.NET Development

ЗАНЯТИЕ 6

Рассматриваемые вопросы

- Наследование классов
- Преобразование и приведение ссылочных типов
- Перекрытие элементов класса и полиморфизм
- Иерархия типов платформы .NET
- Упаковка и распаковка
- Методы класса System.Object

Концепция наследования в ООП

Наследование (inheritance) позволяет описать новый класс (*класс-потомок, дочерний класс, подкласс*) на основе уже существующего класса (*класс-предка, родительского класса, суперкласса*).

Класс-потомок имеет элементы класса-предка и может добавить к ним собственные новые элементы. Это соответствует уточнению, конкретизации понятий (**служащий** – **человек**, получающий ***зарплату***).

Наследование классов в C#

Классы в C# поддерживают наследование. При этом:

- Наследование от двух и более классов запрещено.
- В класс-потомок переносятся все элементы класса-предка, кроме конструкторов.
- `private`-элементы предка не доступны наследнику (это правило не работает, если наследник *вложен* в предок).
- При наследовании нельзя расширить видимость класса (`public`-класс нельзя унаследовать от `internal`-класса).

Синтаксис наследования (на примере)

```
public class Person
{
    public string Name { get; set; }
}

internal class Employee : Person
{
    public int Salary { get; set; }
}
```

Наследование и конструкторы

Конструкторы предка не переносятся в потомок, но доступны из него.

В начале работы конструктор потомка **должен** вызвать другой свой конструктор (`:this(...)`) или какой-то конструктор предка (`:base(...)`). Если это не сделано явно, компилятор «подставляет» вызов `:base()`. Если в предке нет конструктора без параметров, то получим ошибку компиляции.

Наследование и конструкторы

```
public class Person
{
    public string Name { get; set; }

    public Person() => Name = "Unknown";

    public Person(string name) => Name = name;
}
```

Наследование и конструкторы

```
public class Employee : Person
{
    public int Salary { get; set; }

    public Employee() { }

    public Employee(string name) : base(name)
        => Salary = 100;

    public Employee(string name, int salary) : this(name)
        => Salary = salary;
}
```

Наследование и конструкторы

```
var x = new Employee();
// Employee() вызывает Person()
// Name="Unknown", Salary = 0

var y = new Employee("Alex");
// Employee(name) вызывает Person(name)
// Name = "Alex", Salary = 100

var z = new Employee("Alex", 1);
// Employee(name, salary) вызывает Employee(name),
// который вызывает Person(name)
// Name = "Alex", Salary = 1
```

Совместимость при присваивании

В C# действует классическое правило: **объекту предка можно присвоить объект потомка, но не наоборот**:

```
Person p = new Person();
Employee e = new Employee();
p = e;  // допустимо
e = p;  // не компилируется
```

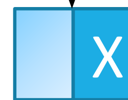
*) Предок и потомок – любые в иерархии наследования, не обязательно непосредственные.

Совместимость при присваивании

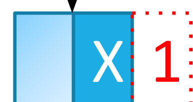
```
class A
{
    public int X;
}


class B : A
{
    public int Y;
}
```

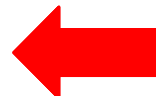
```
A a = new(); B b = new();
```

```
b = a;
```




```
b.Y = 1;
```





Мы «вылезли» за пределы объекта!

