Design-Pattern

Singleton1.java

Design Pattern Singleton
Erster Versuch, ein Singleton zu erstellen
Leider ist die Ausgabe falsch

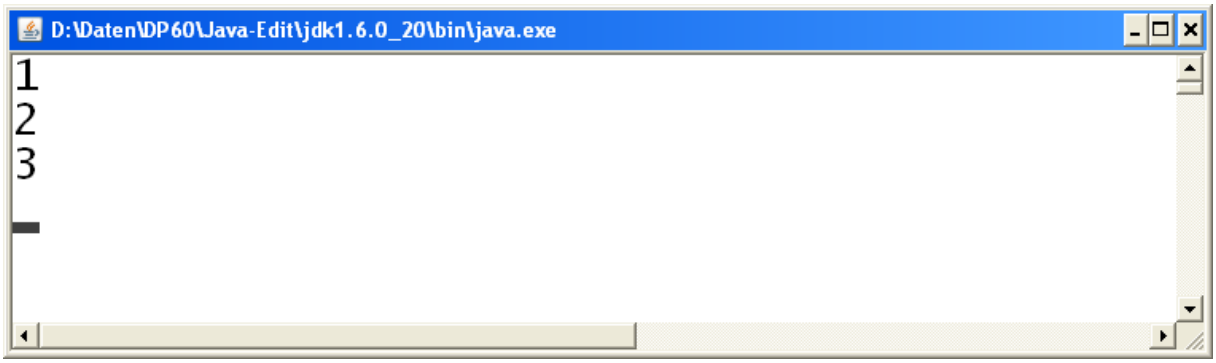


Abbildung 1 Singleton1

```
class Singleton {
    public int value = 0;

        public Singleton(int value){
                this.value = value;
        }
}
```

Singleton2.java

Design Pattern Singleton
Stellt sicher, dass nur eine Instanz existiert, dazu wird der Default-Konstructor als private bzw. protected definiert, das Objekt wird bei der Erzeugung erstellt, das stellt die Einzigartigkeit sicher

In der Methode getInstance wird nun das Objekt erzeugt. Leider nicht Thread sicher.

```
public static Singleton1 getInstance(){
      if (instance == null) {
         instance = new Singleton3(4711);
      }
      return instance;
   }
```

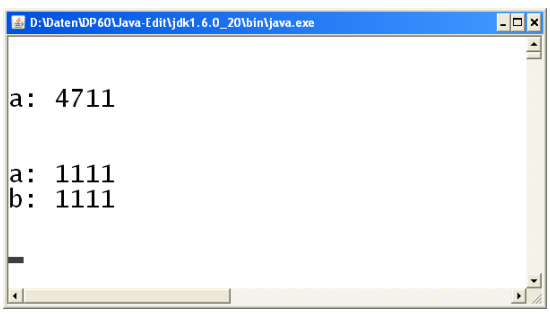


Abbildung 2 Singelton2

Beispiel mit Test-Ausgabe, a und b sind gleiche Instanzen

Design Pattern Singleton
Stellt sicher, dass nur eine Instanz existiert, dazu wird der Default-Konstructor als private bzw. protected definiert,
Beispiel mit Test-Ausgabe, a und b sind gleiche Instanzen

Quellcode:

```
class Singleton {
    private static Singleton instance=null;

    private int a=42;

    public static int b=43;

    protected Singleton(int a) {
       this.a = a;
    }

      // nicht Threadsicher
    public static Singleton getInstance(){
      if (instance == null) {
         instance = new Singleton(4711);
      }
      return instance;
    }

    public int getValue(){
      return a;
    }

    public void setValue(int value) {
      a = value;
    }

}
```

Singleton3.java

Design Pattern Singleton
Stellt sicher, dass nur eine Instanz existiert
dazu wird der Default-Konstrutor als private bzw. protected definiert
die synchronized-Anweisung stellt die Einzigartigkeit sicher

// Beispiel mit Test-Ausgabe
// a und b sind gleiche Instanzen
Beispiel mit Test-Ausgabe, a und b sind gleiche Instanzen

Fehlerhafte Variante, nicht Threadsicher

```
    if (instance == null) {
         instance = new Singleton3(4711);
    }
```

Variante ist Threadsicher, aber verbraucht zuviel Zeit

```
   synchronized(Singleton3.class) {
      if (instance == null) {
           instance = new Singleton3(4711);
      }
   }
```

Jetzt nur synchronized, wenn noch nicht da

```
if (instance == null) {
    synchronized(Singleton3.class) {
        if (instance == null) {
             instance = new Singleton3(4711);
        }
    }
}
```

