Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 12:

Advanced Topics

Last Week

- String Processing
- Regular Expressions

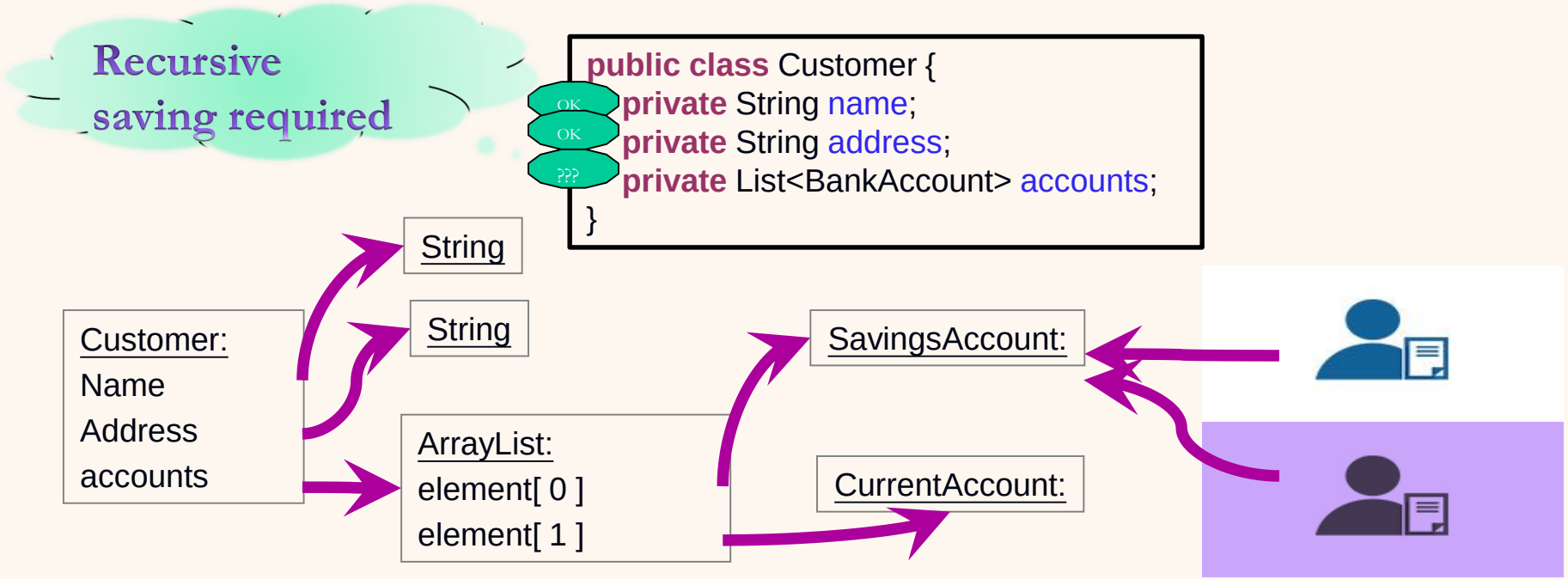
Lecture 12a: Overview

- What is Serialization?
- Intricacies of 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 an Object to a Stream



Serialization Terms

- **Serialization** is the process of transforming an in-memory object to a stream
- This process recursively saves all fields in the given object to memory
  - Fields, fields of fields, fields of fields of fields …
- **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

Java 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** data members of this object should be either primitive or **Serializable** themselves

* see later

Java 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*

```
try (OutputStream out = new FileOutputStream( "save.ser" );
        ObjectOutputStream oos = new ObjectOutputStream(out);) {
                oos.writeObject( new Date() );
} catch (IOException e) {… }
…
try ( InputStream in = new FileInputStream( "save.ser" );
        ObjectInputStream ois = new ObjectInputStream( in );) {
                Date d = (Date) ois.readObject();
} catch (IOException e) {… }
```

