*] or *System.Enum*.
 2. *enum* is a keyword.
 3. *Enum* is class declared in *System.Type* namespace.
 4. *Enum* is a class declared in the current project's root namespace.
 5. *Enum* is a class declared in *System* namespace.
- b. 1, 3
- c. 2, 4
- d. 2, 5
- e. 3, 4

Knowledge Check

14. Which of the following statements is correct about the C#.NET code snippet given below?

```
enum per{  
    married,  
    unmarried,  
    divorced,  
    spinster }  
per.married = 10;  
Console.WriteLine(per.unmarried);
```

- a. The program will output a value 11.
- b. The *enum* elements must be declared *private*.
- c. The program will output a value 2.
- d. The program will report an error since an *enum* element cannot be assigned a value outside the *enum* declaration.

Knowledge Check

15. An *enum* that is declared inside a class, struct, namespace or interface is treated as public.
- a. True
 - b. False
 - c. It depends

Module 7: Using Classes and Objects



Module Objectives

- Become familiar with the components of a class
- Describe how to create and invoke methods.
- Describe how to define a constructor.
- Describe how Visual C# uses namespaces



Module objectives

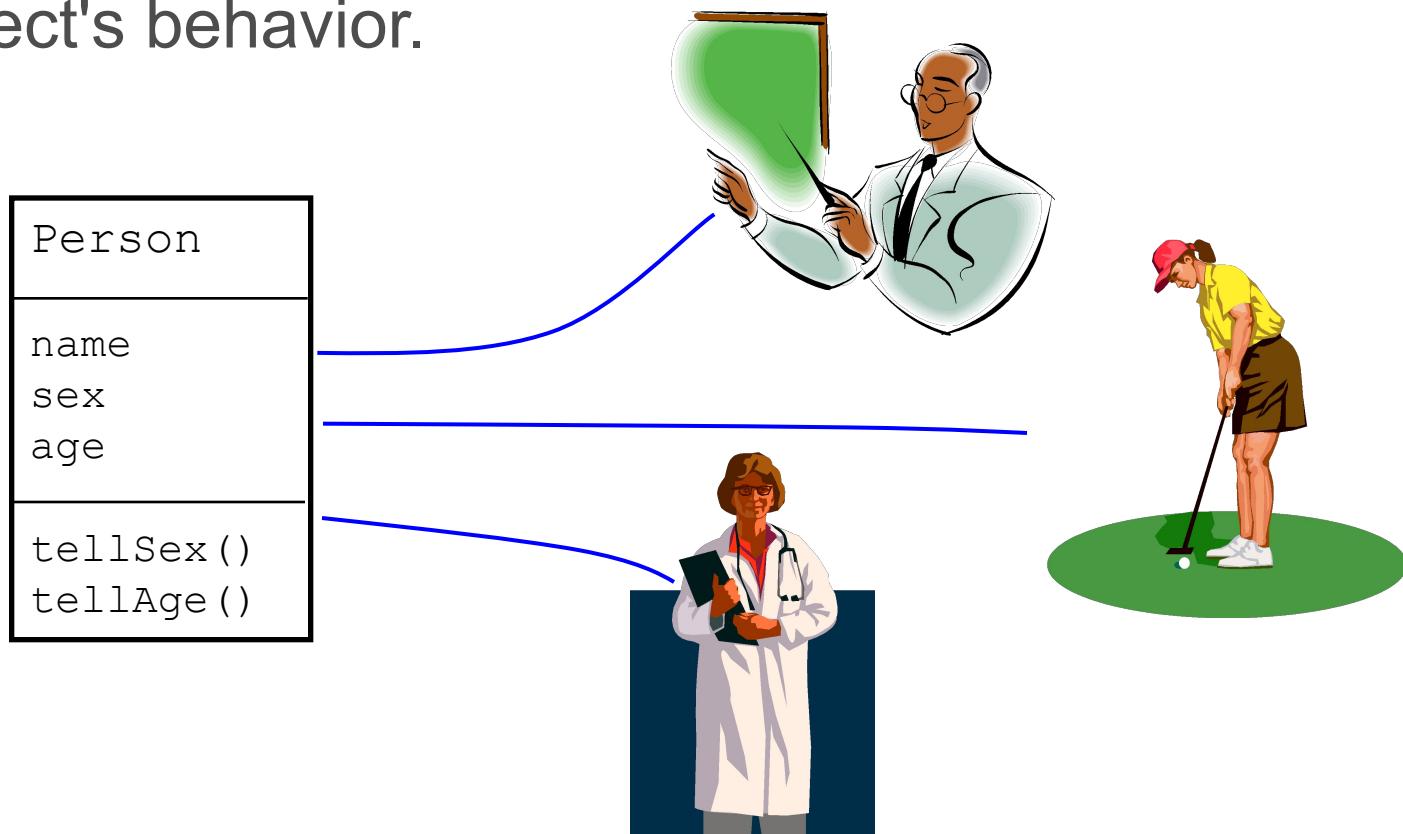
Module Agenda

- Classes and Objects
- Methods
- Constructors

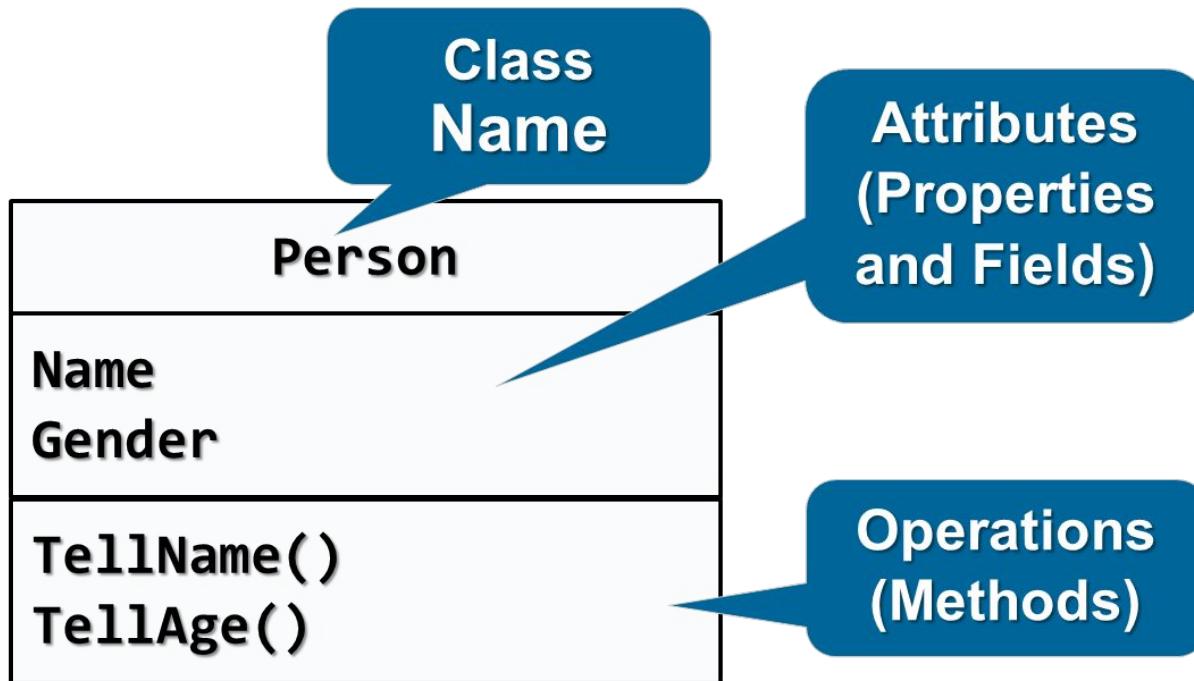


What is a Class?

- Acts as the template from which an instance of an object is created. The class defines the properties of the object and the methods used to control the object's behavior.



Class - Example



Class Definition

Begin of class definition

```
public class Cat : Animal
{
    private string name;
    private string owner;

    public Cat(string name, string owner)
    {
        this.name = name;
        this.owner = owner;
    }

    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }

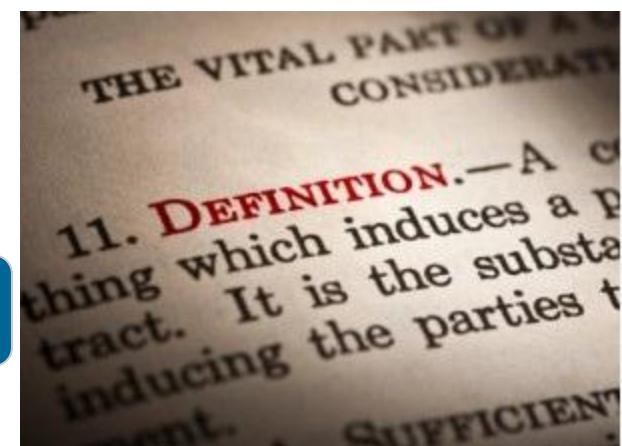
    public void SayMeow()
    {
        Console.WriteLine("Meow");
    }
}
```

Fields

Constructor

Property

Method



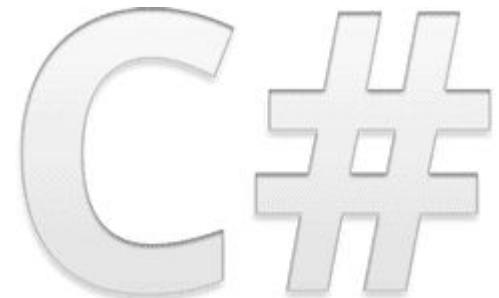
Classes in C#

- Classes are basic units that compose programs
- Class definition consists of:
 - Class declaration
 - Inherited class or implemented interfaces
 - Fields
 - Constructors
 - Properties
 - Methods
 - Events, Inner types, etc.



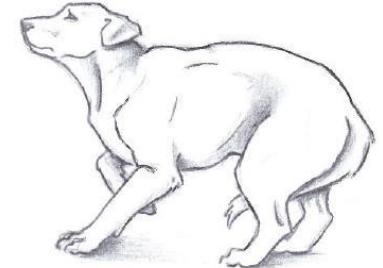
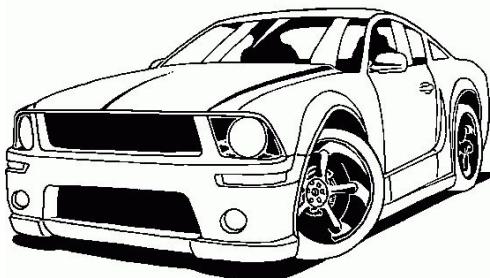
Classes in C# - Example

- Examples of classes:
 - `System.Console`
 - `System.String` (`string` in C#)
 - `System.Int32` (`int` in C#)
 - `System.Array`
 - `System.Math`
 - `System.Random`
 - `System.DateTime`
 - `System.Collections.Generics.List<T>`



Everything is an Object

- Anything that you can describe can be represented as an object, and that representation can be created, manipulated and destroyed to represent how you use the real object that it models.



What is an Object?

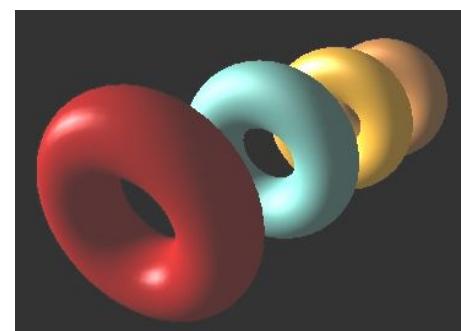
An **object** is a self-contained entity
with **attributes** and **behaviors**

Information an object must know:

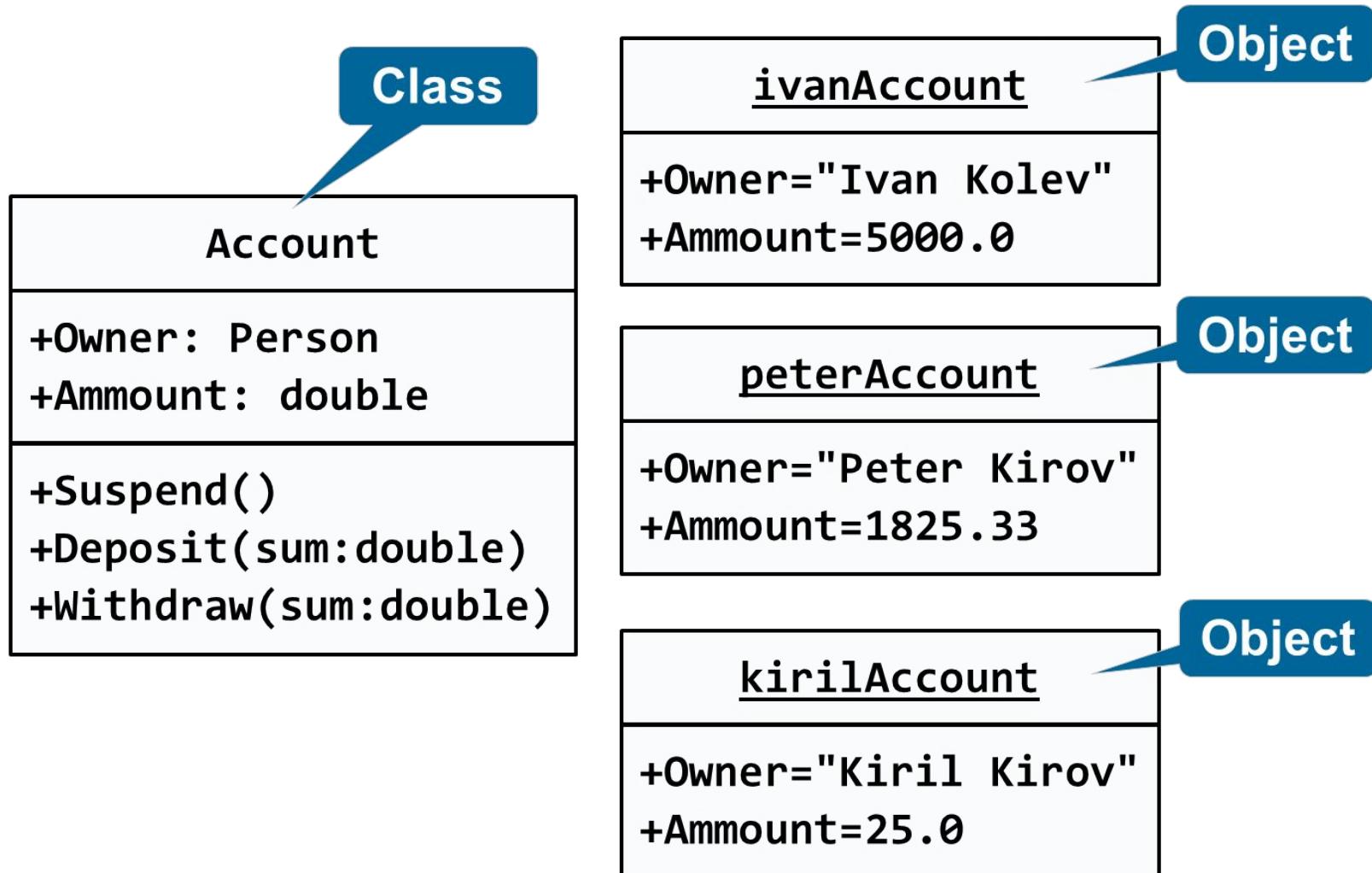
- Identity – uniqueness
- Attributes – structure
- State – current condition

Behavior an object must do:

- Methods – what it can do
- Events – what it responds to



Objects - Example



Object – Creating an Instance of a Class

- An instance of a class or structure can be defined like any other variable
- Instances cannot be used if they are not initialized
- To create a new operator is use to create a new instance of class
- The new operator does two things:
 - Causes the CLR to allocate memory for the object
 - Invokes a constructor to initialize the object

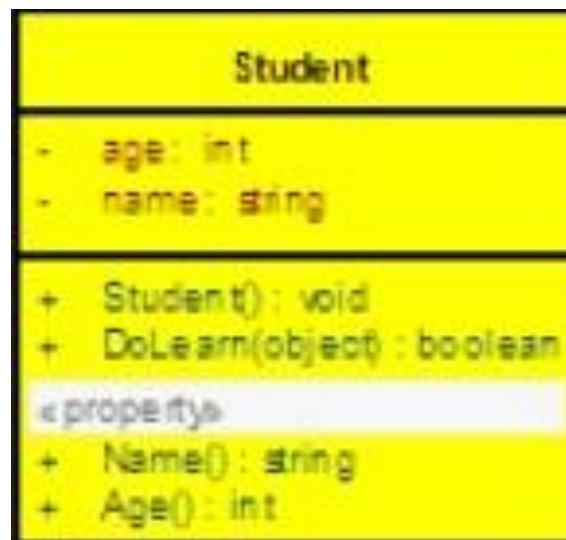


```
<Class Type> <Identifier> = new <Class Type>(<Parameters>)  
Person him = new Person("John Doe", 25, 'M');
```

Fields

- Fields are data members of a class
 - Can be variables and constants (read-only)
- Accessing a field doesn't invoke any actions of the object
- Usually properties are used instead of directly accessing variable fields.

Fields



Constant Fields

- Constant fields are defined like fields, but:
 - Defined with `const` keyword
 - Must be initialized at their definition
 - Their value cannot be changed at runtime



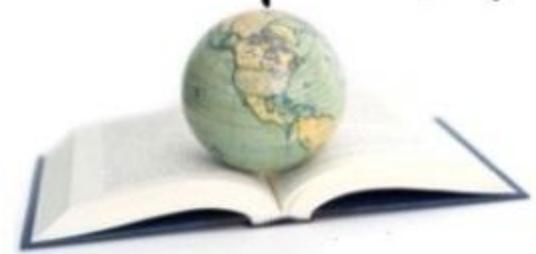
```
public class MathConstants
{
    public const string PI_SYMBOL = "π";
    public const double PI = 3.1415926535897932385;
    public const double E = 2.7182818284590452354;
    public const double LN10 = 2.30258509299405;
    public const double LN2 = 0.693147180559945;
}
```

Read-Only Fields

- Initialized at the definition or in the constructor
 - Cannot be modified further
- Defined with the keyword `read-only`
- Represent runtime constants

```
public class ReadOnlyDemo
{
    private readonly int size;
    public ReadOnlyDemo(int Size)
    {
        size = Size; // can not be
                     // further
                     // modified!
    }
}
```

the world is a book
and those who
do not travel
read only one page



Properties (1 of 2)

- Properties look like fields
 - Have name and type
 - Can contain code, executed when accessed
- Used as wrappers
 - To control the access to the data fields
 - Can contain validation against the data or complex logic
- Can contain two components
 - get accessor – for reading property value
 - set accessor – for changing property value



Properties (2 of 2)

- Implemented Property can be:
 - Read-only (getter only)
 - Read and Write (getter and setter)
 - Write-only (setter only)
- Example of read-only property
 - String.Length
- Example of read-write property
 - Console.BackgroundColor



Defining a Property

Specify the access modifier

Optionally,
specify an
access modifier
for the get or set

Specify the type

Specify the
property name

Add get and/or
set accessors

```
private string myString;  
  
public string MyString  
{  
    get  
    {  
        return myString;  
    }  
    private set  
    {  
        myString =  
        value;  
    }  
}
```

Use the value keyword in the
set accessor to access the
data passed to the property

Automatic Properties

- Useful when there is no need to add custom logic to the property accessors
- Must specify both get and set accessors
- No difference between automatic and normal properties to consuming applications

```
public string Name { get; set; }
```



The automatic property shown above is converted by the compiler to code

```
private string _name;  
  
public string Name  
{  
    get  
    {  
        return _name;  
    }  
    set  
    {  
        this._name = value;  
    }  
}
```

Instance and Static Members

- Fields, Properties and Methods can be:
 - Instance or Object Members
 - Static or Class Members
- Instance members are specific for each object
 - Example: Different employees have different names
- Static members are common for all instances of a class
 - Example: DateTime.MinValue is shared between all instances of DateTime



Accessing Members

- Accessing instance members
 - The name of the instance, followed by the name of the member (field or property), separated by dot (“ . ”)

```
<instance_name>.<member_name>
```

- Accessing static members
 - The name of the class, followed by the name of the member

```
<class_name>.<member_name>
```



Accessing Instance Members – Example

```
using System;

public class Person
{
    private string firstName;
    private string lastName;

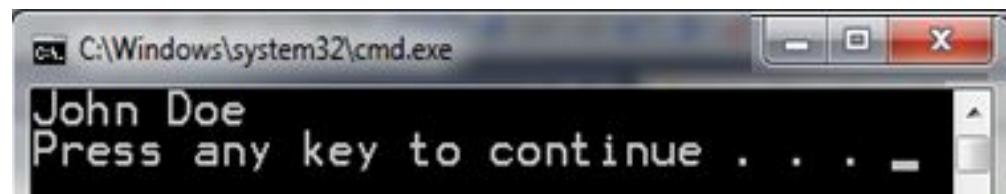
    public string FirstName
    {
        get
        {
            return firstName;
        }
        set
        {
            firstName = value;
        }
    }

    public string LastName
    {
        get
        {
            return lastName;
        }
        set
        {
            lastName = value;
        }
    }
}
```

```
public static int Main(string[] args)
{
    Person person = new Person();
    person.FirstName = "John";
    person.LastName = "Doe";

    System.Console.WriteLine("{0} {1}",
    person.FirstName, person.LastName);

    return 0;
}
```



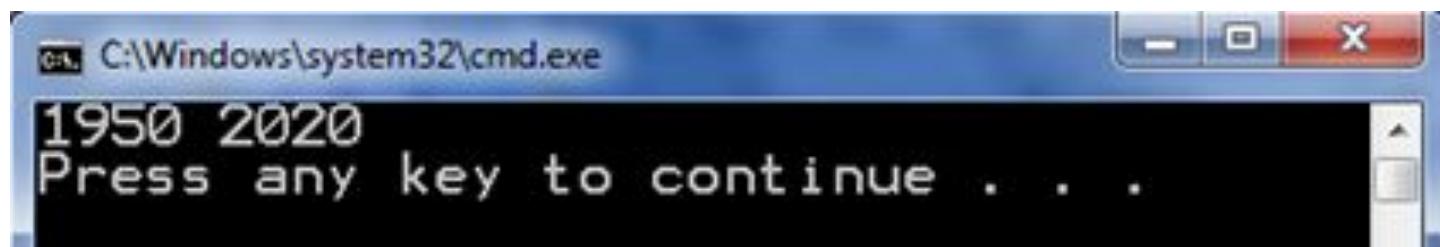
Accessing Properties and Fields – Example

```
using System;

public class DateValidator
{
    public static int MinYear = 1950;
    public static int MaxYear = 2020;
}

public static int Main(string[] args)
{
    System.Console.WriteLine("{0} {1}",
        DateValidator.MinYear,
        DateValidator.MaxYear);

    return 0;
}
```



Methods



What is a Method?

- A method is a kind of building block that solves a small problem
 - A piece of code that has a name and can be called from the other code
 - Can take parameters and return a value
- Method allows programmers to construct large programs from simple pieces
- Method is also known as function, procedure, and subroutine



Why use Methods?

