Programing in C#
Object-Oriented Concepts

EEE-425
Programming Languages

Key Object-Oriented Concepts

- Objects, instances and classes
- Identity
  - Every instance has a unique identity, regardless of its data
- Encapsulation
  - Data and function are packaged together
  - Information hiding
  - An object is an abstraction
    - User should NOT know implementation details

Key Object-Oriented Concepts

- Interfaces
  - A well-defined contract
  - A set of function members
- Types
  - An object has a type, which specifies its interfaces and their implementations
- Inheritance
  - Types are arranged in a hierarchy
  - Base/derived, superclass/subclass
  - Interface vs. implementation inheritance

Key Object-Oriented Concepts

- Polymorphism
  - The ability to use an object without knowing its precise type
  - Three main kinds of polymorphism
    - Inheritance
      - Interfaces
      - Reflection
  - Dependencies
    - For reuse and to facilitate development, systems should be loosely coupled
    - Dependencies should be minimized

Programming in C#
Interfaces

- Interfaces
  - An interface defines a contract
    - An interface is a type
    - Contain definitions for methods, properties, indexers, and/or events
    - Any class or struct implementing an interface must support all parts of the contract
  - Interfaces provide no implementation
    - When a class or struct implements an interface it must provide the implementations
Interfaces

- Interfaces provide polymorphism
- Many classes and structs may implement a particular interface.
- Hence, can use an instance of any one of these to satisfy a contract.
- Interfaces may be implemented either:
  - Implicitly – contain methods with the same signature. The most common approach.
  - Explicitly – contain methods that are explicitly labeled to handle the contract.

Declaring Interfaces

- Syntax: Use the interface keyword to declare methods
  ```
  interface IToken
  {
    int LineNumber();
    string Name();
  }
  ```

Interfaces Example

```csharp
public interface IDelete
{
    void Delete();
}

public class TextBox : IDelete
{
    public void Delete() { ... }
}

public class Car : IDelete
{
    public void Delete() { ... }
}
```

TextBox tb = new TextBox();
tb.Delete();
Car c = new Car();
iDel = c;
iDel.Delete();

Explicit Interfaces

- Explicit interfaces require the user of the class to explicitly indicate that it wants to use the contract.
- Note: Most books seem to describe this as a namespace conflict solution problem. If that is the problem you have an extremely poor software design. The differences and when you want to use them are more subtle.

Explicit Interfaces

```csharp
namespace OhioState.CSE494R.InterfaceTest
{
    public interface IDelete
    {
        void Delete();
    }

    public class TextBox : IDelete
    {
        #region IDelete Members
        void IDelete.Delete(); // compile error
        #endregion
        void Delete();
    }
}
```

The ReadOnlyCollection<T> class is a good example of using an explicit interface implementation to hide the methods of the IList<T> interface that allow modifications to the collection.
- Calling Add() will result in a compiler error if the type is ReadOnlyCollection.
- Calling IList.Add() will throw a run-time exception 😞.
Interfaces Multiple Inheritance

- Classes and structs can inherit from multiple interfaces
- Interfaces can inherit from multiple interfaces

```csharp
interface IControl {
    void Paint();
}
interface IListBox: IControl {
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {
}
```

Implementing Multiple Interfaces

- A class can implement zero or more interfaces
  ```csharp
  interface IToken {
      string Name();
  }
  interface IVisitorable {
      void Accept(IVisitor v);
  }
  class Token: IToken, IVisitorable {
      ...
  }
  ```

Implementing Interface Methods

- The implementing method must be the same as the interface method
- The implementing method can be virtual
  ```csharp
class Token: IToken, IVisitorable {
    public virtual string Name() {
    }
    public void Accept(IVisitor v) {
    }
}
```

Programming in C# Structs

Classes vs. Structs

- Both are user-defined types
- Both can implement multiple interfaces
- Both can contain
  - Data
    - Fields, constants, events, arrays
  - Functions
    - Methods, properties, indexers, operators, constructors
  - Type definitions
    - Classes, structs, enums, interfaces, delegates

