





# Lesson 6

18.12.2023




```
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;  
    }  
}
```



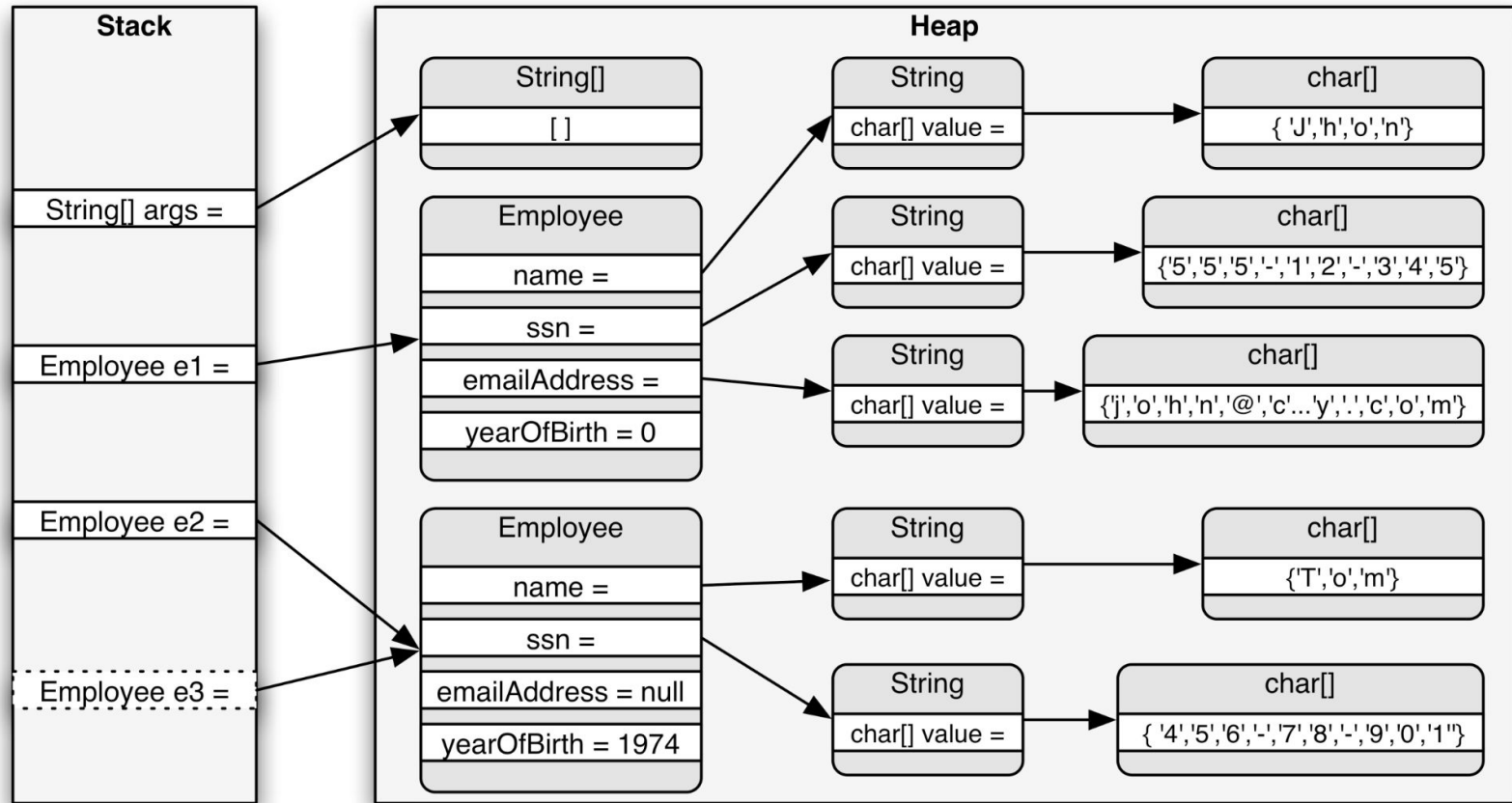
```
public class Ex2 {  
    public static void main(String[] args) {  
        int x = 0;  
        while (x++ < 10){}  
        String message = x > 10 ? "Grather than" : false;  
        System.out.println(message+", "+x);  
    }  
}
```

```
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.



Клас `Object` визначає метод `clone()`, що створює копію об'єкта. Якщо ви хочете, щоб екземпляр вашого класу можна було клонувати, необхідно перевизначити цей метод та реалізувати інтерфейс `Cloneable`. Інтерфейс `Cloneable` – це інтерфейс-маркер, він не містить ні методів, ні змінних. Інтерфейси-маркер просто визначають поведінку класів.

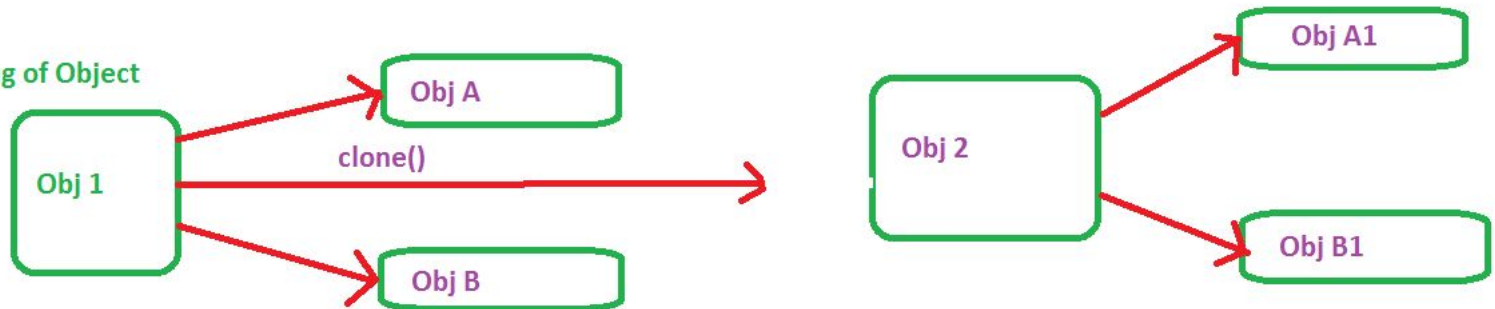
`Object.clone()` викидає виняток `CloneNotSupportedException` при спробі клонувати об'єкт, що не реалізує інтерфейс `Cloneable`. Метод `clone()` у батьківському класі `Object` є `protected`, тому бажано перевизначити його як `public`. Реалізація за умовчанням методу `Object.clone()` виконує неповне/поверхнєве (shallow) копіювання

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

### Shallow Cloning of Object



### Deep Cloning of Object





this vs super

## Чим **this** і **super** схожі

- І **this**, і **super** - це нестатичні змінні, відповідно їх не можна використовувати у статичному контексті, а це означає, що їх не можна використовувати у методі `main`. Це призведе до помилки під час компіляції "на нестатичну змінну цього не можна посилатися зі статичного контексту". те саме станеться, якщо в методі `main` скористатися ключовим словом `super`.