- More manageable code
 - Splitting large problems into small pieces
 - Better organization of the program
 - Improve code readability
 - Improve understanding of the code



Methods - Example

- Examples:
 - `Console.WriteLine(...)`
 - `Console.ReadLine()`
 - `String.Substring(index, length)`
 - `Array.GetLength(index)`
 - `List<T>.Add(item)`
 - `DateTime.AddDays(count)`



Declaring and Creating Methods

- Each method has a name
 - It is used to call the method
 - Describe its purpose
- Each method has a body
 - It contains the programming code
 - Surrounded by curly braces { }
- Methods declared can be called by any other method
- Methods are always declared inside a class



Method - Example

```
public void PrintLogo()  
{  
    Console.WriteLine(".NET C#");  
    Console.WriteLine("www.microsoft.com");  
}
```

Method name

Method body

Calling Methods

- To call a method, simple use:
 - The method's name
 - Parentheses
 - A semicolon



```
PrintLogo();
```

Instance Methods

- Instance methods manipulate the data of a specified object
- Syntax:
 - The name of the instance, followed by the name of the method, separated by dot

```
<object_name>.<method_name>(<parameters>)
```



Calling Instance Methods - Example

- Calling instance methods of the String:

```
String sampleLower = new String('a', 5);
String sampleUpper = sampleLower.ToUpper();

Console.WriteLine(sampleLower); // aaaaa
Console.WriteLine(sampleUpper); // AAAAA
```

- Calling instance methods of DateTime:

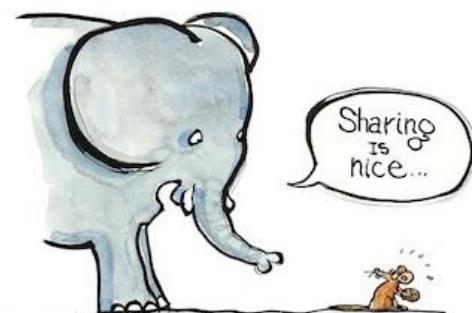
```
DateTime now = DateTime.Now;
DateTime later = now.AddHours(8);

Console.WriteLine("Now: {0}", now);
Console.WriteLine("8 hours later: {0}", later);
```

Static Methods

- Static methods are common for all instance of a class (shared between all instances)
 - No need to create an instance of class
- Syntax:
 - The name of the class, followed by the name of the method, separated by dot

```
<class_name>.<method_name>(<parameters>)
```



Static Methods - Example

```
using System;
```

Constant field

Static
method

```
double radius = 2.9;  
double area = Math.PI * Math.Pow(radius, 2);  
Console.WriteLine("Area: {0}"  
// Area: 26,4207942166902
```

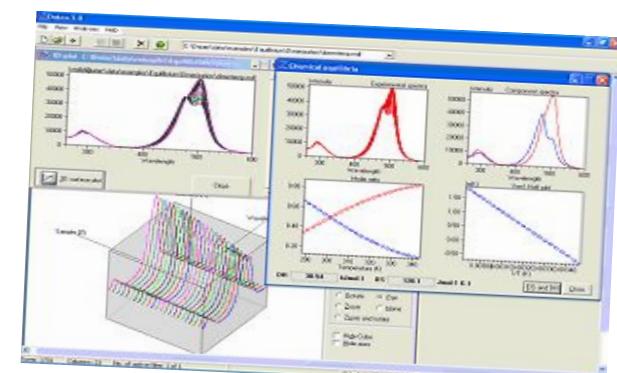
Static
method

```
double precise = 8.7654321;  
double round3 = Math.Round(precise, 3);  
double round1 = Math.Round(precise, 1);  
Console.WriteLine(  
    "{0}; {1}; {2}", precise, round3, round1  
// 8,7654321; 8,765; 8,8
```

Static method

Method Parameters

- Parameters can be used to pass information to a method, also known as arguments.
 - Can pass zero or several input values
 - Can pass values of different types
 - Each parameter has name and type
 - Parameters are assigned to a particular values when the method is called
- Parameters can change the method behavior depending on the passed values



Defining and using Method parameters (1 of 2)

- Method's behavior depends on its parameters
- Parameters can be of any type
 - int, double, string, etc.
 - arrays (int[], double[], etc.)

```
static void PrintSign(int number)
{
    if (number > 0)
        Console.WriteLine("Positive");
    else if (number < 0)
        Console.WriteLine("Negative");
    else
        Console.WriteLine("Zero");
}
```



Defining and using Method parameters (2 of 2)

- Methods can have as many parameters as needed

```
static void PrintMax(float number1, float  
number2)
```

```
{  
    float max = number1;  
    if (number2 > number1)  
        max = number2;
```

- The following syntax is not valid

```
max);  
}
```

```
static void PrintMax(float number1, number2)
```



Calling Methods with Parameters (1 of 2)

- To call a method and pass values to its parameters
 - Use the method's name, followed by a list of expressions for each parameter
- Examples

```
PrintSign (-5) ;
```

```
PrintSign (balance) ;
```

```
PrintSign (2+3) ;
```

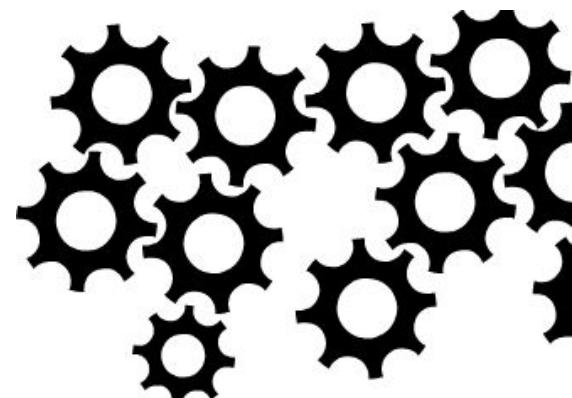
```
PrintMax (100, 200) ;
```

```
PrintMax (oldQuantity * 1.5, quantity * 2) ;
```



Calling Methods with Parameters (2 of 2)

- Expressions must be of the same type as method's parameters or compatible
 - If the method requires a float expression, you can pass int instead
- Use the same order like in method declaration
- For methods with no parameters do not forget the parentheses



Optional Parameters

- C# 4.0 supports optional parameters with default values assigned at their declaration

```
static void PrintNumbers(int start = 0; int end = 100)
{
    for (int i = start; i <= end; i++)
    {
        Console.Write("{0} ", i);
    }
}
```

- The method can be called in several ways

```
PrintNumbers(5, 10);
PrintNumbers(15);
PrintNumbers();
PrintNumbers(end: 40, start: 35);
```

Returning Values from Methods

- A method can return a value to its caller
- Returned value:
 - Can be assigned to a variable

```
string message = Console.ReadLine();  
// Console.ReadLine() returns a string
```

- Can be used in expressions

```
float price = GetPrice() * quantity * 1.20;
```

- Can be passed to another method

```
int age = int.Parse(Console.ReadLine());
```

Defining Methods that return a value

- Instead of void, specify the type of the data to return

```
static int Multiply(int firstNum, int secondNum)
{
    return firstNum * secondNum;
}
```

- Methods can return any type of data (int, string, array, etc.)
- void methods do not return anything
- The combination of method's name, parameters and return value is called method signature
- Use return keyword to return a result

The return statement

- The return statement
 - Immediately terminates method's execution
 - Returns specified expression to the caller
 - Example

```
return -1;
```

- To terminate void method

```
return;
```

- Return can be used several times in a method body

Method Overloading

- Overloading enables you to create multiple methods within a class that have the same name but different signatures

```
class Zoo {  
    public void AddLion(Lion newLion) { ...  
    }  
    public void AddLion(Lion newLion,  
                        int exhibitNumber) {  
        ...  
    }  
}
```

Methods – Best Practices

- Each method should perform a single, well-defined task
- Method's name should describe the task in a clear and non-ambiguous way
 - Good Example: CalculatePrice(), ReadFile()
 - Bad Example: f(), g1(), Process()

Using Partial Classes and Partial Methods

- Split class definition across multiple source files
- Prefix the class and method declarations with partial keyword
- All parts of the partial class and method must be available during compilation
- Compiled into a single entity

```
public partial class Residence
{
    partial void SaleResidence(string name);
}
```

Definition of SalesResidence
method

```
public partial class Residence
{
    partial void SaleResidence(string name)
    {
        //Logic goes here.
    }
}
```

Implementation of
SalesResidence method

Constructors



Constructors

- Constructors are special methods used to assign initial values of the fields in an object
 - Executed when an object of a given type is being created
 - Have the same name as the class that holds them
 - Do not return a value
- A class may have several constructors with different set of parameters
- Constructor is invoked by the new operator

```
<instance_name> = new <class_name>(<parameters>)
```

Constructors - Example

- Examples:

```
String s = new String(new char[] {'a', 'b', 'c'});
```

```
String s = new String('*', 5); // s = "*****"
```

```
DateTime dt = new DateTime(2009, 12, 30);
```

```
DateTime dt = new DateTime(2009, 12, 30, 12, 33, 59);
```

```
Int32 value = new Int32();
```



Static Constructor

- Used to initialize any non-changing data, or to perform a particular action done once.
- It is called automatically before the first instance is created or any static members are referenced.

```
class SimpleClass {
    // Static variable is initialized at run time.
    static readonly long baseline;

    // Static constructor is first before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass() {
        baseline = DateTime.Now.Ticks;
    }
}
```

Default Constructor

- The constructor without parameters is called default constructor
- Example:
 - Creating an object for generating random numbers with a default seed

```
using System;  
...  
Random randomGenerator = new Random();
```

Parameterless constructor

call

The class `System.Random` provides a source of pseudo-random
numbers

Constructor with Parameters

- Example
 - Creating objects for generating random values with specified initial seeds

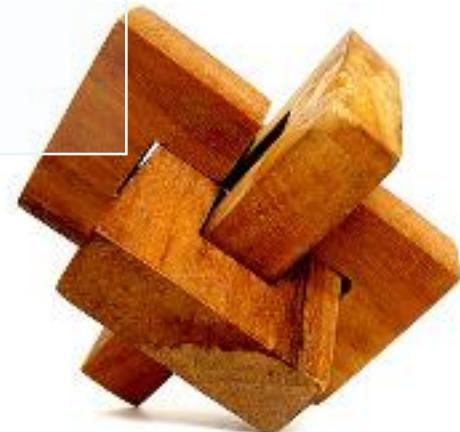
```
using System;  
...  
Random randomGenerator1 = new Random(123);  
Console.WriteLine(randomGenerator1.Next());  
// 2114319875  
  
Random randomGenerator2 = new Random(456);  
Console.WriteLine(randomGenerator2.Next(50));  
// 47
```



Constructor Overloading

- Create multiple constructors that have the same name but different signatures

```
public class Lion {  
    private string      name;  
    private int       age;  
  
    public Lion() : this( "unknown", 0 ) {  
        Console.WriteLine("Default: {0}", name);  
    }  
    public Lion( string theName, int theAge ) {  
        name = theName;  
        age = theAge;  
        Console.WriteLine("Specified: {0}", name);  
    }  
}
```



Constructor Chaining

- Constructor Chaining is an approach where a constructor calls another constructor in the same or base class.

```
public class Point
{
    private int xCoord;
    private int yCoord;

    public Point() : this(0,0) // Reuse constructor
    {
    }

    public Point(int xCoord, int yCoord)
    {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }

    // More code ...
}
```

Namespace



What is a Namespace?

- Namespaces are used to organize the source code into more logical and manageable way
- Namespaces can contain
 - Definition of classes, structures, interfaces and other types and other namespaces
- Namespaces can contain other namespaces
- Example:
 - System namespace contains Data namespace
 - The name of the nested namespace is System.Data



Full Class Names

- A full name of a class is the name of the class preceded by the name of its namespaces

```
<namespace_name>.<class_name>
```

- Example:
 - Array class, defined in the System namespace
 - The full name of the class is System.Array

How to use Namespaces in C#

- The using directive in C#

```
using <namespace_name>
```

- Allows using types in namespace, without specifying the full name of the class

Example:

```
using System;  
DateTime date;
```

Instead of

```
System.DateTime date;
```

Namespace Alias

- Creating Namespace alias

```
using Co = CompanyName.Project.Nested;
```

- Global Namespace alias

```
using colAlias = System.Collections;
namespace System {
    class TestClass {
        static void Main() {
            // Searching the alias:
            colAlias::Hashtable test = new colAlias::Hashtable();

            // Add items to the table.
            test.Add("A", "1");
            test.Add("B", "2");

            foreach (string name in test.Keys) {
                // Searching the global namespace:
                global::System.Console.WriteLine(name + " " + test[name]);
            }
        }
    }
}
```

Namespace – Example (1 of 2)

```
using System;
using A.B.C;

namespace E {
    using D;

    class Program {
        static void Main() {
            // Can access CClass type directly from A.B.C.
            CClass var1 = new CClass();

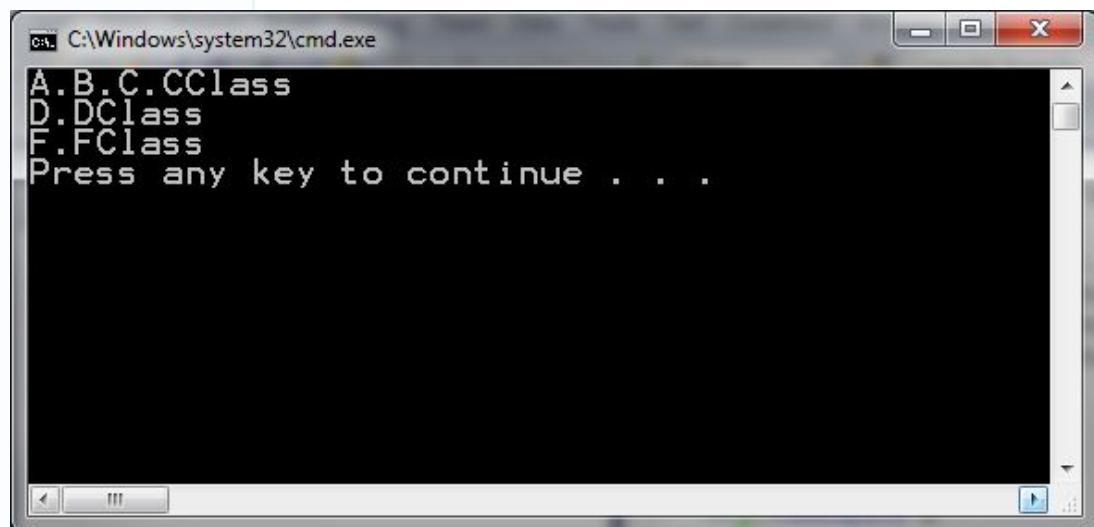
            // Can access DClass type from D.
            DClass var2 = new DClass();

            // Must explicitly specify F namespace.
            F.FClass var3 = new F.FClass();

            // Output.
            Console.WriteLine(var1);
            Console.WriteLine(var2);
            Console.WriteLine(var3);
        }
    }
}
```

Namespace – Example (2 of 2)

```
namespace A {  
    namespace B {  
        namespace C {  
            {  
                public class CClass  
                {  
                }  
            }  
        }  
    }  
}  
  
namespace D {  
    public class DClass  
    {  
    }  
}  
  
namespace F {  
    public class FClass  
    {  
    }  
}
```



Checkpoint

- Which of the following statements is correct about the C#.NET code snippet given below?

```
class Student s1, s2;  
s1 = new Student();  
s2 = new Student();
```

- a. Contents of *s1* and *s2* will be exactly same.
- b. The two objects will get created on the stack.
- c. Contents of the two objects created will be exactly same.
- d. The two objects will always be created in adjacent memory locations.



Checkpoint

Checkpoint

- Which of the following statements is correct about properties used in C#.NET?
 - a. A property can simultaneously be read only or write only.
 - b. A property can be either read only or write only.
 - c. A write only property will have only get accessor.
 - d. A write only property will always return a value.



Checkpoint

Checkpoint

- A method defined with keyword void?
 - a. must specify a return type
 - b. does not accept arguments
 - c. return a value
 - d. does not return a value



Checkpoint

Checkpoint

- A static field or method must be accessed using the class name.
 - a. True
 - b. False
 - c. Information is not enough



Checkpoint

Checkpoint

- An important difference between constructors and other methods is that:
 - a. constructors cannot specify a return type
 - b. constructors cannot specify any parameters
 - c. other methods are implemented as void methods
 - d. constructors can assign values to instance variables



Checkpoint

Checkpoint

- Which of the following does a namespace NOT provide?
 - a. Logical grouping of types
 - b. Help in avoiding name collisions
 - c. Access control



Checkpoint

Knowledge Check



Knowledge Check

1. Instance variables declared private are not accessible.
 - a. outside the class
 - b. by other methods of the same class
 - c. by other members of the same class
 - d. inside the same class
2. A class can yield many , just as a built-in type can yield many variables.
 - a. Names
 - b. Objects
 - c. Values
 - d. types

Knowledge Check

3. The operator is used to create an object.
 - a. createobject
 - b. instantiate
 - c. create
 - d. new

4. The keyword introduces a class declaration.
 - a. newclass
 - b. classdef
 - c. csclass
 - d. class

Knowledge Check

5. Which of the following statements are correct?

1. Instance members of a *class* can be accessed only through an object of that *class*.
 2. A *class* can contain only instance data and instance member *function*.
 3. All objects created from a *class* will occupy equal number of bytes in memory.
 4. A *class* can contain Friend functions.
 5. A *class* is a blueprint or a template according to which objects are created.
- b. 1, 3, 5
- c. 2, 4
- d. 3, 5
- e. 2, 4, 5

Knowledge Check

6. Which of the following is the correct way to create an object of the *class Sample*?
1. Sample s = new Sample();
 2. Sample s;
 3. Sample s; s = new Sample();
 4. s = new Sample();
- b. 1, 3
- c. 2, 4
- d. 1, 2, 3
- e. 4, 5

Knowledge Check

7. Which of the following statements are correct about the C#.NET code snippet given below?

sample s;

c = new Sample();

1. It will create an object called sample.
 2. It will create a nameless object of the type sample.
 3. It will create an object of the type sample on the stack.
 4. It will create a reference c on the stack and an object of the type sample on the heap.
 5. It will create an object of the type sample either on the heap or on the stack depending on the size of the object.
- b. 1, 3
- c. 2, 4
- d. 3, 5
- e. None of the above

Knowledge Check

8. What is the difference between void and non-void methods?
 - a. void methods return values, non-void methods do not.
 - b. non-void methods return values, void methods do not.
 - c. void methods accept parameters, non-void methods do not.
 - d. non-void methods accept parameters, void methods do not.
9. What occurs after a method call is made?
 - a. Control is given to the called method. After the method is run, the application continues execution at the point where the method call was made.
 - b. Control is given to the called method. After the method is run, the application continues execution with the statement after the called method's declaration.
 - c. The statement before the method call is executed.
 - d. The application terminates.

Knowledge Check

10. Methods can return value(s).
 - a. zero or one
 - b. exactly one
 - c. one or more
 - d. any number of
11. Which of the following must be true when making a method call?
 - a. The number of arguments in the method call must match the number of parameters in the method header.
 - b. The argument types must be compatible with their corresponding parameter types.
 - c. Both a and b
 - d. Neither a or b

Knowledge Check

12. Which of the following statements correctly returns the variable intValue from a method?
 - a. return int intValue;
 - b. return void intValue;
 - c. intValue return;
 - d. return intValue;
13. Which of the following statements is correct about constructors?
 - a. If we provide a one-argument constructor then the compiler still provides a zero-argument constructor.
 - b. Overloaded constructors cannot use optional arguments.
 - c. If we do not provide a constructor, then the compiler provides a zero-argument constructor.

Knowledge Check

14. Which of the following statements are correct about static methods?
1. Static methods can access only static data.
 2. Static methods cannot call instance methods.
 3. It is necessary to initialize static data.
 4. Instance methods can call static methods and access static data.
 5. *this* reference is passed to static methods.
- b. 1, 2, 4
- c. 2, 3, 5
- d. 3, 4
- e. 4, 5
- f. None of the above

Knowledge Check

15. In which of the following should the methods of a class differ if they are to be treated as overloaded methods?
- a. Type of arguments
 - b. Return type of methods
 - c. Number of arguments
 - d. Names of methods
 - e. Order of arguments
- b. 2, 4
 - c. 3, 5
 - d. 1, 3, 5
 - e. 3, 4, 5

Module 8: OO Programming with C#



Module Objectives

After completing this module, you will be able to:

- Describe how to control the visibility of type members.
- Describe how to share methods and data.
- Use inheritance to define new reference types.
- Define and implement interfaces
- Define abstract and sealed classes



Module objectives

Module Agenda

- Inheritance
- Abstraction
- Encapsulation
- Polymorphism



OOP Fundamental Principles

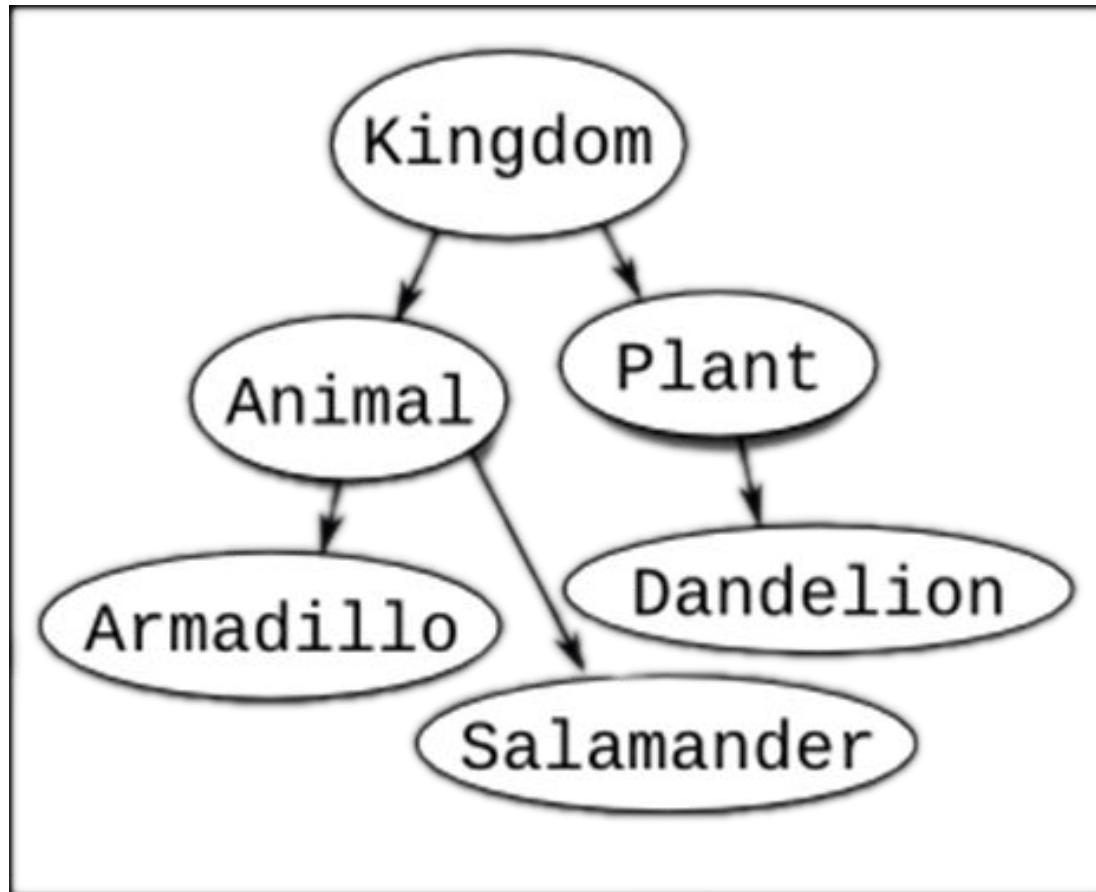
- Inheritance
- Abstraction
- Encapsulation
- Polymorphism



Inheritance



Inheritance



Classes and Interfaces

- Classes define attributes and behavior
 - Fields, properties, methods, etc.
 - Methods contain code for execution

```
public class Employee { ... }
```

- Interfaces define a set of operations
 - Empty methods and properties, left to be implemented later

```
public interface IService { ... }
```

Inheritance

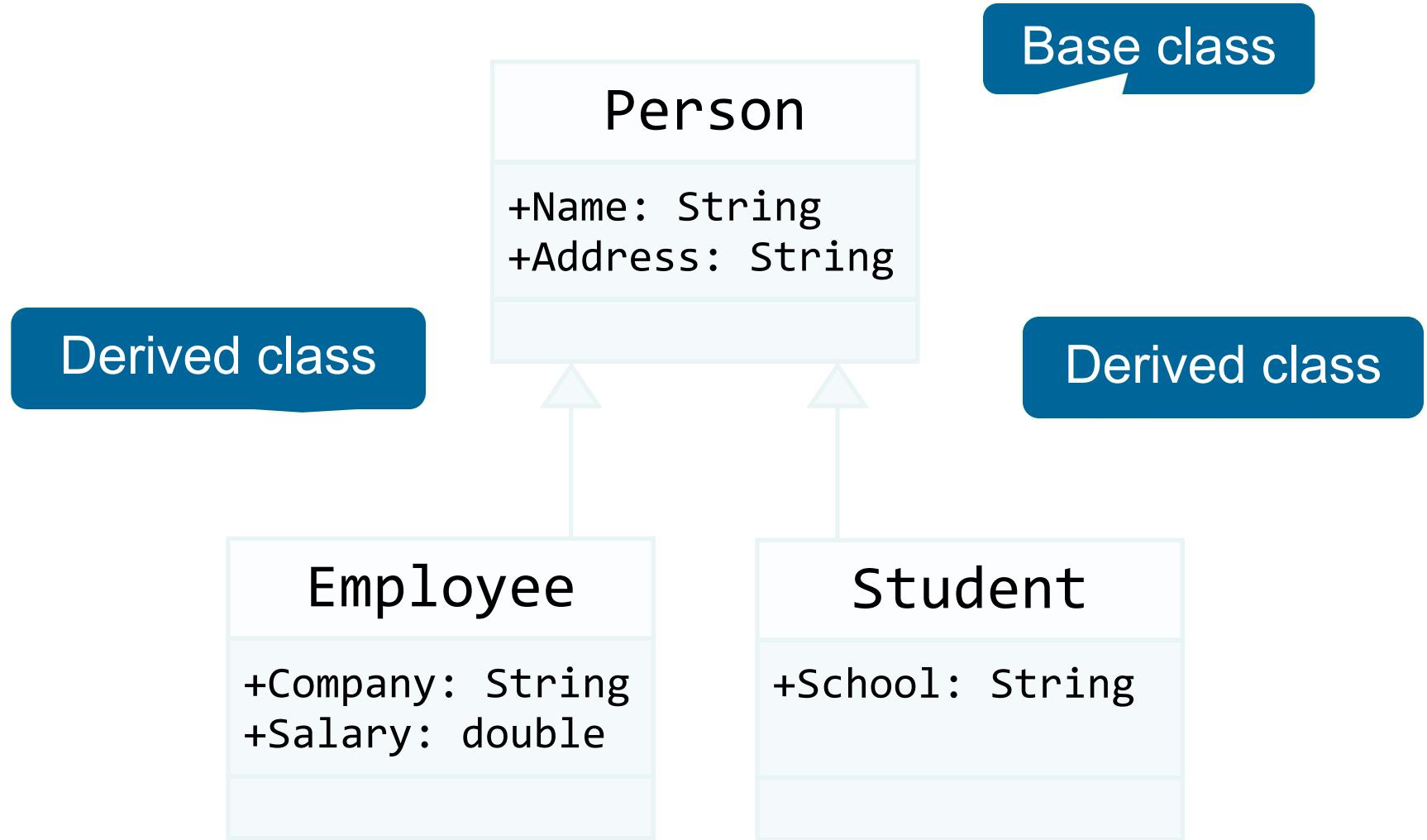
- Inheritance allows child classes inherits the characteristics of existing parent class
 - Attributes
 - Operations
- Inheritance is use for building Is-A relationship
- Child class can extend the parent class
 - Add new fields and methods
 - Redefine methods (modify existing behavior)
- A class can implement an interface by providing implementation for all its methods

