The Language C#

Hanspeter Mössenböck

Features of C#

Very similar to Java

70% Java, 10% C++, 5% Visual Basic, 15% new

Like in Java

- object-orientation (single inheritance)
- interfaces
- exceptions
- threads
- namespaces (like packages)
- strict typing
- garbage collection
- reflection
- dynamic loading
- ...

Like in C++

- operator overloading
- pointer arithmetic in unsafe code
- some syntactic details

New Features in C#

Really new (vs. Java)

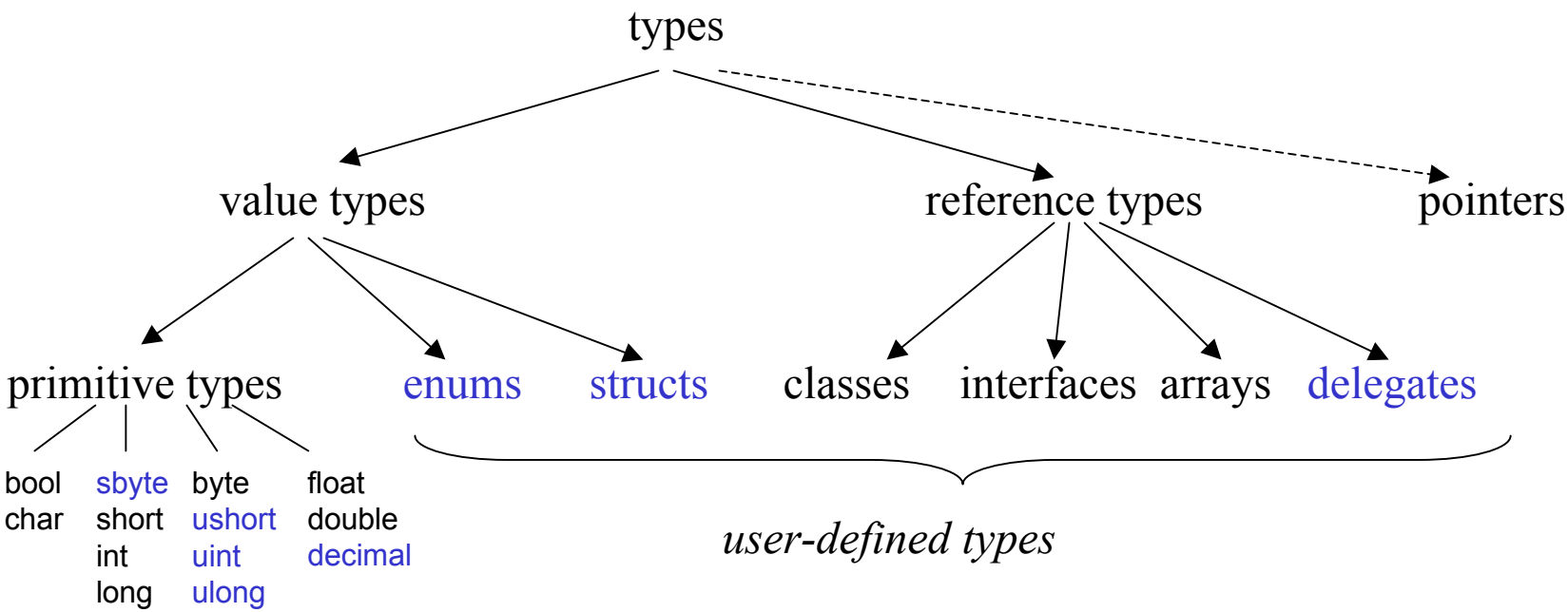
- call by reference parameters
- objects on the stack (structs)
- block matrixes
- enumerations
- uniform type system
- goto statement
- attributes
- low-level programming
- versioning
- compatible with other .NET languages

"Syntactic Sugar"

- support for components
  - properties
  - events
- delegates
- indexers
- operator overloading
- foreach statement
- boxing/unboxing
- ...

