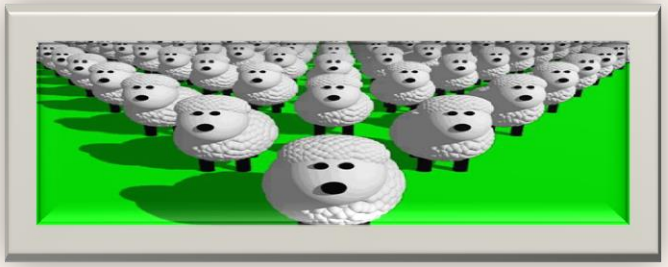
Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 12

Serialization

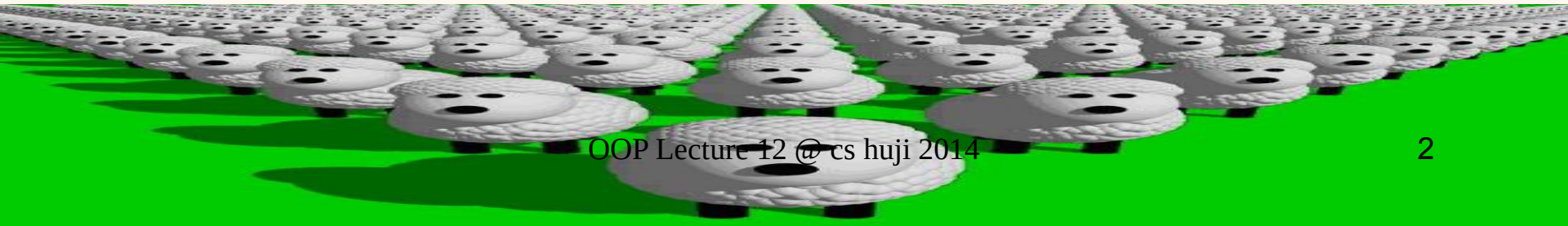
Cloning



Java reflections

Duplicate an Object … Why?

- Storing Data to disk to continue later
- Transfer Data
- Backup
- Making cut & paste copies with required changes
- Generating instances of some “template” object



Goal 1:

Write (save) an Object to a Stream

Recursive saving required

```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<BankAccount> accounts;
}
```

Customer:

name

address

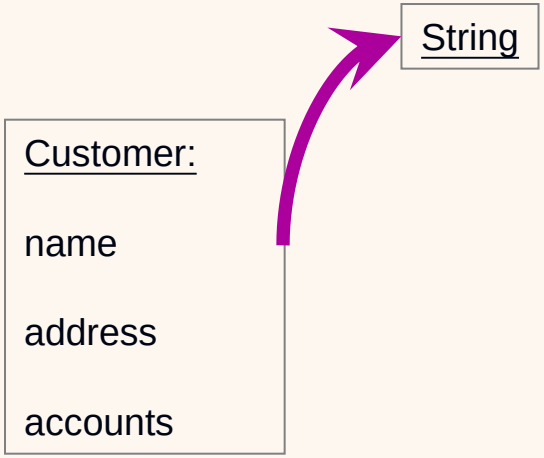
accounts

Goal 1:

Write (save) an Object to a Stream

OK

```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<BankAccount> accounts;
}
```



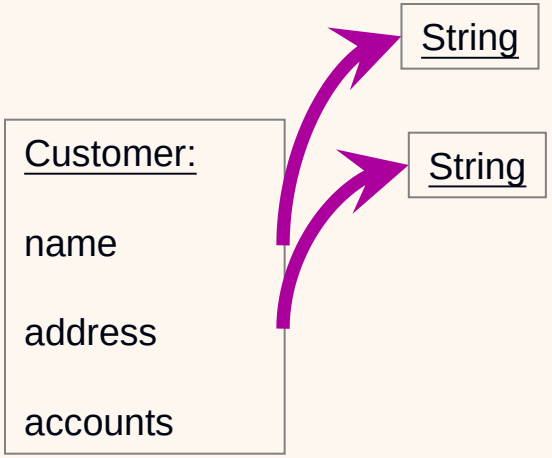
Goal 1:

Write (save) an Object to a Stream


```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<BankAccount> accounts;
}
```

OK

OK

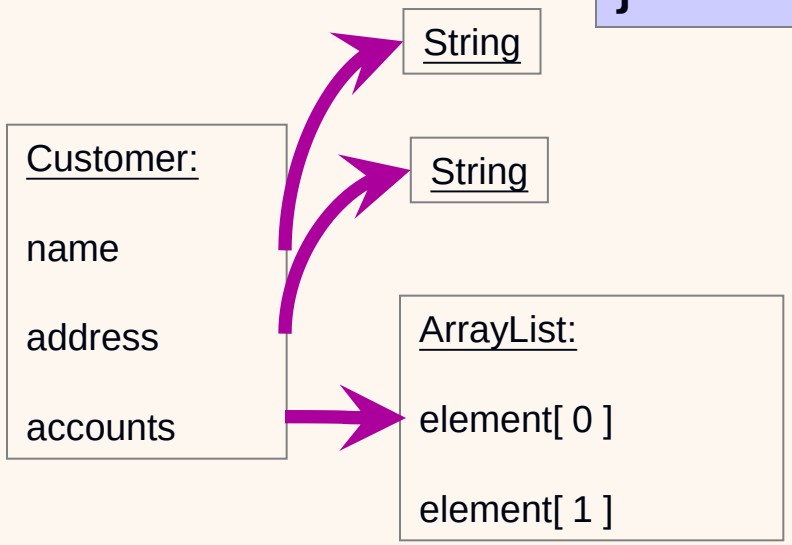


Goal 1:

