Types & basics

Ссылочные и значимые типы

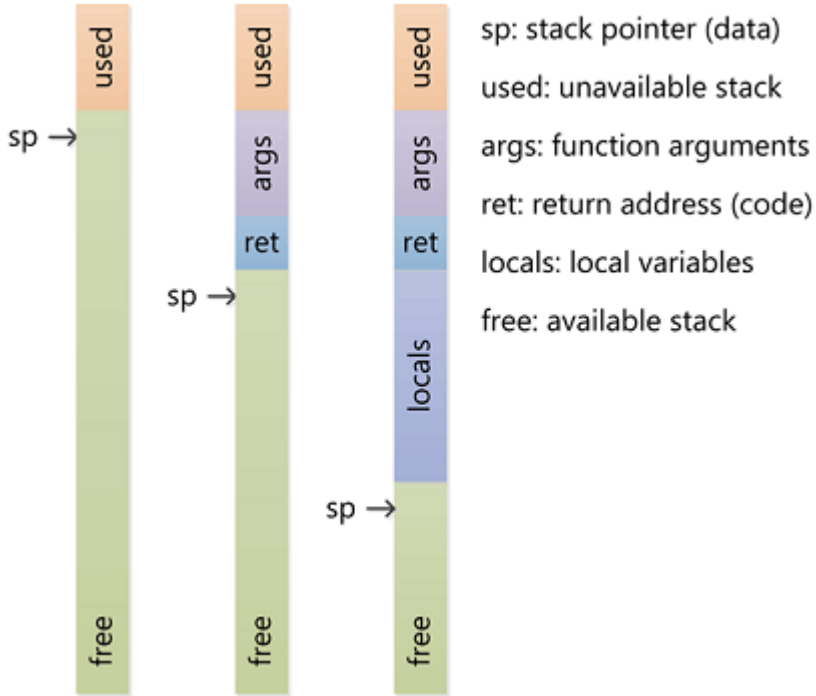
Stack & Heap

Есть Stack (стэк) и есть Heap (управляемая куча) (Ваш КЭП - Вы кстати должны знать эту тему лучше лектора)

- Обычно OS выделяет одну кучу на приложение (можно сделать несколько)
- На каждый поток (thread) OS создает свой выделенный стэк (в винде по-умолчанию 1Mb). И то и другое живет в RAM.
- Куча менеджерится CLR

Стэк намного быстрее из-за более простого управления хранением объектов, плюс cpu имеет регистры для работы со стеком и помещает частодоступные объекты из стека в кэш.

Стек представляет собой LastInFirstOutput очередь. Размер стека конечен, его нельзя расширить и в него нельзя пихать большие объекты. Примерная его работа понятна по картинке:



SOF explanation stack & heap

CLR сама решает, где хранить объекты в стеке или куче, у программиста нет прямой возможности управлять этим.

Это базовое отличие от C++.

Конечно, программист может с помощью выбора типов и того, как он их использует, влиять на то, как clr обращается с объектами, но все равно это получается достаточно ограничено.

Referenced VS Value types

Все объекты в C# делятся на два типа:

- Value types (значимые типы)
  - **могут** храниться в стеке, как локальные переменные.
- Referenced types (ссылочные типы)
  - всегда хранятся в куче, в стеке помещается указатель на объект в куче (поэтому и Reference)

Значимые типы, сохраненные в стеке, «легче» ссылочных:

- для них не нужно выделять память в управляемой куче
- их не затрагивает сборка мусора

