1.7. Строки

> Привет!
> На связи шпаргалка урока 1.7. Строки. Здесь вы найдете дополнительную информацию по уроку и полезные советы по выполнению домашнего задания.
>
> **Время прочтения:** 17 минут.

Так как данные в основном передаются в виде текста (даже веб-сайты, которые вы видите в графическом представлении, «под капотом» являются строками в формате HTML), строки являются самым популярным типом данных в Java.

Создаются строки достаточно простым способом и отличаются от создания классических объектов (как массив, например) отсутствием необходимости в слове new.

```
String helloWorld = “Hello, world!”;
```

Чтобы объявить строку, нам нужно создать переменную типа String (да, именно с большой буквы, как и все остальные объекты в Java, кроме массивов).

Для ее инициализации необходимо после оператора присваивания (=) поместить двойные кавычки, а внутри текст.

Для создания пустой строки достаточно открыть и сразу же закрыть кавычки (“”).

Для того, чтобы «собрать» строку из нескольких частей, будь то другие строки или числа, мы используем оператор ‘+’.

```
String helloWorld = “Hello, ” + “world!”;
```

Мы создали новую строку, которая была сконструирована в результате слияния одной неизменяемой строки и другой неизменяемой строки.

В результате этого действия в памяти оказались три строки, две из которых будут удалены Java как неиспользуемые, а итоговая сохранится для дальнейшего использования.

При сложении строки с цифрами цифры тоже преобразуются в символы, а затем складываются со строкой.

```
String deviceModel = “Samsung Galaxy S” + 9;
```

В результате число будет преобразовано в символ, а затем будет создана новая строка на основе прошлой строки “Samsung Galaxy S” и строки “9”.

В результате в памяти снова окажется три строки, две из которых Java удалит.

Также строки, как и массивы, являются объектами, и если им не присвоить значение (инициализировать), они будут содержать в себе null. Любое обращение к неинициализированной строке приведет к моментальному возникновению NullPointerException и падению вашего приложения.

Строка — неизменяемый массив символов

Как мы с вами помним из урока с массивами, объекты могут содержать в себе описания (свойства) их характеристик и методы по работе с ними.

Строка, как и массив (в случае с переменной length), имеет внутренние свойства.

Главное из них — массив типа char, где и хранятся все символы конкретной строки.

Когда вы объявляете строку, данный массив заполняется посимвольно содержимым этой самой строки и именно в таком формате вашу строку видит (и хранит) компьютер.

В связи с этим у вас должен был возникнуть следующий **вопрос**:

как мы можем изменить строку, если длину массива менять нельзя?

Ответ: никак.

Строка является неизменяемым объектом.

Т. е. в те моменты, когда вы производите конкатенацию (складываете строки через оператор '+') или любое другое изменение, каждый раз создается новая строка, а не изменяется старая.

Именно поэтому складывать строки в цикле считается плохой практикой.

Представьте, что вам нужно создать строку, состоящую из сотен символов, которые вы каждый шаг цикла добавляете к текущей строке.

В этом случае вы создадите столько строк в памяти, сколько раз сработал цикл.

Неизменяемость строк обусловлена способом их хранения в памяти, а также соображениями безопасности.

Представьте, что ваша строка используется во многих местах приложения.

Она содержит логин, пароль, телефон или любую другую информацию, которая может быть идентичной в нескольких местах.

И вот случилась ситуация, когда эту строку нужно изменить в одном из мест.

Допустим, вы хотите изменить ваш email в деловых рассылках, а в личных оставить предыдущий.

В такой ситуации, будь строки изменяемыми, изменив в одном месте значение переменной email, вы бы изменили его везде. Именно потому, что изменение объекта строки в одном месте может сломать код в других местах, было принято решение запретить изменение строк и при необходимости подобных действий каждый раз создавать новую строку.

Методы для работы со строками

> 🧬 **Строка — ваш первый классический объект, работать с ним нужно специальными методами.**

Как мы помним из уроков с массивами, сравнивать объекты через оператор равенства (==) нельзя. Это обусловлено тем, что объекты не являются примитивными типами и хранят в себе ссылку на объект, а не само значение.

При использовании вышеупомянутого оператора сравниваться будут значения ссылок, а не сами объекты. А они будут равными только в том случае, когда ссылки и правда ссылаются на единый объект в памяти.

Если объект является абсолютной копией сравниваемого объекта, применения этого оператора вернет false.

Именно поэтому способы работы со строками отличаются от тех, что мы использовали для работы с числами и другими примитивными типами данных.

