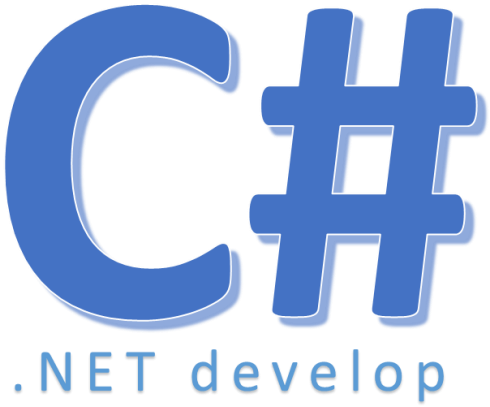
Fachhochschule Nordwestschweiz
Hochschule für Technik

C# ESSENTIALS



Martin Kropp / Yves Senn
University of Applied Sciences Northwestern Switzerland

Learning Targets

- You
  - know the .NET type system
  - can explain essential C# concepts and their differences to Java
  - can apply basic C# constructs like Properties, Indexers for writing a simple C# program
  - can use C# Streams for efficient data handling
  - Understand the inheritance concepts of overriding and new

Content

- C# feature overview
- C# type system
  - Data types, type unification, boxing & unboxing
- Working with classes and collections
  - Classes & interfaces
  - Properties, indexers, ...
  - Collections
- Inheritance & method overloading
  - Polymorphism and hiding
- Generics
- File I/O

C# feature comparison

Also in Java

- Exceptions
- Threading
- Strong typing
- Garbage collection
- Reflection
- Dynamic loading of code
- Enumerations
- Generics (Type erasure)
- Attributes
- Anonymous Types
- foreach
- Lambdas (finally)
- LINQ/Streams API (finally)
- var (finally)

Also in C++

- Structs
- Operator overloading
- Pointer arithmetic in unsafe code
- Some syntactic details
- Generics (Templates)

C# feature differences

Differences

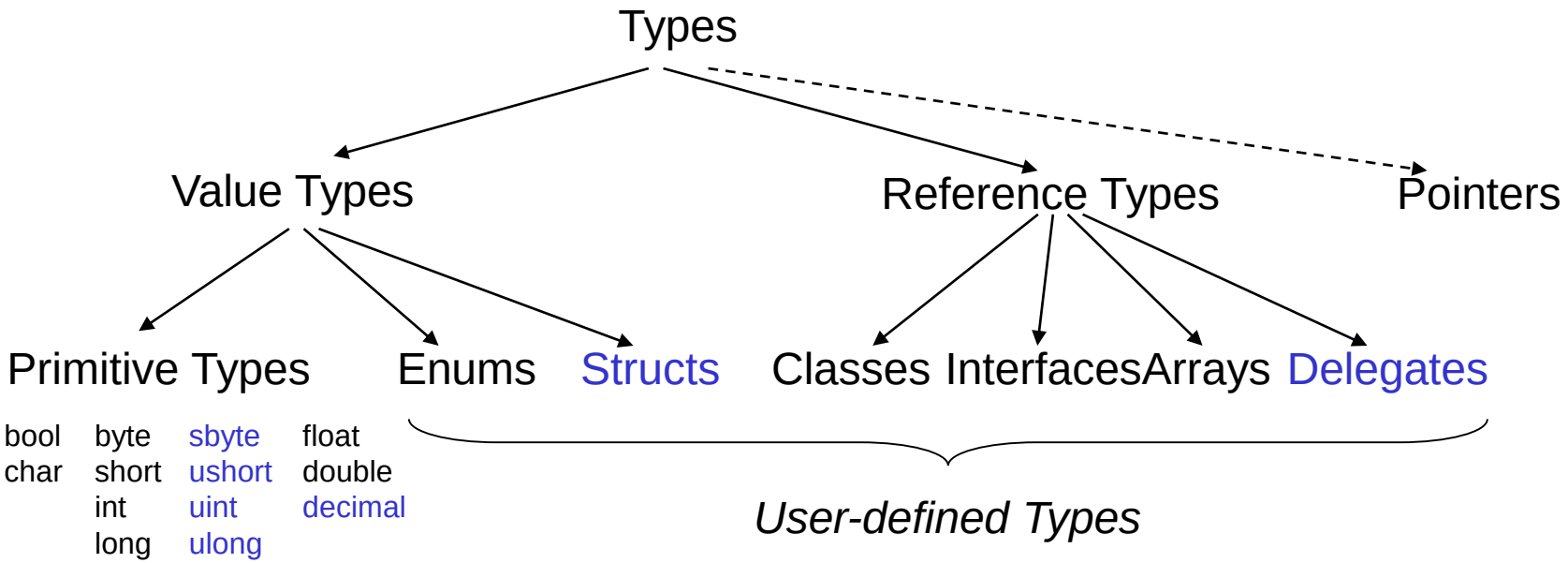
- Call-by-reference parameters
- Value types (Structs)
- Block matrices
- Unified type system
- `goto` statement
- Versioning
- Native interoperability (vs. JNI)
- Dynamic typing
- Nullable types

