

# Курс “Языки программирования”

## Лекция 4.

### Объявление и вызов методов в C#

## Определение и вызов методов

Знакомимся с созданием и вызовом методов, в том числе перегруженных и принимающих переменное число параметров, а также инструментами рефакторинга, предоставленными Microsoft Visual Studio

## Что такое метод?

Методы это имплементация поведения типа

Метод содержит блок кода, определяющий действия, которые может выполнять тип

Весь код принадлежит методу

Методы позволяют инкапсулировать операции, которые защищают данные, хранимые внутри типа

Методы могут быть предназначены для внутреннего использования типа и быть закрытыми для других типов

Другие методы могут разрабатываться, чтобы позволить другим типам запрашивать выполнение определенного действия объекта, эти методы являются открытыми

# Что такое метод?

C# поддерживает два типа методов

Экземплярные методы (instance methods)

```
int count = 99;  
string strCount = count.ToString();
```

Имя метода

Имя объекта

Статические методы (static methods)

```
string strCount = "99";  
count = Convert.ToInt32(strCount);
```

Имя типа

Имя метода

## Создание метода

Метод состоит из двух частей:

Спецификация метода (имя, параметры, возвращаемый тип и уровень доступа)

Тело метода (код)

### Сигнатура метода

1

Имя

2

Список  
параметров

Каждый метод в классе должен иметь уникальную сигнатуру

## Создание метода

### Именованние методов

Использовать при именовании глаголы или фразы глаголов. Это помогает другим разработчикам понять назначение кода

Использовать при именовании стиль «Pascal case». Не начинать имена методов с символа подчеркивания или строчной буквы

### Тело метода

Блок кода `C#`, который реализуется с использованием любой из имеющихся в `C#` программных конструкций

# Создание метода

## Спецификация параметров

Локальные переменными, которые создаются при работе метода и заполняются значениями, указанными при вызове метода

Параметры именуются в соответствии со стилем «Camel case»

## Спецификация типа возвращаемого значения

Все методы должны иметь тип возвращаемого значения

```
string MyMethod()  
{  
    return "Hello";  
}
```

## Вызов метода

Для вызова метода необходимо:

- ✓ указать имя метода
- ✓ предоставить в скобках аргументы, соответствующие параметрам метода
- ✓ если метод возвращает значение, необходимо указать, как использовать это значение

```
public bool LockReport(string reportName, string userName)
{
    bool success = false;
    // Perform some processing here.
    return success;
}
```

```
bool isReportLocked = LockReport("Medical Report", "Don Hall");
```

Аргументы метода вычисляются в строгом порядке слева-направо

```
int Sum(int first, int second)
{
    return first + second;
}
```

```
int i = 1;
int j = 2;
int result = Sum(i++, i+j);
```

result = 5



## Создание и вызов перегруженных методов

```
int intData = 99;
bool booleanData = true;
//...
Console.WriteLine(intData);
Console.WriteLine(booleanData);
```

```
public void Deposit(decimal amount)
{
    _balance += amount;
}
public void Deposit(string amount)
{
    _balance += decimal.Parse(amount);
}
public void Deposit(int dollars, int cents)
{
    _balance += dollars + (cents / 100.0m);
}
```

### Перегруженные методы

- ✓ имеют одинаковое имя
- ✓ имеют уникальную сигнатуру
- ✓ имеют одну семантику

При вызове метода компилятор определяет версию метода, который должен быть вызван, анализируя количество и типы аргументов, указанных при вызове

## Использование массива параметров

Не всегда возможна перегрузка метода, принимающего переменное число параметров, особенно если не существует теоретических ограничений на их количество

```
int Add(int one, int two)
{
    return one + two;
}
```

```
int Add(int one, int two, int three)
{
    return one + two + three;
}
```

```
int Add(int one, int two, int three, int four)
{
    return one + two + three + four;
}
```

```
int Add(. . .)
{
    return one + two + three + four + . . .;
}
```



## Использование массива параметров

```
int Add(int[] data)
{
    int sum = 0;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    return sum;
}
```

```
int[] myData = new int[4];
myData[0] = 99;
myData[1] = 2;
myData[2] = 55;
myData[3] = -26;
//...
int sum = myObject.Add(myData);
```



Необходимо вручную объявлять и  
заполнять массив данных

## Использование массива параметров

Ключевое слово `params` определяет массив параметров

```
int Add(params int[] data)
{
    int sum = 0;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    return sum;
}
...
int sum = myObject.Add(99, 2, 55, -26);
```

При определении метода с массивом параметров компилятор C# автоматически генерирует код, который создает массив из набора аргументов, указываемых при вызове метода

Если существует перегрузка метода, соответствующая указанному типу и количеству параметров, она будет вызываться предпочтительнее, чем версия метода, принимающего массив параметров

## Рефакторинг для извлечения метода

Рефакторинг – это процесс улучшения написанного ранее кода путем изменения его внутренней структуры, не влияющей на внешнее поведение кода

Для осуществления рефакторинга существующего кода в метод необходимо выполнить следующие действия

1

В Visual Studio в окне редактора кода следует выбрать код, который необходимо реорганизовать в метод, щелкнуть правой кнопкой мыши пункт Refactor, а затем нажать кнопку Extract Method

2

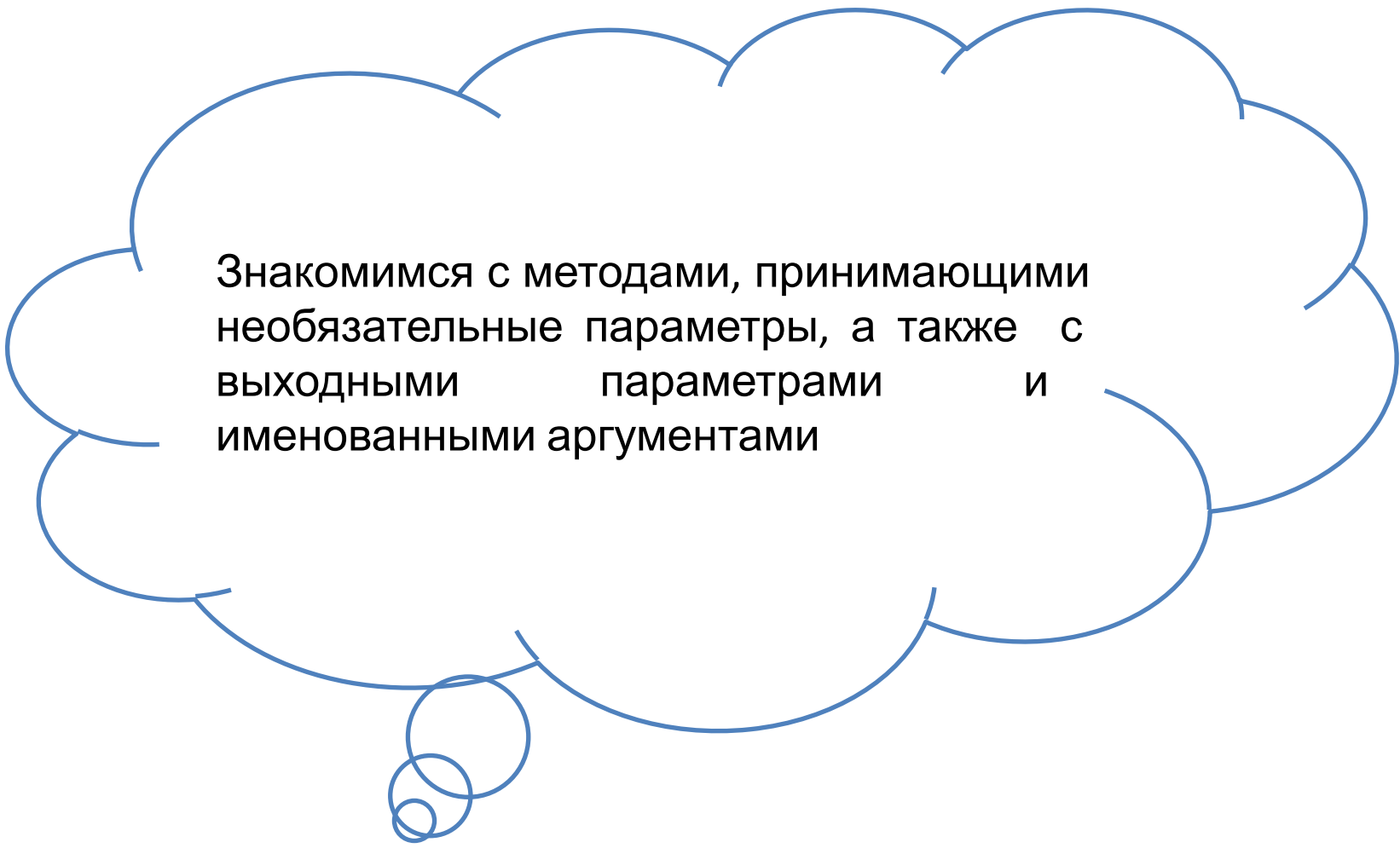
В диалоговом окне Extract Method, в поле New method name нужно ввести имя метода, а затем нажать кнопку OK

