**Добро пожаловать в Java!**

**Java** - это высокоуровневый современный язык программирования, разработанный Sun Microsystems в начале 1990-х годов и в настоящее время принадлежащий Oracle.  
  
Java является **независимой от разных платформ** , что означает, что вам нужно только один раз написать программу, чтобы запустить ее на нескольких разных платформах!  
Java является **портативной** , **надежной** и **динамичной** , с возможностью удовлетворения потребностей практически любого типа приложений.  
Java гарантирует, что вы сможете **писать один раз, запускать в любом месте** .

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Java**

Java работает на более чем **3 миллиардов** устройств.  
Java используется для разработки приложений для ОС **Android,** Google , различных настольных приложений, таких как медиа-плееры, антивирусные программы, веб-приложения, корпоративные приложения (например, банковские услуги) и многие другие!  
Изучайте, практикуйте, а затем присоединяйтесь к огромному сообществу разработчиков Java по всему миру!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Ваша первая Java-программа**

Начнем с создания простой программы, которая выводит на экран «Hello World».

**class** MyClass {  
public static void **main** (String [] args) {  
System.out.println("Hello World");  
}  
} [**Попробуйте сами**](https://code.sololearn.com/716/#java)

В Java каждая строка кода, которая может фактически запускаться, должна находиться внутри **класса** .  
В нашем примере мы назвали класс **MyClass** .Вы узнаете больше о классах в будущих модулях.  
  
В Java, каждое приложение имеет точку входа, или начальную точку, которая представляет собой метод , называемый **main. Наряду с основными ключевыми словами**   **публичный** и **статический**  также будет объяснено позже.  
Как итог:  
- Каждая программа на Java должна иметь **класс** .  
- Каждая программа Java начинается с **основного** метода.

**Основной метод (The main Method)**

Чтобы запустить нашу программу, **основной** метод должен быть идентичен этой сигнатуре: **public** **static** **void** **main (** **String** **[] args)**  
- **public** : любой может получить к нему доступ  
- **static** : метод может быть запущен без создания экземпляра класса, содержащего основной метод  
- **void** : метод не возвращает никакого значения  
- **main** : имя метода  
  
Например, следующий код объявляет метод , называемый **test,** который ничего не возвращает и не имеет параметров: void test()  
Параметры метода объявляются внутри круглых скобок, которые следуют за именем метода .  
Для **main** это массив строк, называемый **args** .Мы будем использовать его в следующем уроке, поэтому не волнуйтесь, если вы все это не понимаете сейчас.

**Точки с запятой в Java**

Вы можете передать другой текст в качестве параметра метода **println** для его печати.

class MyClass { public static void main(String[ ] args) { System.out.println("I am learning Java"); } } Попробуйте сами

В Java каждое выражение кода должно заканчиваться **точкой** с **запятой** .  
Помните: не используйте **точки с запятой** после объявлений методов и классов, которые следуют за телом, определенным с помощью фигурных скобок.

**Комментарии**

Цель включения комментариев в ваш код - объяснить, что делает код.  
Java поддерживает как одиночные, так и многострочные комментарии. Все символы, которые появляются в комментарии, игнорируются компилятором Java.  
  
**Однострочный** комментарий начинается с **двумя слэшами** и продолжается до тех пор, пока не достигнет конца строки.  
Например: **// это однострочный комментарий**  
x = 5; **// однострочный комментарий после кода**  
Добавление комментариев при написании кода является хорошей практикой, потому что они дают разъяснения и понимание, когда вам нужно обратиться к нему, а также для других, которым может потребоваться его прочитать.

**Многострочные комментарии**

Java также поддерживает комментарии, которые охватывают несколько строк.  
Вы начинаете этот тип комментария с помощью косой черты с последующей звездочкой и заканчиваете ее звездочкой, за которой следует косая черта.   
Например: / \* Это также  
ограничение комментариев в  
несколько строк \* /  
Обратите внимание: Java не поддерживает вложенные многострочные комментарии.   
Тем не менее, вы можете вставлять однострочные комментарии в многострочные комментарии./ \* Это однострочный комментарий:  
  
// однострочный комментарий  
  
\* /

**Комментарии для документации**

**Комментарии к документации**   являются специальными комментариями, которые имеют внешний вид многострочных комментариев, с той лишь разницей, что они создают внешнюю документацию вашего исходного кода.Они начинаются с косой черты, а затем двумя звездочками и заканчиваются звездочкой, за которой следует косая черта.    
Например: / \*\* Это комментарий к документации \* /  
  
/ \*\* Это также  
комментарий к документации \* /  
**Javadoc** - это инструмент, который поставляется с JDK и используется для генерации документации на Java-код в формате HTML из исходного кода Java, который требует документации в предопределенном формате.  
  
Когда комментарий к документации начинается с более чем двух звездочек, Javadoc предполагает, что вы хотите создать «поле» вокруг комментария в исходном коде. Он просто игнорирует дополнительные звездочки.  
Например: / \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
Это начало метода  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* /  
Это сохранит только текст «Это начало метода » для документации.

**Переменные**

**Переменные** хранят данные для обработки.  
Переменной присваивается имя (или **идентификатор** ), например, область, возраст, высота и тому подобное.Имя однозначно идентифицирует каждую переменную, присваивая значение переменной и получая сохраненное значение.  
  
Переменные имеют **типы** .Некоторые примеры:  
- **int** : для целых чисел (целых чисел), таких как 123 и -456  
- **double** : для чисел с плавающей точкой или реального числа с необязательными десятичными точками и дробными частями в фиксированных или научных обозначениях, например 3.1416, -55.66.  
- **String** : для текстов, таких как "Привет" или "Доброе утро!". Текстовые строки заключены в двойные кавычки.  
  
Вы можете объявить переменную типа и присвоить ей значение. Пример: String name = "David";  
Это создает переменную, называемую **именем** типа **String** , и присваивает ей значение «Давид».  
Важно отметить, что переменная связана с типом и может только хранить значения этого конкретного типа. Например,   **Int**  переменная может хранить целочисленные значения, например 123; но он не может хранить реальные числа, такие как 12.34, или тексты, такие как "Привет".

**Переменные**

Примеры объявлений переменных:

class MyClass {  
public static void main (String [] args) {  
String name = "Давид";   
**int** age = 42;  
double score = 15,9;  
**char** group = 'Z';  
}  
} [**Попробуйте сами**](https://code.sololearn.com/718/#java)

**char** обозначает символ и имеет один символ.  
  
Другой тип - **Boolean (логический)**  тип, который имеет только два возможных значения: **true** и **false** .  
Этот тип данных используется для простых флагов, которые отслеживают истинные / ложные условия.  
**Например:** **boolean** online = true;  
Вы можете использовать список, разделенный запятыми, для объявления более чем одной переменной указанного типа. Пример: int a = 42, b = 11;

**Математические операторы**

Java предоставляет богатый набор операторов для управления переменными. Значение, используемое по обе стороны оператора, называется   **операнд** .  
Например, в приведенном ниже выражении числа 6 и 3 являются операндами оператора плюс: int x = 6 + 3;  
Java-арифметические операторы:   
+ **дополнение**  
- **вычитание**  
\* **умножение**  
/ **деление**  
% **по модулю**  
Арифметические операторы используются в математических выражениях так же, как они используются в алгебраических уравнениях.

**Прибавление**

Оператор + объединяет два значения, такие как две константы, константа и переменная, или переменная и переменная.Вот несколько примеров добавления: int sum1 = 50 + 10;  
int sum2 = sum1 + 66;  
int sum3 = sum2 + sum2;

**Вычитание**

Оператор - вычитает одно значение из другого. int sum1 = 1000 - 10;  
int sum2 = sum1 - 5;  
int sum3 = sum1 - sum2;

**Умножение**

Оператор \* умножает два значения. int sum1 = 1000 \* 2;  
int sum2 = sum1 \* 10;  
int sum3 = sum1 \* sum2;

**Деление**

Оператор / делит одно значение на другое. int sum1 = 1000/5;  
int sum2 = sum1 / 2;  
int sum3 = sum1 / sum2;  
В приведенном выше примере результатом уравнения деления будет целое число, так как **int**  используется как тип данных.Вы можете использовать   **double,** чтобы получить значение с десятичной точкой.

**Деление по модулю или остаток от деления**

Арифметическая операция по **модулю** (или остаток) выполняет целочисленное деление одного значения другим, и возвращает оставшуюся часть этого деления.  
Оператором операции по модулю является символ процента (%).  
Пример:

int value = 23;  
int res = значение% 6; // res is 5 [**Попробуйте сами**](https://code.sololearn.com/719/#java)

Деление 23 на 6 возвращает коэффициент 3, с остатком 5. Таким образом, значение 5 присваивается   **res** .

