Object & String: part 2

StringBuilder

```java
package l24;

public class StringBuilderExample {
    public static void main(String[] args) {
        // Создаем объект StringBuilder
        StringBuilder shoppingList;
        shoppingList = new StringBuilder();

        // Добавляем пункты в список покупок
        shoppingList.append("Молоко");
        shoppingList.append(", Яйца");
        shoppingList.append(", Хлеб");
        shoppingList.append(", Шоколад");

        // Выведем итоговый список покупок
        System.out.println(
            "Список покупок: " +
             shoppingList.toString()
        );
    }
}
```

StringBuilder - это класс, предназначенный для работы со строками, которые нужно часто изменять.

В отличие от класса String, объекты StringBuilder изменяемы.

Это делает StringBuilder идеальным выбором для операций, требующих множественной модификации строк, так как это снижает нагрузку на вашу программу.

Метод toString

```java
package l24;

class Soda {
    private String name; // Название напитка
    private int volume;  // Объем напитка в миллилитрах

    // тут должен быть конструктор

    // Переопределение метода toString для представления
    // объекта Soda в удобочитаемом формате
    @Override
    public String toString() {
        return "Soda{" + "name='" + name + "'" +
            ", volume=" + volume + "ml" + '}';
    }

    public static void main(String[] args) {
        // Создаем объект cola класса Soda
        Soda cola = new Soda("Cola", 500);
        // Печатаем информацию о cola, автоматически
        // вызывается метод toString
        System.out.println(cola.toString());
        System.out.println(cola); // то же самое
    }
}
```

Метод toString в Java предназначен для возвращения строкового представления объекта.

В классе Object этот метод возвращает строку, которая состоит из имени класса объекта, символа @ и его хеш-кода в шестнадцатеричном представлении.

Хотя это предоставляет базовую информацию об объекте, оно обычно бесполезно.

Переопределяя toString, вы можете предоставить более понятное и подробное описание состояния объекта, что облегчает отладку и логирование, а также улучшает взаимодействие с пользователем программы.

Методы equals и hashCode

```java
package l24;

class Person {
    private String name;
    private int age;
    // тут должен быть конструктор
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;

        Person person = (Person) obj;
        return age == person.age && (
            name == null ? person.name == null :
              name.equals(person.name)
        );
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + age;
        return result;
    }
}
```

Метод equals в Java используется для проверки равенства между двумя объектами.

По умолчанию, реализация equals в классе Object сравнивает ссылки на объекты, то есть проверяет, указывают ли две ссылки на один и тот же объект в памяти.

Однако, часто требуется сравнивать объекты по их содержимому, а не по их ссылкам. В таких случаях метод equals необходимо переопределить.

Методы equals и hashCode

```java
package l24;

class Person {
   private String name;
   private int age;
   // тут должен быть конструктор
   @Override
   public boolean equals(Object obj) {
       if (this == obj)
           return true;
       if (obj == null || getClass() != obj.getClass())
           return false;

       Person person = (Person) obj;
       return age == person.age && (
           name == null ? person.name == null :
             name.equals(person.name)
       );
   }

   @Override
   public int hashCode() {
       int result = name != null ? name.hashCode() : 0;
       result = 31 * result + age;
       return result;
   }
}
```

При переопределении equals, важно следовать контракту equals, который требует:

- Рефлексивность: объект должен быть равен самому себе (x.equals(x) всегда true).
- Симметричность: если x.equals(y) true, то и y.equals(x) должен быть true.
- Транзитивность: если x.equals(y) и y.equals(z) оба true, то и x.equals(z) должен быть true.
- Согласованность: если информация, используемая в сравнении объектов, не изменяется, то многократные вызовы x.equals(y) должны возвращать одинаковый результат.
- Ненулевая ссылка: для любого ненулевого значения x, x.equals(null) должен быть false.

Методы equals и hashCode

```java
package l24;

class Person {
   private String name;
   private int age;
   // тут должен быть конструктор
   @Override
   public boolean equals(Object obj) {
       if (this == obj)
           return true;
       if (obj == null || getClass() != obj.getClass())
           return false;

       Person person = (Person) obj;
       return age == person.age && (
           name == null ? person.name == null :
             name.equals(person.name)
       );
   }

   @Override
   public int hashCode() {
       int result = name != null ? name.hashCode() : 0;
       result = 31 * result + age;
       return result;
   }
}
```

Когда вы переопределяете equals, важно также переопределить hashCode.

Это требуется для соблюдения общего контракта метода hashCode, который гласит, что если два объекта равны согласно методу equals, то вызов метода hashCode для этих объектов должен возвращать одинаковое целочисленное значение.

Это правило необходимо для корректной работы объектов в коллекциях, основанных на хеш-таблицах, таких как HashMap и HashSet.