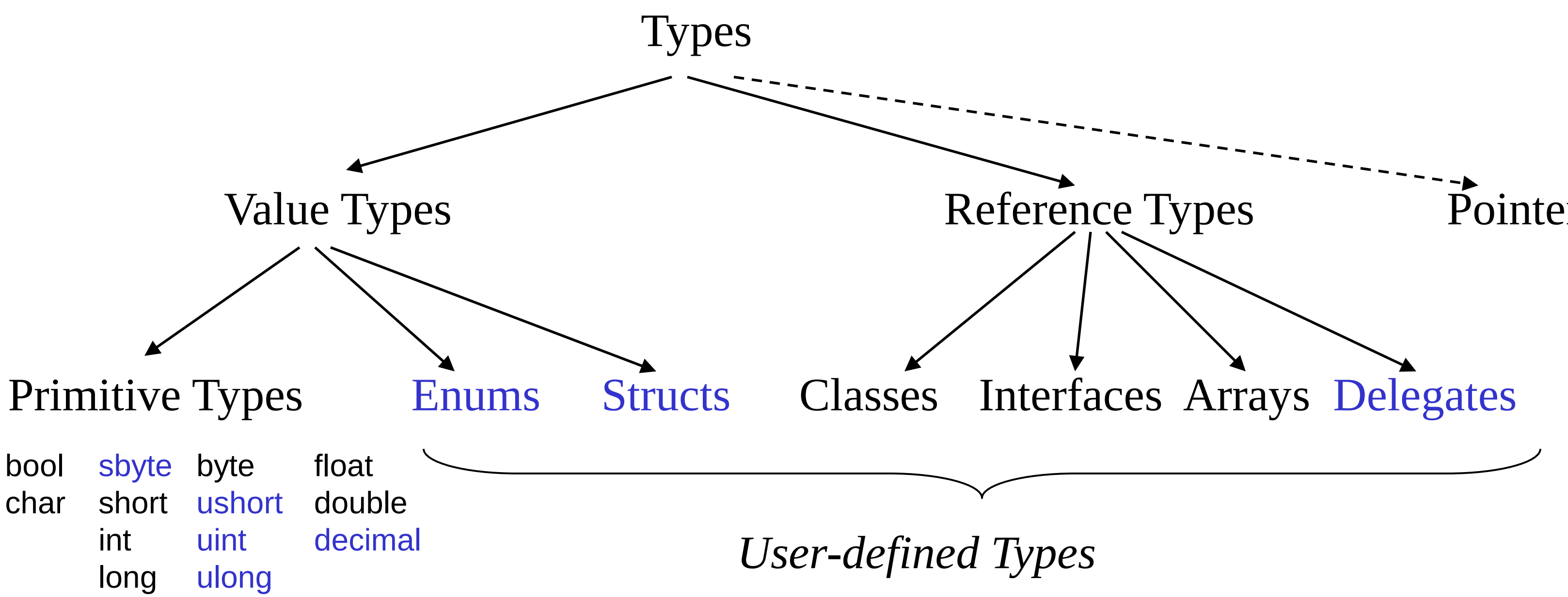
Types

Uniform Type System



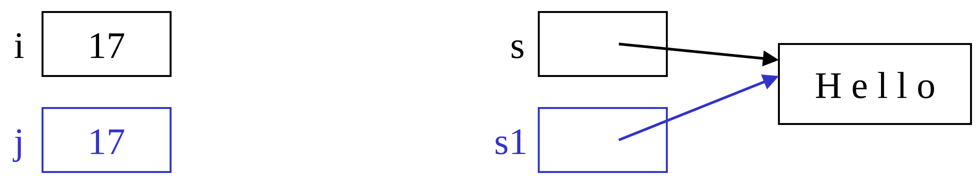
blue types are missing from Java

All types are compatible with *object*

- can be assigned to variables of type *object*
- all operations of type *object* are applicable to them

Value Types and Reference Types

	Value Types	Reference Types
variable contains	value	reference
stored on	stack (or in an object)	heap
initialization	0, false, '\0'	null
assignment	copies the value	copies the reference
example	int i = 17; int j = i;	string s = "Hello"; string s1 = s;



Primitive Types

	long form	in Java	range
sbyte	System.SByte	byte	-128 .. 127
byte	System.Byte	---	0 .. 255
short	System.Int16	short	-32768 .. 32767
ushort	System.UInt16	---	0 .. 65535
int	System.Int32	int	-2^{31} .. $2^{31}-1$
uint	System.UInt32	---	0 .. $2^{32}-1$
long	System.Int64	long	-2^{63} .. $2^{63}-1$
ulong	System.UInt64	---	0 .. $2^{64}-1$
float	System.Single	float	±1.5E-45 .. ±3.4E38 (32 Bit)
double	System.Double	double	±5E-324 .. ±1.7E308 (64 Bit)
decimal	System.Decimal	---	±1E-28 .. ±7.9E28 (128 Bit)
bool	System.Boolean	boolean	true, false
char	System.Char	char	Unicode character