<table>
<thead>
<tr>
<th>Class</th>
<th>Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference type</td>
<td>Value type</td>
</tr>
<tr>
<td>Can inherit from any non-sealed reference type</td>
<td>No inheritance (inherits only from System.ValueType)</td>
</tr>
<tr>
<td>Can have a destructor</td>
<td>No destructor</td>
</tr>
<tr>
<td>Can have user-defined parameterless constructor</td>
<td>No user-defined parameterless constructor</td>
</tr>
</tbody>
</table>
### C# Structs vs. C++ Structs

- Very different from C++ struct

<table>
<thead>
<tr>
<th>C++ Struct</th>
<th>C# Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same as C++ class, but all members are public</td>
<td>User-defined value type</td>
</tr>
<tr>
<td>Can be allocated on the heap, on the stack or as a member (can be used as value or reference)</td>
<td>Always allocated on the stack or in-lined as a member field</td>
</tr>
<tr>
<td>Members are always public</td>
<td>Members can be public, internal or private</td>
</tr>
</tbody>
</table>

### Class Definition

```csharp
public class Car : Vehicle {
    public enum Make { GM, Honda, BMW }
    private Make make;
    private string vid;
    private Point location;
    public Car(Make make, string vid, Point loc) {
        this.make = make;
        this.vid = vid;
        this.location = loc;
    }
    public void Drive() {
        Console.WriteLine("vroom");
    }
}

Car c = new Car(Car.Make.BMW, "JF3559QT98", new Point(3,7));
c.Drive();
```

### Programming in C#

#### Modifiers

- **Static vs. Instance Members**
  - By default, members are per instance
  - Each instance gets its own fields
  - Methods apply to a specific instance
  - Static members are per type
  - Static methods can't access instance data
  - No `this` variable in static methods

```csharp
public struct Point {  
    private int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int X { get { return x; }  
    set { x = value; }   
    }
    public int Y { get { return y; }   
    set { y = value; }   
    }
}

Point p = new Point(2,5);
p.X += 100;
int px = p.X;  // px = 102
```
Access Modifiers

- Access modifiers specify who can use a type or a member
- Access modifiers control encapsulation
- Class members can be public, private, protected, internal, or protected internal
- Struct members can be public, private or internal

<table>
<thead>
<tr>
<th>Access Default</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>to everyone</td>
</tr>
<tr>
<td>private</td>
<td>within T only</td>
</tr>
<tr>
<td>protected</td>
<td>to T or types derived from T</td>
</tr>
<tr>
<td>internal</td>
<td>to types within A</td>
</tr>
<tr>
<td>protected internal</td>
<td>to T or types derived from T</td>
</tr>
</tbody>
</table>

Access Defaults

- You should always explicitly mark what access you want.
- Class definitions default to internal.
- Member fields, methods and events default to private for classes
- Member methods and events for interfaces must be public, so you can not specify an access modifier for interfaces.

Abstract Classes

- An abstract class can not be instantiated
- Intended to be used as a base class
- May contain abstract and non-abstract function members
- A pure abstract class has no implementation (only abstract members) and is similar to an interface.

Sealed Classes

- A sealed class is one that cannot be used as a base class.
- Sealed classes can not be abstract
- All structs are implicitly sealed
- Prevents unintended derivation
- Allows for code optimization
  - Virtual function calls may be able to be resolved at compile-time

Programming in C#

Class Internals
The `this` keyword is a predefined variable available in non-static function members.

- Used to access data and function members unambiguously.

```csharp
public class Person {
    private string name;
    public Person(string name) {
        this.name = name;
    }
    public void Introduce(Person p) {
        if (p == this)
            Console.WriteLine("Hi, I'm " + name);
    }
}
```

The `base` keyword can be used to access class members that are hidden by similarly named members of the current class.

```csharp
public class Shape {
    private int x, y;
    public override string ToString() {
        return "x=" + x + ",y=" + y;
    }
}
```

### Constants

- A constant is a data member that is evaluated at compile-time and is implicitly static (per type).
- e.g. `Math.PI`

```csharp
public class MyClass {
    public const string version = "1.0.0";
    public const string s1 = "abc" + "def";
    public const int i3 = 1 + 2;
    public const double PI_I3 = i3 * Math.PI;
    public const double s = Math.Sin(Math.PI);  //ERROR
}
```

