Back to basics reloaded

Interessantes aus dem Java Core

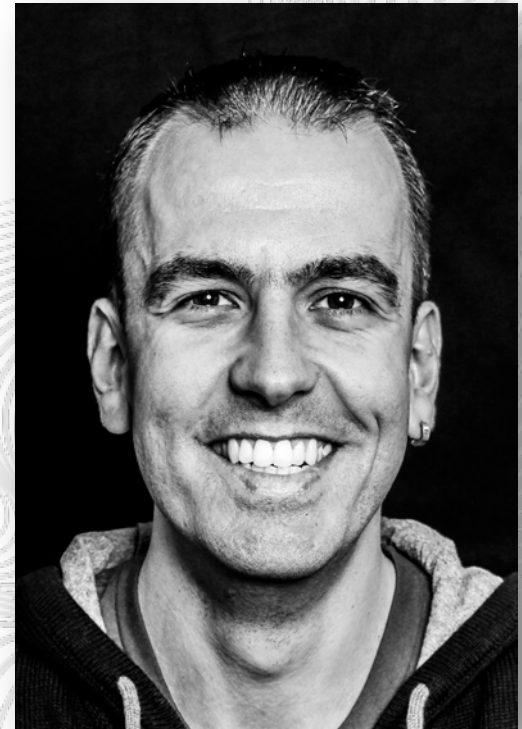
SapientNitro

Christian Robert

Senior Developer Mobile Solutions

SapientNitro
Kaiser-Wilhelm-Ring 17-21
50672 Köln

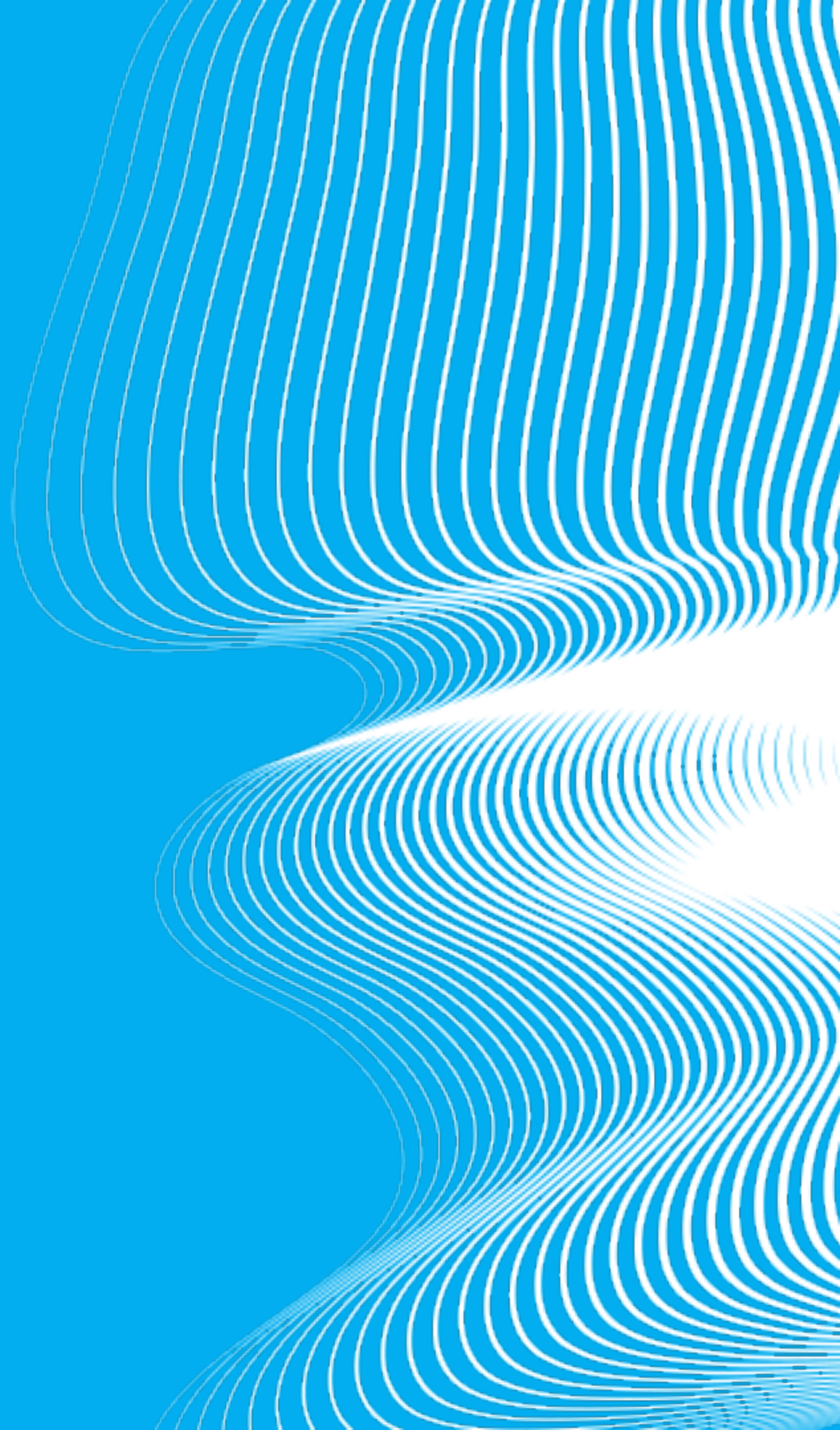
crobert@sapient.com
www.sapientnitro.de



Agenda

1. java.lang | Suppressed Exceptions
2. java.io | Serialisierung
3. java.lang | Objekterzeugung
4. java.lang | Cloning

01 Suppressed Exceptions



Suppressed Exceptions

addSuppressed

`public final void addSuppressed(Throwable exception)`

Appends the specified exception to the exceptions that were suppressed in order to deliver this exception. This method is thread-safe and typically called (automatically and implicitly) by the `try`-with-resources statement.