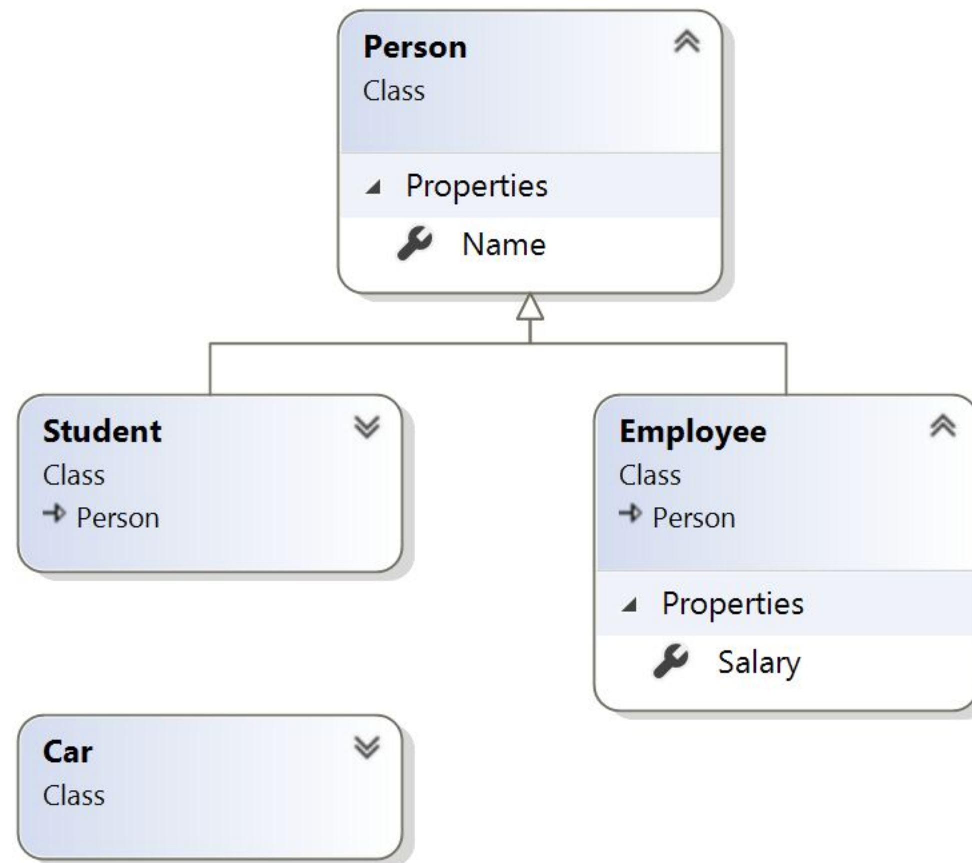
Ссылочные преобразования

Ссылочные преобразования – преобразования типов, выполняемые для объектов.

Любой объект может быть неявно приведён к типу класс-предка и явно приведён к типу класса-потомка. При ошибках явного приведения генерируется исключение `System.InvalidCastException`.

Пример ссылочных преобразований

```
// объявим ещё два класса,
// кроме Person и Employee
public class Student : Person
{
}

public class Car
{
}
```



Пример ссылочных преобразований

```
Person p; Employee e; Student s;

p = new Employee(); // неявное приведение

e = (Employee) p; // компилируется и работает
s = (Student) p;  // при работе - InvalidCastException

e = new Person(); // не компилируется
Car c = new Car();
p = (Person) c; // не компилируется - разные иерархии
```

Операция as – безопасное приведение

```
Obj as T
```

Условие успешной компиляции: для `Obj` существует явное или неявное ссылочное преобразование в тип `T`.

Если преобразование безопасно (типы совместимы), операция `as` возвращает приведённый результат. Иначе возвращается `null` (а исключение не генерируется).

Операция as – пример

```
Person p;
if (DateTime.Now.Second % 2 == 0) // случайно создаём
    p = new Student();
else
    p = new Employee();

Employee e = p as Employee;
if (e != null)
    Console.Write(e.Salary);
else
    Console.Write("This is not an Employee");
```

Операция is – проверка типа

```
Obj is T
```

Операция `is` возвращает `true`, если `Obj` не равен `null`, и преобразование `Obj` к типу `T` определено и безопасно (не генерирует исключений).

Операция is – пример

```
// код случайной инициализации объекта p
// смотрите в предыдущем примере

Employee e;
if (p is Employee)
{
    e = (Employee) p; // точно не будет исключения!
    Console.WriteLine(e.Salary);
}
```

Операция is с приведением типов

При выполнении операции `is` можно указать не только тип, но **и переменную**. При успехе выполняется приведение типов в новую переменную:

```
// переписали предыдущий пример более компактно
if (p is Employee e)
{
    Console.WriteLine(e.Salary);
}
```

Перекрытие элементов

Потомок может объявить элемент с тем же именем, что и элемент в предке – это *перекрытие элемента*.

