GeekBrains

Java Core для Android

Практика ООП и работа со строками

На этом уроке

Разбор практических примеров применения ООП при разработке приложений. Работа с классами String, StringBuilder, StringBuffer

Оглавление

[На этом уроке](#)

[Работа с символьными строками](#)

- [Создание символьных строк](#)
- [Конкатенация строк](#)
- [Методы для работы с символьными строками](#)
- [Классы StringBuilder и StringBuffer](#)
- [Методы для работы с StringBuilder и StringBuffer](#)

[Примеры взаимодействия объектов](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Работа с символьными строками

В Java символьная строка представляет собой последовательность символов.В отличие от других языков программирования, где символьные строки представлены последовательностью символов, в Java они являются объектами класса String.
В результате создания объекта типа String получается неизменяемая символьная строка, т.е. невозможно изменить символы имеющейся строки. При любом изменении строки создается новый объект типа String, содержащий все изменения. Если требуются изменяемые строки, могут быть использованы классы StringBuilder и StringBuffer.

Создание символьных строк

Экземпляр класса String можно создать множеством способов.

```
public class Main {
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = new String("Home");
        String s3 = new String(new char[]{'A', 'B', 'C'});
        String s4 = new String(s3);
        String s5 = new String(new byte[]{65, 66, 67});
        String s6 = new String(new byte[]{0, 65, 0, 66}, StandardCharsets.UTF_16);
         System.out.printf("s1 = %s, s2 = %s, s3 = %s, s4 = %s, s5 = %s, s6 = %s",
```

```
s1, s2, s3, s4, s5, s6);
    }
}
Результат:
s1 = Java, s2 = Home, s3 = ABC, s4 = ABC, s5 = ABC, s6 = AB
```

В примере выше представлена лишь часть возможных вариантов. Самым простым способом является создание строки по аналогии с s1. Обычно строки требуется создавать с начальными значениями. Для этого предоставлены разнообразные конструкторы. При использовании варианта s2 в памяти создается новый экземпляр строки с начальным значением “Home”. Если в программе есть массив символьных строк, то из них можно собрать строку с помощью конструктора, которому в качестве аргумента передается ссылка на массив char[]. Если нужно создать объект типа String, содержащий ту же последовательность символов, что и другой объект типа String, можно использовать вариант s4. Варианты s5 и s6 позволяют «построить» строку из байтового массива без указания кодировки или с ней. Если кодировка не указана, будет использована ASCII.

Конкатенация строк

С помощью операции + можно соединять две символьные строки, порождая в итоге объект типа String. Операции сцепления символьных строк можно объединять в цепочку. Ниже приведено несколько примеров.

```
public class Main {
    public static void main(String[] args) {
        String a1 = "Hello ";
        String a2 = "World";
        String a3 = a1 + a2;
        System.out.println(a3);
        String b1 = "Число 10: ";
        int b2 = 10;
        String b3 = b1 + b2;
        System.out.println(b3);
        String c1 = "Home";
        String c2 = "[" + c1 + "] = " + 1;
        System.out.println(c2);
    }
}
Результат:
Hello World
Число 10: 10
[Home] = 1
```

Как видно из примера выше, строки можно сцеплять с другими типами данных, например, целыми числами(int). Так происходит потому, что значение типа int автоматически преобразуется в своё строковое представление в объекте типа String. После этого символьные строки сцепляются, как и прежде.
Сцепляя разные типы данных с символьными строками, следует быть внимательным, иначе можно получить неожиданные результаты.

```
public class Main {
    public static void main(String[] args) {
        String str = "Десять: " + 5 + 5;
```

```
        System.out.println(str);
    }
}
Результат:
Десять: 55
```

В данном примере выводится следующий результат сцепления: «Десять: 55» вместо: «Десять: 10». Благодаря предшествованию операций сначала выполняется сцепление символьной строки «Десять:» со строковым представлением первого числа 5. Затем полученный результат сцепляется со строковым представлением второго числа 5. Чтобы выполнить сначала целочисленное сложение, следует заключить эту операцию в круглые скобки, как показано ниже.

```
public class Main {
    public static void main(String[] args) {
        String str = "Десять: " + (5 + 5);
        System.out.println(str);
    }
}
Результат:
Десять: 10
```