Inheritance Benefits

- Extensibility
- Reusability
- Provides abstraction
- Eliminates redundant code



Inheritance – Class Diagram



Types of Inheritance

derived class

inherits

base class /
parent class

class

implements

interface

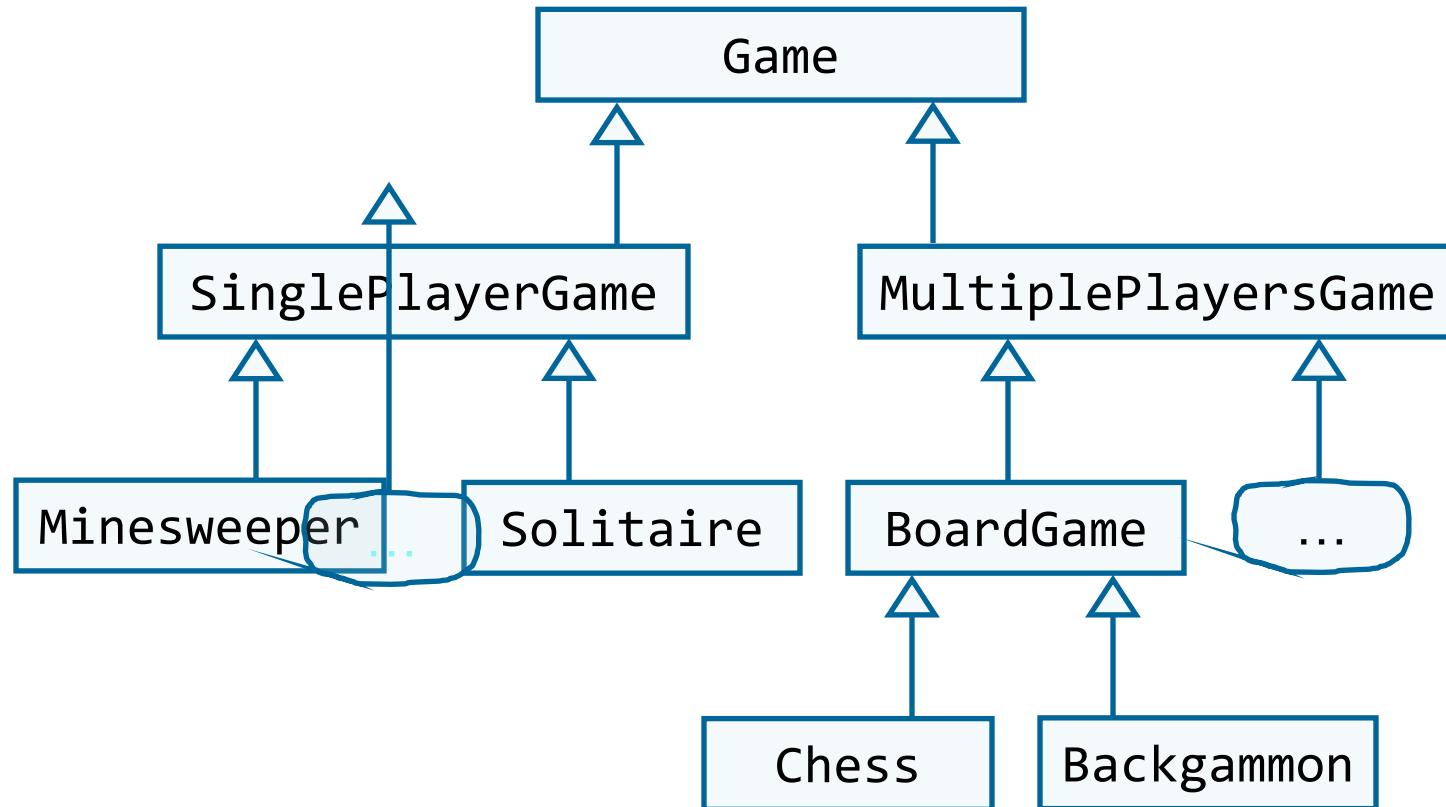
derived interface

implements

base interface

Class Hierarchies

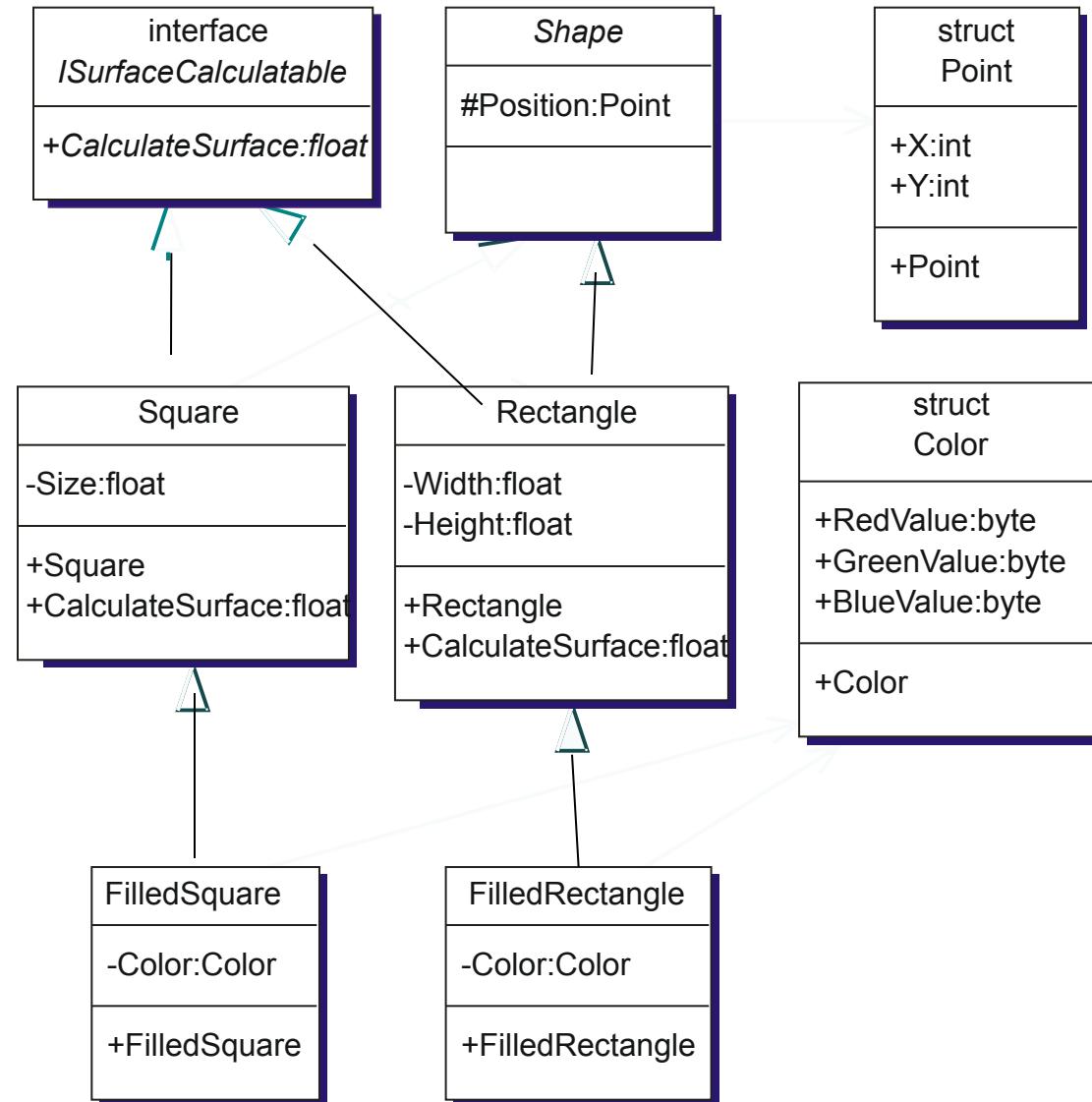
- Inheritance leads to a hierarchy of classes and/or interfaces in an application



Inheritance Hierarchies

- Using inheritance we can create inheritance hierarchies
 - Easily represented by UML class diagrams
- UML class diagrams
 - Classes are represented by rectangles containing their methods and data
 - Relations between classes are shown as arrows
 - Closed triangle arrow means inheritance
 - Other arrows mean some kind of associations

UML Class Diagram – Example



Inheritance in .NET

- A class can inherit only one base class
 - E.g. IOException derives from SystemException and it derives from Exception
- A class can implement several interfaces
 - This is .NET's form of multiple inheritance
 - E.g. List<T> implements IList<T>, ICollection<T>, IEnumerable<T>
- An interface can implement several interfaces
 - E.g. IList<T> implements ICollection<T> and IEnumerable<T>

How to define Inheritance?

- Specify the name of the base class after the name of the derived

```
public class Shape  
{...}  
public class Circle : Shape  
{...}  
  
public Circle (int x, int y) : base(x)  
{...}
```

- In the constructor of the derived class use the keyword base to invoke the constructor of the base class

Inheritance – Example (1 of 2)

```
public class Mammal
{
    public int Age { get; set; }

    public Mammal(int age)
    {
        this.Age = age;
    }

    public void Sleep()
    {
        Console.WriteLine("Shhh! I'm sleeping!");
    }
}
```

Inheritance – Example (2 of 2)

```
public class Dog : Mammal
{
    public string Breed { get; set; }

    public Dog(int age, string breed)
        : base(age)
    {
        this.Breed = breed;
    }

    public void WagTail()
    {
        Console.WriteLine("Tail wagging...");
    }
}
```

Access Modifiers

- Member modifiers change the way class members can be used
- Access modifiers describe how a member can be accessed

- Access

Modifier	Description
public	Access is not restricted
private	Access is restricted to the containing type
protected	Access is limited to the containing type and types derived from it
internal	Access is limited to the current assembly
protected internal	Access is limited to the current assembly or types derived from the containing class

Inheritance and Access Modifiers – Example (1 of 2)

```
class Creature
{
    protected string Name { get; private set; }

    private void Talk()
    {
        Console.WriteLine("I am creature ...");
    }

    protected void Walk()
    {
        Console.WriteLine("Walking ...");
    }
}

class Mammal : Creature
{
    // base.Talk() can be invoked here
    // this.Name can be read but cannot be modified here
}
```

Inheritance and Access Modifiers – Example (2 of 2)

```
class Dog : Mammal
{
    public string Breed { get; private set; }
    // base.Talk() cannot be invoked here (it is private)
}

class InheritanceAndAccessibility
{
    static void Main()
    {
        Dog joe = new Dog(6, "Labrador");
        Console.WriteLine(joe.Breed);
        // joe.Walk() is protected and can not be invoked
        // joe.Talk() is private and can not be invoked
        // joe.Name = "Rex"; // Name cannot be accessed here
        // joe.Breed = "Shih Tzu"; // Can't modify Breed
    }
}
```

Inheritance: Important Pointers

- Structures cannot be inherited
- Multiple inheritance in C# is not supported
 - Only multiple interfaces can be implemented
- Instance and static constructors are not inherited
- Inheritance is transitive relation
 - If C is derived from B, and B is derived from A, the C inherits A as well



Inheritance: Important Features

- A derived class extends its base class
 - It can add new members but cannot remove derived ones
- Declaring new members with the same name or signature hides the inherited ones
- A class can declare virtual methods and properties
 - Derived classes can override the implementation of these members
 - E.g. Object.Equals() is a virtual method

Abstraction



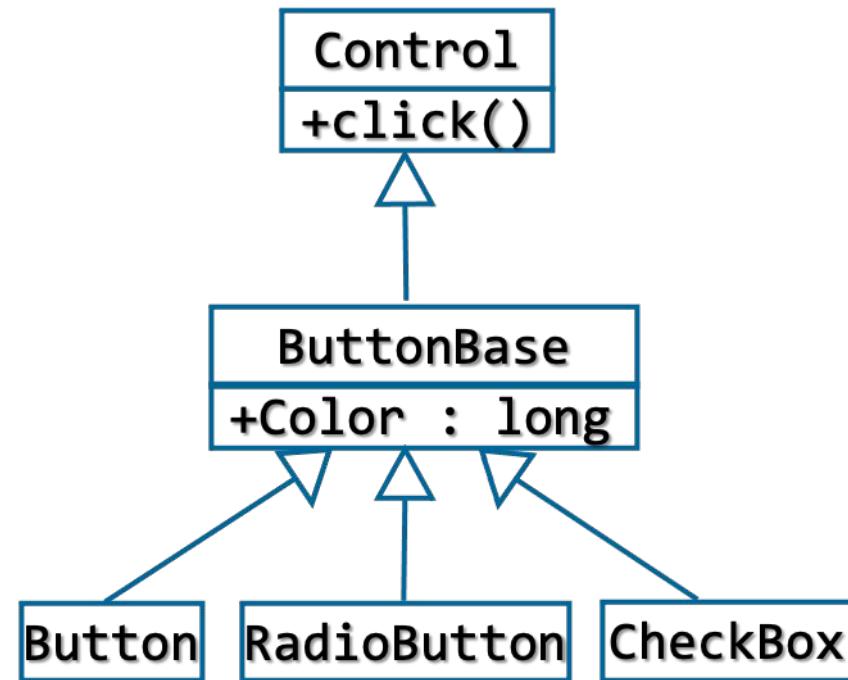
Abstraction

- Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones.
- Allows us to represent a complex reality in terms of a simplified model
- Abstraction highlights the properties of an entity that we need and hides the other
- Abstraction = Managing Complexity

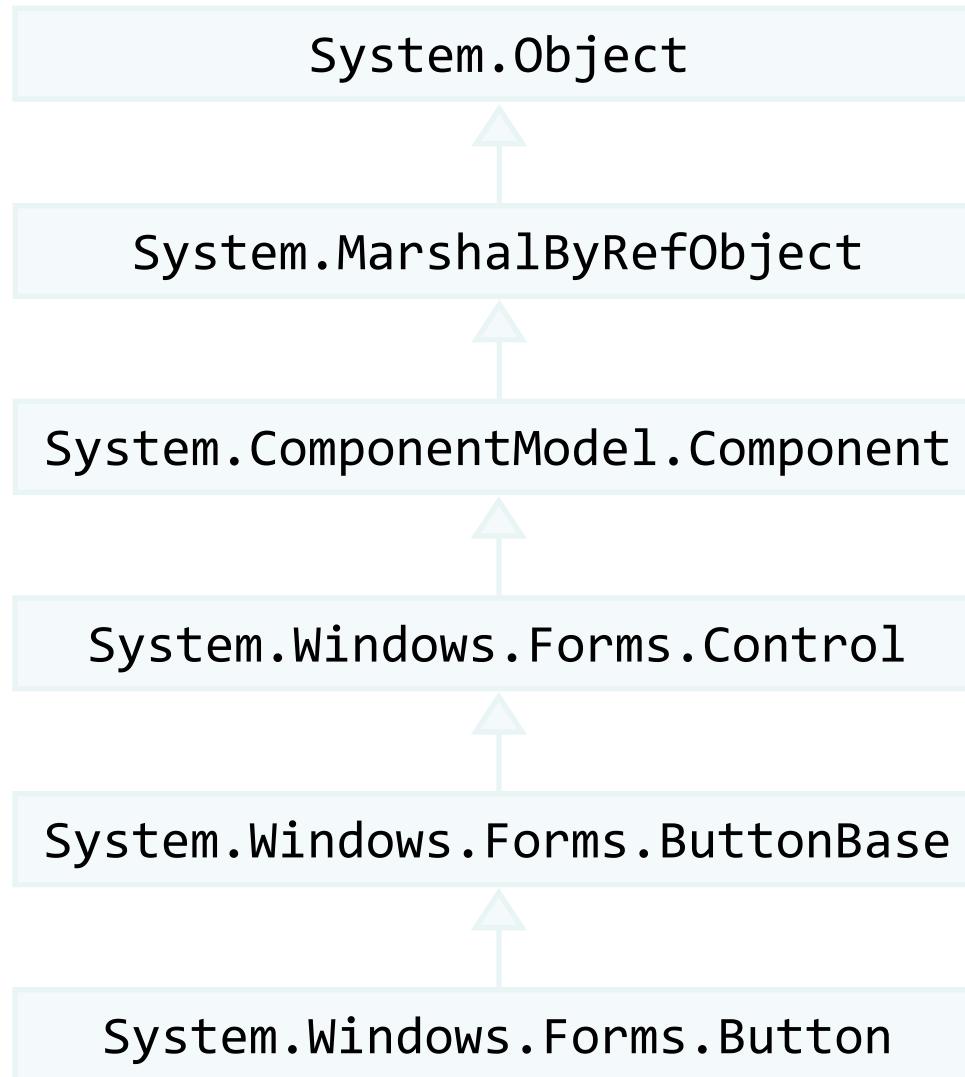


Abstraction in .NET

- In .NET abstraction is achieved in several ways
 - Abstract Classes
 - Interfaces
 - Inheritance



Abstraction in .NET - Example



Interfaces in C#

- An interface is a set of operations (methods) that a given object can perform
 - Also called “contract” for supplying set of operations
 - Defines abstract behavior

- Interfaces provide abstractions
 - You shouldn’t have to know anything about that is in the implementation in order to use it

Abstract Class in C#

- Abstract classes are special classes defined with the keyword `abstract`
 - Mix between class and interface
 - Partially implemented or fully unimplemented
 - Not implemented methods are declared `abstract` and are left empty
 - Cannot be instantiated
- Child classes should implement abstract methods or declare them as `abstract`

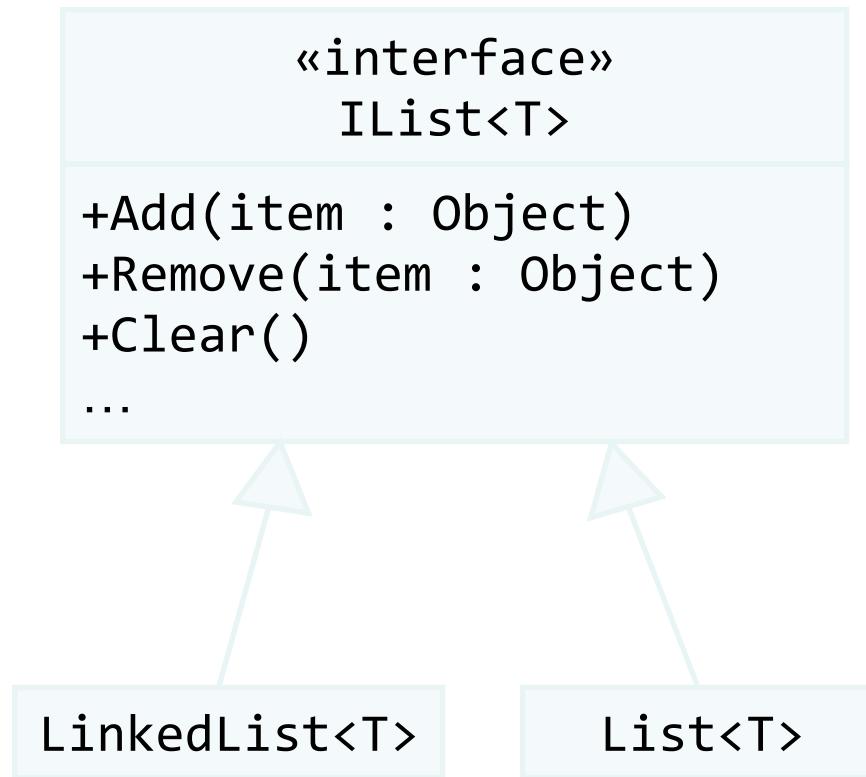
Sealed Classes

- The sealed modifier is applied to an overridden method and prevents further overriding in derived classes

```
class Calculator {  
    public virtual int CalcSum(int x, int y){  
        return x + y;  
    }  
    public virtual int CalcProduct(int x, int y) {  
        return x * y;  
    }  
}  
  
class DerivedCalc : Calculator {  
    public sealed override int CalculateSum(int x, int y){  
        // more specialized method action/calculation  
    }  
    public override int CalculateProduct(int x, int y) {  
        // more specialized implementation of method  
        //action/calculation  
    }  
}
```

Abstract Data Types

- Abstract Data Types (ADT) are data types defined by a set of operations (interface)



Encapsulation

- Encapsulation hides the implementation details
- Class announces some operations (methods) available for its clients – its public interface
- All data members (fields) of class should be hidden
 - Accessed via properties (read-only and read-write)
- No interface members should be hidden

Encapsulation



Encapsulation - Example

- Data fields are private
- Constructors and accessors are define (getters and setters)

Person

-name : string
-age : TimeSpan

+Person(string name, int age)
+Name : string { get; set; }
+Age : TimeSpan { get; set; }

Encapsulation in .NET

- Fields are always declared private
 - Accessed through properties in read-only or read-write mode
- Constructors are almost always declared public
- Interface methods are always public
 - Not explicitly declared with public
- Non-interface methods are declared private / protected

Benefits of Encapsulation

- Ensures that the structural changes remain local:
 - Changing the class internal does not affect any code outside of the class
 - Changing method's implementation does not reflect the clients using them
- Encapsulation allow adding some logic when accessing client's data
 - E.g. Validation on modifying a property value
- Hiding implementation details reduces complexity

Polymorphism



Polymorphism (1 of 2)

- Polymorphism is the ability to take more than one form (objects have more than one type)
 - A class can be used through its parent interface
 - A child class may override some of the behaviors of the parent class
- Allows abstract operations to be defined and used
 - Abstract operations are defined in the base class' interface and implemented in the child class
 - Declared using abstract or virtual keywords

Polymorphism (2 of 2)

- Why handle an object of a given type as object of its base type?
 - To invoke abstract operations
 - To mix different related types in the same collection
 - To pass more specific object to a method that expects a parameter of a more generic type
 - To declare a more generic field which will be initialized and “specialized” later

Virtual Methods

- Virtual method is a method that can be used in the same way on instances of base and derived classes but implementation is different
- A method is said to be virtual when it is declared using the virtual keyword

```
public virtual void CalculateSurface()
```

- Methods that are declared as virtual in a base class can be overridden using the keyword override in the derived class

The override Modifier

- Using override we can modify a method or property
- An override method provides a new implementation of a member inherited from a base class
- You cannot override a non-virtual or static method
- The overridden base method must be virtual, abstract, or override

Polymorphism – How it works?

- Polymorphism ensures that the appropriate method of the subclass is called through its base class interface
- Polymorphism is implemented using a technique called the late method binding
 - Exact method to be called is determined at runtime, just before performing the call
 - Applied for all abstract/virtual methods

Polymorphism – Example 1

Abstract
class

Figure

Abstract
action

+CalcSurface() : double

Concrete
class

Square

Circle

Overridden
action

-x : int
-y : int
-size : int

-x : int
-y : int
-radius: int

Overridden
action

```
override CalcSurface()
{
    return size * size;
}
```

```
override CalcSurface()
{
    return PI * radius * raduis;
}
```

Polymorphism – Example 2

```
abstract class Figure
{
    public abstract double CalcSurface();
}

abstract class Square
{
    public override double CalcSurface() { return ... }
}

Figure f1 = new Square(...);
Figure f2 = new Circle(...);

// This will call Square.CalcSurface()
int surface = f1.CalcSurface();

// This will call Square.CalcSurface()
int surface = f2.CalcSurface();
```

Checkpoint

- Which of the following statements are correct about an interface in C#.NET?
 1. A class can implement multiple interfaces.
 2. Structures cannot inherit a class but can implement an interface.
 3. In C#.NET, it is used to signify that a class member implements a specific interface.
 4. An interface can implement multiple classes.
 5. The static attribute can be used with a method that implements an interface declaration.

b. 1, 2, 3

c. 2, 4

d. 3, 5



Checkpoint

Checkpoint

- In an inheritance chain which of the following members of base class are accessible to the derived class members?
 1. static
 2. protected
 3. private
 4. shared
 5. public
 - b. 1, 3
 - c. 2, 5
 - d. 3, 4
 - e. 4, 5



Checkpoint

Checkpoint

- A derived class can stop virtual inheritance by declaring an override as:
 - a. Inherits
 - b. Extends
 - c. Inheritable
 - d. Sealed



Checkpoint

Checkpoint

- What is the default access modifier for a member of a class?
 - a. Public
 - b. Private
 - c. Protected
 - d. None



Checkpoint

Checkpoint

- For an instance of a derived class to completely take over a class member from a base class, the base class has to declare that member as
 - a. New
 - b. Base
 - c. Virtual
 - d. Overrides
 - e. Overloads



Checkpoint

Knowledge Check



Knowledge Check

1. Which statement is incorrect?
 - a. Inheritance enables you to create powerful computer programs more easily.
 - b. Without inheritance, you *could* create every part of a program from scratch, but reusing existing classes and interfaces makes your job easier.
 - c. Inheritance is frequently inefficient because base class code is seldom reliable when extended to a derived class.

Knowledge Check

2. A derived class:
 - a. Inherits fields and methods defined in its base class
 - b. Can add new fields and methods
 - c. Can hide methods defined in the base class
 - d. All of the above
3. True or false: the CLR supports multiple inheritance for languages that want to support multiple inheritance.
 - a. True
 - b. False
 - c. Depends on the language
 - d. Information is not enough

Knowledge Check

4. True or false: C# supports multiple inheritance
 - a. True
 - b. False
 - c. Depends on the operating system
 - d. Information is not enough
5. Assume *class B* is inherited from *class A*. Which of the following statements is correct about construction of an object of *class B*?
 - a. While creating the object, the constructor of *class B* will be called followed by constructor of *class A*.
 - b. While creating the object, the constructor of *class A* will be called followed by constructor of *class B*.
 - c. The constructor of only *class B* will be called.
 - d. The constructor of only *class A* will be called.