Type System

Uniform Type System



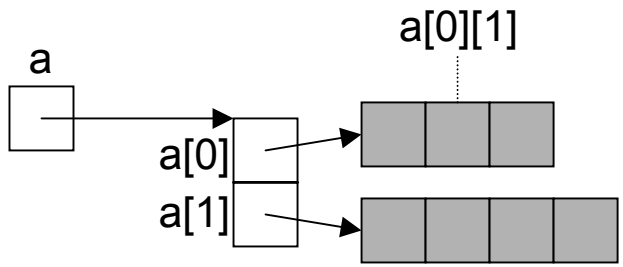
All types are compatible with the type *object*

- can be assigned to *object* variables
- can perform *object* operations

Multi-dimensional Arrays

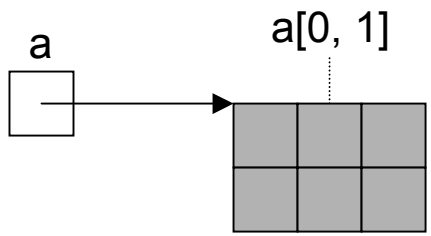
Jagged (like in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];

int x = a[0][1];
```



Rectangular (compact, efficient)

```
int[,] a = new int[2, 3];

int x = a[0, 1];
```



Boxing and Unboxing

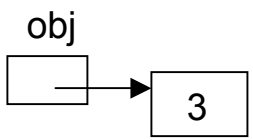
Even value types (int, struct, enum) are compatible with *object*!

Boxing

In the assignment

```
object obj = 3;
```

the value 3 is wrapped up in a heap object




```
class TempInt {
    int val;
    TempInt(int x) {val = x;}
}

obj = new TempInt(3);
```

Unboxing

In the assignment

```
int x = (int) obj;
```

the *int* value is unwrapped again

Boxing/Unboxing

Allows generic container types

```
class Queue {
        ...
        public void Enqueue(object x) {...}
        public object Dequeue() {...}
        ...
}
```

This *Queue* can be used with reference types and value types

```
Queue q = new Queue();

q.Enqueue(new Rectangle());
q.Enqueue(3);

Rectangle r = (Rectangle) q.Dequeue();
int x = (int) q.Dequeue();
```

Object-orientation

Classes and Inheritance

C#

```
class A {
    private int x;
    public A(int x) { this.x = x; }
}

class B : A, I1, I2 {
    private int y;

    public int Bar() { ...}

    public B(int x, int y) : base(x) {
        this.y = y;
    }
}
```

Java

```
class A {
    private int x;
    public A(int x) { this.x = x; }
}

class B extends A implements I1, I2 {
    private int y;

    public int Bar() { ...}

    public B(int x, int y) {
        super(x);
        this.y = y;
    }
}
```

Overriding Methods

C#

```
class A {
    ...
    public virtual void Foo() { ...}
}

class B : A {
    ...
    public override void Foo() {
        base.Foo();
        ...
    }
}
```

Java

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

class B extends A {
    ...
    public void Foo() {
        super.Foo();
        ...
    }
}
```

Method Parameters

value parameters (input parameters)

call by value

```
void Inc(int x) {x = x + 1;}

void Foo() {
    int val = 3;
    Inc(val); // val == 3
}
```

ref parameters (input/output parameters)

call by reference

```
void Inc(ref int x) { x = x + 1; }

void Foo() {
    int val = 3;
    Inc(ref val); // val == 4
}
```

out parameters (output parameters)

like a ref parameter, but used for returning values from a method.

```
void Read (out int first, out int next) {
    first = Console.Read(); next = Console.Read();
}

void Foo() {
    int first, next;
    Read(out first, out next);
}
```

Properties

Syntactic abbreviation for get/set methods

```
class Data {
    FileStream s;

    public string FileName {
        set {
            s = new FileStream(value, FileMode.Create);
        }
        get {
            return s.Name;
        }
    }
}
```

property type

property name

"input parameter" of the *set* operation

Used like fields ("smart fields")

```
Data d = new Data();

d.FileName = "myFile.txt";      // calls d.set("myFile.txt")
string s = d.FileName;          // calls d.get()
```

Due to inlining the get/set calls are as efficient as field accesses.

Properties (continued)