## Рефакторинг для извлечения метода

```
string messageContents = "My message text here";
string filePath = @"C:\Users\Student\Desktop";
if (messageContents != null)
{
    try
    {
        LogMessage(messageContents, filePath);
    }
    catch
    {
        File.AppendAllText(filePath, messageContents);
    }
}
if (filePath != null)
{
    try
    {
        private static void LogMessage(string messageContents, string filePath)
        {
            if (messageContents == null || messageContents == String.Empty)
            {
                throw new ArgumentException("Message cannot be empty");
            }

            if (filePath == null || !System.IO.File.Exists(filePath))
            {
                throw new ArgumentException("File path must exist");
            }
        }
    }
}
File.AppendAllText(filePath, messageContents);
}
```

## Необязательные и выходные параметры



Знакомимся с методами, принимающими  
необязательные параметры, а также с  
выходными параметрами и  
именованными аргументами

## Необязательные параметры

Используются при взаимодействии с другими технологиями, поддерживающими необязательные параметры

Используются, когда не представляется возможным использовать перегрузку, поскольку типы параметров не меняются так, чтобы компилятор проводил различие между реализациями

```
void MyMethod(int intData, float floatData, int moreIntData)
{
    ...
}

void MyMethod(int intData, float floatData)
{
    ...
}
```

```
int arg1 = 99;
float arg2 = 100.0F;
int arg3 = 101;
MyMethod(arg1, arg2, arg3);
MyMethod(arg1, arg2);
```



## Необязательные параметры

```
void MyMethod(int intData)
{...}
void MyMethod(int moreIntData,
{...}
```

**СТЕ**

```
void MyMethod(int intData, float floatData, int moreIntData = 99)
{
    ...
}
```

```
// Arguments provided for all three parameters
MyMethod(10, 123.45F, 99);
// Arguments provided for 1st two parameters only
MyMethod(100, 54.321F);
```

Нужно указать все обязательные параметры, прежде чем указывать любые необязательные

```
void MyMethod(int intData, float floatData = 101.1F, int moreIntData)
{...}

private static void Do(string message, DateTime dt = DateTime.Now)
{...}
```

**СТЕ**

Значение, присваиваемое необязательному параметру, должно быть известно во время компиляции и не может вычисляться во время выполнения

## Именованные аргументы

Указав имена параметров можно обеспечить метод аргументами в последовательности, которая отличается от порядка параметров в его сигнатуре

```
void MyMethod(int first, double second, string third)
{. . .}
. . .
MyMethod(third: "Hello", first: 1234, second: 12.12);
```

Объявление метода

Вызов метода с именованными аргументами

При использовании именованных аргументов в сочетании с необязательными параметрами, можно пропускать параметры

Можно смешивать позиционированные и именованные аргументы

Следует указывать все позиционированные аргументы до именованных

# Выходные параметры

Позволяют получить из метода дополнительные данные

Для определения выходного параметра, следует добавить ключевое слово `out` к параметру метода

```
void MyMethod(int first, double second, out int data)
{
    ...
    data = 99;
}
```

Присваивать начальное значения не обязательно

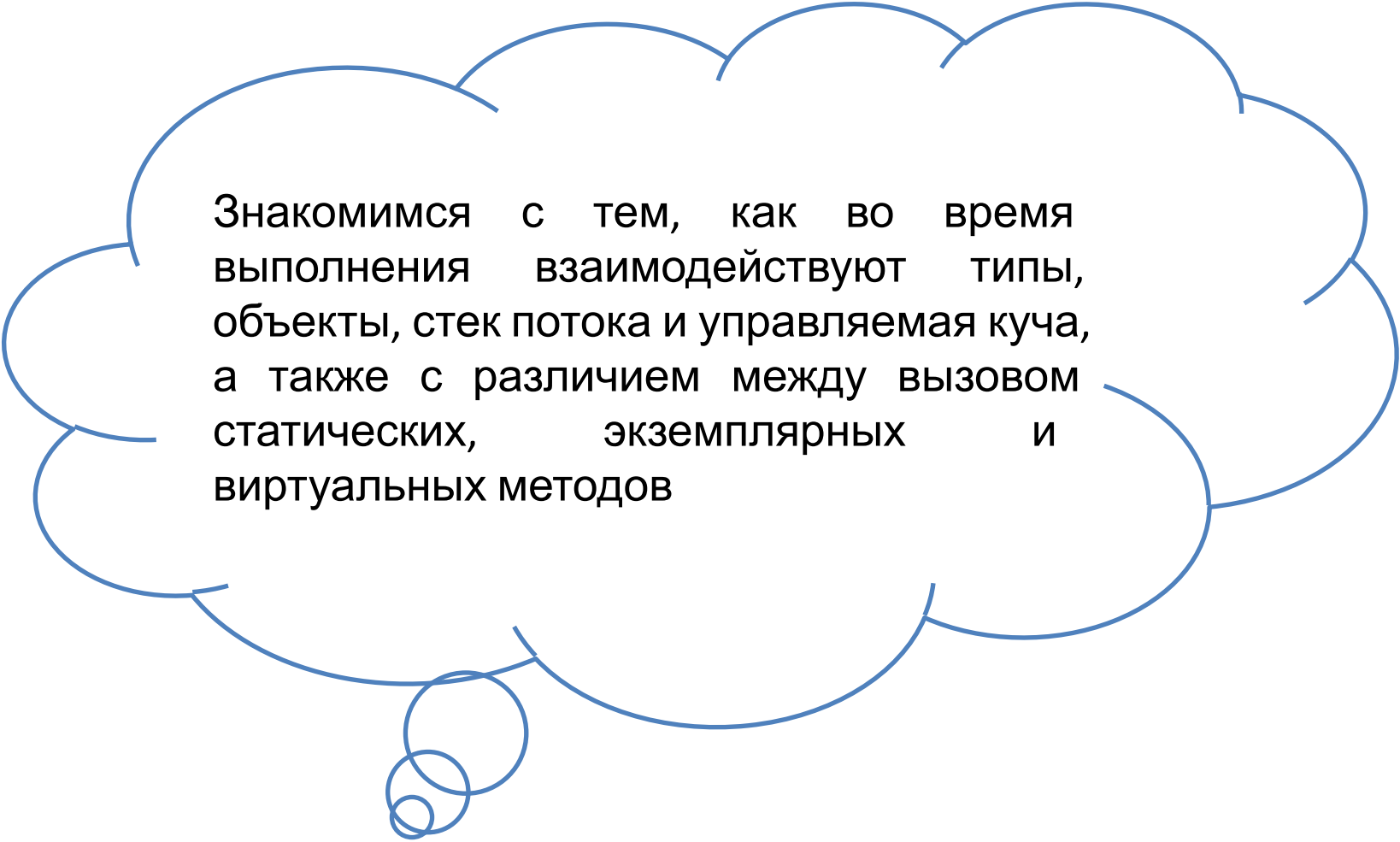
Обязательно присвоить соответствующие значения