The suppression behavior is enabled *unless* disabled via a constructor. When suppression is disabled, this method does nothing other than to validate its argument.

Note that when one exception causes another exception, the first exception is usually caught and then the second exception is thrown in response. In other words, there is a causal connection between the two exceptions. In contrast, there are situations where two independent exceptions can be thrown in sibling code blocks, in particular in the `try` block of a `try`-with-resources statement and the compiler-generated `finally` block which closes the resource. In these situations, only one of the thrown exceptions can be propagated. In the `try`-with-resources statement, when there are two such exceptions, the exception originating from the `try` block is propagated and the exception from the `finally` block is added to the list of exceptions suppressed by the exception from the `try` block. As an exception unwinds the stack, it can accumulate multiple suppressed exceptions.

An exception may have suppressed exceptions while also being caused by another exception. Whether or not an exception has a cause is semantically known at the time of its creation, unlike whether or not an exception will suppress other exceptions which is typically only determined after an exception is thrown.

Note that programmer written code is also able to take advantage of calling this method in situations where there are multiple sibling exceptions and only one can be propagated.

Parameters:

`exception` - the exception to be added to the list of suppressed exceptions

Throws:

`IllegalArgumentException` - if `exception` is this throwable; a throwable cannot suppress itself.

`NullPointerException` - if `exception` is `null`

Since:

1.7

Suppressed Exceptions

```
public class Suppressed {

    public void doStuff() {
        try (InputStream inStream = this.createStream()) {
            this.processStream(inStream);
        }
    }

    ...
```

```
java.io.IOException: Error in stream read
     at test.Suppressed$1.read(Suppressed2.java:32)
     at test.Suppressed.processStream(Suppressed2.java:23)
     at test.Suppressed.doStuff(Suppressed2.java:18)
     at test.Suppressed.main(Suppressed2.java:10)
     Suppressed: java.io.IOException: Error in stream close
          at test.Suppressed$1.close(Suppressed2.java:36)
          at test.Suppressed.doStuff(Suppressed2.java:19)
          ... 1 more
```

Suppressed Exceptions

```
public class Suppressed {

    public void doStuff() {

        List<Row> rows = this.createRows();
        for (int i=0; i < rows.size(); i++) {
            Row row = rows.get(i);
            try {
                this.processRow(row);
            } catch (Exception e) {

                ???

            }
        }

    }

    ...
```

Suppressed Exceptions

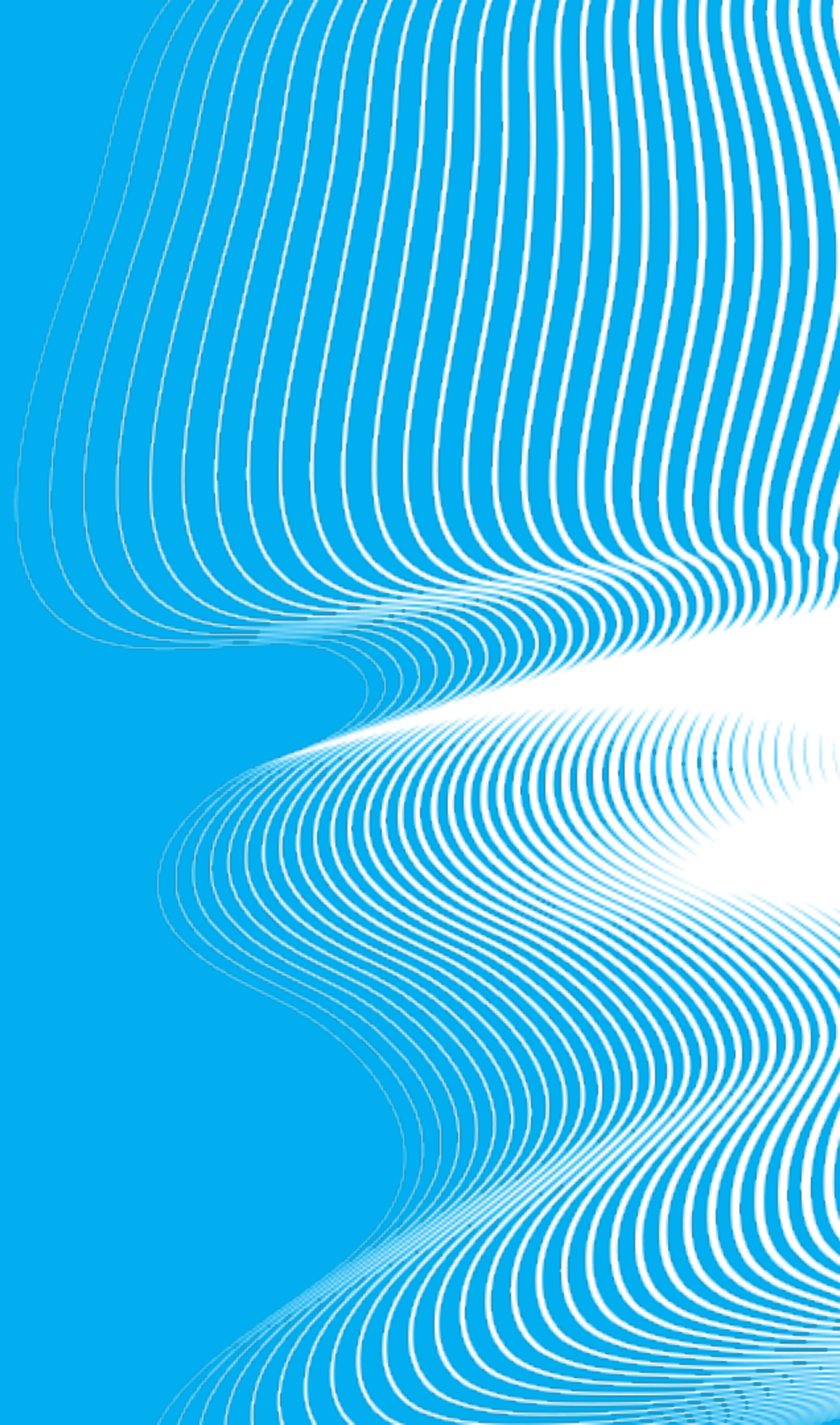
```
public class Suppressed {

    public void doStuff() {

        ProcessingException error = new ProcessingException("Error in processing");
        boolean errorFound = false;

        List<Row> rows = this.createRows();
        for (int i=0; i < rows.size(); i++) {
            Row row = rows.get(i);
            try {
                this.processRow(row);
            } catch (Exception e) {
                errorFound = true;
                error.addSuppressed(e);
            }
        }

        if (errorFound) {
            throw error;
        }

    }

    ...
```