Write (save) an Object to a Stream

Recursive saving required

```
public class Customer implements Serializable {
    private String name;        // OK
    private String address;     // OK
    private List<BankAccount> accounts;  // ???
}
```

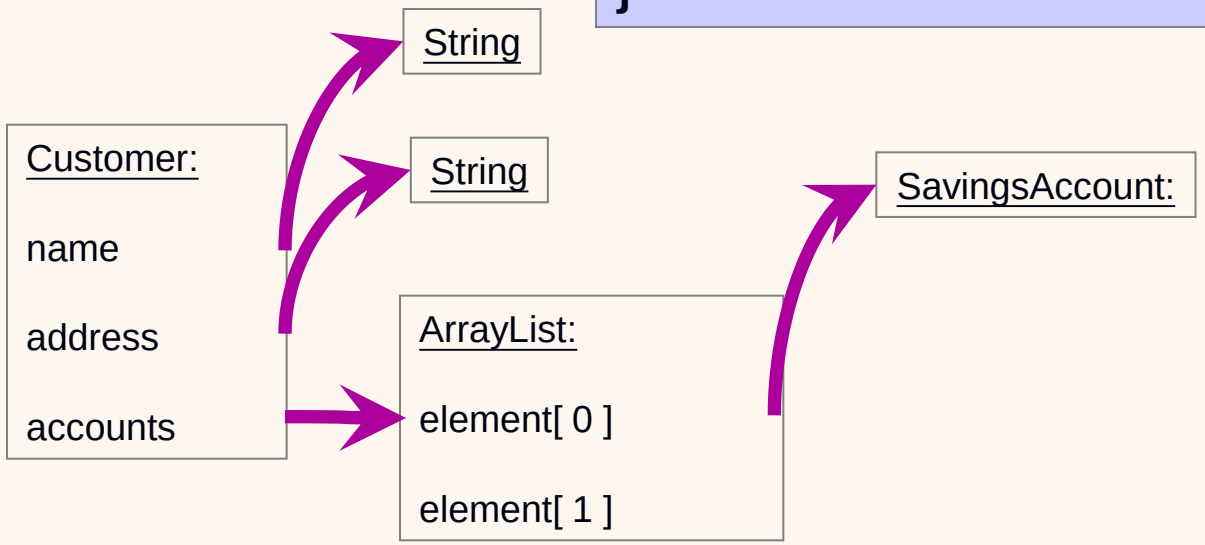


Goal 1:

Write (save) an Object to a Stream

Recursive saving required

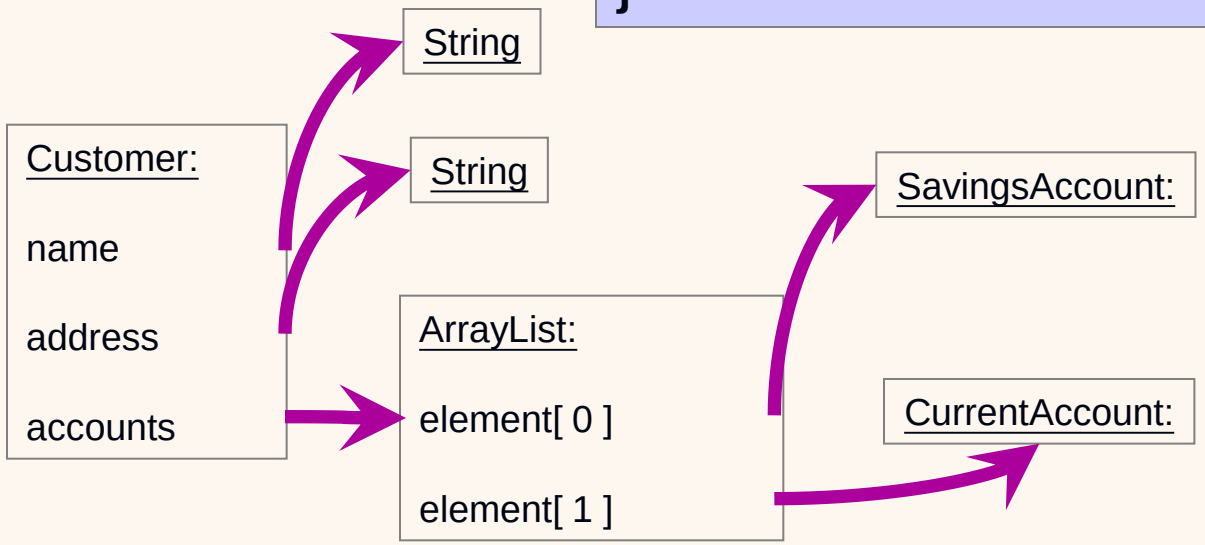
```
public class Customer implements Serializable {
    private String name;          // OK
    private String address;       // OK
    private List<BankAccount> accounts;  // ???
}
```



