# Worksheet: Cloning Variants

In class we have discussed how the clone method is implemented based on Java cloning, i.e. by using the functionality provided by method `Object.clone`. In this worksheet, four variants to this implementation are investigated and compared.

In all implementation variants we use the following class `Dictionary` to implement a clone method (which must return a *deep copy*).

```
public class Dictionary implements Cloneable {
    private String language;
    private final int size;
    private String[] words;

    public Dictionary(String language, int size) {
        this.language = language;
        this.size = size;
        this.words = new String[size];
        for (int i = 0; i < size; i++)
             this.words[i] = "sample word " + i;
    }

    @Override
    public Dictionary clone() {
        try {
             Dictionary d = (Dictionary) super.clone();
             if (words != null) {
                 d.words = words.clone();
             }
             return d;
        } catch (CloneNotSupportedException e) {
             throw new InternalError();
        }
    }
}
```

### (A) Cloning based on Copy-Constructors

If cloning is implemented with copy constructors, then in each class to be cloned a constructor has to be defined which takes a parameter of the same type (a so-called copy constructor).

```
class C {
    public C(C c) {
        if(c == null) throw new IllegalArgumentException();
        ...
    }
}
```

The new instance is initialized with the fields of the prototype instance passed to the copy constructor. For fields which are references to other object, one has to decide individually whether a deep or a shallow copy is appropriate. If the class is an extension of another class, the copy constructor of the base class is invoked first:

```
class D extends C {
    public D(D d){ super(d); … }
}
```

If a copy constructor exists in a class, the clone method (for example, for class C) can be implemented as follows:

```
    public Object clone() { return new C(this); }
```

Such a clone method has to be implemented in each class, as the method inherited from the base class returns objects of the wrong type.

**Tasks:**

1. Implement method `clone` for class `Dictionary` based on the copy-constructor approach described above.

2. Measure the efficiency of this method compared to the built-in Java cloning method.

3. Can this method be used to clone final fields (in class `Dictionary`, for example, the field `size` is declared final).

4. How do you implement a deep copy with this approach?

5. How does cloning based on copy constructors behave on
   a) cyclic structures?
   b) alias references?

   In package `patterns.clone.alias` you will find two small classes, in which data structures with a cycle or with a rhombic structure are generated. Copy these data structures with copy constructor cloning and check whether the structure is preserved.

6. Should a class that implements cloning using copy constructors also implement the `Cloneable` interface?

### (B) Reflective Clone

For the reflective variant, a copy is created with the help of Java reflection. The `clone` method in class C invokes the cloning functionality provided by class `ReflectiveClone` and looks as follows:

```
public Object clone () {
   return ReflectiveClone.clone(this);
}
```

The implementation of class `ReflectiveClone` can be found in this week's project.

Method `clone` defined in this class uses functions from the Java reflection library to make a copy of the given object. Package `java.lang.reflect.*` contains methods to examine objects at runtime. The key entry point to this functionality is class `Class`. Below a few examples are shown:

- Access to the fields of an object `x`:

```
Class objClass = x.getClass();
Field[] fields = objClass.getDeclaredFields();
```

- Access to the value of a particular field on an instance `x`:

```
Field f = fields[0];
Object value = f.get(x)
```

Additional information about reflection can be found in the reflection tutorial at http://docs.oracle.com/javase/tutorial/reflect/