Suppressed Exceptions

```
test.ProcessingException: Error in processing
    at test.Suppressed.doStuff(Suppressed.java:18)
    at test.Suppressed.main(Suppressed.java:10)

    Suppressed: java.lang.IllegalArgumentException: Invalid value for row: 2
        at test.Suppressed.processRow(Suppressed.java:47)
        at test.Suppressed.doStuff(Suppressed.java:25)
        ... 1 more

    Suppressed: java.lang.IllegalArgumentException: Invalid value for row: 4
        at test.Suppressed.processRow(Suppressed.java:47)
        at test.Suppressed.doStuff(Suppressed.java:25)
        ... 1 more
```

02

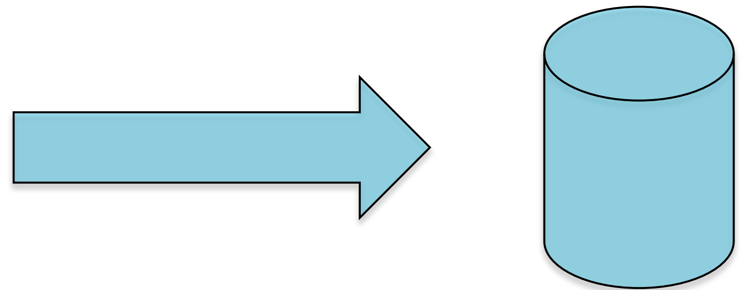
Serialisierung



Serialisierung

```java
public class Address {

    private String firstname = null;
    private String lastname = null;

    @Override
    public String toString() {
        return this.firstname + " " + this.lastname;
    }

}
```

```java
Address a = new Address();
a.firstname = "Christian";
a.lastname = "Robert";
```



Serialisierung

```java
public class Address implements java.io.Serializable {

    private String firstname = null;
    private String lastname = null;

    @Override
    public String toString() {
        return this.firstname + " " + this.lastname;
    }

}


Address a = new Address();
a.firstname = "Christian";
a.lastname = "Robert";

ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(a);

System.out.println(Hex.encodeHex(bos.toByteArray()));
```

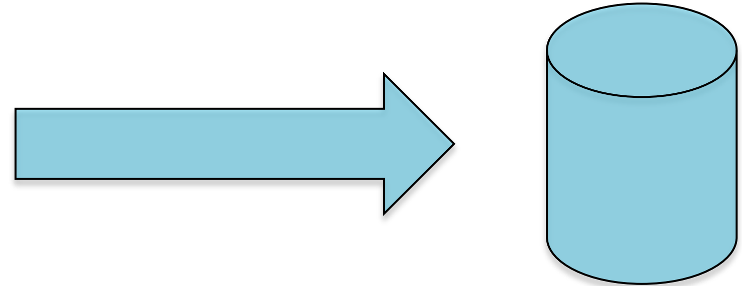

aced00057372001b6
36f6d2e7361706965
6e742e6578616d706
c652e41646472657373
73334484a09182911
80200024c00096669
7273746e616d65740
0124c6a61766...

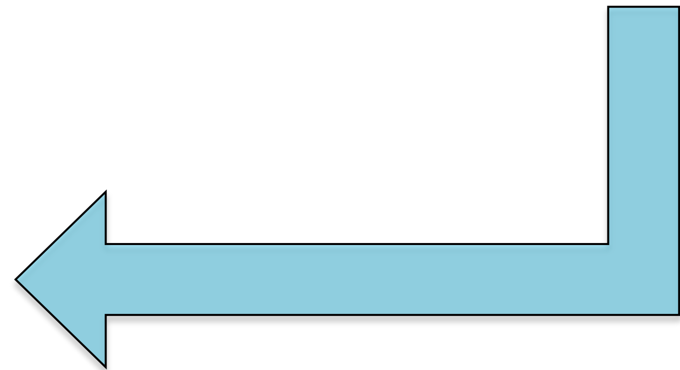
Serialisierung

```java
public byte[] addressToBytes() throws IOException {

    Address a = new Address();
    a.firstname = "Christian";
    a.lastname = "Robert";

    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(bos);
    oos.writeObject(a);
    return bos.toByteArray();

}

public Address bytesToAddress(byte[] data) throws IOException, ClassNotFoundException {

    ByteArrayInputStream bis = new ByteArrayInputStream(data);
    ObjectInputStream ois = new ObjectInputStream(bis);
    return (Address)ois.readObject();

}
```

Serialisierung - Versionierung

```java
public class Address implements Serializable {

    private String firstname = null;
    private String lastname = null;

}
```





```java
public class Address implements Serializable {

    private String firstname = null;
    private String lastname = null;
    private String address = null;

}
```

```
java.io.InvalidClassException: com.sapient.example.Address; local class
incompatible: stream classdesc serialVersionUID = -1494107031540832320,
local class serialVersionUID = 4065003677238715880c
```

Serialisierung - serialVersionUID

"The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization."