value = 99

```
int value;
MyMethod(10, 101.1F, value);
```

**СТЕ**

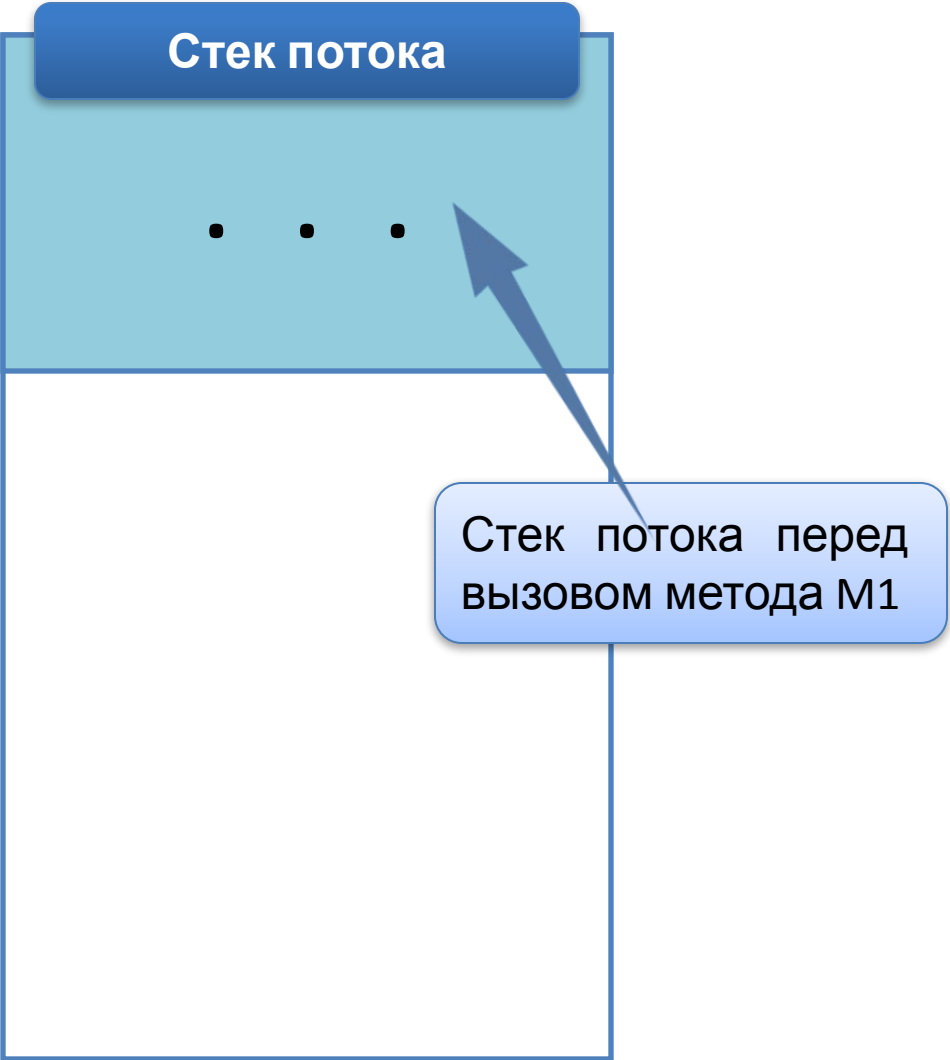
## Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения



Знакомимся с тем, как во время выполнения взаимодействуют типы, объекты, стек потока и управляемая куча, а также с различием между вызовом статических, экземплярных и виртуальных методов

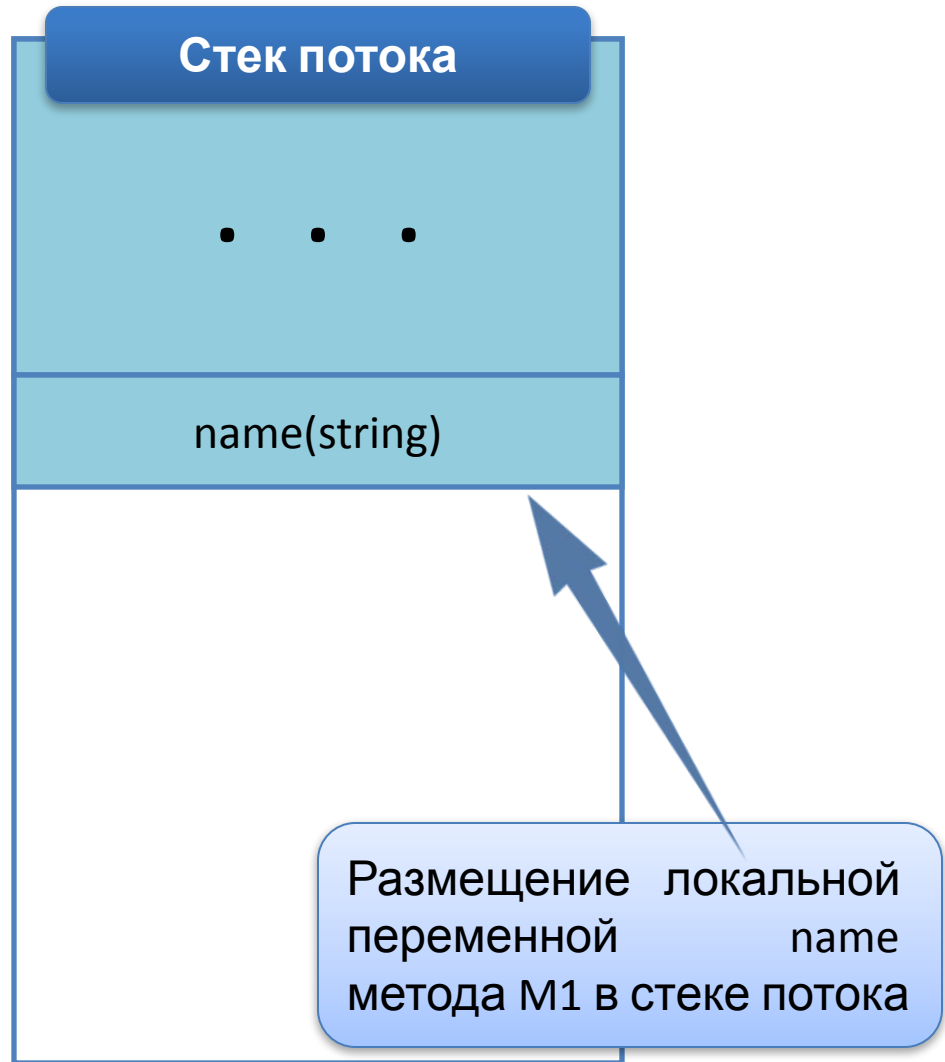
# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения

```
void M1()  
{  
    string name = "Joe";  
    M2(name);  
    . . .  
    return;  
}
```



# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения

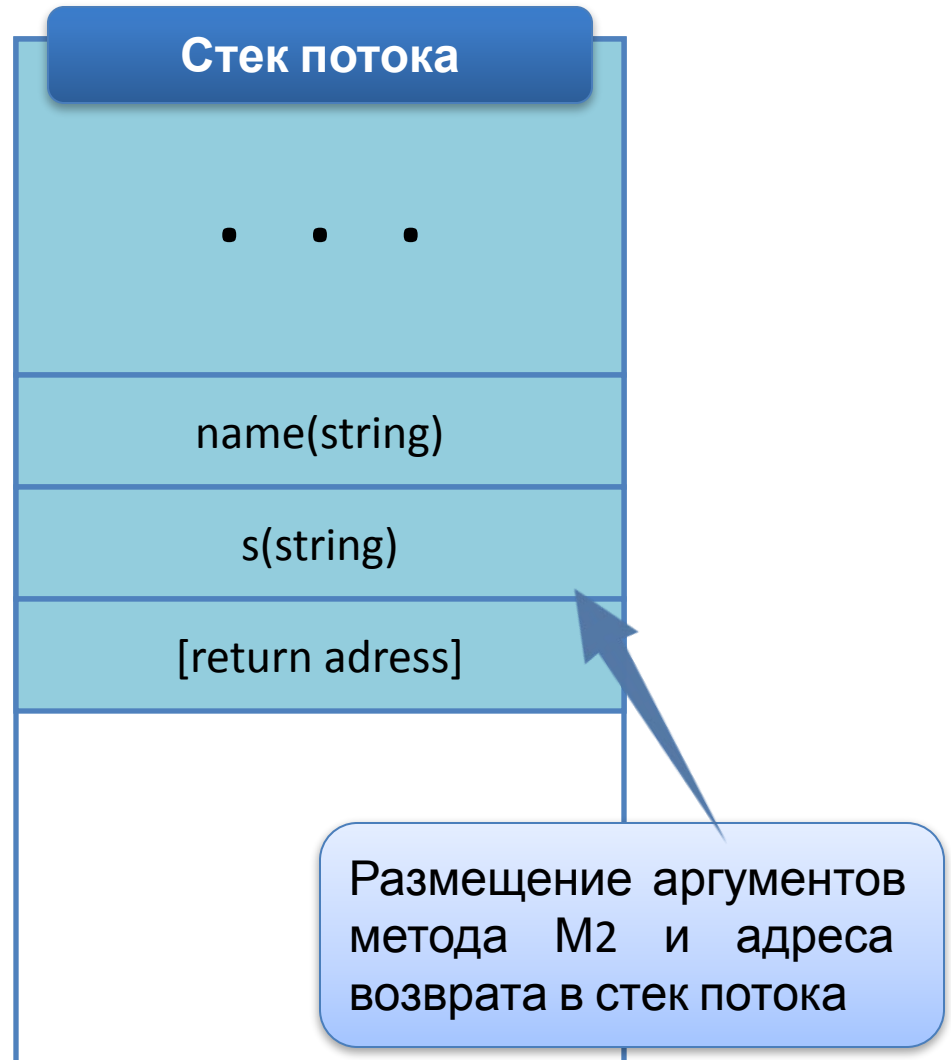
```
void M1()  
{  
    string name = "Joe";  
    M2(name);  
    . . .  
    return;  
}
```



# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения

```
void M1()  
{  
    string name = "Joe";  
    M2(name);  
    . . .  
    return;  
}
```

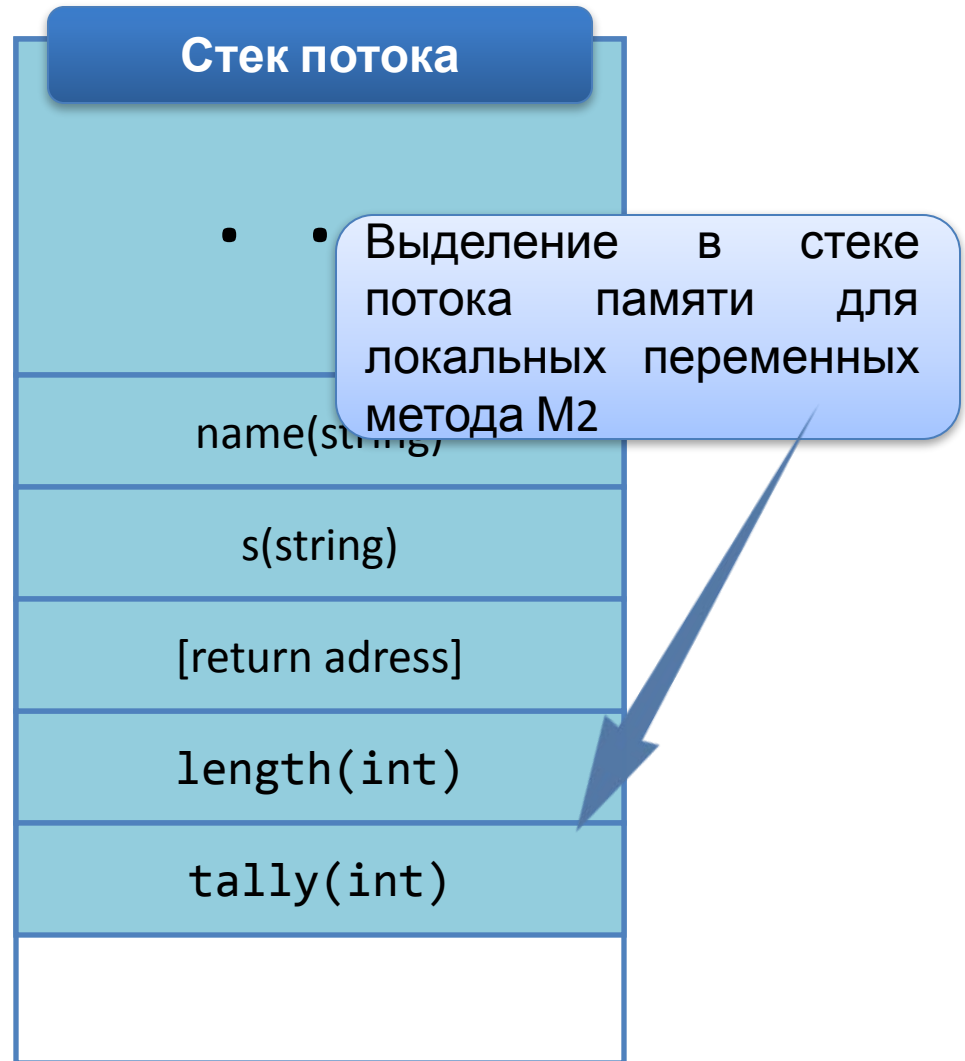
```
void M2(string s)  
{  
    int length = s.Length;  
    int tally;  
    . . .  
    return;  
}
```



# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения

```
void M1()  
{  
    string name = "Joe";  
    M2(name);  
    . . .  
    return;  
}
```

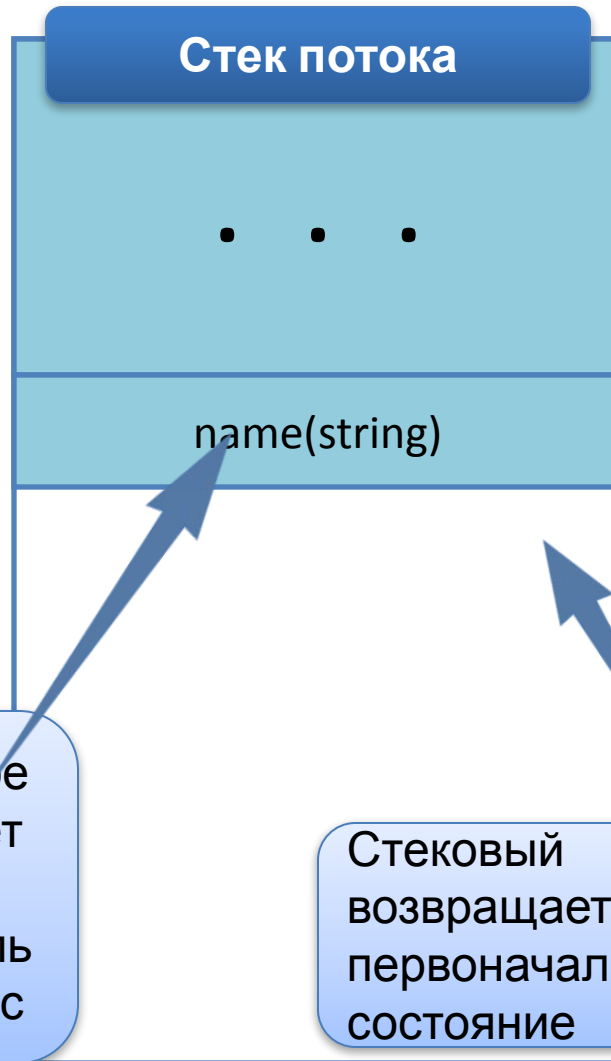
```
void M2(string s)  
{  
    int length = s.Length;  
    int tally;  
    . . .  
    return;  
}
```





# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения

```
void M1()  
{  
    string name = "Joe";  
    M2(name);  
    . . .  
    return;  
}
```



Стековый фрейм M1 возвращается в первоначальное состояние, M1 возвращает управление вызывающей функции, устанавливая указатель команд процессора на адрес возврата

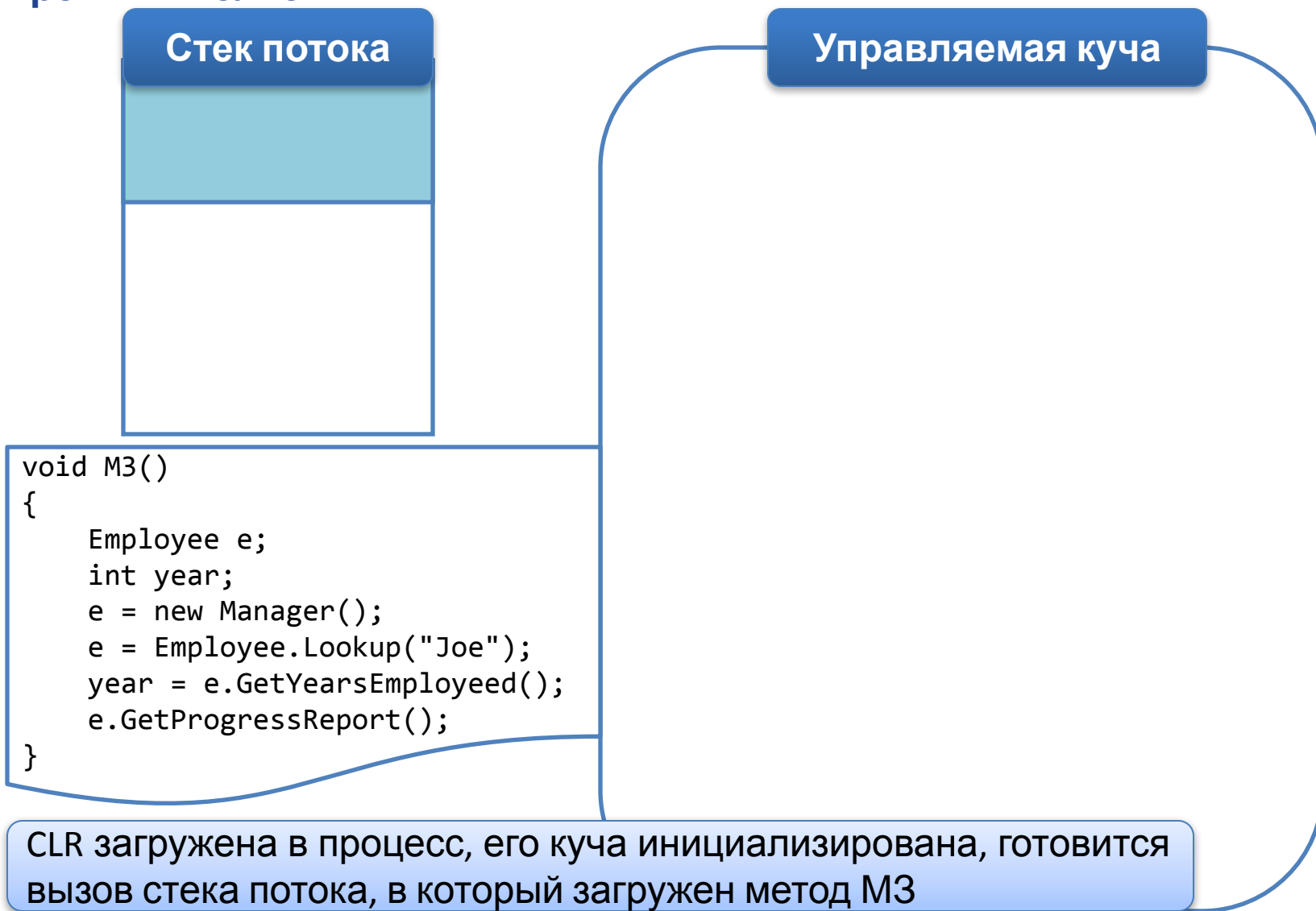
Стековый фрейм M2 возвращается в первоначальное состояние

## Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения

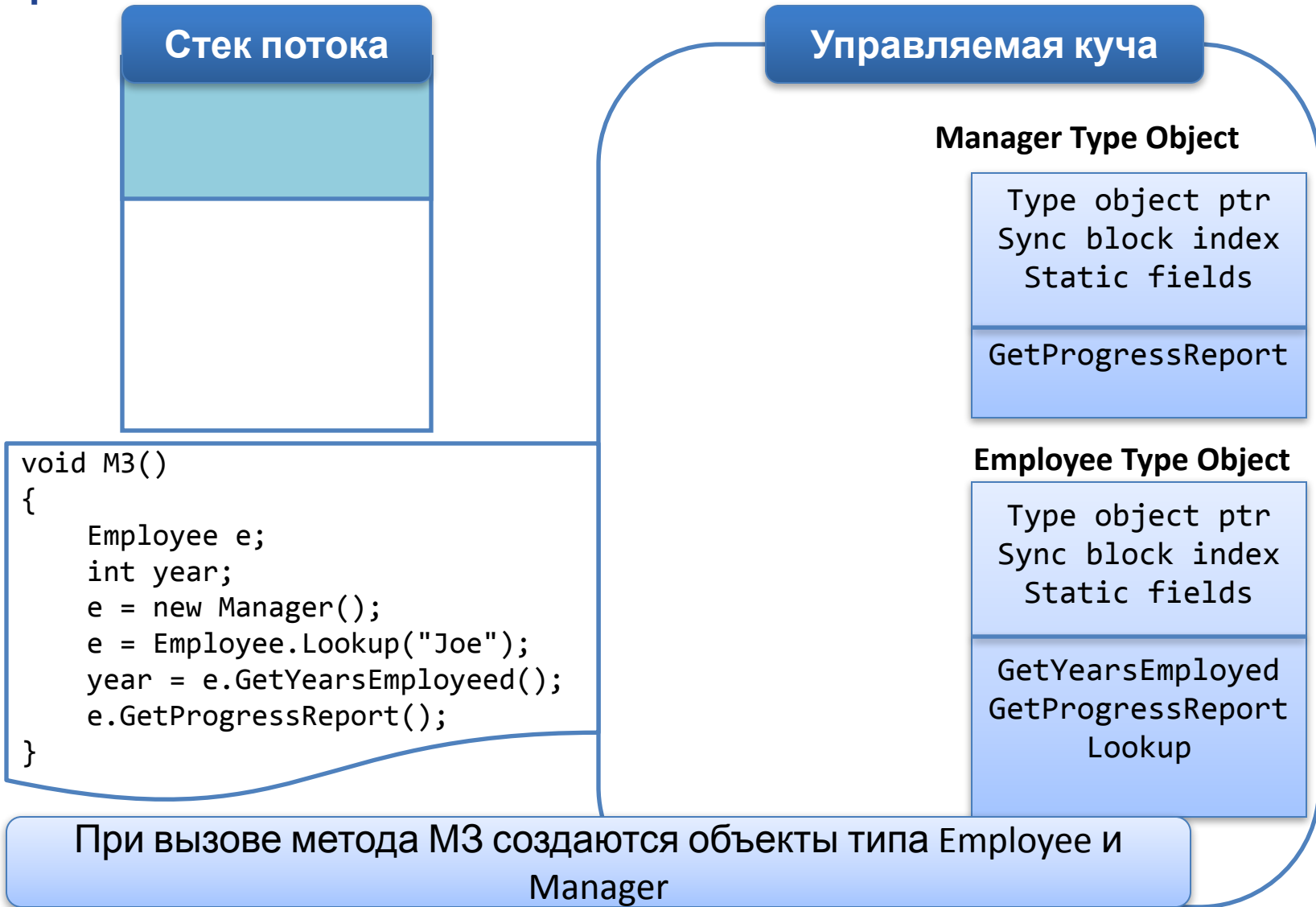
```
internal class Employee
{
    public int GetYearsEmployed { ... }
    public virtual string GetProgressReport { ... }
    public static Employee Lookup(string name) { ... }
}

internal sealed class Manager : Employee
{
    public override string GetProgressReport { ... }
}
```