Goal 1:
Write (save) an Object to a Stream

Recursive saving required

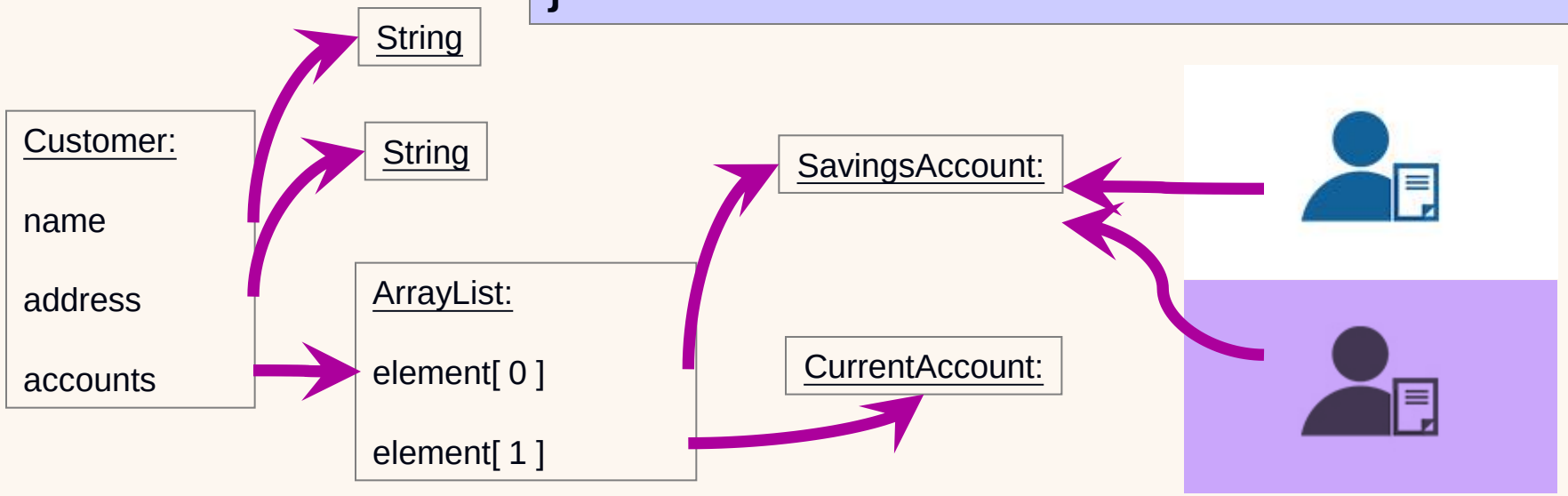
```
public class Customer implements Serializable {
    private String name;          // OK
    private String address;       // OK
    private List<BankAccount> accounts;  // ???
}
```



Goal 1:
Write (save) an Object to a Stream

Recursive saving required

```
public class Customer implements Serializable {
    private String name;          // OK
    private String address;       // OK
    private List<BankAccount> accounts;  // ???
}
```



Serialization Terms

- **Serialization** is the process of transforming an in-memory object to a byte stream

- **Deserialization** is the inverse process of reconstructing an object from a byte stream to the same state in which the object was previously serialized
  - "**Serializing out**" and "**serializing in**" are also used

Serialization Requirements

- For an object to be serializable, its class or some ancestor must implement the *empty* **Serializable** *marker* interface
  - Definition: An empty interface is called a *marker interface*

- (Recursively) All *non-transient** members of this object should be either primitives or **serializables** themselves

* see later

Serialization Streams

- The syntax for serialization is straightforward:
  - An object is *serialized* by writing it to an *ObjectOutputStream*
  - An object is *deserialized* by reading it from an *ObjectInputStream*

Serialization Streams

- The syntax for serialization is straightforward:
  - An object is *serialized* by writing it to an *ObjectOutputStream*
  - An object is *deserialized* by reading it from an *ObjectInputStream*

These are *decorating* classes

```
OutputStream out = new FileOutputStream( “save.ser” );
ObjectOutputStream oos = new ObjectOutputStream( out );
oos.writeObject( new Date() );
oos.close();
…
InputStream in = new FileInputStream( “save.ser” );
ObjectInputStream ois = new ObjectInputStream( in );
Date d = (Date) ois.readObject();
ois.close();
```

Serialization Streams

- The syntax for serialization is straightforward:
  - An object is *serialized* by writing it to an *ObjectOutputStream*
  - An object is *deserialized* by reading it from an *ObjectInputStream*

These are *decorating* classes

```
OutputStream out = new FileOutputStream( “save.ser” );
ObjectOutputStream oos = new ObjectOutputStream( out );
oos.writeObject( new Date() );
oos.close();
…
InputStream in = new FileInputStream( “save.ser” );
ObjectInputStream ois = new ObjectInputStream( in );
Date d = (Date) ois.readObject();
ois.close();
```

Casting is required

Object Graphs in Object Streams

- The entire *object graph* is serialized
  - The object graph consists of members of this class, or members of one of its members, etc.