-- Javadoc java.io.Serializable

"If a serializable class does not explicitly declare a serialVersionUID, then **the serialization runtime will calculate a default serialVersionUID** value for that class based on various aspects of the class […]

[…] the default serialVersionUID computation is **highly sensitive to class details** that may vary depending on compiler implementations […]

Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must

Serialisierung - serialVersionUID

```
public class Address {

    static final long serialVersionUID = ?

    private String firstname = null;
    private String lastname = null;

}
```

Serialisierung - serialVersionUID

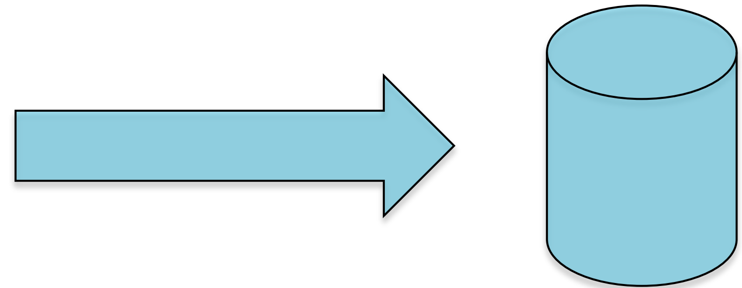
```
$ serialver -classpath /somewhere/classes com.sapient.example.Address

com.sapient.example.Address:
static final long serialVersionUID = 4065003677238715880L;
```

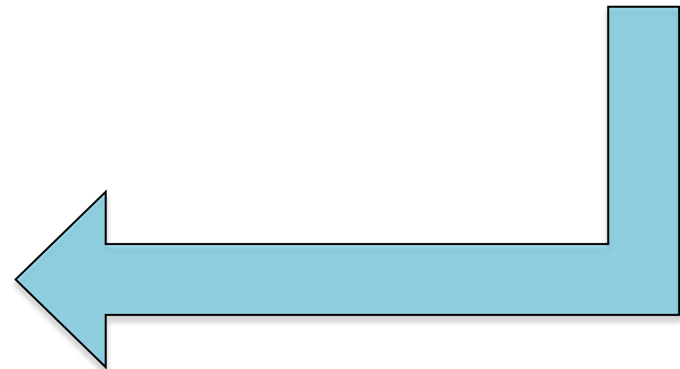
```java
public class Address {

    static final long serialVersionUID = 4065003677238715880L;

    private String firstname = null;
    private String lastname = null;

}
```

```java
public class Address {

    static final long serialVersionUID = 1L;

    private String firstname = null;
    private String lastname = null;

}
```

Serialisierung - serialVersionUID

```
public class Address implements Serializable {

    static final long serialVersionUID = 1L

    private String firstname = null;
    private String lastname = null;

}
```





```
public class Address implements Serializable {

    static final long serialVersionUID = 1L;

    private String firstname = null;
    private String lastname = null;
    private String address = null;

}
```

```
OK!
```

Serialisierung - Transient fields

```java
public class Demo implements Serializable {

    static final long serialVersionUID = 1L;

    private String someString = "someString";
    private transient String someOtherString = "someOtherString";

}

byte[] demoBytes = objectToBytes(new Demo());
Demo demo = objectFromBytes(demoBytes);

System.out.println("1: " + demo.someString);
System.out.println("2: " + demo.someOtherString);
```

```
1: someString
2: null
```

Serialisierung - Transient fields

```
public class Demo implements Serializable {

    static final long serialVersionUID = 1L;

    private static final ObjectStreamField[] serialPersistentFields =
        new ObjectStreamField[] {
            new ObjectStreamField("someString", String.class)
        }
    ;

    private String someString = "someString";
    private String someOtherString = "someOtherString";

}

byte[] demoBytes = objectToBytes(new Demo());
Demo demo = objectFromBytes(demoBytes);

System.out.println("1: " + demo.someString);
System.out.println("2: " + demo.someOtherString);
```

```
1: someString
2: null
```

Serialisierung - Eigene Logik

```
public class BusinessObject implements Serializable {

    static final long serialVersionUID = 1L;

    private String someString = null;
    private transient File localFileCache = null;

}
```

Serialisierung - Eigene Logik

```
public class BusinessObject implements Serializable {

    static final long serialVersionUID = 1L;

    private String someString = null;
    private transient File localFileCache = null;

    private void readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException {

        stream.defaultReadObject();

        this.localFileCache = LocalFileCache.computeNewCacheFile();

    }

}
```

Serialisierung - Eigene Logik

```
public class Address implements Serializable {

    static final long serialVersionUID = 1L;

    private String firstname = null;
    private String lastname = null;
    private String sensitiveInformation = null;

    private void writeObject(java.io.ObjectOutputStream out)
        throws IOException {

        out.writeObject(this.firstname);
        out.writeObject(this.lastname);

    }

    private void readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException {

        this.firstname = (String)stream.readObject();
        this.lastname = (String)stream.readObject();
        this.sensitiveInformation = "Sensitive information not read from Stream";

    }

}
```

