Recitations No. 5:
Java IO (Custom Serialization), Refactoring

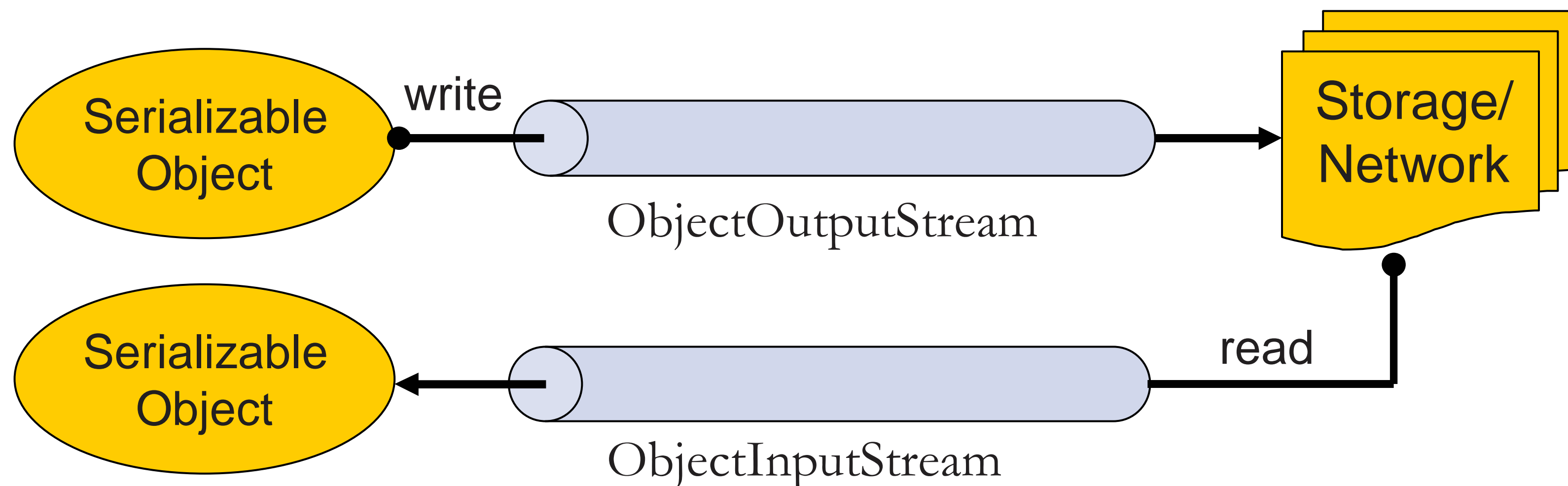
March 24, 2005 Oranit Dror

Object Serialization

- A mechanism that enable objects to be:
  - saved and restored from byte streams
  - persistent (outlive the current process)
- Useful for:
  - persistent storage
  - sending an object to a remote computer