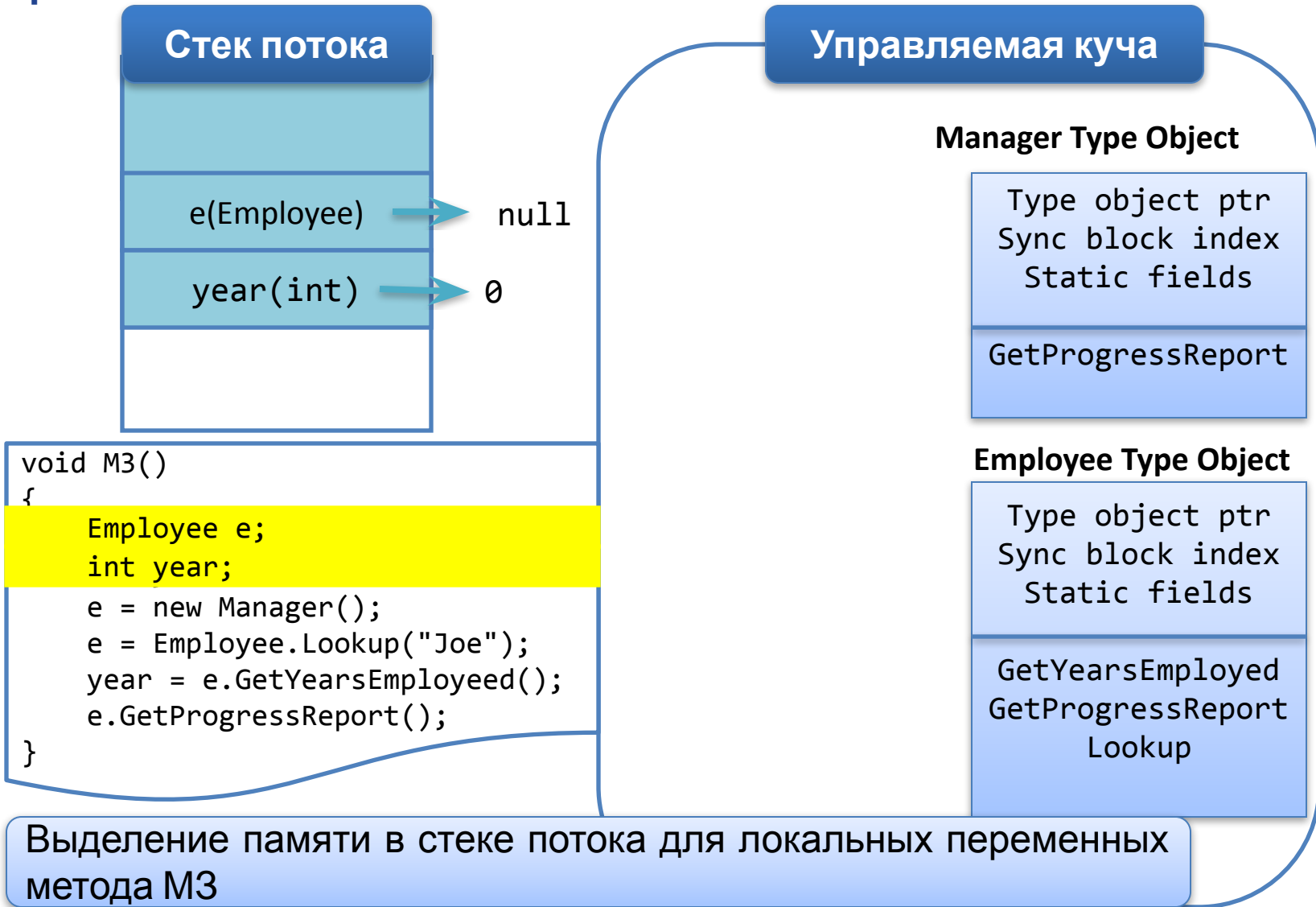
## Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения



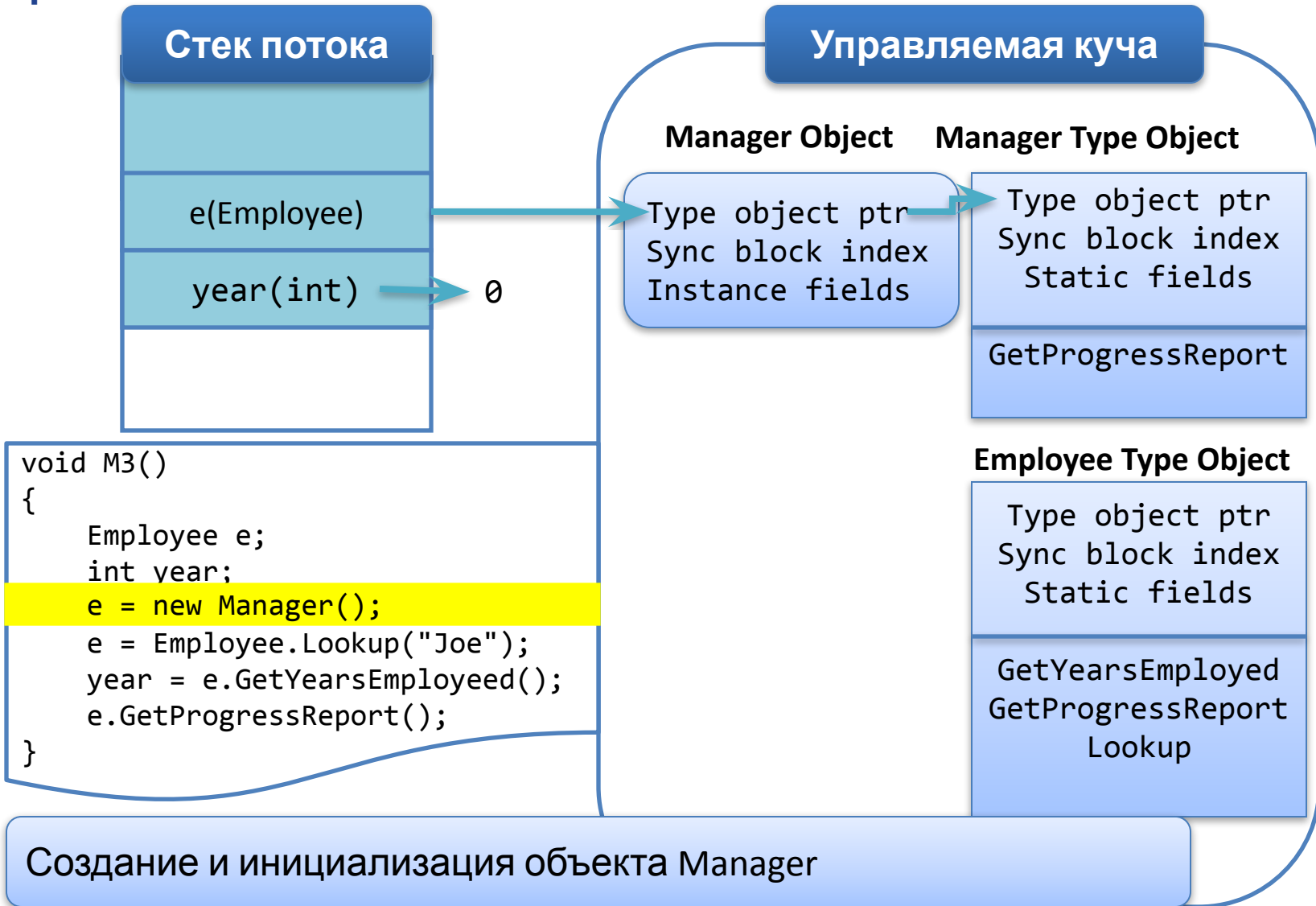
# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения



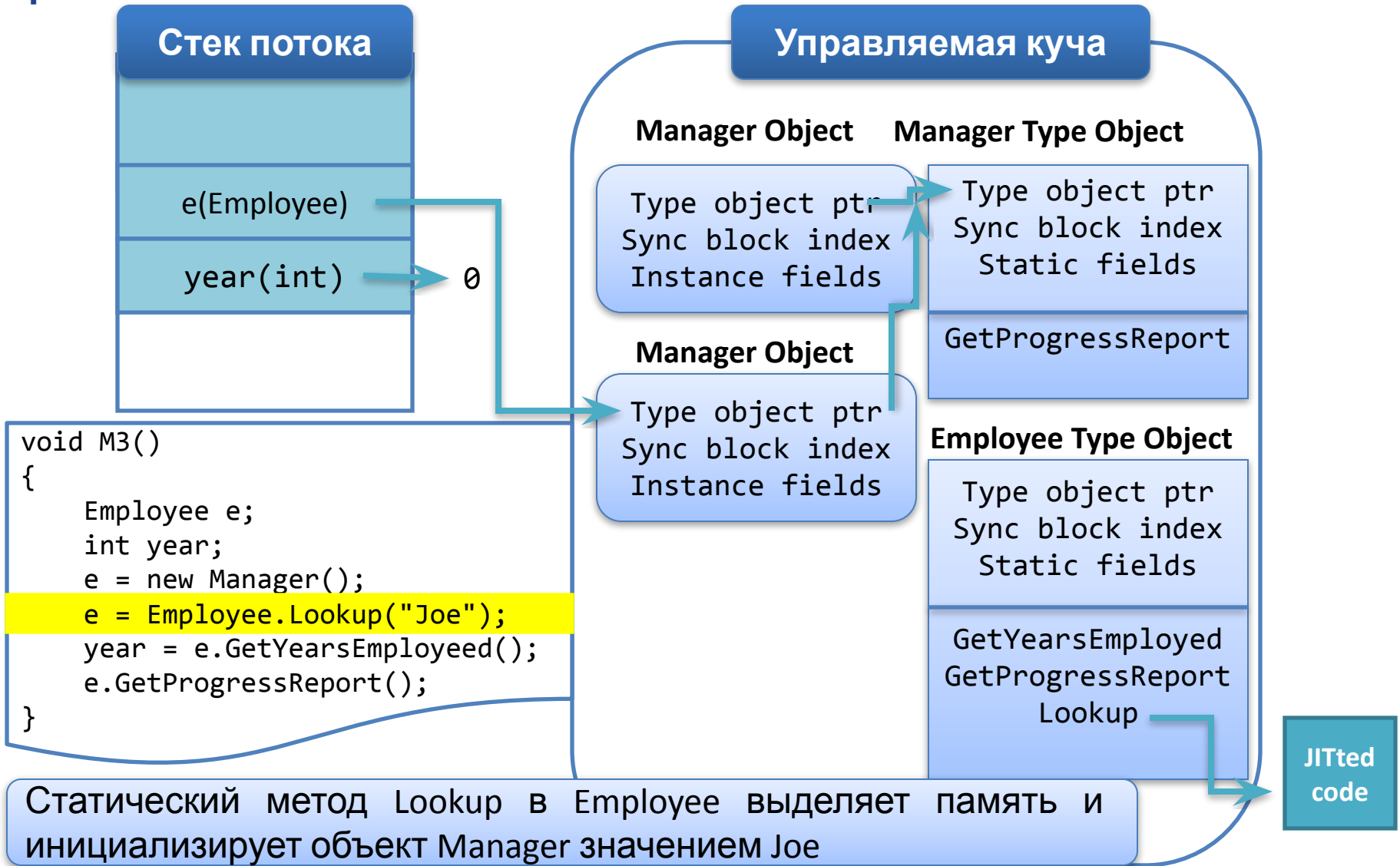
# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения



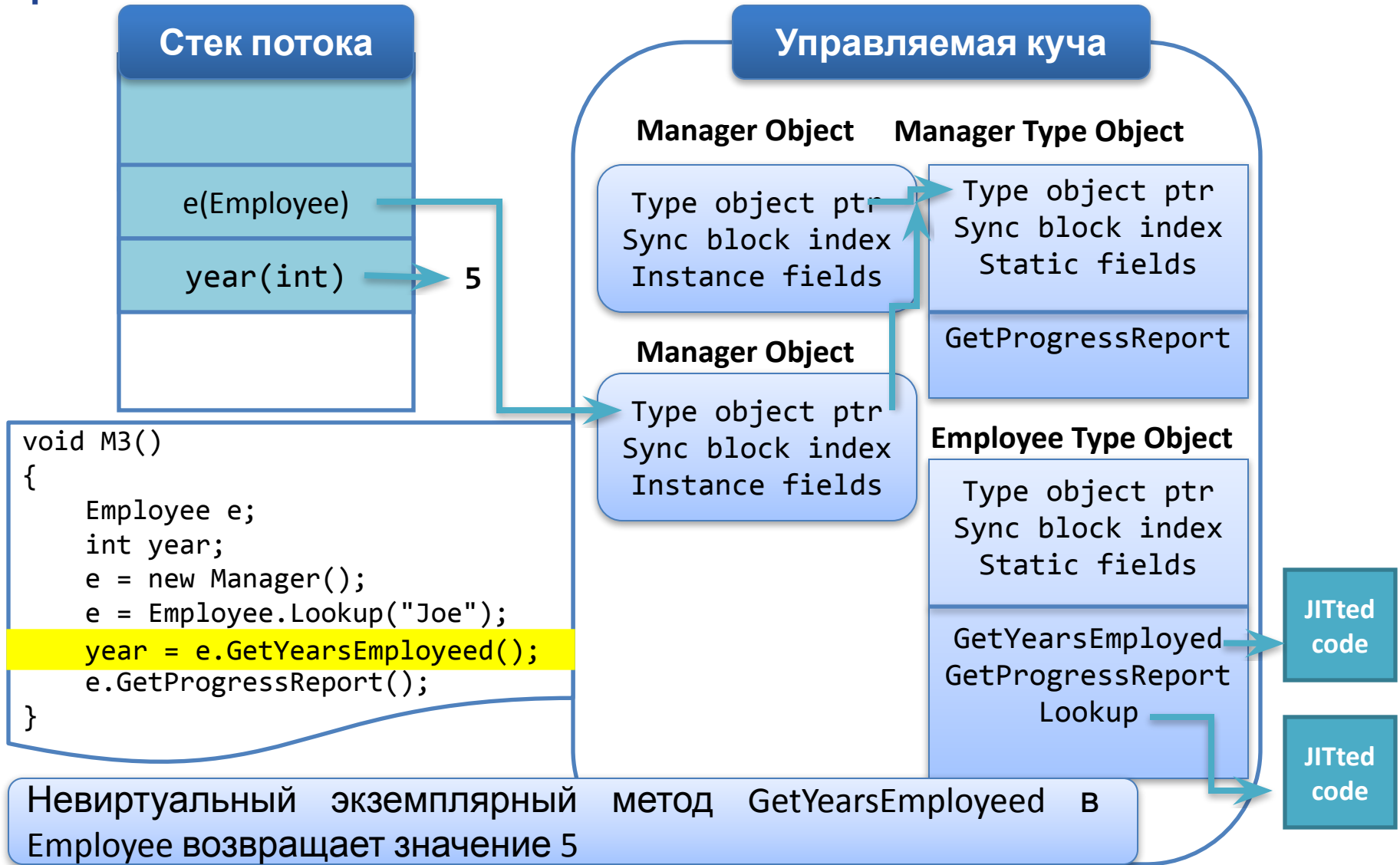
# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения



# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения

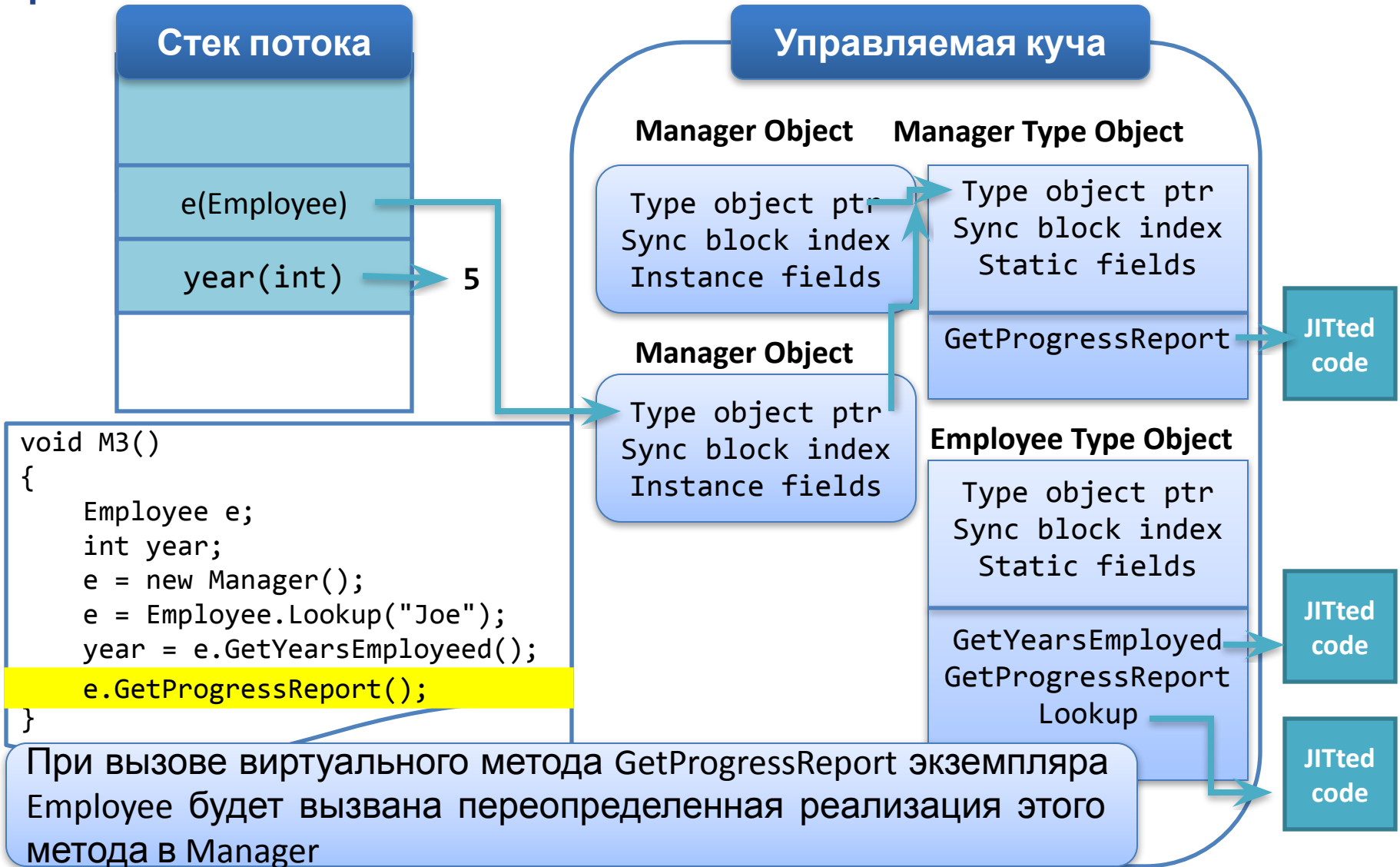


# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения

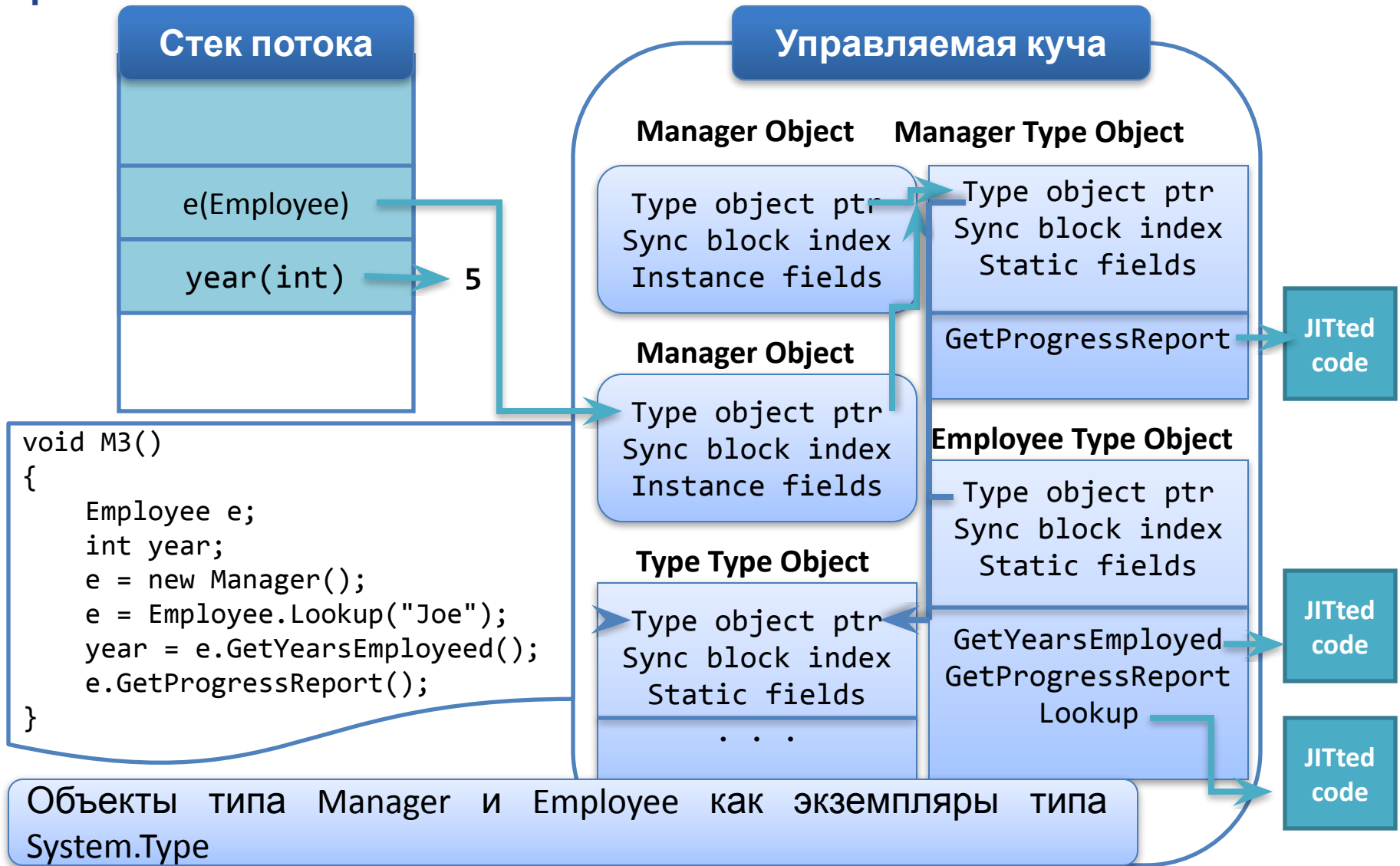




# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения



# Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения



Спасибо за внимание