### Fields

- A field or member variable holds data for a class or struct.
- Can hold:
  - A built-in value type
  - A class instance (a reference)
  - A struct instance (actual data)
  - An array of class or struct instances (an array is actually a reference)
  - An event

### Readonly Fields

- Similar to a const, but is initialized at run-time in its declaration or in a constructor.
- Once initialized, it cannot be modified.
- Differs from a constant:
  - Initialized at run-time (vs. compile-time)
  - Don’t have to re-compile clients
  - Can be static or per-instance

```csharp
public class MyClass {
    public static readonly double d1 = Math.Sin(Math.PI);
    public readonly string s1;
    public MyClass(string s) { s1 = s; }
}
```

### Methods

- All code executes in a method.
- Constructors, destructors and operators are special types of methods.
- Properties and indexers are implemented with get/set methods.
- Methods have argument lists.
- Methods contain statements.
- Methods can return a value.
Virtual Methods

- Methods may be virtual or non-virtual (default)
- Non-virtual methods are not polymorphic
- Abstract methods are implicitly virtual.

```csharp
internal class Foo {
    public void DoSomething(int i) {
        ...
    }
}
```

```
Foo f = new Foo();
f.DoSomething(6);
```

Virtual Methods

```csharp
public class Shape {
    public virtual void Draw() { ... }
}
internal class Box : Shape {
    public override void Draw() { ... }
}
internal class Sphere : Shape {
    public override void Draw() { ... }
}
```

```
protected void HandleShape(Shape s) {
    s.Draw();
    ...
    Handleshape(new Box());
    Handleshape(new Sphere());
    Handleshape(new Shape());
}
```

Abstract Methods

- An abstract method is virtual and has no implementation
- Must belong to an abstract class
- Used as placeholders or handles where specific behaviors can be defined.
- Supports the Template design pattern.

```csharp
public abstract class Shape {
    public abstract void Draw();
}
```

```
internal class Box : Shape {
    public override void Draw() { ... }
}
internal class Sphere : Shape {
    public override void Draw() { ... }
}
```

```
private void HandleShape(Shape s) {
    s.Draw();
    ...
    Handleshape(new Box());
    Handleshape(new Sphere());
    Handleshape(new Shape()); // Error!
}
```

Abstract Methods

Program in C#

Constructors

```
Method Versioning

- Must explicitly use override or new keywords to specify versioning intent
- Avoids accidental overriding
- Methods are non-virtual by default
- C++ and Java produce fragile base classes – cannot specify versioning intent
Constructors

- Instance constructors are special methods that are called when a class or struct is instantiated.
- Performs custom initialization.
- Can be overloaded.
- If a class doesn’t define any constructors, an implicit parameterless constructor is created.
- Cannot create a parameterless constructor for a struct.
- All fields initialized to zero/null.

Constructor Initializers

- One constructor can call another with a constructor initializer.
- Use the this keyword. The called constructor will execute before the body of the current constructor.

```csharp
internal class B {  
    private int h;
    public B() : this(12) { }
    public B(int h) { this.h = h; }
}
```

Constructor Initializers

- The base keyword is also used to control the constructors in a class hierarchy:

```csharp
public class Volunteer : Employee
{
    public Volunteer(string name) : base(name) { }
}
```

Static Constructors

- A static constructor lets you create initialization code that is called once for the class.
- Guaranteed to be executed before the first instance of a class or struct is created and before any static member of the class or struct is accessed.
- No other guarantees on execution order.
- Only one static constructor per type.
- Must be parameterless.

Singleton Design Pattern

```csharp
public class SoundManager
{
    private static SoundManager instance;
    public static SoundManager Instance { get { return instance; } }
    private static SoundManager() { instance = new SoundManager(); }
    private SoundManager() { }
}
```
Destructors

- A destructor is a method that is called before an instance is garbage collected
- Used to clean up any resources held by the instance, do bookkeeping, etc.
- Only classes, not structs can have destructors
- Also called Finalizers.