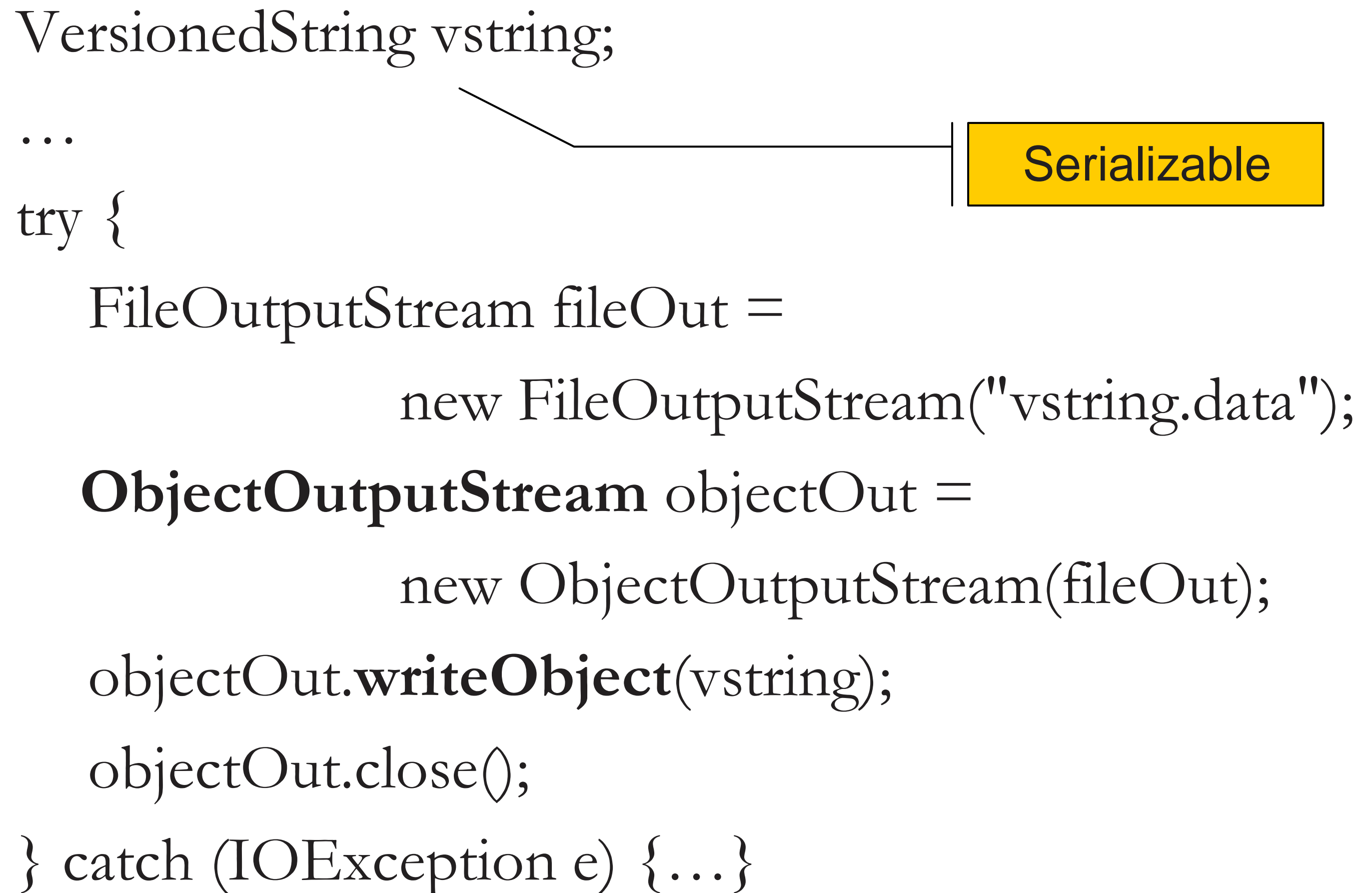
The Default Mechanism

- The default mechanism includes:
  - The Serializable interface
  - The ObjectOutputStream
  - The ObjectInputStream

