1. Если вы возьмете справочник [Шилдта по Java](https://www.ozon.ru/context/detail/id/31249554/), то на 171 странице вы прочитаете о том, что ключевое слово this требуется для того, чтобы метод мог сослаться на вызвавший его объект. На этом, собственно, можно было бы и закончить. Но нам нужна конкретика.

Как правило, применять this нужно в двух случаях:

1. Когда у переменной экземпляра класса и переменной метода/конструктора одинаковые имена;
2. Когда нужно вызвать конструктор одного типа (например, конструктор по умолчанию или параметризированный) из другого. Это еще называется явным вызовом конструктора.

Вот и все, на самом деле не так много, — всего два случая, когда применяется это страшное ключевое слово. Теперь давайте рассмотрим эти две ситуации на примерах.

**2)** Абстракция – это, принцип ООП, согласно которому при проектировании классов и создании объектов необходимо выделять только главные свойства сущности, и отбрасывать второстепенные.

Например, если будем проектировать класс SchoolTeacher — школьный учитель — вряд ли понадобится характеристика «рост». Действительно: для преподавателя эта характеристика не важна. Но вот если мы будем создавать в программе класс BasketballPlayer — игрок в баскетбол — рост станет одной из главных характеристик.

Так вот, абстрактный класс — это максимально абстрактная «заготовка» для группы будущих классов. Эту заготовку нельзя использовать в готовом виде — слишком «сырая». Но она описывает некое общее состояние и поведение, которым будут обладать будущие классы — наследники абстрактного класса.

**Абстрактный класс** в [объектно-ориентированном программировании](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — базовый [класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), который не предполагает создания экземпляров. Абстрактные классы реализуют на практике один из принципов ООП — [полиморфизм](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Абстрактный класс может содержать (и не содержать[[1]](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BB%D0%B0%D1%81%D1%81#cite_note-1)) [абстрактные методы](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4) и [свойства](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Абстрактный метод не реализуется для класса, в котором описан, однако должен быть реализован для его неабстрактных потомков. Абстрактные классы представляют собой наиболее [общие](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D0%BD%D1%8F%D1%82%D0%B8%D0%B9) абстракции, то есть имеющие наибольший [объём](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D1%91%D0%BC_%D0%BF%D0%BE%D0%BD%D1%8F%D1%82%D0%B8%D1%8F) и наименьшее [содержание](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B4%D0%B5%D1%80%D0%B6%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D0%BD%D1%8F%D1%82%D0%B8%D1%8F).

В одних языках создавать экземпляры абстрактных классов запрещено, в других это допускается (например, Delphi), но обращение к абстрактному методу объекта этого класса в процессе выполнения программы приведёт к ошибке. Во многих языках допустимо объявить любой класс абстрактным, даже если в нём нет абстрактных методов (например, Java), именно для запрета создания экземпляров. Абстрактный класс можно рассматривать в качестве [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) к семейству классов, порождённому им, но, в отличие от классического интерфейса, абстрактный класс может иметь определённые методы, а также свойства.

3) Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

Допустим, некоторому алгоритму нужно выполнить 4n3 + 7n условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление 7n. Тогда говорят, что временная сложность этого алгоритма равна О(n3), т. е. зависит от размера входных данных кубически.

Использование заглавной буквы О (или так называемая О-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

### O(n) — линейная сложность

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

**O(log n) — логарифмическая сложность**

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

**O(n2) — квадратичная сложность**

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.

Бывают и другие оценки по сложности, но все они основаны на том же принципе.

Также случается, что время работы алгоритма вообще не зависит от размера входных данных. Тогда сложность обозначают как O(1). Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных третий элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.

Аналогично проводят оценку и по памяти, когда это важно. Однако алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

**4)** Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями

**5)** **Наследование** классов (inheritance) – иерархическое отношение между классами типа «частное – общее»; «частные» классы наследуют состояние и поведение (поля и методы) «общих» классов, то есть «частный» класс обладает как собственным состоянием и поведением (полями и методами), так и «унаследованным» состоянием и поведением, определенным в «общем» классе.

У нас есть следующий класс, описывающий коробку:

public class Box6 {

double width;

double height;

double depth;

Box6(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

public Box6() {

}

double getVolume() {

return width \* height \* depth;

}

}

И вдруг заказчик решает добавить цветную коробку - ColorBox и тяжелую коробку - HeavyBox. Цветная коробка будет отличаться от обычной только цветом, а тяжелая - весом. Получается, что в класс ColorBox мы должны добавить те же переменные width, height и depth, конструкторы и методы, которые существуют в классе Box. Дублирование кода в программировании не приветствуется, поэтому для таких случаев придуман такой механизм как наследование.