```csharp
internal class Foo {
    private ~Foo() {
        Console.WriteLine("Destroyed {0}", this);
    }
}
```

Destructors

- Unlike C++, C# destructors are non-deterministic
- They are not guaranteed to be called at a specific time
- They are guaranteed to be called before shutdown
- You can not directly call the destructor
- Slows down the garbage collection if you define one, so don't unless you have to.

Dispose Design Pattern

- Use the using statement and the IDisposable interface to achieve deterministic clean-up of unmanaged resources.
- The destructor optionally calls a public Dispose method, that is also user-callable.

Operator Overloading

- User-defined operators
- Must be a static method

```csharp
internal class Car {
    private string vid;
    public static bool operator ==(Car x, Car y) {
        return x.vid == y.vid;
    }
}
```
### Operator Overloading

#### Overloadable unary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>!</code></td>
<td>Logical NOT</td>
</tr>
<tr>
<td><code>~</code></td>
<td>Bitwise NOT</td>
</tr>
<tr>
<td><code>true</code></td>
<td>True</td>
</tr>
<tr>
<td><code>false</code></td>
<td>False</td>
</tr>
<tr>
<td><code>++</code></td>
<td>Increment</td>
</tr>
<tr>
<td><code>--</code></td>
<td>Decrement</td>
</tr>
</tbody>
</table>

#### Overloadable binary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>!</code></td>
<td>Logical NOT</td>
</tr>
<tr>
<td><code>~</code></td>
<td>Bitwise NOT</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulus</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Equality</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>Left shift</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Right shift</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal</td>
</tr>
</tbody>
</table>

### Conversion Operators

#### Can also specify user-defined explicit and implicit conversions

```csharp
internal class Note {
    private int value;
    // Convert to hertz - no loss of precision
    public static implicit operator double(Note x) {
        return ...;
    }
    // Convert to nearest note
    public static explicit operator Note(double x) {
        return ...;
    }
}
```

```csharp
Note n = (Note)442.578;
double d = n;
```

### The is Operator

- The `is` operator is used to dynamically test if the run-time type of an object is compatible with a given type

```csharp
private static void DoSomething(object o) {
    if (o is Car) {
        ((Car)o).Drive();
    }
}
```

### The as Operator

- The `as` operator tries to convert a variable to a specified type; if no such conversion is possible the result is null
- More efficient than using `is` operator
- Can test and convert in one operation

```csharp
private static void DoSomething(object o) {
    Car c = o as Car;
    if (c != null) c.Drive();
}
Type Unification

- Single universal base type ("object")
  - Object variable can hold any value
  - Any piece of data can be stored, transported, and manipulated with no extra work
- Unification enables:
  - Calling virtual functions on any value
  - Collection classes for any type

Let's look at this feature!

Other Unification Approaches

- Java and Eiffel
  - Intrinsic types are not classes
    - Good performance
    - Can’t convert “int” to “Object” – the primitive types are in a separate world
    - Requires special wrapper classes (e.g., “Integer”) to “wrap” a primitive type so that it works in the “Object” world.

Unification in SmallTalk

- Make everything a real object
  - Performance implications
    - All objects have a type descriptor or virtual function table
    - May require all object to be heap-allocated to prevent dangle pointers
  - Behavior and expectation mismatch
    - “int” variables can be “null”

C# Type Unification

- Types are divided into two kinds: Reference types and Value types
- Reference types are full-featured:
  - Always allocated in heap
  - Arbitrary derivation
- Value types have restrictions:
  - Only inherit from object
  - Can’t be used as base classes
  - Allocated from stack or inline inside other objects
  - Assignment copies value, not reference

Unification Implementation

- Value types don’t need type descriptors or vtables (efficient!)
- “object” does need a type descriptor, because it can contain any type
- Value types become reference types when they are converted to “object”
  - Value is copied to heap and a type descriptor is attached ("boxing")
  - When cast back to value type, the value is copied out of heap ("unboxing")

Desired Picture:

- Object
- Stream
- HasTable
- int
- double
- MemoryStream
- FileStream

How to deal with the primitive types without losing performance?
How to create user-defined types that are as efficient as “int” or “double”?
A Unified Collection Class

- One of the great benefits of this is the ability to create generic collections.
- C# 1.1 had/has several of these:
  - ArrayList – a dynamically allocated list.
  - Queue – a FIFO container.
  - Stack – a LIFO container.
  - HashTable – a dictionary or partial map.