**Операторы инкремент**

Оператор **инкремента** или **декремента** обеспечивает более удобный и компактный способ увеличения или уменьшения значения переменной на **единицу.**  
Например, утверждение **x = x + 1;** может быть упрощена до **++ x;**    
Пример:

int test = 5;  
++ test; // test теперь 6

Оператор **декремента** (-) используется для уменьшения значения переменной на единицу.

int test = 5;  
--test; // test теперь 4

**Префикс и постфикс**

Две формы, **префикс** и **постфикс** могут использоваться как с операторами инкремента, так и с декрементом.  
С формой префикса оператор появляется перед операндом, а в постфиксной форме оператор появляется после операнда. Ниже приводится объяснение того, как работают две формы:  
**Префикс** : увеличивает значение переменной и использует новое значение в выражении.  
Пример:

int x = 34;  
int y = **++ x** ; // y is 35 [**Try It Yourself**](https://code.sololearn.com/722/#java)

Значение x сначала увеличивается до 35, а затем присваивается y, поэтому значения x и y теперь равны 35.  
**Postfix** : значение переменной сначала используется в выражении и затем увеличивается.  
Пример:

int x = 34;  
int y = **x ++** ; // y is 34 [**Try It Yourself**](https://code.sololearn.com/723/#java)

x сначала присваивается y, а затем увеличивается на единицу. Следовательно, x становится 35, а y присваивается значение 34.  
То же самое относится к оператору **декремента** .

**Операторы присваивания**

Вы уже знакомы с оператором **присваивания** (=), который присваивает значение переменной. int value = 5;  
Это присвоило значение 5 переменной, называемой **значением** типа **int** .  
  
Java предоставляет ряд операторов присваивания для упрощения написания кода.  
**Добавление и присвоение (+ =):**

int num1 = 4;  
int num2 = 8;  
**num2 + = num1** ; // num2 = num2 + num1;  
  
// num2 - 12, а num1 - 4. [**Попробуйте сами.**](https://code.sololearn.com/724/#java)

**Вычитание и присвоение(- =):**

int num1 = 4;  
int num2 = 8;  
**num2 - = num1;** // num2 = num2 - num1;  
  
// num2 - 4, а num1 - 4. [**Попробуйте сами.**](https://code.sololearn.com/725/#java)

Аналогично, Java поддерживает умножение и присвоение (\* =), деление и присвоение (/ =), а также остаток и присвоение (% =).

**Строки**

**Строка**  представляет собой объект, который представляет последовательность символов.  
Например, "Hello" - это строка из 5 символов.  
  
Например: **String** s = "SoloLearn";

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Конкатенация строк**

Оператор + (плюс) между строками добавляет их вместе, чтобы создать новую строку. Этот процесс называется   **конкатенация** .  
Приведенная строка - это первая строка, помещенная вместе со второй строкой.  
Например:

String firstName, lastName;  
firstName = "Давид";  
lastName = "Williams";  
  
System.out.println ("My name is " + firstName + "" + lastName);  
  
// Печатает: My name is David Williams [**Попробуй сам**](https://code.sololearn.com/726/#java)

Тип данных **char** представляет собой один символ.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Получение пользовательского ввода**

Хотя Java предоставляет множество различных методов для ввода данных пользователя, объект Scanner является наиболее распространенным и, возможно, самым простым в реализации. Импортируйте класс Scanner для использования объекта Scanner, как показано здесь:

import java.util.Scanner;

Чтобы использовать класс Scanner, создайте экземпляр класса, используя следующий синтаксис:

Scanner myVar = new Scanner(System.in);

Теперь вы можете читать разные входные данные, которые вводит пользователь.

Ниже приведены некоторые методы, доступные в классе Scanner:

Чтение байта - nextByte ()

Чтение short - nextShort ()

Чтение int - nextInt ()

Чтение long-nextLong ()

Чтение float - nextFloat ()

Чтение double - nextDouble ()

Чтение boolean - nextBoolean ()

Чтение полной строки - nextLine ()

Чтение слова - next ()

Пример программы, используемой для ввода пользователя:

import java.util.Scanner;

class MyClass {

  public static void main (String [] args) {

    Scanner myVar = new Scanner(System.in);

    System.out.println (myVar.nextLine ());

  }

}

Попробуй сам

Это будет ждать ввода пользователем чего-либо и печати этого ввода.

Код может показаться сложным, но вы поймете все это на предстоящих уроках.

ТЕМА Условные выражения и циклы

Условные выражения

**Принимать решение**

**Условные операторы** используются для выполнения различных действий на основе разных условий.  
Оператор **if** является одним из наиболее часто используемых условных операторов.  
Если выражение условия, истинно, то блок кода внутри **if** выражения выполняется.Если выражение найдено ложным, то выполняется первый набор кода после окончания   **if** (после закрытия фигурной скобки).  
**Синтаксис:** **if** (condition) {  
// Выполняется, когда условие истинно  
}  
Для формирования условия можно использовать любой из следующих операторов сравнения:  
**<** меньше  
**>** больше, чем  
**!=** не равно  
**==** равно  
**<=** меньше или равно  
**> =** больше или равно  
  
**Например:**

int x = 7;  
**if** (x <42) {  
System.out.println ( "Привет");  
} [**Попробуйте сами**](https://code.sololearn.com/727/#java)

Помните, что вам нужно использовать два равных знака ( **==** ) для проверки равенства, так как единственным символом равенства является оператор присваивания.

**if ... else выражение**

За оператором **if** может следовать необязательный оператор **else** , который выполняется, когда условие принимает значение false.  
**Например:**

int age = 30;  
  
**if** ((age <16) {  
System.out.println ("Too Young");  
} **else** {  
System.out.println ( "Welcome!");  
}  
//Вывод "Welcome" [**Попробуйте сами**](https://code.sololearn.com/728/#java)

Поскольку возраст равен 30, условие в выражении **if** вычисляется как false и выполняется инструкция **else** .

**Вложенные, if выражения**

Вы можете использовать один оператор **if-else** внутри другого оператора **if** или **else** .  
**Например:**

int age = 25;  
if(age > 0) {  
if(age > 16) {  
System.out.println("Welcome!");  
} else {  
System.out.println("Too Young");  
}  
} else {  
System.out.println("Error");  
}  
//Вывод "Welcome!"[**Try It Yourself**](https://code.sololearn.com/729/#java)

Вы можете вложить столько операторов **if-else,** сколько хотите.

**else if Statementments**

Вместо использования вложенных операторов **if-else** вы можете использовать оператор **else if** для проверки нескольких условий.  
**Например:**

int age = 25;  
  
**if**(age <= 0) {  
System.out.println("Error");  
} **else if**(age <= 16) {  
System.out.println("Too Young");  
} **else if**(age < 100) {  
System.out.println("Welcome!«);  
} **else** {  
System.out.println("Really?«);  
}  
//Вывод "Welcome! [**Попробуйте сами**](https://code.sololearn.com/730/#java)

Код проверяет условие, чтобы оценить значение true и выполнить инструкции внутри этого блока.  
Вы можете включить столько **аргументов,** сколько нужно.

**Логические операторы**

Логические операторы используются для объединения нескольких условий.  
  
Предположим, вы хотели, чтобы ваша программа выводила "Welcome!"только тогда, когда переменная   **age** больше 18, а переменная **money** больше 500.  
Один из способов добиться этого - использовать вложенные операторы **if** :

if (age > 18) {  
if (money > 500) {  
System.out.println("Welcome!");  
}  
} [**Попробуйте сами**](https://code.sololearn.com/731/#java)

Однако использование логического оператора **AND** ( **&&** ) является лучшим способом:

if (age > 18 **&&** money > 500) {  
System.out.println("Welcome!");  
} [**Попробуйте сами**](https://code.sololearn.com/732/#java)

Если оба операнда оператора AND истинны, то условие становится истинным.

**Оператор OR (ИЛИ)**

Оператор **OR** (||) проверяет, выполнено ли какое-либо из условий.  
Условие становится истинным, если какой-либо из операндов оценивается как true.  
**Например:**

int age = 25;  
int money = 100;  
  
if (age > 18 **||** money > 500) {  
System.out.println("Welcome!«);  
}  
//Вывод "Welcome!" [**Попробуйте сами**](https://code.sololearn.com/733/#java)

Вышеприведенный код печатает "Welcome!" если age больше 18   **или** если money превышает 500.  
  
**НЕ** **(!)**- этот логический оператор используется для изменения логического состояния его операнда. Если условие истинно,   логический оператор НЕ (!) сделает его ложным.  
**Пример:**

int age = 25;  
if(**!(age > 18)**) {  
System.out.println("Too Young");  
} else {  
System.out.println("Welcome");  
}  
//Вывод"Welcome"[**Try It Yourself**](https://code.sololearn.com/735/#java)

!(age > 18) читается как "если age НЕ больше чем 18".