Если перекрываются поля, свойства, а также методы и индексаторы с той же сигнатурой, говорят, что элемент потомка *скрывает* (hides) элемент предка. В этой ситуации рекомендуется использовать модификатор `new`, чтобы показать, что сокрытие не случайно.

Пример перекрытия методов

```
public class Person
{
    public string Name { get; set; }

    public void Setup(string name) => Name = name;

    public void Display() => Console.WriteLine(Name);
}
```

Пример перекрытия методов

```
public class Employee : Person
{
    public int Salary { get; set; }
    public void Setup(string name, int salary)
    {
        Name = name; Salary = salary;
    }
    public new void Display()
    {
        base.Display(); // base – для доступа к элементу предка
        Console.WriteLine($"Salary = {Salary}");
    }
}
```

Вызов перекрытых методов

По умолчанию при наличии перекрытия вызываемый метод определяется **на этапе компиляции** по объявленному (а не по фактическому) типу объекта:

```
Person p;
p = new Employee();

// это вызов Person.Display()
p.Display();
```

Полиморфный вызов

При таком вызове нужный метод будет определятся **на этапе выполнения** кода по фактическому типу объекта (для нахождения метода нужно «залезть» в объект).

Для организации полиморфного вызова метод в предке помечается модификатором `virtual`, а перекрытый метод в потомке – модификатором `override`.

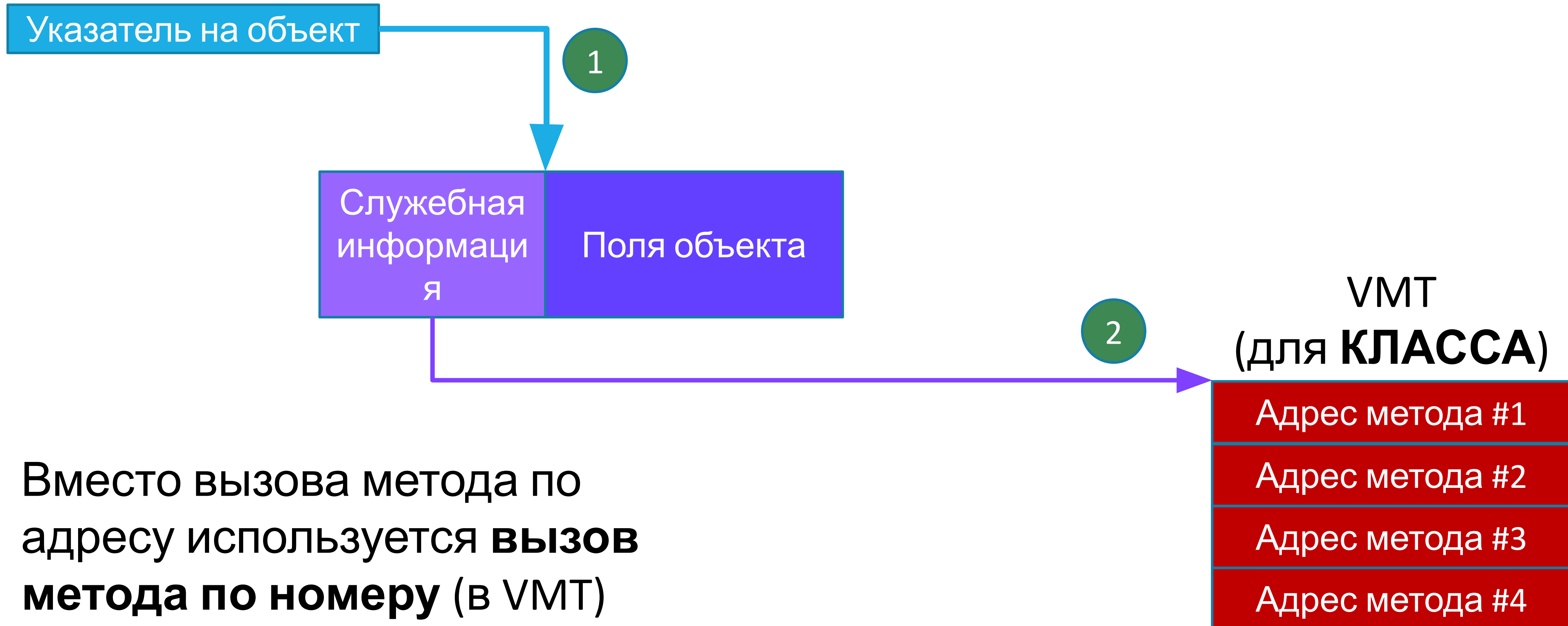
Полиморфизм – пример

```
public class Person
{
    ...
    public virtual void Display() => Console.WriteLine(Name);
}

public class Employee : Person
{
    ...
    public override void Display()
    {
        base.Display();
        Console.WriteLine($"Salary = {Salary}");
    }
}
```

