Plan

2023-10-16

1. Homework Review
2. StringBuilder, StringBuffer
3. Perfomance test for String, StringBuilder, StringBuffer
4. Abstraction

---

1. Разбор домашнего задания
2. StringBuilder, StringBuffer
3. Тест производительности для String, StringBuilder, StringBuffer
4. Абстракция

Theory

▶ **English**

▼ **На русском**

StringBuilder:

StringBuilder представляет собой изменяемый (mutable) класс в Java, который предназначен для эффективной работы со строками, особенно при выполнении множества операций изменения строки. Он не создает новых объектов строк при каждой операции, а изменяет существующий объект. Пример:

```
StringBuilder sb=new StringBuilder();
sb.append("Hello");
sb.append(" World");
String result=sb.toString();
```

В данном случае, мы создаем объект StringBuilder `sb`, который позволяет нам многократно добавлять и изменять строки. Только в конце мы вызываем `toString()`, чтобы получить результирующую строку.

Разница между StringBuilder и обычной конкатенацией:

При использовании обычной конкатенации строк с оператором `+` в Java каждый раз, когда строки объединяются, создается новый объект строки. Это означает, что при каждой операции конкатенации создается новый экземпляр строки в памяти. Пример:

1. Эффективность: StringBuilder более эффективен при множестве операций изменения строки, так как он не создает лишних объектов строк в памяти.
2. Удобство: Обычная конкатенация с оператором `+` более удобна для простых операций, но может стать неэффективной при больших объемах данных.
3. Изменяемость: StringBuilder позволяет изменять строки, в то время как строки (String) в Java неизменяемы (immutable).

Заключение:

StringBuilder - это мощный инструмент для работы со строками в Java, который обеспечивает эффективность и гибкость при многократных операциях изменения строк. При выборе между обычной конкатенацией строк и StringBuilder, учитывайте потребности вашего проекта и объем данных, с которыми вы работаете.

`StringBuilder` и `StringBuffer` в Java представляют собой два класса, предназначенных для работы со строками и обеспечивающих изменяемость (mutable) строк. Они имеют много общих черт, но есть и различия. Вот основные отличия между ними:

1. **Потокобезопасность (Thread Safety):**
   - `StringBuilder` не является потокобезопасным. Это означает, что если несколько потоков пытаются изменять один и тот же `StringBuilder` одновременно, могут возникнуть проблемы с синхронизацией, и результат может быть непредсказуемым.
   - `StringBuffer`, напротив, является потокобезопасным. Это означает, что он синхронизирует доступ к своим методам, что делает его безопасным для многопоточных операций. Однако синхронизация может сказаться на производительности, поэтому `StringBuffer` обычно менее эффективен, чем `StringBuilder`, если потокобезопасность не требуется.
2. **Производительность:**
   - Из-за отсутствия синхронизации `StringBuilder` обычно быстрее `StringBuffer` в однопоточных приложениях. Если вам не нужна потокобезопасность, `StringBuilder` предпочтительнее.
   - `StringBuffer`, как уже упомянуто, обеспечивает потокобезопасность, но это может привести к небольшим накладным расходам на синхронизацию.
3. **Использование в современных приложениях:**
   - С `Java 5` введен класс `StringBuilder`, который предпочтительнее использовать в новых приложениях, если не требуется потокобезопасность.
   - `StringBuffer` остается полезным в старых приложениях или в случаях, когда потокобезопасность необходима.

В общем, если вам не нужна потокобезопасность, `StringBuilder` - это более эффективный выбор для манипуляции строками. Он предоставляет те же методы, что и `StringBuffer`, но без накладных расходов на синхронизацию. `StringBuffer` следует использовать только в случаях, когда потокобезопасность является критически важной.

ООП - абстракция

Абстрактные классы в Java

Что такое абстрактный класс:

Абстрактный класс в Java - это класс, который объявлен с использованием ключевого слова `abstract`. Абстрактные классы предоставляют средство для создания классов, которые служат в качестве абстрактных шаблонов для других классов. Они могут содержать как абстрактные (без реализации) методы, так и конкретные (с реализацией) методы. Главное отличие абстрактного класса от обычного заключается в том, что вы не можете создать объект абстрактного класса напрямую.

Создание абстрактного класса:

Для создания абстрактного класса используется ключевое слово `abstract` перед объявлением класса:

```
abstract class Animal {
    String name;

    abstract void makeSound(); // Абстрактный метод без реализации

    void eat() {
        System.out.println(name + " ест.");
    }
}
```

Абстрактные методы:

Абстрактный метод - это метод, который объявлен в абстрактном классе, но не имеет реализации в самом классе. Он определяется с помощью ключевого слова `abstract`. Абстрактные методы обязательно должны быть реализованы в производных (конкретных) классах.

Пример абстрактного метода:

```
abstract void makeSound();
```

Производные классы:

Для создания объектов на основе абстрактного класса вы должны создать производный (конкретный) класс и реализовать все абстрактные методы, которые объявлены в абстрактном классе.

Пример производного класса:

```
class Dog extends Animal {
    Dog(String name) {
        this.name = name;
    }

    void makeSound() {
        System.out.println(name + " лает: Гав-гав!");
    }
}
```

Использование абстрактных классов:

```
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog("Шарик");
        dog.eat();
        dog.makeSound();
    }
}
```

В этом примере мы создаем объект класса `Dog`, который является производным от абстрактного класса `Animal`. Мы реализуем абстрактный метод `makeSound()` в классе `Dog`. Таким образом, абстрактные классы позволяют нам создавать абстрактные шаблоны для классов и обеспечивают наследование и реализацию в производных классах.

Заключение:

Абстрактные классы являются важным инструментом в объектно-ориентированном программировании. Они позволяют создавать общие шаблоны для классов и обеспечивать структуру наследования в Java. Абстрактные методы в абстрактных классах гарантируют, что производные классы предоставят необходимую реализацию для этих методов.

Наследование от абстрактного класса и наследование от обычного (конкретного) класса в Java имеют сходства, но также имеют существенные различия.

Наследование от абстрактного класса:

1. **Абстрактные методы:** Абстрактные классы могут содержать абстрактные методы (методы без реализации). Подкласс, наследующий абстрактный класс, **обязан** предоставить реализацию всех абстрактных методов. Это означает, что абстрактные классы предоставляют абстрактные шаблоны для производных классов.

2. **Создание объектов: Нельзя** создать объект абстрактного класса напрямую. Вы можете создать объект только для производного класса, который реализует все абстрактные методы.

3. **Использование наследования:** Наследование от абстрактного класса используется, когда вы хотите создать семейство классов с общими характеристиками и методами, но не хотите предоставлять реализацию для всех методов в базовом классе. Абстрактные классы предоставляют общий интерфейс и соглашения о том, как должны быть реализованы методы в производных классах.

Разница между наследованием от обычного и абстрактного класса заключается в том, что абстрактные классы предоставляют абстрактные методы и создают абстрактные шаблоны для производных классов, в то время как обычные классы предоставляют конкретные реализации методов и могут быть использованы для создания объектов напрямую.

Общее заключение:

Абстрактный класс:

1. **Создание объектов:** Вы не можете создать объект напрямую от абстрактного класса. Он служит как абстрактный шаблон для производных классов.

2. **Общие характеристики:** Абстрактный класс может содержать поля и методы, которые будут общими для всех его производных классов.

3. **Абстрактные методы:** Абстрактный класс может содержать абстрактные методы (методы без реализации), которые должны быть реализованы во всех производных классах.

4. **Методы с реализацией:** Абстрактный класс также может содержать методы с конкретной реализацией, которые могут быть унаследованы и использованы производными классами.

Абстрактный метод:

1. **Без реализации:** Абстрактный метод объявлен в абстрактном классе без конкретной реализации. Он не содержит фактического кода и зависит от конкретных классов, чтобы предоставить свою реализацию.

2. **Обязательность реализации:** Все производные классы, наследующие от абстрактного класса, обязаны предоставить реализацию абстрактных методов. В противном случае компилятор выдаст ошибку.

3. **Полиморфизм:** Абстрактные методы позволяют использовать полиморфизм. Это означает, что вы можете вызывать абстрактный метод на объекте производного класса, и будет выполнена реализация этого метода в соответствии с типом объекта.

Сравнение абстрактного класса и обычного класса:

- **Абстрактный класс**:
  - Может содержать абстрактные методы без реализации.
  - Нельзя создать объект напрямую от абстрактного класса.
  - Используется для создания общего шаблона (абстрактного) для производных классов.
  - Может содержать методы с реализацией и поля.
- **Обычный класс**:
  - Все методы имеют конкретную реализацию.
  - Можно создавать объекты от обычных классов напрямую.
  - Используется для создания конкретных объектов и классов.

Сравнение абстрактного метода и обычного метода:

- **Абстрактный метод**:
  - Объявлен без конкретной реализации в абстрактном классе.
  - Обязан быть реализован во всех производных классах.
  - Позволяет использовать полиморфизм.
- **Обычный метод**:
  - Имеет конкретную реализацию в классе.
  - Не требует обязательной реализации в подклассах.
  - Вызывается так, как определено в конкретном классе.

В итоге, абстрактные классы и методы позволяют создавать абстрактные шаблоны для классов и методов, обеспечивая гибкость и структуру в проектах, где нужно описывать общие характеристики и требования для производных классов.

Homework

▶ **English**

▼ **На русском**

Задача 1.

- Создайте **абстрактный** класс Shape с типом поля double и абстрактными методами calcArea и calcPerimeter.
- Создать классы-наследники Circle, Triangle, Square.
- Убедитесь, что все классы правильно вычисляют площадь и периметр
- Напишите класс FigureAppl с методом main. В методе создайте массив фигур.
- Добавьте в массив три круга, два треугольника и один квадрат.
- Выведи на печать площадь и периметр для **каждой** фигуры
- Рассчитайте общую площадь и общий периметр **всех** фигур из массива фигур.

Задача 2 (опционально)

Создайте абстрактный класс `GameCharacter`, который представляет базовый класс для игровых персонажей.

- Определите абстрактный метод attack(), который будет различаться для разных типов персонажей (например, воин атакует мечом, а маг использует магические заклинания).
- Создайте несколько конкретных подклассов, представляющих разные типы персонажей, такие как `Warrior`, `Mage`, `Archer`, и т. д.
- Переопределите метод `attack()` для каждого класса.
- Создайте массив из игровых персонажей разных типов. Вызовите у каждого метод `attack()` в цикле.

Задание 3.

Создайте массив из 20 случайных целых чисел в интервале от -10 до 10.

Напишите методы, которые ответят на вопросы:

- сколько положительных чисел;
- сколько отрицательных чисел;
- сколько четных чисел;
- какая сумма всех элементов массива

Оформите решение данной задачи методами и напишите для каждого метода тесты.

Code

code/Classwordk_29/src/string_builder/StringBuilderBuffer.java

```
package string_builder;

/**
 * @author Andrej Reutow
 * created on 15.10.2023
```

```
 */
public class StringBuilderBuffer {
}
```

code/Classwordk_29/src/string_builder/StringBuilderPerformanceTest.java

```
package string_builder;

/**
 * @author Andrej Reutow
 * created on 15.10.2023
 */
public class StringBuilderPerformanceTest {

    public static void main(String[] args) {
        String param1 = args.length >= 0 ? args[0] : null; // текст
        String param2 = args.length >= 1 ? args[1] : null; // количесво ите
        int counter = Integer.parseInt(param2); // конвертация строчного зн

        for (int i = 0; i <= 3; i++) {
            System.out.println("#".repeat(60));
            System.out.println("Start test nr " + i);
            System.out.println("#".repeat(60));
            System.out.println();

            long stringPerformanceTest = stringPerformanceTest(param1, coun
            long stringBuilderPerformanceTest = stringBuilderPerformanceTes
            long stringBufferPerformanceTest = stringBufferPerformanceTest(

            System.out.println("FAZIT:");
            System.out.println("stringPerformanceTest\t" + stringPerformanc
            System.out.println("stringBuilderPerformanceTest\t" + stringBui
            System.out.println("stringBufferPerformanceTest\t" + stringBuff

            System.out.println("End test nr " + i);
        }

    }

    public static long stringPerformanceTest(String value, int counter) {
        String str = "";

        // System.currentTimeMillis(); производит отсечку времени в милисек
        long startTimeStringConcat = System.currentTimeMillis();
```

```
            for (int i = 0; i < counter; i++) {
                str += value;
            }

            long endTimeStringConcat = System.currentTimeMillis();

            long result = endTimeStringConcat - startTimeStringConcat;

            return result;
        }

        public static long stringBuilderPerformanceTest(String value, int count
            StringBuilder stringBuilder = new StringBuilder();

            long startTimeStringConcat = System.currentTimeMillis();
            for (int i = 0; i < counter; i++) {
                stringBuilder.append(value);
            }

            long endTimeStringConcat = System.currentTimeMillis();

            long result = endTimeStringConcat - startTimeStringConcat;

            return result;
        }

        public static long stringBufferPerformanceTest(String value, int counte
            StringBuffer stringBuffer = new StringBuffer();

            long startTimeStringConcat = System.currentTimeMillis();
            for (int i = 0; i < counter; i++) {
                stringBuffer.append(value);
            }

            long endTimeStringConcat = System.currentTimeMillis();

            long result = endTimeStringConcat - startTimeStringConcat;

            return result;
        }
    }
```

code/Classwordk_29/src/abstraction/Bicycle.java

```java
package abstraction;

/**
 * @author Andrej Reutow
 * created on 16.10.2023
 */
public class Bicycle extends Transport {

    public Bicycle(int wheels, int speed) {
        super(wheels, speed);
    }

    @Override
    public void move() {
        System.out.println("Велосипед движется на скорости " + this.speed +
    }

    public void ringBell() {
        System.out.println("Велосипед звонит в колокольчик");
    }
}
```

code/Classwordk_29/src/abstraction/Car.java

```java
package abstraction;

/**
 * @author Andrej Reutow
 * created on 16.10.2023
 */
public class Car extends Transport {

    public Car(int wheels, int speed) {
        super(wheels, speed);
    }

    @Override
    public void move() {
        System.out.println("Автомобиль движется на скорости " + this.speed
        // System.out.println("Автомобиль движется на скорости " + speed +
    }
```

```
    public void honk(){
        System.out.println("Автомобиль сигналит");
    }
}
```

code/Classwordk_29/src/abstraction/Lesson29Runner.java

```
package abstraction;

/**
 * @author Andrej Reutow
 * created on 16.10.2023
 */
public class Lesson29Runner {

    public static void main(String[] args) {
        Transport transportCar = new Car(4, 250);
        Transport transportBicycle = new Bicycle(2, 45);

        if (transportCar instanceof Car) { // проверка, является ли transpo
            ((Car) transportCar).honk(); // downCasting до Car
        }
        transportCar.move();

        transportBicycle.move();
    }
}
```

code/Classwordk_29/src/abstraction/Transport.java

```
package abstraction;

/**
 * @author Andrej Reutow
 * created on 16.10.2023
 */
public abstract class Transport { // абстрактный класс

    // общие характеристики/свойства для всех будующих транспортных средств
    protected int wheels; // количество колес
```

```
    protected int speed; // скорость

    public Transport(int wheels, int speed) {
        this.wheels = wheels;
        this.speed = speed;
    }

    abstract public void move(); // абстрактный метод/поведение объекта
}

//  class Transport определяет общие характеристики для всех транспортных с
//  Он также содержит абстрактный метод move(), который должен быть реализо
//  Подклассы, такие как Car и Bicycle, наследуют от абстрактного класса Tr
//  Они также могут иметь свои собственные методы, такие как honk() для авт


//  Таким образом, абстрактный класс Transport служит абстрактным шаблоном д
//  характеристики и методы, но оставляя реализацию конкретных деталей кажд
//  собственных подклассов.
```