Issues with ArrayList

- How do I get a value out:
  ```csharp
  object myInt = myList[2];
  ```

- How do I get the correct type – cast:
  ```csharp
  int myInt = (int)myList[2];
  ```

- Ooops. That should have been index 1!

- Very unsafe programming practice
  - Equivalent to using void* in C or C++
  - Does have its use though.

Issues with ArrayList

- Memory costs:
  - A boxed int may cost about 4 times the memory.
  - ArrayList is hence very inefficient for value types (including structs).

- Solution: Generics!

ArrayList

- Sample ArrayList code:
  ```csharp
  ArrayList myList = new ArrayList();
  myList.Add("DotNetSpider"); // Add a string.
  myList.Add(1032); // Add an integer
  myList.Add(DateTime.Now); // Add current time.
  myList.Add(new DataTable()); // Add a datatable
  ```

Issues with ArrayList

- What if I didn’t want a heterogenous collection of objects, but a nice, say queue, of integers?
  ```csharp
  Queue myQueue = new Queue();
  myQueue.Put(4);
  int myInt = (int)myQueue.Get();
  ```
  - Code littered with casting 😞
  - What actually happens – boxing! 😞

Programming in C#

Generics
Motivation

- See the Type Unification and the use of the ArrayList set of slides.
- In summary, four main goals:
  1. Increase type safety (statically)
  2. Eliminate type casts
  3. Eliminate boxing and unboxing
  4. C++ has templates
- Syntax is similar to C++

Terminology

- Why the name Generic?
  - We separate the behavior from the type allowing more generic behavior descriptions.
  - Also called Parametric Polymorphism
  - We supply a type parameter and the same code or behavior applies to this type.

Generic Syntax

- Write public class Stack<T> {...}
- T is the type parameter:
  - Stack<int> myStack
  - Dictionary<TKey, TValue>
- Can have several type parameters:
  - Dictionary<TKey, TValue>
- Compiler will now enforce type safety:
  - myStack.Push(4.3) // Compiler error

Design Note

It is customary to use T for a generic single type. For multiple types or in cases where the type is clear a more specific name should be used. This is pre-fixed by a capital T.

Generic Parameterization

- Generics can be used with:
  - Types
    - Struct
    - Interface
    - Class
    - Delegate
  - Methods

Using Generics - Types

- Can be used to easily create non-generic derived types:
  
  ```csharp
  public class IntStack : Stack<int>
  {
  }
  ```

- Can be used in internal fields, properties and methods of a:
  
  ```csharp
  public struct Customer<T>
  {
    private static List<T> customerList;
    private T customerInfo;
    public T CustomerInfo { get; set; }
    public int CompareCustomers(T customerInfo);
  }
  ```
Using Generics - Types

- Using the type is like using any other non-generic type.
- The type parameter only needs to be specified during instantiation.

Customer<int> fred = new Customer<int>();
fred.CustomerInfo = 4;

Verifying Generic Types

- In C#, generic types can be compiled into a class library or dll and used by many applications.
- Differs from C++ templates, which use the source code to create a new type at compile time.
- Hence, when compiling a generic type, the compiler needs to ensure that the code will work for any type.

Generic Constraints

- What if we have a class constraint?

public class CarFactory<T> where T : class
{
    private T currentCar = null;
}

- Why would the compiler produce an error for this?

Generic Constraints

- Can also specify a class constraint.
- That is, require a reference type:

public class CarFactory<T> where T : class
{
    private T currentCar = null;
}

- Forbids CarFactory<int> and other value types.
- Useful since I can not set an int to null.