Полиморфизм – пример

```
Person p;
p = new Employee();

// так как Display() – виртуальный метод, заглянем в p
// по факту, p – это объект Employee
// значит, вызываем Employee.Display()
p.Display();
```

Полиморфизм – технические детали



Вместо вызова метода по адресу используется **вызов метода по номеру** (в VMT)

Полиморфизм – определение

Полиморфизм – особый способ перекрытия методов при наследовании, при котором код, работающий с методами класса-предка, пригоден без изменений для работы с методами класса-потомка (даже если этот потомок не описан в момент создания кода).

Полиморфный вызов: нюансы

- Можно организовывать *полиморфные цепочки*, используя `override` в потомках потомка и так далее.
- Для работы полиморфного вызова методы должны совпадать по **имени**, **сигнатуре** и **типу возвращаемого значения** (в C# 9 – можно указать тип-потомок).
- Полиморфизм возможен для свойств и индексаторов (ведь свойство – это пара методов).

Covariant return types (C# 9)

```
public class Animal
{
    public virtual Food GetFood() { . . .}
}

public class Tiger : Animal
{
    // C# 9 - допустимо, если Meat - наследник Food
    public override Meat GetFood() { . . .}
}
```

Наследование и полиморфизм

Демонстрация кода 01: классы, связанные наследованием, без использования полиморфизма.

Демонстрация кода 02: классы, связанные наследованием и реализующие полиморфизм.

Наследование – модификаторы классов

Модификатор `sealed` определяет *запечатанный класс*, от которого запрещено наследование.

Модификатор `abstract` определяет *абстрактный класс*, у которого обязательно должны быть потомки. Объект абстрактного класса нельзя создать, хотя статические элементы такого класса можно вызвать.

Абстрактные методы

Для методов в **абстрактных классах** можно применить модификатор `abstract`. Он говорит о том, что метод не реализуется в классе, **не содержит тела** и должен обязательно переопределяться в классе-потомке.

В такой ситуации модификатор метода `abstract` эквивалентен модификатору `virtual`.

*) также возможны абстрактные свойства и индексаторы

Пример абстрактного класса

```
public abstract class Figure
{
    public string Name { get; }

    protected Figure(string name) => Name = name;

    // реализация будет в обязательных наследниках
    public abstract double GetArea();
}
```

Запечатанные методы

Модификатор `sealed` может применятся к методам с модификатором `override`. Такой *запечатанный метод* не может быть перекрыт в классе-потомке.

```
public class Employee : Person
{
    ...
    public sealed override void Display() { ... }
}
```