Используя наследование Java, можно создать класс, который определяет характеристики, общие для набора связанных элементов - Box. Затем этот общий класс может наследоваться другими, более специализированными классами ColorBox и HeavyBox, каждый из которых будет добавлять свои особые характеристики.

В терминологии Java наследуемый класс называется супер классом, а наследующий класс – подклассом. Подкласс наследует все члены, определенные в супер классе, добавляя к ним собственные, особые элементы. Набор классов, связанных отношением наследования, называют иерархией, что изображается таким образом:

**6)** В мире программирования возникновение ошибок и непредвиденных ситуаций при выполнении программы называют **исключением**. В программе исключения могут возникать в результате неправильных действий пользователя, отсутствии необходимого ресурса на диске, или потери соединения с сервером по сети. Причинами исключений при выполнении программы также могут быть ошибки программирования или неправильное использование API. В отличие от нашего мира, программа должна четко знать, как поступать в такой ситуации. Для этого в Java предусмотрен механизм исключений.

Обработка исключений в Java основана на использовании в программе следующих ключевых слов:

* try – определяет блок кода, в котором может произойти исключение;
* catch – определяет блок кода, в котором происходит обработка исключения;
* finally – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций: try{}catch, try{}catch{}finally, try{}finally{}.

* throw – используется для возбуждения исключения;
* throws – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.

Когда вы не планируете обрабатывать исключение в своем методе, но хотите предупредить пользователей метода о возможных исключительных ситуациях — используйте ключевое слово throws. Это ключевое слово в сигнатуре метода означает, что при определенных условиях метод, может выбросить исключение. Такое предупреждение является частью интерфейса метода и предоставляет право пользователю на собственный вариант реализации обработчика исключения. После throws мы указываем тип выбрасываемого исключения. Обычно это наследники класса Exception Java. Поскольку Java является объектно-ориентированным языком, все исключения в Java представляют собой объекты.

**7)** **Интерфейс** — это дитя Абстракции и Полиморфизма. Интерфейс очень напоминает абстрактный класс, у которого все методы абстрактные. Он объявляется так же, как и класс, только используется ключевое слово interface.

**interface** Кошачьи

{

**void** мурчать();

**void** мяукать();

**void** рычать();

}

Вот несколько полезных фактов об интерфейсах:

**1. Объявление интерфейса**

**interface** Drawable

{

**void** draw();

}

**interface** HasValue

{

**int** getValue();

}

1. Вместо слова class пишем interface.
2. Содержит только абстрактные методы (слово abstract писать не нужно)
3. На самом деле у интерфейсов все методы – public

**2. Наследование интерфейсов**

Интерфейс может наследоваться только от интерфейсов. Зато родителей у интерфейса может быть много. Еще говорят, что в Java есть множественное наследование интерфейсов. Примеры:

**interface** Element **extends** Drawable, HasValue

{

**int** getX();

**int** getY();

}

**3. Наследование классов от интерфейсов**

Класс может наследоваться от нескольких интерфейсов (и только от одного класса). При этом используется ключевое слово implements. Пример:

**abstract** **class** ChessItem **implements** Drawable, HasValue

{

**private** **int** x, y, value;

**public** **int** getValue()

{

**return** value;

}

**public** **int** getX()

{

**return** x;

}

**public** **int** getY()

{

**return** y;

}

}

Класс ChessItem объявлен абстрактным: он реализовал все унаследованные методы, кроме draw. Т.е. класс ChessItem содержит один абстрактный метод — draw().

Технически между словами extends и implements нет никакой разницы: и то, и то — это наследование. Так было сделано, чтобы повысить читабельность кода. В английском языке принято говорить, что классы наследуются (extends), а интерфейсы реализуются (implements)

**4. Переменные**

И самое важное: в интерфейсах нельзя объявлять переменные (хотя статические можно).

## **8)** Инкапсуляция

Все мы изучали инкапсуляцию как сокрытие элементов данных и предоставление пользователям доступа к данным с помощью общедоступных методов. Мы это называем геттерами и сеттерами. А сейчас забудем об этом и найдем более простое определение. Инкапсуляция — это метод ограничения пользователя от прямого изменения элементов данных или переменных класса для поддержания целостности данных. Как мы это делаем? Мы ограничиваем доступ к переменным, переключая модификатор доступа на private и открывая публичные методы, которые можно использовать для доступа к данным. Давайте посмотрим на конкретные примеры ниже. Это поможет нам понять, как мы можем использовать инкапсуляцию для поддержания целостности данных. Без инкапсуляции:

/\*\*

\* @author thegeekyasian.com

\*/

**public** **class** Account {

**public** **double** balance;

**public** **static** **void** main(String[] args) {

Account theGeekyAsianAccount = **new** Account();

theGeekyAsianAccount.balance = -54;

}

}

В приведенном выше фрагменте кода метод main() обращается к переменной balance напрямую. Это позволяет пользователю установить любое двойное значение для переменной balance класса Account. Мы можем потерять целостность данных, позволив любому установить значение balance как любое недопустимое число, например -54 в данном случае. С инкапсуляцией:

/\*\*

\* @author thegeekyasian.com

\*/

**public** **class** Account {

**private** **double** balance;

**public** **void** setBalance(**double** balance) {

**if**(balance >= 0) { // Validating input data in order to maintain data integrity

**this**.balance = balance;

}

**throw** **new** IllegalArgumentException("Balance cannot be less than zero (0)");

}

**public** **static** **void** main(String[] args) {

Account theGeekyAsianAccount = **new** Account();

theGeekyAsianAccount.setBalance(1); // Valid input - Allowed

theGeekyAsianAccount.setBalance(-55); // Stops user and throws exception

}

}

### В этом коде мы ограничили доступ к переменной balance и добавили метод setBalance(), который дает возможность пользователям устанавливать значение balance для Account. Установщик проверяет предоставленное значение перед присвоением его переменной. Если значение меньше нуля, генерируется исключение. Это гарантирует, что целостность данных не будет нарушена. После объяснения приведенных выше примеров я надеюсь, что ценность инкапсуляции как одного из четырех столпов ООП очевидна.

### Все члены класса в языке Java - поля и методы - имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.

В Java используются следующие модификаторы доступа:

* **public**: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.
* **private**: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.
* **protected**: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах, даже если они находятся в других пакетах
* **Модификатор по умолчанию**. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете.

## **9)** Что такое конструкторы классов

Конструктор — это специальный метод, который имеет имя, совпадающее с именем класса, и вызывается при создании экземпляра объекта совместно с оператором new. Результатом работы этого метода всегда является экземпляр класса.

Следовательно, конструктор класса Cat называется Cat (). В результате работы этого конструктора будет создан новый объект класса Cat. Цель конструктора — правильно инициализировать объект перед его использованием.

Самый простой способ создать объект — это строка вида:

Cat murka = new Cat();

Рассмотрим порядок создания объекта. В этой строке выполняется три действия:

* Во-первых, задаётся переменная класса Cat под именем murka. Эта переменная ещё не определяет объект, она просто даёт возможность ссылаться на него.
* Во-вторых, создаётся физическая копия объекта, а ссылка на него присваивается переменной murka. Это с помощью оператора new. Операторnew динамически — во время выполнения программы — выделяет память для объекта и возвращает ссылку на него, которая представляет собой адрес области памяти.
* В-третьих, ссылка на объект сохраняется в переменной. За это отвечает оператор =.

**10)** Скорость!

Если вы просто хотите сохранить объекты и не слишком беспокоитесь о последующем поиске определенного объекта или об их порядке, то несортированный контейнер поможет быстро вставить ваши объекты.

Отсортированным контейнерам нужен какой-то способ сортировки (например, с помощью определяемой пользователем функции сравнения (...)) при вставке объекта. Это замедляет процесс вставки, но может облегчить вашу жизнь.

## **11)** Полиморфизм

Последний и самый важный из четырех столпов ООП — это полиморфизм. Полиморфизм означает “множество форм”. Судя по названию, это функция, которая позволяет выполнять действие несколькими или разными способами. Когда мы говорим о полиморфизме, обсуждать особо нечего, если мы не говорим о его типах. Существует два типа полиморфизма:

1. Перегрузка метода — статический полиморфизм (Static Binding).
2. Переопределение метода — динамический полиморфизм (Dynamic Binding).

Давайте обсудим каждый из этих типов и посмотрим, в чем между ними разница.

### Перегрузка метода — статический полиморфизм:

Перегрузка метода или статический полиморфизм, также известный как статическое связывание (Static Binding) или связывание во время компиляции, — это тип, в котором вызовы методов определяются во время компиляции. Перегрузка методов позволяет нам иметь несколько методов с одним и тем же именем, имеющих разные типы данных параметра, или разное количество параметров, или то и другое. Но вопрос в том, чем полезна перегрузка метода (или статический полиморфизм)?