Вместо операторов равенства, неравенства и других взаимодействие между объектами построено на методах.

Мы впервые столкнулись с таким подходом в массивах, когда познакомились с классом Arrays. Разница в том, что массив сам не содержал в себе методов для взаимодействия с ним, потому была разработана отдельная сущность Arrays, где были объявлены все методы для работы с массивами.

Методы — это действия, описывающие поведения объектов в определенных условиях.

Это именованные блоки кода, которые принадлежат определенным сущностям (обычно объектам) и выполняют с ними какие-то действия.

Методов бывает два типа:

1. Те, что принимают наш объект в виде параметра (как было с методами Arrays, когда мы писали Arrays.имя_метода(параметры_метода);).

2. И те, что объявлены напрямую в объекте и позволяют вызывать эти методы прямо из объекта. объект.имя_метода(параметры_метода).

Для лучшего понимания рассмотрим наиболее популярные операции со строками.

Сравнение строк

При необходимости сравнить строки мы используем метод equals, который в качестве параметра принимает другую строку и возвращает true/false.

В его реализацию мы пока что заглядывать не будем, но запомним, что подобный метод существует у абсолютно всех объектов и является «объектной» альтернативой оператору равенства (==). Т .е. примитивы сравниваем через ==, а объекты — через метод equals.

Этот метод вызывается у одного объекта, а к нему в скобки (в качестве параметра) передается другой объект, «равняться» с которым мы будем.

▼ Код тут.

```
String s = “Hello, world!”;
String s2 = “Hello, world!”;
String s3 = “Hello, Ivan!”;
String s4 = “Hello, ” + “world!”;

boolean sEqualsS2 = s.equals(s2);
// Будет присвоено значение true, так как содержимое строк равно

boolean sEqualsS3 = s.equals(s3);
// Будет присвоено false, так как содержимое разное

boolean sEqualsS4 = s.equals(s4);
// Будет присвоено true, так как содержимое двух разных строк сложилось и превратилось в одну строку, содержимое которой идентично соде
```

Чтобы сравнить строки без учета регистра, нам необходимо заменить метод equals на equalsIgnoreCase.

▼ Код тут.

```
String s = “Anna”;
String s2 = “ANNA”;

boolean sEqualsS2 = s.equals(s2);
/* false, так как данный метод сравнивает полную идентичность,
в том числе регистр буквы (строчная или заглавная) */

boolean sEqualsIgnoreCaseS2 = s.equalsIgnoreCase(s2);
/* true, так как при вызове метода будет запущен цикл по всем символам обеих
строк, каждый из которых будет преобразован в нижний регистр, а затем будет
осуществлено сравнение каждой пары символов через оператор ==, так как
символы (char) являются примитивами */
```

Размер строки

Так как строка внутри является массивом символов, кажется логичным, что у нее должна быть переменная length, хранящая длину.

Верно? Не совсем.

Такого свойства у строки нет, но есть метод с таким же названием и функционалом.

Чтобы обратиться не к свойству, а к методу, нужно добавить круглые скобки.

```
String s = “abc”;
int sLength = s.length(); // Будет присвоено значение 3
```

Представим достаточно жизненную ситуацию.

Мы разработали регистрационную форму, где просим пользователя ввести email.

Чтобы проверить, что пользователь что-то ввел, а уже в дальнейшем проводить более глубокий анализ над содержимым, нам требуется осуществить проверку, пуста ли строка (равна ли ее длина 0).

Подобный код мог выглядеть так:

```
if (s.length() == 0) {
// Печатаем ошибку и просим пользователя ввести email
}
```

Но мы ведь теперь работаем с методами, а они зачастую облегчают для нас шаблонные задачи.

```
if (s.isEmpty()) {
// Печатаем ошибку и просим пользователя ввести email
}
```

Эти два блока имеют абсолютно одинаковую логику, но код с использованием методов выглядит читабельнее и пишется проще.

Различные проверки строки

Так как мы с вами уже познакомились с isEmpty(), нельзя не упомянуть, что проверять строку можно не только на длину, но и на содержание.

Ниже приведена бóльшая часть проверок строки.

▼ Код тут.

```
String s = “abcde”;

s.contains(“bcd”)
// true, так как строка s действительно содержит внутри себя последовательность символов “bcd”

s.endsWith(“de”)
// true, так как строка s действительно заканчивается на “de”

s.startsWith(“ab”)
// true, так как строка s действительно начинается с “ab”

s.equals(“abcde”)
// true, так как содержимое строк равно

s.equalsIgnoreCase(“ABCDE”)
// true, так как при приведении к нижнему регистру содержимое строк равно

s.isEmpty()
// false, так как s.length() не равен 0

s.isBlank()
// false, так как s.length() не равен 0 и имеет буквы и/или числа

“   “.isBlank()
// true, так как строка не имеет внутри букв и чисел, но состоит из пробелов, т. е. пуста (этот метод отсутствует в 8-й версии Java)
```

Другие популярные методы для работы со строками

Теперь пора узнать и о других методах, позволяющих нам работать со строками комфортно.

▼ Код тут.

```
String s = “abcde”;
char c = s.charAt(2);
/* c будет инициализировано значением c, так как именно оно стоит
на третьей позиции в строке s (строка — это массив, т. е. 3-я позиция
располагается под индексом 2) */
```

```
s = “abcd”;
String s1 = s.toUpperCase();
/* s1 будет присвоена копия строки s, где все символы будут в верхнем
регистре, т. е. “ABCD” */

s = “ABCD”;
String s1 = s.toLowerCase();
/* s1 будет присвоено значение строки s, но все символы снова перейдут
 в нижний регистр, т. е. “abcd” */

s = “   abcd    “;
String s1 = s.trim();
/* s1 будет присвоено значение строки s, но без «лидирующих» и
«завершающих» пробелов, они будут удалены, т. е. “abcd” */

s = “My name is Ivan”;
String[] words = s.split(“ “);
/* Данный метод создаст из строки массив, разбив ее на части.
Разделитель указывается в скобках. Мы указали в скобках пробел,
следовательно, делить строку на элементы метод будет в тех местах,
где стоит пробел, т. е. words будет равно {“My”, “name”, “is”, “Ivan”} */

s = “abcd”;
char[] c = s.toCharArray();
/* Преобразует строку в массив символов, присвоив с массив
вида {‘a’, ‘b’, ‘c’, ‘d’} */

s = “abcdef”;
String s1 = s.substring(2, 4);
/* s1 получит значение, которое находится в строке s, начиная
с индекса 2 включительно и заканчивая индексом 4 не включительно, т. е. “cd” */

s = “ab c de”;
String s1 = s.replace(‘ ‘, ‘’);
/* s1 получит копию строки s, но из этой копии будут удалены все
 пробелы (заменены символы из первого параметра (пробел) на символ
из второго параметра (пустая строка)) */

String s2 = s.replace(‘a’, ‘b’);
/* s2 получит строку s, где все символы ‘a’ (первый параметр)
будут заменены на ‘b’ (второй параметр), т. е. “bb c de” */

s = “#”;
String s1 = s.repeat(10);
/* s1 получит строку s, которая будет повторена 10 раз (значение из скобок),
т. е. “##########” */
```

Следует знать, что указанные выше методы могут иметь разный набор параметров и от этого работать по-разному.

Примеры выше являются самыми популярными, но не всеми методами класса String.

Больше методов и подробности об их работе вы найдете в документации класса String по ссылке.

Изменяемые строки (StringBuilder)

Java бы не была самым популярным серверным языком планеты, если бы не имела возможности модернизировать строки без создания новых.

Для этого был создана сущность StringBuilder.

Это сущность, которую можно создать на основе существующей строки или абсолютно новой (пустой).

Делается это следующими способами.

▼ Код тут.

```
StringBuilder sb = new StringBuilder();
// Будет создана сущность StringBuilder на основе пустой строки

StringBuilder sb = new StringBuilder(“123”);
// Будет создана сущность StringBuilder на основе строки “123”
```

Добавить к текущей строке, хранящейся в StringBuilder, другие строки или символы можно через метод append.

▼ Код тут.

```
sb.append(“456”);
// Добавит нашей сущности sb (с “123” внутри) строку “456”, изменив содержимое на “123456”.
```

Эта сущность тоже состоит из массива символов, потому имеет много общего со строкой, в том числе методы.

▼ Код тут.

```
char c = sb.charAt(1);
// Также вернет символ по индексу 1, т. е. c получит значение ‘2’, так как именно этот символ находится на второй позиции

int sbLength = sb.length();
// Получит значение 6, так как именно такая длина у нашей строки “123456”, что лежит внутри сущности

sb.replace(‘1’,’2’);
// Заменит все единицы в содержимом сущности на двойки, т. е. “223456”
```

Но есть и методы, позволяющие изменять текущее состояние хранимой строки. Этим StringBuilder отличается от String.

Например, следующие методы позволяют изменять состояние хранимой внутри sb строки.