"Syntactic Sugar"

- Properties
- Events
- Delegates
- Indexers
- Boxing/unboxing
- Static classes
- Partial classes
- null conditional operator
- interpolated strings
- ...

more to see on http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java

Unified type system



blue types are missing from Java

Boxing & unboxing

- `object` is „The mother of all types“
- Value types are compatible with reference type object
  ➔ Type unification

- Boxing
  - converting a value type into a reference type
  - wraps up the value of i1 from the stack in a heap object

```
int i1 = 3;
object obj = i1;
```

- Unboxing
  - converting a reference type into a value type
  - unwraps the value again

```
int i2 = (int) obj;
```

Classes

- Declaration

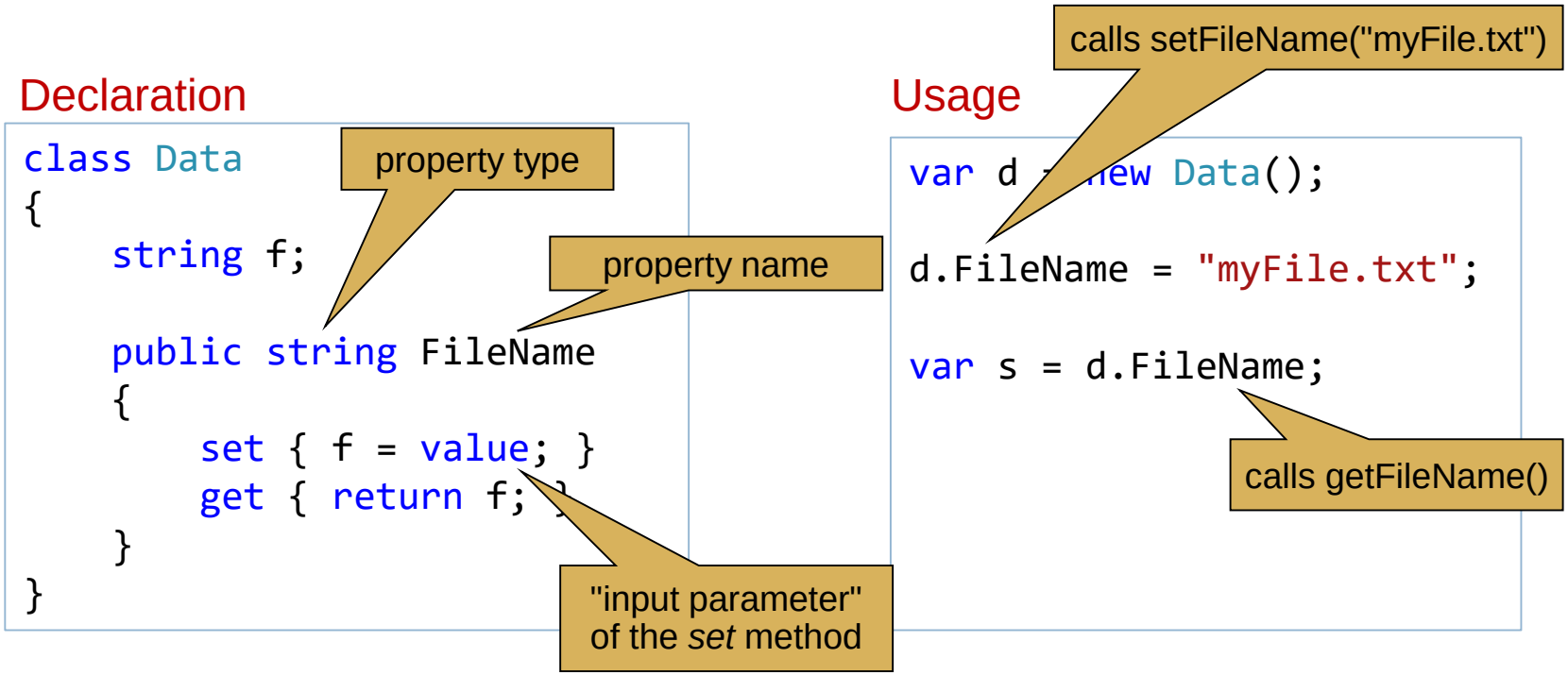
```
class Hello
{
    private string name;
    private void Greet() {...}
    public static void Main(string[] args) {...}

    static Hello() { /* static constructor */ }
}
```