Serialisierung - Eigene Logik - Stolperfallen!

```
public class Base {

    public String stringInBase = "stringInBase";

    private void writeObject(ObjectOutputStream stream) throws IOException {
        System.out.println("writeObject in Base");
        stream.defaultWriteObject();
    }

}

public class Derived extends Base implements Serializable {

    public String stringInDerived = "stringInDerived";

    private void writeObject(ObjectOutputStream stream) throws IOException {
        System.out.println("writeObject in Derived");
        stream.defaultWriteObject();
    }

}

ObjectOutputStream objectOutputStream = createObjectOutputStream();
objectOutputStream.writeObject(new Derived());
```

```
writeObject in Derived
```

Serialisierung - Eigene Logik - Stolperfallen!

```
public class Base implements Serializable {

    public String stringInBase = "stringInBase";

    private void writeObject(ObjectOutputStream stream) throws IOException {
        System.out.println("writeObject in Base");
        stream.defaultWriteObject();
    }

}

public class Derived extends Base implements Serializable {

    public String stringInDerived = "stringInDerived";

    private void writeObject(ObjectOutputStream stream) throws IOException {
        System.out.println("writeObject in Derived");
        stream.defaultWriteObject();
    }

}

ObjectOutputStream objectOutputStream = createObjectOutputStream();
objectOutputStream.writeObject(new Derived());
```

```
writeObject in Base
writeObject in Derived
```

Serialisierung - Eigene Logik

```
public class DummyBuilder implements Serializable {

    private Object writeReplace() throws ObjectStreamException {
        return "DUMMY!";
    }

}


ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(new DummyBuilder());
oos.flush();

ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
ObjectInputStream ois = new ObjectInputStream(bis);
String string = (String)ois.readObject();
```

Serialisierung - Eigene Logik

```
public class DummyBuilder implements Serializable {

    private Object readResolve() throws ObjectStreamException {
        return "DUMMY!";
    }

}


ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(new DummyBuilder());
oos.flush();

ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
ObjectInputStream ois = new ObjectInputStream(bis);
String string = (String)ois.readObject();
```

Serialisierung - Eigene Logik - Alles selbst!

```
public interface Externalizable extends Serializable {

    public void writeExternal(ObjectOutput out)
        throws IOException;

    public Object readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException;

}
```

"Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances. […]
These methods must explicitly coordinate with the supertype to save its state. These methods supersede customized implementations of writeObject and readObject methods."

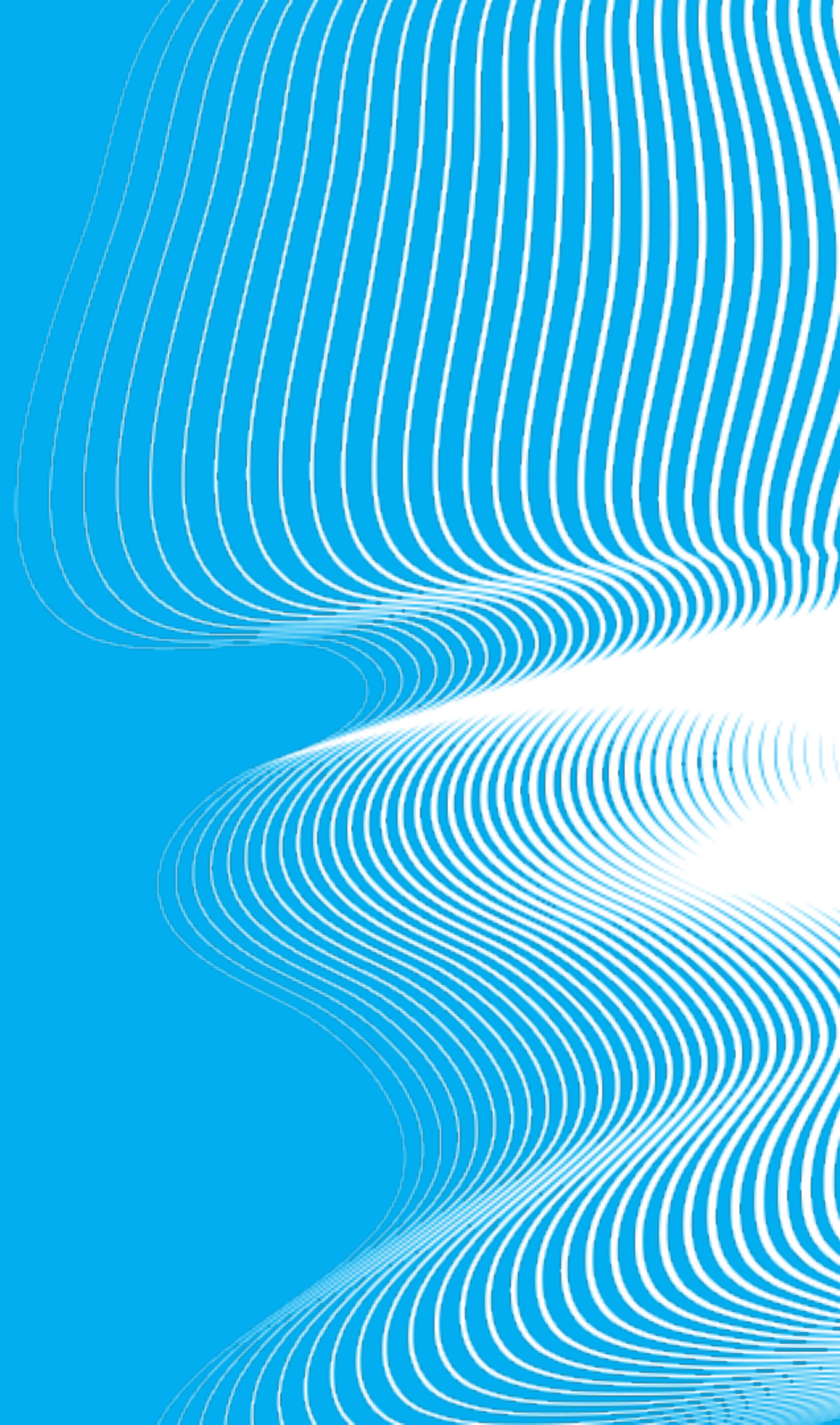
-- Javadoc java.io.Externalizable

Serialisierung

- Kompletter Objektgraph muss serialisierbar sein
- Anpassungen über readObject, writeObject, readResolve, writeReplace - immer nur für aktuelle Klasse!
- Komplette Kontrolle über serialisierte Inhalte mittels java.io.Externalizable