Knowledge Check

6. Which of the following are reuse mechanisms available in C#.NET?

- 1. *Inheritance*
 - 2. *Encapsulation*
 - 3. Abstraction
 - 4. Polymorphism
-
- b. 1, 4
 - c. 1, 3
 - d. 2, 4
 - e. 3, 4

Knowledge Check

7. Which of the following statements is correct about the C#.NET code snippet given below?

```
interface IMyInterface {  
    void fun1();  
    int fun2();}  
class MyClass: IMyInterface {  
    void fun1()  
    { }  
    int IMyInterface.fun2()  
    { }  
}
```

- a. A function cannot be declared inside an interface.

Knowledge Check

8. Which of the following can be declared in an interface?

a. Properties

b. Methods

c. Enumerations

d. Events

e. Structures

b. 1, 3

c. 1, 2, 4

d. 3, 5

e. 4, 5

Knowledge Check

9. Which of the following statements is correct about Interfaces used in C#.NET?
 - a. All interfaces are derived from an *Object* class
 - b. Interfaces can be inherited
 - c. All interfaces are derived from an *Object* interface
 - d. Interfaces can contain method implementation

10. Which of the following modifier is used when a virtual method is redefined by a derived class ?
 - a. Overloads
 - b. Override
 - c. Virtual
 - d. Base

Knowledge Check

11. Which of the following are necessary for Run-time Polymorphism?
1. The overridden base method must be virtual, abstract or override.
 2. Both the override method and the virtual method must have the same access level modifier.
 3. An override declaration can change the accessibility of the virtual method.
 4. An abstract inherited property cannot be overridden in a derived class.
 5. An abstract method is implicitly a virtual method.
- b. 1, 3
- c. 1, 2, 5
- d. 2, 3, 4
- e. 4 only

Knowledge Check

12. In order for an instance of a derived class to completely take over a class member from a base class, the base class has to declare that member as?
 - a. New
 - b. Base
 - c. Override
 - d. Virtual

13. Information hiding is possible in object-oriented programming because of:
 - a. inheritance
 - b. Instantiation
 - c. objects
 - d. Classes
 - e. encapsulation

Knowledge Check

14. Which of the following statements is correct about the C#.NET code snippet given below?

```
interface IMyInterface {  
    void fun1();  
    void fun2();  
}  
class MyClass: IMyInterface {  
    private int i;  
    void IMyInterface.fun1() {  
        // Some code  
    }  
}
```

- a. Class *MyClass* is an abstract class.
- b. Class *MyClass* cannot contain instance data.
- c. Class *MyClass* fully implements the interface *IMyInterface*.
- d. The compiler will report an error since the interface *IMyInterface* does not contain the method *fun1()*.

Knowledge Check

15. Which of the followings is the correct way to overload + operator?
- a. public sample operator + (sample a, sample b)
 - b. public abstract operator + (sample a, sample b)
 - c. public abstract sample operator + (sample a, sample b)
 - d. public static sample operator + (sample a, sample b)

Module 9: Events, Delegates and Lambda



Module Overview

At the end of this module, you will be able to:

- Describe the purpose of delegates and how to use a delegate.
- Describe how to use a lambda expression to define an anonymous method.
- Explain the purpose of events and describe how to use events.



Module objectives

Module Agenda

- Delegates
- Events
- Lambda



Delegates



What are Delegates?

- At the lowest level of the CLR, delegates are objects that contain a reference to some method on some object.
- It is also referred to as a type safe function pointer.
- Advantage:
 - It can be used to call different methods during program runtime by simply changing the method to which the delegate refers.
- They can point to both static or instance methods
- Their “values” are methods
- Used to perform callbacks



Delegate

- Example

```
delegate int BinOp(int a, int b);
static class Program
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static void Main()
    {
        BinOp add = new BinOp(Add);
        int three = add(1, 2); // calls the Add method
                               // through the delegate
    }
}
```

- This code can be further improved using Method Group Conversion

```
static void Main()
{
    BinOp add = Add;
    int three = add(1, 2); // calls the Add method
                          // through the delegate
}
```

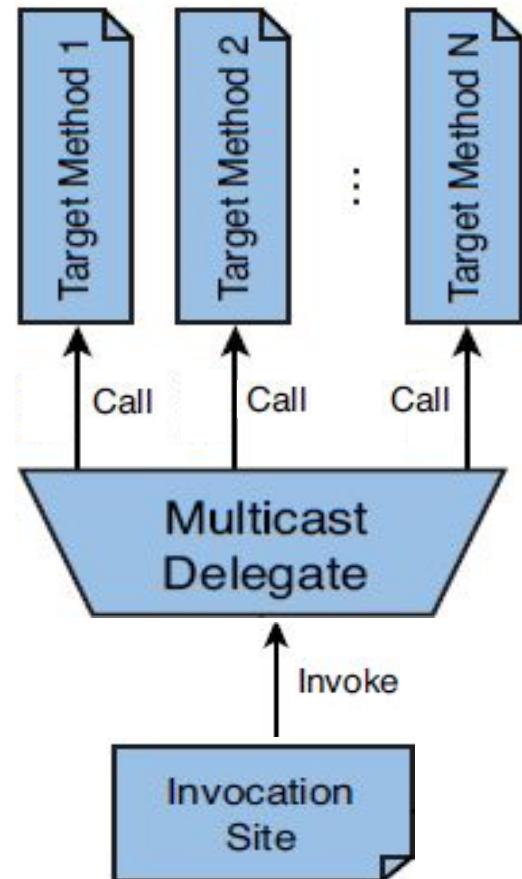


Delegate - Multicast

- Multicasting is the ability to create an invocation list, or chain, of methods that will be automatically called when a delegate is invoked.
- Steps to multicast:
 - instantiate a delegate
 - Use the **+** or **+ =** operator to add methods to the chain
 - Use the **-** or **- =** operator to remove methods to the chain
- A delegate that makes use of multicasting will often have a **void** return type.
- Use a **ref** parameter to return the altered string to the caller.

Multicast

```
// Declare a delegate type.  
delegate void StrMod(ref string str);  
  
static void Main() {  
    // Construct delegates.  
    StrMod strOp;  
    StrMod replaceSp = ReplaceSpaces;  
    StrMod removeSp = RemoveSpaces;  
    StrMod reverseStr = Reverse;  
    string str = "This is a test";  
    // Set up multicast.  
    strOp = replaceSp;  
    strOp += reverseStr;  
    // Call multicast.  
    strOp(ref str);  
    Console.WriteLine("Resulting string: " + str);  
    Console.WriteLine();  
    // Remove replace and add remove.  
    strOp -= replaceSp;  
    strOp += removeSp;  
    str = "This is a test."; // reset string  
    // Call multicast.  
    strOp(ref str);  
    Console.WriteLine("Resulting string: " + str);  
    Console.WriteLine();  
}
```



Anonymous Function

- An anonymous function is an unnamed block of code that is passed to a delegate constructor.
- Advantages:
 - Less coding
 - Improve code readability
- C# defines two types of anonymous function:
 - Anonymous Methods - is one way to create an unnamed block of code that is associated with a specific delegate instance.
 - Lambda Expressions



Anonymous Methods

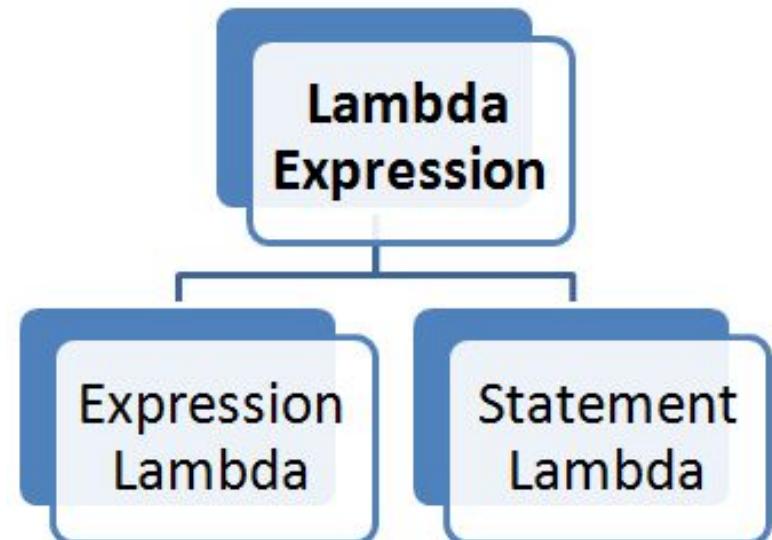
```
// This delegate returns a value.
delegate int CountIt(int end);
class AnonMethod
{
    static void Main()
    {
        int result;
        // Ending value for count is passed to the anonymous method.
        // A summation of the count is returned.
        CountIt count = delegate(int end)
        {
            int sum = 0;
            for (int i = 0; i <= end; i++)
            {
                Console.WriteLine(i);
                sum += i;
            }
            return sum; // return a value from an anonymous method
        };
        result = count(3);
        Console.WriteLine("Summation of 3 is " + result);
        Console.WriteLine();
        result = count(5);
        Console.WriteLine("Summation of 5 is " + result);
    }
}
```

Lambda



What is a Lambda Expression?

- A lambda expression is another way an anonymous function can be created but more streamlined, a recommended approach.
- Defined by the Lambda operator: =>
- Lambda operator is read as: “goes to” or “becomes.”
- Two types:
 - Expression Lambda
 - Statement Lambda



Expression Lambda

- General form of an expression lambda.

param => expr

- Parenthesis is required if there are more than one parameter:

(param-list) => expr

- Examples:

`count => count + 2`

`n => n % 2 == 0`



Expression Lambda

```
// Declare a delegate that returns an int result.  
delegate int Incr(int v);  
// Declare a delegate that returns a bool result.  
delegate bool IsEven(int v);  
class SimpleLambda  
{  
    static void Main()  
    {  
        Incr incr = count => count + 2;  
        Console.WriteLine("Use incr lambda expression: ");  
        int x = -10;  
        while (x <= 0)  
        {  
            Console.Write(x + " ");  
            x = incr(x); // increase x by 2  
        }  
        Console.WriteLine("\n");  
        IsEven isEven = n => n % 2 == 0;  
        Console.WriteLine("Use isEven lambda expression: ");  
        for (int i = 1; i <= 10; i++)  
            if (isEven(i)) Console.WriteLine(i + " is even.");  
    }  
}
```

Statement Lambda

- A statement lambda expands the types of operations that can be handled within a lambda expression because it allows the body of lambda to contain multiple statements including loops, method calls and **if** statements.
- In Statement Lambda, the block of statements are enclosed by braces.
- Example:

```
// Reverse a string.  
StrMod Reverse = s => {  
    string temp = "";  
    int i, j;  
    Console.WriteLine("Reversing string.");  
    for(j=0, i=s.Length-1; i >= 0; i--, j++)  
        temp += s[i];  
    return temp;  
};
```

Statement Lambda

```
//statement lambda
// IntOp takes one int argument and returns an int result.
delegate int IntOp(int end);
class StatementLambda
{
    static void Main()
    {
        // A statement lambda that returns the factorial
        // of the value it is passed.
        IntOp fact = n => {
            int r = 1;
            for (int i = 1; i <= n; i++)
                r = i * r;
            return r; }

        Console.WriteLine("The factorial of 3 is " + fact(3));
        Console.WriteLine("The factorial of 5 is " + fact(5));
    }
}
```

Events



Event

- Event is an automatic notification that some action has occurred.
- General Form:

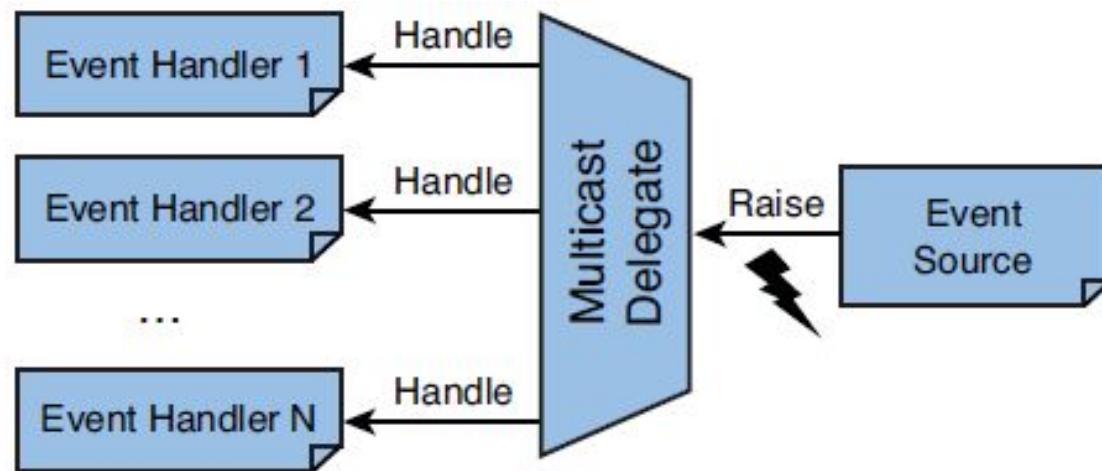
event event-delegate event-name;

- *event-delegate* is the name of the delegate used to support the event
- *event-name* is the name of the specific event object being declared.



How Event works?

- Event Steps:
 - Define the Event
 - Register an “event handler” for that event
 - When the event occurs, all registered handlers are called.
- Event handlers are represented by delegates.



Events vs. Delegates

- Events are not the same as member fields of type Delegate

```
public MyDelegate m;
```

#

```
public event MyDelegate m;
```

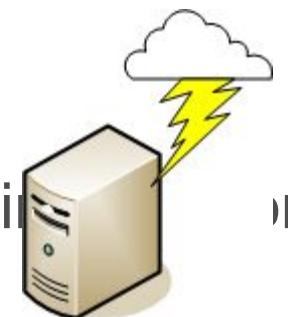
- The event is process by a delegate
- Events can be members of an interface unlike delegates
- Calling an event can only be done in the class it is defined in
- By default the access to the events is synchronized (thread-safe)

System.EventHandler Delegate

- Defines a reference to a callback method, which handles events
 - No additional information is sent

```
public delegate void EventHandler(Object sender,  
EventArgs e);
```

- Used in many occasions internally in .NET
 - E.g. in ASP.NET and Windows Forms
- The EventArgs class is a base class with no information about the event



Event Example

```
// Declare a delegate type for an event.  
delegate void MyEventHandler();  
// Declare a class that contains an event.  
class MyEvent {  
    public event MyEventHandler SomeEvent;  
    // This is called to raise the event.  
    public void OnSomeEvent() {  
        if(SomeEvent != null)  
            SomeEvent();  
    }  
}  
class EventExample {  
    // An event handler.  
    static void Handler() {  
        Console.WriteLine("Event occurred");  
    }  
    static void Main() {  
        MyEvent evt = new MyEvent();  
        // Add Handler() to the event list.  
        evt.SomeEvent += Handler;  
        // Raise the event.  
        evt.OnSomeEvent();  
    }  
}
```

Multicast Event 1/2

```
// Declare a delegate type for an event.  
delegate void MyEventHandler();  
// Declare a class that contains an event.  
class MyEvent {  
    public event MyEventHandler SomeEvent;  
    // This is called to raise the event.  
    public void OnSomeEvent() {  
        if (SomeEvent != null)  
            SomeEvent();  
    }  
}  
class X {  
    public void Xhandler() {  
        Console.WriteLine("Event received by X object");  
    }  
}  
class Y {  
    public void Yhandler() {  
        Console.WriteLine("Event received by Y object");  
    }  
}
```

Multicast Event 2/2

```
class EventExample {
    static void Handler() {
        Console.WriteLine("Event received by EventExample");
    }
    static void Main() {
        MyEvent evt = new MyEvent();
        X xOb = new X();
        Y yOb = new Y();
        // Add handlers to the event list.
        evt.SomeEvent += Handler;
        evt.SomeEvent += xOb.Xhandler;
        evt.SomeEvent += yOb.Yhandler;

        evt.OnSomeEvent(); // Raise the event.
        Console.WriteLine();
        evt.SomeEvent -= xOb.Xhandler; // Remove a handler.
        evt.OnSomeEvent();
    }
}
```

Lambda Expressions with Events

```
// Declare a delegate type for an event.  
delegate void MyEventHandler(int n);  
// Declare a class that contains an event.  
class MyEvent  
{  
    public event MyEventHandler SomeEvent;  
    // This is called to raise the event.  
    public void OnSomeEvent(int n)  
    {  
        if (SomeEvent != null)  
            SomeEvent(n);  
    }  
}  
class LambdaEventExample  
{  
    static void Main()  
    {  
        MyEvent evt = new MyEvent();  
        // Use a lambda expression as an event handler.  
        evt.SomeEvent += (n) =>  
        Console.WriteLine("Event received. Value is " + n);  
        // Raise the event twice.  
        evt.OnSomeEvent(1);  
        evt.OnSomeEvent(2);  
    }  
}
```

.NET Event Guidelines 1/2

- Event Handlers have two parameters.
 - The first is a reference to the object that generated the event.
 - The second is a parameter of type EventArgs that contains any other information required by the handler.

```
// Derive a class from EventArgs.
class MyEventArgs : EventArgs {
public int EventNum;
}

// Declare a delegate type for an event.
delegate void MyEventHandler(object sender, MyEventArgs e);
// Declare a class that contains an event.
class MyEvent
{
    static int count = 0;
    public event MyEventHandler SomeEvent;
    // This raises SomeEvent.
    public void OnSomeEvent()
```



.NET Event Guidelines 2/2

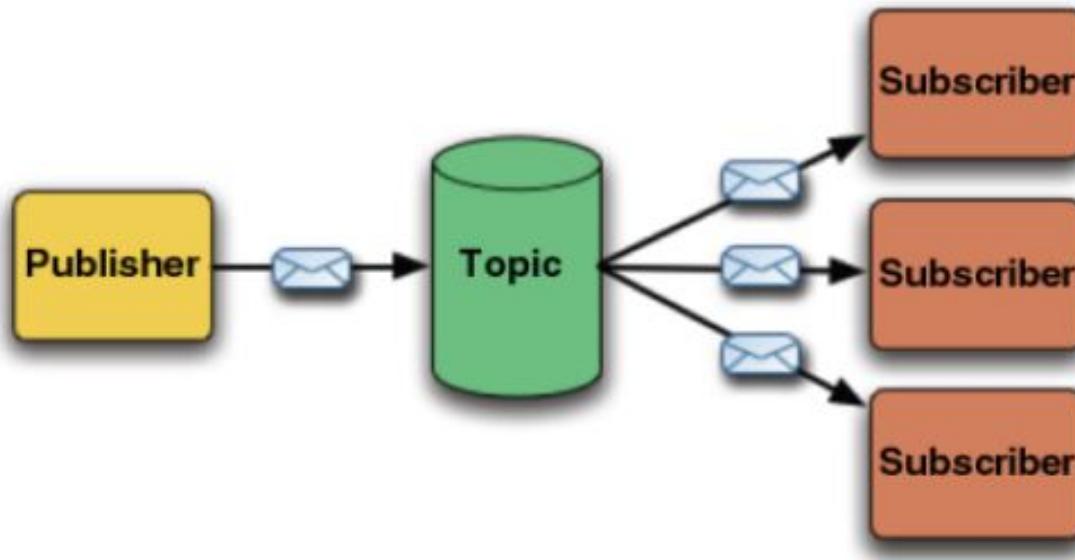
- Use EventHandler<TEventArgs> and EventHandler
 - .NET Framework provides a built-in delegate called EventHandler<TEventArgs>

```
public event EventHandler<MyEventArgs> SomeEvent;
```
 - .NET Framework includes a non-generic delegate called **EventHandler**, which can be used to declare event handlers in which no extra information is needed.

```
public event EventHandler SomeEvent;
```



Publisher-Subscriber Concept



Publish Events 1/4

```
namespace DotNetEvents
{
    using System;
    using System.Collections.Generic;

    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string s)
        {
            message = s;
        }
        private string message;

        public string Message
        {
            get { return message; }
            set { message = value; }
        }
    }
}
```

Publish Events 2/4

```
// Class that publishes an event
class Publisher
{
    // Declare the event using EventHandler<T>
    public event EventHandler<CustomEventArgs> RaiseCustomEvent;

    public void DoSomething()
    {
        // Write some code that does something useful here
        // then raise the event. You can also raise an event
        // before you execute a block of code.
        OnRaiseCustomEvent(new CustomEventArgs("Did something"));

    }

    // Wrap event invocations inside a protected virtual method
    // to allow derived classes to override the event invocation behavior
    protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
    {
        // Make a temporary copy of the event to avoid possibility of
        // a race condition if the last subscriber unsubscribes
        // immediately after the null check and before the event is raised.
        EventHandler<CustomEventArgs> handler = RaiseCustomEvent;

        // Event will be null if there are no subscribers
        if (handler != null)
        {
            // Format the string to send inside the CustomEventArgs parameter
            e.Message += String.Format(" at {0}", DateTime.Now.ToString());

            // Use the () operator to raise the event.
            handler(this, e);
        }
    }
}
```

Publish Events 3/4

```
//Class that subscribes to an event
class Subscriber
{
    private string id;
    public Subscriber(string ID, Publisher pub)
    {
        id = ID;
        // Subscribe to the event using C# 2.0 syntax
        pub.RaiseCustomEvent += HandleCustomEvent;
    }

    // Define what actions to take when the event is raised.
    void HandleCustomEvent(object sender, CustomEventArgs e)
    {
        Console.WriteLine(id + " received this message: {0}", e.Message);
    }
}
```

Publish Events 4/4

```
class Program
{
    static void Main(string[] args)
    {
        Publisher pub = new Publisher();
        Subscriber sub1 = new Subscriber("sub1", pub);
        Subscriber sub2 = new Subscriber("sub2", pub);

        // Call the method that raises the event.
        pub.DoSomething();

        // Keep the console window open
        Console.WriteLine("Press Enter to close this window.");
        Console.ReadLine();

    }
}
```

Best Practices for using Events

- Use the standard event signature
- Use a protected virtual method to raise an event
- Do not pass null to an event



Checkpoint

- You need to write a multicast delegate that accepts a DateTime argument. Which code segment should you use?
 - a. public delegate int PowerDeviceOn(bool, DateTime)
 - b. public delegate bool PowerDeviceOn(Object, EventArgs)
 - c. public delegate void PowerDeviceOn(DateTime)
 - d. public delegate bool PowerDeviceOn(DateTime)



Checkpoint

Checkpoint

- What statement is incorrect about Lambda expression?
 - a. A lambda expression is an anonymous function that can contain expressions and statements
 - b. Lambda expression can be used to create delegates
 - c. All lambda expressions use the symbol:
`<=`



Checkpoint

Checkpoint

- What is the effect of adding the event keyword to a delegate field?
 - a. The delegate is automatically made public
 - b. Client code is limited to accessing the delegate using only the += and -= operators
 - c. The delegate is limited to handling only one subscribe



Checkpoint

Knowledge Check



Knowledge Check

1. Which statement is incorrect in handling events?
 - a. An action such as a key press or button click raises an event.
 - b. A method that performs a task in response to an event is an event handler.
 - c. The control that generates an event is an event receiver.
 - d. None of above

2. Which statement is incorrect about delegates?
 - a. A delegate is an object that contains a reference to a method.
 - b. Once you have created a delegate, it can encapsulate any method with the same identifier as the delegate.
 - c. A composed delegate can be created using the `+=` operator; it calls the delegates from which it is built.
 - d. None of the above

Knowledge Check

3. Which of the following statements are correct about a delegate?
1. Inheritance is a prerequisite for using delegates.
 2. Delegates are type-safe.
 3. Delegates provide wrappers for function pointers.
 4. The declaration of a delegate must match the signature of the method that we intend to call using it.
 5. Functions called using delegates are always late-bound.
- b. 1 and 2 only
- c. 1, 2 and 3 only
- d. 2, 3 and 4 only
- e. All of above

Knowledge Check

4. Which of the following are the correct ways to declare a delegate for calling the function *func()* defined in the sample class given below?

```
class Sample {  
    public int func(int i, Single j) {  
        /* Add code here. */  
    }  
}
```

- a. delegate d(int i, Single j);
- b. delegate void d(int, Single);
- c. delegate int d(int i, Single j);
- d. delegate int sample.func(int i, Single j);

Knowledge Check

5. Which statement is incorrect?

- a. When an event occurs, any delegate that a client has given or passed to the event is invoked
- b. Built-in event handler delegates have two arguments, but those you create yourself have only one
- c. You can take only two actions on an event field: composing a new delegate onto the field using the `+=` operator, and removing a delegate from the field using the `-=` operator.
- d. All statements are correct

6. How is the lambda operator (`=>`) read?

- a. goes to
- b. will get
- c. will have
- d. go there

Knowledge Check

7. Which statement is incorrect?
- a. A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces.
 - b. Statement lambdas, like anonymous methods, cannot be used to create expression trees.
 - c. Both a and b are correct
 - d. Both a and b are incorrect

Knowledge Check

8. Which of the following is the correct way to call subroutine *MyFun()* of the *Sample* class given below?

```
class Sample {  
    public void MyFun(int i, Single j) {  
        Console.WriteLine("Welcome to HelloWorld!");  
    }  
}
```

- a. delegate void del(int i);
Sample s = new Sample();
del d = new del(ref s.MyFun);
d(10, 1.1f);
- b. delegate void del(int i, Single j);
del d;
Sample s = new Sample();
d = new del(ref s.MyFun);
d(10, 1.1f);
- c. Sample s = new Sample();
delegate void d = new del(ref MyFun);
d(10, 1.1f);
- d. delegate void del(int i, Single];)
Sample s = new Sample();
del = new delegate(ref MyFun);
del(10, 1.1f);

Module 10:Exception Handling



Module Objectives

At the end of this module, you will be able to:

- Describe how to catch and handle exceptions.
- Describe how to create and raise exceptions.



Module objectives

Module Agenda

Exceptions



What are Exceptions?