- Static constructor
  - Executed once per **type** before any instances of the type are created and any other static members are accessed
  - **Exact timing (“before”) is undefined**

- Static fields
  Initialized before the static constructor is called

Properties

Properties provide safe and convenient field access



Automatic properties

```
class Data
{
    public string CreateDate { get; set; } = DateTime.Now;
}
```

- Compiler generates a private field internally
- get and set can have different modifiers
- get and set can contain arbitrary code (calls to methods, for example)

Automatic properties used

```
class Data
{
    public string FileName { get; set; }

    public string FilePath { get; private set; }

    public DateTime FileCreateDate { get; } = DateTime.Now;

    public DateTime FileChangeDate { get; set; } = DateTime.Now;

}
```

String interpolation

You want to output this string: `"Car is 5.73m wide and is not red"`

```
s.Name + " is " + s.Width.ToString("F2") + "m wide and is "+(s.IsRed ?
"red":"not red")
```

Slightly more readable (error prone!):

```
string.Format("{0} is {1:F2}m wide and is {2}", s.Name, s.Width, s.IsRed ?
"red":"not red")
```

With string interpolation:

```
$"{s.Name} is {s.Width:F2}m wide and is {(s.IsRed ? "red":"not red")}"
```

More String interpolation

String interpolation:

```
$"{s.Name} is {s.Width:F2}m wide and is {(s.IsRed ? "red":"not red")}“
```

How it should be formatted (optional)

Turn on string interpolation

What you want to insert into the resulting string

Arbitrary code is allowed, not just fields

- String interpolation may be used anywhere
- Improved readability and performance
- Use {{ and }} as escaped versions of { and }
- Safer than working with placeholder {0}
- Can be combined with @: `$@"c:\{pathname}"`

Parameters

- Default passing: By-value
- By-reference modifiers:

```
void PEx(int i, ref int ref_i, out int out_i)
```

  - Implications on assignment
  - Can be used to return values
  - `out` variables *must* be assigned to
  - `out` variables discarded by calling them with _ (e.g. `PEx`(1, ref i, out _)

- Variable number of parameters:

```
void Parameters(int i, params int[] ints)
```

  - Must be an array
  - Must be the last parameter

Optional parameters

```
void Parameters(int i = 0, int j = 0)
```

If a parameter is not supplied by caller, C# will use the default value:

- Calling `Parameters()` will use `0` for `i` and `0` for `j`.
- Calling `Parameters(1)` will use 1 for `i` and `0` for `j`.
- Calling `Parameters(1, 2)` will use 1 for `i` and 2 for `j`.
- Calling `Parameters(j: 2)` will use `0` for `i` and 2 for `j`.

"Named argument"

Worksheet – Part 1 & 2

Indexers


```
var s = "hello";
Console.WriteLine(s[0]);
```

- Implementation

```
class Portfolio
{
    Stock[] stocks;

    public Stock this[int index] //indexer implementation
    {
        get { return this.stocks[index]; }
        set { this.stocks[index] = value; }
    }

    ...
```

- Usage

```
Console.WriteLine(portfolio[i].Symbol);
```

Abstract classes

```
abstract class Stream
{
    public abstract void Write(char ch);
    public void WriteString(string s)
    {
        foreach (char ch in s)
            Write(s);
    }
}

class File : Stream
{
    public override void Write(char ch)
    {
        ...
```

- Abstract methods have no implementation
- Abstract methods are implicitly ***virtual***
- A class with abstract methods must be *abstract* itself
- One cannot create objects of abstract classes