- І **this**, і **super** можуть використовуватися всередині конструкторів для виклику інших конструкторів по ланцюжку, нпр., **this()** і **super()** викликають конструктор без аргументів спадкового та батьківського класів відповідно.
- Всередині конструктора `this` і `super` повинні стояти вище за всіх інших виразів, на початку, інакше компілятор видасть повідомлення про помилку. З чого випливає, що в одному конструкторі не може бути одночасно і `this()`, та `super()`.

## Відмінності у super та this

- Змінна `this` посилається на поточний екземпляр класу, в якому вона використовується, тоді як `super` - на екземпляр батьківського класу.
- Кожен конструктор за відсутності явних викликів інших конструкторів неявно викликає за допомогою `super()` конструктор без аргументів батьківського класу, при цьому у вас завжди залишається можливість явно викликати будь-який інший конструктор з допомогою або `this()`, або `super()`.

**super**

- refer superclass object
- super() can call super class constructor

## this vs super

**this**

- refer current instance of a class inside the class
- this() can call overloaded constructor

**'this' keyword**

Child Class

**'super' keyword**

Parent Class



**Інтерфейс** – це контракт, в рамках якого частини програми, найчастіше написані різними людьми, взаємодіють між собою та із зовнішніми додатками. Інтерфейси працюють з шарами сервісів, безпеки, DAO та і т.д. Це дозволяє створювати модульні конструкції, у яких для зміни одного елемента не потрібно чіпати решту.

