




Lesson 6

31.10.2022



```
public class Ex1 {  
    public static void main(String[] args) {  
        Salmon s = new Salmon();  
        System.out.println(s.count);  
    }  
}
```

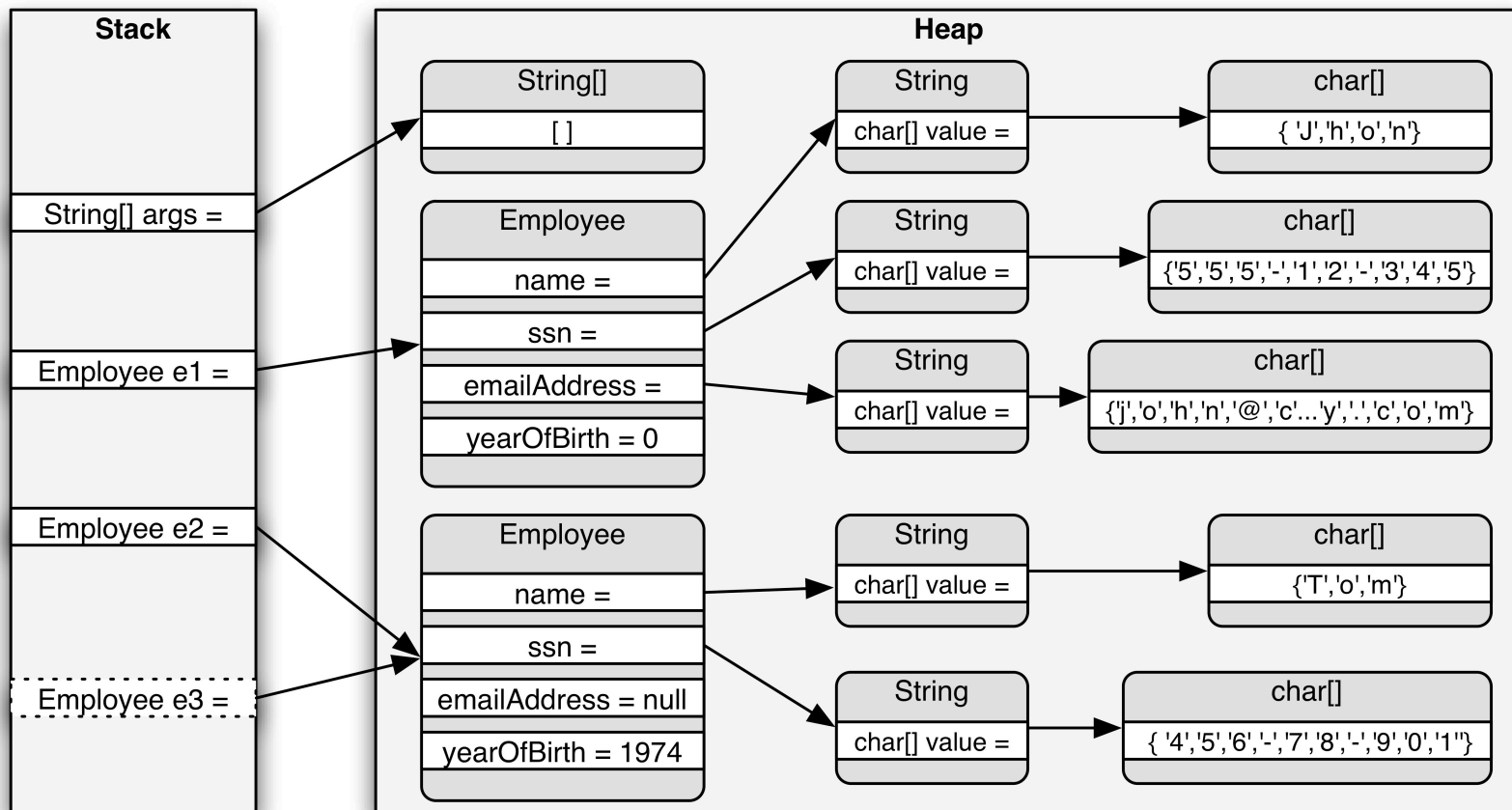
```
class Salmon{  
    int count;  
  
    public Salmon() {  
        count = 4;  
    }  
}
```


```
public class Ex2 {  
    public static void main(String[] args) {  
        int x = 0;  
        while (x++ < 10){}  
        String message = x > 10 ? "Grather than" : false;  
        System.out.println(message+", "+x);  
    }  
}
```

```
public class Ex3 {  
    public static void main(String[] args) {  
        int rez = 5 * 4 % 3;  
        System.out.println(rez);  
    }  
}
```

```
public class Ex4 {  
    public static void main(String[] args) {  
        for (int i = 0; i < 10; ) {  
            i = i++;  
            System.out.println("Hello World");  
        }  
    }  
}
```

```
public class Ex5 {  
    public static void main(String[] args) {  
        int m = 9, n = 1, x = 0;  
        while (m > n) {  
            m--;  
            n += 2;  
            x += m + n;  
        }  
        System.out.println(x);  
    }  
}
```





Java variables do not contain the actual objects, they contain *references* to the objects.