Quellcode Singleton3.java

```
class Singleton {
    private static Singleton instance = null;
    private int i;

    protected Singleton(int i) {
      this.i = i;
    }


   public static Singleton getInstance(){
     synchronized(Singleton.class) {     // sicher aber langsam
      if (instance == null) {
        instance = new Singleton(42);
      }
    }
    return instance;
   }  // getInstance

    public int getValue() {
      return i;
    }
    public void setValue(int value) {
      i = value;
    }

};
```

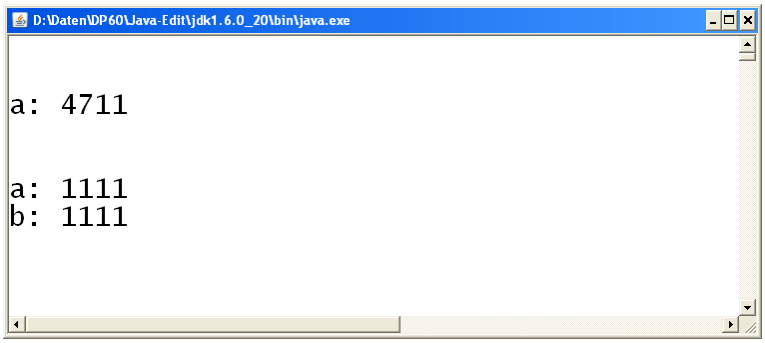


Abbildung 3 Singleton3

Singleton4.java

Design Pattern Singleton
Stellt sicher, dass nur eine Instanz existiert, dazu wird der Default-Konstructor als private bzw. protected definiert,
Durch die statische Erzeugung ist diese Variante Thread-sicher
Beispiel mit Test-Ausgabe, a und b sind gleiche Instanzen

Double Check and lock-Variante

```
class Singleton {
   private static Singleton instance = null;
   private int i;

   protected Singleton(int i) {
     this.i = i;
   }


   public static Singleton getInstance(){              // sicher und schnell
       if (instance == null) {
          synchronized(Singleton.class) {
            if (instance == null) {
              instance = new Singleton(4711);
            }
          }
       }
       return instance;
   }


   public int getValue() {
     return i;
   }
   public void setValue(int value) {
     i = value;
   }

}
```