- The exceptions in .NET Framework are classic implementation of the OOP exception model
- Deliver powerful mechanism for centralized handling of errors and unusual events
- Substitute procedure-oriented approach, in which each function returns error code
- Simplify code construction and maintenance
- Allow the problematic situations to be processed at multiple levels



Handling Exceptions

- In C# the exceptions can be handled by the try-catch-finally constructions

```
try
{
    // Do some work that can raise an exception
}
catch (SomeException)
{
    // Handle the caught exception
}
finally
{
    // Executed with/without exception
}
```

- Catch blocks can be used multiple times to process different exception types

Handling Exceptions - Example

```
static void Main()
{
    string s = Console.ReadLine();
    try
    {
        Int32.Parse(s);
        Console.WriteLine(
            "You entered valid Int32 number {0}.", s);
    }
    catch (FormatException)
    {
        Console.WriteLine("Invalid integer number!");
    }
    catch (OverflowException)
    {
        Console.WriteLine(
            "The number is too big to fit in Int32!");
    }
}
```

The System.Exception Class

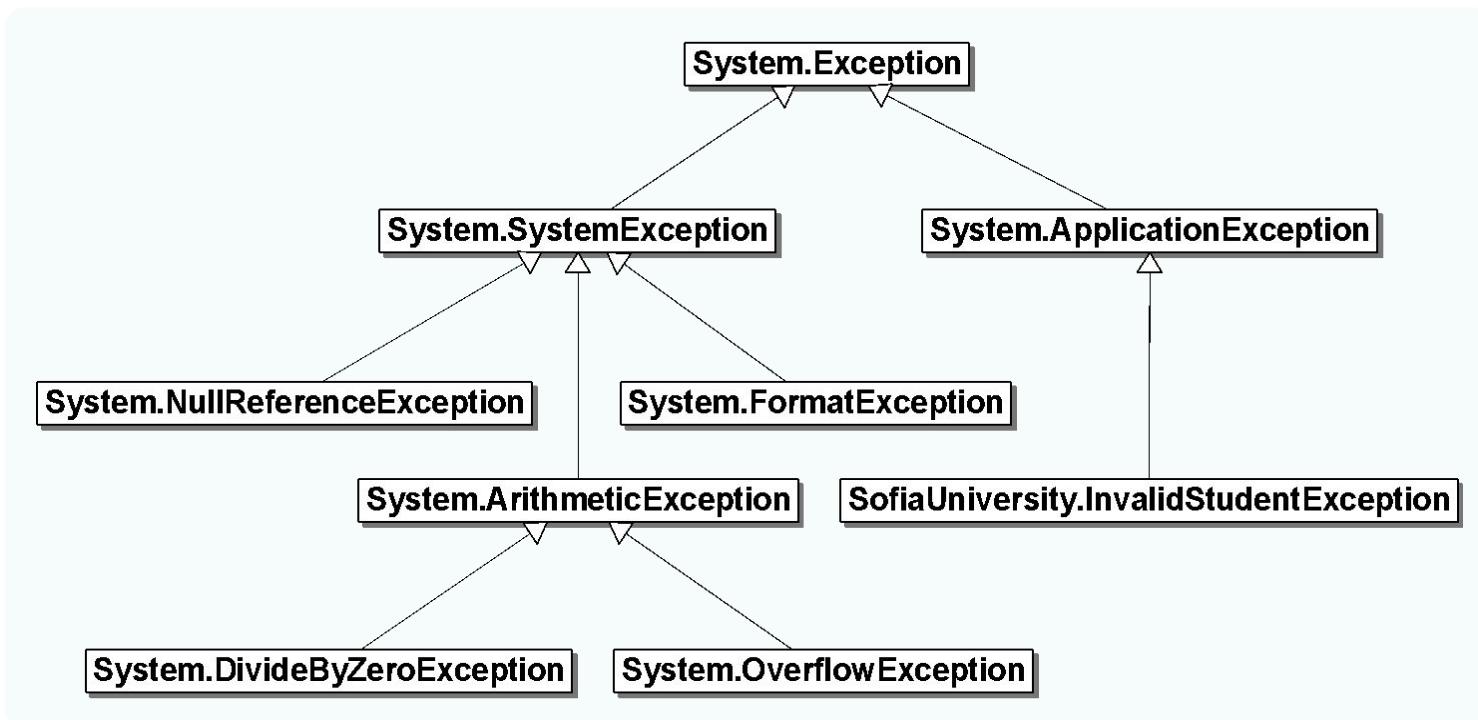
- Exceptions in .NET are objects
- The System.Exception class is base for all exceptions in CLR
 - Contains information for the cause of the error or the unusual situation
 - Message – text description of the exception
 - StackTrace – the snapshot of the stack at the moment of exception throwing
 - InnerException – exception caused the current exception (if any)

Types of Exceptions

- All .NET exceptions inherit from `System.Exception`
- The system exceptions inherit from `System.SystemException`, e.g.
 - `System.ArgumentException`
 - `System.NullReferenceException`
 - `System.OutOfMemoryException`
 - `System.StackOverflowException`
- User-defined exceptions should be inherit from `System.ApplicationException`

Exception Hierarchy

- Exceptions in .NET Framework are organized in a hierarchy



Exception Properties - Example

```
class ExceptionsTest
{
    public static void CauseFormatException()
    {
        string s = "an invalid number";
        Int32.Parse(s);
    }

    static void Main()
    {
        try
        {
            CauseFormatException();
        }
        catch (FormatException fe)
        {
            Console.Error.WriteLine("Exception caught: {0}\n{1}",
                fe.Message, fe.StackTrace);
        }
    }
}
```

Exception Properties (1 of 2)

- The Message property gives brief description of the problem
- The StackTrace property is extremely useful when identifying the reason behind the exception

Exception caught: Input string was not in a correct format.

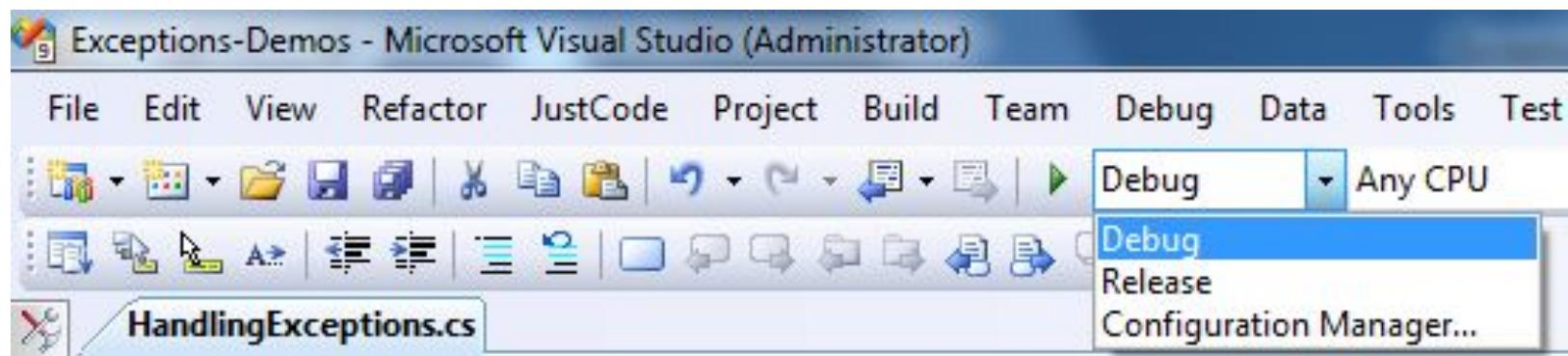
```
at System.Number.ParseInt32(String s, NumberStyles
style, NumberFormatInfo info)
at System.Int32.Parse(String s)
at ExceptionsTest.CauseFormatException() in
c:\consoleapplication1\exceptionstest.cs:line 8
at ExceptionsTest.Main(String[] args) in
c:\consoleapplication1\exceptionstest.cs:line 15
```

Exception Properties (2 of 2)

- File names and line numbers are accessible only if the compilation was in Debug mode
- When compiled in Release mode, the information in the property StackTrace is quite different:

Exception caught: Input string was not in a correct format.

```
at System.Number.ParseInt32(String s, NumberStyles
style, NumberFormatInfo info)
at ExceptionsTest.Main(String[] args)
```



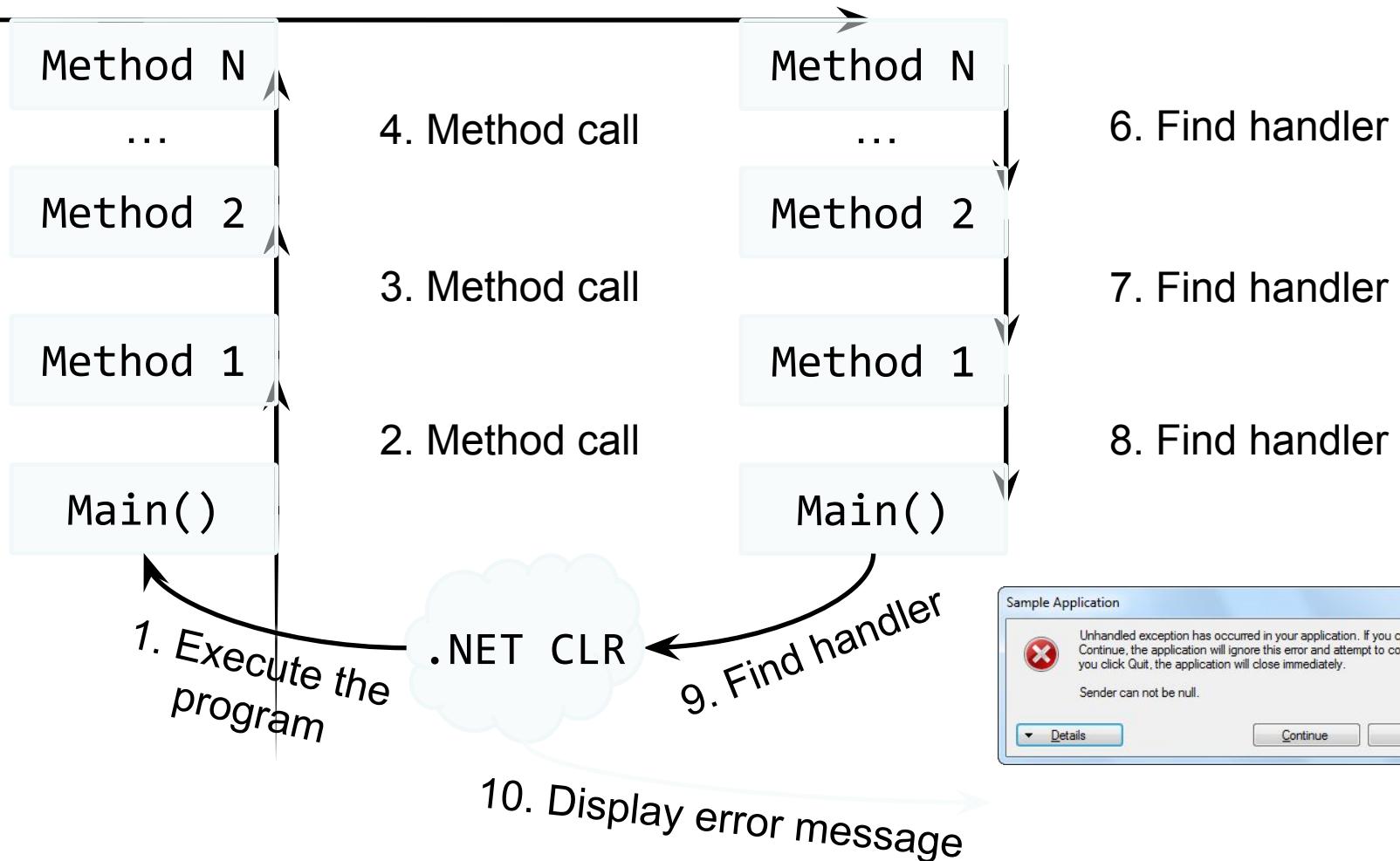
Throwing Exceptions

- Exceptions are thrown (raised) by throw keyword in C#
 - Used to notify the calling code in case of error or unusual situation
- When an exception is thrown:
 - The program execution stops
 - The exception travels over the stack until a suitable catch block is reached to handle it
- Unhandled exceptions will display an error message



How Exceptions Work?

5. Throw an exception



Using throw Keyword

- Throwing an exception with error message:

```
throw new ArgumentException("Invalid amount!");
```

- Exceptions can take a message and cause:

```
try
{
    Int32.Parse(str);
}
catch (FormatException fe)
{
    throw new ArgumentException("Invalid number", fe);
}
```

- If the original exception is not passed the initial cause of the exception is lost

Throwing Exceptions - Example

```
public static double Sqrt(double value)
{
    if (value < 0)
        throw new System.ArgumentOutOfRangeException(
            "Sqrt for negative numbers is undefined!");
    return Math.Sqrt(value);
}
static void Main()
{
    try
    {
        Sqrt(-1);
    }
    catch (ArgumentOutOfRangeException ex)
    {
        Console.Error.WriteLine("Error: " + ex.Message);
        throw;
    }
}
```

Checkpoint

- Which of the following statements are correct about exception handling in C#.NET?

1. *try* blocks cannot be nested.
2. An exception must be caught in the same function in which it is thrown.
3. All values set up in the exception object are available in the *catch* block.
4. While throwing a user-defined exception, multiple values can be set in the exception object.

- a. 1 only
- b. 2 only
- c. 3 and 4 only
- d. 3 and 4 only



Checkpoint

Checkpoint

- Which statements are correct about exception handling in C#.NET?
 1. If our program does not catch an exception then the .NET CLR catches it.
 2. It is possible to create user-defined exceptions.
 3. All types of exceptions can be caught using the *Exception* class.
 4. *CLRExceptions* is the base class for all exception classes.
 5. For every *try* block there must be a corresponding *finally* block.
 - b. 1 and 2 only
 - c. 1, 2 and 3 only
 - d. 4 and 5 only
 - e. All of the above



Checkpoint

Checkpoint

- Which of the following statements is correct about an Exception?
 - a. It occurs during compilation.
 - b. It occurs during linking.
 - c. It occurs at run-time.
 - d. It occurs during Just-In-Time compilation.
 - e. It occurs during loading of the program.



Checkpoint

Knowledge Check



Knowledge Check

1. Which of the following is not a .Net Exception class?
 - a. Exception
 - b. StackMemoryException
 - c. DividedByZeroException
 - d. OutOfMemoryException

2. In C#.NET if we do not catch the exception thrown at runtime then which of the following will catch it?
 - a. Compiler
 - b. CLR
 - c. Loader
 - d. Operating system

Knowledge Check

3. Which of the following statements are correct about exception handling in C#.NET?
1. If an exception occurs then the program terminates abruptly without getting any chance to recover from the exception.
 2. No matter whether an exception occurs or not, the statements in the *finally* clause (if present) will get executed.
 3. A program can contain multiple *finally* clauses.
 4. A *finally* clause is written outside the *try* block.
 5. *finally* clause is used to perform clean up operations like closing the network/database connections
- b. 1 only
- c. 2 only
- d. 2 and 5 only
- e. 3 and 4 only

Knowledge Check

4. All code inside *finally* block is guaranteed to execute irrespective of whether an exception occurs in the *protected* block or not.
 - a. True
 - b. False
5. Which statement is incorrect when creating own Exception Classes?
 - a. To create your own Exception that you can throw, you can extend the ApplicationException class or the Exception class.
 - b. In C#, you can throw any object you create if it is appropriate for the application.

Knowledge Check

6. Dealing with exceptional situations as an application executes is called
 - a. exception detection
 - b. exception handling
 - c. exception resolution
 - d. exception debugging
7. A(n) _____ is always followed by at least one catch block or a finally block.
 - a. if statement
 - b. event handler
 - c. try block
 - d. None of the above

Knowledge Check

8. The method call `Int32.Parse("123.4a")` will throw a(n) _____.
- a. FormatException
 - b. ParsingException
 - c. DivideByZeroException
 - d. None of the above
9. If no exceptions are thrown in a try block, _____.
- a. the catch block(s) are skipped
 - b. all catch blocks are executed
 - c. an error occurs
 - d. the default exception is thrown

Knowledge Check

10. A(n) _____ is an exception that does not have an exception handler, and therefore might cause the application to terminate execution.
- uncaught block
 - uncaught exception
 - error handler
 - Thrower
11. A try block can have associated with it.
- only one catch block
 - several finally blocks
 - one or more catch blocks
 - None of the above

Knowledge Check

12. The _____ statement is used to rethrow an exception from inside a catch block.
- a. rethrow
 - b. throw
 - c. try
 - d. catch
13. The exception you wish to handle should be declared as a parameter of the _____ block.
- a. try
 - b. catch
 - c. finally
 - d. None of the above

Knowledge Check

14. A finally block is located .
- after the try block, but before each catch block
 - before the try block
 - after the try block and the try block's corresponding catch blocks
 - Either b or c.
15. A _____ is executed if an exception is thrown from a try block or if no exception is thrown.
- Catch block
 - Finally block
 - Exception handler
 - All of the above

Module 11: Collections and Generics



Module Objectives

At the end of this module, you will be able to:

- Describe use of collection classes.
- Define and use generic types.
- Explain the concepts of Variance through covariance and contra variance.



Module objectives

Module Agenda

- Collections
- Generics
- Variance



Collections



Collection Overview

- Specialized Classes used for storage and retrieval.
- Usage:
 - Individual elements serve similar purposes and are of equal importance.
 - The number of elements is unknown or is not fixed at compile time.
 - You need to support iteration over all elements.
 - You need to support sorting of the elements.
 - You need to expose the elements from a library where a consumer will expect a collection type.



Common Collection Classes

Collection	Description
ArrayList	An unordered collection, similar to an array. Items are accessed by index
Queue	A first-in, first-out collection. Use the Enqueue method instead of Add
Stack	A first-in, last-out collection. Use the Push method instead of Add
Hashtable	A collection of key and value pairs. Suitable for large collections
SortedList	A collection of key and value pairs. Items are ordered based on the key

Iterating through a Collection

- A foreach loop displays every item in a collection in turn

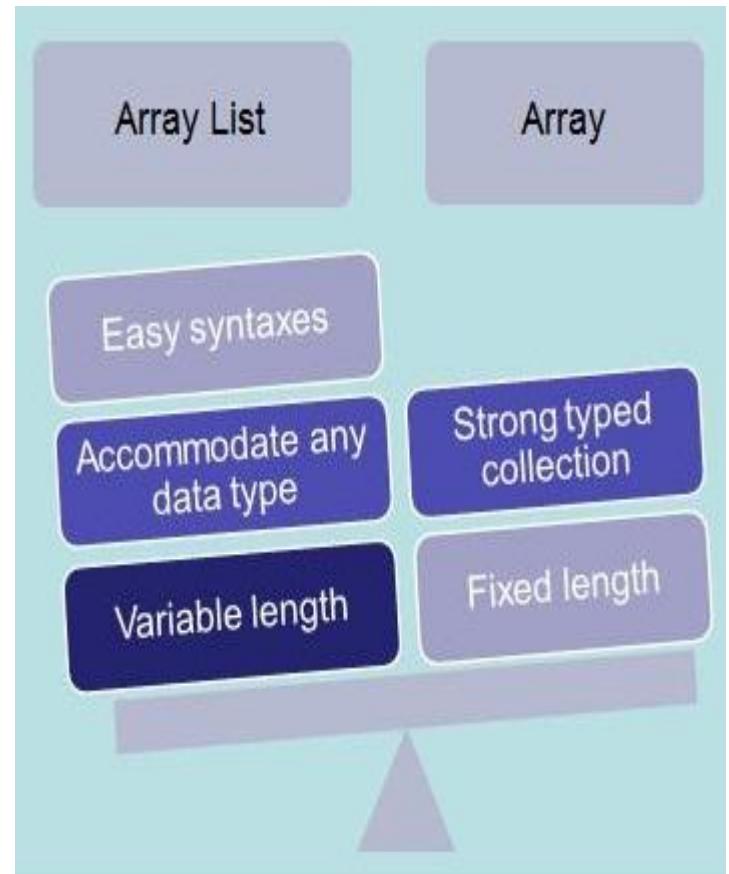
```
foreach(<type> <control_variable> in <collection>)
{
    <foreach_statement_body>
}
```

- To use foreach loop, the collection must expose an enumerator. The ICollection interfaces defines a GetEnumerator method

```
ArrayList list = new ArrayList();
list.Add(99);
list.Add(10001);
list.Add(25);
...
foreach (int i in list)
{
    Console.WriteLine(i);
}
```

ArrayList

- The **ArrayList collection class** is similar to an array. Adding items to the array and retrieve items by using a zero-based index.
- The **ArrayList class** dynamically increases in size as you add values to the collection. Does not automatically shrink when items are removed from the collection.
 - Use the **Capacity property** to get or set the current size of the collection.
 - Use the **TrimToSize method** to reduce the size of the collection, or alternatively, you can set the **Capacity property** to a lower value.

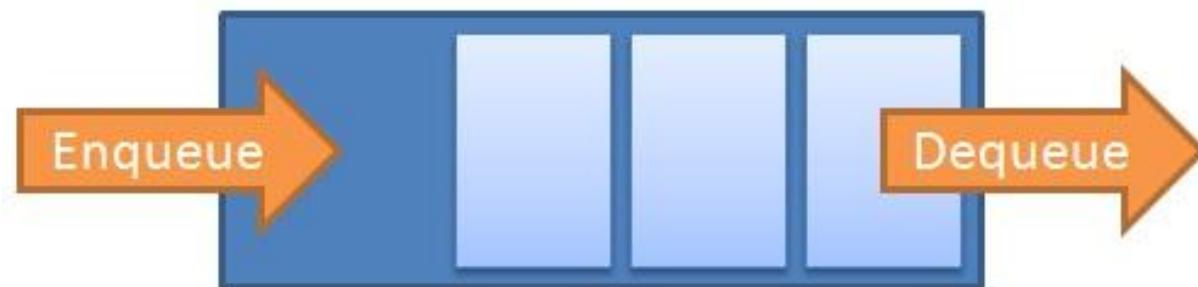


ArrayList - Example

```
ArrayList al = new ArrayList();
// Add values to the ArrayList collection.
al.Add("Value");
al.Add("Value 2");
al.Add("Value 3");
al.Add("Value 4");
// Remove a specific object from the ArrayList.
al.Remove("Value 2"); // Removes "Value 2"
// Remove an object from a specified index.
al.RemoveAt(2); // Removes "Value 4"
// Retrieve an object from a specified index.
string valueFromCollection = (string)al[1]; // Returns
"Value 3"
```

Queue

- The **Queue class** is a **First-In-First-Out(FIFO)** data structure.
 - **Enqueue()** method to automatically add the object to the end of the collection
 - **Dequeue()** method to automatically remove the object from the start of the collection.
 - **Peek()** method to retrieve the first item in the queue without removing it.
- The **Queue class** grows automatically as objects are added to the collection.

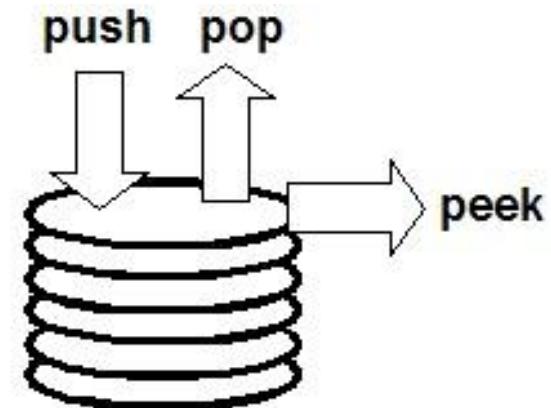


Queue - Example

```
Queue queue = new Queue();
// Add values to the Queue collection.
queue.Enqueue("Value");
queue.Enqueue("Value 2");
queue.Enqueue("Value 3");
queue.Enqueue("Value 4");
// Retrieve an object from the Queue collection.
string valueFromCollection = (string)queue.Dequeue();
```

Stack Collection

- The **Stack class** is a First-In-Last-Out data structure.
 - **Push() method** to add object to the start of the collection;
 - **Pop() method** to automatically removed the object from the start of the collection and returned.
 - **Peek() method** to return the item at the start of the Stack collection without removing it.
- The **Stack class** grows automatically as objects are added to the collection.



Stack - Example

```
Stack stack = new Stack();
// Add values to the Stack collection.
stack.Push("Value");
stack.Push("Value 2");
stack.Push("Value 3");
stack.Push("Value 4");
// Retrieve without removing it from the Stack collection.
string peekValueFromCollection = (string)stack.Peek();
// Returns "Value 4"
// Retrieve an object from the Stack collection.
string valueFromCollection = (string)stack.Pop();
// Returns "Value 4"
// Retrieve another object from the Stack collection.
string valueFromCollection2 = (string)stack.Pop();
// Returns "Value 3"
```

Hashtable Collection

- The **Hashtable class** enables to **store key and value pairs** in a rapid access collection.
 - **Add method**, provide both a key and a value. The key must be unique in the collection, but the value can be a duplicate.
- To retrieve a value from a **Hashtable class** by using an indexer and specify the key of the value that you want to retrieve.

Hash Table

Key	Value
1	New York
2	Boston
3	Mexico
4	Kansas
5	Detroit
6	California

Hashtable - Example

```
Hashtable hashtable = new Hashtable();
// Add values to the Hashtable collection.
hashtable.Add("Key A", "Value");
hashtable.Add("Key B", "Value 2");
hashtable.Add("Key C", "Value 3");
hashtable.Add("Key D", "Value 4");
// Remove an item from the Hashtable by specifying the key.
hashtable.Remove("Key C");
// Retrieve an item from the Hashtable collection
// by specifying the key.
string valueFromCollection = (string)hashtable["Key B"];
// Returns "Value 2""
```

SortedList Collection

- The **SortedList collection**, like a Hashtable class, class stores a collection of key/value object pairs.
- However, values in the collection are sorted by using the key. If you iterate through the data in a **SortedList** collection, the data will be presented in key order.

