

17 Object serialization and copying

Overview

- Object serialization
 - Object streams
 - Object duplication - cloning
- In a following lecture:*
- Equality and identity relations
 - Algebraic properties
 - Comparison methods and objects for ordering relations

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 2

Well behaved classes

- Objects that are handled by the JVM and many standard classes should have
 - No-arg constructor
 - String representation (toString())
 - Serialization (for stream i/o)
 - Cloning (deep copying)
 - Equality and hashCode methods

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 3

Object i/o

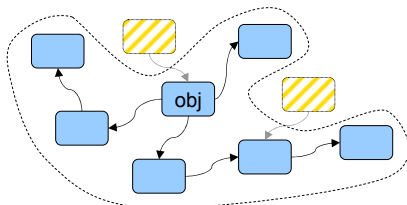
- Whole networks of inter-connected objects may be "flattened" and written to object streams
 - and later read back into the program again.
- Typical application: Saving the program state for later resumption, e.g. in computer games.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 4

Serialization

- The *object graph* of obj consists of obj and all objects that are directly or indirectly referenced from it.



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 5

Serialization - deserialization

- When serializing an object a *linear representation* of the object graph is built.
- Deserialization means reconstruction of an object graph from a linear representation.
- A class declares that instances may be serialized by implementing

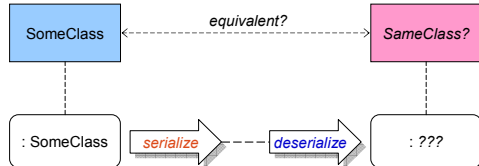
interface serializable

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 6

Serialization - deserialization (2)

- How can the runtime environment verify that the class used when deserializing an object is compatible with the class used when serializing it?



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 7

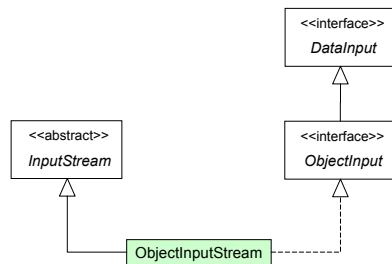
Class version unique identifiers

- The run-time system associates a default **serialVersionUID** with each serializable class.
- In case of mismatch **InvalidClassException** is thrown on deserialization.
- The **serialVersionUID** can (should) be declared explicitly.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 8

Object input stream class relations



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 9

Object input operations

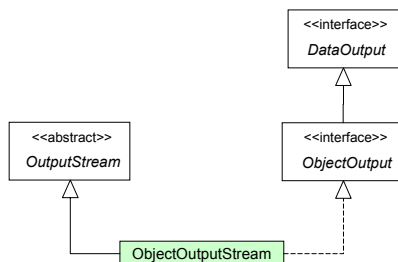
```
interface ObjectInput
Object readObject() throws *
...
+ many other methods
```

* **ClassNotFoundException**, **InvalidClassException**, **StreamCorruptedException**, **OptionalDataException**, **IOException**

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 10

Object output stream class relations



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 11

Object output operations

```
interface ObjectOutput
void writeObject() throws *
...
+ many other methods
```

* **InvalidClassException**, **NotSerializableException**, **IOException**,

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 12

Example: Adventure game

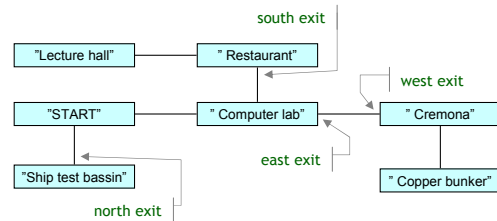
- Rooms can be created, connected, and explored.
- Rooms contain information.
- Additional information can be added.
- The *room graph* can be saved in a file and reloaded in a future execution.
- Explore the *labyrinth* project.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 13

Adventure game (2)

- The noncyclic room graph is a *tree*.
- Room connections are navigable two-ways.

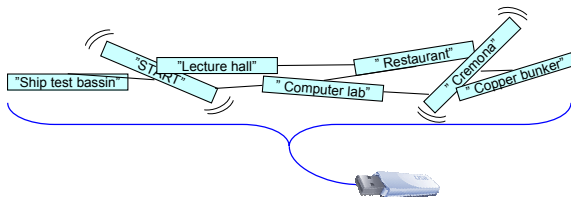


Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 14

"Graph crushing"

- The room graph is *serialized* and written to an *object stream* connected to a file.



