



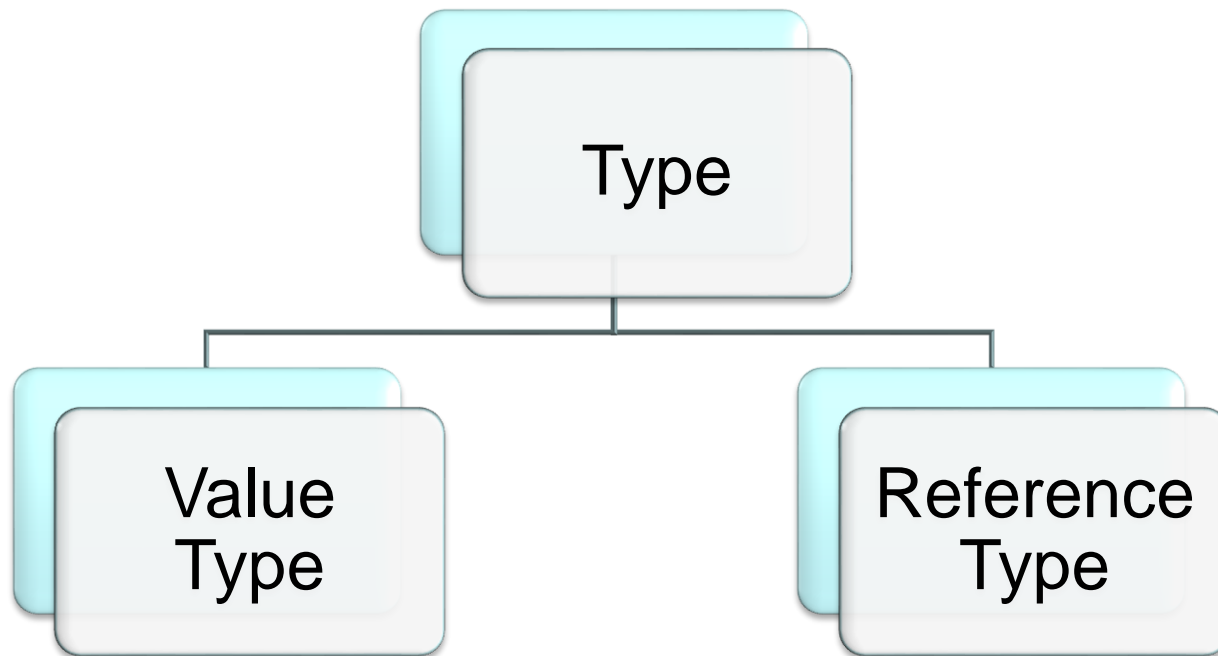
Module 3

Value Types and Expressions



Anatomy of the Common Type System

- Every variable has a specified type
- C# is type-safe...!





Memory allocation in C#

Stack

- Static memory
- Contains values
- Have to be initialized
- Local variables and value types
- **Types**
 - int, long etc (integral types)
 - double, float (floating-point types)
 - decimal
 - bool
 - DateTime
 - struct
 - enumerations
 - char

Stack Frame / Call stack / Scope

Heap

- Dynamic memory
- Store references to data ("objects")
- Reference types
- Has a default value of null
- Several references can refer to same data
- **Types**
 - string
 - object
 - class
 - interface
 - delegate
 - dynamic
 - array



Value types (stack)

- Value types goes on the stack
 - Variables contain **values**

```
int i = 0;  
double d = 0;  
bool b = true;  
Point p = new Point(x: 0, y: 0);
```

Stack
i = 0
d = 0
b = true
p = { X=0, Y=0 }



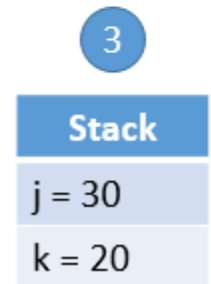
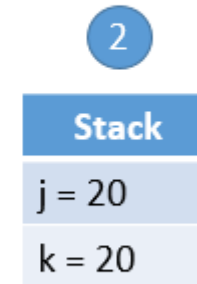
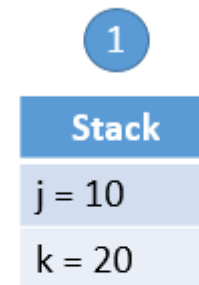
Value types (stack)

- Value types goes on the stack
 - Values are **copied** from one variable to another

1 `int j = 10;`
`int k = 20;`

2 `j = k;` *// copy (j=20, k=20)*

3 `j = 30;` *// no ref (j=30, k=20)*





Value types (stack)

- Value types goes on the stack
 - Values are **copied** from one variable to another

```
1 Point p1 = new Point(x: 100, y: 100);  
2 Point p2 = new Point(x: 200, y: 200);  
3 p1 = p2; // copy (p1.X = 200, p1.Y = 200)  
           //      (p2.X = 200, p2.Y = 200)  
p1.X = 300; // no ref (p1.X = 300, p1.Y = 200)  
           //      (p2.X = 200, p2.Y = 200)
```

1

Stack

```
p1 = {  
  X = 100,  
  Y = 100  
}
```

```
p2 = {  
  X = 200,  
  y = 200  
}
```

2

Stack

```
p1 = {  
  X = 200,  
  Y = 200  
}
```

```
p2 = {  
  X = 200,  
  y = 200  
}
```

3

Stack

```
p1 = {  
  X = 300,  
  Y = 200  
}
```

```
p2 = {  
  X = 200,  
  y = 200  
}
```