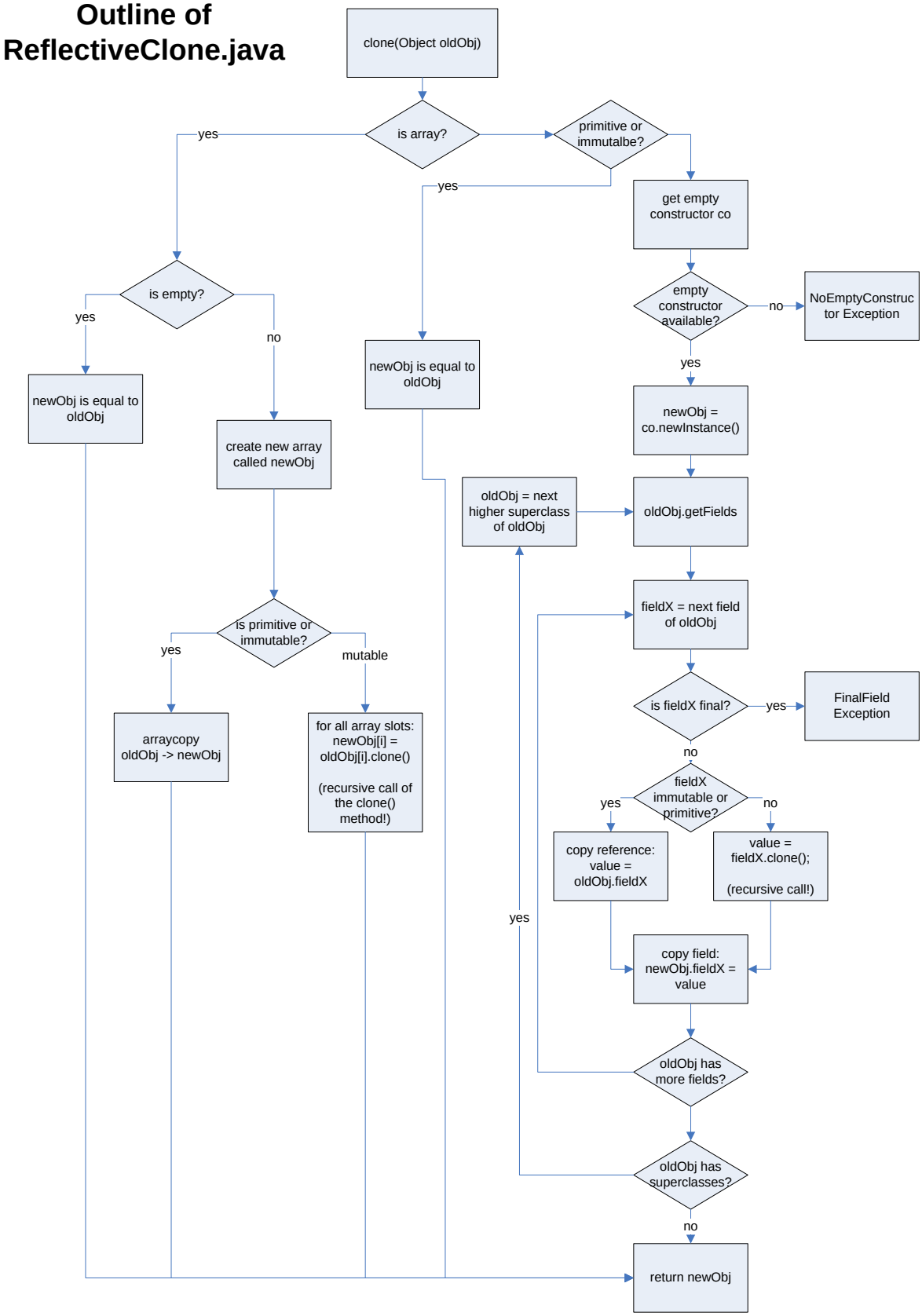
Reflection can be used to copy objects generically. Reflection allows to access the constructor of a class and to create a new instance using this constructor. Then the fields of this new instance can be set. This also works for protected and for private fields. Fields with a primitive type (`double`, `int`, ...) or fields that refer to immutable object (such as a `String`) are copied, while the other objects are recursively cloned. This idea is simple, but to get it right, a few details need to be considered. An implementation can be found in class `RecursiveClone`. A flowchart which describes how this method works is shown on the back side.

**Tasks:**

1. Study class `ReflectiveClone` and try to understand how method `clone` works. You need not to understand every detail. It is sufficient if you roughly understand the process. The flowchart on the back side may help to understand the code.
   Questions:
   - Is the copy shallow or a deep by default?
   - Where does the code distinguish between references to mutable objects and references to immutable objects or primitive fields? What is the purpose of this distinction?
2. Compare the speed of this cloning variant with the built-in Java cloning method.
3. Does this method support the cloning of final fields (in class `Dictionary`, for example, the field `size` is declared final)?
4. What happens if classes do not contain a default constructor?; e.g. if a field of type `UUID` is defined in your class (class `UUID` does not declare a default constructor)?
5. How does `ReflectiveClone` behave on
   a) cyclic structures?
   b) alias references?

   In package `patterns.clone.alias` you will find two small classes, in which data structures with a cycle or with a rhombic structure are generated. Copy these data structures with reflective cloning and check whether the structure is preserved.
6. Can `ReflectiveClone` deal with inner (non-static) element classes?
7. Should a class that implements cloning using `ReflectiveClone` also implement the `Cloneable` interface?