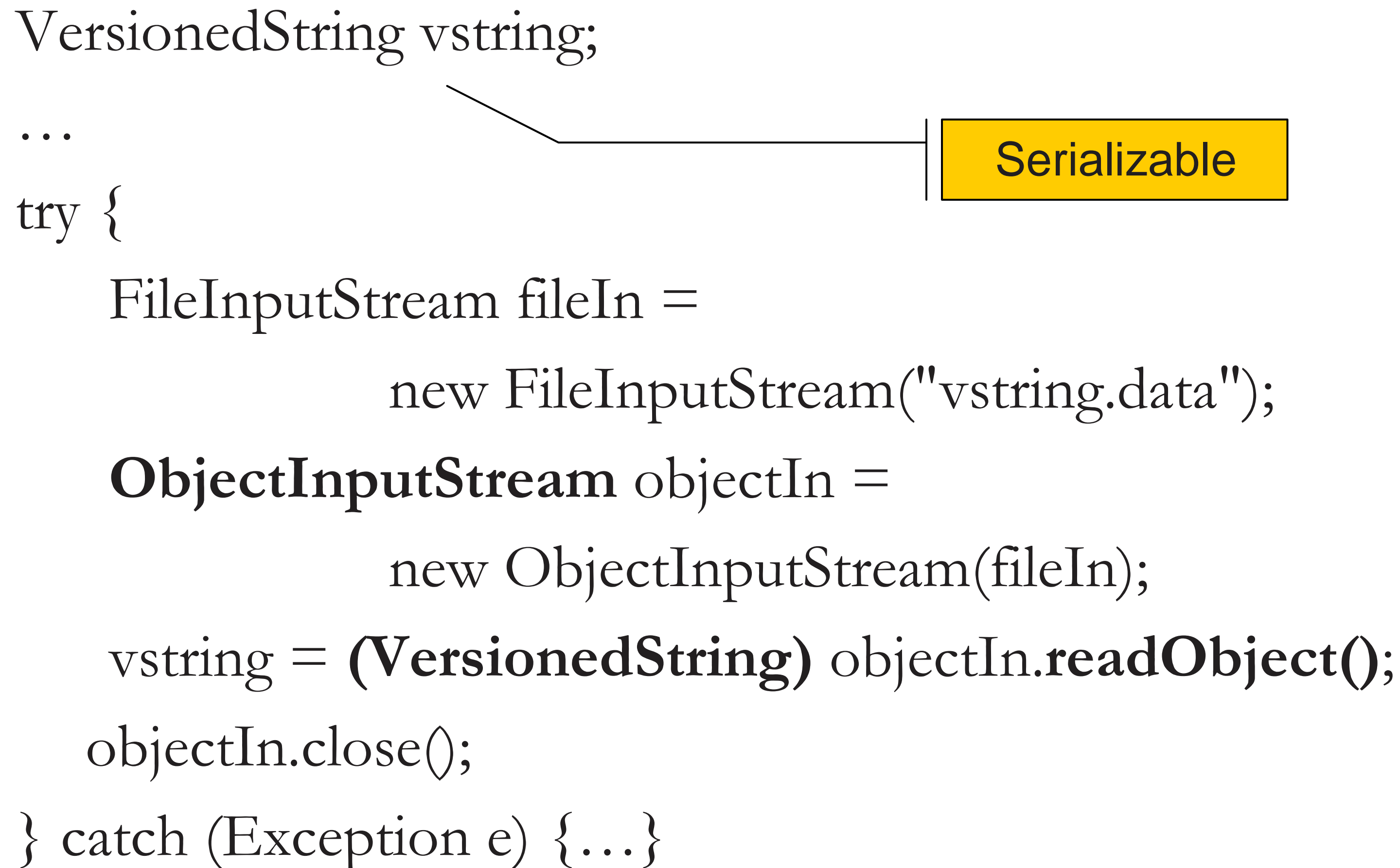


Writing Serializable Objects

```
VersionedString vstring;
…
try {
    FileOutputStream fileOut =
                    new FileOutputStream("vstring.data");
    ObjectOutputStream objectOut =
                    new ObjectOutputStream(fileOut);
    objectOut.writeObject(vstring);
    objectOut.close();
} catch (IOException e) {…}
```



Reading Serializable Objects



Customized Serialization

- Customized Serialization is useful for:
  - Recomputing unserialized field values
  - Transferring encrypted data
  - Improving Efficiency
  - Storage Savings

Custom Serialization

- The class to be serialized must implement:
  - **private** void **writeObject**(ObjectOutputStream s) throws IOException;
  - **private** void **readObject**(ObjectInputStream s) throws IOException, ClassNotFoundException;

• are invoked during serialization/deserialization
Java Kludge Methods:
- are not defined in any interface
- are private

Custom Serialization

```
public class ArrayListVersionedString extends VersionedString {
    ArrayList list = new ArrayList();

    private void writeObject(ObjectOutputStream out)
            throws IOException {
        list.trimToSize();
        out.defaultWriteObject();
    }

    private void readObject(ObjectInputStream in)
            throws IOException, ClassNotFoundException {
        in.defaultReadObject();
    }
    …
}
```

Serializable

For saving storage or network transportation

Invoking the default mechanism

Custom Serialization

- No need to restore the superclass' fields:

```
public abstract class VersionedString implements Serializable {
    protected Date creationDate = new Date();
    public Date getCreationDate() {return creationDate;}
    ….
}

public class ArrayListVersionedString extends VersionedString {
    ArrayList list = new ArrayList();

    private void writeObject(ObjectOutputStream out)
            throws IOException { // nothing }

    private void readObject(ObjectInputStream in)
            throws IOException, ClassNotFoundException { // nothing }
}
```