Object Graphs in Object Streams

- The entire *object graph* is serialized
  - The object graph consists of members of this class, or members of one of its members, etc.
- Each location in memory holding an object is written "once"
  - Further attempts to write the same location will return reference to the object in the stream

What about Loops?

```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<Customer> friends;
}
```

A

What about Loops?

```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<Customer> friends;
}
```

A

```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<Customer> friends;
}
```

B

What about Loops?

```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<Customer> friends;
}
```

A

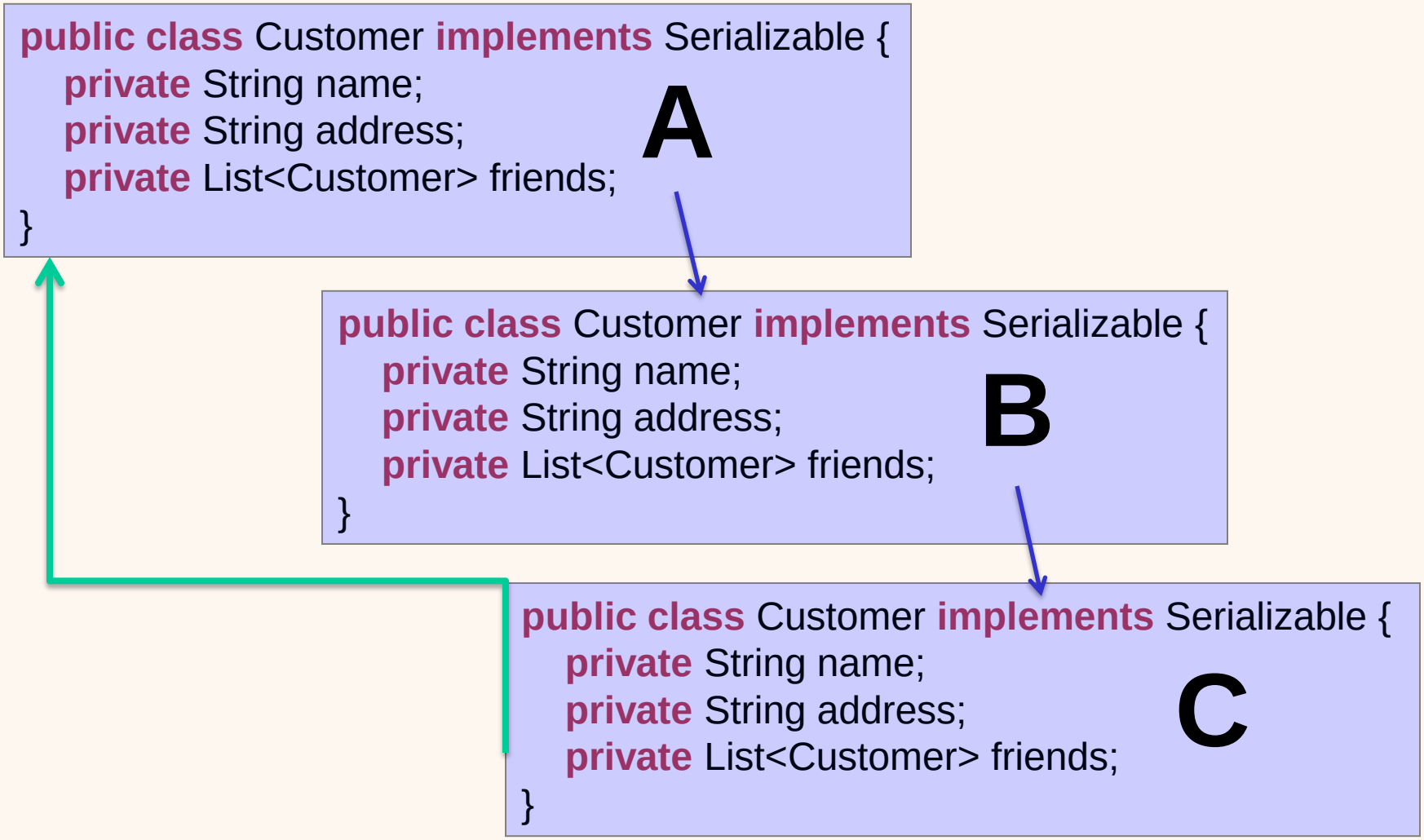
```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<Customer> friends;
}
```

B

```
public class Customer implements Serializable {
    private String name;
    private String address;
    private List<Customer> friends;
}
```

C

What about Loops?



What will this Program Print?