**Выражение switch**

Оператор **switch** проверяет переменную для равенства в отношении списка значений.Каждое значение называется   **case** , и переменная, в switch , проверяется для каждого случая (case).  
**Синтаксис:** **switch** (выражение) {  
**case** значение1:  
//выражение  
**break;** //необязательный  
**case** значение2:  
//выражение  
**break;** //необязательный  
// У вас может быть любое количество операторов case.  
**default** : // Необязательный  
//выражение  
}  
- Когда переменная, в switch, равна **case** , утверждения, следующие за этим **case,** будут выполняться до тех пор, пока не будет достигнут оператор **break** .  
- Когда оператор **break** достигнут, **switch** завершается, и поток управления переходит к следующей строке после оператора **switch** .  
- Не каждый **случай** должен содержать **break.** .Если нет   **break.** , то поток контроля будет проходить по следующим case, пока не будет достигнут **break.** .  
  
Пример ниже проверяет **day** на набор значений и печатает соответствующее сообщение.

int day = 3;  
  
**switch**(day) {  
**case** 1:  
System.out.println("Monday");  
break;  
**case** 2:  
System.out.println("Tuesday");  
break;  
**case** 3:  
System.out.println("Wednesday");  
break;  
}  
// Outputs "Wednesday"[**Try It Yourself**](https://code.sololearn.com/736/#java)

Вы можете иметь любое количество операторов **case** в **switch** . За каждым   **case** следует сравнительное значение и двоеточие.

**The default Statement**

Оператор switch может иметь необязательный default case. .  
Default case может быть использован для выполнения задания , когда ни один из случаев не совпадают.  
  
**Например:**

int day = 3;  
  
switch(day) {  
case 6:  
System.out.println("Saturday");  
break;  
case 7:  
System.out.println("Sunday");  
break;  
**default**:  
System.out.println("Weekday");  
}  
// Outputs "Weekday"[**Try It Yourself**](https://code.sololearn.com/737/#java)

Никакого break не требуется в default case , так как он всегда является последним в switch.

**while циклы**

Выражение **цикла** позволяет повторно выполнить выражение или группу выражений.  
  
Цикл while повторно выполняет целевое выражение, пока заданное условие истинно.  
  
**Пример:**

int x = 3;  
  
**while**(x > 0) {  
System.out.println (х);  
x--;  
}  
/ \*   
Outputs  
3  
2  
1  
\* / [**Попробуйте сами**](https://code.sololearn.com/738/#java)

Цикл while проверяет условие х> 0.Если он оценивает значение как true,то он выполняет выражения внутри своего тела. Затем он снова проверяет выражение и повторяет.  
Обратите внимание на выражение x--. Это уменьшает x каждый раз на единицу, когда цикл работает, и останавливает цикл, когда x достигает 0.    
Без выражения х<0 цикл будет работать вечно.

**While циклы**

Когда выражение проверено и результат является ложным, тело цикла пропускается и выполняется первый оператор после цикла while.  
  
**Пример:**

int x = 6;  
  
**while**( x < 10 )  
{  
System.out.println(x);  
x++;  
}  
System.out.println("Loop ended");  
  
/ \*  
6  
7  
8  
9  
Loop ended  
\* /

**Цикл for**

Другая структура по циклам - цикл **for** .Цикл for позволяет эффективно писать цикл, который необходимо выполнить определенное количество раз.  
**Синтаксис:** **for (инициализация, условие, инкремент / декремент) {**  
выражения)  
}  
**Инициализация** : выражение выполняется только один раз в начале цикла  
**Условие** : оценивается каждый раз, когда цикл повторяется.Цикл выполняет инструкцию повторно, пока это условие не вернет false.  
**Increment / Decrement** : Выполняется после каждой итерации цикла.  
  
Следующий пример печатает цифры с 1 по 5.

for(int x = 1; x <=5; x++) {  
System.out.println(x);  
}  
  
/\* Outputs  
1  
2  
3  
4  
5  
\* / [**Попробуйте сами**](https://code.sololearn.com/740/#java)

Это инициализирует x значением 1 и многократно печатает значение x, пока условие x <= 5 не станет ложным. На каждой итерации выполняется инструкция x ++, увеличивая x на единицу.  
Обратите внимание на точку с запятой (;) после инициализации и условия в синтаксисе.

**Цикл for**

В цикле for вы можете иметь любой тип условия и любые инструкции инкремента.  
В приведенном ниже примере печатаются только четные значения от 0 до 10:

**for**(int x=0; x<=10; x=x+2) {  
System.out.println (х);  
}  
/ \*  
0  
2  
4  
6  
8  
10  
\* / [**Попробуйте сами**](https://code.sololearn.com/741/#java)

Цикл **for** лучше всего использовать, когда известны начальные и конечные числа.

**do ... while циклы**

Аdo...while цикл похож на **while** цикл, за исключением того, что **do...while** цикл гарантированно выполниться по крайней мере один раз.  
**Пример:**

int x = 1;  
**do** {  
System.out.println (х);  
х ++;  
} **while** (x <5);  
  
/ \*  
1  
2  
3  
4  
\* / [**Попробуйте сами**](https://code.sololearn.com/742/#java)

Обратите внимание, что условие появляется в конце цикла, поэтому операторы цикла выполняются один раз перед его тестированием.  
Даже с ложным условием код будет работать один раз. Пример:

int x = 1;  
**do** {  
System.out.println (х);  
х ++;  
} **while** (x <0);  
  
//Outputs 1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Выражения управления циклом**

Операторы **break** и **continue** изменяют поток выполнения цикла.  
Оператор **break** завершает цикл и передает выполнение оператору сразу после цикла.  
**Пример:**

int x = 1;  
  
while(x > 0) {  
System.out.println(x);  
if(x == 4) {  
**break;**  
}  
x++;  
}  
  
/\* Outputs  
1  
2  
3  
4  
\* / [**Попробуйте сами**](https://code.sololearn.com/744/#java)

Оператор **continue** заставляет цикл пропускать оставшуюся часть своего тела, а затем сразу же повторять его условие до повторного повторения.Другими словами, он заставляет цикл пропускать свою следующую итерацию.  
**Пример:**

for(int x=10; x<=40; x=x+10) {  
if(x == 30) {  
**continue**;  
}  
System.out.println(x);  
}  
/ \* Вывод  
10  
20  
40  
\* / [**Попробуйте сами**](https://code.sololearn.com/745/#java)

Как вы можете видеть, приведенный выше код пропускает значение 30, как указано в инструкции **continue** .

**Массивы**

**Массив**  представляет собой набор переменных того же типа.  
Когда вам нужно сохранить список значений, например числа, вы можете сохранить их в массиве , вместо объявления отдельных переменных для каждого числа.  
  
Чтобы объявить массив , вам нужно определить тип элементов с **квадратными скобками** .  
Например, для объявления массива целых чисел: **int** **[]** arr;  
Имя массива - **arr.** .Тип элементов, которые он будет содержать  это **int** .  
  
Теперь вам нужно определить емкость массива или количество элементов, которые он будет удерживать.Для этого используйте ключевое слово   **new** . int [] arr = **new** int [5];  
Вышеприведенный код объявляет массив из 5 целых чисел.(интеджеров)  
В массиве элементы упорядочиваются, и каждый из них имеет конкретную и постоянную позицию, которая называется **индексом** .  
  
Чтобы ссылаться на элементы в массиве , введите имя массива, за которым следует позиция индекса в паре квадратных скобок.  
**Пример:** **arr [2]** = 42;  
Это присваивает элементу значение, равное 42, а в качестве индекса - 2.  
Обратите внимание, что элементы в массиве идентифицируются с **нулевым**   индексом, что означает, что индекс первого элемента равен 0, а не единице.Таким образом, максимальный индекс   массива из пяти элементов int [5] это 4.

**Длина массива (Array Length)**

Вы можете получить доступ к длине массива (количество сохраняемых элементов) через его свойство **length** .  
**Пример:**

int [] intArr = new int [5];  
System.out.println ( **intArr.length** );  
  
// Вывод 5

**Массивы**

Теперь, когда мы знаем, как устанавливать и получать элементы массива , мы можем вычислить сумму всех элементов в массиве с помощью циклов.  
Цикл for является наиболее часто используемым циклом при работе с массивами, так как мы можем использовать **длину** массива , чтобы определить , сколько раз , пробегать по циклу.

int [] myArr = {6, 42, 3, 7};  
int sum = 0;  
for ( int x = 0; x <myArr.length; x ++) {  
sum + = myArr [x];  
}  
System.out.println(sum);  
  
// 58 [**Попробуйте сами**](https://code.sololearn.com/748/#java)

В приведенном выше коде мы объявили переменную **sum** для хранения результата и присвоили ему 0.  
Затем мы использовали цикл **for** для итерации по массиву и добавили значение каждого элемента к переменной.  
Условие цикла **for** равно x <myArr.length, поскольку индекс последнего элемента - **myArr.length-1** .