SortedList

Key	Value
1	Boston
2	California
3	Detroit
4	Kansas
5	Mexico
6	New York

SortedList - Example

```
SortedList sortedList = new SortedList();
// Add values to the SortedList collection.
sortedList.Add("Key A", "Value");
sortedList.Add("Key B", "Value 2");
sortedList.Add("Key C", "Value 3");
sortedList.Add("Key D", "Value 4");
// Remove an item by specifying a key from the SortedList.
sortedList.Remove("Key C");
// Retrieve an item by specifying a key from the SortedList.
string valueFromCollection = (string)sortedList["Key B"];
// Returns "Value 2".
// Retrieve an item from the SortedList collection
// by specifying the index.
string valueFromCollection2 = (string)sortedList.GetByIndex(0);
// Returns "Value".
```

Collection Initializers

- Use Add method to add items to a collection

```
ArrayList a1 = new ArrayList();
a1.Add('Value');
a1.Add('Another Value');
```

- Use collection initializer to add items when defining the collection

```
//person1 and person2 are instantiated person objects
ArrayList a12 = new ArrayList() {
    person1,
    person2
};
```

- Combine collection initializers with object initializers

```
ArrayList a13 = new ArrayList() {
    new person() {Name="Tom", Age =20},
    new person() {Name="Jerry", Age =10},
};
```

Generics



What are Generic Types?

- A generic type is a type that specifies one or more type parameters
- Type parameters are like parameters except that they represent a type, not an instance type
 - Defined by using angle brackets after the class name
- This example shows the definition of class named List that takes a single type parameter named T. You can use T like any other type in the class.

```
public class List<T>
```

Compiling Generic Types and Type Safety

- The `List<T>` class is used several times with different type parameters

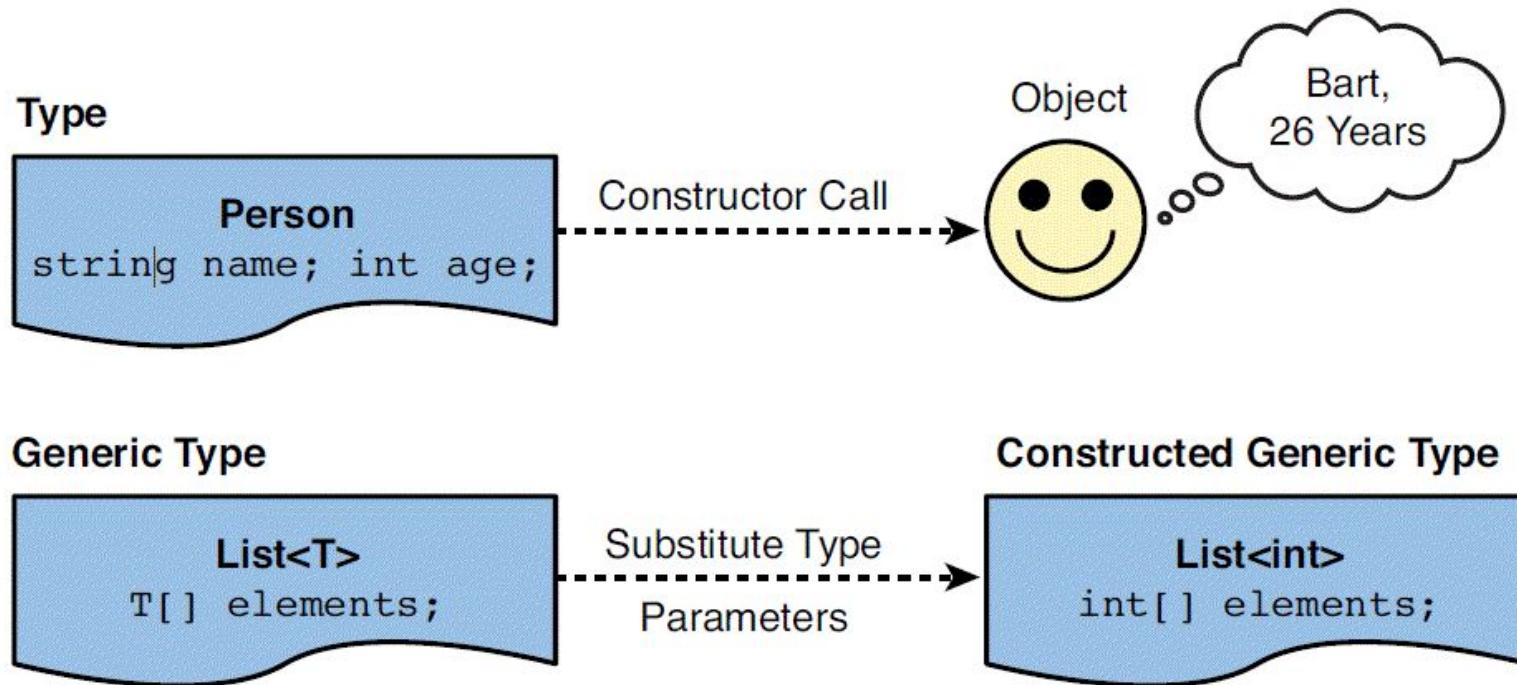
```
List<string> names = new List<string>();  
names.Add("John");  
...  
string name = names[0];  
  
List<List<string>> listOfLists = new List<List<string>>();  
listOfLists.Add(names);  
...  
List<string> data = listOfLists[0];
```

- The compiler generates a strongly typed equivalent of the generic class, effectively generating the following methods

```
public void Add(string item)  
...  
public void Add(List<string> item)
```

Generics

- Analogies between constructing objects and constructing generic types.



Generics – Benefits (1 of 2)

- Without generics, typing needs to be enforced at runtime.

```
static void Main()
{
    ArrayList xs = new ArrayList();
    xs.Add(1);
    xs.Add(2);
    xs.Add(3);

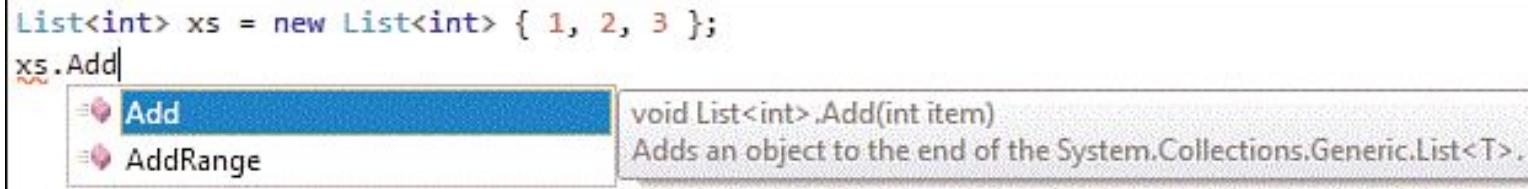
    // An evil guy messing with the collection...
    xs[0] = "Should not be here";

    // This cast now blows up...
    int first = (int)xs[0];
}
```

The screenshot shows a C# code editor with a tooltip overlay. The tooltip is titled 'InvalidOperationException was unhandled' and contains the message 'Specified cast is not valid.' Below this, under 'Troubleshooting tips:', it says 'When casting from a number, the value must be a number less than infinity.' and 'Make sure the source type is convertible to the destination type.' It also provides a link to 'Get general help for this exception.' At the bottom of the tooltip, there are links for 'Search for more Help Online...' and 'Actions:' followed by 'View Detail...' and 'Copy exception detail to the clipboard.'

Generic – Benefits (2 of 2)

- Generics improve static type checking.



- Better compile-time checking facilitated by using generic types

A screenshot of an IDE showing a code editor and an Error List window. The code editor contains the following C# code:

```
static void Main()
{
    List<int> xs = new List<int> { 1, 2, 3 };

    // An evil guy trying to mess with the collection...
    xs[0] = "Should not be here";

    // No need to cast here anymore...
    int first = xs[0];
}
```

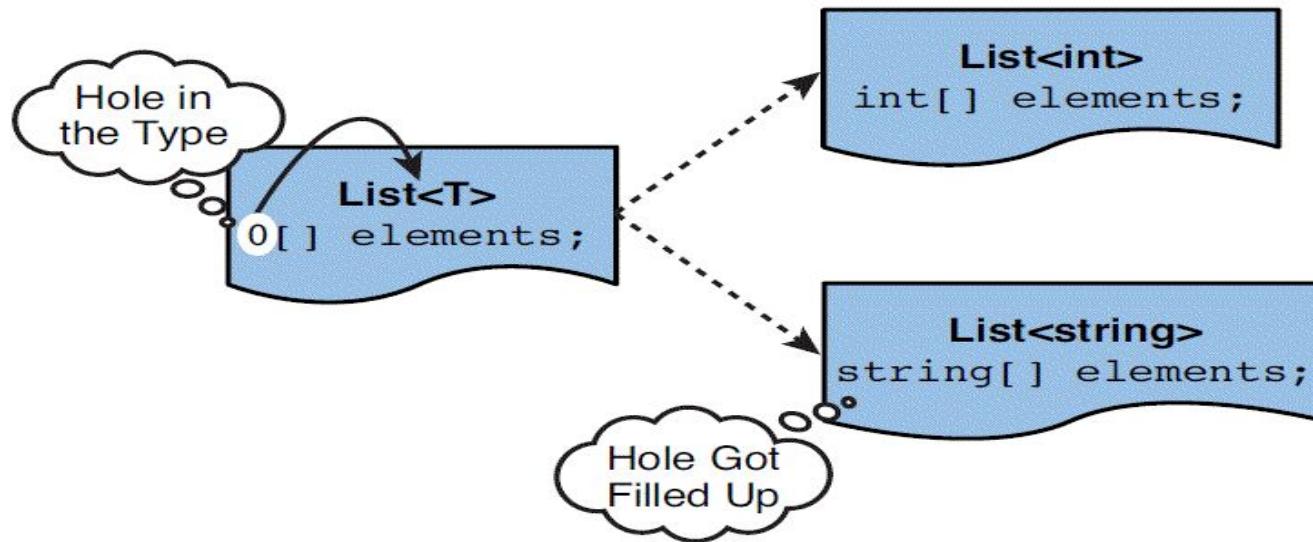
The Error List window shows one error:

Description
1 Cannot implicitly convert type 'string' to 'int'

Generics with Visual Studio Help

- Creating a constructed generic type with help from Visual Studio.

```
var numbers = new Dictionary<  
    TKey, TValue>;  
    class System.Collections.Generic.Dictionary<TKey, TValue>  
    Represents a collection of keys and values.  
    TKey: The type of the keys in the dictionary.
```



Generic Classes

- Collection classes
- Collection interfaces
- Collection base classes
- Utility classes
- Reflection

`List<T>`

`Dictionary<K,V>`

`SortedDictionary<K,V>`

`Stack<T>`

`Queue`

`IList<T>`

`IDictionary<K,V>`

`ICollection<T>`

`IEnumerable<T>`

`IEnumerator<T>`

`Comparable<T>`

`IComparer<T>`

`Collection<T>`

`KeyedCollection<T>`

`ReadOnlyCollection<T>`

`Nullable<T>`

`EventHandler<T>`

`Comparer<T>`

Defining a Custom Generic Type (1 of 2)

- Define a class and specify a type parameter in angle brackets after the class name

```
class PrintableCollection<TItem>
{
    // type parameter for fields and properties
    TItem [] data;
    int index;
    ...

    // type parameter for methods
    public void Insert(TItem item)
    {
        ...
        data[index] = item;
        ...
    }
}
```

Defining a Custom Generic Type (2 of 2)

- When naming Generic type parameters
 - Have a descriptive name that would have value.
 - Consider using T as type parameter for types with one single letter type parameter
 - Prefix descriptive type parameter names with “T”

Adding Constraints to Generic Types

Constraint	Description
where T: struct	The type argument must be a value type
where T : class	The type argument must be a reference type
where T : new()	The type argument must have a public default constructor
where T : <base class name>	The type argument must be, or derive from, the specified base class
where T : <interface name>	The type argument must be, or implement, the specified interface
where T : U	The type argument that is supplied for T must be, or derive from, the argument that is supplied for U

Defining Generic Interfaces - Example

- Generic interface with type parameters

```
interface IPrinter<DocumentType>
    where DocumentType : IPrintable
{
    void PrintDocument(DocumentType Document);

    PrintPreview PreviewDocument(DocumentType Document)
}
```

- Generic class that implements the generic interface

```
class Printer<DocumentType> : IPrintable<DocumentType>
    where DocumentType : Iprintable {
    public void PrintDocument(DocumentType Document) {
        // Send document to printer.
        PrintService.Print((IPrintable)Document);
    }

    public PrintPreview PreviewDocument(DocumentType Document) {
        // Return a new PrintPreview object.
        return new PrintPreview((IPrintable)Document);
    }
}
```

Generics Collection Initializers

- Example 1

```
var numbers = new Dictionary<string, int>
{
    { "Bart" , 415 },
    { "John" , 613 }
};

foreach(KeyValuePair<string, int> in numbers)
{
    // do something here
}
```

- Example 2

```
var _temp = new Dictionary<string, int>();
_temp.Add("John",415);
_temp.Add("Bart",613);

var numbers = _temp;
```

The List Abstract Data Type

- Data structure (container) that contains a sequence of elements
 - Can have variable size
 - Elements are arranged linearly, in sequence
- Can be implemented in several ways
 - Statically (using array □ fixed size)
 - Dynamically (linked implementation)
 - Using resizable array (the List<T> class)

The List<T> Class

- Implements the abstract data structure list using an array
 - All elements are of the same type T
 - T can be any type, e.g. List<int>, List<string>, List<DateTime>
 - Size is dynamically increased as needed
- Basic functionality
 - Count – returns the number of elements
 - Add(T) – appends given element at the end

List<T> - Example

```
static void Main()
{
    List<string> list = new List<string>();

    list.Add("C#");
    list.Add("Java");
    list.Add("PHP");

    foreach (string item in list)
    {
        Console.WriteLine(item);
    }

    // Result:
    //      C#
    //      Java
    //      PHP
```

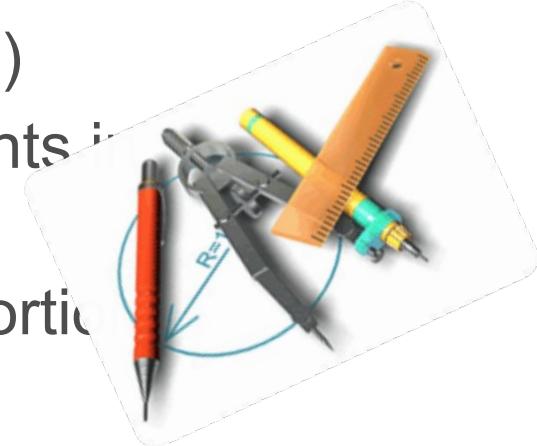
List<T> Functionality (1 of 2)

- `list[index]` – access element by index
- `Insert(index, T)` – inserts given element to the list at a specified position
- `Remove(T)` – removes the first occurrence of a given element
- `RemoveAt(index)` – removes the element at the specified position
- `Clear()` – removes all elements
- `Contains(T)` – determines whether an element is present in the list



List<T> Functionality (2 of 2)

- `IndexOf()` – returns the index of the first occurrence of a value in the list(zero-based)
- `Reverse()` – reverse the order of the elements in the list or a portion of it
- `Sort()` – sorts the elements in the list or a portion of it
- `ToArray()` – converts the elements of the list to array
- `TrimExcess()` – sets the capacity of the actual number of elements

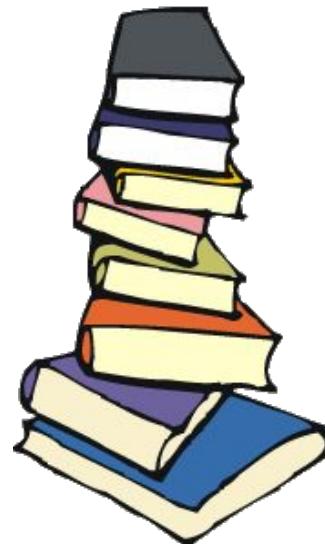


List<T> - Example

```
static List<int> FindPrimes(int start, int end)
{
    List<int> primesList = new List<int>();
    for (int num = start; num <= end; num++)
    {
        bool prime = true;
        for (int div = 2; div <= Math.Sqrt(num); div++)
        {
            if (num % div == 0)
            {
                prime = false;
                break;
            }
        }
        if (prime)
        {
            primesList.Add(num);
        }
    }
}
```

The Stack<T> Class

- Implements the stack data structure using an array
 - Elements are of the same type T
 - T can be any type, e.g. Stack<int>
 - Size is dynamically increase as needed
- Basic functionality
 - Push(T) – inserts elements to the stack
 - Pop() – removes and returns the top element from the stack



Stack<T> - Example

```
static void Main()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("1. John");
    stack.Push("2. Mary");
    stack.Push("3. Lisa");
    stack.Push("4. Tim");
    Console.WriteLine("Top = {0}", stack.Peek());
    while (stack.Count > 0)
    {
        string personName = stack.Pop();
        Console.WriteLine(personName);
    }
}
```

The Queue<T> Class

- Implements the queue data structure using a circular resizable array
 - Elements are from the same type T
 - T can be any type, e.g. Stack<int>
 - Size is dynamically increased as needed
- Basic functionality
 - Enqueue(T) – adds an element to the end of the queue
 - Dequeue(T) – removes and returns the element at the beginning of the queue



Queue<T> - Example

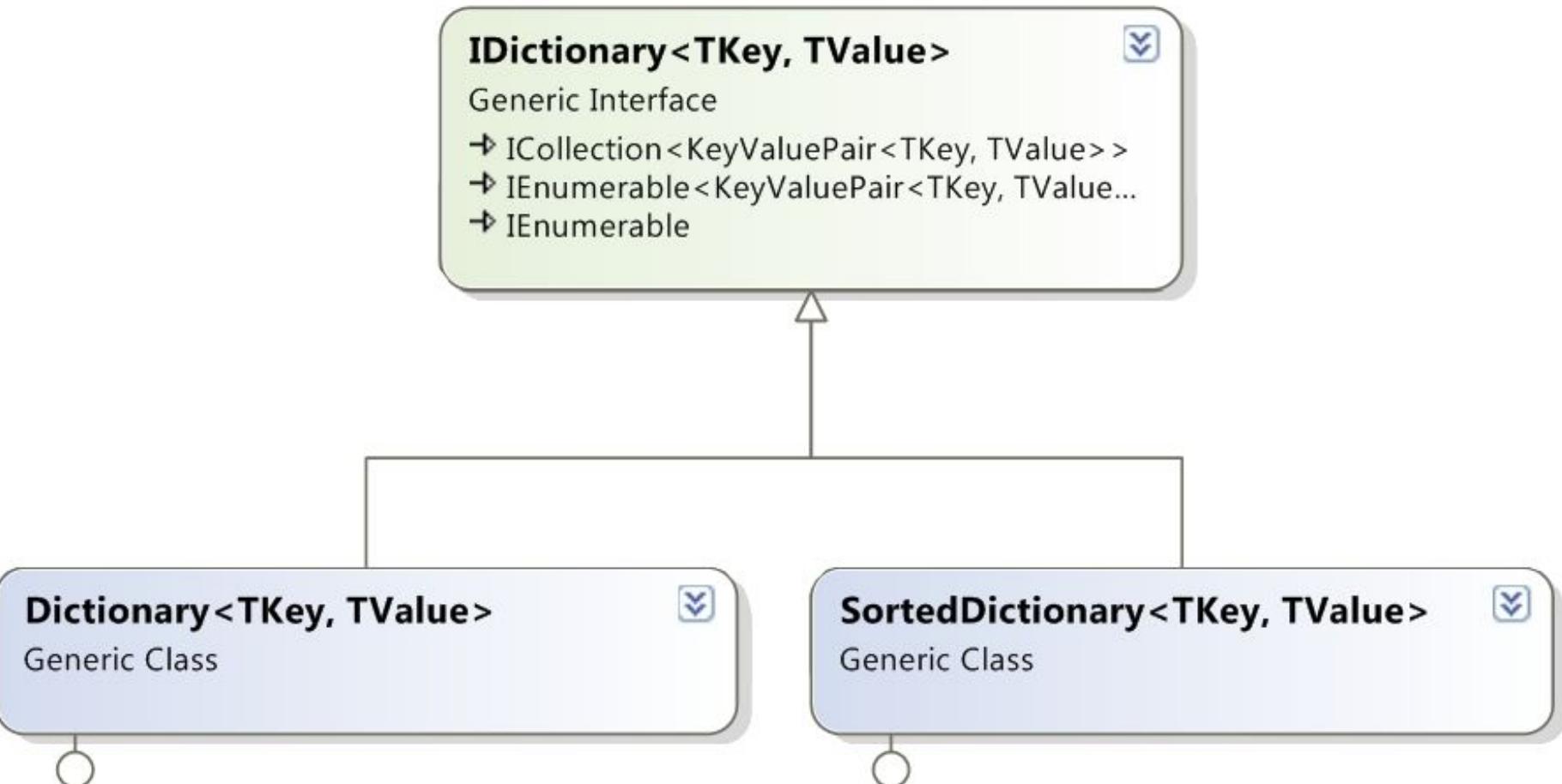
```
static void Main()
{
    Queue<string> queue = new Queue<string>();
    queue.Enqueue("Message One");
    queue.Enqueue("Message Two");
    queue.Enqueue("Message Three");
    queue.Enqueue("Message Four");
    while (queue.Count > 0)
    {
        string message = queue.Dequeue();
        Console.WriteLine(message);
    }
}
```

The Dictionary ADT

- The abstract data type (ADT) “dictionary” maps key to values
 - Also known as “map” or “associative array”
 - Contains a set of(key, value) pairs
- Dictionary ADT operations
 - Add(key, value)
 - FindByKey(Key)
 - Delete(Key)



Dictionaries – Interfaces and Implementations



Dictionary<Tkey, Tvalue> Class

- Implements the abstract data type “Dictionary” as hashtable
 - Size is dynamically increased as needed
 - Contains a collection of key-value pairs arranged by the hash code of the k – $h(\text{key}) = \text{value}$
 - Collisions are resolved by chaining
- Dictionary<Tkey,Tvalue> class relies on
 - Object.Equals() method for comparing elements
 - Object.GetHashCode() method for calculating the hash codes of the elements

Dictionary<Tkey, Tvalue> Operations (1 of 2)

- Add(TKey, TValue) – adds an element with the specified key and value into the dictionary
- Remove(Tkey) – removes the element with the specified key
- Clear() – removes all elements
- this[] – returns element by key
- Count – returns the number of elements

Dictionary<Tkey, Tvalue> Operations (2 of 2)

- Contains(TKey) – determines whether the dictionary contains given key
- ContainsValue(TValue) – determines whether the dictionary contains given value
- Keys – returns a collection of keys
- Values – returns a collection of the values
- TryGetValue(Tkey,out TValue) – if the key is found, returns it in the TValue, otherwise returns the default value for the TValue type

Dictionary< TKey, TValue > - Example 1

```
Dictionary<string, int> studentsMarks
    = new Dictionary<string, int>();
studentsMarks.Add("Ivan", 4);
studentsMarks.Add("Peter", 6);
studentsMarks.Add("Maria", 6);
studentsMarks.Add("George", 5);

int peterMark = studentsMarks["Peter"];
Console.WriteLine("Peter's mark: {0}", peterMark);
Console.WriteLine("Is Peter in the hash table: {0}",
    studentsMarks.ContainsKey("Peter"));

Console.WriteLine("Students and grades:");
foreach (var pair in studentsMarks)
{
    Console.WriteLine("{0} --> {1} ", pair.Key, pair.Value);
```

Dictionary< TKey, TValue> - Example 2

```
Dictionary<string, int> studentsMarks
    = new Dictionary<string, int>();
studentsMarks.Add("Ivan", 4);
studentsMarks.Add("Peter", 6);
studentsMarks.Add("Maria", 6);
studentsMarks.Add("George", 5);

int peterMark = studentsMarks["Peter"];
Console.WriteLine("Peter's mark: {0}", peterMark);
Console.WriteLine("Is Peter in the hash table: {0}",
    studentsMarks.ContainsKey("Peter"));

Console.WriteLine("Students and grades:");
foreach (var pair in studentsMarks)
{
    Console.WriteLine("{0} --> {1}", pair.Key, pair.Value);
```

The SortedDictionary<TKey, TValue> Class

- `SortedDictionary<TKey, TValue>` implements the ADT "dictionary" as self-balancing search tree
 - Elements are arranged in the tree ordered by key
 - Traversing the tree returns the elements in increasing order
 - Add / Find / Delete perform $\log_2(n)$ operations
- Use `SortedDictionary<TKey, TValue>` when you need the elements sorted
 - Otherwise use `Dictionary<TKey, TValue>` – it has better performance