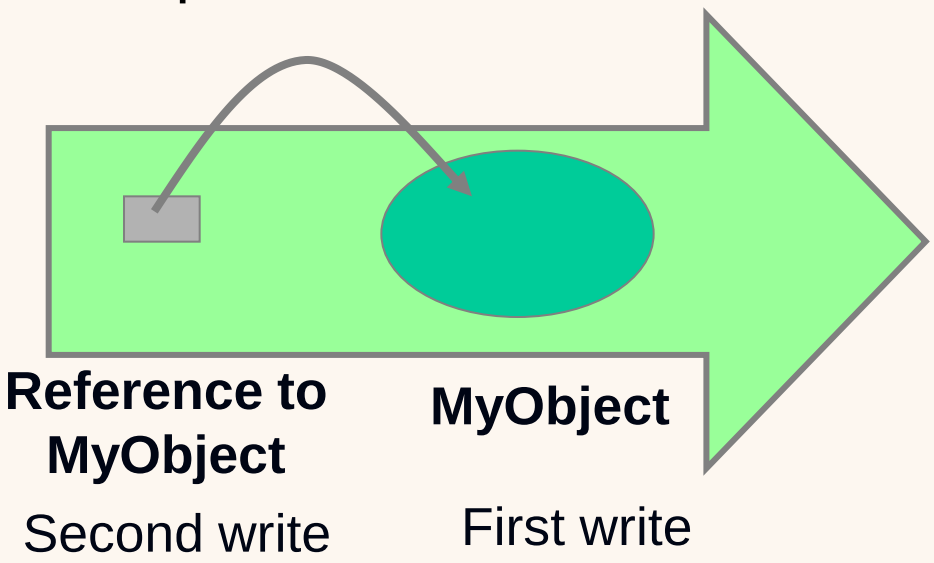
```
MyObject obj = new MyObject();          // must be Serializable

ObjectOutputStream out = new ObjectOutputStream(...);
obj.setState(100);                      // state – a data member of MyObject
out.writeObject(obj);                   // saves object with state = 100
obj.setState(200);
out.writeObject(obj);                   // saves object with state = ?

ObjectInputStream in = new ObjectInputStream(...);
obj = (MyObject)in.readObject();
System.out.println(obj);                // prints the state of the obj
obj = (MyObject)in.readObject();
System.out.println(obj);
```

Answer:

- The program will print:

100

100



- First time: Save object
- Second time: Save reference

transient and *static* Fields

- A field marked as **transient** is not serialized

```
public class MyObject implements Serializable {
        transient String str;
```

- During deserialization, **transient** fields are restored to their default values
  - (e.g., **transient** numeric fields are restored to zero, objects are restored to **null**)
- We can use it for non-serializable data members such as Streams

- **static** fields are also not impacted by serialization

Serialization and Primitive Types

- Primitive types cannot be serialized or deserialized
  - out.writeObject(5); // Illegal

Serialization and Primitive Types

- Primitive types cannot be serialized or deserialized
  - out.writeObject(5); // Illegal

  However, the *ObjectOutputStream* class implements the **DataOutput** interface
  - out.**writeInt**(5) // Legal
  - Similarly, ObjectInputStream implements **DataInput** for reading primitive types

Serialization and Primitive Types

- Primitive types cannot be serialized or deserialized
  - out.writeObject(5); // Illegal

  However, the *ObjectOutputStream* class implements the **DataOutput** interface
  - out.**writeInt**(5) // Legal
  - Similarly, ObjectInputStream implements **DataInput** for reading primitive types

- Note: We are not talking about primitive **data members**. They are serialized along with their containing object

Modifying a Class (1)

- After saving an object to a file, we might wish/need to modify the class description
- Such changes make the modified file **different** from the original file
  - Different members
  - Different class hierarchy
  - …
- This might make deserialization impossible
  - How do you put a saved **String** member into a new **int** field?

Modifying a Class (2)

- However, not all changes are such where we wish to treat the new class as a different class
  - Removing a member can be resolved by ignoring it

- Solution: A class's **version** is stored in a static attribute named *SerialVersionUID*
  - We can change it when we modify the source code and the changes are **crucial**

serialVersionUID (1)

- *serialVersionUID* changed → two different classes
  - Cannot deserialize object into changed class
  - *InvalidClassException* is thrown upon deserialization of an object with a different *serialVersionUID*

- *serialVersionUID* unchanged → changed class should be treated as original class
  - In order to deserialize an object, there should exist the corresponding *.class* file

serialVersionUID (2)

- This field is not mandatory
  - If not specified, java compiler computes it based on attributes and signatures of methods defined by the class
  - Editing and recompiling a class between serialization and deserialization will modify the class's *SerialVersionUID*
- If you explicitly set *SerialVersionUID*, deserialization will work with a modified .class file in many cases
  - Compatible changes: add/remove methods or data members
  - Incompatible changes: change class hierarchy