get or _set_ can also be missing

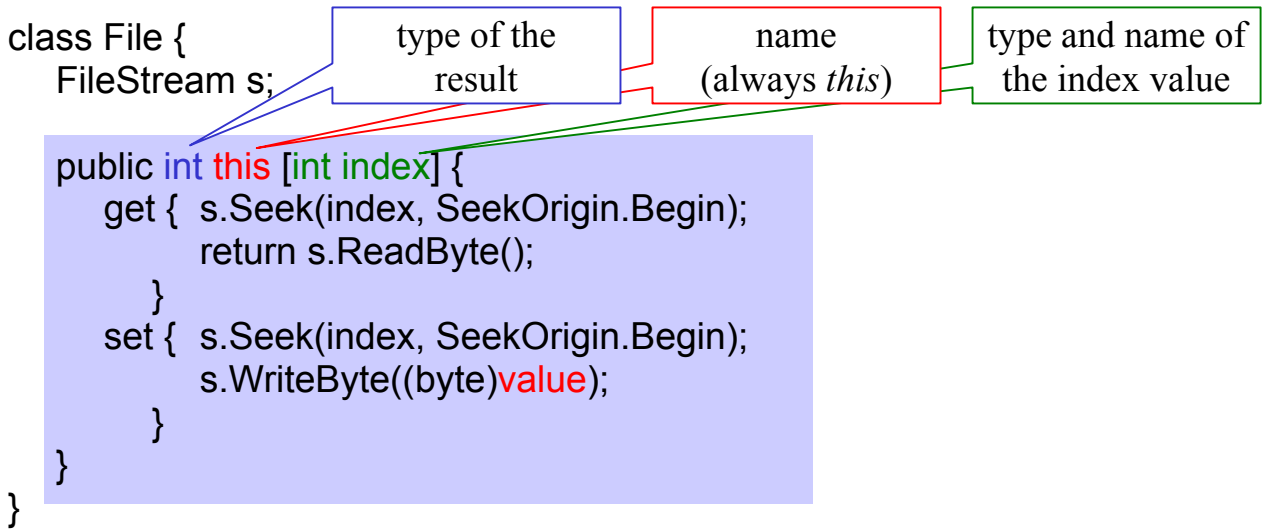
```
class Account {
    long balance;

    public long Balance {
        get { return balance; }
    }
}

x = account.Balance;        // ok
account.Balance = ...;      // forbidden
```

Why properties?

- they allow read-only and write-only data
- they allow plausibility checks for field accesses
- the interface and the implementation of the data can differ

Indexers

User-defined operations for indexing a collection



Usage

```
File f = ...;
int x = f[10];          // calls f.get(10)
f[10] = 'A';            // calls f.set(10, 'A')
```

- *get* or *set* operation can be missing (write-only or read-only indexers)
- indexers can be overloaded with different index types

Indexers (another example)


```
class MonthlySales {
      int[] apples = new int[12];
      int[] bananas = new int[12];
      ...
      public int this[int i] {                    // set operation missing => read-only
          get { return apples[i-1] + bananas[i-1]; }
      }

      public int this[string month] {         // overloaded read-only indexer
          get {
              switch (month) {
                  case "Jan": return apples[0] + bananas[0];
                  case "Feb": return apples[1] + bananas[1];
                  ...
              }
          }
      }
}

MonthlySales sales = new MonthlySales();
...
Console.WriteLine(sales[1] + sales["Feb"]);
```

Dynamic Binding (with Hiding)


```
class Animal {
    public virtual void WhoAreYou() { Console.WriteLine("I am an animal"); }
}
```

```
class Dog : Animal {
    public override void WhoAreYou() { Console.WriteLine("I am a dog"); }
}
```

```
class Beagle : Dog {
    public new virtual void WhoAreYou() { Console.WriteLine("I am a beagle"); }
}
```

```
class AmericanBeagle : Beagle {
    public override void WhoAreYou() { Console.WriteLine("I am an American beagle"); }
}
```

```
Beagle beagle = new AmericanBeagle();
beagle.WhoAreYou();                    // "I am an American beagle"

Animal animal = new AmericanBeagle();
animal.WhoAreYou();                    // "I am a dog" !!
```

Delegates

Delegate = Method Type

Declaration of a delegate type

```
delegate void Notifier (string sender);      // normal method signature
                                             // with the keyword delegate
```

Declaration of a delegate variable

```
Notifier notify;
```

Methods can be assigned to a delegate variable

```
void SayHello(string sender) {
    Console.WriteLine("Hello from " + sender);
}

notify = new Notifier(SayHello);
```

Delegate variables can be "called"

```
notify("Max");                               // invokes SayHello("Max") => "Hello from Max"
```

Assignment of different methods

Any compatible method can be assigned to a delegate variable

```
void SayGoodBye(string sender) {
    Console.WriteLine("Good bye from " + sender);
}

notify = new Notifier(SayGoodBye);

notify("Max");          // SayGoodBye("Max") => "Good bye from Max"
```

Creation of delegate values

new DelegateType (obj.Method)

- *obj* can be *this* (i.e. it can be missing)
- *Method* can be *static*. In this case *obj* denotes the class name.
- *Method* signature must be compatible with *DelegateType*
  - same number of parameters (and same order)
  - same parameter types (including the result type)
  - same parameter kinds (ref, out, value)

A delegate variable stores a method ***and*** its receiver object !