Interfaces

```
public interface IList : ICollection, IEnumerable
{
    int Add(object value);                  //methods
    bool Contains(object value);
    bool IsReadOnly { get; }                //property
    object this [int index] { get; set; } //indexer
    void Log(string t) {Console.WriteLine(Prefix + t);} //default impl.
    static string Prefix=""; //static field
}
```

- Interface = only signatures, no implementation
- May contain methods, properties, indexers, events and fields
  (no fields, constants, constructors, destructors, operators or nested types)
- Interface members are implicitly *public abstract* (*virtual*)
- Interface members can be *static*
- Classes and structs may implement multiple interfaces
- Interfaces can extend other interfaces

Partial classes

Visual Studio uses partial classes to separate generated code from your code in several files:

HelloWorld.Generated.cs

```
partial class Hello // machine-generated code
{

    private String name;
    private void Greet() {...}
}
```

HelloWorld.cs

```
partial class Hello // manually created code
{

    private void Sing() {...}
}
```

Worksheet – Part 3

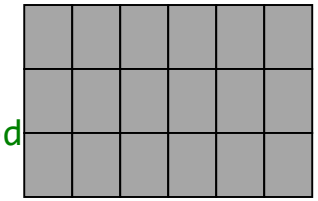
Arrays

One-dimensional arrays

```
var a = new int[6];
int[] c = { 3, 4, 5 };
var d = new String[10]; //array of references
```

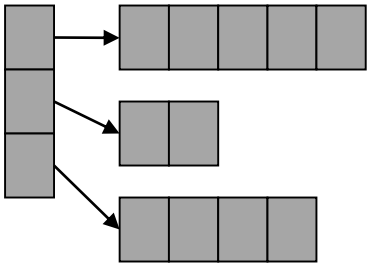
Multidimensional arrays (rectangular)

```
var a = new int[4, 6]; //block matrix
int[,] b = { { 1, 2, 3 }, { 4, 5, 6 } }; //can be initialized
int[, ,] c = new int[2, 4, 2];
```



Multidimensional arrays (jagged)

```
int[][] a = new int[2][];      //an array of arrays
a[0] = new int[] { 1, 2, 3 }; //cannot be initialized
a[1] = new int[] {4, 5, 6};   //directly
```



Arrays – Ranges/Indices

Indices

```
int[] c = { 3, 4, 5 };
int i2 = c[^1] //last element of the array
int i1 = c[^2] //second last element of the array

Index x1 = ^3;
int i3 = c[x1] //third last element of the array
```

Ranges

```
int[] c = { 3, 4, 5 };
int[] c1 = c[1..]; //slice array from second element to the end
int[] c2 = c[..2]; // slice array till second element from start
                   // (exclusive element at position 2)
int[] c3 = c[^2..]; // take the two last elements of the array

Range r1 = 1..3;
int[] c4 = c[r1]; //take subarray from second to third element
```

Switch

```
switch (country)
{
    case "England":
    case "USA":
         return "English";
    case "Germany":
    case "Austria":
         return "German";
    case null:
         throw new NullReferenceException();
    default:
         Console.WriteLine($"don't know the language of {country}");
         break;
}
```

- **No fall-through** (unlike in C and Java)
  Every block **must** be terminated with `break`, `return`, `goto` or `throw`

Switch: Pattern matching


```
object o = "ecnf";

switch (o)
{
    case byte b:
         Console.WriteLine($"I’m a byte with value {b}");
         break;
    case string s when s == "ecnf":
         Console.WriteLine("I’m THE ecnf string");
         break;
    case string s:
         Console.WriteLine("I’m a string that contains {0}", s);
         break;
    default:
         Console.WriteLine("Don’t know anything");
         break;
}
```

- Is very usefull with the “dynamic” keyword (more on this later)