Advice
serialVersionUID

- Always declare this field when writing Serialzable classes

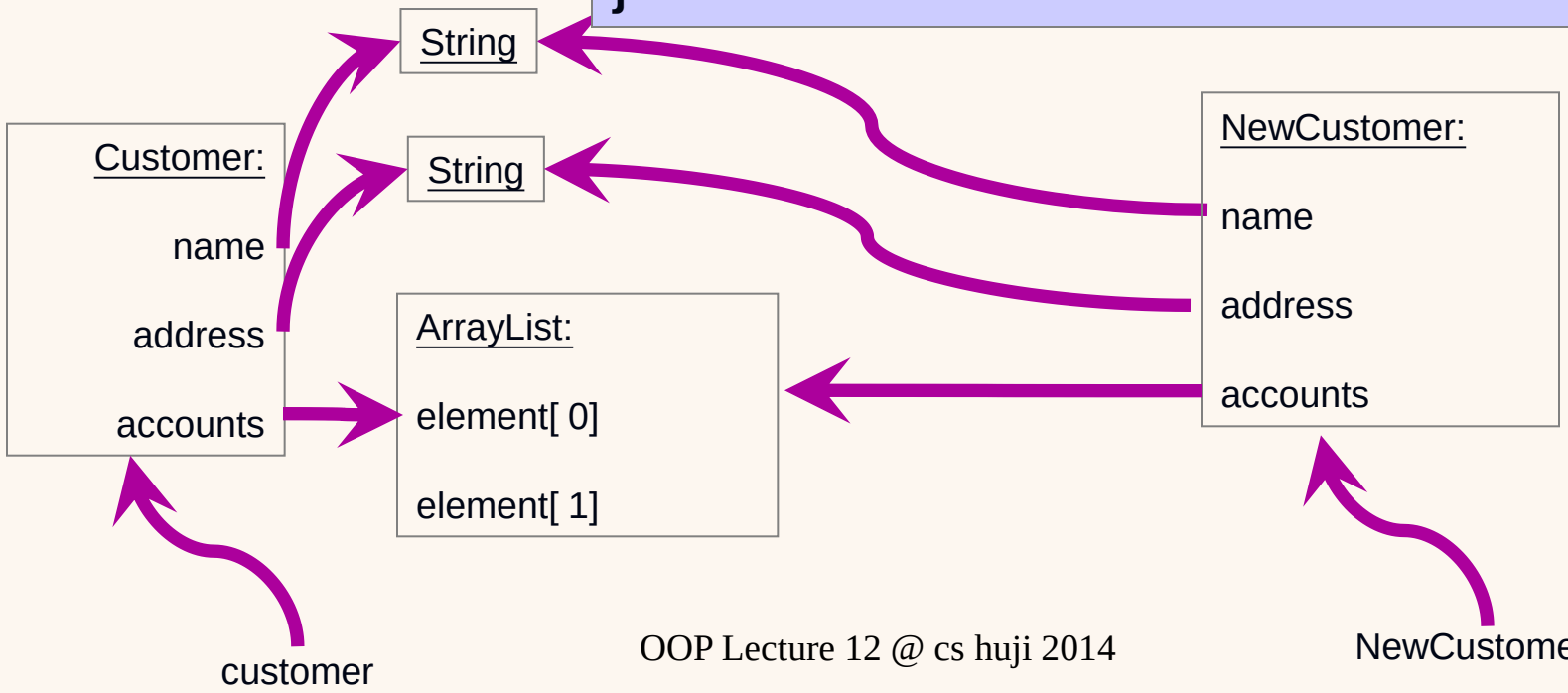
- Declare it **private** in order to avoid access by subclasses (this value is not useful for them)

Goal 2:

Make an Exact Copy of an Object

References copied, no recursion

```
public class Customer implements Clonable {
    private String name;          // OK
    private String address;       // OK
    private List<BankAccount> accounts;  // ???
}
```



Shallow vs. Deep Copy

- Shallow Copy
  - Results of using the *Object.clone()* method
  - If the class has non-primitive data type members, their *references (*and ***not*** the objects) are copied

  → *members in both the **original object** and **the cloned object** refer to **the same** object*

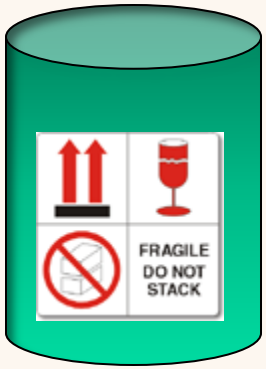
- Deep Copy
  - Non-primitive members of the class are cloned as well

  → *members in the **original object** and the **cloned object** refer to **different** objects*

Cloning Requirements

- For an object to be clonable, its class or some ancestor must
  - Implement the empty *Clonable* marker interface
  - Override Object's protected *clone*() method

- By default, cloning creates a shallow copy (i.e., references are copied, not values)
  - Thus, overriding the *clone*() method is crucial



Cloning Example

```
class MyPet implements Cloneable {
  private Date birthDate;
  public Object clone() throws CloneNotSupportedException {
    MyPet pet = (MyPet) super.clone();            // First – creating a shallow copy.
    pet.birthDate = (Date)birthDate.clone();      // Cloning date for deep copy.
    return pet;
  }
  ….
}
public class PetCreator {
  public static void main(String[] args) {
    try {
      MyPet myPet = new MyPet();
      myPet.setType(“Dog”);
      MyPet myPet1 = (MyPet) myPet.clone();
      MyPet myPet2 = (MyPet) myPet.clone();
      myPet1.setName(“Woofi");
      myPet2.setName(“Goofi");
       …..
    } catch (CloneNotSupportedException e) {   // Checked Exception
      e.printStackTrace();
    }
  }
}
```