Type decimal

128 bit floating point type

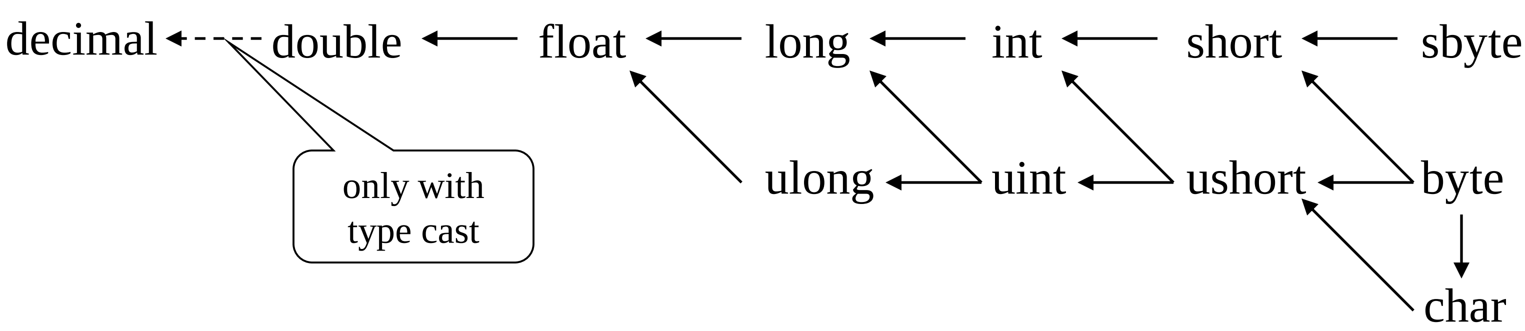
$$(-1)^{s} * m * 10^{-e}$$

$s = 0$ or 1
$0 \leq m < 2^{96}$
$0 \leq e \leq 28$

For calculations with
- large numbers
- high decimal precision (e.g. $0.1 = 1 * 10^{-1}$)

=> e.g. in financial mathematics

Compatibility Between Primitive Types



The following assignments are legal

```
intVar = shortVar;
intVar = charVar;
floatVar = charVar;
decimalVar = (decimal)doubleVar;
```

Enumerations

List of named constants

Declaration (on the namespace level)

```
enum Color {Red, Blue, Green}    // values: 0, 1, 2
enum Access {Personal=1, Group=2, All=4}
enum Access1 : byte {Personal=1, Group=2, All=4}
```

Usage

```
Color c = Color.Blue;      // enumeration constants must be qualified

Access a = Access.Personal | Access.Group;
                                       // a contains a "set" of values now

if ((Access.Personal & a) != 0) Console.WriteLine("access granted");
```

Operations on Enumerations

Valid operations

comparisons	`if (c == Color.Red) ...` `if (c > Color.Red && c <= Color.Green) ...`
+, -	`c = c + 2;`
++, --	`c++;`
&	`if ((c & Color.Red) == 0) ...`
|	`c = c | Color.Blue;`
~	`c = ~ Color.Red;`

The compiler does not check if the result is a valid enumeration value.

Note

- Enumerations cannot be assigned to *int* and vice versa (except after a type cast).
- Enumeration types inherit from *object* (*Equals*, *ToString*, ...).
- Class *System.Enum* (base type of all enumeration types) provides operations on enumerations (*GetName*, *Format*, *GetValues*, ...).

Arrays

One-dimensional arrays

```
int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
SomeClass[] d = new SomeClass[10];      // array of references
SomeStruct[] e = new SomeStruct[10];    // array of values (directly in the array)
```

Multidimensional arrays (jagged)

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

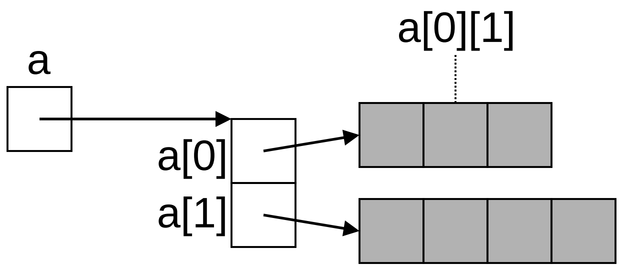
Multidimensional arrays (rectangular)

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

Multidimensional Arrays

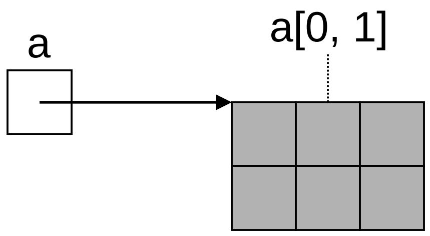
Jagged (like in Java)

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

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



Rectangular (more compact and efficient)