Switch: Expressions

- Switch one or more values with expressions

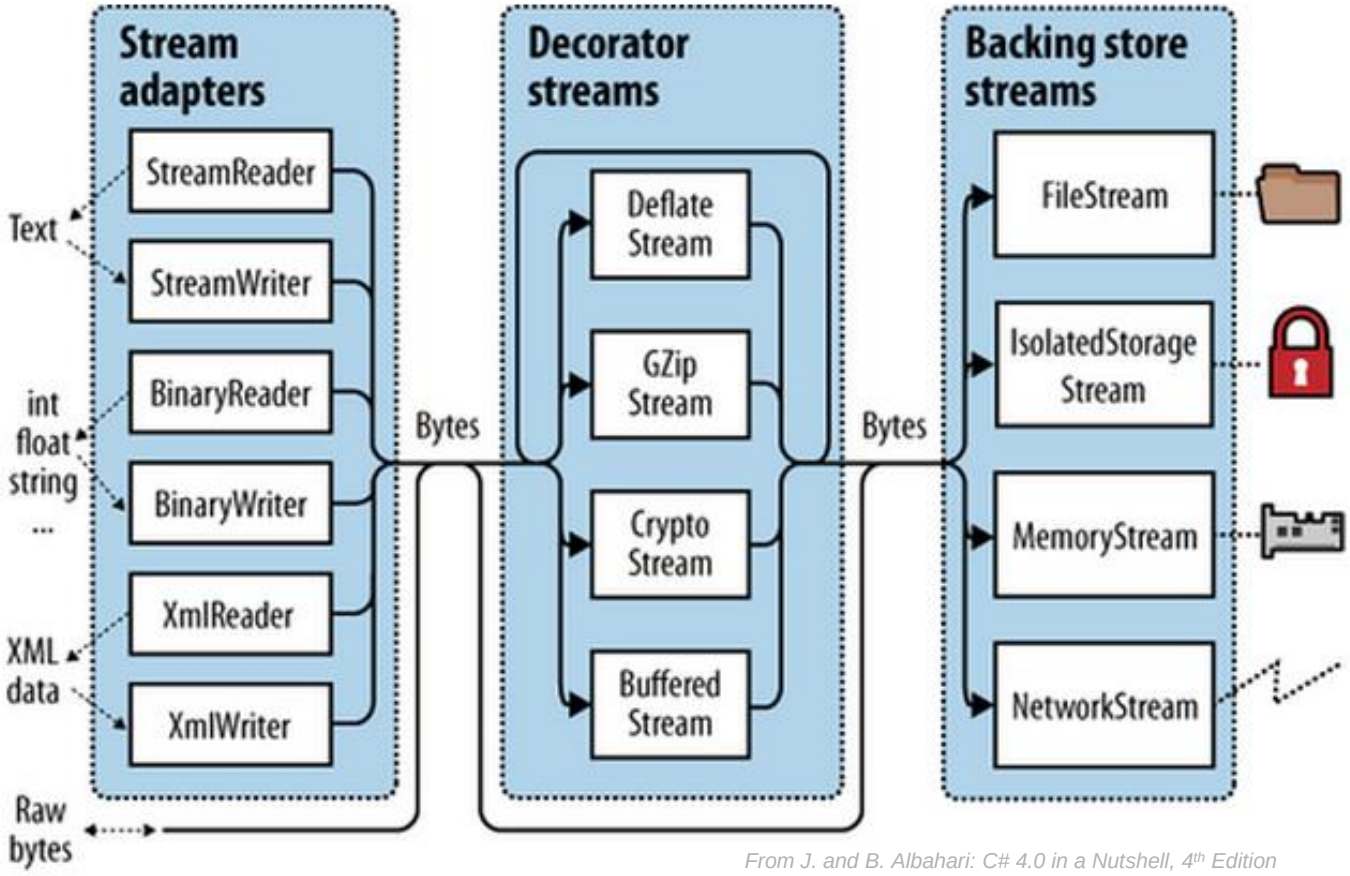
```
string module = "ecnf";
char grade = 'B';

string msg = (module, grade) switch
{
   ("ecnf",'A') => "Congrats, regards Yves Senn",
   ("ecnf",'B') => "Very good, regards Yves Senn"
   ("ecnf",'C') => "Good job, regards Yves Senn",
   ("oop1",'A') => "Nice one, regards Dieter Holz",
   ("oop1",'B') => "Well done, regards Dieter Holz",
   ("oop1",'C') => "Good job, regards Dieter Holz",
   _ => throw new InvalidArgumentException() // equivalent to default
}
```