Caveat

- Using the Clonable interface is not a recommended method cloning object
  - API is messy (implementing Clonable is not enough)
- A better alternative to cloning is using a copy constructor
  - Simpler
  - Allows you to clone an object from a different type (say, cloning an ArrayList into a LinkedList)

Copy Ctor. Example

```
class MyPet  {
  private Date birthDate;
  public MyPet(MyPet other) {
    this();                                   // First – calling default ctor.
    this.birthDate = other.birthDate.clone();  // Date class doesn't have a copy constructor
                                              // Use cloning instead
  }
  ….
}
public class PetCreator {
  public static void main(String[] args) {
    try {
      MyPet myPet = new MyPet();
      myPet.setType("Dog");
      MyPet myPet1 = (MyPet) myPet.clone();
      MyPet myPet2 = (MyPet) myPet.clone();
      myPet1.setName("Woofi");
      myPet2.setName("Goofi");
       …..
    } catch (CloneNotSupportedException e) {   // Check Exception
      e.printStackTrace(System.err);
    }
  }
}
```

So Far…

- Serialization
  - Used to save objects to disk or transfer them over a network
  - Serialization of an object recursively stores all included objects
  - Each object stored only once
- Cloning
  - Used to copy an object's data in memory

So Far 2…

- Both Serialization & Cloning:
  - Utilize marker interfaces
  - Can be customized

- Further Reading
  - http://java.sun.com/developer/technicalArticles/Programming/serialization/

Java Reflections

- Allows an execution of a Java program to examine or "*introspect*" upon itself, and manipulate internal properties of the program
  - For example, it's possible for a Java class to obtain the names of all its members and display them
- Very powerful technique
- Should be handled with care

The *Class* class

- Every java object is either a reference or primitive type
  - Reference types: all inherit from *java.lang.Object*: *Classes*, *arrays*, and *interfaces* are all reference types
  - Primitive types: Include a fixed set: **boolean**, **byte**, **char**, **double**, **float**, **int**, **long**, and **short**
- For every **reference type**, JVM instantiates an immutable instance of *java.lang.Class*

Reflection (1)

- Class *cls* = Class.forName("ClassName");
  - *cls* is now the *Class* object of *ClassName*
  - ⇔ Class *cls* = *ClassName*.getClass();
  - Throws a *ClassNotFoundException* if *ClassName* is not found

- Method methlist[] = *cls*.getDeclaredMethods();
  - Returns a list of the class's methods (with any modifier, **including private**)
  - *Method* class has methods that answer various questions about the method
    - *getDeclaringClass*()
    - *getParameterTypes*() – a list of the method's parameter types

Reflection (2)

- Constructor ctorlist[] = *cls.getDeclaredConstructors*()
  - A list of the class's constructors
  - Similar to *Method* class

- Field fieldlist[] = *cls.*getDeclaredFields();
  - Class's members
  - set(Object *obj*, Object *data*) / get(Object *obj*) – set / get the value of *obj*'s field
  - Can answer questions about the member's type, modifier's, etc.
  - By default, does not allow access to **private** members
    - To gain such access: field.setAccessible(**true**);
    - **Be careful!!!**

Creating new Objects

- Object retobj = ctorlist[i].newInstance(*arglist*);
  - Creates a new object using the constructor
  - *arglist* should be created according to the constructor's getParameterTypes() method

- Invoking a method: methlist[j].invoke(*obj*, *arglist*)
  - Where *obj* is either an object created via *newInstance*() *method* of the *Constructor* class, **or** an object created in the "old fashioned way" (i.e. *obj* = **new** ClassName(…))
  - *arglist* – same as in *Cosntructor's newInstance*()
  - Always returns an object of type Object (**null** if the method returns **void**)

Java Reflections Example

```
import java.lang.reflect.*;


public class DumpMethods {
        public static void main(String args[]) throws ClassNotFoundException,
            IllegalArgumentException, InstantiationException, IllegalAccessException,
            InvocationTargetException{
              Class cls= Class.forName(args[0]);
              Field fields[] = cls.getDeclaredFields();
              Constructor[] ctors = cls.getDeclaredConstructors();
              Object obj = ctors[0].newInstance();
              for (Field field:fields)
                    if (field.getModifiers() == Modifier.PUBLIC) {
                        System.out.println(field.getName()+": "+field.get(obj));
                    }
        }
}
```

Java Reflections Example

```
import java.lang.reflect.*;

public class DumpMethods {
    public static void main(String args[]) throws ClassNotFoundException,
        IllegalArgumentException, InstantiationException, IllegalAccessException,
        InvocationTargetException{
        Class cls= Class.forName(args[0]);
        Field fields[] = cls.getDeclaredFields();
        Constructor[] ctors = cls.getDeclaredConstructors();
        Object obj = ctors[0].newInstance();
        for (Field field:fields)
            if (field.getModifiers() == Modifier.PUBLIC) {
                System.out.println(field.getName()+": "+field.get(obj));
            }
    }
}
```

Create a class by its name

Get a list of its members

Assuming the class has only one constructor (the default one)