- The actual objects are stored in an area of memory known as the *heap*.
- Local variables referencing those objects are stored on the stack.
- More than one variable can hold a reference to the same object.

Класс *Object* определяет метод *clone()*, который создает копию объекта. Если вы хотите, чтобы экземпляр вашего класса можно было клонировать, необходимо переопределить этот метод и реализовать интерфейс *Cloneable*. Интерфейс *Cloneable* - это интерфейс-маркер, он не содержит ни методов, ни переменных. Интерфейсы-маркер просто определяют поведение классов.

Object.clone() выбрасывает исключение *CloneNotSupportedException* при попытке клонировать объект не реализующий интерфейс *Cloneable*.


Метод *clone()* в родительском классе *Object* является *protected*, поэтому желательно переопределить его как *public*. Реализация по умолчанию метода *Object.clone()* выполняет неполное/поверхностное (shallow) копирование.

```
public class Student implements Cloneable {  
  
    @Override  
    protected Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```



Поверхностное клонирование копирует настолько малую часть информации об объекте, насколько это возможно. По умолчанию, клонирование в Java является поверхностным, т.е. класс `Object` не знает о структуре класса, которого он копирует. Клонирование такого типа осуществляется JVM по следующим правилам:

- Если класс имеет только члены примитивных типов, то будет создана совершенно новая копия объекта и возвращена ссылка на этот объект.
- Если класс помимо членов примитивных типов содержит члены ссылочных типов, то тогда копируются ссылки на объекты этих классов. Следовательно, оба объекта будут иметь одинаковые ссылки.



Глубокое копирование дублирует абсолютно всю информацию объекта:

- Нет необходимости копировать отдельно примитивные данные;
- Все члены ссылочного типа в оригинальном классе должны поддерживать клонирование. Для каждого такого члена при переопределении метода `clone()` должен вызываться `super.clone()`;
- Если какой-либо член класса не поддерживает клонирование, то в методе клонирования необходимо создать новый экземпляр этого класса и скопировать каждый его член со всеми атрибутами в новый объект класса, по одному.



this vs super


Чем **this** и **super** похожи

- И **this**, и **super** — это нестатические переменные, соответственно их нельзя использовать в статическом контексте, а это означает, что их нельзя использовать в методе `main`. Это приведет к ошибке во время компиляции "на нестатическую переменную `this` нельзя ссылаться из статического контекста". То же самое произойдет, если в методе `main` воспользоваться ключевым словом `super`.
- И **this**, и **super** могут использоваться внутри конструкторов для вызова других конструкторов по цепочке, нпр., `this()` и `super()` вызывают конструктор без аргументов наследующего и родительского классов соответственно.
- Внутри конструктора **this** и **super** должны стоять выше всех других выражений, в самом начале, иначе компилятор выдаст сообщение об ошибке. Из чего следует, что в одном конструкторе не может быть одновременно и `this()`, и `super()`.



Различия в `super` и `this`

- Переменная `this` ссылается на текущий экземпляр класса, в котором она используется, тогда как `super` — на экземпляр родительского класса.
- Каждый конструктор при отсутствии явных вызовов других конструкторов неявно вызывает с помощью `super()` конструктор без аргументов родительского класса, при этом у вас всегда остается возможность явно вызвать любой другой конструктор с помощью либо `this()`, либо `super()`.




Интерфейс – это контракт, в рамках которого части программы, зачастую написанные разными людьми, взаимодействуют между собой и со внешними приложениями. Интерфейсы работают со слоями сервисов, безопасности, DAO и т.д. Это позволяет создавать модульные конструкции, в которых для изменения одного элемента не нужно трогать остальные.

```
package com.hillel;

public interface Say {


    void sayHello();

    default void sayGoodbye() {
        System.out.println("Goodbye ... ");
    }
}
```

В имплементирующем интерфейс классе должны быть реализованы все предусмотренные интерфейсом методы, за исключением методов по умолчанию.

Методы по умолчанию впервые появились в Java 8. Их обозначают модификатором `default`. В нашем примере это метод `sayGoodbye`, реализация которого прописана прямо в интерфейсе. Дефолтные методы изначально готовы к использованию, но при необходимости их можно переопределять в применяющих интерфейс классах.



Если у интерфейса только один абстрактный метод, перед нами функциональный интерфейс. Его принято пометить аннотацией `@FunctionalInterface`, которая указывает компилятору, что при обнаружении второго абстрактного метода в этом интерфейсе нужно сообщить об ошибке. Стандартных (default) методов у интерфейса может быть множество – в том числе принадлежащих классу `java.lang.Object`.

