



Java Secrets

Escaping References & «Friends»

Michael Inden



- **PART 1: Escaping References**
 - **PART 2: Other Pitfalls**
-



PART 1:

Escaping References



- Immer mal wieder sieht die Übergabe der `this`-Referenz aus eigenen Konstruktoren an andere Objekte.
 - Das wird als »Escaping Reference« bezeichnet.
 - Dass wirkt harmloser, als es in Wirklichkeit ist.
 - Folge: Sichtbarkeit von ungültigen oder teilinitialisierten Zuständen
-



```
static class MainService
{
    private final Callback caller;
    private final String info;

    public MainService()
    {
        caller = new Callback(this);
        info = "Initialized " +
            MainService.class.getSimpleName();
    }

    public String getInfo()
    {
        return info;
    }
}
```

```
static class Callback
{
    private final MainService mainService;

    public Callback(final MainService mainService)
    {
        this.mainService = mainService;

        final byte[] infoAsBytes =
            mainService.getInfo().getBytes();

        System.out.println(Arrays.toString(infoAsBytes));
    }
}
```



Wo liegt das Problem?

Tipp: Herausgabe von this im Konstruktor und Zugriffe



```
static class MainService
{
    private final Callback caller;
    private final String info;

    public MainService()
    {
        caller = new Callback(this);
        info = "Initialized " +
            MainService.class.getSimpleName();
    }

    public String getInfo()
    {
        return info;
    }
}
```

```
public static void main(String[] args)
{
    new MainService();
}
```

```
static class Callback
{
    private final MainService mainService;

    public Callback(final MainService mainService)
    {
        this.mainService = mainService;

        final byte[] infoAsBytes =
            mainService.getInfo().getBytes();

        System.out.println(Arrays.toString(infoAsBytes));
    }
}
```

Herausgabe der this-Referenz



```
static class MainService
{
    private final Callback caller;
    private final String info;

    public MainService()
    {
        caller = new Callback(this);
        info = "Initialized " +
            MainService.class.getSimpleName();
    }

    public String getInfo()
    {
        return info;
    }
}
```

```
static class Callback
{
    private final MainService mainService;

    public Callback(final MainService mainService)
    {
        this.mainService = mainService;
        // Zugriff auf Methode der
        // teilinitialisierten(!) Klasse MainService
        final byte[] infoAsBytes =
            mainService.getInfo().getBytes();

        System.out.println(Arrays.toString(infoAsBytes));
    }
}
```

Auf den ersten Blick korrekt erscheinender Sourcecode verhält sich merkwürdig:

Exception in thread "main" [java.lang.NullPointerException](#): Cannot invoke "String.getBytes()" because the return value of "escaping.EscapingExample\$MainService.getInfo()" is null
at escaping.EscapingExample\$Callback.<init>(EscapingExample.java:37)
at escaping.EscapingExample\$MainService.<init>(EscapingExample.java:21)
at escaping.EscapingExample.main(EscapingExample.java:13)

Analyse Escaping References



```
static class Callback
{
    private final MainService mainService;

    public Callback(final MainService mainService)
    {
        this.mainService = mainService;
        // Zugriff auf Methode der teilinitialisierten(!) Klasse MainService
        final byte[] infoAsBytes = mainService.getInfo().getBytes();
    }
}
```

1. Der Konstruktors der Klasse `Callback` erhält als Eingabe die `this`-Referenz des gerade in der Erzeugung befindlichen `MainService`-Objekts.
2. Zu diesem Zeitpunkt ist die Konstruktion der Klasse `MainService` allerdings noch nicht vollständig abgeschlossen, es steht noch die Initialisierung des Attributs `info` aus.
3. Während der Konstruktion der Klasse `MainService` ist es der Klasse `Callback` bereits möglich, auf Attribute und Methoden der Klasse `MainService` zuzugreifen. Im Speziellen gilt dies auch für das Attribut `info`.
4. Wenn die Klasse `Callback` die Methode `getInfo()` der Klasse `MainService` aufruft, so kommt es durch die nicht initialisierte Referenz `info` zu einer `NullPointerException`.



DEMO



- Durch die Herausgabe der `this`-Referenz können andere Objekte ein unvollständig initialisiertes Objekt sehen und darauf zugreifen. ☹
 - »Escaping References« sind daher zu vermeiden.
 - Es bietet sich der Einsatz des Musters **ERZEUGUNGSMETHODE** an:
 1. Man kann dann ein Objekt zunächst ohne aggregierte Objekte erzeugen.
 2. Anschließend werden die Konstruktoren der aggregierten Objekte aufgerufen, die die Referenz benötigen.
 3. Die eigene Klasse muss allerdings um entsprechende `set()`- bzw. `init()`-Methoden erweitert werden, um die ansonsten im Konstruktor erzeugten Objekte zu initialisieren.
-



```
static class MainServiceCorrected
{
    private Callback caller;
    private final String info;

    public MainServiceCorrected()
    {
        info = "Initialized " +
            MainServiceCorrected.class.getSimpleName();
    }

    public String getInfo()
    {
        return info;
    }
}
```

```
static MainServiceCorrected createMainService()
{
    MainServiceCorrected service = new MainServiceCorrected();
    service.caller = new Callback(service);
    return service;
}
```

```
static class Callback
{
    private final MainServiceCorrected mainService;

    Callback(final MainServiceCorrected mainService)
    {
        this.mainService = mainService;

        final byte[] infoAsBytes =
            mainService.getInfo().getBytes();

        System.out.println(
            Arrays.toString(infoAsBytes));
    }
}
```

```
public static void main(String[] args)
{
    MainServiceCorrected.createMainService();
}
```