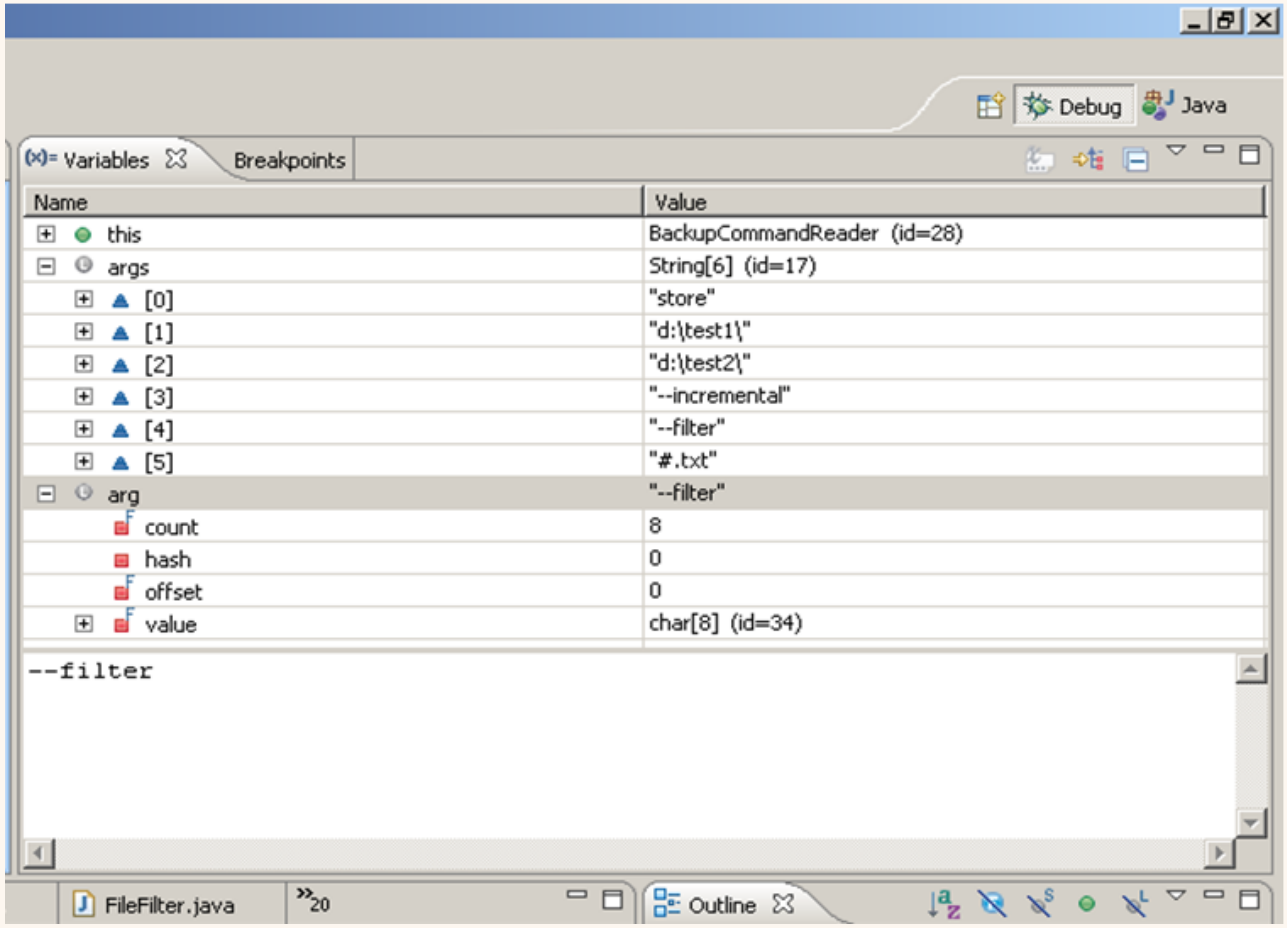
Look for public members

Get name and value of member in obj

Java Reflections – Why?

- Flexibility & Extensibility
  - Allows the adding of new classes which the original class doesn't know about in compile time
  - Avoids ugly switch block
- Class Browsers, Visual Development Environments and Debugging tools
  - These tools need to be able to enumerate the members of classes (in order to browse the class, auto-complete, view the class's state or run methods in a specific context)

Sounds Familiar?



Java Reflections – Drawbacks

- Exposure of Internals
  - Using java reflection comes in contrast to the encapsulation and information hiding principles
  - Can result in unexpected side-effects, which may render code **dysfunctional** and may **destroy portability**

- Performance Overhead
  - Reflective operations have slower performance than their non-reflective counterparts

Reminder: Private is not Secret!

- A common misconception is that **private** means secret
  - Sensitive information (e.g., passwords) should not be stored in **private** members
  - If you want to protected your data, **encrypt** it
  - More to come next year
- The **private** modifier is used for **better design**
  - Using java reflection to bypass the private restriction is cheating **nobody but yourself**

So Far…

- Java Reflection is a very powerful tool
  - Allows us to examine the internal structure of any class
  - Can extend flexibility and extensibility

- However, we must be very careful when using reflections
  - Ruins some of the basic OOP principles (encapsulation, information hiding)
  - Has performance overhead