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 15

class Room

```
public class Room implements Serializable {
    private String description;
    private HashMap<String, Room> exits;

    public String getInfo() {...}
    public void addInfo(String info) {...}
    public Room getExit(String direction) {...}
    public void connect(String direction, Room room) {...}
}

// Directions: "north", "south", "east", "west"
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 16

class Labyrinth

```
public class Labyrinth {
    private static class State implements Serializable {
        Room start = new Room("START");
        Room current = start;
    }
    private static State state = new State();

    public void walk(String direction) {...}
    public void addInfo(String comment) {...}
    public void printInfo() {...}
    public void printExits() {...}
    public void save(String fileName) {...}
    public void load(String fileName) {...}
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 17

Labyrinth.save()

```
public void save(String fileName)
{
    try {
        ObjectOutputStream out =
            new ObjectOutputStream(
                new FileOutputStream(fileName));
        out.writeObject(state);
    }
    catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
    }
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 18

Labyrinth.load()

```
public void load(String fileName) {
    try {
        ObjectInputStream in =
            new ObjectInputStream(
                new FileInputStream(fileName));
        state = (State)in.readObject();
    }
    catch(Exception e) {
        e.printStackTrace();
        System.exit(0);
    }
}
```

Deserialization

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 19

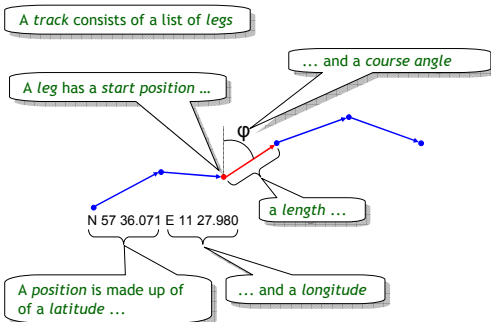
Class Object

- The class `Object` defines basic default implementations of
 - `public String toString()`
 - `protected Object clone()` *next!*
 - `public boolean equals(Object other)`
 - `public int hashCode()`
- They can be overridden in subclasses.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 20

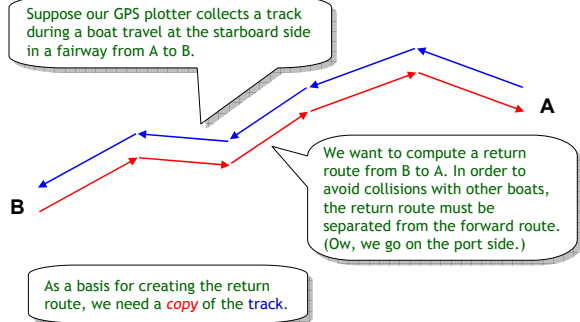
Ex. Object model for GPS tracks



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 21

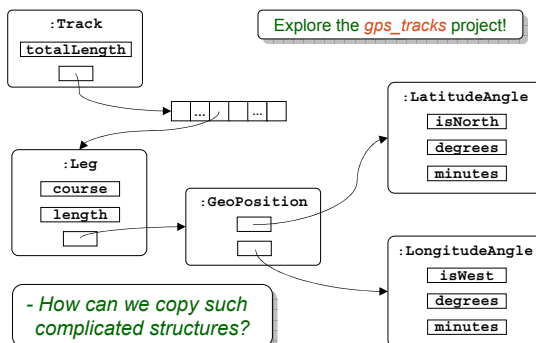
Ex. Object model for GPS tracks



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 22

GPS track object configuration



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 23

The `Object.clone()` method

- The class `Object` defines a default implementation of
 - `protected Object clone()`
- It returns a *shallow* field-by-field copy of the calling object.
 - it does *not* copy referenced objects.
- `clone` should be overridden:
 - `public MySubClass clone()`

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 24

Recommended properties

- It is recommended that overridings of clone satisfy the following properties for all x:
- Weak independence
`x.clone() != x`
- Equality
`(x.clone()).equals(x)`
- Type identity
`(x.clone()).getClass() == x.getClass()`

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 25

The Cloneable interface

- By implementing **interface Cloneable** a class declares that `Object.clone` may copy instances of the class field-by-field.
- CloneNotSupportedException** is thrown if `Object.clone` is called for instances not implementing **Cloneable**
- Classes which implement **Cloneable** should override `clone`.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 26

Properties of Object.clone

- When **Object.clone** is called for an instance of a class
 - it will return a field-by-field copy of the *whole instance*.
 - the copy will have the *same dynamic type* as the calling instance, this applies also to overridden clone methods in subclasses.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 27

The Cloneable interface