Value Types

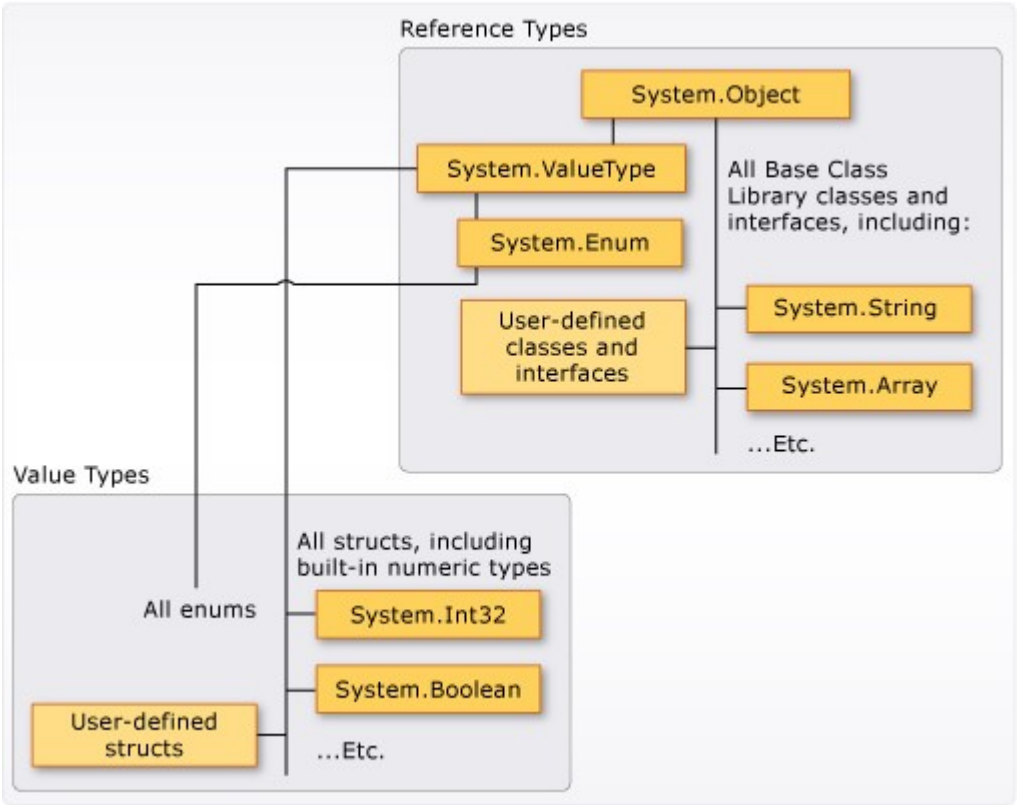
- **enum**
- **struct**
    - bool
    - byte / short / int / long
    - decimal
    - char
    - float / double

Reference Types

- object
- **class**
    - string

Все классы - ссылочные типы (в том числе всякие делегаты, интерфейсы, массивы и пр).
Все структуры и перечисления (enum) - значимые (базовые типы - это тоже структуры).

Разделение по классам:



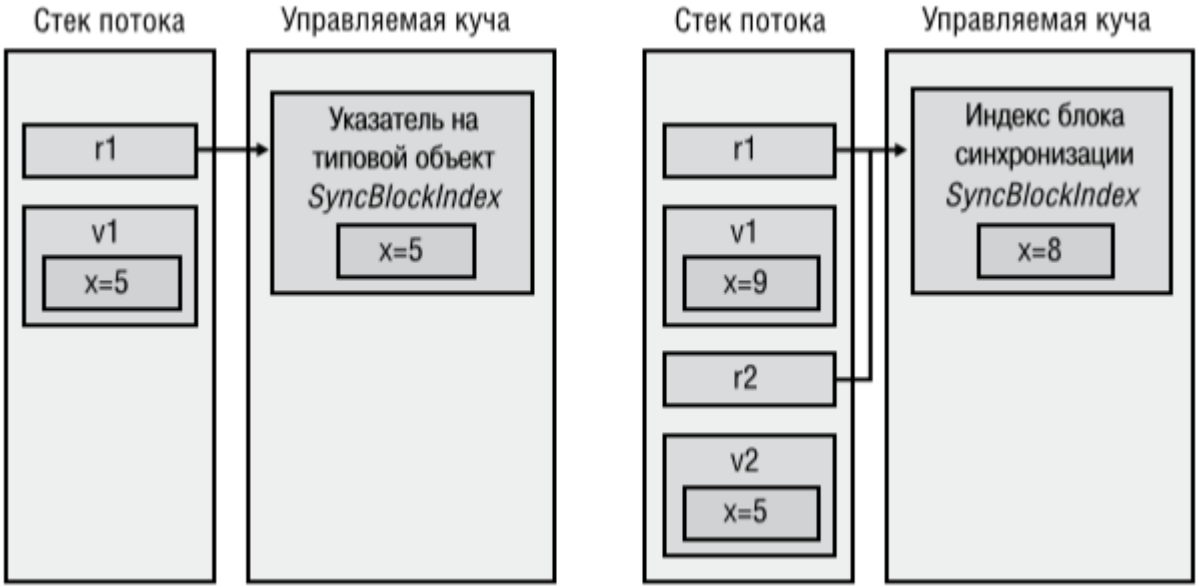
- System.ValueType
  - наследник oject, переопределяет его методы
  - базовый класс для всех значимых типов
  - нельзя создать его наследника напрямую
  - сам он является ссылочным, но все его реалзиации - значимые
- System.Enum - базовый тип для всех пользовательских перечислений

Рассмотрим на примере кода:

```
class SomeRef { public Int32 x; } // Ссылочный тип
struct SomeVal { public Int32 x; } // Значимый тип

static void ValueTypeDemo()
{
    SomeRef r1 = new SomeRef(); // Размещается в куче
    SomeVal v1 = new SomeVal(); // Размещается в стеке
    r1.x = 5; // Разыменовывание указателя, изменение в куче
    v1.x = 5; // Изменение в стеке

    SomeRef r2 = r1; // Копируется только ссылка (указатель)
    SomeVal v2 = v1; // Помещаем в стек и копируем члены
    r1.x = 8; // Изменяются r1.x и r2.x
    v1.x = 9; // Изменяется v1.x, но не v2.x
    Console.WriteLine($"{r1.x}, {r2.x}, {v1.x}, {v2.x} "); // "8,8,9,5"
}
```


Стек потока
Управляемая куча
r1
v1
x=5
Указатель на типовой объект SyncBlockIndex
x=5
Стек потока
Управляемая куча
r1
v1
x=9
r2
v2
x=5
Индекс блока синхронизации SyncBlockIndex
x=8

Почитать:

- Heap vs steak in C#
- Value Types stored, Eric Lippert

Передача параметров в методы

- Когда передается значимый тип: он копируется
- Когда передается ссылочный тип: копируется ссылка на объект в куче

```
public static void Main()
{
    var r = new RefType { X = 1 };
    var v = new ValueType { X = 1 };
    RefMethod(r);
    Console.WriteLine(r.X);
    ValueMethod(v);
    Console.WriteLine(v.X);
}

class RefType { public Int32 X; }
struct ValueType { public Int32 X; }

static void RefMethod(RefType r) { r.X = 2; }
static void ValueMethod(ValueType r) { r.X = 2; }
```

Есть 4 keywords для передачи параметров в методы:

- ref - параметр передается по ссылке
- out - параметр является "выходным" для метода
    - метод обязательно должен устанавливать значение для параметра
- in - параметр не изменяется в методе
- params - передает массив параметров

Нельзя использовать ref, in, and out keywords в:

- Async методах
- Методы с итераторами yield return / yield break

ref

- Параметр передается по ссылке
- Должен быть инициализирован до вызова

```
public static void Main()
{
    int v = 1;
    ValueMethod(ref v);
    Console.WriteLine(v); // 11
}

static void ValueMethod(ref int v)
{
    v = v + 10;
}
```

- Если мы передаем ссылочный тип, то ссылка не копируется!

```
public static void Main()
{
    var r = new RefType { X = 1 };
    RefMethod(r);
    Console.WriteLine(r.X);
    RefMethod(ref r);
    Console.WriteLine(r.X);
}

class RefType { public Int32 X; }
static void RefMethod(RefType r) { r = new RefType { X = 15 }; }
static void RefMethod(ref RefType r)
{
     r = new RefType { X = 15 };
}
```

- Можно возвращать результат метода по ссылке

```
public static void Main()
{
    var array = new int[5];
    ref int value = ref ElementAt(ref array, 3);
    value = 5;
    Console.WriteLine(array[3]);
}

public static ref T ElementAt<T>(ref T[] array, int position)
{
     if (array == null)
         throw new ArgumentNullException(nameof(array));
     if (position < 0 || position >= array.Length)
         throw new ArgumentOutOfRangeException(nameof(position));

     return ref array[position];
}
```

- Есть много дополнительных оптимизаций для ref + value types: `ref local`, `ref struct` type, `ref readonly struct` etc:
  - https://docs.microsoft.com/en-us/dotnet/csharp/reference-semantics-with-value-types
  - https://blogs.msdn.microsoft.com/mazhou/2017/12/12/c-7-series-part-7-ref-returns/
- Которые нужны для использования новых классов `Span<T>`, `Memory<T>`

out

- Значение тоже передается по ссылке, но может быть не инициализировано до вызова метода
- метод внутри себя обязательно должен присвоить это значение

```
public static void Main()
{
    string s = "33";

    if (Int32.TryParse(s, out int value))
        Console.WriteLine(value);
    else
        Console.WriteLine($"Unable to convert '{s}'");
}
```

in

- Тоже по ссылке, метод не может внутри себя присваивать такой параметр

params

- Указывает на переменное количество параметров в методе

```
public static void Main()
{
    WriteParams();
    WriteParams(1);
    WriteParams(3, 3, 4, 1, 10);
    WriteParams(new int[]{2,2,2});
}

public static void WriteParams(params int[] array)
{
    Console.WriteLine(string.Join(",", array));
}
```

System.Object

Все классы неявно наследуются от object (System.Object)
Общие методы:

- Public
  - ToString * - строковое представление экземпляра объекта, по дефолту `this.GetType().FullName()`
  - GetType - получить тип объекта
  - GetHashCode * - хэш-код для хранения в качестве ключа хэш-таблиц
  - Equals * - true, если объекты равны
- Protected
  - MemberwiseClone - создает новый экземпляр и присваивает все поля исходного объекта (без вложенных классов)
  - Finalize * - используется для очистки ресурсов, вызывается, когда сборщик мусора пометил объект для удаления, но до освобождения памяти

* - Методы, которые можно переопределить в своих классах

Primitive types

Integers

Type	Alias	Size	Explanation
System.Byte	**byte**	1 byte	unsigned `0` to `255`
System.SByte	sbyte	1 byte	signed `-128` до `127`
System.Int16	**short**	2 byte	signed `±32,767`
System.UInt16	ushort	2 byte	unsigned `0` до `65 535`
System.Int32	**int**	4 byte	signed `± 2 147 483 647`
System.UInt32	uint	4 byte	unsigned `0` до `4 294 967 295`
System.Int64	**long**	8 byte	signed `± 9 223 372 036 854 775 807`
System.UInt64	ulong	8 byte	unsigned `0` до `18 446 744 073 709 551 615`

Не рекомендуется использовать sbyte / uint / ushort / ulong

- Не CLS совместимые
- Многие стандартные методы возвращают обычные типы (получится дополнительная конвертация)
- Если не хватает размера, то увеличение в 2 раза не решает проблему

Короче, используйте int, long, short, byte.

Float numbers

Type	Alias	Size	Explanation	base	mantissa	exponent	precision digits
System.Single	float	4 byte	Single-precision floating-point ±3.4*10^38	2	23	8	7
System.Double	double	8 byte	Double-precision floating point ±1.7*10^308	2	52	11	15-16
System.Decimal	decimal	16 byte	decimal number ±7.9*10^28	10	96	5 (0-28)	28-29

decimal - десятичное число с плавающей запятой, это не примитивный тип и работает сильно медленее double (до 20 раз).

Основное различие можно понять на примере:

```
double a = 0.1;
double b = 0.2;
Console.WriteLine(a + b == 0.3); // false

decimal c = 0.1M;
decimal d = 0.2M;
Console.WriteLine(c + d == 0.3M); // true
```

`decimal` используется для валют и чисел, которые исконно "десятичные" (CAD, engineering, etc).

Не надо сравнивать double через `==`.
У double есть зарезервированные значения `double.NaN`, `double.Epsilon`, `double.Infinity`.

Other common types

Type	Alias	Size	Explanation
System.Boolean	bool	1 byte	true / false
System.Char	char	2 byte	Single unicode char
System.String	string	потом	Sequence of char
System.Object	object	потом	Base Type
System.Guid		16 byte	Unique identifier
System.DateTime		8 byte	Date and time

`bool` хоть и содержит информации на 1 бит хранится в байте.
При особом желании можно упаковать его для использования в массиве, скажем с помощью классов `BitVector32`, `BitArray`,
но заниматься подобными извращениями надо в исключительных ситуациях.

`Guid`, `DateTime` не являются примитивными.

Инициализация

```
<datatype> <variable name>;
<datatype> <variable name> = <value>;
```

```
int x;
System.Int32 x = new System.Int32(); // Эквивалент предыдущей строки, подробнее
про new позднее
int t = 0x1D;        // шестнадцатиричное представление 29

bool isValid = true;

double y = 3.0;      // По-дефолту число с точкой считается компилятором как
double
float f = 33.1f;     // Используем суфикс, чтобы студия не считала его double
decimal d = 11.1m;   // m для decimal
```

```
char c = 's';
string s = "Hello!"; // Разные кавычки

int z = x + 5;       // Сразу присваиваем
int i, j, k, l = 0;  // Много переменных сразу УИИИХУ, ниразу не видел такого в
реальном коде
```

Неявная типизация

Компилятор сам понимает, какой тип.

```
var x = 1;
var y = null; // Нельзя
```

Microsoft C# coding convensions var usage:

```
// Используйте неявную типизацию для локальных переменных, когда тип элементарно
понимается из правого выражения или не важен
var x = new MyClass();
var i = 3;
var list = new List<int>();
var db = new Data(_connection) { RetryPolicy = _retryPolicy };
```

```
// Не используете var, если тип не очевиден из правой части
var ExampleClass.ResultSoFar();
var ticketLifeTime = getTicketLifeTime(licenses);
var newCounters = mergeResult
    .Where(x => x.LicenseId == license.Id)
    .ToDictionary(x => x.Name, y => y.Value); // Дискуссионно, потому что итоговый
тип может быть громоздким и будет отвлекать внимание от основного кода

// Используйте в циклах
foreach (var element in myList)
{
}
```

Операторы

MSDN Операторы

Арифметические

- Бинарные
  - `+` - сложение `int x = 5 + 7;`
  - `-` - вычитание
  - `/` - деление. Надо иметь в виду, что деление двух целых чисел вернет результат округленный до целого числа
  - `*` - умножение
  - `%` - остаток от деления
- Унарные
  - `++` Инкремент
  - `--` Декремент

У инкремента, декремента выше приоритет, чем у операций умножения, сложения, остатка.

```
int x = 2;
int y = ++x; // префиксная форма
Console.WriteLine($"{y} - {x}"); // y=3; x=3

int a = 2;
int b = a++; // постфиксная
Console.WriteLine($"{b} - {a}"); // b=2; a=3
```

Поразрядные

Поразрядные операции над двоичной формой числа:

- `&` И
- `|` ИЛИ
- `^` исключающее ИЛИ / XOR
- `~` инверсия
- `x<<y` / `x>>y` сдвигает число `x` на `y` разрядов

Операторы с присваиванием

`+=`, `-=`, `^=`, и все остальные варианты

```
x = x + y;

x += y; // Записи эквиваленты
```

Логические операторы

Логические операторы возвращают bool

- `|`, `&` - логическое ИЛИ / И
- `||` / `&&` - оптимизированные операции ИЛИ / И
  второе условие вычисляется, если первое прошло проверку
- `!` - логическое отрицание
- `^` - исключающие ИЛИ

- `==` равенство `if (a == b)`
- `!=` неравенство

Всегда используйте операторы || и && вместо | и &.

Они и быстре и позволяют делать проверки, которые невозможны при одновременном вычислении обоих полей логического оператора

```
if ((myObj != null) && (myObj.A == 1)) { ... }
```

Ternary operator

Тернарный оператор `?:` по bool условию возвращает левое или правое значение. SOF Examples of usage

```
result = condition ? left : right
```

```
// Аналог
if (condition)
    result = left;
else
    result = right;

// Лучше всего использовать для присвоения / возврата простых значений
int result = Check() ? 1 : 0;

// Использование в качестве параметра метода
someMethod((sampleCondition) ? 3 : 1);
```

```
int ticketLifetime = licenses.Any()
    ? licenses.Select(x => x.TicketExpiration).Min()
    : ConstTicketMinutesLifetime;

// Можно делать вложенные, но не стоит увлекаться, делает код нечитаемым
int x = 1, y = 2;
string result = x > y
    ? "x > y"
    : x < y
        ? "x < y"
        : x == y
            ? "x = y"
            : "lul";
```

Null coalescing operator

Null-coalescing `??` оператор возвращает левый объект, если он не равен null, иначе возвращает правый.

`result = left ?? right;`

Можно складывать в цепочку.

```
int x = param1 ?? localDefault;
string anybody = getValue() ?? localDefault ?? globalDefault;
```

SOF Discussion

Null conditional operator

null-conditional operator `?.` проверяет на null до доступа к полю/свойству/индексу обьекта, и если объект null, то возвращает null, иначе возвращает член объекта.