These are *decorating* classes

Down-casting required

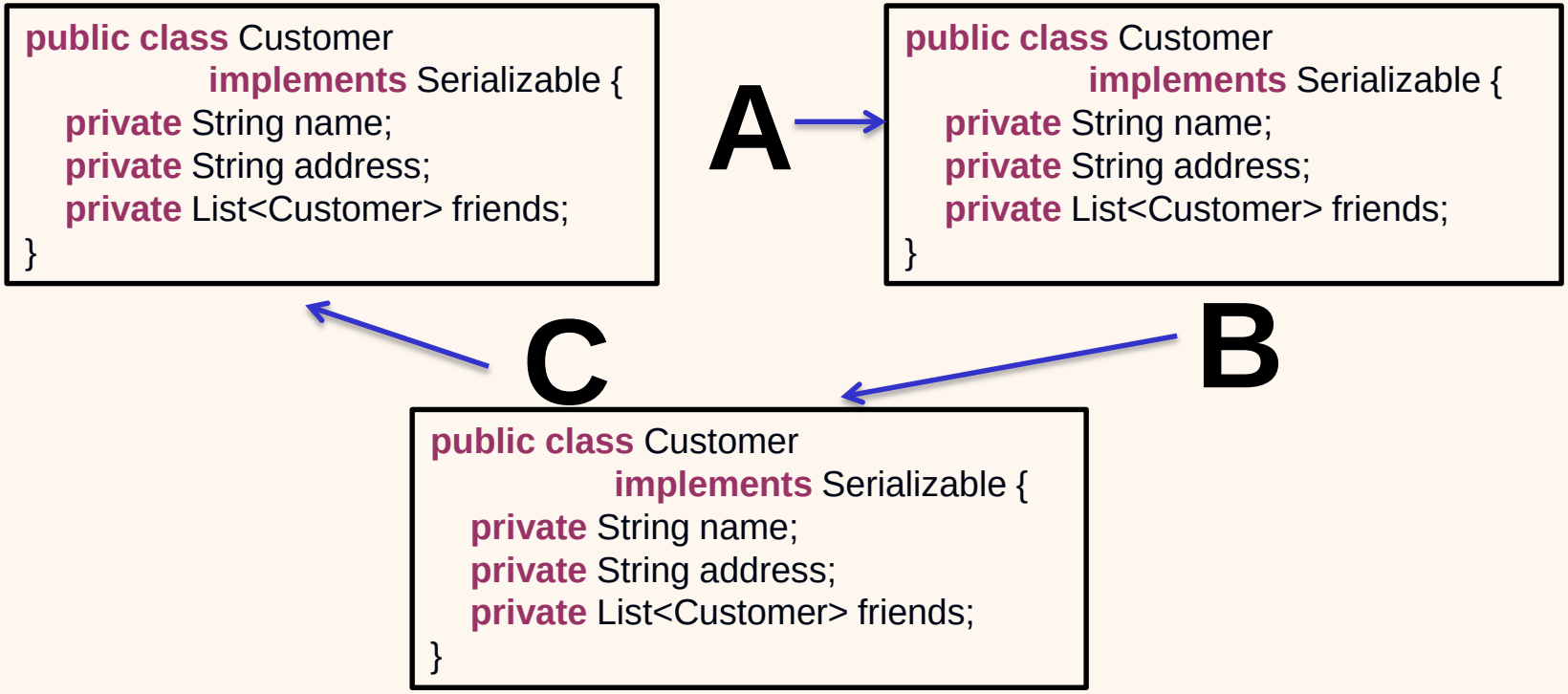
Lecture 12b: Overview

- What is Serialization?
- Intricacies of Serialization
- Cloning
- Java Reflections

Object Graphs in Object Streams

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

What about Cycles?



What will this Program Print?