Overview of Predefined Value Types

C# Data Type	CTS Type	Description
bool	System.Boolean	True or false values
int	System.Int32	Signed integers
short	System.Int16	Signed short integers
long	System.Int64	Signed long integers
uint	System.UInt32	Unsigned integers
char	System.Char	Character values
float	System.Single	Single-precision floating
double	System.Double	Double-precision floating
decimal	System.Decimal	128-bit precision number
	System.DateTime	An instant in time
	System.TimeSpan	A time interval




Declaring Variables

- Declare by data type and variable name
- Multiple variables can be declared simultaneously
- Local variables can be declared everywhere in methods
- Variables are “scoped”

```
int i;  
int x, y;  
double myNumber;  
bool isStarted;
```

- You cannot use an unassigned variable

```
int u;  
u = u + 1;
```

 (local variable) `int u`

Use of unassigned local variable 'u'



Assigning Values

- Assign values by using the assignment operator =
- Variables can be declared and assigned simultaneously

```
bool isStarted;  
isStarted = false;
```

```
bool isStarted = false;
```

```
int i = 10;  
double number = 2003.45;  
DateTime date = DateTime.Now;
```



Naming of Variables

- Compiler requires that only letters, digits, and underscores are used
- C# keywords are reserved
- Case-sensitive!
- Recommendations
 - Don't abbreviate. Characters are cheap! 😊
 - Use camelCasing for variables
 - Use PascalCasing for types, classes, methods.
 - Use StyleCop



Members from ValueType

- All value types will always contain several methods – including
 - ToString()
 - Equals()
 - GetType()
 - GetHashCode()
- ToString() can convert a value to a string value – with or without formatting

```
int i = 10596050;
Console.WriteLine(i.ToString());           // 10596050
Console.WriteLine(i.ToString("n2"));       // 10.596.050,00
DateTime d = DateTime.MinValue;
Console.WriteLine(d.ToString("dddd d MMMMM yyyy")); // monday 1 january 0001
```

- Find format codes on MSDN or <https://goo.gl/ZdFfhv>



Constants

- Use the const keyword to declare constants
- Must be initialized when declared

```
const int favoriteNumber = 87;
```

- Cannot change at runtime – needs recompiling

```
favoriteNumber = 90;
```

 (local constant) int favoriteNumber = 87

The left-hand side of an assignment must be a variable, property or indexer

- Other ways of using application scoped variables (like VAT – that could change in the future)
 - XML/JSON/DB
 - App/User scoped settings



Operators

Operator Type	Operators
Equality	== !=
Relational	< <= > >= is
Conditional	(or) && (and)
Arithmetic	+ - * / %
Increment, Decrement	++ --
Assignment	= += -= *= /=



Operator Precedence

- Operator precedence determines order of evaluation
 - Multiplicative > Additive.
- Use parentheses whenever there is doubt!

```
int y = 2, z = 10;  
int x = y + 5 * z;    // z * 5 = 50 + 2 = 52  
int w = y + (5 * z);
```



Implicit Conversions

- Also known as “widening” conversions
- Never lose precision or value
- Are always allowed by the compiler
- Always succeeds

```
short i = 16384;  
int j = i;    // ok  
i = j;        // not ok
```

 (local variable) `int j`

Cannot implicitly convert type 'int' to 'short'. An explicit conversion exists (are you missing a cast?)



Explicit Conversions

- Also known as “narrowing” conversions
- Can lose precision or value
- Are allowed by the compiler

```
int i = 50000000;  
short j = (short)i;           // Under/overflow (-3968)
```

- Use `System.Convert` or the `checked` keyword to avoid under/overflow and use error handling instead

```
short k = Convert.ToInt16(i); // Exception  
checked {  
    short l = (short)i;       // Exception  
}
```




Implicitly Typed Variables

- You can define local implicitly typed variables using the var keyword

```
var myInteger = 87;  
var myBoolean = true;  
var myDouble = 20.23;
```

- The compiler infers the type of the local variable!
- Everything is still completely type-safe
- Must be assigned a value when declared
- Very practical when you “don’t know” the type – just let the compiler figure it out
 - Way to advanced example – just to illustrate the point

```
int[] numbers = { 2, 6, 5, 2, 1, 2, 6, 5, 4, 2 };  
var res = numbers.OrderBy(i=>i).GroupBy(i => i);  
▸ res {System.Linq.GroupedEnumerable<int, int, int>}
```

- As a beginner you properly should avoid using var



Enumerations

- Used for creating a set of symbolic names
- Many build-in enumerations

```
DayOfWeek w = DayOfWeek.Saturday; // Saturday = 6
```

- Create your own

```
enum Fruit
{
    Apple,
    Banana,
    Orange
}

Fruit f = Fruit.Apple;
```

- Underlying enumeration type can be explicitly chosen

```
enum Team : byte
{
    AGF = 1,
    Brøndby = 6,
    FCK = 5,
    Randers = 12
}

Team t = Team.AGF;
Console.WriteLine(t); // AGF
int r = (int)t;
Console.WriteLine(r); // 1
```



Structures

- Used for defining a structured value consisting of several subvalues

```
struct Point
{
    public int x, y;
}
```

```
Point pt;
pt.x = 42;
pt.y = 100;
Console.WriteLine(pt.x); // 42
```

- Members are private by default
- All subvalues must be initialized before use!
- Value can be default initialized using the new construct
 - All values get initialized

```
Point p1 = new Point();
Console.WriteLine(p1.x); // 0
```

```
Point p2;
Console.WriteLine(p2.x); // Syntax error
```

(local variable) Point p2

Use of possibly unassigned field 'x'