Custom Serialization

```
public static void main (String[] args) {
    VersionedString vstring = new ArrayListVersionedString();
    vstring.add("Version 1");
    System.out.println("date = " + vstring.getCreationDate());

    Thread.sleep(…)

    // Storing the vstring object in a file
    …

    // Restoring the vstring object from the file
    …

    System.out.println(“”after restoring:”);
    System.out.println("date = " + vstring.getCreationDate());
    System.out.println(“last version = " + vstring.getLastVersion());
}
```

```
date = Tue Dec 14 12:41:24 IST 2004
after restoring:
date = Tue Dec 14 12:41:24 IST 2004
java.lang.NullPointerException
```

Stopping the Serialization

- Defining a non-serializable class whose superclass is serializable:

```
public class ArrayListVersionedString extends VersionedString {

    private void writeObject(ObjectOutputStream out)
            throws IOException {
        throw new NotSerializableException();
    }

    private void readObject(ObjectInputStream in)
            throws IOException, ClassNotFoundException {
        throw new NotSerializableException();
    }
    …
}
```

Serializable

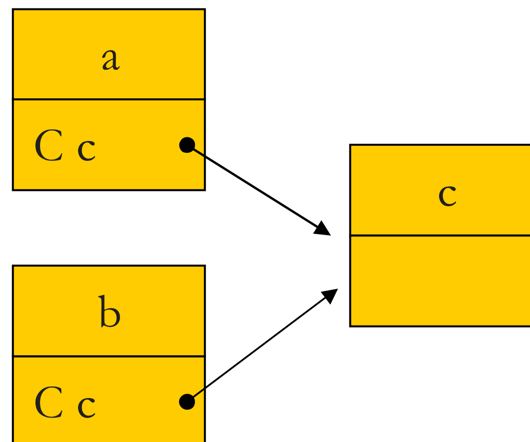
Not elegant

The Externalizable Interface

- For a complete control over serialization
- Extends the Serializable interface by:
  - **public** void writeExternal(ObjectOutput out) throws IOException
  - **public** void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
- You are responsible for the superclass's fields
- Supercedes custom serialization
- Should have a public no-argument constructor

Shared References

- Consider the following case:
  - We have 3 serializable objects a, b and c
  - Object c is shared by a and b
- When serializing a and b, each of them will try to serialize c



Shared References

- What will happen when deserialing a and b?
- If a and b were serialized in the same stream without calling ObjectOutputStream.reset(), they will contain a reference to the same object c
- ObjectInput/OutputStream use keys and a map to ensure object uniqueness
- The keys are stream-specific

Class Versioning

- Assume you have two versions of the same class
- The two versions will be behave the same if …
  - the public/package/protected members are the same
  - the private members and implementation can be different

Class Versioning

- What will happen if a version of a class reads data written by another version?
- An InvalidClassException will be thrown
- Even if only the private members of the two versions are different.
- How does the VM know that the classes are different if it doesn't have a copy of the original class?

Class Versioning

- Java tracks the classes' version numbers
- A version number of a class is:
  - a long value
  - computed from the names of all the data and method members of the class
- When serializing a class, its version number is stored in the stream.
- When deserializing a class, its version number is compared to the one in the stream

Class Versioning

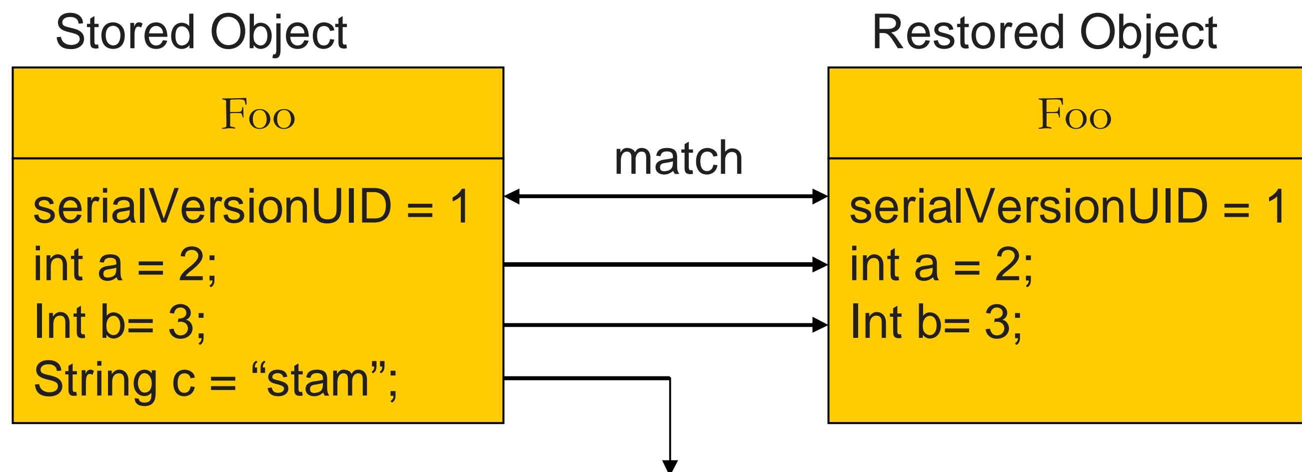
- To short-circuit this mechanism
  - provide your own version number:

    ```
    static final long serialVersionUID = 1;
    ```

  - Use the **serialver** tool to obtain version numbers of old classes
- Your responsibility is to make sure that the two class versions are compatible.
- How do I know that they are compatible?

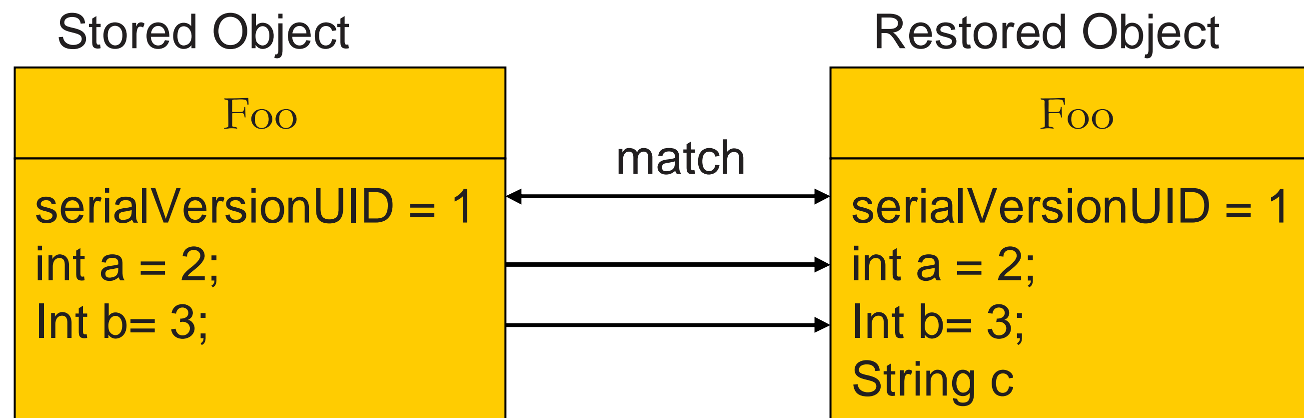
Class Compatibility

- Only changes to fields affect the class compatibility (for serialization)
- Removing a field:
  - No affect after restoring the object



Class Compatibility

- Adding a field:
  - The new field gets no value, even from if it has a default value



Class Compatibility

- Changing the name of a field:
  - introduces both scenarios
- Marking a field as transient
  - causes similar behavior

Other IO Topics

- The `java.nio` package
- The `java.util.zip` package

Refactoring

- Changing the structure of a software without changing its functionality
- Comes from the *factorization* math term:
  - $x^2 - a^2 = (x-a)\cdot(x+a)$
- A kind of code reorganization
- Not code rewriting
- Done in small steps with testing
- An aspect of extreme programming

Eclipse Refactoring

- An automated tool:
  - More efficient
  - Reduce the risk of introducing bugs
- Online Tutorial:

  http://www.cs.umanitoba.ca/~eclipse/13-Refactoring.pdf
- Refactoring types:
  - Physical Organization
  - Logical Organization
  - Restructure within a class

Eclipse Refactoring
Physical Reorganization

- Rename variables, fields, methods, classes, interfaces, packages etc.
- Move packages, classes, methods etc.
- Change Method Signature
- Convert Anonymous Class to Nested
- Move Member Type to New File