```
public class MyClass implements Cloneable
{
    private int x = 123;
    private boolean y = true;
    private char z = 'A';

    @Override
    public MyClass clone() throws CloneNotSupportedException
    {
        return (MyClass)super.clone();
    }
}
...
MyClass x = new MyClass();
MyClass copy = x.clone();
```

*Object.clone copies x, y, z
It returns an object having static type
Object, but dynamic type MyClass.*

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 28

The Cloneable interface (2)

```
public class MyClass // implements Cloneable
{
    private int x = 123;
    private boolean y = true;
    private char z = 'A';

    @Override
    public MyClass clone() throws CloneNotSupportedException
    {
        return (MyClass)super.clone();
    }
}
...
MyClass cc = new MyClass();
MyClass copy = cc.clone();
```

*Object.clone throws
CloneNotSupportedException*

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 29

The Cloneable interface (3)

```
public class NoCopy
{
    private int x = 123;
    private boolean y = true;
    private char z = 'A';

    @Override
    public NoCopy clone() throws CloneNotSupportedException
    {
        throw new CloneNotSupportedException();
    }
}
...
NoCopy nc = new NoCopy();
nc.clone();
```

*Use this scheme if instances
of a class may not be cloned*

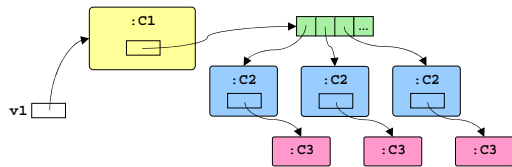
throws CloneNotSupportedException

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 30

Deep and shallow copying

Suppose we want a copy of the entire object graph referenced by v1



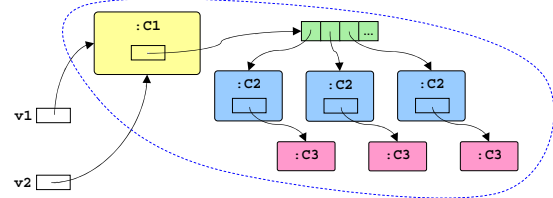
Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 31

The first attempt - Shallow copy

A simple assignment of an object reference is in most cases inadequate - some objects will be shared

`v2 = v1;`



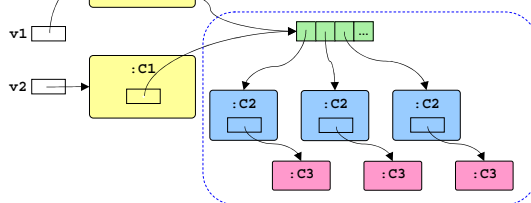
Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 32

The second attempt

`v2 = v1.clone();`

C1.clone() calls super.clone() (as in previous slides) - the list is shared



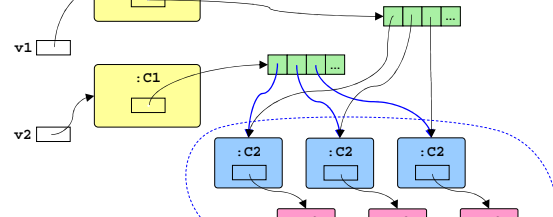
Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 33

The third attempt

`v2 = v1.clone();`

Inadequately overridden clone in C1 copies the list but not the list elements - the list elements become shared



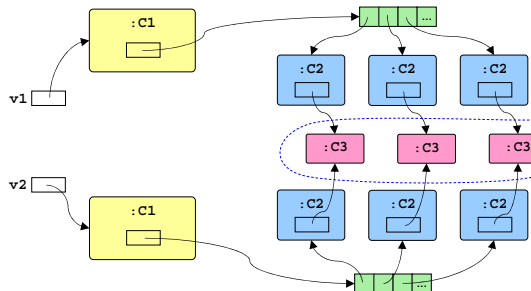
Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 34

The fourth attempt

`v2 = v1.clone();`

Overridden clone in C1 copies the list and the list elements correctly - but C2 does just a shallow copy!



Object oriented programming, DAT042, D2, 11/12, lp 1

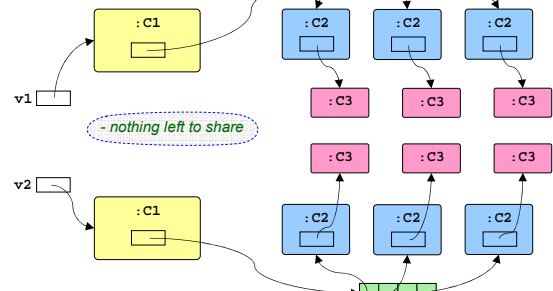
Lecture 17 35

The final version

Finally, also C2 copies deeply

`v2 = v1.clone();`

- nothing left to share



Object oriented programming, DAT042, D2, 11/12, lp 1

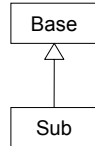
Lecture 17 36

Cloning and inheritance

- Redefined clone methods in subclasses calls **super.clone()**.