▼ Код тут.

```
sb.setCharAt(1, ‘5’);
// Установит вместо символа по индексу 1 символ ‘5’, т. е. текущая строка изменится на “153456”

sb.insert(1, “abc”);
// Установит в ячейки, начиная с первой, символы a, b и c, а ранее находившиеся там ячейки сдвинутся на позиции вперед, т. е. “1abc2345

sb.delete(1, 3);
// Удалит из текущей строки все символы, начиная от ячейки с индексом 1 (включительно) и до ячейки с индексом 3 (не включительно), т. е

sb.deleteCharAt(1);
// Удалит из текущей строки символ, находящийся в ячейке по индексу 1, т. е. строка изменит свое состояние на “13456”

sb.reverse;
// Полностью развернет текущую строку, т. е. превратит ее в “654321”
```

Вы могли заметить, что нам не требуется писать *StringBuilder sb1 =* перед *sb.replace(...)* или *sb.append(...)*.

Это происходит потому, что *StringBuilder* изменяет сам себя, а не создает новый измененный объект и возвращает его. Именно поэтому он называется изменяемым в отличие от String.

Данная особенность позволяет работать со StringBuilder внутри циклов, так как избавляет нас от создания промежуточных объектов на каждом шаге цикла.

По завершению работы со StringBuilder его результат можно присвоить строке или, например, распечатать.

Для этого необходимо вызвать метод ***toString()***

```
String sbResult = sb.toString();
```

Но при печати через System.out.println() вызывать метод ***toString()*** не требуется, так как println делает это за вас.

```
System.out.println(sb);
```

Пул строк (String pool)

Java, как и Москва, не сразу строилась. За время ее жизненного цикла сообщество разработчиков языка внедрило огромное количество оптимизаций, позволяющих Java выполняться очень быстро и экономить ресурсы.

Пул строк — один из внутренних механизмов Java, благодаря которому в памяти сохраняется только ОДИН экземпляр строки идентичного содержания.

Это значит, что если вы объявите в своем приложении две строки в совершенно разных местах приложения, но имеющих одно содержание, Java создаст только одну сущность в памяти и будет подставлять в оба места именно эту одну сущность.

Это и есть одна из причин неизменяемости строк, упомянутых в самом начале этого материала.

Если бы мы могли изменять строку, в пуле бы строка тоже меняла свое содержимое, а значит во всех местах программы, где она была, состояние бы менялось.

Итак, что же происходит, когда мы создаем строку?

1. Вы объявляете переменную типа String.
2. Инициализируете ее путем написания текста в двойных кавычках.
3. Java проверяет, имеется ли такая строка в пуле.
    a. Если да, то присваивает вашей переменной ссылку именно на эту строку.
    b. Если нет, то создает новую строку в памяти и присваивает ссылку на нее вашей переменной

Надеемся, теперь вы понимаете, что строки сделаны неизменяемыми не просто так.

Способы создания строк

В нашем любимом языке создать строку можно не только через написание текста в двойных кавычках.

Так как String является объектом, мы можем создать ее с помощью забытого нами в этом материале, но достаточно важного оператора new.

```
String s = new String(“123”);
```

Этот способ помечен как *Deprecated*. Это значит, что его использование крайне не рекомендуется. Но знать о его существовании следует.

Также следует знать, что когда мы создаем строку с помощью оператора new (а он, напомним, принудительно заставляет Java выделить новый кусок памяти для объекта), она не попадает в пул строк. Это может сыграть с нами злую шутку.

Бывают ситуации, когда у нас есть символьный массив, а мы хотим превратить его в строку.

В такой ситуации нам поможет другой способ создания строки:

```
char[] symbols = {‘a’, ‘b’, ‘c’};
String stringFromSymbols = new String(symbols);
```

Этот способ позволит нам получить строку на основе тех символов, что лежат в массиве.

Важно не забыть упомянуть, что из массива байтов мы тоже можем собрать строку.

Веб-приложения, помимо текста, могут передавать данные еще и в байтах.

Это быстрее, безопаснее и зачастую выигрывает в производительности.

Предположим, что мы получили от соседнего приложения текстовые данные в байтовом формате и собрали их в массив.

```
byte[] textInBytes = {33, 33, 33};
String stringFromBytes = new String(textInBytes);
System.out.println(stringFromBytes);
```

В консоль выведется строка “!!!”, так как значение 33 соответствует символу восклицательного знака в таблице символов ASCII.

На этом всё. Спасибо за прочтение и да будет путь ваш в мире строк проще.

> Шпаргалка урока в PDF-формате:
>
> https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5ecfba75-dc21-40a0-9828-c9a393527e41/1.7._.pdf