Eclipse Refactoring
Logical Reorganization

- Push Down
- Pull Up
- Extract Interface
- Generalize Type
- Use Supertype Where Possible

Eclipse Refactoring
Restructure within a class

- Inline
- Extract Method
- Extract Local Variable
- Extract Constant
- Introduce Parameter
- Introduce Factory
- Convert Local Variable to Field
- Encapsulated Fields

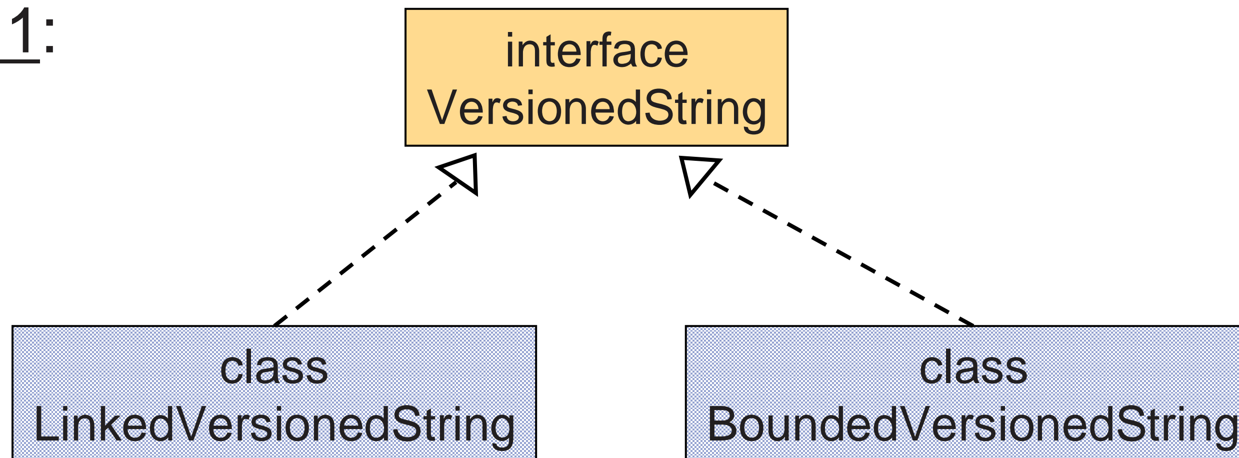
Eclipse Refactoring
Demo

Initial Stage:

class
VersionedString

class
BoundedVersionedString

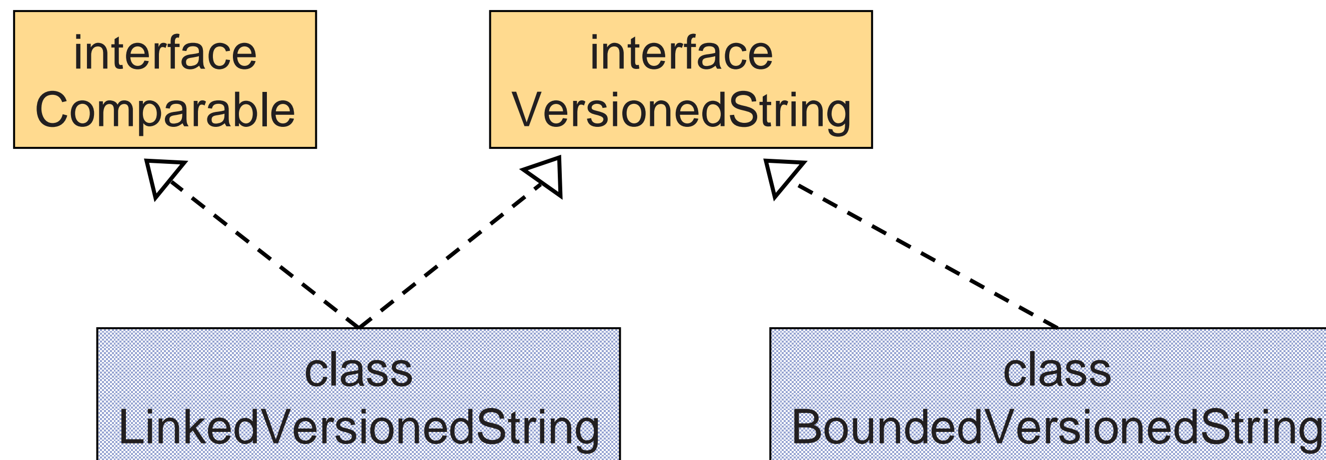
Stage 1:



Eclipse Refactoring
Demo (cont.)