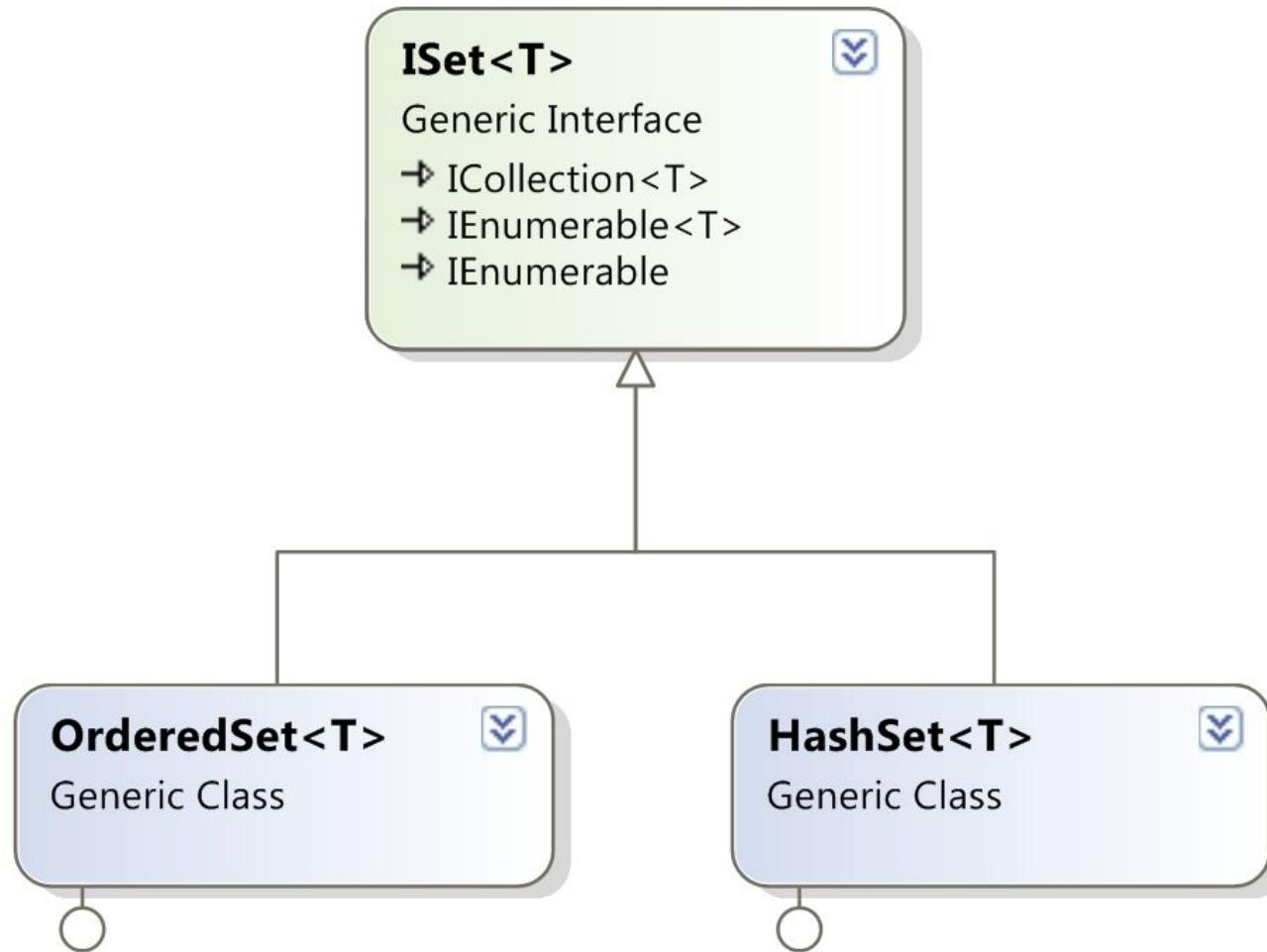
SortedDictionary< TKey, TValue > - Example

```
string text = "a text, some text, just some text";
IDictionary<string, int> wordsCount =
    new SortedDictionary<string, int>();
string[] words = text.Split(' ', ',', '.', '.');
foreach (string word in words)
{
    int count = 1;
    if (wordsCount.ContainsKey(word))
        count = wordsCount[word] + 1;
    wordsCount[word] = count;
}
foreach(var pair in wordsCount)
{
    Console.WriteLine("{0} -> {1}", pair.Key, pair.Value);
}
```

Set and Bag

- The abstract data type (ADT) "set" keeps a set of elements with no duplicates
- Sets with duplicates are also known as ADT "bag"
- Set operations:
 - Add(element)
 - Contains(element)
 - Delete(element)
 - Union(set) / Intersect(set)

Sets – Interfaces and Implementations



The HashSet<T> Class

- HashSet<T> implements ADT set by hash table
 - Elements are in no particular order
- All major operations are fast:
 - Add(element) – appends an element to the set
 - Does nothing if the element already exists
 - Remove(element) – removes given element
 - Count – returns the number of elements
 - UnionWith(set) / IntersectWith(set) – performs union / intersection with another set

HashSet<T> - Example

```
ISet<string> firstSet = new HashSet<string>(
    new string[] { "SQL", "Java", "C#", "PHP" });
ISet<string> secondSet = new HashSet<string>(
    new string[] { "Oracle", "SQL", "MySQL" });

ISet<string> union = new HashSet<string>(firstSet);
union.UnionWith(secondSet);

PrintSet(union); // SQL Java C# PHP Oracle MySQL

private static void PrintSet<T>(ISet<T> set)
{
    foreach (var element in set)
    {
        Console.Write("{0} ", element);
    }
    Console.WriteLine();
```

The SortedSet<T> Class

- SortedSet<T> implements ADT set by balanced search tree
- Elements are sorted in increasing order
- Example:

```
ISet<string> firstSet = new SortedSet<string>(  
    new string[] { "SQL", "Java", "C#", "PHP" });  
  
ISet<string> secondSet = new SortedSet<string>(  
    new string[] { "Oracle", "SQL", "MySQL" });  
  
ISet<string> union = new HashSet<string>(firstSet);  
union.UnionWith(secondSet);  
  
PrintSet(union); // C# Java PHP SQL MySQL Oracle
```

What are Generic Methods and Delegates?

- Instead of duplicating methods with different parameters, use a generic method with a type parameter

```
void AddToQueue<DocumentType>(DocumentType document)
{
    printQueue.Add(report);
}
```

```
delegate void
PrintDocumentDelegate<DocumentType>(DocumentType document);
```

Using Generic Methods (1 of 2)

- To invoke generic method, specify the type parameter

```
T PerformUpdate<T>(T input)
{
    T output = // Update parameter.
    return output;
}
...

string result = PerformUpdate<string>("Test");
int result2 = PerformUpdate<int>(1);
```

Using Generic Methods (2 of 2)

- When a generic method is invoked, the compiler generates a concrete method for each invocation

```
string PerformUpdate(string input) // You cannot call this
method.
{
    string output = // Update parameter.
    return output;
}

int PerformUpdate(int input) // You cannot call this method.
{
    int output = // Update parameter.
    return output;
}
```

Variance

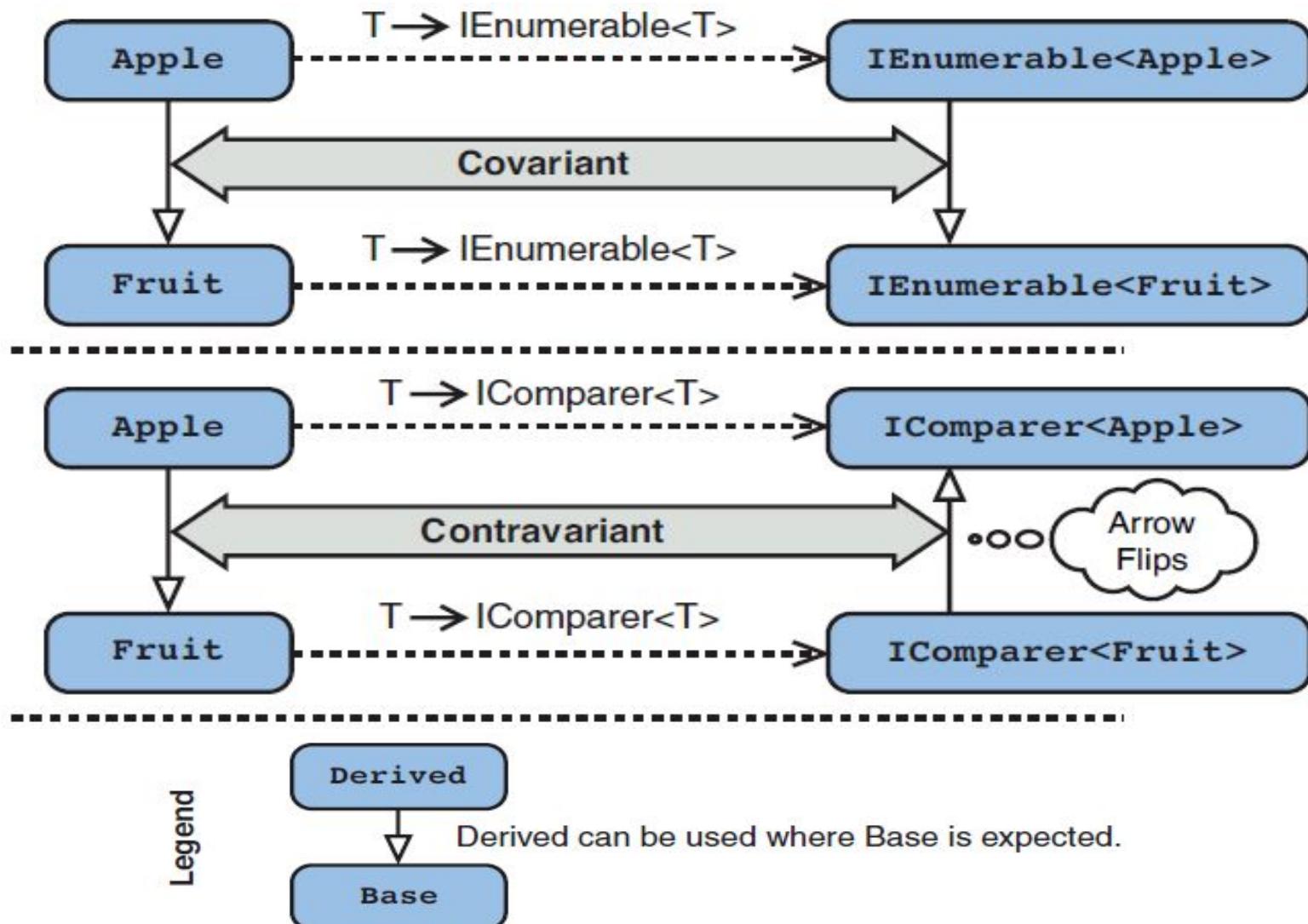


Variance

- Variance enables implicit reference conversion for array types, delegate types and generic interfaces.
- Classes and Structs does not support variance.
- Type variance is broken down into:
 - Covariance - an instance of a subclass can be used when an instance of a parent class is expected. Uses OUT keyword.
 - Contravariance - an instance of a superclass can be used when an instance of a subclass is expected. Uses IN keyword.
- Invariance prevents casting type parameters with other types in the inheritance hierarchy



Variance



Generic Interface - Covariance

- Example:

```
// This generic interface supports covariance.  
public interface IMyCoVarGenIF<out T> {  
    T GetObject();  
}  
// Implement the IMyCoVarGenIF interface.  
class MyClass<T> : IMyCoVarGenIF<T> {  
    T obj;  
    public MyClass(T v) { obj = v; }  
  
    public T GetObject() { return obj; }  
}  
static void Main()  
{  
    // Create a IMyCoVarGenIF reference to a MyClass<Alpha> object.  
    IMyCoVarGenIF<Alpha> AlphaRef =  
        new MyClass<Alpha>(new Alpha("Alpha #1"));  
    Console.WriteLine("Name of object referred to by AlphaRef is " +  
        AlphaRef.GetObject().GetName());  
    // Now create a MyClass<Beta> object and assign it to AlphaRef.  
    // *** This line is legal because of covariance. ***  
    AlphaRef = new MyClass<Beta>(new Beta("Beta #1"));  
    Console.WriteLine("Name of object referred to by AlphaRef is now " +  
        AlphaRef.GetObject().GetName());  
}
```

Generic Interface - Contravariance

- Example

```
// This generic interface supports contravariance.
public interface IMyContraVarGenIF<in T> {
    void Show(T obj);
}

// Implement the IMyContraVarGenIF interface.
class MyClass<T> : IMyContraVarGenIF<T> {
    public void Show(T x) { Console.WriteLine(x); }
}

static void Main()
{
    // Create an IMyContraVarGenIF<Alpha> reference to a MyClass<Alpha> object.
    IMyContraVarGenIF<Alpha> AlphaRef = new MyClass<Alpha>();
    // Create an IMyContraVarGenIF<beta> reference to a MyClass<Beta> object.
    IMyContraVarGenIF<Beta> BetaRef = new MyClass<Beta>();
    // Create an IMvContraVarGenIF<beta> reference to a MyClass<Alpha> object.
    // *** This is legal because of contravariance. ***
    IMvContraVarGenIF<Beta> BetaRef2 = new MvClass<Alpha>();
    BetaRef.Show(new Beta());
    // Assign AlphaRef to BetaRef.
    // *** This is legal because of contravariance. ***
    BetaRef = AlphaRef;
    BetaRef.Show(new Beta());
}
```

Variant Delegates

- Example:

```
// Declare a generic delegate that is contravariant on T.
delegate bool SomeOp<in T>(T obj);
// Declare a generic delegate that is covariant on T.
delegate T AnotherOp<out T, V>(V obj);
static void Main() {
    Alpha objA = new Alpha(4);
    Beta objB = new Beta(9);
    // First demonstrate contravariance.
    SomeOp<Alpha> checkIt = IsEven;
    // Declare a SomeOp<Beta> delegate.
    SomeOp<Beta> checkIt2;
    // *** This is legal only because of contravariance. ***
    checkIt2 = checkIt;
    // Call through the delegate.
    Console.WriteLine(checkIt2(objB));
    // Now, demonstrate covariance.
    // Here, the return type is Beta and the parameter type is Alpha.
    AnotherOp<Beta, Alpha> modifyIt = ChangeIt;
    // Here, the return type is Alpha and the parameter type is Alpha.
    AnotherOp<Alpha, Alpha> modifyIt2;
    // *** This statement is legal only because of covariance. ***
    modifyIt2 = modifyIt;
    // Actual call of the method and display the results.
    objA = modifyIt2(objA);
    Console.WriteLine(objA.Val);
}
```

Checkpoint

- In which of the following collections is the Input/Output based on a key?
 1. Map
 2. Stack
 3. BitArray
 4. HashTable
 5. SortedList
 - b. 1 and 2 only
 - c. 2 and 3 only
 - d. 1, 2 and 3 only
 - e. 4 and 5 only
 - f. All of the above



Checkpoint

Checkpoint

- Which of the following statements are correct about the C#.NET code snippet given below?

```
Stack st = new Stack();  
st.Push("hello");  
st.Push(8.2);  
st.Push(5);  
st.Push('b');  
st.Push(true);
```

- a. Dissimilar elements like "hello", 8.2, 5 cannot be stored in the same *Stack* collection.
- b. Boolean values can never be stored in *Stack* collection.
- c. This is a perfectly workable code.



Checkpoint

Checkpoint

- Which one of the following classes are present *System.Collections.Generic* namespace?
 1. Stack
 2. Tree
 3. SortedDictionary
 4. SortedArray
 - b. 1 and 2 only
 - c. 2 and 4 only
 - d. 1 and 3 only
 - e. All of the above
 - f. None of the above



Checkpoint

Checkpoint

- Which statement is incorrect?
 - a. Covariance and Contravariance enable implicit reference conversion for array types, delegate types, and generic interface.
 - b. Covariance preserves assignment compatibility and contravariance reverses it.
 - c. Arrays are covariant since C# 1.0.
 - d. Classes and structs support variance in C# 4.0.



Checkpoint

Knowledge Check



Knowledge Check

1. You create an instance of the Stack class. After adding several integers to it, you need to remove all objects from the Stack. Which method should you call?
 - a. Stack.Pop
 - b. Stack.Push
 - c. Stack.Clear
 - d. Stack.Peek

Knowledge Check

2. You need to create a collection to act as a shopping cart. The collection will store multiple instances of your custom class, ShoppingCartItem. You need to be able to sort the items according to price and time added to the shopping cart (both properties of the ShoppingCartItem). Which class should you use for the shopping cart?
- a. Queue
 - b. ArrayList
 - c. Stack
 - d. StringCollection

Knowledge Check

3. You are creating a collection that will act as a database transaction log. You need to be able to add instances of your custom class, DBTransaction, to the collection. If an error occurs, you need to be able to access the most recently added instance of DBTransaction and remove it from the collection. The collection must be strongly typed. Which class should you use?
- a. HashTable
 - b. SortedList
 - c. Stack
 - d. Queue

Knowledge Check

4. The classes in the System.Collections namespace store what type?
 - a. Base
 - b. Collection
 - c. Object
 - d. None of the above

5. The Hashtable class implements which abstraction?
 - a. List
 - b. Set
 - c. Map

Knowledge Check

6. How many enumerators will exist if four threads are simultaneously working on an *ArrayList* object?
 - a. 1
 - b. 3
 - c. 2
 - d. 4
7. A *HashTable* *t* maintains a collection of names of states and capital city of each state. Which of the following is the correct way to find out whether "Pasko" state is present in this collection or not?
 - a. *t.ContainsKey("Pasko");*
 - b. *t.HasValue("Pasko");*
 - c. *t.HasKey("Pasko");*
 - d. *t.ContainsValue("Pasko");*

Knowledge Check

8. Which of the following statements are correct about the *Stack* collection?
1. It can be used for evaluation of expressions.
 2. All elements in the *Stack* collection can be accessed using an enumerator.
 3. It is used to maintain a FIFO list.
 4. All elements stored in a *Stack* collection must be of similar type.
 5. Top-most element of the *Stack* collection can be accessed using the *Peek()* method.
- b. 1 and 2 only
- c. 3 and 4 only
- d. 1, 2 and 5 only
- e. All of the above

Knowledge Check

9. Which of the following statements are correct about a *HashTable* collection?
1. It is a keyed collection.
 2. It is a ordered collection.
 3. It is an indexed collection.
 4. It implements a *IDictionaryEnumerator* interface in its inner class.
 5. The key - value pairs present in a *HashTable* can be accessed using the *Keys* and *Values* properties of the inner class that implements the *IDictionaryEnumerator* interface.
- b. 1 and 2 only
- c. 1, 2 and 3 only
- d. 4 and 5 only
- e. 1, 4 and 5 only

Knowledge Check

10. Which of the following is the correct way to find out the number of elements currently present in an *ArrayList* Collection called arr?
- a. arr.Count
 - b. arr.GrowSize
 - c. arr.MaxIndex
 - d. arr.Capacity
 - e. arr.UpperBound

Module 12: InputOutput Streams



Module Objectives

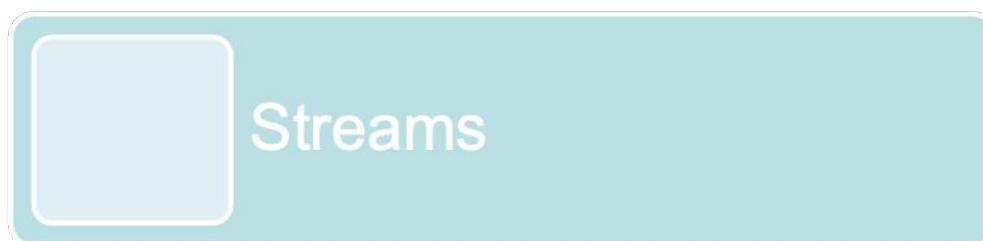
At the end of this module, you will be able to:

- Describe how to read and write files by using streams.
- Define serialization



Module objectives

Module Agenda

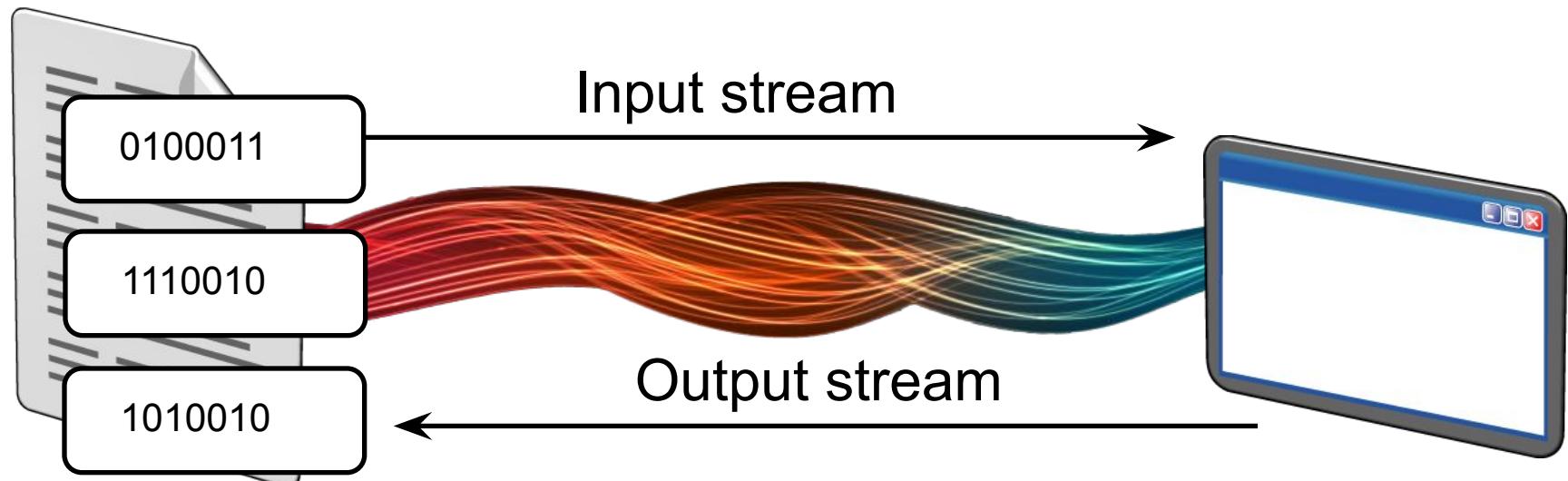


Streams



What is a Stream?

- A stream is a mechanism that enables you to manipulate data in manageable chunks



Stream Basics

- Streams are used for reading and writing data into and from devices
- Streams are ordered sequences of bytes
 - Provide consecutive access to its elements
- Different types of streams are available to access different data sources
 - File access, Network access, Memory streams, etc.
- Streams are open before using them and closed after that



Manipulating Files and Directories

- The System.IO namespace contains many classes that simplify interactions with the file system, such as the File and FileInfo classes
- The System.IO namespace contains the Directory and DirectoryInfo classes to help simplify interactions with directories

File class	FileInfo class
Copy()	CopyTo()
Create()	Delete()
Delete()	Length
Exists()	Open()

Accessing the File System

- Constructing a FileStream

- Read-Only

```
FileStream fs1 = File.OpenRead ("readme.bin");
```

- Write-Only

```
FileStream fs2 = File.OpenWrite (@"c:\temp\writeme.tmp");
```

- Read/Write

```
var fs = new FileStream ("readwrite.tmp", FileMode.Open);
```

- Specifying a file name

```
string baseFolder = AppDomain.CurrentDomain.BaseDirectory;
string logoPath = Path.Combine (baseFolder, "logo.jpg");
Console.WriteLine (File.Exists (logoPath));
```

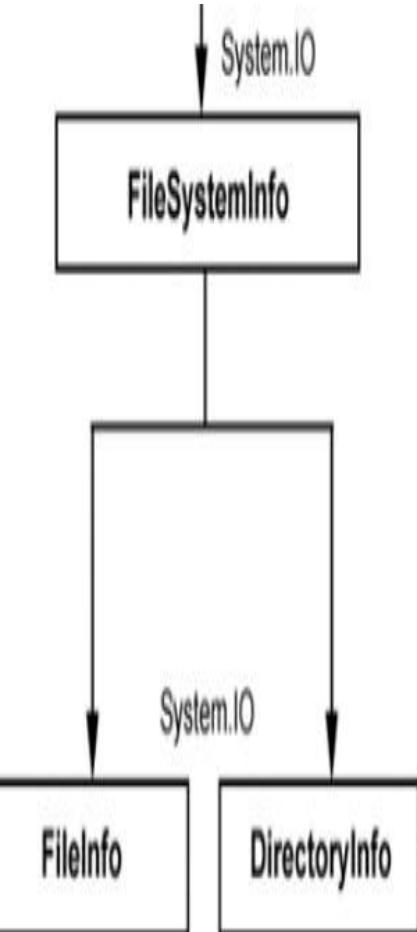
Manipulating Directories

- Directory class

```
string dirPath = @"C:\Users\Student\MyDirectory";
...
Directory.CreateDirectory(dirPath);
Directory.Delete(dirPath);
string[] dirs = Directory.GetDirectories(dirPath);
string[] files = Directory.GetFiles(dirPath);
```

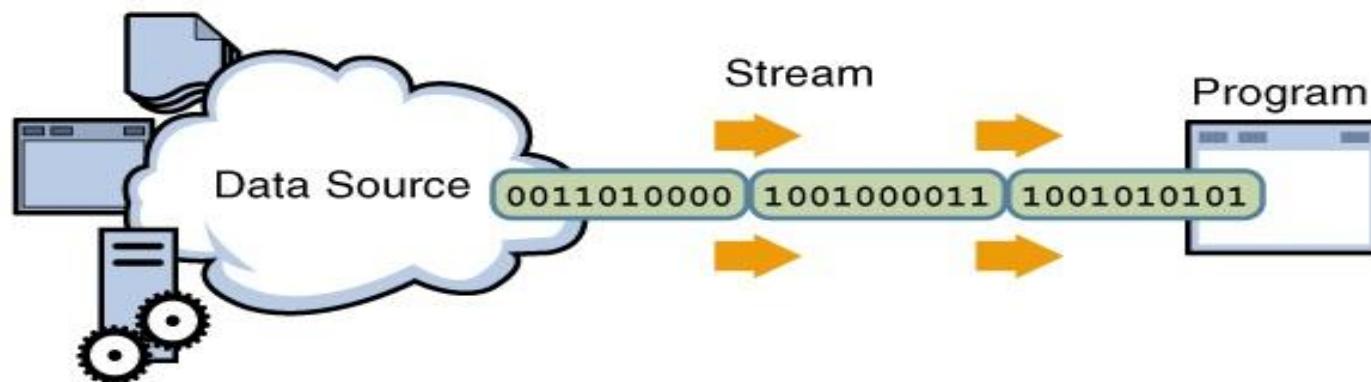
- DirectoryInfo class

```
string dirPath = @"C:\Users\Student\MyDirectory";
DirectoryInfo dir = new DirectoryInfo(dirPath);
...
bool exists = dir.Exists;
DirectoryInfo[] dirs = dir.GetDirectories();
FileInfo[] files = dir.GetFiles();
string fullName = dir.FullName;
```



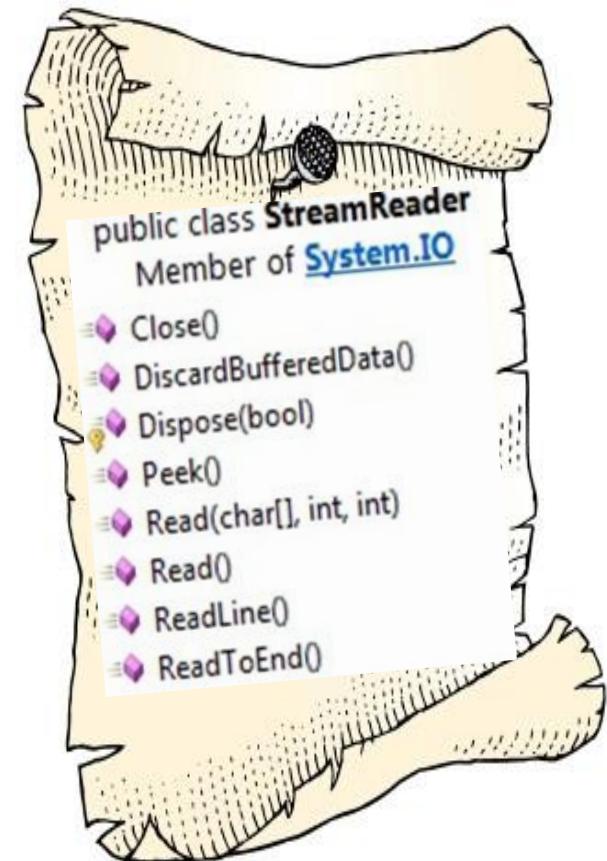
The StreamReader Class

- System.IO.StreamReader
 - The easiest way to read a text file
 - Implements methods for reading text lines and sequences of characters
 - Constructed by file name or other stream
 - Can specify the text encoding (Ex. Cyrillic use Windows-1251)
- Works like Console.Read() / ReadLine() but over text files.



StreamReader Methods

- new StreamReader(Filename)
 - Constructor for creating reader from a give file
- ReadLine()
 - Reads a single text line from the stream
 - Returns null when end-of-file is reached
- ReadToEnd()
 - Reads all the text until the end of the stream
- Close()
 - Closes the stream reader



Reading a Text File

- Reading a text file and printing its content to the console

```
StreamReader reader = new StreamReader("test.txt");
string fileContents = streamReader.ReadToEnd();
Console.WriteLine(fileContents);
streamReader.Close();
```

- Specifying the text encoding

```
StreamReader reader = new StreamReader(
    "cyr.txt", Encoding.GetEncoding("windows-1251"));
// Read the file contents here ...
reader.Close();
```

Using StreamReader - Practices

- The StreamReader instances should always be closed by calling the Close() method
 - Otherwise system resources can be lost
- In C# the preferable way to close streams and readers is by the “using” construction

```
using (<stream object>)
{
    // Use the stream here. It will be closed at the end
}
```

- It automatically calls the Close() after the using construction is completed

Reading a Text File - Example

- Read and display a text file line by line

```
StreamReader reader =  
    new StreamReader("somefile.txt");  
using (reader)  
{  
    int lineNumber = 0;  
    string line = reader.ReadLine();  
    while (line != null)  
    {  
        lineNumber++;  
        Console.WriteLine("Line {0}: {1}",  
            lineNumber, line);  
        line = reader.ReadLine();  
    }  
}
```

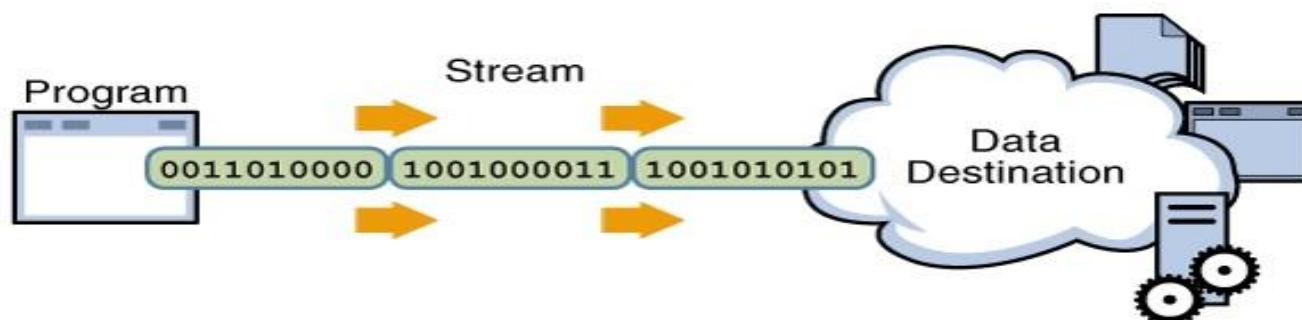
The StreamWriter Class

- System.IO.StreamWriter
 - Similar to StreamReader, but instead of reading, it provides writing functionality
- Constructed by file name or other stream

```
StreamWriter streamWriter = new StreamWriter("test.txt");
```

- Can define encoding (Ex. Cyrillic use “windows-1251”)

```
StreamWriter streamWriter = new StreamWriter("test.txt");
```



StreamWriter Methods

- Write()
 - Writes string or other object to the stream
 - Like Console.WriteLine()
- WriteLine()
 - Like Console.WriteLine()
- AutoFlush
 - Indicates whether to flush the internal buffer after each writing



Writing to a text file – Example

- Creates a text file named “numbers.txt” and print in it numbers from 1 to 20 (one per line)

```
StreamWriter streamWriter =  
    new StreamWriter("numbers.txt");  
using (streamWriter)  
{  
    for (int number = 1; number <= 20; number++)  
    {  
        streamWriter.WriteLine(number);  
    }  
}
```

Handling Exceptions when Opening a File

```
try
{
    StreamReader streamReader = new StreamReader(
        "c:\\\\NotExistingFileName.txt");
}
catch (System.NullReferenceException exc)
{
    Console.WriteLine(exc.Message);
}
catch (System.IO.FileNotFoundException exc)
{
    Console.WriteLine(
        "File {0} is not found!", exc.FileName);
}
catch
{
    Console.WriteLine("Fatal error occurred.");
}
```

Making a class Serializable

- To mark the class a serializable

```
[Serializable()]  
public class Loan { }
```

- To prevent a member from being serialized

```
[field: NonSerialized()]  
public float Pin;
```

- To add references to namespaces

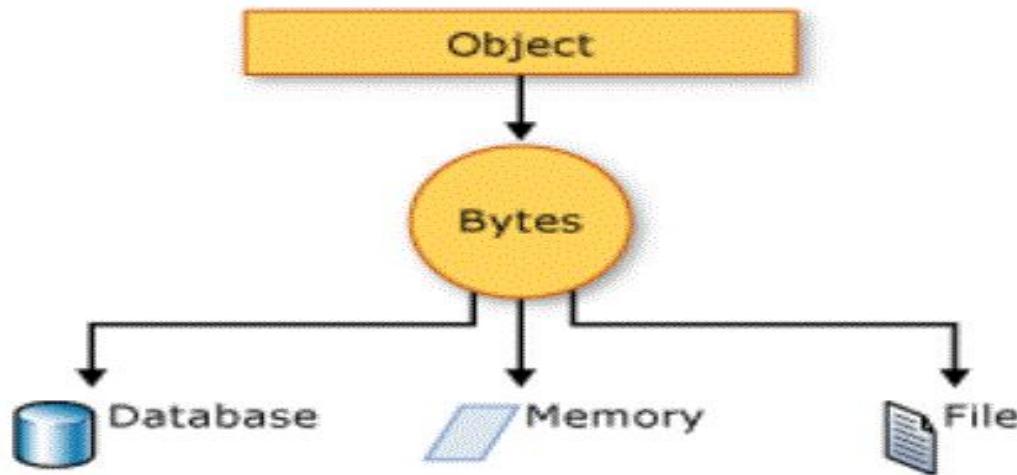
```
using System.IO;  
Using System.Runtime.Serialization.Formatters.Binary;
```

Serialization



Serialization

- The act of taking an in-memory object or object graph (set of objects that reference each other) and flattening it into a stream of bytes or XML nodes that can be stored or transmitted.



BinaryFormatter Class

- The BinaryFormatter class enables entire objects to be written to or read from a stream
 - Serialize() method – writes an object's representation to a file
 - Deserialize() method – reads this representation from a file and reconstructs the original object
 - Both methods throw a SerializationException if an error occurs during serialization or deserialization process

Serialization – Example (1 of 2)

```
public class MyObject
{
    public int n1 = 0;
    public int n2 = 0;
    public string str = null;
}

MyObject obj = new MyObject();
obj.n1 = 1;
obj.n2 = 24;
obj.str = "Some String";
```

SomeString

1
24

Class MyObject

+String str
+ int n1
+ int n2

1011001000110

Serialization – Example (2 of 2)

- **Serialization Example**

```
Iformatter formatter = new BinaryFormatter();

Stream stream = new FileStream("MyFile.bin", FileMode.Create,
FileAccess.Write, FileShare.None);

Formatter.Serialize(stream, obj);

stream.Close();
```

- **Deserialization Example**

```
Iformatter formatter = new BinaryFormatter();

Stream stream = new FileStream("MyFile.bin", FileMode.Open,
FileAccess.Read, FileShare.Read);

MyObject obj = (MyObject) formatter.Deserialize(stream);

stream.Close();

Console.WriteLine("{0}, {1}, {2}", obj.n1, obj.n2, obj.str);
```

XML Serialization

- Performed using `System.Xml.Serialization` class
- The `XmlSerializer` produces XML, but it can only include the values of an object's public fields and properties in the serialization data
- The values of private fields and properties are omitted
- Two Approaches
 - Sprinkle attributes throughout your types (Defined in `System.Xml.Serialization`)
 - Implement `IXmlSerializable` interface



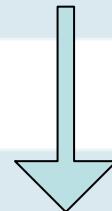
Attribute-Based Serialization (1 of 3)

```
Person p = new Person();
p.Name = "Stacey"; p.Age = 30;
XmlSerializer xs = new XmlSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
xs.Serialize (s, p);
Person p2;

using (Stream s = File.OpenRead ("person.xml"))
p2 = (Person) xs.Deserialize (s);
Console.WriteLine (p2.Name + " " + p2.Age); //Stacey 30
```

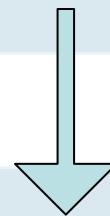
```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<Name>Stacey</Name>
<Age>30</Age>
</Person>
```



Attribute-Based Serialization (2 of 3)

```
public class Person
{
    [XmlElement ("FirstName")] public string Name;
    [XmlAttribute ("RoughAge")] public int Age;
}
```

```
<Person RoughAge="30" ...>
    <FirstName>Stacey</FirstName>
</Person>
```



Attribute-Based Serialization (3 of 3)

- Serializing base class

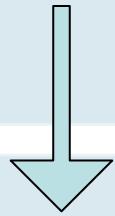
```
public void SerializePerson (Person p, string path){  
    XmlSerializer xs = new XmlSerializer (typeof (Person),  
                                         new Type[] { typeof (Student),  
                                         typeof (Teacher) } );  
    using (Stream s = File.Create (path)) xs.Serialize (s, p);  
}
```

- Serializing derived class

```
public class Person      {  
    public string Name;  
    public Address HomeAddress = new Address();  
}  
  
public class Address { public string Street, PostCode; }  
Person p = new Person(); p.Name = "Stacey";  
p.HomeAddress.Street = "Odo St";  
p.HomeAddress.PostCode = "6020";
```

Serializing Collections - Example

```
public class Person {  
    public string Name;  
    [XmlArray ("PreviousAddresses")]  
    [XmlArrayItem ("Location")]  
    public List<Address> Addresses = new List<Address>();  
}
```



```
<Person ... >  
  <Name>...</Name>  
  <PreviousAddresses>  
    <Location>  
      <Street>...</Street>  
      <Postcode>...</Postcode>  
    </Location>  
    <Location>  
      <Street>...</Street>  
      <Postcode>...</Postcode>  
    </Location>  
    . . .  
  </PreviousAddresses>  
</Person>
```

I~~X~~mlSerializable – Example

```
using System;
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;

public class Address : IXmlSerializable {
    public string Street, PostCode;
    public XmlSchema GetSchema() { return null; }

    public void ReadXml(XmlReader reader) {
        reader.ReadStartElement();
        Street = reader.ReadElementContentAsString ("Street", "");
        PostCode = reader.ReadElementContentAsString ("PostCode", "");
        reader.ReadEndElement();
    }

    public void WriteXml (XmlWriter writer) {
        writer.WriteElementString ("Street", Street);
        writer.WriteElementString ("PostCode", PostCode);
    }
}
```

Checkpoint

- Class StreamReader inherits from class Stream.
 - a. True
 - b. False
 - c. Information is not enough
- Which of the following is not a FileStream property?
 - a. CanRead
 - b. CanExist
 - c. CanSeek
 - d. CanWrite



Checkpoint

Checkpoint

- Which of the following statements is incorrect?
 - a. Only public fields, members, properties can be serialized & deserialized.
 - b. Can serialize object of type System.Int32[,,].
 - c. Serialization is the process of converting objects into streams of bytes.
 - d. To create a class that can be serialized, mark it with the [Serializable] attribute.



Checkpoint

Knowledge Check



Knowledge Check

1. You need to retrieve a list of subdirectories. Which class should you use?
 - a. FileInfo
 - b. DriveInfo
 - c. FileSystemWatcher
 - d. DirectoryInfo

Knowledge Check

2. You want to copy an existing file, File1.txt, to File2.txt. Which code samples do this correctly? (Choose two. Each answer forms a complete solution.)
- a.

```
File fi = new File();
    fi.Copy("file1.txt", "file2.txt");
```
 - b.

```
File.Copy("file1.txt", "file2.txt");
```
 - c.

```
FileInfo fi = new FileInfo("file1.txt");
    fi.CreateText();
    fi.CopyTo("file2.txt");
```
 - d.

```
FileInfo fi = new FileInfo("file1.txt");
    fi.CopyTo("file2.txt");
```

Knowledge Check

3. Which of the following attributes should you add to a class to enable it to be serialized?
 - a. ISerializable
 - b. Serializable
 - c. SoapInclude
 - d. OnDeserialization

4. Which of the following attributes should you add to a member to prevent it from being serialized by BinaryFormatter?
 - a. NonSerialized
 - b. Serializable
 - c. SerializationException
 - d. SoapIgnore

Knowledge Check

5. Which of the following interfaces should you implement to enable you to run a method after an instance of your class is deserialized?
 - a. IFormatter
 - b. ISerializable
 - c. IDeserializationCallback
 - d. IObjectReference
6. Which of the following are requirements for a class to be serialized with XML serialization? (Choose all that apply.)
 - a. The class must be public.
 - b. The class must be private.
 - c. The class must have a parameterless constructor.
 - d. The class must have a constructor that accepts a SerializationInfo parameter.

Knowledge Check

7. Which of the following attributes would you use to cause a member to be serialized as an XML attribute, rather than as an XML element?
- a. XmlAnyAttribute
 - b. XmlType
 - c. XmlElement
 - d. XmlAttribute

Module 12: Attributes



Module Objective

At the end of this module, you will be able to:

- Understand attributes in C#: its meaning, usage and types



Module objectives

Module Agenda

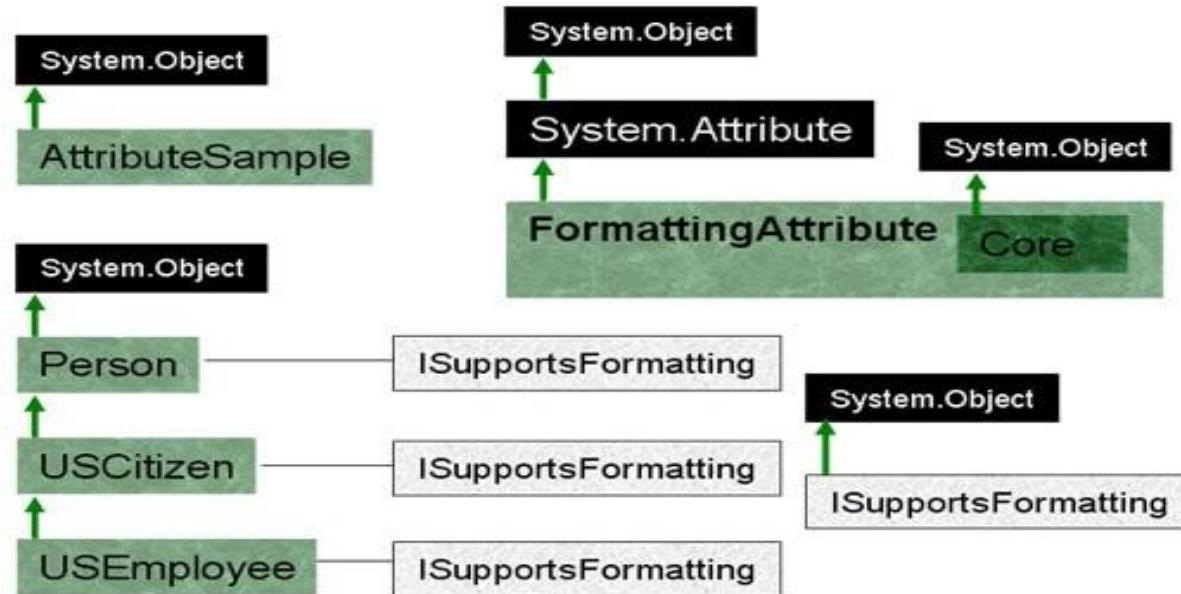


Attributes



What are Attributes?

- Declarative tags that convey information to the runtime
- Stored with the metadata of the element
- The .NET Framework provides predefined attributes
 - The runtime contains code to examine values of attributes and act on them
- Instances of classes derived from System.Attribute



Applying Attributes

- Use square brackets to specify an attribute

```
[attribute(positional_parameters, named_parameter=value, ...)]  
element
```

- To apply multiple attributes to an element, you can:
 - Specify multiple attributes in separate square brackets
 - Use a single square bracket and separate attributes with commas
 - For some elements such as assemblies, specify the element name associated with attribute explicitly

Common Predefined Attributes

- .NET provides many predefined attributes
 - General attributes
 - COM interoperability attributes
 - Transaction handling attributes
 - Visual designer component building attributes
- Restrictions on methods
 - Must have a return type
 - Must not be declared as override
 - Must not be from an inherited interface

ATTRIBUTE

Using the Conditional Attribute

- Serves as debugging tool
 - Causes conditional compilation of method calls, depending on the value of a programmer-defined symbol
 - Does not cause conditional compilation of the method itself
- Restrictions on methods
 - Must have a return type of void
 - Must not be declared as override
 - Must not be from an inherited interface

```
using System.Diagnostics;  
...  
class MyClass  
{  
    [Conditional ("DEBUGGING")]  
    public static void MyMethod( )  
    {  
        ...  
    }  
}
```

Using the DllImport Attribute

- With the `DllImportAttribute`, you can:
 - Invoke unmanaged code in DLLs from a C# environment
 - Tag external method to show that it resides in an unmanaged DLL

```
using System.Runtime.InteropServices;  
...  
public class MyClass( )  
{  
    [DllImport("MyDLL.dll", EntryPoint="MyFunction")]  
    public static extern int MyFunction(string param1);  
    ...  
    int result = MyFunction("Hello Unmanaged Code");  
    ...  
}
```

Using the Transaction Attribute

- To manage transactions in COM+
 - Specify that your component be included when a transaction is requested
 - Use a Transaction attribute on the class that implements the component

```
using System.EnterpriseServices;  
...  
[Transaction(TransactionOption.Required)]  
public class MyTransactionalComponent  
{  
    ...  
}
```

Defining Custom Attribute Scope

- Use the `AttributeUsage` tag to define scope

```
[AttributeUsage(AttributeTargets.Method)]  
public class MyAttribute: System.Attribute  
{ ... }
```

- Use the bitwise “or” operator (`|`) to specify multiple elements

```
[AttributeUsage(AttributeTargets.Class |  
AttributeTargets.Struct)]  
public class MyAttribute: System.Attribute  
{ ... }
```

Defining an Attribute Class

- Deriving an attribute class
 - All attribute classes must derive from the System.Attribute, directly or indirectly
 - Suffix name of attribute class with “Attribute”
- Components of an attribute class
 - Define a single constructor for each attribute class by using a propositional parameter
 - Use properties to set an optional value by using a named parameter

Preprocessing a Custom Attribute

1. Searches for the attribute class
2. Checks the scope of the attribute
3. Checks for a constructor in the attribute
4. Creates an instance of the object
5. Checks for a named parameter
6. Sets field or property to named parameter value
7. Saves current state of attribute class

Using Multiple Attributes

- An element can have more than one attribute
 - Define both attributes separately
- An element can have more than one instance of the same attribute
 - Use AllowMultiple = true

ATTRIBUTE
ATTRIBUTE
ATTRIBUTE

Examining Class Metadata

- To query class metadata information:
 - Use the MemberInfo class in System.Reflection
 - Populate a MemberInfo object by using System.Type
 - Create a System.Type object by using the typeof operator
- Example

```
System.Reflection.MemberInfo typeInfo;  
typeInfo = typeof(MyClass);
```

Querying for Attribute Information

- To retrieve attribute information:
 - Use **GetCustomAttributes** to retrieve all attribute information as an array

```
System.Reflection.MemberInfo typeInfo;  
typeInfo = typeof(MyClass);  
object[ ] attrs = typeInfo.GetCustomAttributes(false);
```

- Iterate through the array and examine the values of each element in the array
- Use the **IsDefined** method to determine whether a particular attribute has been defined for a class

Custom Attributes

- It allows anyone to define information that can be applied to almost any metadata table entry.
- This extensible metadata information can be queried at run time to dynamically alter the way code executes.
- Defining own attribute steps:
 - Define an attribute and make it derived from System.Attribute to identify attribute definition in metadata fast and easy.
 - It must have a public accessibility.
 - Apply the attribute to a property or a method. If property, it must be read-only.

Custom Attribute – Example (1 of 2)

```
[System.Attribute(System.AttributeTargets.Class |  
    System.AttributeTargets.Struct,  
    AllowMultiple = true)  
]  
public class Author : System.Attribute  
{  
    private string name;  
    public double version;  
  
    public Author(string name)  
    {  
        this.name = name;  
        version = 1.0;  
    }  
}
```

Custom Attribute – Example (2 of 2)

```
[Author("J. Doe", version = 1.1)]  
[Author("B. Gates", version = 1.2)]  
Class SampleClass  
{  
    // J. Doe's code goes here  
    // B. Gates' code goes here  
}
```

Checkpoint

- Which statements are correct about Attributes in C#.NET?
 1. On compiling a C#.NET program the attributes applied are recorded in the metadata of the assembly.
 2. On compilation all the attribute's tags are deleted from the program.
 3. The attributes applied can be read from an assembly using Reflection class.
 4. An attribute can have parameters.
 - b. 1 and 2 only
 - c. 2 and 3 only
 - d. 1, 3 and 4 only
 - e. All of the above



Checkpoint

Knowledge Check



Knowledge Check

1. Which of the following are correct ways to specify the targets for a custom attribute?
 - a. By applying *AttributeUsage* to the custom attribute's class definition.
 - b. By applying *UsageAttribute* to the custom attribute's class definition.
 - c. Once an attribute is declared it applies to all the targets.
 - d. By applying *AttributeUsageAttribute* to the custom attribute's class definition.
 - e. None of the above.

Knowledge Check

2. Which of the following correctly describes the contents of the filename AssemblyInfo.cs?
 - a. It contains method-level attributes.
 - b. It contains class-level attributes.
 - c. It contains assembly-level attributes.
 - d. It contains structure-level attributes.
 - e. It contains namespace-level attributes.

Knowledge Check

3. It possible to create a custom attribute that can be applied only to specific programming element(s) like ____.
 - a. Classes
 - b. Methods
 - c. Classes and Methods
 - d. Class, Methods and Member variables
4. Which of the following CANNOT be a target for a custom attribute?
 - a. Enum
 - b. Event
 - c. Delegate
 - d. Interface
 - e. Namespace

Knowledge Check

5. Which of the following is the correct way of applying the custom attribute called Tested which receives two-arguments - name of the tester and the testgrade?
1. Custom attribute cannot be applied to an assembly.
 2. `[assembly: Tested("Sachine", testgrade.Good)]`
 3. `[Tested("ViRus", testgrade.Excellent)]`
class customer { / */ }*
 4. Custom attribute cannot be applied to a method.
 5. Custom attribute cannot be applied to a class.
- b. 1 only
- c. 1, 5
- d. 2, 3
- e. 4, 5
- f. None of the above