*) возможны запечатанные свойства и индексаторы

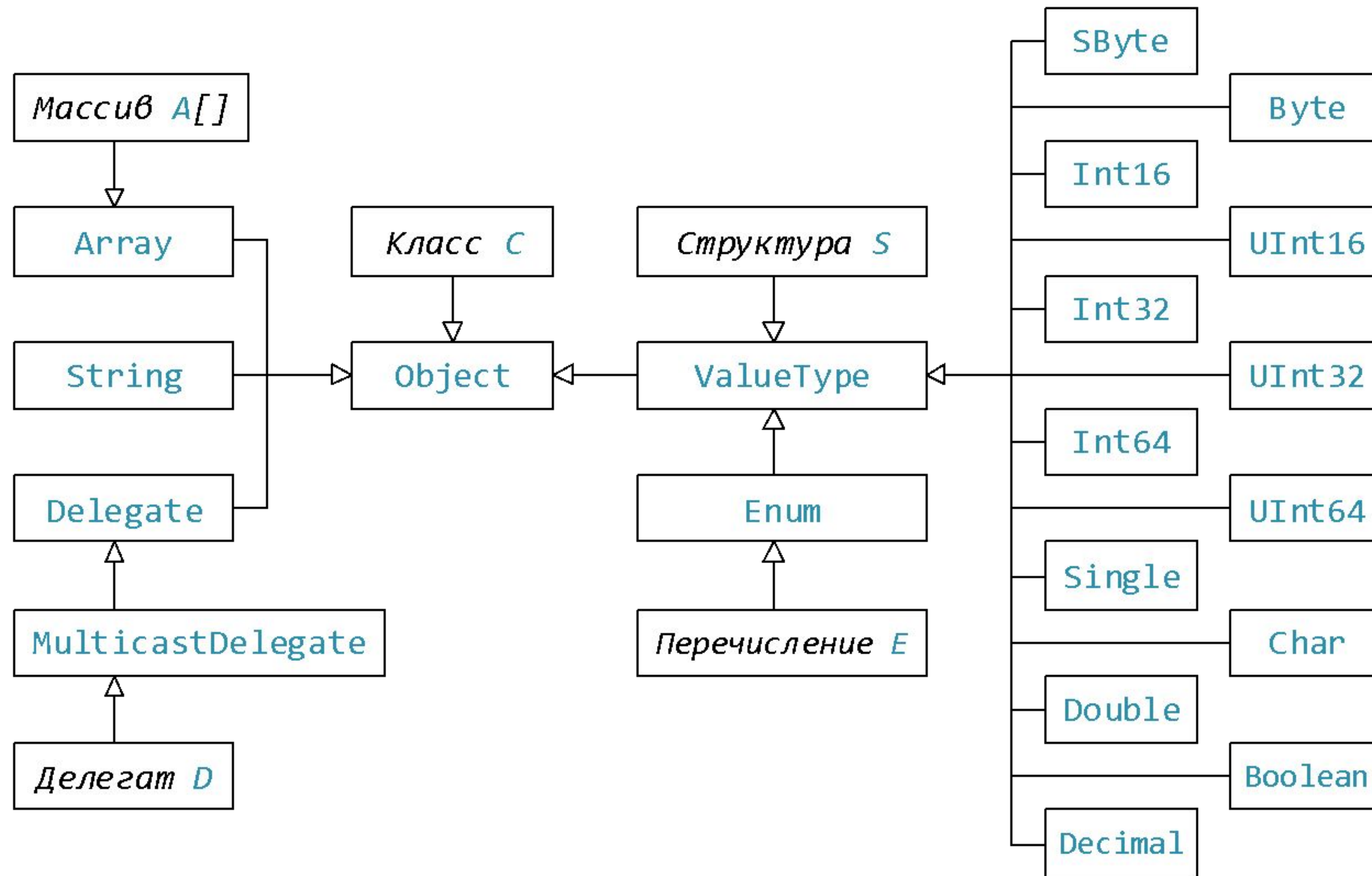
Иерархия типов платформы .NET

В .NET **все типы (кроме интерфейсов)** связаны общим отношением наследования.

У всех типов есть общий предок – класс `System.Object` (псевдоним в C# – `object`).

Предок всех типов значений – класс `System.ValueType`.

Иерархия типов платформы .NET



Упаковка

Напомним, что переменные типов значений хранятся в стеке, а объекты хранятся в динамической памяти.

С другой стороны, так как все типы связаны в иерархию, то логично предположить, что переменной типа `object` можно присвоить переменную типа значения.

Такое действие в .NET и C# разрешено и называется *операцией упаковки* (boxing).

Упаковка

```
int x = 1234;
object o = x;  // здесь выполняется boxing
```

При упаковке в динамической памяти создаётся объект, хранящий значение и информацию о типе значения.

*) Упаковка – медленная операция (подумайте, почему).