Stage 2:
Sort LinkedVersionedString objects by:

```
void insertionSort(Comparable a[]);
```



Eclipse Refactoring
Demo (cont.)

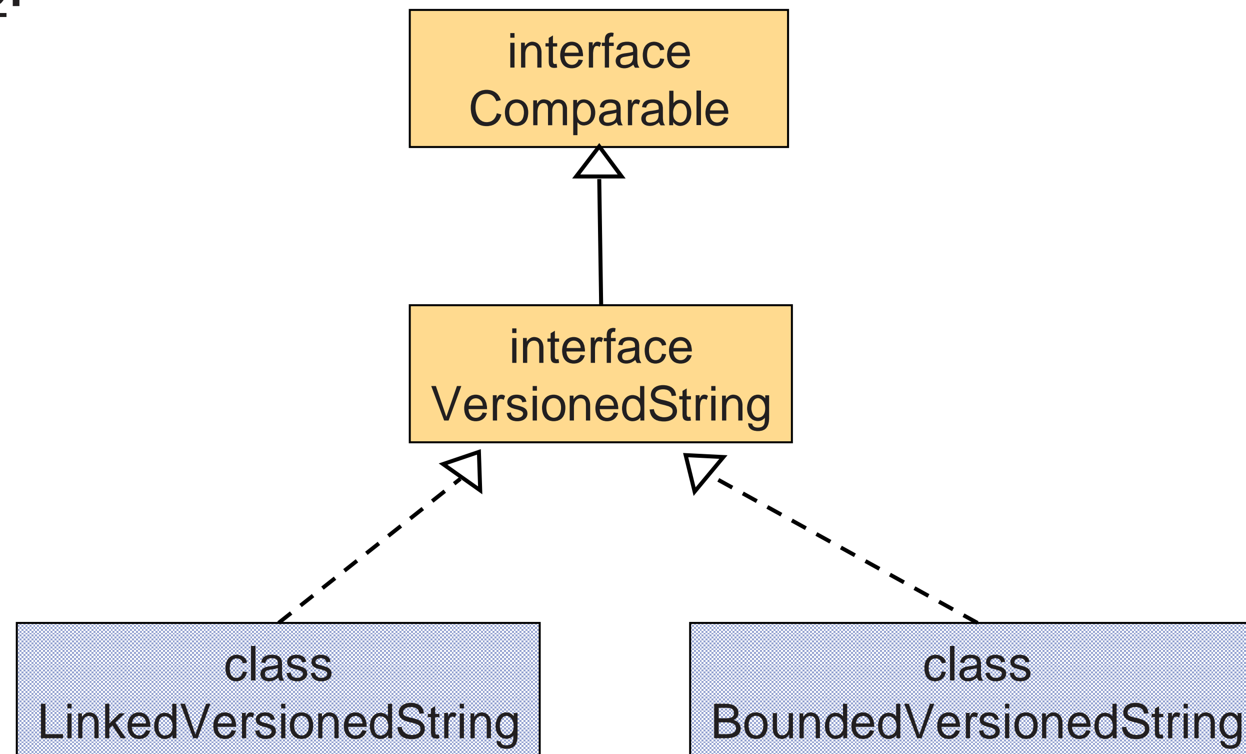
Stage 2 (cont.):

```
public class LinkedVersionedString  implements VersionString, Comparable {
    ….
    public int compareTo(Comparable other) throwsIncomparableException {
            LinkedVersionedString other_vs;
            try {
                    other_vs = (LinkedVersionedString) other;
            } catch (java.lang.ClassCastException ce) {
                    throw new IncomparableException();
            }

            if (this.length() > other_vs.length())  return  1;
            if (this.length() < other_vs.length()) return -1;
            return 0;
    }
}
```

Eclipse Refactoring
Demo (cont.)

Stage 3:



Eclipse Refactoring
Demo (cont.)

Stage 4: