Towards the .NET Junior Developer

The extremely solid course

BIA Technologies

Яндекс Деньги

epam

OCS DISTRIBUTION

Aleksey Chirkin,
.NET Team Lead at OCS

Lesson 1

.NET Basics – Part I

Agenda

- **.NET history**
- **.NET basics**
- .NET type system
- Managed Heap
- Stack
- Reference types
- Value types
- Boxing/unboxing
- Type casting
- Garbage collection basics
- **Books of the day**
- **Links of the day**
- **Hometask**

.NET history

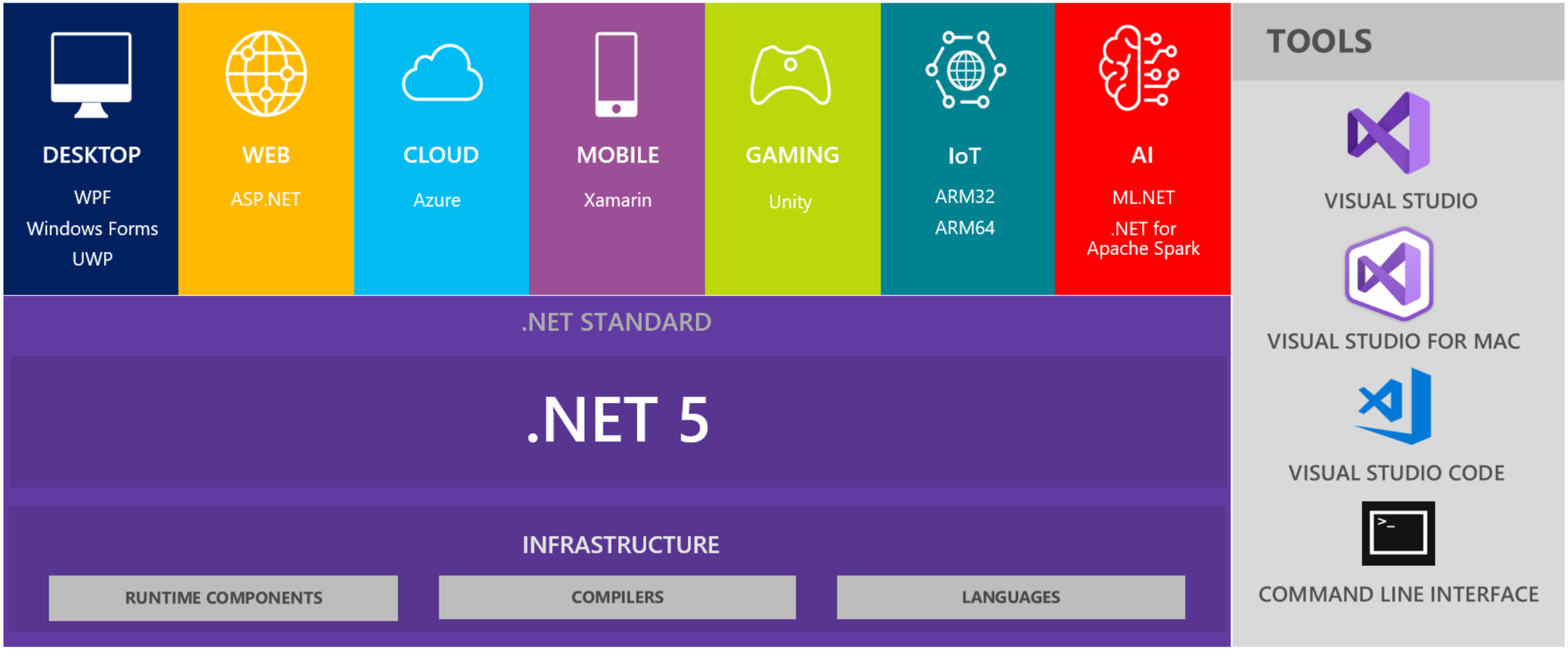
Brief overview

Introducing .NET

*“**.NET** (pronounced as "dot net"; previously named **.NET Core**) is a free and open-source, managed computer software framework for Windows, Linux, and macOS operating systems. It is a cross-platform successor to .NET Framework”*

Wikipedia

.NET – A unified platform



C/C++

1980's - 1990's

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout<<"Hello World";

    return 0;
}
```

```
main:
    PUSH %BP
    MOV  %SP, %BP
@main_body:
    SUB  %SP, $4, %SP
    MOV  $1, -4(%BP)
    PUSH $4
    CALL malloc
```

```
01001000 01100101
01101100 01101100
01101111 00100001
```

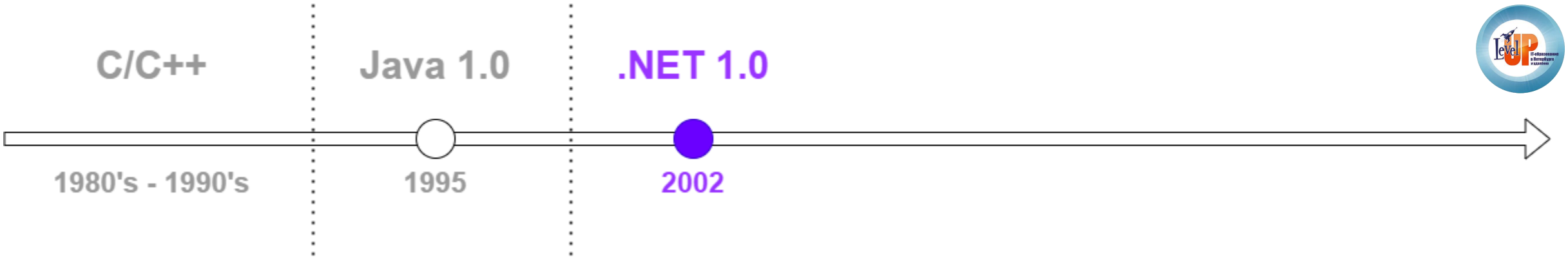
- Manual memory allocation/deallocation
- Compiles directly to binary code (optionally, to assembly code)


```java
public class Lesson1
{
    public void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

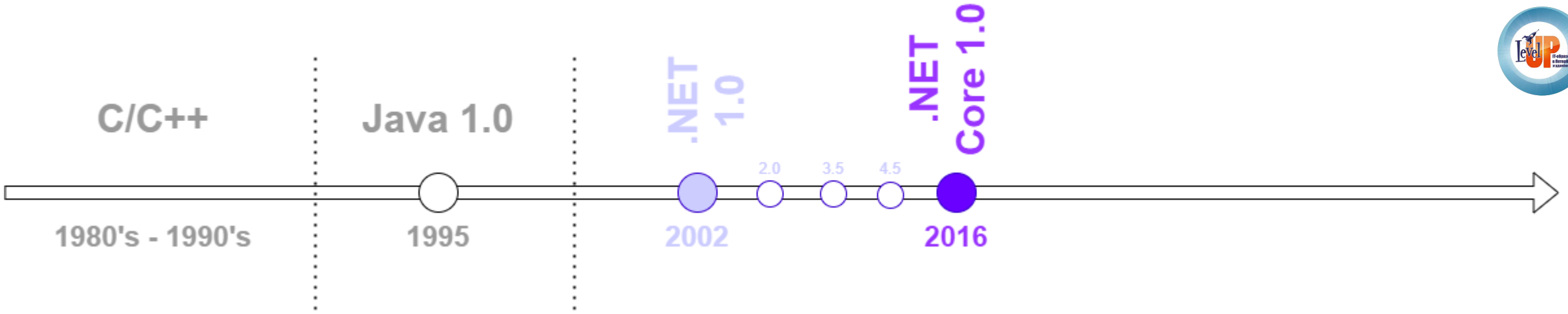
```
public main([Ljava/lang/String;)V
 L0
  LINENUMBER 4 L0
  GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
  LDC "Hello, World!"
  INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
 L1
  LINENUMBER 5 L1
  RETURN
 L2
  LOCALVARIABLE this LLesson1; L0 L2 0
  LOCALVARIABLE args [Ljava/lang/String; L0 L2 1
  MAXSTACK = 2
  MAXLOCALS = 2
```

01001000 01100101
01101100 01101100
01101111 00100001

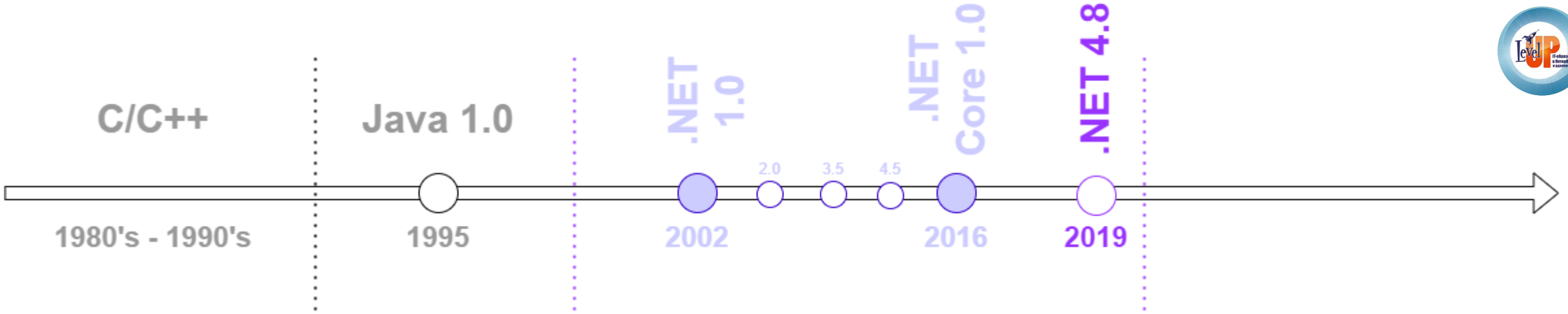
- Automatic memory management
- Compiles into intermediate representation (byte code)
- Runs on virtual machine (Java Virtual Machine - JVM)



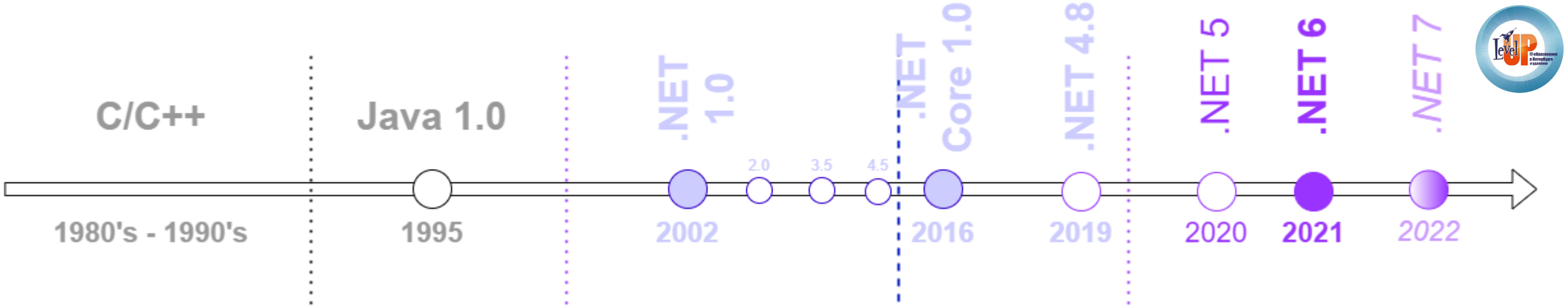
- Automatic memory management
- Compiles into IL (Intermediate Language) code
- Runs on virtual machine (Common Language Runtime – CLR)
- Windows only (98+)
- ASP.NET as a way to build web applications



- Cross-platform (Windows, Linux, MacOS)
- A lot of rewritten code in Base Class Library (BCL) and Framework Class Library (FCL)
- Open-source, community supported
- ASP.NET Core with the absolutely new philosophy of web applications creation



- Last versions of .NET Framework and ASP.NET



- .NET 5 finally removes the “duality” of .NET Framework / .NET Core
- New release cycle (per-year)

Other implementations of .NET

Mono (from 2004) – open source, cross-platform implementation

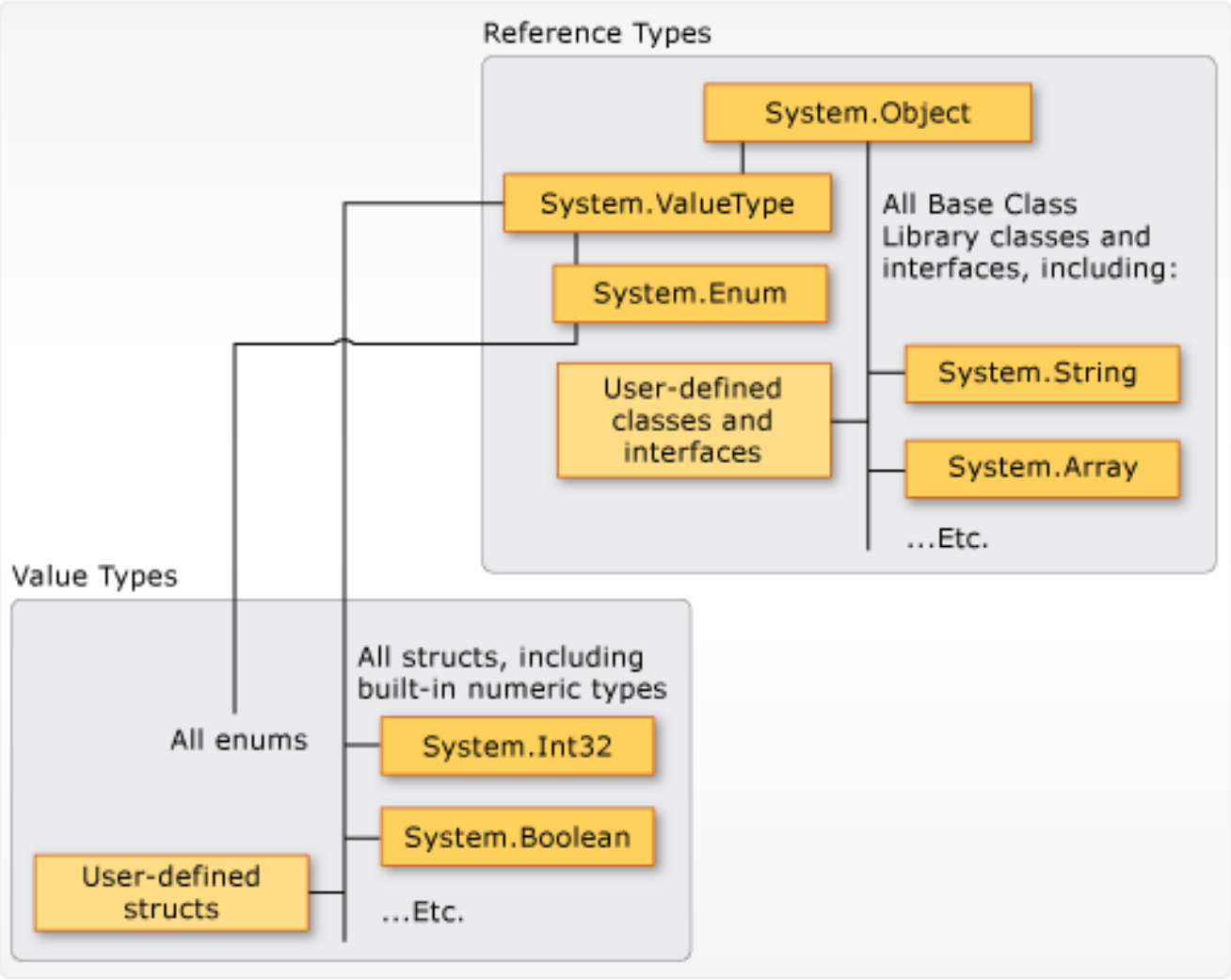
Silverlight (from 2007-2021) – platform for rich interface applications (RIA)

.NET Micro (from 2007) – platform for resource-constrained devices

.NET Basics

Core aspects

.NET type system



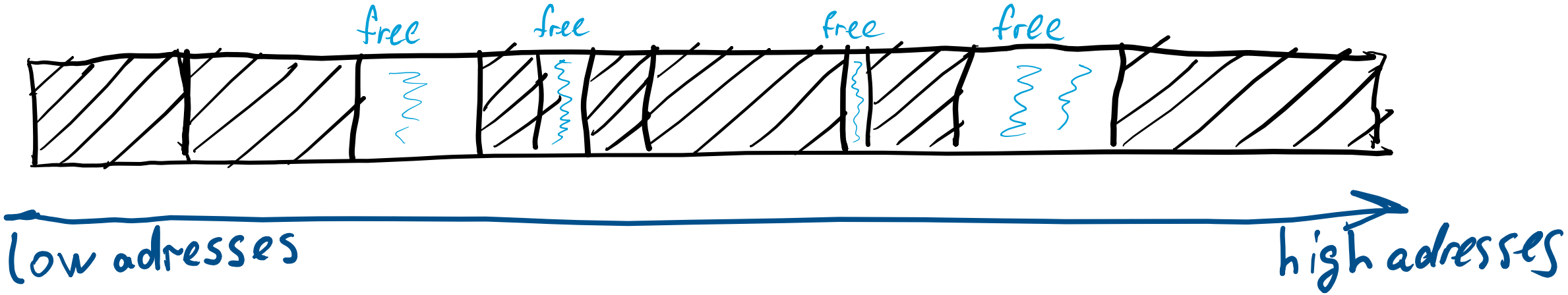
Reference types:

- classes
- interfaces
- delegates
- records

Value types:

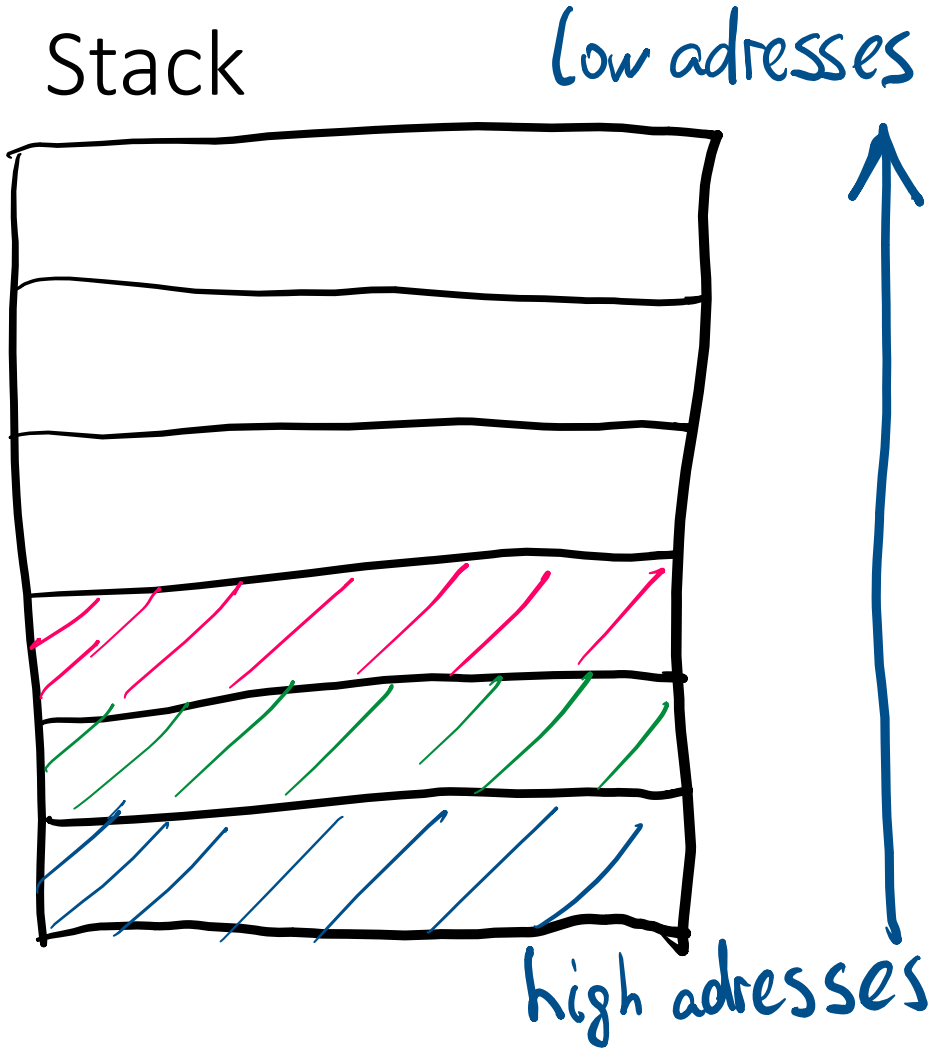
- structs
- enums
- record structs

Managed Heap



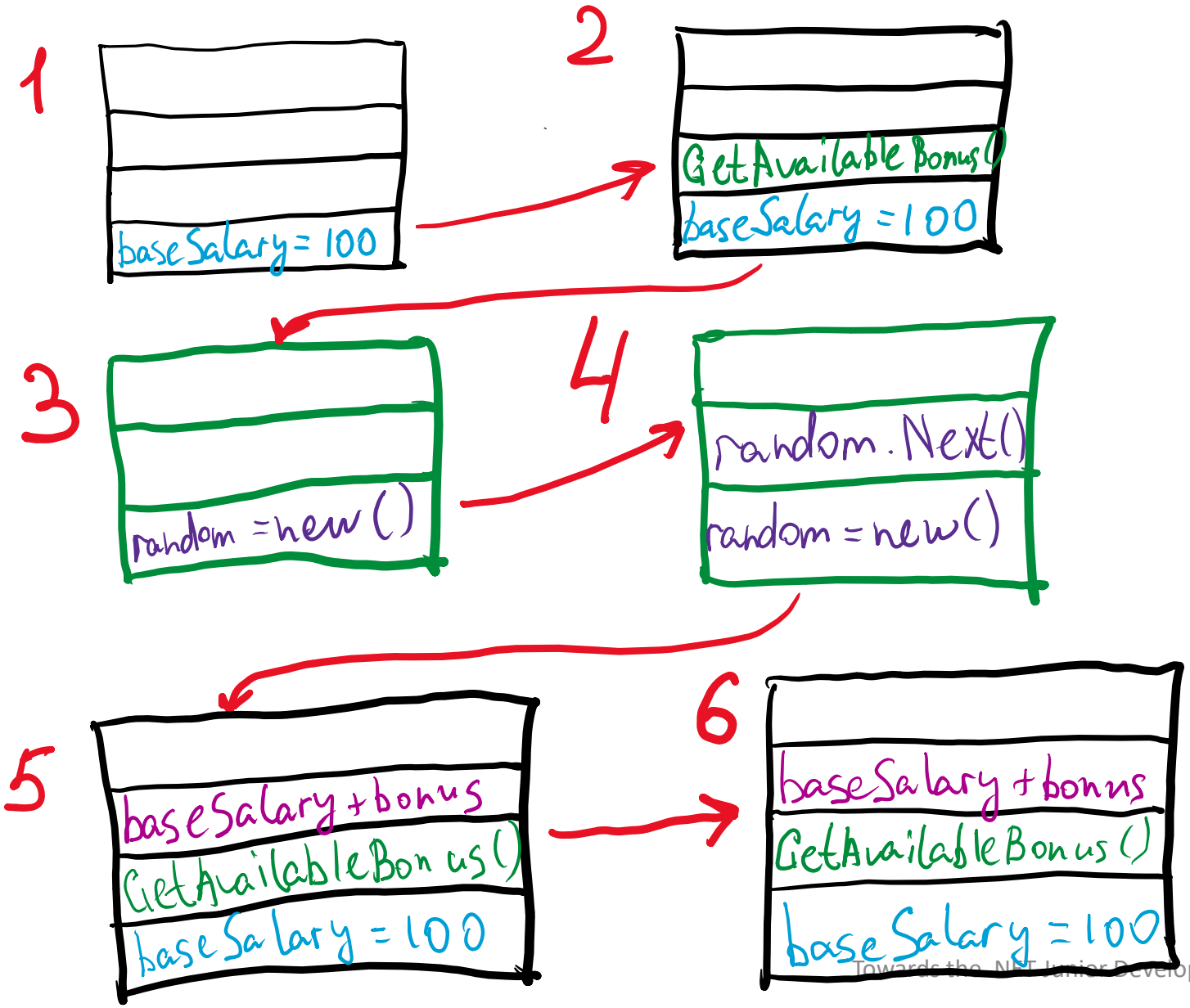
- area of the virtual memory space, grows from lower addresses to higher
- new objects can be placed into the free segments in the allocated memory
- new memory segments should be required from the OS via the *allocator* (part of the GC)
- can be used to store managed reference objects and wrapped value objects
- for the objects with size more than 85000 bytes there is a separate heap called Large Object Heap (LOH)
- there are several other heaps in the .NET memory system that service the runtime needs

Stack



- area of the virtual memory space, grows from higher addresses to lower
- new objects can be placed on top of the stack frame
- every function call creates new stack frame
- can be used to store value objects “in-place”
- stack has the maximum size (depends on the OS)
- stack can store the references on the objects from the heap

Stack




```
0 references
static void Main(string[] args)
{
    // CREATE stack frame

1   decimal baseSalary = 100;

    // CREATE child stack frame

2   var bonus = GetAvailableBonus();

    // CONTINUE with the top frame

5   var totalSalary = baseSalary + bonus;

6   Console.WriteLine(totalSalary);

    // DROP stack frame
}
```

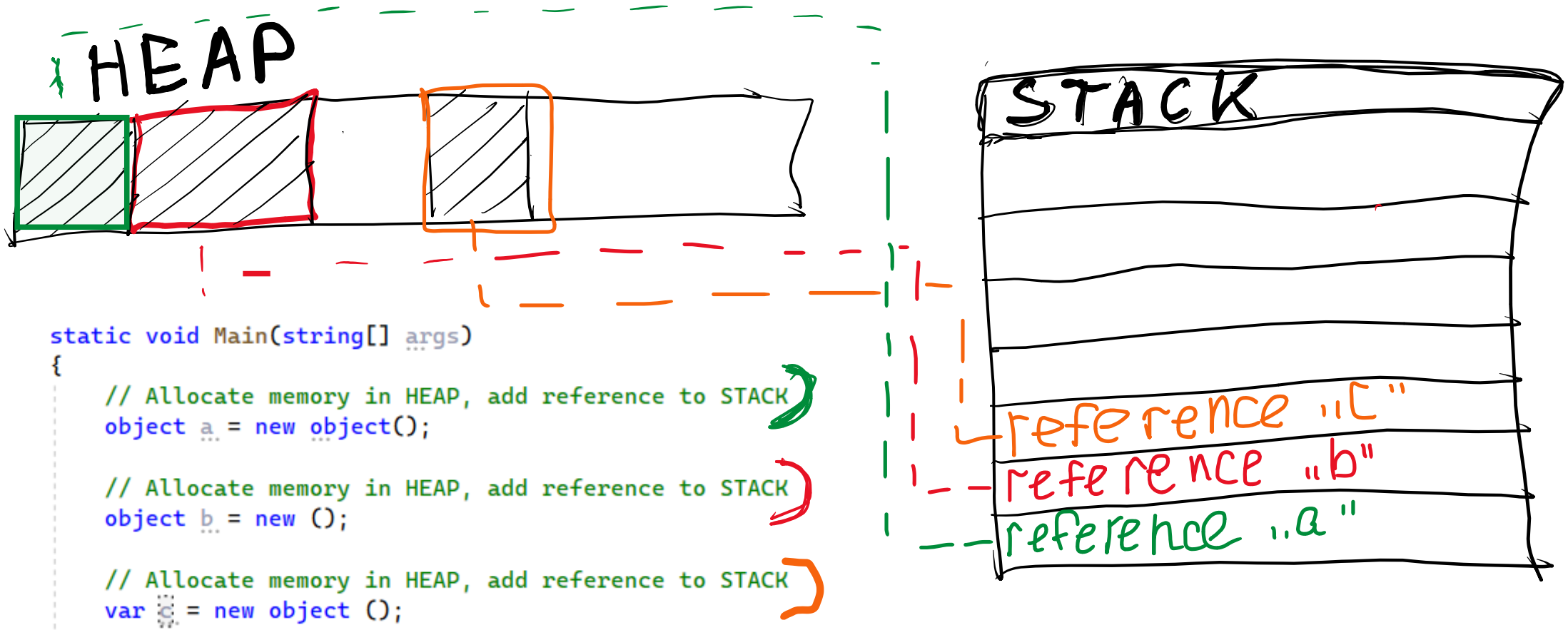
```
1 reference
static decimal GetAvailableBonus()
{
    // CREATE stack frame

3   var random = new Random();
4   return random.Next(50);

    // DROP stack frame
}
```

It’s coffee time!

Reference types

- live in managed HEAP (normally)
- used by references
- pass by reference (without copy)
- can participate in polymorphism
- can participate in inheritance
- managed by garbage collector (GC)

Reference types




```
static void Main(string[] args)
{
    // Allocate memory in HEAP, add reference to STACK
    object a = new object();

    // Allocate memory in HEAP, add reference to STACK
    object b = new ();

    // Allocate memory in HEAP, add reference to STACK
    var c = new object ();

    // Remove references from STACK
}
```

Classes and records

```
0 references
public class Engineer
{
    0 references
    public string Name { get; set; }

    1 reference
    public Position Position { get; set; }

    0 references
    public DateTime HireDate { get; set; }

    0 references
    public bool IsTemporaryContract() => false;

    0 references
    public bool IsManagementRole()
    {
        return Position.PositionType == PositionType.Management;
    }
}
```

```
1 reference
public record Position
{
    0 references
    public string Name { get; set; }

    0 references
    public string Description { get; set; }

    0 references
    public decimal Salary { get; set; }

    1 reference
    public PositionType PositionType { get; set; }
}

2 references
public enum PositionType
{
    Management = 0,
    Engineering = 1
}
```

Classes

```
4 references
public class Engineer
{
    private readonly IComparer<Engineer> _comparer;

    1 reference
    public string Name { get; set; }

    1 reference
    public Position? Position { get; set; }

    0 references
    public DateTime HireDate { get; set; }

    0 references
    public Engineer(string name, IComparer<Engineer> comparer)
    {
        Name = name;
        _comparer = comparer;
    }

    0 references
    public bool IsManagementRole()
    {
        return Position?.PositionType == PositionType.Management;
    }

    0 references
    public override bool Equals(object? other)
    {
        if (other is not Engineer otherEngineer) return false;
        return _comparer.Compare(this, otherEngineer) == 0;
    }

    0 references
    public override int GetHashCode()
    {
        throw new NotImplementedException();
    }
}
```

Records

```
2 references
public record Position
{
    1 reference
    public string Name { get; init; }

    0 references
    public string? Description { get; init; }

    0 references
    public decimal Salary { get; init; }

    1 reference
    public PositionType PositionType { get; init; }

    0 references
    public Position(string name)
    {
        Name = name;
    }
}
```

```
var manager = new Position("Head Of Department")
{
    Description = "Cool position. Really.",
    Salary = 100500 * 3,
    PositionType = PositionType.Management
};
```

Positional records and “with” blocks

```
1 reference
public record Position(string Name, string? Description, decimal Salary, PositionType PositionType);
```

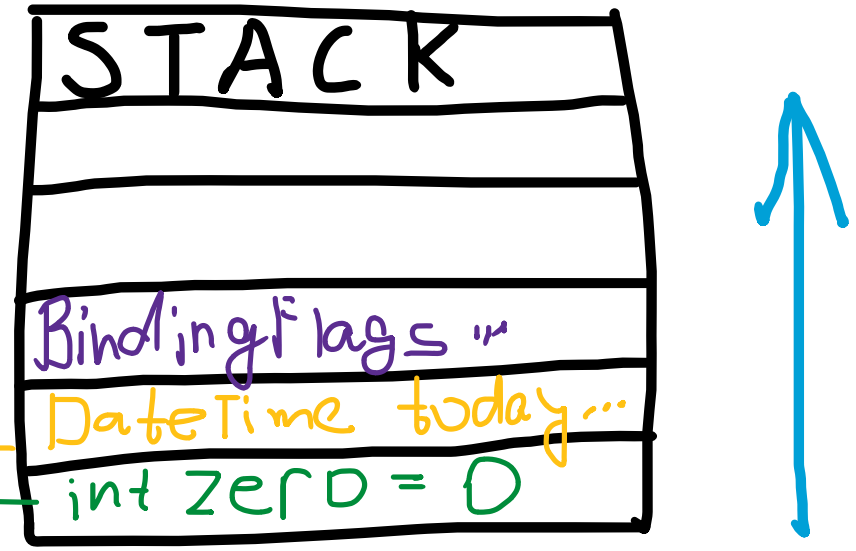
```
var junior = new Position("Junior .NET Developer", "Junior Engineering position", 1000, PositionType.Engineering);

var middle = junior with { Name = "Middle .NET Developer", Description = "Aah, doesn't matter", Salary = 2000 };

var senior = middle with { Name = "Senior .NET Developer", Description = "Just leave me alone", Salary = 100500 };
```

Value types

- live on STACK (excluding special cases, not guaranteed by CLS*)
- used directly (in-place data, without references, excluding special cases)
- pass by value (as a copy)
- can not participate in polymorphism
- can not participate in inheritance (excluding interfaces and System.ValueType)
- die with stack frame

Value types

```
static void Main(string[] args)
{
    // Allocate memory directly in STACK
    int zero = 0;

    // Allocate memory directly in STACK
    DateTime today = DateTime.Now;

    // Allocate memory directly in STACK
    var flag = BindingFlags.Public;

    // DROP stack frame
}
```

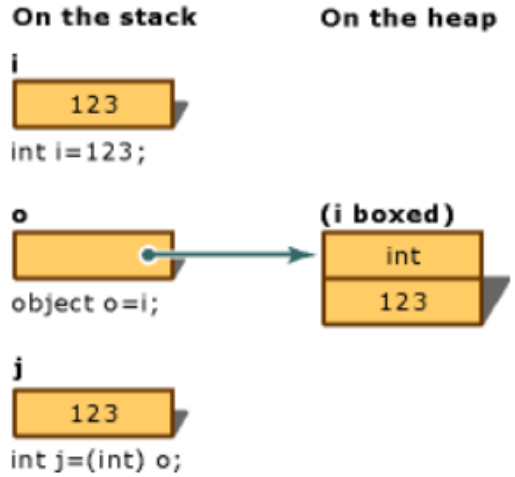


Value types – special cases

```
0 references
public class HeapCitizens
{
    // Field of the reference type always stores in HEAP
    private int _counter = 0;

    0 references
    public decimal GetPrice()
    {
        // This price will be stored on STACK
        decimal price = 150;

        // This one is BOXED. Hence it will be placed in HEAP
        object boxedPrice = price;

        // This price is UNBOXED and will be stored on STACK after the explicit type cast
        var unboxedPrice = (decimal)boxedPrice;

        return unboxedPrice;
    }

    0 references
    public int[] FillArray()
    {
        // Array is ref type. Array object with all items will be stored in HEAP
        return new int[5] { 1, 2, 3, 4, 5 };
    }
}
```

Boxing and unboxing

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```



Boxing and unboxing

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

Type casting – implicit conversions

```
int intVal = 15;
// Implicit conversion
long longVal = intVal;

// Compile-time error: couldn't convert "bigger" type to "smaller"
int intFromLongVal = longVal;

string text = "SomeText";
// Implicit conversion - all objects can be stored in 'object' variable
object wrapper = text;

// Compile-time error: there's no guarantee that object 'wrapper' is definitely string
string strFromObject = wrapper;
```

Type casting – explicit conversions

```
5 references
class Transport { }

4 references
class Car : Transport { }

3 references
class Bus : Transport { }
```

```
var cityBus = new Bus();
// Implicit conversion to base type is OK
Transport transport = cityBus;

// Conversion to derived type requires explicit casting
var car = new Car();
Transport carTransport = car;
Car sportCar = (Car)carTransport;

// But what if we're trying to create Bus from Car?
var oneMoreCar = new Car();
Transport oneMoreCarTransport = oneMoreCar;
Bus bus = (Bus)oneMoreCarTransport;
```

Type casting – explicit conversions

```
// Conversion to derived type requir
var car = new Car();
Transport carTransport = car;
Car sportCar = (Car)carTransport;

// But what if we're trying to creat
var oneMoreCar = new Car();
Transport oneMoreCarTransport = oneMor Car;
Bus bus = (Bus)oneMoreCarTransport;
```

Exception Unhandled

System.InvalidCastException: 'Unable to cast object of type 'Car' to type 'Bus'.'

View Details | Copy Details | Start Live Share session...

▷ Exception Settings

It’s coffee time!

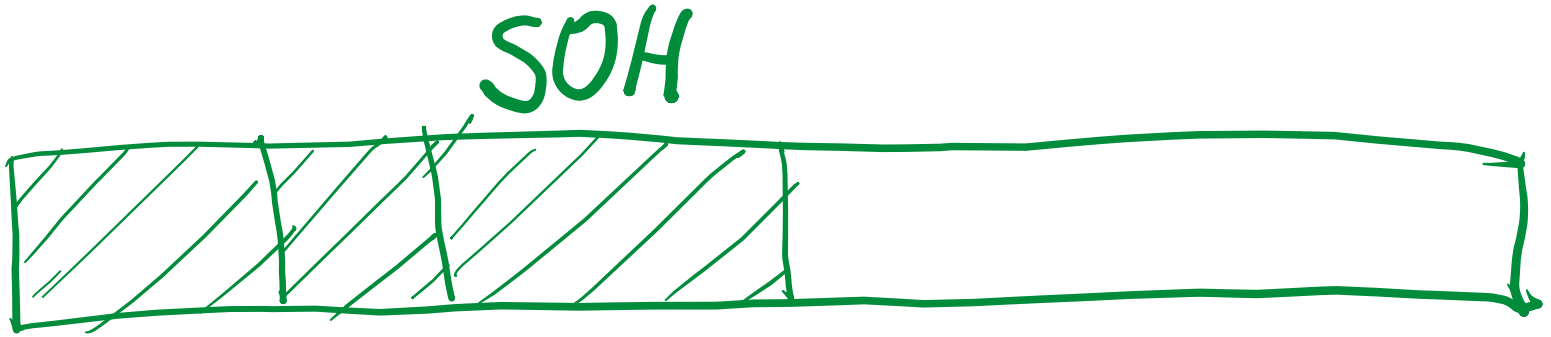
Garbage collection basics

- memory management in .NET includes *allocation* and *collection*
- both processes are provided by Common Language Runtime (CLR)
- the main way to allocate memory – *new ()* operator
- “garbage” is a general term for the objects in memory that can be safely deleted
- Garbage Collector (GC) collects only reference type objects in HEAP

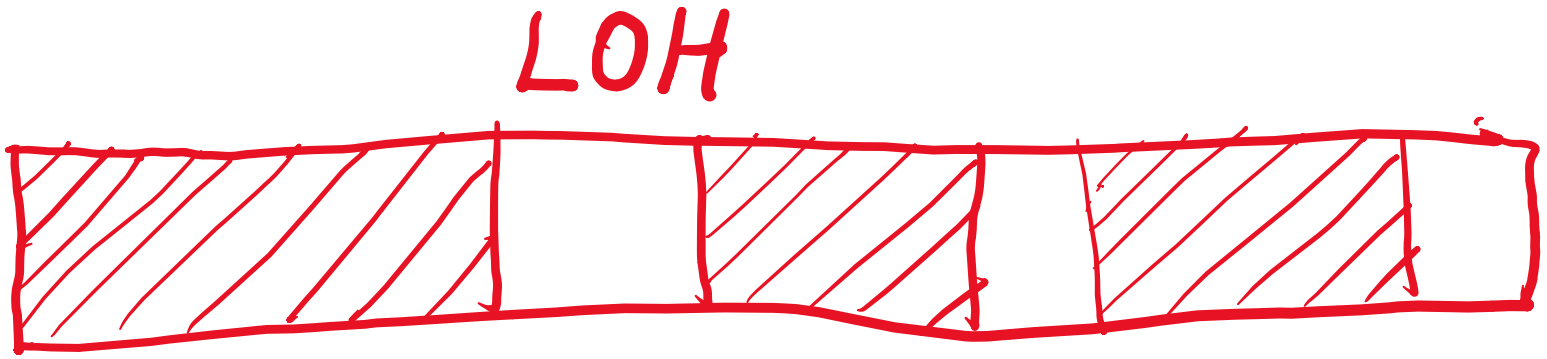
Garbage collection basics

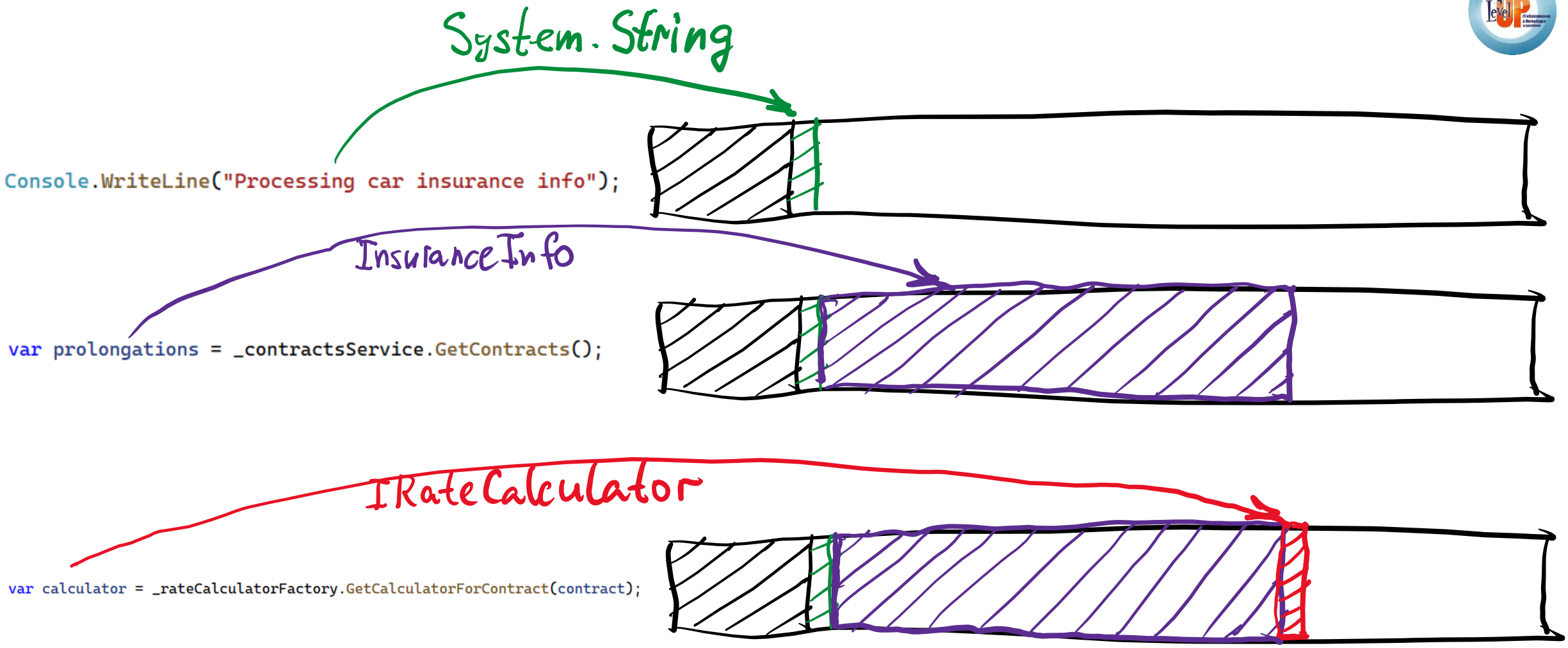
```
public void UpdateContracts()
{
    // Strings allocate on HEAP as usual reference objects
    Console.WriteLine("Processing car insurance info");

    // Here we can have a lot of InsuranceInfo class instances. All of them will live on HEAP
    var prolongations = _contractsService.GetContracts();

    foreach (var contract in prolongations)
    {
        ProlongateContract(contract, _prolongationPeriod);
    }

    _contractsService.UpdateContracts(prolongations);

    // String allocation again
    Console.WriteLine("Done.");
}

public void ProlongateContract(InsuranceInfo contract, TimeSpan prolongationPeriod)
{
    // Factory creates instance of the calculator for each contract.
    var calculator = _rateCalculatorFactory.GetCalculatorForContract(contract);
    var updatedRate = calculator.Calculate();

    contract.Rate = updatedRate;
    contract.EndDate = contract.EndDate.Add(prolongationPeriod);
}
```

Small Objects Heap
< 85000 bytes

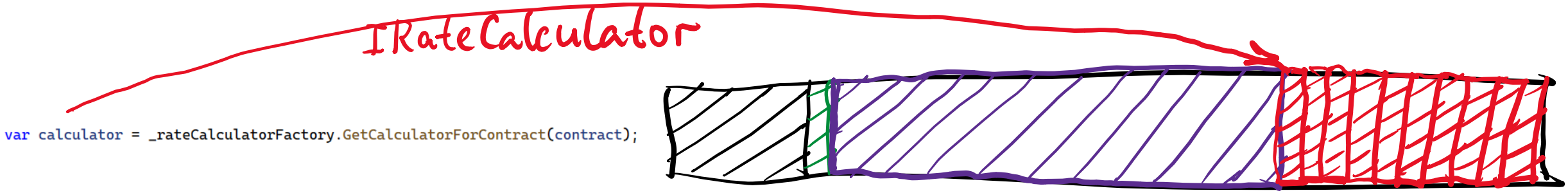


Large Object Heap
≥ 85000 bytes




System.String
Console.WriteLine("Processing car insurance info");
InsuranceInfo
var prolongations = _contractsService.GetContracts();
IRateCalculator
var calculator = _rateCalculatorFactory.GetCalculatorForContract(contract);

```
foreach (var contract in prolongations)
{
    ProlongateContract(contract, _prolongationPeriod);
}
```




```
public void ProlongateContract(InsuranceInfo contract, TimeSpan prolongationPeriod)
{
    // Factory creates instance of the calculator for each contract.
    var calculator = _rateCalculatorFactory.GetCalculatorForContract(contract);
    var updatedRate = calculator.Calculate();

    contract.Rate = updatedRate;
    contract.EndDate = contract.EndDate.Add(prolongationPeriod);
}
```

← START OF LIFE

← END OF LIFE

Fig. 1 – SOH Allocations And GCs

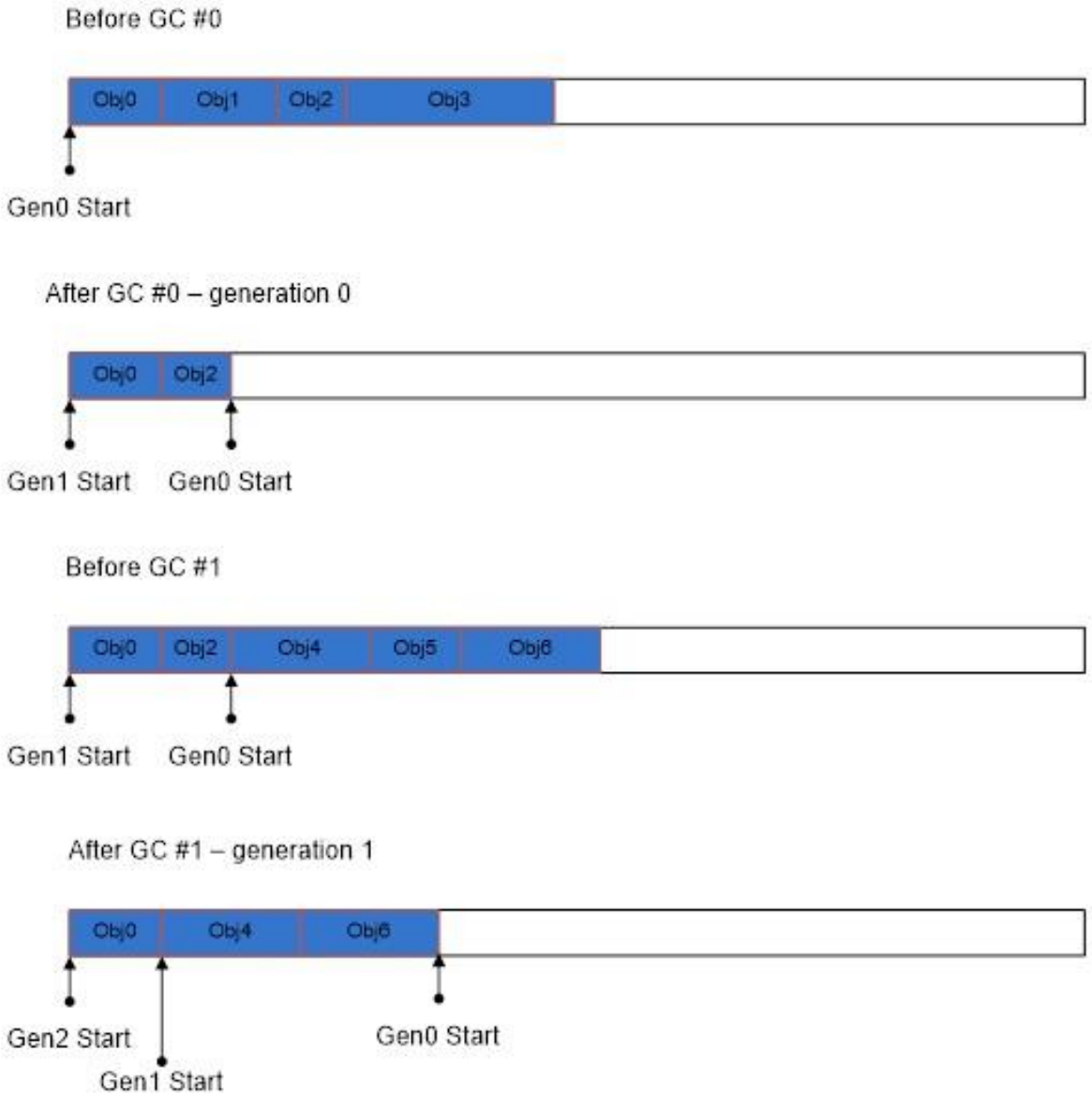
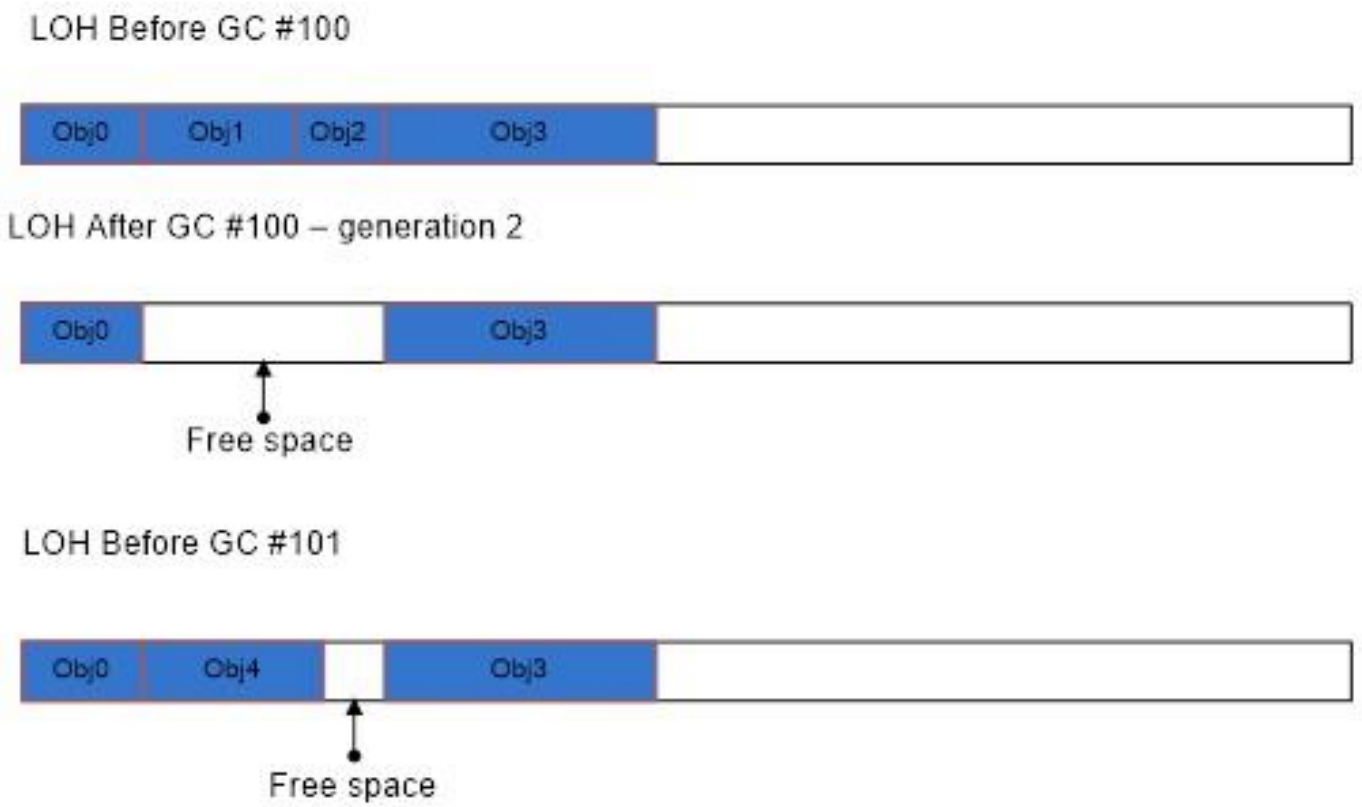
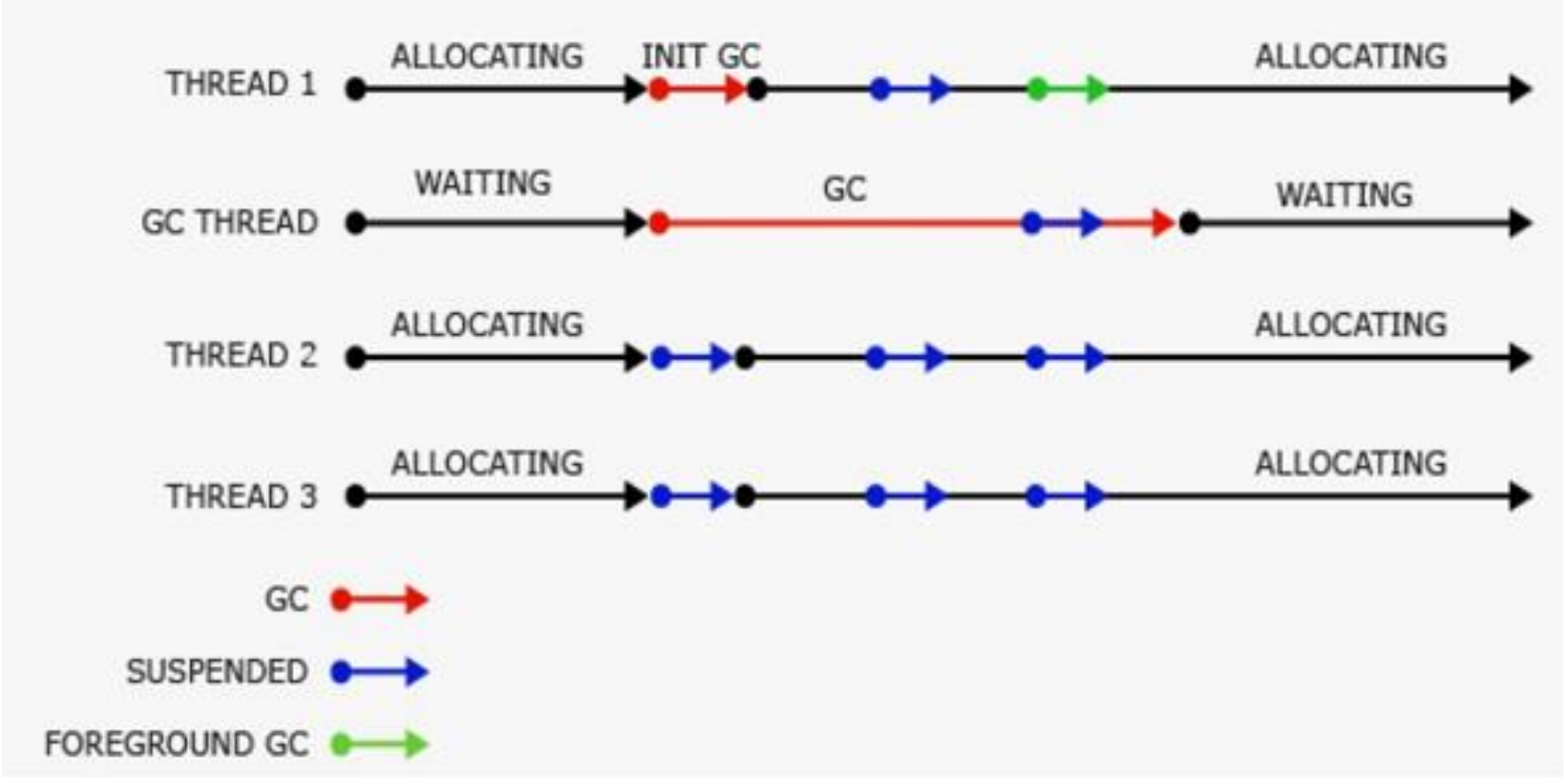


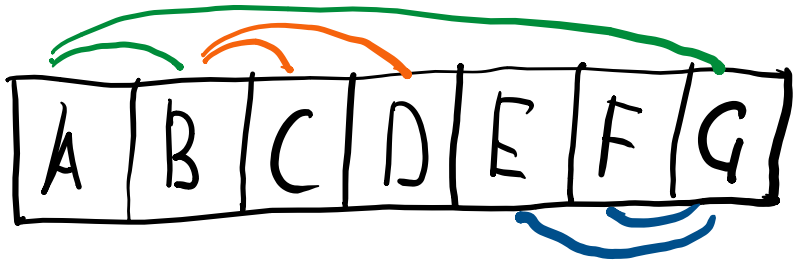
Fig. 2 – LOH Allocations And GCs

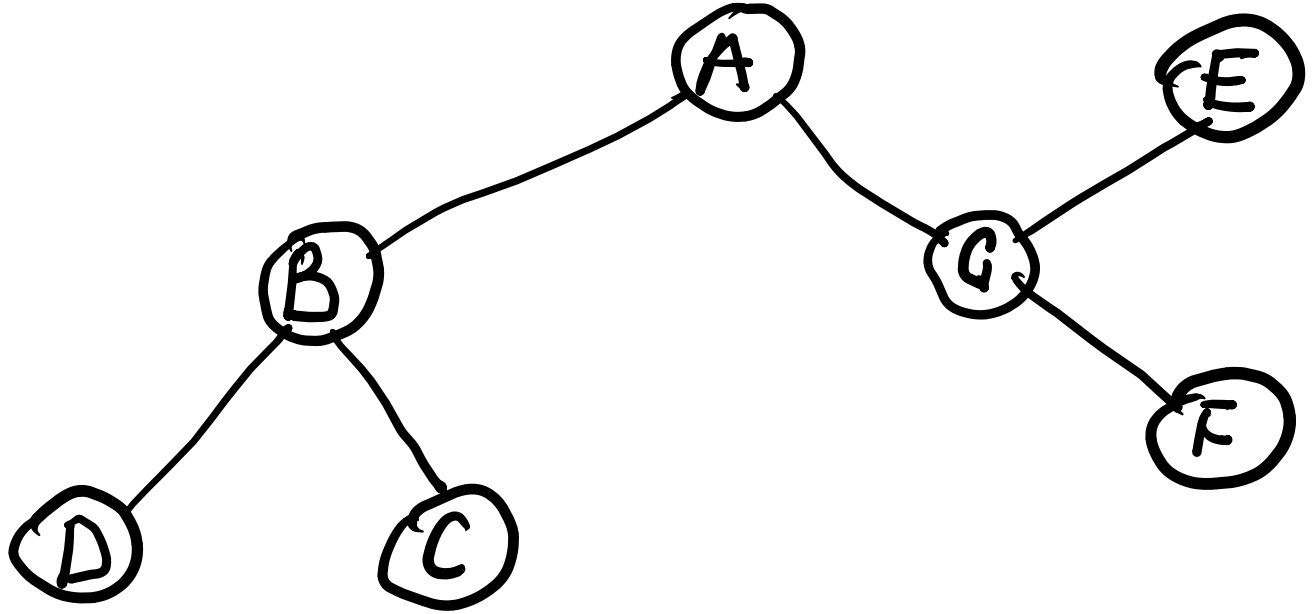



THREAD 1
ALLOCATING
INIT GC
ALLOCATING
GC THREAD
WAITING
GC
WAITING
THREAD 2
ALLOCATING
ALLOCATING
THREAD 3
ALLOCATING
ALLOCATING
GC
SUSPENDED
FOREGROUND GC