03

Objekterzeugung



Objekterzeugung - Static inner classes

```java
public class OuterType {

    public void startThread(ExecutorService executorService) {
        executorService.submit(new InnerType());
    }

    static class InnerType implements Runnable {

        @Override
        public void run() {
            System.err.println("Hello from run!");
        }

    }

}
```

Objekterzeugung - Static inner classes

```
public class OuterType {

    public void startThread(ExecutorService executorService) {
        executorService.submit(new OuterType$InnerType());
    }

}

class OuterType$InnerType implements Runnable {

    @Override
    public void run() {
        System.err.println("Hello from run!");
    }

}
```

Objekterzeugung - Inner classes

```
public class OuterType {

    private String outerTypeString = "foo";

    public void startThread(ExecutorService executorService) {
        executorService.submit(new InnerType());
    }

    class InnerType implements Runnable {

        @Override
        public void run() {
            System.err.println("Hello from: " + OuterType.this.outerTypeString);
        }

    }

}
```

Objekterzeugung - Inner classes

Read access to enclosing field OuterType.outerTypeString is emulated by a synthetic accessor method

```
public class OuterType {

    private String outerTypeString = "foo";

    public void startThread(ExecutorService executorService) {
        executorService.submit(new OuterType$InnerType(this));
    }

    static synthetic String access$0(OuterType arg0) {
        return arg0.outerTypeString;
    }

}

class OuterType$InnerType implements Runnable {

    private OuterType this$0;

    OuterType$InnerType(OuterType arg0) {
        this.this$0 = arg0;
    }

    @Override
    public void run() {
        System.err.println("Hello from: " + OuterType.access$0(this.this$0));
    }
```

OuterType.this.outerTypeString --> this$0.outerTypeString

Objekterzeugung - Inner classes

```java
public class OuterType {

    String outerTypeString = "foo";

    public void startThread(ExecutorService executorService) {
        executorService.submit(new OuterType$InnerType(this));
    }

    static synthetic String access$0(OuterType arg0) {
        return arg0.outerTypeString;
    }

}

class OuterType$InnerType implements Runnable {

    private OuterType this$0;

    OuterType$InnerType(OuterType arg0) {
        this.this$0 = arg0;
    }

    @Override
    public void run() {
        System.err.println("Hello from: " + this.this$0.outerTypeString);
    }
```

Objekterzeugung - Anonymous Inner classes

```
public class OuterType {

    String outerTypeString = "foo";

    public void startThread(ExecutorService executorService) {
        executorService.submit(new Runnable() {

            @Override
            public void run() {
                System.err.println("Hello from: " + OuterType.this.outerTypeString);
            }

        });
    }

}
```

Objekterzeugung - Anonymous Inner classes

```java
public class OuterType {

    String outerTypeString = "foo";

    public void startThread(ExecutorService executorService) {
        executorService.submit(new OuterType$1(this));
    }

}

class OuterType$1 implements Runnable {

    private OuterType this$0;

    OuterType$1(OuterType arg0) {
        this.this$0 = arg0;
    }

    @Override
    public void run() {
        System.err.println("Hello from: " + this.this$0.outerTypeString);
    }

}
```

Objekterzeugung - Double brace initialization

```java
public class InitializationTest {

    private Map<Integer, String> map = new HashMap<>();

    public InitializationTest() {
        this.map.put(1, "Eins");
        this.map.put(2, "Zwei");
        this.map.put(3, "Drei");
    }

    public void storeMap(MapStorageTarget target) {
        target.addMap(this.map);
    }

}
```

Objekterzeugung - Double brace initialization

```
public class InitializationTest {

    private Map<Integer, String> map = new HashMap<Integer, String>() {{
        put(1, "Eins");
        put(2, "Zwei");
        put(3, "Drei");
    }}

    public void storeMap(MapStorageTarget target) {
        target.addMap(this.map);
    }

}
```

Objekterzeugung - Double brace initialization

```java
public class InitializationTest {

    private Map<Integer, String> map = new InitializationTest$1<Integer, String>(this);

    public void storeMap(MapStorageTarget target) {
        target.addMap(this.map);
    }

}

class InitializationTest$1 extends HashMap<Integer, String> {

    private InitializationTest this$0;

    {
        this.put(1, "Eins");
        this.put(2, "Zwei");
        this.put(3, "Drei");
    }

    InitializationTest$1(InitializationTest arg0) {
        this.this$0 = arg0;
    }

}
```

Objekterzeugung - Initialization blocks

```
public class InitializationTest {

    private Object property = new Object();

    static {
        System.out.println("Static Initializer");
    }

    {
        System.out.println("Object Initializer");
    }

    InitializationTest() {
        System.out.println("Constructor");
    }

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

}
```