# Outline of ReflectiveClone.java



### (C) Cloning based on Java Serialization

Java Serialization allows to convert an object (including referenced objects) into a byte stream, and vice versa, to convert a serialized object graph back into Java objects.

This technique can be used to create a copy of an object. For this purpose, the object graph is written into a byte array in main memory via a stream and then read from it again. A clone method which uses this approach looks as follows:

```
public Object clone () {
    try {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        ObjectOutputStream oout = new ObjectOutputStream(out);
        oout.writeObject(this);

        ObjectInputStream oin = new ObjectinputStream(
            new ByteArrayInputStream(out.toByteArray()));

        return oin.readObject();
    }
    catch (Exception e)       {
        throw new RuntimeException ("cannot clone class");
    }
}
```

This method only works if the class which declares this clone method and the classes of all referenced objects are declared to be serializable (by implementing the marker interface `java.io.Serializable`).

**Note:**
Java serialization is convenient. Classes can be made serializable by implementing the `Serializable` marker interface. If a class is derived from a non-serializable class and declared as serializable, the base class must have a default constructor.

**Tasks:**

1. Compare the speed of this cloning variant with the built-in Java cloning method.

2. Is the default constructor called when deserializing an object? And how does this variant behave if the class does not contain a parameterless constructor of if such a constructor is declared private?

3. Does this method support the cloning of final fields (in class `Dictionary`, for example, the field `size` is declared final)?

4. What happens with fields of type `String`? Is a deep or a shallow copy performed on such objects?

5. How does cloning based on Java serialization behave on
   a) cyclic structures?
   b) alias references?

   In package `patterns.clone.alias` you will find two small classes, in which data structures with a cycle or with a rhombic structure are generated. Copy these data structures with copy constructor cloning and check whether the structure is preserved.

6. Should a class that implements cloning using serialization also implement the `Cloneable` interface?

### (D) Cloner Clone

At https://github.com/kostaskougios/cloning a small Java library is available which allows to copy object structures deeply. This library is already contained in this week's project (and the sources are also already linked).

The copy of a data structure can be created using method `deepClone` declared in class `Cloner`. An instance of class `Cloner` can be created using method `Cloner.standard()`.

The cloning library uses Java Reflection to create copies. Package `java.lang.reflect.*` contains methods to examine objects at runtime. The key entry point to this functionality is class `Class`. Below a few examples are shown:

- Access to the fields of an object `x`:

```
Class objClass = x.getClass();
Field[] fields = objClass.getDeclaredFields();
```

- Access to the value of a particular field on an instance `x`:

```
Field f = fields[0];
Object value = f.get(x)
```

Additional information about reflection can be found in the reflection tutorial at http://docs.oracle.com/javase/tutorial/reflect/

Reflection can be used to copy objects generically. Reflection allows to access the constructor of a class and to create a new instance using this constructor. Then the fields of this new instance can be inspected throughout the whole class hierarchy and can be set. This also works for protected and for private fields. Fields with a primitive type (`double`, `int`, ...) or fields that refer to immutable object (such as a String) are copied, while the other objects are recursively cloned. This idea is simple, but to get it right, a few details need to be considered.

The implementation of method `clone` using this cloning library looks as follows:

```
public Object clone () {
   return Cloner.standard().deepClone(this);
}
```

**Tasks:**

1. Compare the speed of this cloning variant with the built-in Java cloning method.

2. Does this method support the cloning of final fields (in class `Dictionary`, for example, the field `size` is declared final)?

3. What happens if classes do not contain a default constructor, e.g. if a field of type `UUID` is defined in your class (class `UUID` does not declare a default constructor)?

4. How does `deepClone` behave on
   a) cyclic structures?
   b) alias references?

   In package `patterns.clone.alias` you will find two small classes, in which data structures with a cycle or with a rhombic structure are generated. Copy these data structures with `deepClone` and check whether the structure is preserved.

5. Can `deepClone` deal with inner (non-static) element classes?

6. Try to understand how method `deepClone` works. In addition to method `deepClone`, class `Cloner` also provides method `shallowClone`. How does the code internally differentiate between performing a deep clone and a shallow clone?

7. Should a class that implements cloning using `deepClone` also implement the `Cloneable` interface?