Методы для работы с символьными строками

Ниже представлено несколько часто используемых методов для работы со строками. Это не полный список.

int length()	Получение длины строки
char charAt(int pos)	Извлечение из строки символа, находящегося на позиции pos, индексация символов начинается с нуля
char[] toCharArray()	Преобразование строки в массив символов
boolean equals(Object obj)	Посимвольное сравнение строк
boolean equalsIgnoreCase(Object obj)	Сравнение строк без учета регистра
Strinq concat(String obj)	Объединение двух строк в одну. Этот метод создает новый строковый объект, содержащий вызываемую строку, в конце которой добавляется содержимое параметра строка. Метод выполняет то же действие, что и операция +.
String toLowerCase(), String toUpperCase()	Преобразование всех символов строки из верхнего регистра в нижний, из нижнего регистра в верхний

Если у вас есть переменная примитивного типа, и ее необходимо преобразовать в строку, то можно воспользоваться методом valueOf().

```
public class Main {
    public static void main(String[] args) {
        int n = 200;
        String str = String.valueOf(n);
    }
}
```

Классы StringBuilder и StringBuffer

В отличие от класса String класс StringBuilder представляет собой **изменяемые** последовательности символов. В классе StringBuilder определены следующие четыре конструктора: StringBuilder(), StringBuilder (int size), StringBuilder (String obj), StringBuilder (CharSequence obj).
Первый конструктор по умолчанию резервирует место для 16 символов. Второй конструктор принимает целочисленный аргумент, задающий размер буфера. Третий конструктор принимает аргумент типа String, задающий начальное содержимое объекта типа StringBuilder и резервирующий место для 16 символов. Выделение места для 16 дополнительных символов позволяет сэкономить время, затрачиваемое на перераспределение памяти при незначительных изменениях начальной строки. Четвертый конструктор создает объект из последовательности символов.

Методы для работы с StringBuilder и StringBuffer

int length()	Получение длины строки.
int capacity()	Получение объема выделенной памяти.
void ensureCapacity(int minimumCapacity)	Предварительное выделение места для определенного количества символов после создания объекта типа StringBuffer, чтобы установить емкость буфера. Это удобно, если заранее известно, что к объекту типа StringBuffer предполагается присоединить большое количество мелких символьных строк.
void setLength(int length)	Установка длины строки. Если указанная длина больше текущей, то в конце строки добавляются пустые символы, если меньше - символы, оказавшиеся за пределами вновь заданной длины строки, удаляются.
void append(...)	Присоединение любого типа данных в конце вызывающего объекта.

Класс StringBuffer отличается от класса StringBuilder только тем, что он является потокобезопасным. Более подробно этот вопрос будет изучен в будущем при рассмотрении темы «Многопоточность».

Примеры взаимодействия объектов

Рассмотрим несколько примеров взаимодействия классов и объектов между собой. Допустим, в программе есть класс Кот. У кота есть кличка и аппетит (сколько единиц еды он потребляет за приём пищи). Эти поля мы заполняем с помощью конструктора. Также есть метод eat(), который заставляет кота покушать, но пока что он пустой, так как неизвестно, откуда кот должен брать еду.

```
public class Cat {
    private String name;
    private int appetite;
    public Cat(String name, int appetite) {
        this.name = name;
        this.appetite = appetite;
    }
    public void eat() { }
}
```

Чтобы можно было хранить еду, создадим класс Тарелка с полем food (еда измеряется в целых числах и не важно, что это за единицы измерения). При создании тарелки мы можем указать начальное значение food. В процессе работы с помощью метода info() можно вывести в консоль информацию о тарелке.

```
public class Plate {
    private int food;
    public Plate(int food) {
        this.food = food;
    }
    public void info() {
        System.out.println("plate: " + food);
    }
}
```

Если в методе main() создать объекты этих классов, то можно узнать информацию о тарелке и вызвать пустой метод eat() у кота. Эти объекты пока никак не могут взаимодействовать между собой.

```
public class MainClass {
    public static void main(String[] args) {
        Cat cat = new Cat("Barsik", 5);
        Plate plate = new Plate(100);
        plate.info();
        cat.eat();
    }
}
```

Можно добавить класса геттеры и сеттеры и получить код вроде.

```
public class MainClass {
    public static void main(String[] args) {
        Cat cat = new Cat("Barsik", 5);
        Plate plate = new Plate(100);
        plate.info();
        cat.eat();
        plate.setFood(plate.getFood() - cat.getAppetite());
    }
}
```

В этом случае получится, что из тарелки мы вычитаем количество еды, потребляемое котом, но при этом метод eat() все также бесполезен. Желательно сделать так, чтобы мы могли указывать коту, из какой тарелки он должен покушать, и при этом сам кот должен уменьшить количество еды в тарелке. Чтобы удобно было уменьшать количество еды в тарелке без использования геттеров и сеттеров, добавим в класс Тарелка метод decreaseFood(int n), который уменьшает food на указанную величину n. Чтобы кот мог взаимодействовать с тарелкой, в метод eat передадим в качестве параметра ссылку на объект класса Тарелка. Ниже приведён код, учитывающий описанные выше моменты. (*Код целиком перенесен на следующую страницу, чтобы его удобно было просматривать.*).

```
public class Plate {
    private int food;
    public Plate(int food) {
        this.food = food;
    }
    public void decreaseFood(int n) {
        food -= n;
    }
    public void info() {
        System.out.println("plate: " + food);
```

```
    }
}
public class Cat {
    private String name;
    private int appetite;
    public Cat(String name, int appetite) {
        this.name = name;
        this.appetite = appetite;
    }
    public void eat(Plate p) {
        p.decreaseFood(appetite);
    }
}
public class MainClass {
    public static void main(String[] args) {
        Cat cat = new Cat("Barsik", 5);
        Plate plate = new Plate(100);
        plate.info();
        cat.eat(plate);
        plate.info();
    }
}
Результат:
plate: 100
plate: 95
```

Как видите, теперь мы можем взять любой объект класса Кот и «попросить» его покушать из любого объекта класса Тарелка с помощью строки вида cat.eat(plate). То есть можно заставить каждого кота кушать из своей миски или взять много котов и заставить кушать из общей. В текущей реализации есть несколько вопросов, которые необходимо решить, но они оставлены на самостоятельное выполнение в качестве домашнего задания.

Домашнее задание

1. Расширить задачу про котов и тарелки с едой.
2. Сделать так, чтобы в тарелке с едой не могло получиться отрицательного количества еды (например, в миске 10 еды, а кот пытается покушать 15-20).
3. Каждому коту нужно добавить поле сытость (когда создаем котов, они голодны). Если коту удалось покушать (хватило еды), сытость = true.
4. Считаем, что если коту мало еды в тарелке, то он её просто не трогает, то есть не может быть наполовину сыт (*это сделано для упрощения логики программы*).
5. Создать массив котов и тарелку с едой, попросить всех котов покушать из этой тарелки и потом вывести информацию о сытости котов в консоль.
6. Добавить в тарелку метод, с помощью которого можно было бы добавлять еду в тарелку.

Дополнительные материалы

1 Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. – М.: Вильямс, 2014. – 864 с.
2 Брюс Эккель Философия Java // 4-е изд.: Пер. с англ. – СПб.: Питер, 2016. – 1168 с.
3 Г. Шилдт Java 8. Полное руководство // 9-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 1376 с.

4 Г. Шилдт Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.

Используемая литература

1. Брюс Эккель Философия Java // 4-е изд.: Пер. с англ. – СПб.: Питер, 2016. – 1168 с.
2. Г. Шилдт Java 8. Полное руководство // 9-е изд.: Пер. с англ. – М.: Вильямс, 2015. 1376 с.
3. Г. Шилдт Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.