12) *Д*воичные деревья – это структуры данных, состоящие из узлов, которые хранят значение, а также ссылку на свою левую и правую ветвь. Каждая ветвь, в свою очередь, является деревом. Узел, который находится в самой вершине дерева принято называть корнем (root), узлы, находящиеся в самом низу дерева и не имеющие потомков называют листьями (leaves). Ветви узла называют потомками (descendants). По отношению к своим потомкам узел является родителем (parent) или предком (ancestor). Также, развивая аналогию, имеются сестринские узлы (siblings – родные братья или сёстры) – узлы с общим родителем. Аналогично, у узла могут быть дяди (uncle nodes) и дедушки и бабушки (grandparent nodes). Такие названия помогают понимать различные алгоритмы.

**Двоичное дерево поиска**- это структура данных, которая позволяет поддерживать отсортированный список чисел.

* Двоичным (бинарным) деревом называется, потому что каждый узел дерева имеет максимально два дочерних элементов.
* Деревом поиска, потому что его можно использовать для поиска числа в O(log(n)) time (алгоритм с временной сложностью T(n) = O(log(n))(прим.ре

struct tnode {  
  int field;           // поле данных  
  struct tnode \*left;  // левый потомок  
  struct tnode \*right; // правый потомок  
};

**13)** Каждый узел двунаправленного (двусвязного) линейного списка (ДЛС) содержит два поля указателей — на следующий и на предыдущий узлы. Указатель на предыдущий узел корня списка содержит нулевое значение. Указатель на следующий узел последнего узла также содержит нулевое значение.

Узел ДЛС можно представить в виде структуры: 

struct list  
{  
  int field; // поле данных  
  struct list \*next; // указатель на следующий элемент  
  struct list \*prev; // указатель на предыдущий элемент  
};

Основные действия, производимые над узлами ДЛС:

* Инициализация списка
* Добавление узла в список
* Удаление узла из списка
* Удаление корня списка
* Вывод элементов списка
* Вывод элементов списка в обратном порядке
* Взаимообмен двух узлов списка

**14)** Каждый узел однонаправленного (односвязного) линейного списка (ОЛС) содержит одно поле указателя на следующий узел. Поле указателя последнего узла содержит нулевое значение (указывает на NULL).

Узел ОЛС можно представить в виде структуры 

struct list  
{  
  int field; // поле данных  
  struct list \*ptr; // указатель на следующий элемент  
};

Основные действия, производимые над элементами ОЛС:

* Инициализация списка
* Добавление узла в список
* Удаление узла из списка
* Удаление корня списка
* Вывод элементов списка
* Взаимообмен двух узлов списка

**15)** **Связный список** (linked list) — это структура данных, в которой элементы линейно упорядочены, но порядок определяется не номерами элементов (как в массивах), а указателями, входящих в состав элементов списка и указывают на следующий элемент. У списка должна быть «голова» (первый элемент) и «хвост» (последний элемент).

## Преимущества и недостатки по сравнению с массивами

В отличие от массивов, вставка и удаление элементов в Связный список производится за постоянное время (**О (1)**). Также значительным преимуществом связных списков является возможность легкого расширения: чтобы увеличить размер списка, нужно только добавить еще один элемент.

Недостатком связных списков необходимость проходить весь список, чтобы найти элемент (то есть время доступа к элементу списка = **О (n)**).

### Элемент связанного списка (узел)

Элементы связанного списка называют узлами. Каждый узел содержит данные, которые мы хотим сохранить, а также указатель на следующий элемент в списке или NULL, если этот элемент является последним.

В коде на картинке приведен код структуры, который реализует элемент связанного списка (и собственно список). Данные представлены числовым значением.

Иными словами, мы используем синтаксис структуры для создания нашего собственного типа данных для инкапсуляции узла.

**int n** — это данные, которые мы хотим сохранить в узле.  
**struct node \* next** — указатель на следующий узел в связанном списке.

Помните, что узел не будет **typedef-ed** до тех пор, пока все эти строки не будут выполнены. Таким образом, мы должны написать узел **struct** вместо простого узла, как перед фигурными скобками, так и внутри фигурных скобок. В самой последней строке мы предоставляем **node** для **typedef** как имя, которое мы хотим использовать для этого нового типа данных для остальной части нашей программы.

**16)** Модификатор **static в Java** напрямую связан с классом. Если поле статично, значит оно принадлежит классу, если метод статичный — аналогично: он принадлежит классу. Исходя из этого, можно обращаться к статическому методу или полю, используя имя класса. Например, если поле count статично в классе Counter, значит, вы можете обратиться к переменной запросом вида: Counter.count.

**Static** — модификатор, применяемый к полю, блоку, методу или внутреннему классу. Данный модификатор указывает на привязку субъекта к текущему классу.