2

1

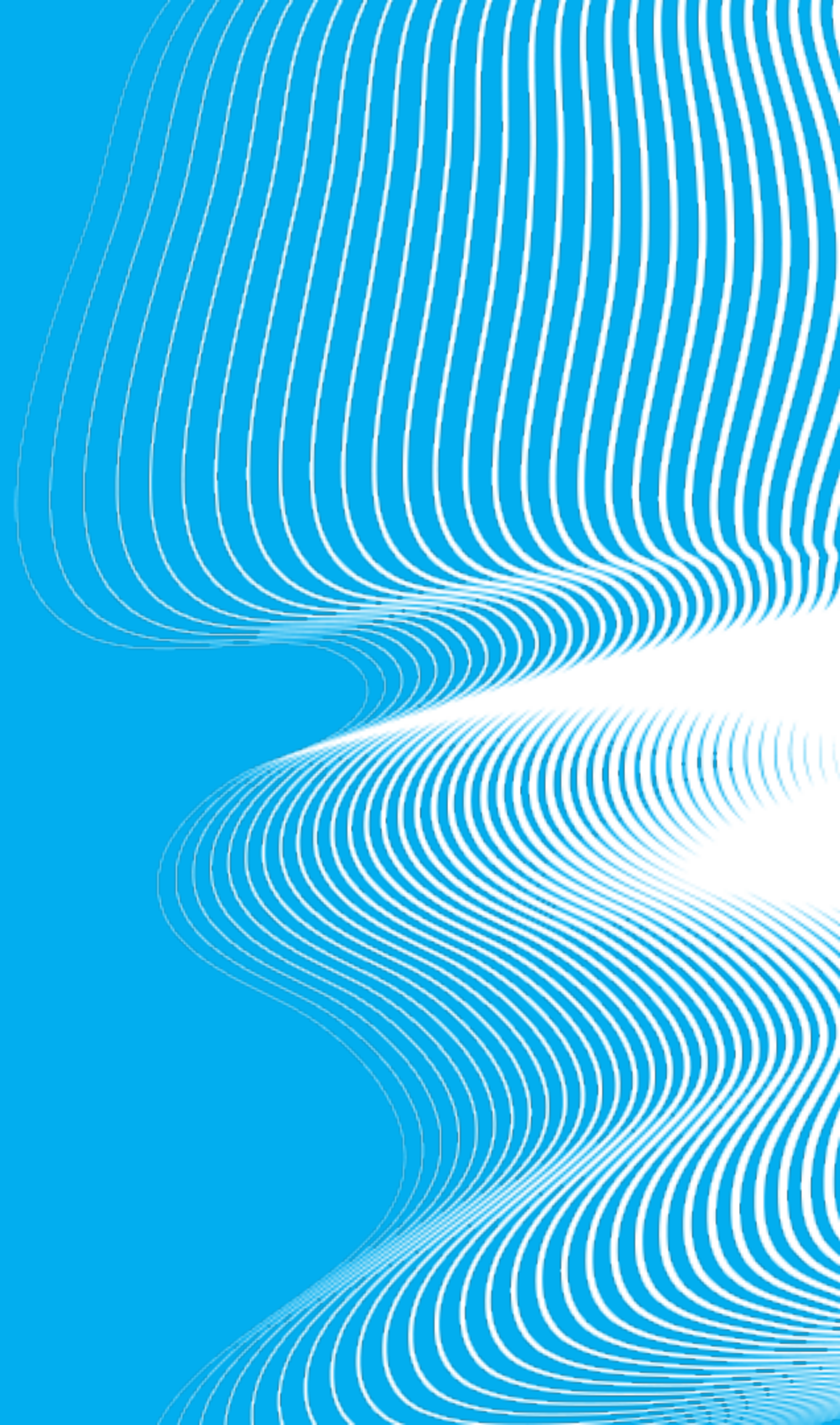
3

4

Objekterzeugung

- Achtung vor "versteckten" Referenzen
- Double Brace Initialization möglichst vermeiden

04 Cloning



Cloning

```java
public class Object {

    ...

    /**
     * Creates and returns a copy of this object. [...]
     *
     * By convention, the returned object should be obtained by calling
     * super.clone. [...]
     *
     * The method clone for class Object performs a specific cloning operation.
     * First, if the class of this object does not implement the interface Cloneable,
     * then a CloneNotSupportedException is thrown. [...]
     *
     * Otherwise, this method creates a new instance of the class of this object and
     * initializes all its fields with exactly the contents of the corresponding fields
     * of this object, as if by assignment; the contents of the fields are not
     * themselves cloned. Thus, this method performs a "shallow copy" of this object,
     * not a "deep copy" operation. [...]
     */
    protected native Object clone() throws CloneNotSupportedException;

    ...

}
```