Позволяет убрать некоторое количество проверок объектов на null / упростить использование тернарного оператора

```
int? length = customers?.Length; // null if customers is null

// Вместо примерно такого кода
int? length = customers == null ? (int?) null : customers.Length;
// или такого
if (customers == null)
    length = null;
else
    length = customers.Length;

Customer first = customers?[0];  // Доступ к индексу
int? count = customers?[0]?.Orders?.Count();  // Множественный сложный вариант
```

Контроль переполнения

По-умолчанию проверка переполнения **выключена**. Код выполняется быстрее.

Операторы `checked`/`unchecked`

```
byte a = 100;
byte b = checked((Byte) (a + 200));  // OverflowException
byte c = (Byte)checked(a + 200);   // b содержит 44, потому что сначала сложение конвертируется к Int32, потом проверяется, а потом кастуется к byte

checked
{
    // Начало проверяемого блока
    Byte d = 100;
    b = (Byte) (d + 200);
}
```

Decimal не примитивный тип. `checked / unchecked` для него не работают. Кидает `OverflowException`.

Рихтер рекомендует в процессе разработки ставить флаг компилятору `checked+`, чтобы проверка по-дефолту была включена всегда, программист уже руками расставляет `cheched / unchecked`, где нужно. А при релизе убрать этот флаг компилятора.

Приведение типов

CLR гарантирует безопасность типов.
Всегда можно получить `.GetType()`, который нельзя переопределить.
В C# разрешено неявное безопасное приведение к базовому типу:

```
int i = 1;
object a = i;
```

А так же расширяющие безопасные приведения базовых типов:

> byte > short > int > long > decimal
> int > double
> short > float > double

Для приведения к производному типу или в небезопасных нужно явное приведение:

```
int b = (int) a;
```

IS - проверяет совместимость с типом, возвращает bool, никогда не генерирует исключение.

```
object a = new object();
bool b = a is object; // true
bool b2 = a is int; // false
```

По факту CLR приходится 2 раза производить проверку типов при использовании `is`. Поэтому сделали:

AS - проверяет совместимость и если можно, то приводит к заданному типу и возвращает его. Иначе возвращает `null`

- Ипользуется только со ссылочными / Nullable типами (потому что иначе null не присвоить)
- По факту отличается от явного приведения только тем, что не генерит исключение

```
MyClass a = o as MyClass;
if (a != null)
{
    // Используем a
}
```

switch

- Обязательно должен использоваться `break`, `goto case`, `return` или `throw` при условии
- Нельзя просто так выполнить два условия сразу

```
int value = 1;
switch (value)
{
    case 1:
        Console.WriteLine("case 1");
        goto case 3; // переход к case 3
    case 2:
        Console.WriteLine("case 2");
        break;
    case 3:
        Console.WriteLine("case 3");
        break;
    default:
        Console.WriteLine("default");
        break;}
```

Pattern Matching

Patterns (C# 7.0):

- const
- type
- var - всегда успешен

```
public static void IsPattern(object o)
{
    if (o is null) Console.WriteLine("Const pattern");
    if (o is int i) Console.WriteLine($"Type, int = {i}");
    if (o is Person p) Console.WriteLine($"Type, person: {p.FirstName}");
    if (o is var x) Console.WriteLine($"var pattern, type {x?.GetType()?.Name}");
```

- В switch можно использовать любой тип
- В case можно использовать паттерны и дополнительные условия
- Порядок важен, но дефолт всегда выполнится последним независимо от места

```
switch(shape)
{
    case Circle c:
        WriteLine($"circle with radius {c.Radius}");
        break;
    case Rectangle s when (s.Length == s.Height):
        WriteLine($"{s.Length} x {s.Height} square");
        break;
    case Rectangle r:
        WriteLine($"{r.Length} x {r.Height} rectangle");
        break;
    default:
        WriteLine("<unknown shape>");
        break;
    case null:
        throw new ArgumentNullException(nameof(shape));}
```

Boxing / Unboxing

- При boxing мы создаем обертку над элементом в куче
- Очень дорогое занятие

```
int i = 123;
object o = i;  // boxes i

o = 123;
i = (int)o;  // unboxing
```

```
double e = 2.33333333;
int ee = (int)e;
Console.WriteLine(ee);

object o = (object) e;
int e2 = (int)o;
Console.WriteLine(e2);
```

```
double i = 3;
double i2 = i;
object o1 = i;
object o2 = i2;
object o3 = i;
Console.WriteLine(i == i2);
Console.WriteLine(o1 == o2);
Console.WriteLine(o1 == o3);
```