```
class Sub extends Base
{
    @Override
    public Sub clone() {
        Sub result = (Sub)super.clone();

        ...
        return result;
    }
    ...
}
```



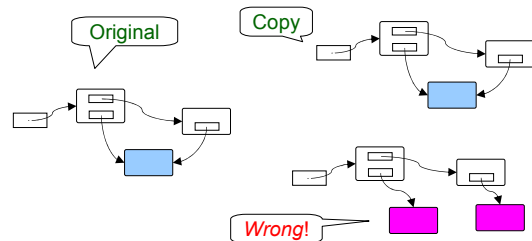
Add copies of sub class specific fields to result.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 37

Copying cyclic structures

- In cyclic structures and structures where parts are shared, care must be taken to preserve the sharing patterns of the original in the copy.



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 38

Ex. Angle clone method

```
public class Angle implements Cloneable {
    private int degrees;
    private float minutes;
    ...
    @Override
    public Angle clone() {
        try {
            return (Angle)super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
    ...
}
```

Explore the [gps_tracks](#) project!

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 39

Ex. LatitudeAngle clone method

```
public class LatitudeAngle extends Angle {
    private boolean isNorth;
    ...
    @Override
    public LatitudeAngle clone() {
        return (LatitudeAngle)super.clone();
    }
    ...
}

// LongitudeAngle similar
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 40

Ex. GeoPosition clone method

```
public class GeoPosition implements Cloneable {
    private LatitudeAngle latitude;
    private LongitudeAngle longitude;
    ...
    @Override
    public GeoPosition clone() {
        GeoPosition copy = null;
        try {
            copy = (GeoPosition)super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
        copy.latitude = latitude.clone();
        copy.longitude = longitude.clone();
        return copy;
    }
    ...
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 41

Ex. Leg clone method

```
public class Leg implements Cloneable {
    private GeoPosition startPos;
    private int course;
    private float length;
    ...
    @Override
    public Leg clone() {
        Leg copy = null;
        try {
            copy = (Leg)super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
        copy.startPos = startPos.clone();
        return copy;
    }
    ...
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 42

Ex. Track clone method

```
public class Track implements Cloneable {
    private float totalLength;
    private ArrayList<Leg> legs;
    ...
    @Override
    @SuppressWarnings("unchecked")
    public Track clone() {
        Track copy = null;
        try {
            copy = (Track)super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
        copy.legs = (ArrayList<Leg>)legs.clone();
        for ( int i = 0; i < legs.size(); i++ )
            copy.legs.set(i, legs.get(i).clone());
        return copy;
    }
    ...
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 43

References

- Angelica Langer, Das Kopieren von Objekten in Java,
<http://www.angelikalanger.com/Articles/EffectiveJava/05.Clone-Part1/05.Clone-Part1.html>
<http://www.angelikalanger.com/Articles/EffectiveJava/06.Clone-Part2/06.Clone-Part2.html>
<http://www.angelikalanger.com/Articles/EffectiveJava/07.Clone-Part3/07.Clone-Part3.html>

(In German)

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 17 44