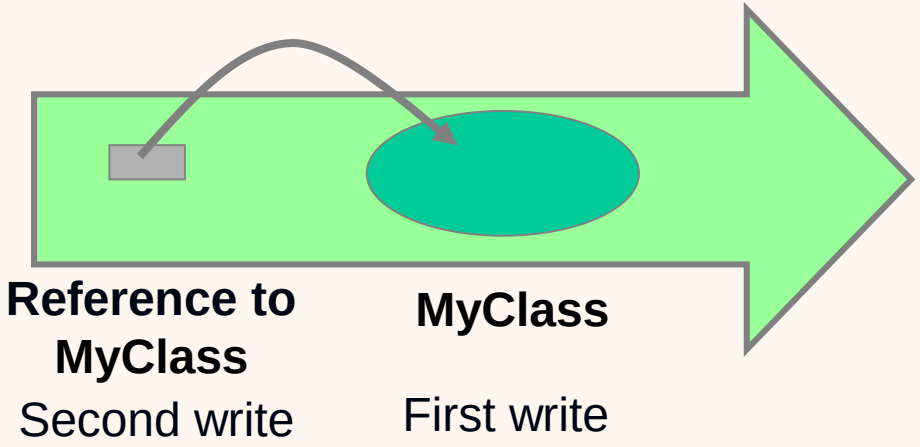
```
MyClass obj = new MyClass();                    // must be Serializable

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

ObjectInputStream in = new ObjectInputStream(...);
obj = (MyClass)in.readObject();
System.out.println(obj);                        // prints the state of the obj
obj = (MyClass)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 by the **transient** keyword is not serialized

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

- During deserialization, **transient** fields are restored to their default values
  - E.g., **transient** numeric fields are restored to zero, references to **null**
- We can use it for non-serializable data members such as Streams
- **static** fields are also not included in the process of serialization

Serialization and Primitive Types

- Primitive types cannot be serialized or deserialized
  - out.writeObject(5); // Illegal
- However, the *ObjectOutputStream* class implements the **DataOutput** interface
  - Similarly, ObjectInputStream implements **DataInput** for reading primitive types
  - out.**writeInt**(5) // Legal
- 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 edit the class
- Some changes make the modified class inherently **different** from the original class
  - Different data members
  - Different class hierarchy
  - …
- This might make deserialization impossible
  - How do you put a saved **String** data 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 data 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* is the only static field saved during serialization

serialVersionUID (1)

- *serialVersionUID* changed → two different classes
  - Cannot deserialize object into the changed class
  - *InvalidClassException* is thrown upon deserialization of an object with a different *serialVersionUID*
- *serialVersionUID* unchanged → changed class should be treated as original class

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
  - In this case, editing and recompiling a class between serialization and deserialization modifies 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

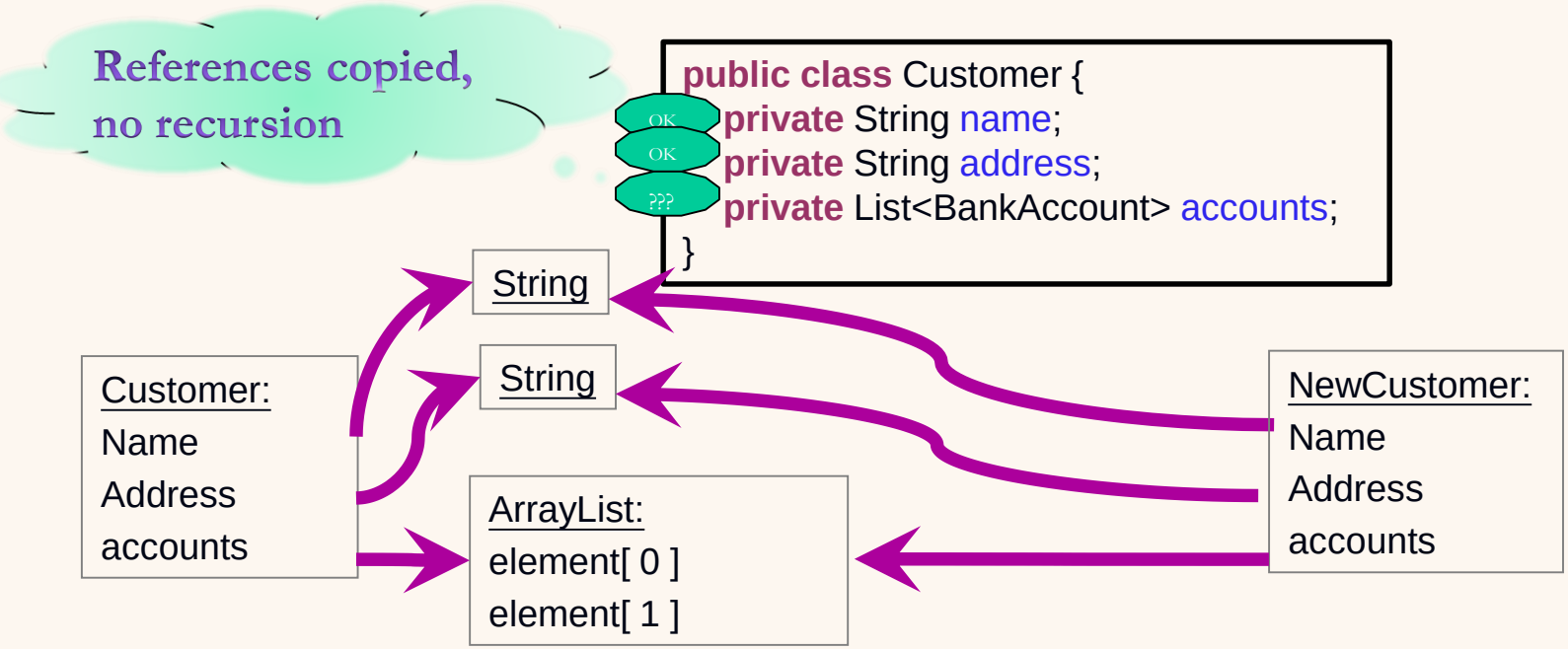
serialVersionUID

- Always declare this field when writing Serializable classes
- Declare it **private** in order to avoid access by subclasses (this value is not useful for them)

Lecture 12c: Overview

- What is Serialization?
- Intricacies of Serialization
- Cloning
- Java Reflections

Goal 2: Make an Exact Copy of an Object



Shallow vs. Deep Copy

- Shallow Copy
  - If the class has non-primitive data members, their *references (*and ***not*** the objects) are copied

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

- Deep Copy
  - Non-primitive data members are recursively cloned as well

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

Java Cloning Requirements

- For an object to be cloneable, its class or some ancestor must
  - Implement the empty *Cloneable* 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

Object.clone()

- The Object.clone() method does two things:
- Check that the calling class **implements** Cloneable
  - The Object class itself **does not** implement Cloneable
  - If the calling class doesn't implement Cloneable, an exception is thrown (CloneNotSupportedException)
- Perform **shallow** copy
- Arrays also implement Cloneable
  - And also perform a shallow copy

Cloning Example