Generic Constraints

- Alternatively, require a value (struct) type:

public struct Nullable<T> where T : struct
{
    private T value;
}

- Fixes the new problem (but is limited):

public class Stack<T> where T : struct
{
    public T PopEmpty() {
        return new T();
    }
}
3.11.2013

Using a Default Value

- You may need to initialize a variable

```java
public class GraphNode<T> {
    private T nodeLabel;
    private void ClearLabel() {
        nodeLabel = null;
    }
}
```

- Why doesn’t this work?

```java
public class GraphNode<T> {
    private T nodeLabel;
    private void ClearLabel() {
        nodeLabel = default(T);
    }
}
```

- If T is a reference type default(T) will be null.
- For value types all bits are set to zero.

Constructor Constraint

- Special constraint using the `new` keyword:

```java
public class Stack<T> where T : new() {
    public T PopEmpty() {
        return new T();
    }
}
```

- Parameter-less constructor constraint
  - Type T must provide a public parameter-less constructor
  - No support for other constructors or other method syntaxes.
  - The `new()` constraint must be the last constraint.

Primary Constraints

- A generic type parameter, like a regular type, can have zero or one primary constraints, including:
  - Derived from a non-sealed concrete or abstract base type
  - The class constraint
  - The struct constraint

Secondary Constraints

- A generic type parameter, like a regular type, can have zero or more interface constraints

```java
public class GraphNode<T> {
    where T : ICloneable, IComparable ...
}
```

The where clause

- A type parameter can only have one where clause, so all constraints must be specified within a single where clause.

- Not allowed:

```java
public class GraphNode<T> {
    where T : ICloneable, IComparable new()

    ...
}
```
Multiple Type Parameters

- A generic type can be parameterized with many type place-holders;
  ```java
  public interface IFunction<TDomain, TRange> {
    TRange Evaluate(TDomain sample);
  }
  ```
- 2D, 3D, complex function support with mappings from one domain to another.

Dependent Constraints

- Each type parameter can have its own set of constraints (and own where class).
- You can also have one type parameter be dependent on another.
  ```java
  public class SubSet<U, V> where U : V
  public class Group<U, V> where V : IEnumerable<U> { ... }
  ```

Compilation Errors

- class A {...}
- class B {...}
- class Incompat<S, T>
  ```java
  where S: A, T
  where T: B
  {
    ...
  }
  ```

Compilation Errors

- class StructWithClass<S, T, U>
  ```java
  where S: struct, T
  where T: U
  where U: A
  {
    ...
  }
  ```

Compilation Errors

```java
interface I<T>
{
  void F();
}
class X<U, V>: I<U>, I<V>
{
  void I<U>.F() { ... }
  void I<V>.F() { ... }
}
```

Generic Methods

- C# also allow you to parameterize a method with generic types:
  ```java
  public static void Swap<T>( ref T a, ref T b )
  {
    T temp = a;
    a = b;
    b = temp;
  }
  ```
Generic Methods

- The method does not need to be static.
  ```csharp
  public class Report<T> : where T : IFormatter {
  }
  public class Insurance {
    public Report<T> ProduceReport<T>() where T : IFormatter {
      ...
    }
  }
  ```

Type Covariance

- We say a type Derived is Covariant to the type, Base, if Derived can be cast to Base.
- Generic types are not covariant.
  ```csharp
  MyClass<Derived> md;
  MyClass<Base> mb = md;
  ```

Java Generics v. C#

- Java made the decision to keep backward compatible bytecode.
  - Hence old JVM’s can run the new Java with generics code.
  - Ruins run-time type reflection.
- C# 2.0 requires a new CLR.
  - Generics are supported in the IL code.

C++ Templates v. C#

- C++ has slightly more powerful parametric polymorphism in that non-type parameters can also be used.
- No run-time type support or reflection.
- Run-time (generics) versus compile-time (templates)
- Requires you to expose your source code to everyone.

Assignment

- In addition to Reading Chapters 1-3 of the textbook and going through these lectures, you should:
  - Memorize the C# keywords a-l in the appendix of the book.
  - Think of how you would design a program or set of programs to display memorization questions or flashcards.
  - Read careful through the errata for book for Chapters 1-4.

References

- Prof. Roger Crawfis
  - [http://www.cse.ohio-state.edu/~crawfis/cse459_CSharp/index.html](http://www.cse.ohio-state.edu/~crawfis/cse459_CSharp/index.html)
- Pls listen the podcast about the chapter 3
- These slides are changed and modified