- Details on lambdas will follow in lesson 5

I/O

I/O Stream Model

Stream abstraction



From J. and B. Albahari: C# 4.0 in a Nutshell, 4th Edition

I/O Streams

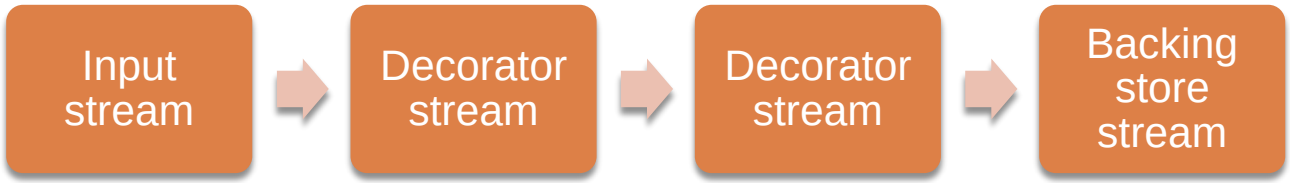
- Backing store streams

  Hard-wired to particular type of store like, files, network, memory, ...

- Decorator streams

  Transformations like encryption, compression, etc.

  - Transform the input in some way and feed the result into an other stream
  - Can be chained



Working with streams

Working with file streams:

```
using (var stream = new FileStream("test.txt", FileMode.Create))
{
    Console.WriteLine(stream.CanRead);  // true
    Console.WriteLine(stream.CanWrite); // true
    Console.WriteLine(stream.CanSeek);  // true

    stream.WriteByte(201);
    stream.WriteByte(210);
    stream.Position = 0;
    Console.WriteLine(stream.ReadByte());
}
```

`using` vs. ~~`.Dispose()`~~ vs. ~~`.Close()`~~

I/O – Typed streams

Typed streams for convenience:

- `StreamReader`/`StreamWriter` for text
- `BinaryReader`/`BinaryWriter` for `int`/`double`/`string`/…
- `XMLTextReader`/`XMLTextWriter` for XML data
- …

I/O – TextReader/Writer

Using `TextReader`/`TextWriter` for text files:

```
using (var writer = new StreamWriter("text.txt"))
{
    writer.WriteLine("First line.");
    writer.WriteLine("Last line.");
}
```

```
using (var reader = new StreamReader("text.txt"))
{
    Console.WriteLine(reader.ReadLine());
    Console.WriteLine(reader.ReadLine());
}
```

- Since C# 8: “using” don’t need braces themselves. Without braces will be disposed when the current block ends.

Inheritance

Inheritance

```
class B : A                 // subclass (inherits from A, extends A)
{
    int b;
    public B() {...}
    public void G() {...}
}
```

- Constructors are not inherited
- Inherited methods can be overridden (see later)
- Classes can only inherit from a single base class, but can implement multiple interfaces (applies to structs as well)
- Classes can only inherit from classes, not from structs
- Classes without explicit base class inherit from *`object`*

Overriding methods

Only `virtual` methods can be overridden in subclasses

```
class A
{
    public void F() {...} // cannot be overridden
    public virtual void G() {...} // can be overridden in a subclass
}
```

Overriding methods must be marked with `override`

```
class B : A
{
    // warning: hides inherited F(), default is new
    public void F() {...}
    // warning: hides inherited G(), default is new
    public void G() {...}

    // ok: overrides inherited G()
    public override void G() {...}
    ...
}
```

- Properties and indexers can also be overridden (virtual, override).
- Static methods cannot be overridden.

Hiding methods

Members can be declared as new in a subclass. They **hide** inherited members with the same name and signature.

```
class Vehicle
{
    public int x;
    public void F() {...}
    public virtual void G(){...}
}
```

```
class Car : Vehicle
{
    public new int x;
    public new void F() {...}
    public new void G() {...}
}
```

```
Car b = new Car();
b.x = ...; // accesses Car.x
b.F(); ... b.G(); // calls Car.F and Car.G

((Vehicle)b).x = ...; // accesses Vehicle.x!
((Vehicle)b).F(); ... // calls Vehicle.F
((Vehicle)b).G(); // Vehicle.G!
```

Constructors and inheritance

Implicit call of the base class constructor

```
class A
{
  ...
}

class B : A
{
  public B(int x)
  {...}
}
```

```
var b = new B(3);
```

OK
- Default constr. `A()`
- `B(int x)`

```
class A
{
  public A()
  {...}
}

class B : A
{
  public B(int x)
  {...}
}
```

```
var b = new B(3);
```