Распаковка

Упакованное значение можно поместить в переменную типа значения при помощи *операции распаковки* (unboxing). «Коварство» в том, что в C# распаковка выглядит как операция явного приведения типов.

```
object o = 1234;  // упаковка
int x = (int) o;  // распаковка в int

// распаковка в int, а затем приведение к short
short y = (short) (int) o;
```

Упаковка и распаковка

Стек

«Куча»

```
int x = 1234;
```

1234

```
object o = x;
```

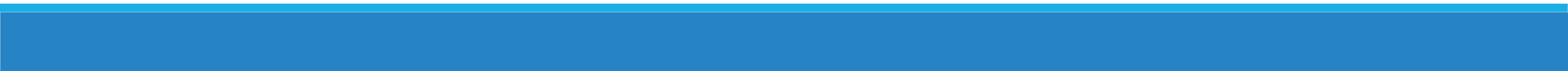
int

1234

```
int y = (int)o;
```

1234

Контроль типа!



Методы класса System.Object

Так как все типы наследуются от `System.Object`, то все значения в .NET обладают методами этого класса (в `System.Object` нет полей и свойств).

Кроме этого, в пользовательском типе можно перекрыть некоторые методы `System.Object`.

Далее рассматриваются методы класса `System.Object` в алфавитном порядке.

Методы класса System.Object

```
public virtual bool Equals(object obj)
```

Этот метод определяет, равен ли текущий объект переданному объекту `obj`. Стандартная реализация `Equals()` проверяет равенство ссылок для ссылочных типов и побитовое равенство полей для типов значений.

Правила переопределения Equals()

- `x.Equals(x) == true`
- `x.Equals(y) == y.Equals(x)`
- `(x.Equals(y) && y.Equals(z)) == true` ⟹ `x.Equals(z) == true`
- Вызовы метода `x.Equals(y)` возвращают одинаковое значение, если x и y остаются неизменными
- `x.Equals(null) == false`, если `x != null`
- Метод `Equals()` не должен генерировать исключений

Методы класса System.Object

```
public static bool Equals(object a, object b)
```

Метод `Equals()` определяет, равны ли его аргументы:

- Если оба аргумента равны `null`, метод возвращает `true`.
- Если только один аргумент `null`, возвращается `false`.
- Если оба аргумента не равны `null`, возвращается результат `a.Equals(b)`.

Методы класса System.Object

```
protected virtual void Finalize()
```

Метод `Finalize()` позволяет объекту освободить ресурсы и выполнить операции очистки, перед тем как объект будет утилизирован в процессе сборки мусора.

Методы класса System.Object

```
public virtual int GetHashCode()
```

Метод играет роль хеш-функции. Пользовательские типы могут переопределять это метод для эффективного вычисления хеш-кода.

Если два объекта равны (то есть `a.Equals(b)=true`), методы `GetHashCode()` этих объектов должны возвращать одинаковые значения.

Методы класса System.Object

```
public Type GetType()
```

Метод `GetType()` возвращает объект класса `System.Type`, который содержит *метаданные*, связанные с типом текущего экземпляра.

Методы класса System.Object

```
protected object MemberwiseClone()
```

Метод выполняет создание *неглубокой копии* объекта.

Метод создаёт новый объект (без вызова конструктора!) и копирует в него экземплярные поля текущего объекта. Если поле относится к типу значения, выполняется побитовое копирование. Если поле относится к ссылочному типу – копируется ссылка на объект.

Методы класса System.Object

```
public static bool ReferenceEquals(object a, object b)
```

Статический метод `ReferenceEquals()` возвращает значение `true`, если `a` соответствует тому же экземпляру, что и `b` (совпадение ссылок), или же оба они равны `null`. В противном случае метод возвращает `false`.

Методы класса System.Object

```
public virtual string ToString()
```

Метод возвращает строковое представление объекта.

Класс System.Object – сводка методов

```
public virtual bool Equals(object obj)
public static bool Equals(object a, object b)
protected virtual void Finalize()
public virtual int GetHashCode()
public Type GetType()
protected object MemberwiseClone()
public static bool ReferenceEquals(object a, object b)
public virtual string ToString()
```

Методы класса System.Object

Демонстрация кода 03: примеры классов с перекрытием методов `System.Object`.