**Улучшеный цикл for**

**Усовершенствованный цикл for** (иногда называется « цикл for each») используются для обхода элементов в массивах.(он пробегается по всему массиву)  
Преимущества заключаются в том, что он устраняет возможность ошибок и упрощает чтение кода.  
**Пример:**

int [] primes = {2, 3, 5, 7};  
  
**for (int t: primes)** {  
System.out.println(t);   
}  
  
/ \*  
2  
3  
5  
7  
\* / [**Попробуйте сами**](https://code.sololearn.com/749/#java)

**Усовершенствованный цикл for** объявляет переменную типа , совместимого с элементами массива к которому осуществляется доступ.Переменная будет доступна в пределах   **for** блока, и его значение будет таким же, как текущий элемент массива .  
Таким образом, на каждой итерации цикла переменная **t** будет равна соответствующему элементу в массиве .  
Обратите внимание на **двоеточие** после переменной в синтаксисе.

**Многомерные массивы**

**Многомерные** массивы - массивы   которые содержат другие массивы.Двумерный   массив (array) - это самый простой многомерный массив .  
Чтобы создать многомерные массивы, поместите каждый массив   в своем собственном наборе квадратных скобок.Пример двумерного   массива : **int** **[] []** sample = {{1, 2, 3}, {4, 5, 6}};  
Это объявляет массив с двумя массивами в качестве его элементов.  
Чтобы получить доступ к элементу в двумерном массиве , укажите два индекса: один для массива и другой для элемента внутри этого массива .  
Следующий пример обращается к первому элементу во втором массиве выборки.

int x = sample **[1] [0]** ;  
System.out.println (х);  
  
// Вывод 4 [**Попробуйте сами**](https://code.sololearn.com/750/#java)

Два индекса массива называются **индекс строки** и **индекс столбца.**

**Многомерные массивы**

Вы можете получить и установить элементы многомерного массива , используя ту же пару квадратных скобок.  
**Пример:**

int [] [] myArr = {{1, 2, 3}, {4}, {5, 6, 7}};  
myArr [0] [2] = 42;  
int x = myArr [1] [0]; // 4 [**Попробуйте сами**](https://code.sololearn.com/751/#java)

Вышеупомянутый двумерный массив   содержит три массива.Первый   массив имеет три элемента, второй - один элемент, а последний из них имеет три элемента.  
В Java вы не ограничены только двумерными массивами. Массивы могут встраиваться в массивы на столько уровней, сколько требуется вашей программе. Все, что вам нужно, объявить   массив   с более чем двумя измерениями, то есть суть состоит в том, чтобы добавить столько наборов пустых скобок, сколько вам нужно.Однако их сложнее поддерживать.  
Помните, что все элементы массива должны быть одного типа.

Классы и объекты

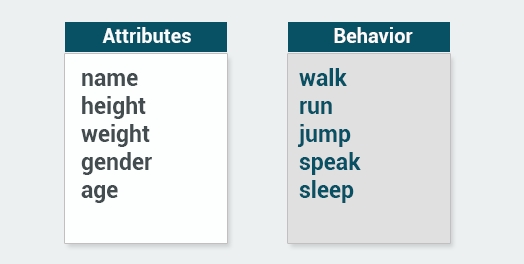
Объектно-ориентированное программирование

**Объектная ориентация**

Java использует Обьектно -Ориентированное программирование (OOP), стиль программирования, который предназначен для того, чтобы думать о программировании как о реальном мире.   
В ООП каждый объект является независимым юнитом с **уникальной идентичностью** , точно так же, как объекты в реальном мире.  
Яблоко - это объект; также как и кружка. Каждый имеет свою уникальную   **идентичность** .Возможно, что две кружки выглядят одинаково, но они все еще являются отдельными уникальными объектами.  
Объекты также имеют **характеристики** , которые используются для их описания.  
Например, автомобиль может быть красным или синим, кружка может быть полной или пустой, и так далее. Эти характеристики также называются   **атрибуты** .Атрибут описывает текущее состояние объекта.  
В реальном мире каждый объект ведет себя по-своему. Машина движется, телефон звонит и так далее.  
То же самое относится к объектам: **поведение** специфично для типа объекта.  
Таким образом, при объектно-ориентированном программировании каждый объект имеет три измерения: **идентификатор** , **атрибуты** и **поведение** .  
Атрибуты описывают текущее состояние объекта, и то, что объект способен сделать, демонстрируется через поведение объекта.

**Классы**

**Класс** описывает то , чем будет объект , но отдельно от самого объекта.  
Другими словами, классы могут быть описаны как чертежи, описания или определения для объекта. Вы можете использовать тот же класс, что и проект для создания нескольких объектов. Первый шаг - определить класс, который затем станет чертежом для создания объекта.  
  
Каждый класс имеет имя, и каждый используется для определения **атрибутов** и **поведения** .  
Некоторые примеры атрибутов и поведения:



**Методы**

Методы определяют **поведение** .  Метод   представляет собой набор операторов, которые сгруппированы для выполнения операции.System.out.println () является примером   метода .  
Вы можете определить свои собственные методы для выполнения ваших желаемых задач.   
Рассмотрим следующий код:

class MyClass {  
  
static void **sayHello()** {  
System.out.println("Hello World!");  
}  
  
public static void main(String[ ] args) {  
**sayHello();**  
}  
}  
// Вывод "Hello World!" [**Попробуйте сами**](https://code.sololearn.com/752/#java)

В приведенном выше коде объявляется метод "sayHello", который печатает текст, а затем вызывается в **main** .  
Чтобы вызвать метод , введите его имя и затем следуйте имени с помощью набора круглых скобок.

**Вызов методов**

Вы можете вызвать метод столько раз, сколько необходимо.  
Когда метод запускается, код перескакивает вниз до того места, где определяется метод , выполняет код внутри него, затем возвращается и переходит к следующей строке.  
**Пример:**

class MyClass {  
  
static void sayHello() {  
System.out.println("Hello World!");  
}  
  
public static void main(String[ ] args) {  
sayHello();  
sayHello();  
sayHello();  
}  
}  
  
// Hello World!  
// Hello World!  
// Hello World!

**Параметры метода**

Вы также можете создать метод, который принимает некоторые данные, называемые **параметрами** , вместе с ним, когда вы его вызываете.Пишите параметры в пределах   круглых скобок методов .  
Например, мы можем изменить наш метод **sayHello** () чтобы принимать и выводить  параметр типа String.

class MyClass {  
  
static void sayHello(**String name**) {  
System.out.println("Hello " + **name**);  
}  
  
public static void main(String[ ] args) {  
sayHello("David");  
sayHello("Amy");  
}  
  
}  
// Hello David  
// Hello Amy[**Try It Yourself**](https://code.sololearn.com/754/#java)

В приведенном выше методе в качестве параметра используется строка, называемая **name** , которая используется в теле метода .Затем, когда вызывается   метод , то мы передаем значение параметра в круглых скобках.  
Методы могут принимать несколько параметров, разделенных запятыми.  
Преимущества использования методов вместо простых операторов включают следующее:  
- **повторное использование кода** : вы можете написать метод один раз и использовать его несколько раз, не переписывая код каждый раз.  
- **параметры** : на основе переданных параметров методы могут выполнять различные действия.

**Тип возврата (The Return Type)**

**Return** это ключевое слово которое можно использовать в методах чтобы вернуть значение.  
Например, мы могли бы определить метод с именем **sum,** который возвращает сумму двух своих параметров. static int sum(int val1, int val2) {  
**return** val1 + val2;  
}  
Обратите внимание, что в определении метода мы определили **тип возвращаемого значения** до того, как мы написали имя метода .Для нашего   метода sum , это **int** , поскольку он принимает два параметра типа **int**  и возвращает их сумму, которая также является **int** .  
**static**  это ключевое слово будет обсуждаться в следующем уроке.  
Теперь мы можем использовать метод в нашем main.

class MyClass {  
  
static int sum(int val1, int val2) {  
return val1 + val2;  
}  
  
public static void main(String[ ] args) {  
int x = **sum**(2, 5);  
System.out.println(x);  
}  
}  
// Вывод "7" [**Попробуйте сами**](https://code.sololearn.com/755/#java)

Когда метод возвращает значение, мы можем присвоить его переменной.  
Когда вам не нужно возвращать какое-либо значение из вашего метода , используйте ключевое слово **void** .  
Обратите внимание  на ключевое слово void в определении main метода - это означает, что main ничего не возвращает.

**Тип Return**

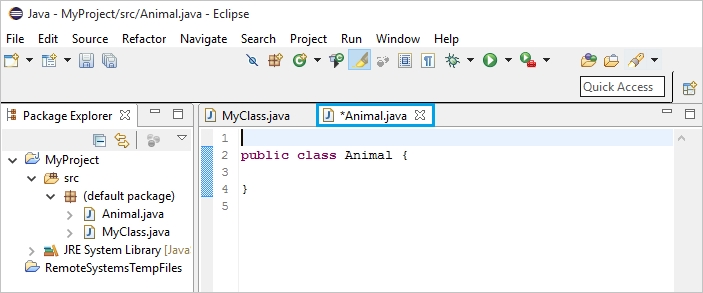
Взгляните на тот же код из нашего предыдущего урока с объяснением комментариев, чтобы вы могли лучше понять, как он работает: **// возвращает** **значение** **int** **5**  
static int returnFive() {  
return 5;  
}  
  
**// имеет параметр**  
static void sayHelloTo(String name) {  
System.out.println("Hello " + name);  
}  
  
**// просто печатает «Hello World!"**  
static void sayHello() {  
System.out.println("Hello World!");  
}  
Получив знания о типах и параметрах возвращаемых методов , давайте еще раз взглянем на определение main метода . public static void main(String[ ] args)  
Это определение указывает, что **основной** метод принимает массив строк как его параметры и не возвращает значение.

**Тип Return**

Давайте создадим метод, который принимает два параметра типа **int**  и возвращает больший параметр, затем вызовите его в **main** :

public static void main(String[ ] args) {  
int res = max(7, 42);  
System.out.println(res); //42  
}  
  
static **int** max(**int** a, **int** b) {  
if(a > b) {  
**return** a;  
}  
else {  
**return** b;  
}  
}

**Создание классов**

Чтобы создать свои собственные пользовательские объекты, вы должны сначала создать соответствующие классы. Это выполняется щелчком правой кнопки мыши по   **src** в Eclipse и выберите Create-> New-> Class.Дайте классу имя и нажмите   **Finish,** чтобы добавить новый класс в свой проект:   


Как вы можете видеть, Eclipse уже добавил исходный код для класса.  
Теперь давайте создадим простой метод в нашем новом классе.  
**Animal.java**

public class Animal {  
void bark() {  
System.out.println("Woof-Woof");  
}  
}  
Мы объявили **()**   метод bark() в нашем классе **Animal** .  
Теперь, чтобы использовать класс и его методы, нам нужно объявить **объект** этого класса.

**Создание объектов**

Давайте перейдем к нашему методу **main** и создадим новый объект нашего класса.  
**MyClass.java**

class MyClass {  
public static void main(String[ ] args) {  
**Animal dog = new Animal();**  
**dog.bark();**  
}  
}  
// Вывод "Woof-Woof" [**Попробуйте сами**](https://code.sololearn.com/757/#java)

Теперь **dog** является объектом типа **Animal** .Таким образом, мы можем вызвать его   () метод bark(), используя имя объекта и **точки.**  
**Точечная** нотация используется для доступа к **атрибутам** и **методам** объекта.  
Вы только что создали свой первый объект!

**Определение атрибутов**

Класс имеет **атрибуты** и **методы** .Атрибуты являются в основном переменными внутри класса.  
Давайте создадим класс под названием **Vehicle** , с его соответствующими атрибутами и методами. public class Vehicle {  
int maxSpeed;  
int wheels;  
String color;  
double fuelCapacity;   
  
void horn() {  
System.out.println("Beep!");  
}   
}  
**maxSpeed, wheels, color, and fuelCapacity** - атрибуты нашего класса Vehicle, а метод **horn ()** - единственный метод .  
Вы можете определить сколько хотите атрибутов и методов.

**Создание объектов**

Затем мы можем создать несколько объектов нашего класса **Vehicle** и использовать точечный синтаксис для доступа к их атрибутам и методам.

class MyClass {  
public static void main(String[ ] args) {  
Vehicle v1 = new Vehicle();  
Vehicle v2 = new Vehicle();  
v1.color = "red";  
v2.horn();  
}  
}

**Модификаторы доступа**

Теперь давайте обсудим ключевое слово **public** перед main методом . public static void main(String[ ] args)  
**public** - это **модификатор доступа** , что означает, что он используется для установки уровня доступа.Вы можете использовать модификаторы доступа для классов, атрибутов и методов.  
  
Для классов доступные модификаторы являются public или default (оставлены пустыми), как описано ниже:  
**public** : класс доступен любому другому классу.  
**default** : класс доступен только для классов в одном пакете .  
  
Для атрибутов и методов доступны следующие варианты:  
**default** : переменная или метод, объявленные без модификатора управления доступом, доступны для любого другого класса в одном пакете .  
**public** : Доступен из любого другого класса.  
**protected** : предоставляет тот же доступ, что и модификатор доступа default (по умолчанию), с добавлением того, что подклассы могут получить доступ к защищенным методам и переменным суперкласса (подклассы и суперклассы рассматриваются в будущих уроках).  
**private** : Доступно только внутри самого объявленного класса.  
  
**Пример:** public class Vehicle {  
**private** int maxSpeed;  
**private** int wheels;  
**private** String color;  
**private** double fuelCapacity;  
  
**public** void horn() {  
System.out.println("Beep!«);  
}  
}  
Рекомендуется хранить переменные внутри класса private. Переменные доступны и изменены с помощью   **Геттеров** и **сеттеров** .  
Нажмите « **Продолжить»,** чтобы узнать о Getters and Setters.

**Getters & Setters**

**Геттеры** и **Сеттеры** используются для эффективной защиты ваших данных, особенно при создании классов.Для каждой переменной   **get** метод возвращает свое значение, а метод **set** устанавливает значение.  
  
**Геттеры** начинаются с **get** , за которым следует имя переменной, с первой буквой name переменной с заглавной буквы.  
**Сеттеры** начинаются с **set** , за которым следует имя переменной, с первой буквой переменной name с заглавной буквы.  
  
**Пример:** public class Vehicle {  
private String color;  
  
// Геттер  
public String **getColor**() {  
return color;  
}  
  
// Сеттер  
public void **setColor**(String c) {  
this.color = c;  
}  
}  
 Метод get возвращает значение атрибута.  
 Метод set принимает параметр и присваивает его атрибуту.  
Ключевое слово **this** используется для ссылки на текущий объект.В основном,   **this.color** - это атрибут **цвета** текущего объекта.

**Getters & Setters**

Как только наш геттер и сеттер были определены, мы можем использовать его в нашем **main** :

public static void main(String[ ] args) {  
Vehicle v1 = new Vehicle();  
v1.setColor("Red");  
System.out.println(v1.getColor());  
}  
  
// Вывод "Red" [**Попробуйте сами**](https://code.sololearn.com/759/#java)

Геттеры и сеттеры позволяют нам контролировать значения. Вы можете, например, подтвердить данное значение в   сеттере, прежде чем устанавливать значение.  
Getters и seters являются фундаментальными строительными блоками для **инкапсуляции** , которые будут рассмотрены в следующем модуле.

**Конструкторы**

**Конструкторы** - это специальные методы, вызываемые при создании объекта и используемые для их инициализации.  
Конструктор может быть использован для обеспечения начальных значений для атрибутов объектов.  
  
- Имя конструктора должно совпадать с именем его класса.  
- Конструктор не должен иметь явного типа return .  
  
Пример конструктора :

public class Vehicle {  
private String color;  
**Vehicle()** {  
color = "Red";  
}  
}  
Метод **Vehicle** () является конструктором нашего класса, поэтому всякий раз , когда объект этого класса создается, атрибут color будет установлен в "Red"  
Конструктор может также принимать параметры для инициализации атрибутов. public class Vehicle {  
private String color;  
**Vehicle(String c)** {  
color = c;  
}  
}

**Использование конструкторов**

Конструктор вызывается при создании объекта с помощью ключевого слова new .  
**Example:**public class MyClass {  
public static void main(String[ ] args) {  
Vehicle v = new **Vehicle("Blue")**;  
}  
}  
Это вызовет конструктор , который установит атрибут **цвета** "Blue".

**Конструкторы**

Один класс может иметь несколько конструкторов с различным числом параметров.  
Методы сеттеры внутри конструкторов могут быть использованы для установки значения атрибутов.  
  
**Пример:** public class Vehicle {  
private String color;  
  
Vehicle() {  
this.setColor("Red");  
}  
Vehicle(String c) {  
this.setColor(c);  
}  
  
// Сеттер  
public void setColor(String c) {  
this.color = c;  
}  
}  
В приведенном выше классе есть два конструктора: один без каких-либо параметров, устанавливающий атрибут цвета в значение по умолчанию "Red" и другой конструктор, который принимает параметр и присваивает его атрибуту.  
  
Теперь мы можем использовать конструкторы для создания объектов нашего класса.

// цвет будет «красным»  
Vehicle v1 = new Vehicle();  
  
// цвет будет "зеленым"  
Vehicle v2 = new Vehicle("Green"); [**Try It Yourself**](https://code.sololearn.com/760/#java)

Java автоматически предоставляет конструктор по умолчанию , поэтому все классы имеют конструктор , независимо от того, определен он или нет.

**Типы значений**

**Типы значений** являются базовыми типами и включают byte, short, int, long, float, double, boolean, и char.  
Эти типы данных сохраняют значения, присвоенные им в соответствующих ячейках памяти.  
Итак, когда вы передаете их в метод, вы в основном оперируете **значением** переменной, а не самой переменной.  
**Пример:**

public class MyClass {  
public static void main(String[ ] args) {  
**int x = 5;**  
addOneTo(**x**);  
System.out.println(**x**);   
}  
static void addOneTo(int num) {  
num = num + 1;  
}  
}  
// Вывод " **5** " [**Попробуйте сами**](https://code.sololearn.com/761/#java)

Метод из примера выше , принимает **значение** его параметра, поэтому исходная переменная не влияет и 5 остается в качестве своего значения.

**Ссылочные Типы**

Ссылочный Тип хранит ссылку (или адрес) в ячейке памяти, где хранятся соответствующие данные.  
Когда вы создаете объект с помощью конструктора , вы создаете ссылочную переменную.  
Например, подумайте о том, чтобы определить класс **Person** :

public class MyClass {  
public static void main(String[ ] args) {  
Person j;  
j = new Person("John");  
j.setAge(20);  
celebrateBirthday(j);  
System.out.println(j.getAge());  
}  
static void celebrateBirthday(Person p) {  
p.setAge(p.getAge() + 1);  
}  
}  
// Вывод " **21** "

Метод   **celebrateBirthday** принимает объект Person как свой параметр и увеличивает его атрибут.  
Поскольку **j** является ссылочным типом, метод влияет на сам объект и способен изменять фактическое значение его атрибута.  
**Массивы** и **строки** также являются ссылочными типами данных.

**Класс Math**

**JDK** определяет ряд полезных классов, один из которых является класс **Math,** который обеспечивает заранее определенные методы для математических операций.  
Вам не нужно создавать объект класса **Math** для его использования.Чтобы получить доступ к нему, просто введите   **Math.** и соответствующий метод .  
  
**Math.abs ()** возвращает абсолютное значение своего параметра.

int a = Math.abs (10); // 10  
int b = Math.abs (-20); // 20 [**Попробуйте сами**](https://code.sololearn.com/763/#java)

**Math.ceil ()** округляет значение с плавающей запятой до ближайшего   значения целого числа.Округленное значение возвращается как   **double** .

double c = Math.ceil (7.342); // 8.0 [**Попробуйте сами**](https://code.sololearn.com/764/#java)

Аналогично, **Math.floor ()** округляет значение с плавающей запятой до ближайшего целочисленного значения.

double f = Math.floor (7.343); // 7.0 [**Попробуйте сами**](https://code.sololearn.com/765/#java)

**Math.max ()** возвращает самый большой из своих параметров.

int m = Math.max (10, 20); // 20 [**Попробуйте сами**](https://code.sololearn.com/766/#java)

И наоборот, **Math.min ()** возвращает наименьший параметр.

int m = Math.min (10, 20); // 10 [**Попробуйте сами**](https://code.sololearn.com/767/#java)

**Math.pow ()** принимает два параметра и возвращает первый параметр, возведенный в степень второго параметра.

double p = Math.pow (2, 3); // 8.0 [**Попробуйте сами**](https://code.sololearn.com/768/#java)

В классе Math имеется ряд других методов, в том числе:  
**sqrt ()** для квадратного корня, **sin () для** синуса, **cos () для** косинуса и других.

**Static**

При объявлении переменной или метода как **static,** они принадлежит к классу, а не к конкретному экземпляру.Это означает, что только один   экземпляр **статического**  члена существует, даже если вы создаете несколько объектов класса, или если вы его не создаете.Он будет использоваться всеми объектами.  
**Пример:** public class Counter {  
public **static** int COUNT=0;  
Counter() {  
COUNT++;  
}  
}  
Переменная **COUNT** будет общим для всех объектов этого класса.  
Теперь мы можем создавать объекты нашего класса Counter в **main** и обращаться к статической переменной.

public class MyClass {  
public static void main(String[ ] args) {  
Counter c1 = new Counter();  
Counter c2 = new Counter();  
System.out.println(**Counter.COUNT** );  
}  
}  
// Вывод "2" [**Попробуйте сами**](https://code.sololearn.com/769/#java)

Вывод равен 2, поскольку переменная **COUNT** является статической   и каждый раз увеличивается , когда создается новый объект класса Counter.В приведенном выше коде мы создали 2 объекта.  
Вы также можете получить доступ к статической переменной с помощью любого объекта этого класса, такого как **c1.COUNT** .  
Общепринятой практикой является использование верхнего регистра при присвоении имен статической переменной, хотя это и не обязательно.

**Static**

Эта же концепция применяется к **static**  методам. public class Vehicle {  
public **static** void horn() {  
System.out.println("Beep");  
}  
}  
Теперь метод **horn** можно вызвать без создания объекта:

public class MyClass {  
public static void main(String[ ] args) {  
Vehicle.horn();  
}  
} [**Попробуйте сами**](https://code.sololearn.com/770/#java)

Другим примером статических методов являются методы класса **Math** , поэтому вы можете вызывать их без создания объекта **Math** .  
Кроме того, **main**   метод всегда должен быть **статичным** .

**final**

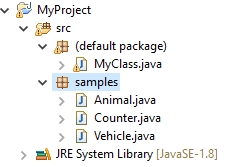
Используйте ключевое слово **final,** чтобы пометить константу переменной, чтобы ее можно было присвоить только один раз.  
**Пример:**

class MyClass {  
public static **final** double PI = 3.14;   
public static void main(String[ ] args) {  
System.out.println(PI);  
}  
} [**Попробуйте сами**](https://code.sololearn.com/771/#java)

**PI** теперь является константой.Любая попытка присвоить ему значение вызовет ошибку.  
Методы и классы также могут быть отмечены как **окончательные** .Это служит для ограничения методов, чтобы они не могли быть переопределены и чтобы классы не могли иметь подклассы.    
Эти концепции будут рассмотрены в следующем модуле.

**Пакеты**

**Пакеты** используются для предотвращения конфликтов имен и контроля доступа к классам.  
**Пакет**  может быть определен как группа, состоящая из аналогичных типов классов, а также подпакетов.  
Создать пакет   в Java довольно просто.Просто щелкните правой кнопкой мыши   **src** и щелкните New-> Package.Дайте своему пакету имя и нажмите Finish.  
Вы заметите, что новый пакет   появился в каталоге проекта.Теперь вы можете перемещать и создавать классы внутри   пакета .Мы переместили наши   КлассыVehicle, Counter и Animal в пакет   **samples.** .



Когда вы перемещаете / создаете класс в своем пакете , следующий код появится в верхней части списка files.package samples;  
Это указывает пакет, к которому принадлежит класс.  
Теперь нам нужно импортировать классы, которые находятся внутри пакета в нашем основном, чтобы иметь возможность использовать их.  
В следующем примере показано, как использовать класс **Vehicle** пакета **samples** . package.import samples.Vehicle;  
  
class MyClass {  
public static void main(String[ ] args) {  
Vehicle v1 = new Vehicle();  
v1.horn();  
}  
}  
Два основных результата возникают, когда класс помещается в пакет .Во-первых, название   пакет становится частью имени класса.Во-вторых, название   пакета должно соответствовать структуре каталогов, в которой находится соответствующий файл класса.  
Используйте шаблон для импорта всех классов в пакет .  
Например, **import samples.\*** импортирует все классы в samples package. .

**Инкапсуляция**

В ООП существует 4 основных понятия: **инкапсуляция** , **наследование** , **полиморфизм** и **абстракция** .  
  
Суть **инкапсуляции** **в том** чтобы гарантировать, что детали реализации не видны пользователям.Переменные одного класса будут скрыты от других классов и доступны только через методы текущего класса. Это называется   **сокрытие данных** .  
Для достижения инкапсуляции в Java, следует объявлять переменные как **private** и сделать **методы сеттеры и геттеры** public для изменения и просмотра значений переменных   
  
**For example:**class BankAccount {  
private double balance=0;  
public void deposit(double x) {  
if(x > 0) {  
balance += x;  
}  
}  
}  
Эта реализация скрывает переменную **balance** , предоставляя доступ к ней только через метод **deposit** , который проверяет сумму, подлежащую внесению до изменения переменной.  
В итоге, **инкапсуляция**  обеспечивает следующие преимущества:  
- Контроль доступа к данным или их изменения  
- Более гибкий и легко изменяемый код  
- Возможность изменить одну часть кода, не затрагивая другие части

**Наследование**

**Наследование** - это процесс, который позволяет одному классу приобретать свойства (методы и переменные) другого. С   наследованием , информация размещается в более управляемом, иерархическом порядке.  
  
Класс, наследующий свойства другого, является **подклассом** (также называемым производным классом или дочерним классом); класс, свойства которого наследуются, является **суперклассом** (базовый класс или родительский класс).  
  
Чтобы наследовать от класса, используйте **ключевое слово extends**  .  
В этом примере показано, как наследовать класс **Dog** от класса **Animal** .

class Dog extends Animal {

// some code

}   
  
Здесь Dog - это **подкласс** , а Animal - **суперкласс** .

**Наследование**

Когда один класс наследуется от другого класса, он наследует все переменные и методы суперкласса, которые не private  
**Example:**

class Animal {  
protected int legs;  
public void eat() {  
System.out.println("Animal eats");  
}  
}  
  
class Dog **extends** Animal {  
Dog() {  
**legs** = 4;  
}  
}  
Как вы можете видеть, класс Dog наследует переменную leg от класса Animal.  
Теперь мы можем объявить объект Dog и вызвать метод **eat** его суперкласса:

class MyClass {  
public static void main(String[ ] args) {  
Dog d = new Dog();  
**d.eat();**  
}  
}[**Try It Yourself**](https://code.sololearn.com/772/#java)

Вспомните модификатор **защищенного** доступа (protected), который делает элементы видимыми только для подклассов.

**Наследование**

Конструкторы не являются методами-членами и поэтому не наследуются подклассами.  
Однако конструктор суперкласса вызывается при создании экземпляра подкласса.  
**Пример:**

class A {  
public A() {  
System.out.println("New A");  
}  
}  
class B extends A {  
public B() {  
System.out.println("New B");  
}  
}  
  
class Program {  
public static void main(String[ ] args) {  
B obj = new B();  
}  
}  
  
/\*Outputs  
"New A"  
"New B"  
\*/[**Try It Yourself**](https://code.sololearn.com/773/#java)

Вы можете получить доступ к суперклассу из подкласса, используя ключевое слово **super** .  
Например, **super.var** обращается к члену суперкласса под названием var.

**Полиморфизм**

**Полиморфизм** , который относится к идее «иметь много форм», возникает, когда существует иерархия классов, связанных друг с другом посредством наследования .  
Вызов метода приведет к выполнению другой реализации, в зависимости от типа объекта, вызывающего метод .  
Вот пример: **Dog** и **Cat** - это классы, которые наследуются от класса **Animal** .Каждый класс имеет свою собственную реализацию метода  **makeSound** () .

class Animal {  
public void makeSound() {  
System.out.println("Grr...");  
}  
}  
class **Cat** extends Animal {  
public void **makeSound**() {  
System.out.println("Meow");  
}  
}  
class **Dog** extends Animal {  
public void **makeSound**() {  
System.out.println("Woof");  
}  
}  
Поскольку все объекты **Cat** и **Dog** являются объектами **Animal** , в **основном** мы можем сделать следующее :

public static void main ( String [] args) {  
**Animal** a = new **Dog**();  
**Animal** b = new **Cat**();  
}  
Мы создали две ссылочные переменные типа Animal и указали их на объекты **Cat** и **Dog** .  
Теперь мы можем вызвать методы makeSound ().

a.makeSound();  
//Outputs "**Woof**"  
  
b.makeSound();  
//Outputs "**Meow**"[**Try It Yourself**](https://code.sololearn.com/774/#java)

Поскольку ссылочная переменная **a** содержит объект Dog, будет вызван метод makeSound () класса Dog.  
То же самое относится и к переменной **b** .  
Это демонстрирует, что вы можете использовать переменную **Animal,** фактически не зная, что она содержит объект подкласса.  
Это очень полезно, когда у вас есть несколько подклассов суперкласса.

**Переопределение метода**

Как мы видели в предыдущем уроке, подкласс может определять поведение, специфичное для типа подкласса, что означает, что подкласс может реализовывать метод родительского класса на основе его требования.  
Эта функция известна как метод   **переопределения.**  
**Пример:**

class Animal {  
public void makeSound() {  
System.out.println("Grr...");  
}  
}  
class Cat extends Animal {  
**public void makeSound()** {  
System.out.println("Meow");  
}  
}[**Try It Yourself**](https://code.sololearn.com/775/#java)

В приведенном выше коде, класс Кошка переопределяет метод **makeSound** () своего суперкласса Animal.  
  
**Правила для переопределения метода:**  
- должен иметь **одинаковый** тип возвращаемого значения и аргументы  
- Уровень **доступа** не может быть более ограничительным, чем уровень доступа переопределенного метода (Пример: если метод суперкласса объявлен как public, переопределяющий метод в подклассе не может быть ни private, ни protected)  
- Метод, объявленный как **final** или **static**  не может быть переопределен  
- Если метод не может быть унаследован, он не может быть переопределен  
- Конструкторы не могут быть переопределены  
Переопределение метода также известно как **полиморфизм во** **время выполнения (runtime polymorphism.)** .

**Перегрузка метода**

Когда методы имеют одинаковое имя, но разные параметры, это называется методом   **перегрузки** .  
Это может быть очень полезно, когда вам нужны одинаковые функциональные возможности метода для разных типов параметров.  
В следующем примере показан метод, который возвращает максимум двух его параметров.

**int max(int a, int b)** {  
if(a > b) {  
return a;  
}  
else {  
return b;  
}  
}  
Способ , показанный выше , будет работать только для параметров типа **целого числа (integer.).**  
Однако, мы могли бы хотеть использовать это и для **doubles** .Для этого нужно перегрузить   метод max :

**double** max(**double** a, **double** b) {  
if(a > b) {  
return a;  
}  
else {  
return b;  
}  
}[**Try It Yourself**](https://code.sololearn.com/776/#java)

Теперь наш метод **max** также будет работать с типом **double** .  
Перегруженный метод   **должен** иметь другой список аргументов ; параметры должны отличаться по своему типу, количеству или и тем и тем.  
Другое название перегрузки методов - **полиморфизм во** **время компиляции (compile-time polymorphism)** .

**Абстракция**

**Абстракция** данных предоставляет внешнему миру только важную информацию в процессе представления основных функций без включения деталей реализации.  
Хороший пример из реальной жизни - *книга* .Когда вы слышите термин «книга», вы не знаете точных особенностей, таких как количество страниц, цвет или размер, но понимаете идею или абстракцию книги.  
Концепция **абстракции** заключается в том, что мы фокусируемся на основных качествах, а не на конкретных характеристиках одного конкретного примера.  
  
В Java абстракция достигается с помощью **абстрактных классов** и **интерфейсов** .  
Абстрактный класс определяется с помощью ключевого слова abstract.  
- Если класс объявлен как абстрактный, он не может быть создан (вы не можете создавать объекты этого типа).  
- Чтобы использовать абстрактный класс , вы должны наследовать его от другого класса.  
- Любой класс, который содержит абстрактный метод, должен быть определен как абстрактный.  
Абстрактный метод - это метод, который объявлен без реализации (без фигурных скобок и сопровождается точкой с запятой): **abstract** **void** **walk ();**

**Абстрактный класс**

Например, мы можем определить наш класс Animal как абстрактный:

abstract class Animal {  
int legs = 0;  
**abstract** void makeSound();  
}  
Метод makeSound также абстрактный, так как он не имеет реализации в суперклассе.  
Мы можем наследовать от класса Animal и определить метод makeSound () для подкласса:

class Cat extends Animal {  
public void makeSound() {  
System.out.println("Meow");  
}  
} [**Попробуйте сами**](https://code.sololearn.com/777/#java)

Каждое животное издает звук, но у каждого свой способ это делать. Вот почему мы определяем абстрактный класс Animal и оставляем реализацию того, как они издают звуки, подклассам.  
Это используется, когда нет никакого значимого определения для метода в суперклассе.

**Интерфейсы**

**Интерфейс**  является полностью абстрактным классом, который содержит только абстрактные методы.  
Некоторые спецификации для интерфейсов:  
- Определяется с помощью  ключевого слова interface .  
- Может содержать только статические ( static) конечные (final) переменные.  
- Не может содержать конструктор, потому что интерфейсы не могут быть созданы.  
- Интерфейсы могут расширять другие интерфейсы.  
- класс может реализовывать любое количество интерфейсов.  
  
Пример простого интерфейса : interface Animal {  
public void eat();  
public void makeSound();  
}  
Интерфейсы имеют следующие свойства:  
- интерфейс   неявно абстрактный.Вам не нужно использовать ключевое слово abstract при объявлении   интерфейса.  
- Каждый метод в интерфейсе также неявно абстрактный, поэтому ключевое слово abstract не требуется.  
- Методы в интерфейсе неявно public.  
Класс может наследовать только от **одного** суперкласса, но может реализовывать **несколько** интерфейсов!

**Интерфейсы**

Используйте ключевое слово implements , чтобы использовать интерфейс с классом.

interface Animal {  
public void eat();  
public void makeSound();  
}  
  
class Cat **implements** Animal {  
public void makeSound() {  
System.out.println("Meow");  
}  
public void eat() {  
System.out.println("omnomnom");  
}  
} [**Попробуйте сами**](https://code.sololearn.com/778/#java)

Когда вы реализуете интерфейс , вам необходимо переопределить все его методы.

**Приведение типа**

Присвоение значения одного типа переменной другого типа называется приведением **типа** .  
  
Чтобы привести значение к определенному типу, поместите тип в скобки и поместите его перед значением.  
**Пример:**

int a = **(int)** 3.14;  
System.out.println(a);  
//Outputs 3[**Try It Yourself**](https://code.sololearn.com/779/#java)

Приведенный выше код преобразует значение 3.14 в целое число с результирующим значением 3.  
**Другой пример:**

double a = 42.571;  
int b = **(int)** a;  
System.out.println(b);  
//Outputs 42[**Try It Yourself**](https://code.sololearn.com/780/#java)

Java поддерживает автоматическое приведение типов целых чисел к плавающим точкам, поскольку нет потери точности.  
С другой стороны, приведение типов обязательно при присвоении значений с плавающей точкой целочисленным переменным.

**Приведение типа**

Для классов есть два типа кастинга.

**Повышающее преобразование/ приведение к типу(Upcasting)**

Вы можете привести экземпляр подкласса к его суперклассу.  
Рассмотрим следующий пример, предполагая, что Cat является подклассом Animal.Animal a = new Cat();  
Java автоматически преобразовала переменную типа Cat в тип Animal.

**Понижающее приведение к типу (Downcasting)**

Преобразование объекта суперкласса в его подкласс называется **понижением** .  
Example:Animal a = new Animal();  
(**(Cat)**a).makeSound();  
Здесь пытаются привести переменную а к типу **Cat** и вызвать его метод makeSound ().  
Почему upcasting автоматический, downcasting ручной? Ну, upcasting никогда не может потерпеть неудачу(сработает всегда). Но если у вас есть группа разных животных и вы хотите превратить их всех в кошку, тогда есть шанс, что некоторые из этих животных на самом деле являются собаками, поэтому процесс завершится неудачно.

**Анонимные классы**

**Анонимные классы** - это способ расширить существующие классы на лету.  
Например, рассмотрим наличие класса Machine:

class Machine {  
public void start() {  
System.out.println("Starting...");  
}  
}  
При создании объекта Machine мы можем изменить метод start на лету.

public static void main(String[ ] args) {  
Machine m = new Machine() {  
@Override public void start() {  
System.out.println("Wooooo");  
}  
};  
m.start();  
}  
//Outputs "Wooooo";[**Try It Yourself**](https://code.sololearn.com/782/#java)

После вызова конструктора мы открыли фигурные скобки и на лету переопределили реализацию метода **start** .  
**Аннотация @Override** используется , чтобы сделать код легче для понимания, потому что это делает его более очевидным , когда методы переопределены.

**Анонимные классы**

Модификация применима только к текущему объекту, а не к самому классу. Так что, если мы создадим еще один объект этого класса,   то реализация метода start будет той, которая определена в классе.

class Machine {  
public void start() {  
System.out.println("Starting...");  
}  
}   
public static void main(String[ ] args) {  
Machine m1 = new Machine() {  
@Override public void start() {  
System.out.println("Wooooo");  
}  
};  
Machine m2 = new Machine();  
m2.start();  
}  
//Outputs "Starting..."[**Try It Yourself**](https://code.sololearn.com/783/#java)

Нажмите **Попробуйте сами,** чтобы поиграться с кодом!

**Внутренние классы (Inner Classes)**

Java поддерживает классы **вложенности** ; класс может быть членом другого класса.  
Создать внутренний класс довольно просто. Просто напишите класс в классе. В отличие от класса, внутренний класс может быть закрытым. Как только вы объявляете внутренний класс закрытым, к нему нельзя получить доступ из объекта вне класса.  
**Пример:**

class Robot {  
int id;  
Robot(int i) {  
id = i;  
Brain b = new Brain();  
b.think();  
}  
  
**private class Brain {**  
**public void think() {**  
**System.out.println(id + " is thinking");**  
**}**  
**}**  
  
} [**Попробуйте сами**](https://code.sololearn.com/784/#java)

Класс **Robot** имеет внутренний класс **Brain** .Внутренний класс может обращаться ко всем переменным-членам и методам своего внешнего класса, но к нему нельзя получить доступ ни из какого внешнего класса.

**Сравнение объектов**

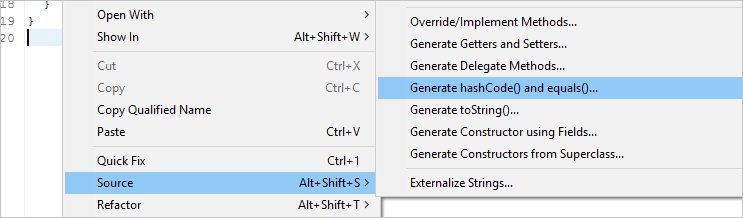
Помните, что при создании объектов переменные хранят ссылки на объекты.  
Таким образом, когда вы сравниваете объекты с помощью оператора проверки равенства ( **==** ), он фактически сравнивает ссылки, а не значения объектов.  
**Пример:**

class Animal {  
String name;  
Animal(String n) {  
name = n;  
}  
}  
  
class MyClass {  
public static void main(String[ ] args) {  
Animal a1 = new Animal("Robby");  
Animal a2 = new Animal("Robby");  
System.out.println(**a1 == a2**);  
}  
}  
//Outputs **false**[**Try It Yourself**](https://code.sololearn.com/785/#java)

Несмотря на наличие двух объектов с одинаковым именем, проверка на равенство возвращает false, потому что у нас есть два разных объекта (две разные ссылки или места в памяти).

**equals()**

Каждый объект имеет предопределенный метод **equals** (), который используется для проверки семантического равенства.  
Но чтобы заставить его работать для наших классов, нам нужно переопределить его и проверить условия, которые нам нужны.  
Существует простой и быстрый способ генерации метода equals () , кроме написания его вручную.  
Просто щелкните правой кнопкой мыши в своем классе, перейдите в **Source** -> **Generate hashCode () и equals () ...**



Это автоматически создаст необходимые методы.

class Animal {  
String name;  
Animal(String n) {  
name = n;  
}  
@Override  
public int hashCode() {  
final int prime = 31;  
int result = 1;  
result = prime \* result + ((name == null) ? 0 : name.hashCode());  
return result;  
}  
@Override  
public boolean **equals**(Object obj) {  
if (this == obj)  
return true;  
if (obj == null)  
return false;  
if (getClass() != obj.getClass())  
return false;  
Animal other = (Animal) obj;  
if (name == null) {  
if (other.name != null)  
return false;  
} else if (!name.equals(other.name))  
return false;  
return true;  
}  
}  
Автоматически сгенерированный метод hashCode ()   используется для определения, где хранить объект внутри.Всякий раз, когда вы реализуете   **equals** вы ДОЛЖНЫ также реализовать **hashCode** .  
Мы можем снова запустить тест, используя метод **equals** :

public static void main(String[ ] args) {  
Animal a1 = new Animal("Robby");  
Animal a2 = new Animal("Robby");  
System.out.println(a1.equals(a2));  
}  
//Outputs **true**[**Try It Yourself**](https://code.sololearn.com/786/#java)

Вы можете использовать это же меню для генерации других полезных методов, таких как методы **геттеры** и **сеттеры** для ваших атрибутов класса.

**Перечисления**

Enum - это специальный тип, используемый для определения коллекций констант.   
Вот простой пример Enum: **enum Rank {**   
SOLDIER,  
SERGEANT,  
CAPTAIN  
}  
Обратите внимание, что значения **разделены запятыми.**  
Вы можете ссылаться на константы в перечислении выше с помощью синтаксиса **точки** .Rank a = Rank.SOLDIER;   
В основном, Enums определяют переменные, которые представляют собой члены фиксированного набора.

**Перечисления**

После объявления Enum мы можем проверить соответствующие значения, например, с помощью оператора **switch** .

Rank a = Rank.SOLDIER;  
  
switch(a) {  
case SOLDIER:  
System.out.println("Soldier says hi!");  
break;  
case SERGEANT:  
System.out.println("Sergeant says Hello!");  
break;  
case CAPTAIN:  
System.out.println("Captain says Welcome!");  
break;  
}  
//Outputs "Soldier says hi! [**Попробуйте сами**](https://code.sololearn.com/787/#java)

Нажмите **Попробуйте сами,** чтобы поиграться с кодом!

**Перечисления**

Вы всегда должны использовать Enums, когда переменная (особенно параметр метода ) может принять только одно из небольшого набора возможных значений.  
Если вы используете Enums вместо целых чисел (или строковых кодов (String codes)), вы увеличиваете проверку во время компиляции и избегаете ошибок при передаче недопустимых констант, а также документируете, какие значения допустимы для использования.  
Некоторые примеры использования Enum включают названия месяцев, дни недели, колоду карт и т. д.

**Java API**

Java API - это набор классов и интерфейсов, которые были написаны для вас.  
Документацию по API Java со всеми доступными API можно найти на веб-сайте Oracle по адресу  
**http://docs.oracle.com/javase/7/docs/api/**  
Как только вы найдете пакет, который хотите использовать, вам нужно импортировать его в свой код.  
Пакет можно импортировать с помощью ключевого слова import .  
**Например:** import java.awt.\*;  
Пакет **AWT** содержит все классы для создания пользовательских интерфейсов и для рисования графики и изображений.  
Подстановочный знак (\*) используется для импорта всех классов в пакете .