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

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



Other Array Properties

Indexes start at 0

Array length

```
int[] a = new int[3];
Console.WriteLine(a.Length);  // 3
int[][] b = new int[3][];
b[0] = new int[4];
Console.WriteLine("{0}, {1}", b.Length, b[0].Length);   // 3, 4
int[,] c = new int[3, 4];
Console.WriteLine(c.Length);   // 12
Console.WriteLine("{0}, {1}", c.GetLength(0), c.GetLength(1));  // 3, 4
```

System.Array provides some useful array operations

```
int[] a = {7, 2, 5};
int[] b = new int[2];
Array.Copy(a, b, 2);                // copies a[0..1] to b
Array.Sort(b);
...
```

Class System.String

Can be used as the standard type *string*

```
string s = "Alfonso";
```

Note

- Strings are immutable (use *StringBuilder* if you want to modify strings)
- Can be concatenated with +: `"Don " + s`
- Can be indexed: `s[i]`
- String length: `s.Length`
- Strings are reference types => reference semantics in assignments
- but their values can be compared with == and != : `if (s == "Alfonso") ...`
- Class *String* defines many useful operations:
  CompareTo, *IndexOf*, *StartsWith*, *Substring*, ...

Variable-length Arrays

```
using System;
using System.Collections;

class Test {

	static void Main() {
		ArrayList a = new ArrayList();
		a.Add("Charly");
		a.Add("Delta");
		a.Add("Alpha");
		a.Sort();
		for (int i = 0; i < a.Count; i++)
			Console.WriteLine(a[i]);
	}
}
```

Output

```
Alpha
Charly
Delta
```

Associative Arrays

```
using System;
using System.Collections;

class Test {

    static void Main() {
        Hashtable phone = new Hashtable();
        phone["Karin"] = 7131;
        phone["Peter"] = 7130;
        phone["Wolfgang"] = 7132;
        foreach (string key in phone.Keys)
            Console.WriteLine("{0} = {1}", key, phone[key]);
    }
}
```

Output

```
Karin = 7131
Peter = 7130
Wolfgang = 7132
```

Structs

Declaration

```
struct Point {
    public int x, y;                                        // fields
    public Point (int x, int y) { this.x = x; this.y = y; } // constructor
    public void MoveTo (int a, int b) { x = a; y = b; }     // methods
}
```

Usage

```
Point p;                  // still unititialized
Point p = new Point(3, 4); // constructor initializes object on the stack
p.x = 1; p.y = 2;         // field access
p.MoveTo(10, 20);         // method call
Point q = p;              // value assignment of objects (all fields are assigned)
```

Note

- Structs are value types!
  A struct declaration allocates an object directly on the stack or within some other object.
- Structs must not declare a parameterless constructor (they have one by default).
  However, they may use it: `p = new Point(); // initializes fields to 0, null, false, ...`

Classes

Declaration

```
class Rectangle {
	Point origin;
	public int width, height;
	public Rectangle() { origin = new Point(0,0); width = height = 0; }
	public Rectangle (Point p, int w, int h) { origin = p; width = w; height = h; }
	public void MoveTo (Point p) { origin = p; }
}
```

Usage

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int area = r.width * r.height;
r.MoveTo(new Point(3, 3));
Rectangle r1 = r ;  // reference assignment
```

Note

- Classes are *reference types;*
  Their objects are allocated on the heap.
- The "new" operator allocates an object and calls its constructor.
  Classes may declare a parameterless constructor.

Differences Between Classes and Structs

Classes	Structs
Reference types (objects are allocated on the heap)	Value types (objects are allocated on the stack)
support inheritance (all classe are derived from *object*)	no inheritance (but they are compatible with *object*)
can implement interfaces	can implement interfaces
may declare a parameterless constructor	must not declare a parameterless constructor
may have a destructor	no destructors

Class System.Object

Base class of all reference types

```
class Object {
    public virtual bool Equals(object o) {...}
    public virtual string ToString() {...}
    public virtual int GetHashCode() {...}
    ...
}
```

Can be used as the standard type *object*

```
object obj;    // compiler maps object to System.Object
```

Assignment compatibility

```
obj = new Rectangle();
obj = new int[3];
```

Allows you to write methods that work on arbitrary objects

```
void Push(object x) {...}

Push(new Rectangle());
Push(new int[3]);
```

Boxing and Unboxing

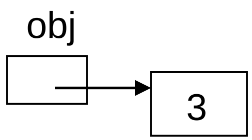
Value types (int, struct, enum) are compatible with *object*!

Boxing

The assignment

```
object obj = 3;
```

wraps up the value 3 in a heap object



Unboxing

The assignment

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

unwraps the value again

Boxing/Unboxing

Allows the implementation of "generic" container types

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

This *Queue* can then be used for reference types <u>and</u> value types

```
Queue q = new Queue();

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

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

But there is also true genericity (see later)