OK
- `A()`
- `B(int x)`

```
class A
{
  public A(int x)
  {...}
}

class B : A
{
  public B(int x)
  {...}
}
```

```
var b = new B(3);
```

DOES NOT WORK
- no explicit call of the `A()` constructor
- default constr. `A()` does not exist

Explicit call

```
class A
{
  public A(int x)
  {...}
}

class B : A
{
  public B(int x)
    :base(x)
  {...}
}
```

```
var b = new B(3);
```

OK
- `A(int x)`
- `B(int x)`

Type checks and casts

- **Run-time type checks – the "is" operator**

```
var a = new C();
if (a is C) ...// true, if the dynamic type of a is C or a subclass; otherwise false
a = null;
if (a is C) ...// false: if a == null, a is never T
```

- **Cast: type cast with runtime exception**

```
A a = new C();
B b = (B)a;   // if a can be cast to b (B is superclass/interface of A)
C c = (C)a;   // exception otherwise

a = null;
c = (C)a;     // ok: null can be cast to any reference type
```

- **as: similar to (T)v but no runtime exception**

```
A a = new B();
B b = a as B; // (if (a is B) == true) b = (B)a; else b = null;
C c = a as C; // c  == null, because a is not of type C: (a is C) == false

a = null;
c = a as C;   // c == null
```

Worksheet – Part 4

References

Generics

Generics

- Generics express reusability through placeholder types

```
class Buffer <TElement>
{
    private TElement[] data;
    public void Put(TElement x) {...}
    public void Get(out TElement x) {...}
}
```

- Inheritance expresses reusability through base types
- Similar to Java Generics and C++ Templates
- .NET provides
  - Generic collections interfaces
  - Generic collection classes

Generics Example

Buffer with priorities

```
class Buffer <TElement, TPriority>
{
    private TElement[] data;
    private TPriority[] prio;
    public void Put(TElement x, TPriority prio) {...}
    public void Get(out TElement x, out TPriority prio) {...}
}
```

Usage

```
var a = new Buffer<int, int>();
a.Put(100, 0);
int elem, prio;
a.Get(out elem, out prio);

var b = new Buffer<Rectangle, double>();
b.Put(new Rectangle(), 0.5);
Rectangle r; double prio;
b.Get(out r, out prio);
```

Generic constraints

Constraints about placeholder types are specified as base types

```
class OrderedBuffer <TElement, TPriority> where TPriority: IComparable
{
    TElement[] data;
    TPriority[] prio;
    int lastElement;

    // sorts x according to its priority into buffer
    public void Put(TElement x, TPriority p)
    {
        var i = lastElement;
        while (i >= 0 && p.CompareTo(prio[i]) > 0)
        {
            ...
```

interface or base class

Allows operations on instances of placeholder types

Usage

```
var a = new OrderedBuffer<int, int>();
a.Put(100, 3);
```

parameter must implement IComparable

Generic and inheritance

```
class Buffer<TElement> : List<TElement>
{
    public void Put(TElement x)
    {
        this.Add(x);  // Add inherited from List
    }
}
```

can also implement generic interfaces

From which classes can a generic class derive?

- from a non-generic class
  ```
  class T<X>: B {...}
  ```
- from a concrete generic class
  ```
  class T<X>: B<int> {...}
  ```
  Concrete type
- from a generic class with the same placeholder
  ```
  class T<X>: B<X> {...}
  ```

Constraints

```
public class BaseClass<T> where T : ISomeInterface
{...}
```

- Must be repeated
  ```
  public class SubClass<T> : BaseClass<T> where T: ISomeInterface
  {...}
  ```

Generic methods

Working with arbitrary data types

```
static void Sort<T>(T[] a) where T : IComparable
{
    for (int i = 0; i < a.Length - 1; i++)
        for (int j = i + 1; j < a.Length; j++)
            if (a[j].CompareTo(a[i]) < 0)
            {
                T x = a[i];
                a[i] = a[j];
                a[j] = x;
            }
}
```

Sorts any array, as long as the elements implement *IComparable*

```
int[] a = {3, 7, 2, 5, 3};
Sort<int>(a);
```

```
string[] s = {"one", "two", "three"};
Sort<string>(s);
```

Generic type inference

The compiler usually infers the type from the parameters:

```
// a == {2, 3, 3, 5, 7}
Sort(a);
```

```
// s == {"one", "three", "two"}
Sort(s);
```

Collections (Generic)