Multicast Delegates

A delegate variable can store multiple methods!

```
Notifier notify;
notify = new Notifier(SayHello);
notify += new Notifier(SayGoodBye);
```

```
notify("Max");                    // "Hello from Max"
                                  // "Good bye from Max"
```

```
notify -= new Notifier(SayHello);
```

```
notify("Max");                    // "Good bye from Max"
```

Threads

Threads

C#

```
void P() {
    ... thread actions ...
}

Thread t = new Thread(new ThreadStart(P));
t.Start();
```

- Requires no subclass of *Thread*
- Any parameterless void method can be started as a thread.

Java

```
class MyThread extends Thread {
    public void run() {
        ... thread actions ...
    }
}

Thread t = new MyThread();
t.start();
```

- Requires a custom thread class, which must be a subclass of *Thread*.
- The method *run* must be overridden.
- (or the interface *Runnable* must be implemented).

Thread Synchronisation

C#

```
class Buffer {
   int[] buf = new int[10];
   int head = 0, tail = 0, n = 0;

   void put(int data) {
      lock(this) {
         while (n == 10) Monitor.Wait(this);
         buf[tail] = data; tail = (tail+1)%10; n++;
         Monitor.PulseAll(this);
      }
   }

   int get() {
      lock(this) {
         while (n == 0) Monitor.Wait(this);
         int data = buf[head];
         head = (head+1)%10; n--;
         Monitor.PulseAll(this);
         return data;
      }
   }
}
```

Java

```
class Buffer {
   int[] buf = new int[10];
   int head = 0, tail = 0; n = 0;

   void put(int data) {
      synchronized(this) {
         while (n == 10) wait();
         buf[tail] = data; tail = (tail+1)%10; n++;
         notifyAll();
      }
   }

   int get() {
      synchronized(this) {
         while (n == 0) wait();
         int data = buf[head];
         head = (head+1)%10; n--;
         notifyAll();
         return data;
      }
   }
}
```

Attributes

Attributes

User-defined information about program elements

- Can be attached to types, members, assemblies, etc.
- Stored in the metadata of the assembly
- Implemented as subclasses of *System.Attribute*
- Can be queried at run time

Example

```
[Serializable]
class C {...}                          // makes C serializable
```

Possible to attach multiple attributes

```
[Serializable] [Obsolete]
class C {...}

[Serializable, Obsolete]
class C {...}
```

Example: Conditional Attribute

Conditional invocation of methods

```
#define debug                                   // preprocessor command

class C {

   [Conditional("debug")]                       // can only be attached to void methods
   static void Assert (bool ok, string errorMsg) {
      if (!ok) {
         Console.WriteString(errorMsg);
         System.Environment.Exit(0);      // terminates the program
      }
   }

   static void Main (string[] arg) {
      Assert(arg.Length > 0, "no arguments specified");
      Assert(arg[0] == "...", "invalid argument");
      ...
   }
}
```

Assert is only called if *debug* is defined.
Also useful for trace output.

Example: Serialization


```
[Serializable]
class List {
    int val;
    [NonSerialized] string name;
    List next;

    public List(int x, string s) {val = x; name = s; next = null;}
}
```

[Serializable]	Can be attached to classes. Objects of these classes can be automatically serialized.
[NonSerialized]	Can be attached to fields. These fields are excluded from serialization.

Summary

C# is often more convenient than Java

Java

```
String s = ..;
...
for (int i = 0; i < s.length(); i++)
    process(s.charAt(i));
```

C#

```
String s = ...;
...
foreach (char ch in s)
    process(ch);
```

```
Hashtable tab = new Hashtable();
...
tab.put("John", new Integer(123));
...
int x = ((Integer) tab.get("John")).intValue();
```

```
Hashtable tab = new Hashtable();
...
tab["John"] = 123;
...
int x = (int) tab["John"];
```

C# versus Java

Equivalent features

- object-orientation
- exception handling
- threading
- reflection

Additional features

- compatible with other .NET languages (but predominantly under Windows)
- uniform type system (boxing, unboxing)
- block matrixes
- call by reference parameters
- properties, indexers
- delegates
- attributes
- versioning