```
package com.hillel;


public interface Say {

    void sayHello();

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

У класі, що імплементує інтерфейс, повинні бути реалізовані всі передбачені інтерфейсом методи, за винятком методів замовчуванням (**default**).

Методи за замовчуванням вперше з'явилися в Java 8. Їх позначають модифікатором `default`. У нашому прикладі це метод `sayGoodbye`, реалізація якого прописано прямо в інтерфейсі. Дефолтні методи спочатку готові до використання, але при необхідності їх можна перевизначати в застосовуючи інтерфейс класи.



Якщо інтерфейс має лише один абстрактний метод, перед нами функціональний інтерфейс. Його прийнято позначати інструкцією **@FunctionalInterface**, яка вказує компілятору, що при виявленні другого абстрактного методу у цьому інтерфейсі потрібно повідомити про помилку. Стандартних (default) методів у інтерфейсу може бути безліч – у тому числа що належать класу java.lang.Object

```
@FunctionalInterface
public interface Developer {

    boolean isDeveloper();
}
```






## Інтерфейси та поліморфізм.

У Java поліморфізм можна реалізувати через:

**успадкування** - з перевизначенням параметрів та методів базового класу;

**абстрактні класи** - шаблони для роздільної реалізації у різних класах;

**інтерфейси** – для імплементзації класами.



Крім звичайних класів Java є **абстрактні класи**. Абстрактний клас схожий на клас. В абстрактному класі також можна визначити поля та методи, водночас не можна створити об'єкт або екземпляр абстрактного класу. Абстрактні класи покликані надавати базовий функціонал для класів спадкоємців. А похідні класи вже реалізують цей функціонал.

```
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);  
    }  
}
```

- Інтерфейс описує лише поведінку. Він не має стану. А у абстрактного класу стан є: він описує і те, й інше.
- Абстрактний клас пов'язує між собою та об'єднує класи, що мають дуже близький зв'язок. У той же час, той самий інтерфейс можуть реалізовувати класи, які взагалі немає нічого спільного.
- Класи можуть реалізовувати скільки завгодно інтерфейсів, але успадковуватися можна лише від одного класу



Abstract class	Interface
1) Abstract class can <b>have abstract and non-abstract</b> methods.	Interface can have <b>only abstract</b> methods. Since Java 8, it can have <b>default and static methods</b> also.
2) Abstract class <b>doesn't support multiple inheritance</b> .	Interface <b>supports multiple inheritance</b> .
3) Abstract class <b>can have final, non-final, static and non-static variables</b> .	Interface has <b>only static and final variables</b> .
4) Abstract class <b>can provide the implementation of interface</b> .	Interface <b>can't provide the implementation of abstract class</b> .
5) The <b>abstract keyword</b> is used to declare abstract class.	The <b>interface keyword</b> is used to declare interface.
6) An <b>abstract class</b> can extend another Java class and implement multiple Java interfaces.	An <b>interface</b> can extend another Java interface only.
7) An <b>abstract class</b> can be extended using keyword "extends".	An <b>interface</b> can be implemented using keyword "implements".
8) A Java <b>abstract class</b> can have class members like private, protected, etc.	Members of a Java interface are public by default.