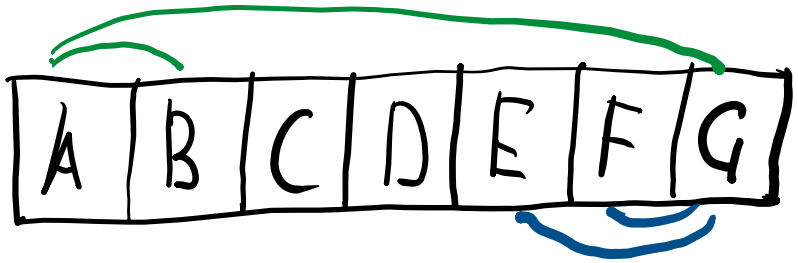
Garbage collection stages

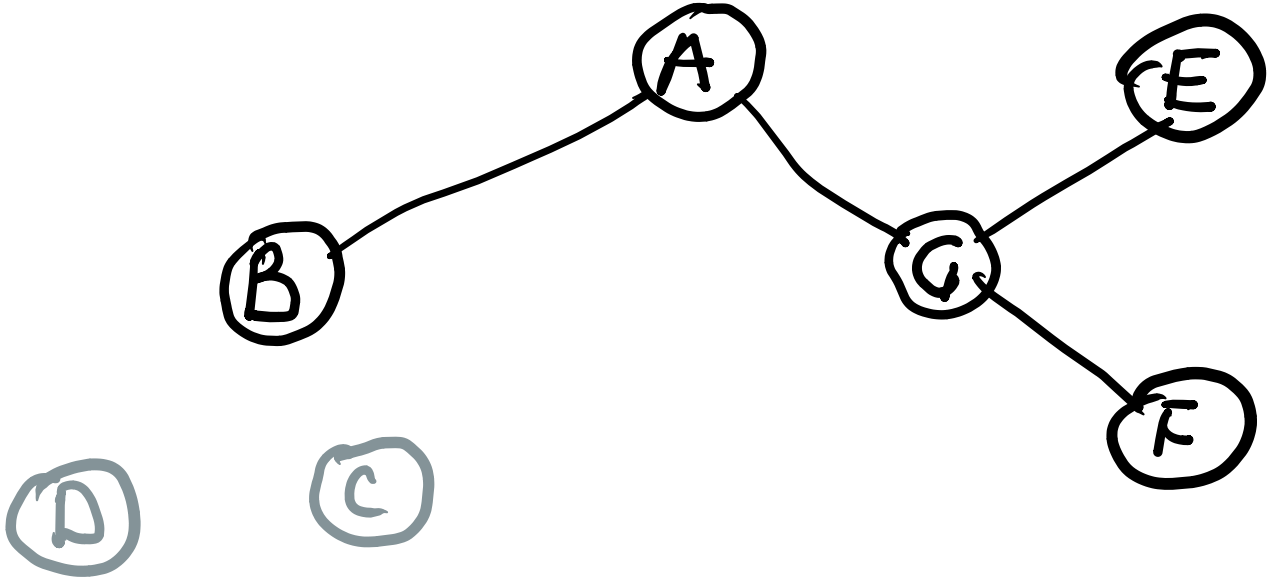
Input





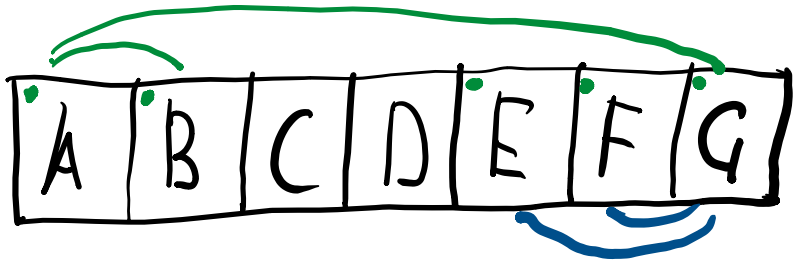
Garbage collection stages

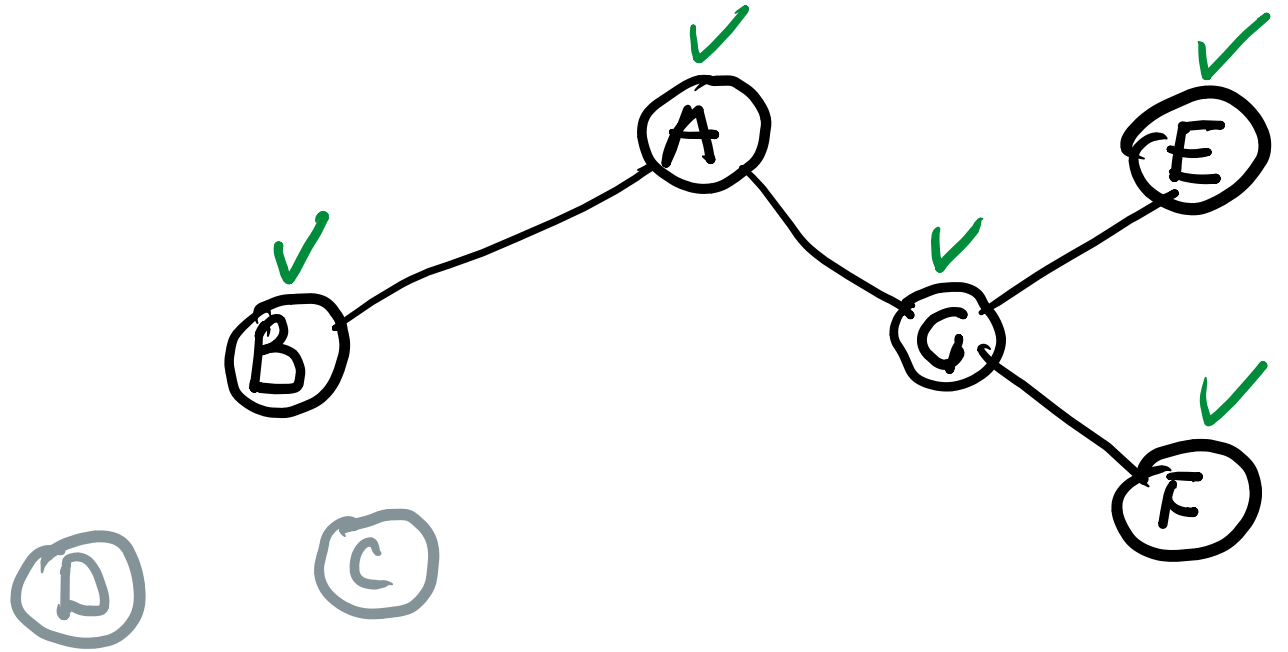




Garbage collection stages

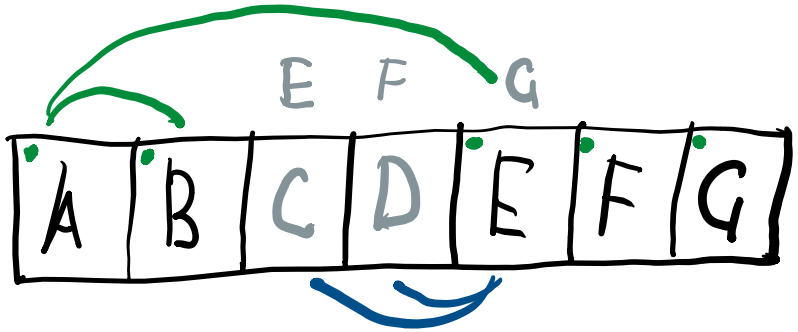
I. Detect/Mark

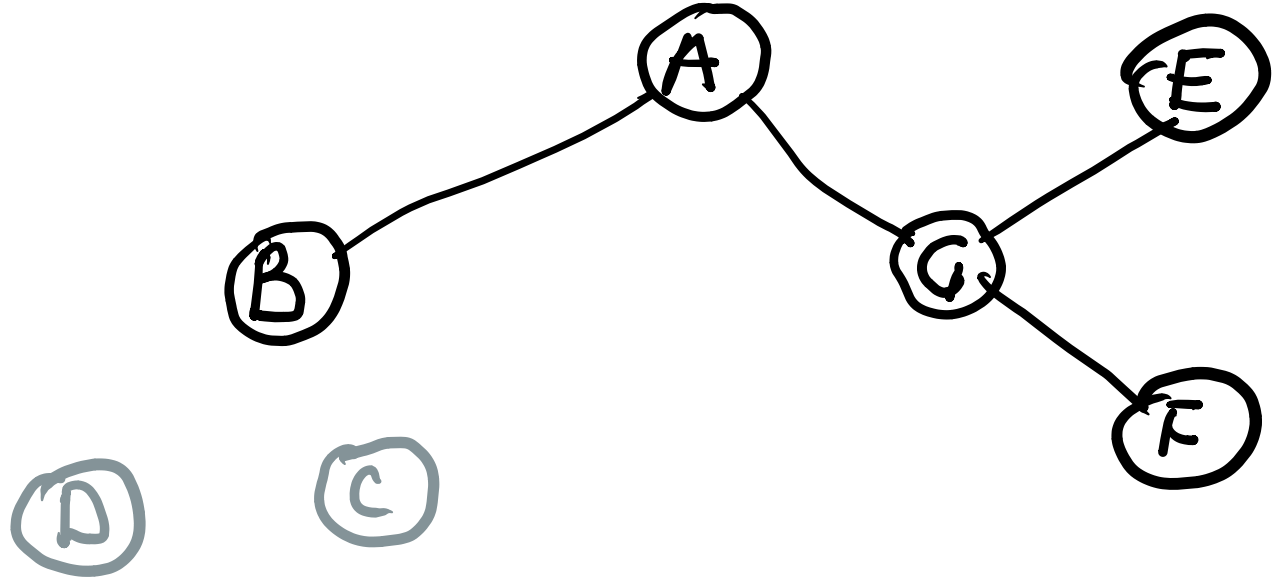




Garbage collection stages

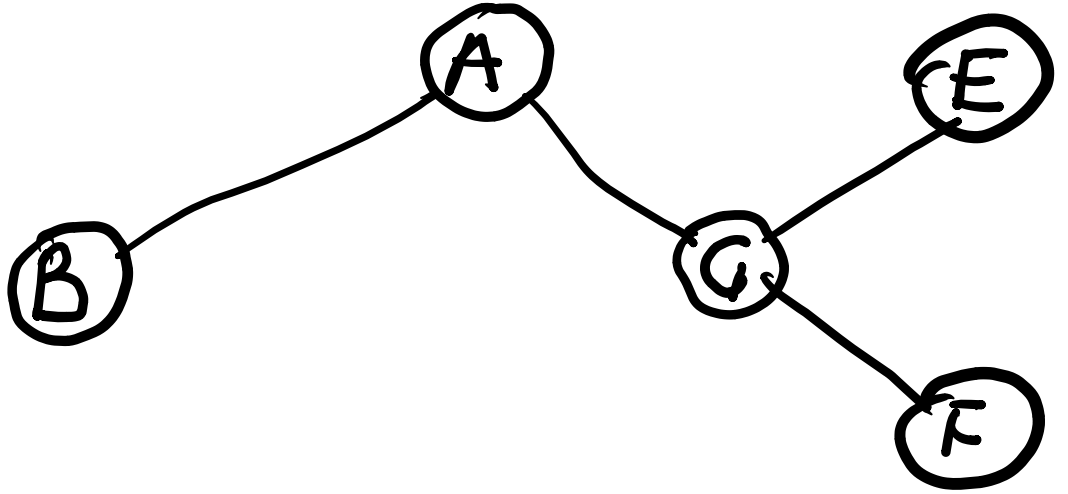
II. Relocating





Garbage collection stages

III. Compacting



Books of the day

Microsoft Docs - Introduction to C# - interactive tutorials

Andrew Troelsen – Pro C# 10 with .NET 6: Foundational Principles and Practices in Programming

Mark J. Price - C# 10 and .NET 6 - Modern Cross-Platform Development - Sixth Edition. Build apps, websites, and services with ASP.NET Core 6, Blazor, and EF Core 6 using Visual Studio 2022 and Visual Studio Code

Links of the day

[.NET reference types vs value types advanced comparison]

[Records vs structs]

[Stack is an implementation detail]

[Boxing and unboxing]

[Fundamentals of garbage collection]

Hometask

- Prepare the short speech about the .NET type system. Tell a few sentences about classes, records, structs – what is the difference between them? Can we build a memory management system based only on a heap? Only on a stack?
- Search for the information about is/as operators on MSDN. How can these operators help us with the type checking/casting?
- Read about types of GC and garbage collection modes. What's the difference between "workstation" and "server" garbage collection? What are the main benefits and drawbacks of the concurrent GC?

That’s all for this time!