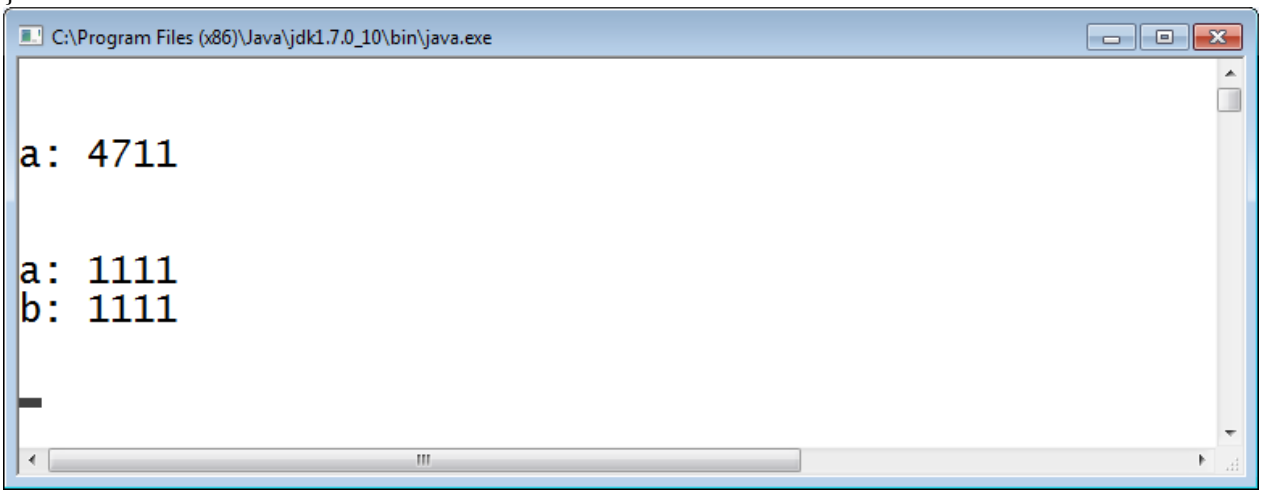


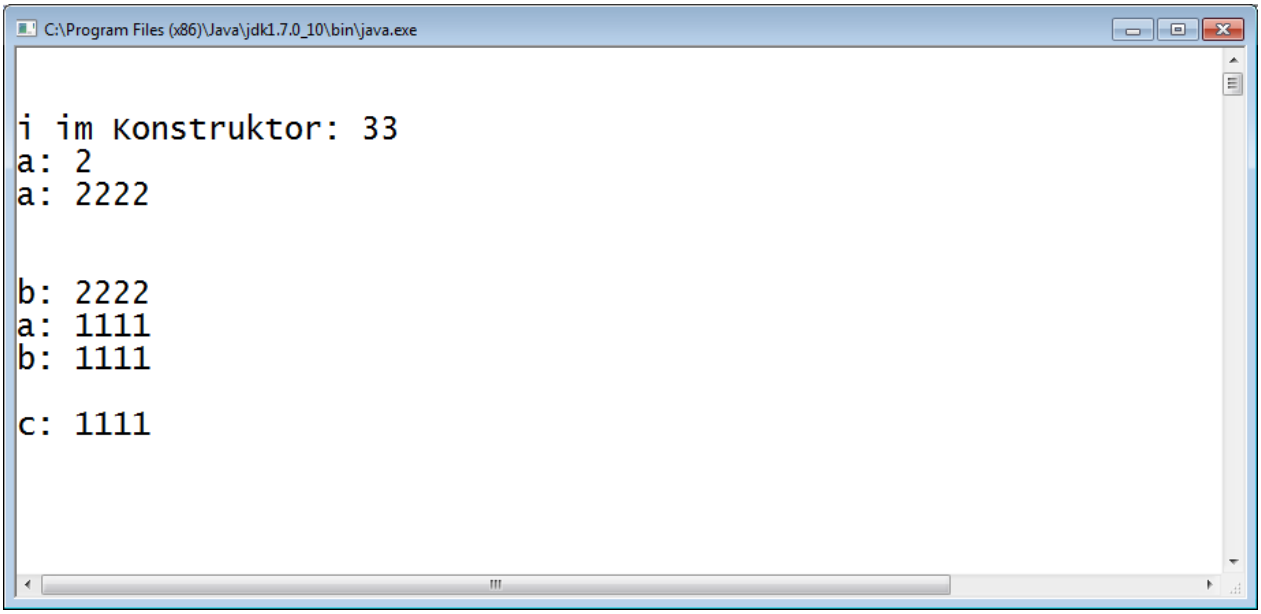
Abbildung 4 Singleton4

Singleton5.java

Korrekte und einfache Version

Quellcode:

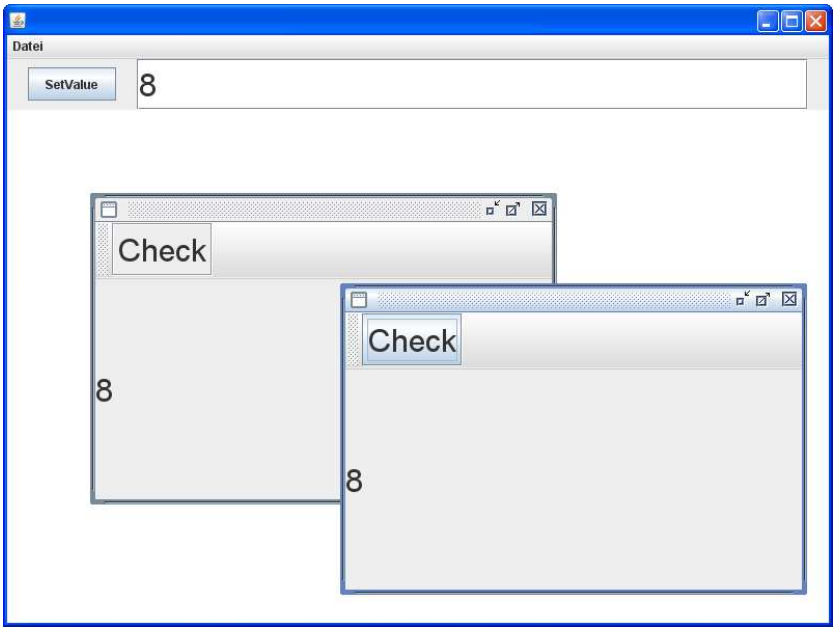
```
class Singleton {
    private static Singleton instance = new Singleton(2);

    private int i=33;

    protected Singleton(int i) {
      System.out.println("i im Konstruktor: "+this.i);
      this.i = i;
    }
    public static Singleton getInstance(){
         return instance;
    }
    public int getValue() {
      return i;
    }
    public void setValue(int value) {
      i = value;
    }

}
```



Singleton6.java

Beispiel eines MDI-Frames zur Demonstration eine Design Pattern Singleton
Im Frame wird eine „globale Variable“ gesetzt.
In den Client wird diese Variable abgeprüft.

einfaches Beispiel eines Internal Frame
mit New einfügen
Internes Fenster zur Demonstration des Singleton-Pattern
Mit dem Schalter "Set" wird die Variable gesetzt
er sollte dann in alle anderen Fenstern zu sehen sein
das geht über den Schalter Check



Mit den Schaltern wird der Wert abgefragt.