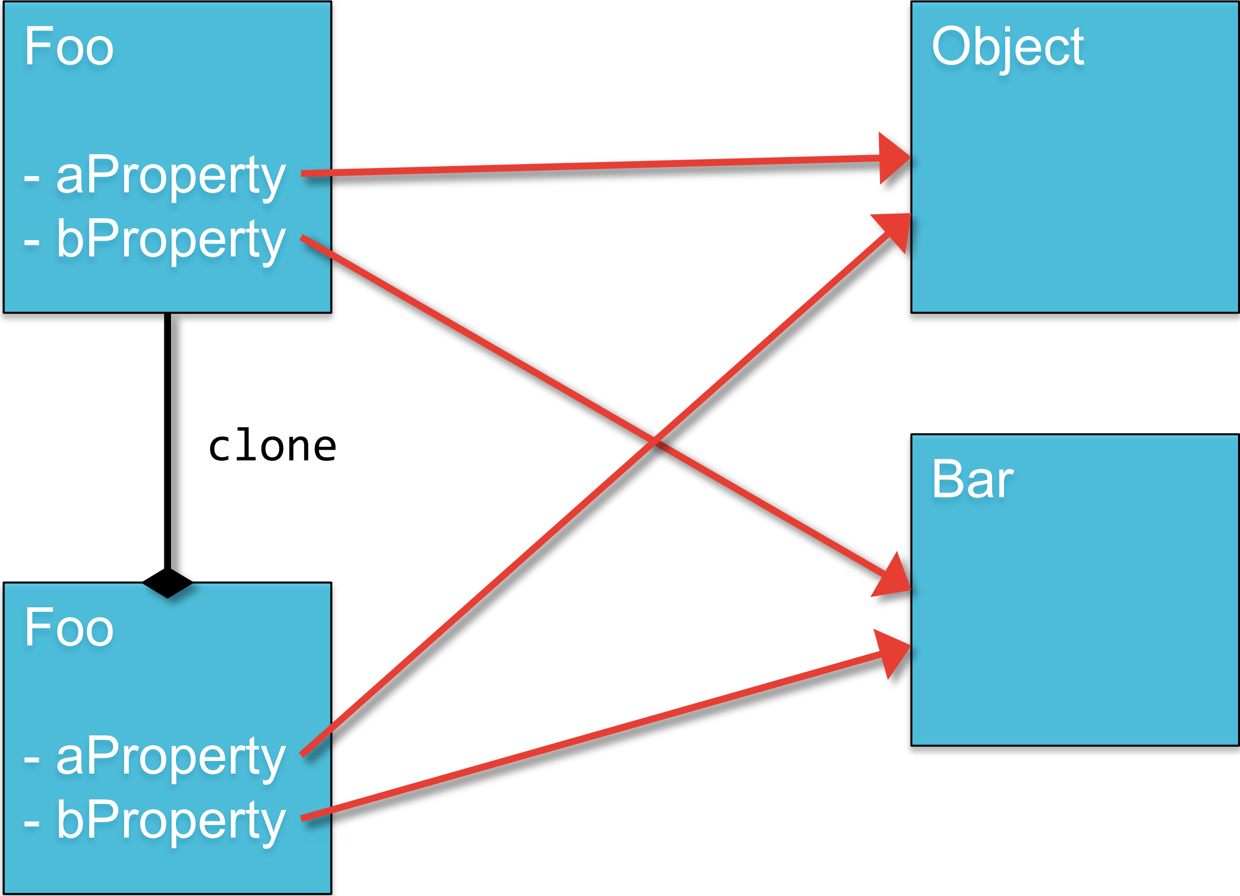
Cloning

```
public class Foo implements Cloneable {

    @Override
    public Foo clone() {
        try {
            Foo clonedFoo = (Foo)super.clone();
            return clonedFoo;
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Cannot clone Foo", e);
        }
    }

    public static void main(String[] args) {

        Foo originalFoo = new Foo();
        Foo clonedFoo = originalFoo.clone();

        Assert.assertFalse(originalFoo == clonedFoo);

    }

}
```

Cloning

```
public class Foo implements Cloneable {

    private Object aProperty = new Object();
    private Bar bProperty = new Bar();

    @Override
    public Foo clone() {
        try {
            Foo clonedFoo = (Foo)super.clone();
            return clonedFoo;
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Cannot clone Foo", e);
        }
    }

    public static void main(String[] args) {

        Foo originalFoo = new Foo();
        Foo clonedFoo = originalFoo.clone();

        Assert.assertFalse(originalFoo == clonedFoo);
        Assert.assertTrue(originalFoo.aProperty == clonedFoo.aProperty);
        Assert.assertTrue(originalFoo.bProperty == clonedFoo.bProperty);

    }
}
```

Cloning



Cloning - Usecase

```java
public class CustomerQueryExecutor {

    public List<Customer> executeQuery(CustomerQuery originalQuery) {

        CustomerQuery internalQuery = originalQuery.clone();

        if (!UserHelper.getCurrentUser().isAdministrator()) {
            internalQuery.setLoadUserDirectories(false);
            internalQuery.setLoadUserPermissions(false);
        }

        return this.executeSql(internalQuery.toSql());

    }

    ...

}
```

Cloning - Shallow clone vs. deep clone

```
public class ShallowCloneFoo implements Cloneable {

    private Bar aProperty = new Bar();

    @Override
    public ShallowCloneFoo clone() throws CloneNotSupportedException {
        return (ShallowCloneFoo)super.clone();
    }

}

public class DeepCloneFoo implements Cloneable {

    private Bar aProperty = new Bar();

    @Override
    public DeepCloneFoo clone() throws CloneNotSupportedException {
        DeepCloneFoo clonedFoo = (DeepCloneFoo)super.clone();
        clonedFoo.aProperty = this.aProperty.clone();
        return clonedFoo;
    }

}
```

Cloning - Shallow clone vs. deep clone

```java
public class DeepCloneFoo implements Cloneable {

    private Bar aProperty = new Bar();
    private Bazz anotherProperty = new Bazz();

    @Override
    public DeepCloneFoo clone() throws CloneNotSupportedException {
        DeepCloneFoo clonedFoo = (DeepCloneFoo)super.clone();
        clonedFoo.aProperty = this.aProperty.clone();
        clonedFoo.anotherProperty = ???;
        return clonedFoo;
    }

}
```

Cloning - Deep Cloning duch Serialisierung

```java
Foo originalFoo = new Foo();

ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(originalFoo);

byte[] fooBytes = bos.toByteArray();

ByteArrayInputStream bis = new ByteArrayInputStream(fooBytes);
ObjectInputStream ois = new ObjectInputStream(bis);

Foo clonedFoo = (Foo)ois.readObject();
```

Deep Cloning durch Serialisierung

- Serialisierung ist ca. 100x teuer als .clone()
- Nicht alle Klassen sind serialisierbar
- Die Welt ist nicht immer schwarz oder weiß bzw. deep oder shallow!
- Cloning durch Serialisierung entspricht eher der Strategie: Holzhammermethode!

Fragen?
Offene Punkte?

Vielen Dank!