```
@FunctionalInterface
public interface Developer {

    boolean isDeveloper();
}
```

Функциональные интерфейсы появились в Java 8. Они обеспечили поддержку лямбда-выражений, использование которых делает код лаконичным и понятным:


Интерфейсы и полиморфизм

В Java полиморфизм можно реализовать через:

наследование — с переопределением параметров и методов базового класса;

абстрактные классы — шаблоны для отдельной реализации в разных классах;

интерфейсы — для имплементации классами.



Кроме обычных классов в Java есть **абстрактные классы**. Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

```
public abstract class Human {  
  
    abstract void see();  
  
    public void talk(String str){  
        System.out.println(str);  
    }  
  
    public void hear(String str){  
        System.out.println(str);  
    }  
}
```



- ☐ Интерфейс описывает только поведение. У него нет состояния. А у абстрактного класса состояние есть: он описывает и то, и другое.
- ☐ Абстрактный класс связывает между собой и объединяет классы, имеющие очень близкую связь. В то же время, один и тот же интерфейс могут реализовать классы, у которых вообще нет ничего общего.
- ☐ Классы могут реализовывать сколько угодно интерфейсов, но наследоваться можно только от одного класса

Отличия абстрактного класса от интерфейса в java

	Abstract Class	Interface
Ключевое слово, используемое для описания	Для определения абстрактного класса используется ключевое слово abstract	Для определения интерфейса используется ключевое слово interface
Ключевое слово, используемое для реализации	Для наследования от абстрактного класса используется ключевое слово extends	Для определения интерфейса используется ключевое слово implements
Конструктор	Определение конструктора разрешено	Объявление/определение конструктора запрещено
Реализация методов по умолчанию	Допустима	Вплоть до Java 8 разрешено только объявлять методы, но реализация их запрещена. Начиная с Java 8 стала допустима реализация static методов, а реализация non-static методов стала доступна посредством ключевого слова default
Поля	Может иметь как static , так и non-static поля (аналогичное относится и к final)	Любое поля интерфейса по умолчанию является public static final (и иных иметь не может)
Модификаторы доступа	Члены абстрактного класса могут иметь любой модификатор доступа: public , protected , private , либо модификатор доступа по умолчанию (package)	Члены интерфейса по умолчанию являются public (и иной модификатор доступа иметь не могут)
Наследование	Может расширить (extends) любой другой класс и реализовать (implements) не более 65535 интерфейсов	Может расширить (extends) не более 65535 интерфейсов


Виды классов в Java

В Java есть 4 вида классов внутри другого класса:

- **Вложенные внутренние классы (Nested Inner classes)** – нестатические классы внутри внешнего класса.
- **Вложенные статические классы (Static Nested classes)** – статические классы внутри внешнего класса.
- **Локальные классы Java (Method Local Inner classes)** – классы внутри методов.
- **Анонимные Java классы (Anonymous Inner classes)** – классы, которые создаются на ходу.

Вложенные внутренние классы (Nested Inner classes)

```
1 public class Outer {  
2     // Простой вложенный класс  
3     class Inner {  
4         public void show() {  
5             System.out.println("Метод внутреннего класса");  
6         }  
7     }  
8  
9     public static void main(String[] args) {  
10         Outer.Inner inner = new Outer().new Inner();  
11         inner.show();  
12     }  
13 }
```



Вложенный внутренний класс может получить доступ к любому приватному полю или методу экземпляра внешнего класса. Вложенный внутренний класс может иметь любой модификатор доступа (`private`, `package-private`, `protected`, `public`). Так же как и классы, интерфейсы могут быть вложенными и иметь модификаторы доступа.

При этом, вложенный внутренний класс не может содержать в себе статических методов или статических полей. Это связано с тем что, внутренний класс неявно связан с объектом своего внешнего класса, поэтому он не может объявлять никаких статических методов внутри себя.

Интерфейсы могут так же быть вложенными, и они имеют некоторые интересные особенности. Мы будем рассматривать вложенные интерфейсы в следующем посте.

Вложенные статические классы(Static Nested classes)

Статические вложенные классы технически не являются внутренними классами. По сути, они представляют собой члены внешнего класса.

```
1 public class Outer {  
2     // Статический внутренний класс  
3     static class Inner {  
4         public void show() {  
5             System.out.println("Метод внутреннего класса");  
6         }  
7     }  
8  
9     public static void main(String[] args) {  
10        Outer.Inner inner = new Outer.Inner();  
11        inner.show();  
12    }  
13 }
```



Локальные классы Java (Method Local Inner classes)

Внутренний класс может быть объявлен внутри метода или блока инициализации внешнего класса.

```
1 public class Outer {
2     void outerMethod() {
3         System.out.println("Метод внешнего класса");
4         // Внутренний класс является локальным для метода outerMethod()
5         class Inner {
6             public void innerMethod() {
7                 System.out.println("Метод внутреннего класса");
8             }
9         }
10        Inner inner = new Inner();
11        inner.innerMethod();
12    }
13
14    public static void main(String[] args) {
15        Outer outer = new Outer();
16        outer.outerMethod();
17    }
18 }
```

Анонимные Java классы(Anonymous Inner classes)

Анонимные внутренние классы объявляются без указания имени класса

```
1 public class Outer {
2     // Анонимный класс наследуется от класса Demo
3     static Demo demo = new Demo() {
4         @Override
5         public void show() {
6             super.show();
7             System.out.println("Метод внутреннего анонимного класса");
8         }
9     };
10
11     public static void main(String[] args) {
12         demo.show();
13     }
14 }
15
16 class Demo {
17     public void show() {
18         System.out.println("Метод суперкласса");
19     }
20 }
```