```
class Pet implements Cloneable {
  private Date birthDate;
  public Object clone()
                throws CloneNotSupportedException {
    // First – creating a shallow copy.
    Pet pet = (Pet) super.clone();
    // Cloning date for deep copy.
    pet.birthDate = (Date)birthDate.clone();

    return pet;
  }
  ….
}
```

```
try {
      Pet myPet = new Pet();
      myPet.setType(“Dog”);
      Pet myPet1 = (Pet) myPet.clone();
      Pet myPet2 = (Pet) myPet.clone();
      myPet1.setName(“Woofi");
      myPet2.setName(“Goofi");
       …..
} catch (CloneNotSupportedException e) {
      e.printStackTrace(); // Checked Exception
}
```

Caveat

- Using the Cloneable interface is not a recommended method for cloning objects
  - API is messy (implementing Cloneable 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 Pet implements Cloneable {
  private Date birthDate;
  public Pet(Pet other) {
    this();     // First – calling default ctor.

    // Date class doesn't have a copy constructor
    // Use cloning instead
    this.birthDate = other.birthDate.clone();
  }
  ….
}
```

```
Pet myPet = new Pet();
myPet.setType("Dog");
Pet myPet1 =  new Pet(myPet);
Pet myPet2 =  new Pet(myPet);
myPet1.setName("Woofi");
myPet2.setName("Goofi");
```

Lecture 12d: Overview

- What is Serialization?
- Intricacies of Serialization
- Cloning
- Java Reflections

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*

The *Class* class

- Class *cls* = Class.forName(“MyClass”);
  - *cls* is now the *Class* object of *MyClass*
  - ⇔ Class *cls* = myObj.getClass();
    - Where myObj is an instance of type MyClass
  - Throws a *ClassNotFoundException* if *MyClass* is not found

Creating a New Object

- Constructor[] ctorlist = *cls.getDeclaredConstructors*()
  - A list of the class's constructors
- Object retobj = ctorlist[i].newInstance(*arglist*);
  - Creates a new object using the constructor
  - *arglist* should be created according to the constructor's getParameterTypes() method

Methods

- Method[] methlist = *cls*.getDeclaredMethods();
  - Returns a list of the class's methods (**including private** methods)
  - This class has methods that provide information about the method
    - *getDeclaringClass*() – which class declared this method
    - *getParameterTypes*() – a list of the method's parameter types
- Invoking a method: methlist[j].invoke(*obj*, *arglist*)
  - *obj* is an instance of the class
  - *arglist* – same as in *Cosntructor's newInstance*()
  - Returns the method's return value up-cast to Object, or **null** if the method returns **void**

Fields

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

Java Reflections Example

```
import java.lang.reflect.*;
public class DumpMembers {
    public static void main(String args[]) throws ClassNotFoundException, InstantiationException,
      IllegalArgumentException, IllegalAccessException, InvocationTargetException {
            Class cls= Class.forName(args[0]);              // args[0]  is a class name
            Field[] fields = cls.getDeclaredFields();       // Get class fields
            Constructor[] ctors = cls.getDeclaredConstructors();      // Get class constructors
            Object obj = ctors[0].newInstance();            // Create new instance of the input class
            for (Field field:fields)                        // Traverse object fields
                if (Modifier.isPublic(field.getModifiers())) // Print public ones
                    System.out.println(field.getName()+": "+field.get(obj));
    }
}
```

Java Reflections – Why?

- Flexibility & Extensibility
  - Allows the addition of new classes, which the original class did not know about when it was compiled
  - Avoids ugly switch block
- Class Browsers, Visual Development Environments and Debugging tools
  - These tools can make use of reflection 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?

- In **Serialization**, some class (ObjectOutputStream) gains access to the **private** data of an object of another class (the saved object)
- This process is done using reflection

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…

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

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
  - Contradicts some of the basic OOP principles (encapsulation, information hiding)
  - Has performance overhead

Next Week

- Summary