DEMO



PART 2:

Other Pitfalls

Bad Smell: Calling abstract methods in constructor



- Manchmal wünscht man sich, in Basisklassen einen gewissen Ablauf bei der Initialisierung vorgeben zu können, etwa den Aufruf einer `init()`-Methode aus dem Konstruktor.
- Die konkrete Realisierung der Initialisierung ist in der Basisklasse jedoch noch unbekannt und soll von Subklassen realisiert werden.
- Idee: Aufruf einer abstrakten Methode `init()` aus dem Konstruktor der Basisklasse.

```
public final class AbstractMethodInConstructorExample
{
    abstract static class AbstractBase
    {
        protected final Integer baseValue = 42;

        AbstractBase()
        {
            init(); // Aufruf der Initialisierung
        }

        abstract void init();
    }
}
```

Bad Smell: Calling abstract methods in constructor



```
static class Derived extends AbstractBase
{
    private final Integer value = 13;

    void init()
    {
        // Zugriff auf Attribut der Basisklasse
        System.out.println("baseValue = " + baseValue);
        // Zugriff auf Attribut dieser Klasse
        System.out.println("value = " + value);
    }
}
```




**Das wirkt doch
ganz vernünftig ...**

Bad Smell: Calling abstract methods in constructor



```
static class Derived extends AbstractBase
{
    private final Integer value = 13;

    void init()
    {
        // Zugriff auf Attribut der Basisklasse
        System.out.println("baseValue = " + baseValue);
        // Zugriff auf Attribut dieser Klasse
        System.out.println("value = " + value);
    }
}

public static void main(String[] args)
{
    // Konstruktion zur Demonstration der Probleme
    new Derived();
}
```

```
baseValue = 42
value = null
```

Analyse Abstract Methods



```
public static void main(String[] args)
{
    new Derived();
}

static class Derived extends AbstractBase
{
    private final Integer value = 13;

    void init() {
        // Zugriff auf Attribut der Basisklasse
        System.out.println("baseValue = " + baseValue);
        // Zugriff auf Attribut dieser Klasse
        System.out.println("value = " + value);
    }
}

abstract static class AbstractBase
{
    protected final Integer baseValue = 42;

    AbstractBase() {
        init(); // Aufruf der Initialisierung
    }

    abstract void init();
}
```

1. In der Methode main() wird ein Objekt der Klasse Derived konstruiert.
2. Der Aufruf des Defaultkonstruktors Derived ruft wiederum den Konstruktor der Basisklasse AbstractBase auf.
3. Der dortige Aufruf der init()-Methode führt aufgrund der Polymorphie und des dynamischen Bindens zum Aufruf der init()-Methode der Klasse Derived.
4. Die Variable value ist zu diesem Zeitpunkt aber noch nicht zugewiesen – der Konstruktor von Derived ist noch nicht vollständig abgearbeitet –, daher wird der Wert null ausgegeben.
5. Erst nach Beendigung des Aufrufs der Methode init() wird auch die Abarbeitung des Basisklassenkonstruktors abgeschlossen. Erst danach wird das Attribut value als Teilschritt der Konstruktion und Initialisierung der Klasse Derived auf den Wert 13 gesetzt.



DEMO



Multi Threading



```
public class ThreadAutoStart implements Runnable
```

```
{  
    public ThreadAutoStart()  
    {
```

```
        new Thread(this).start();  
    }
```

```
public class ErroneousDataAccessThread extends ThreadAutoStart
```

```
{  
    private final DataService service;
```

```
    public ErroneousDataAccessThread(final DataService service)  
    {
```

```
        // Aufwendige Initialisierung
```

```
        // ...
```

```
        this.service = service;  
    }
```

```
@Override
```

```
public void run()  
{
```

```
    // Möglicherweise hier schon Zugriff, weil Initialisierung
```

```
    // im Konstruktor noch nicht abgeschlossen => NullPointerException
```

```
    final SomeData data = service.retrieveData(...);
```

```
    // ...  
}
```

```
}
```





Komplexe Initialisierung



Bad Smell: Calling abstract methods in constructor



```
public abstract class CommunicationBase {
    CommunicationBase() {
        createComponents();
    }

    abstract protected void createComponents();
}

public final class RadioCommunication extends CommunicationBase {
    // Definition des Synchronisationsobjekts
    private final Object sharedSyncObject = new Object();

    private RadioSender dataSender = null;

    @Override
    protected final void createComponents() {
        // Übergabe des Synchronisationsobjekts
        dataSender = new RadioSender(sharedSyncObject);
    }
}
```


Bad Smell: Calling abstract methods in constructor



```
public final class RadioSender {
    private final Object sharedSyncObject;

    public RadioSender(final Object sharedSyncObject) {
        this.sharedSyncObject = sharedSyncObject;
    }

    public void send(final byte[] msg) {
        synchronized (sharedSyncObject) {
            sendBytes(msg);
        }
    }

    private void sendBytes(final byte[] msg) {
        System.out.println(Arrays.toString(msg));
    }
}
```

Bad Smell: Calling abstract methods in constructor



```
public static void main(String[] args)
{
    var radioCommunication = new RadioCommunication();
    radioCommunication.dataSender.send(new byte[] { 64, 65, 66, 67 });
}
```

Exception in thread "main" [java.lang.NullPointerException](#): Cannot invoke "abstractmethods.RadioSender.send(byte[])" because "dataSender" is null at abstractmethods.RadioCommunication.main([RadioCommunication.java:23](#))



DEMO

RadioCommunication



Questions?

Hilfe





Thank You
