Advanced Programming Methods

Seminar 12

Overview

1. instanceof operator
2. Java Serialization
3. Discuss how we can serialize our ToyLanguage interpreter. Please discuss the implementation of different interesting scenarios (one ProgramState is serialized, entire Repository is serialized, etc.)

Note: Notes are based on some online (including Oracle) tutorials.

Instanceof operator

- compares an object to a specified type.
- to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.
- null is not an instance of anything.

Instanceof operator

```
class Parent {}

class Child extends Parent implements MyInterface {}

interface MyInterface {}


Parent obj1 = new Parent();

Parent obj2 = new Child();
```

```
obj1 instanceof Parent: true

obj1 instanceof Child: false

obj1 instanceof MyInterface: false

obj2 instanceof Parent: true

obj2 instanceof Child: true

obj2 instanceof MyInterface: true
```

Real use of Instanceof operator

```
interface Printable{}
class A implements Printable{
public void a(){System.out.println("a method");}
}
class B implements Printable{
public void b(){System.out.println("b method");}
}
class Call{
void invoke(Printable p){
if(p instanceof A){
A a=(A)p;//Downcasting
a.a();
}
if(p instanceof B){
B b=(B)p;//Downcasting
b.b();
} } }
```

Java Serialization

Serializable in Java

- If you want a class object to be serializable, all you need to do it implement the java.io.Serializable interface.

- Serializable in java is a marker interface and has no fields or methods to implement.

- Serialization in java is implemented by ObjectInputStream and ObjectOutputStream, so all we need is a wrapper over them to either save it to file or send it over the network.

An Example

```
package seminar12.serialization;
import java.io.Serializable;
public class Employee implements Serializable {
private String name;
private int id;
transient private int salary;// to be not serialized to stream
@Override
public String toString(){ return "Employee{name="+name+",id="+id+",salary="+salary+"}";}
//getter and setter methods
public String getName() {return name;}
public void setName(String name) {this.name = name;}
public int getId() {return id; }
public void setId(int id) { this.id = id;}
public int getSalary() { return salary; }
public void setSalary(int salary) {this.salary = salary;}
}
```

General utility methods for serialization

```
package seminar12.serialization;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;


public class SerializationUtil {

// deserialize to Object from given file

public static Object deserialize(String fileName) throws IOException, ClassNotFoundException {

FileInputStream fis = new FileInputStream(fileName);

ObjectInputStream ois = new ObjectInputStream(fis);

Object obj = ois.readObject();

ois.close();

return obj;

}
```

General utility methods for serialization

```
// serialize the given object and save it to file

public static void serialize(Object obj, String fileName) throws IOException {

FileOutputStream fos = new FileOutputStream(fileName);

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(obj);


fos.close();

}


}
```

An example

```
/package com.journaldev.serialization;

import java.io.IOException;

public class SerializationTest {

public static void main(String[] args) {

String fileName="employee.ser";

Employee emp = new Employee();

emp.setId(100);

emp.setName("ABC");

emp.setSalary(5000);

//serialize to file

try {

SerializationUtil.serialize(emp, fileName);

} catch (IOException e) {

e.printStackTrace();

return;

}
```

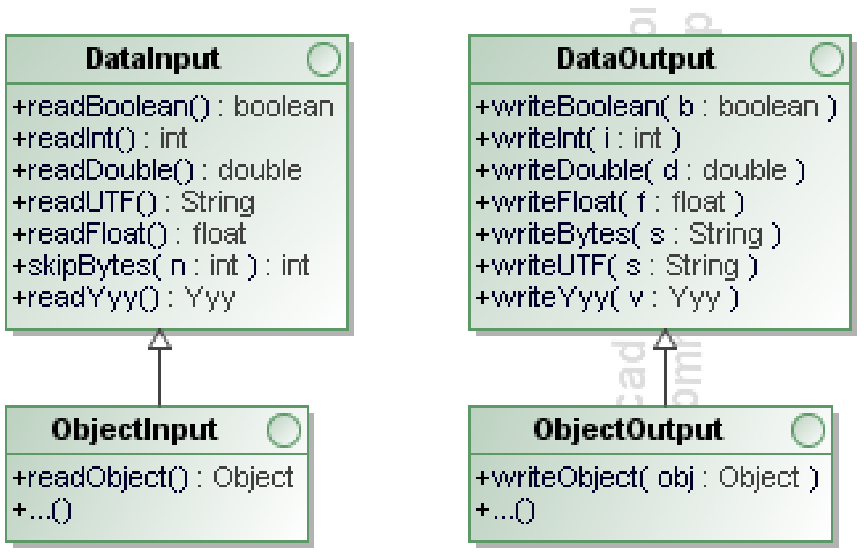
An example

```
Employee empNew = null;
try {
empNew = (Employee) SerializationUtil.deserialize(fileName);
} catch (ClassNotFoundException | IOException e) {
e.printStackTrace();
}
System.out.println("emp Object::"+emp);
System.out.println("empNew Object::"+empNew);
}}
//OUTPUT
emp Object::Employee{name=ABC,id=100,salary=5000}
empNew Object::Employee{name=ABC,id=100,salary=0}
```

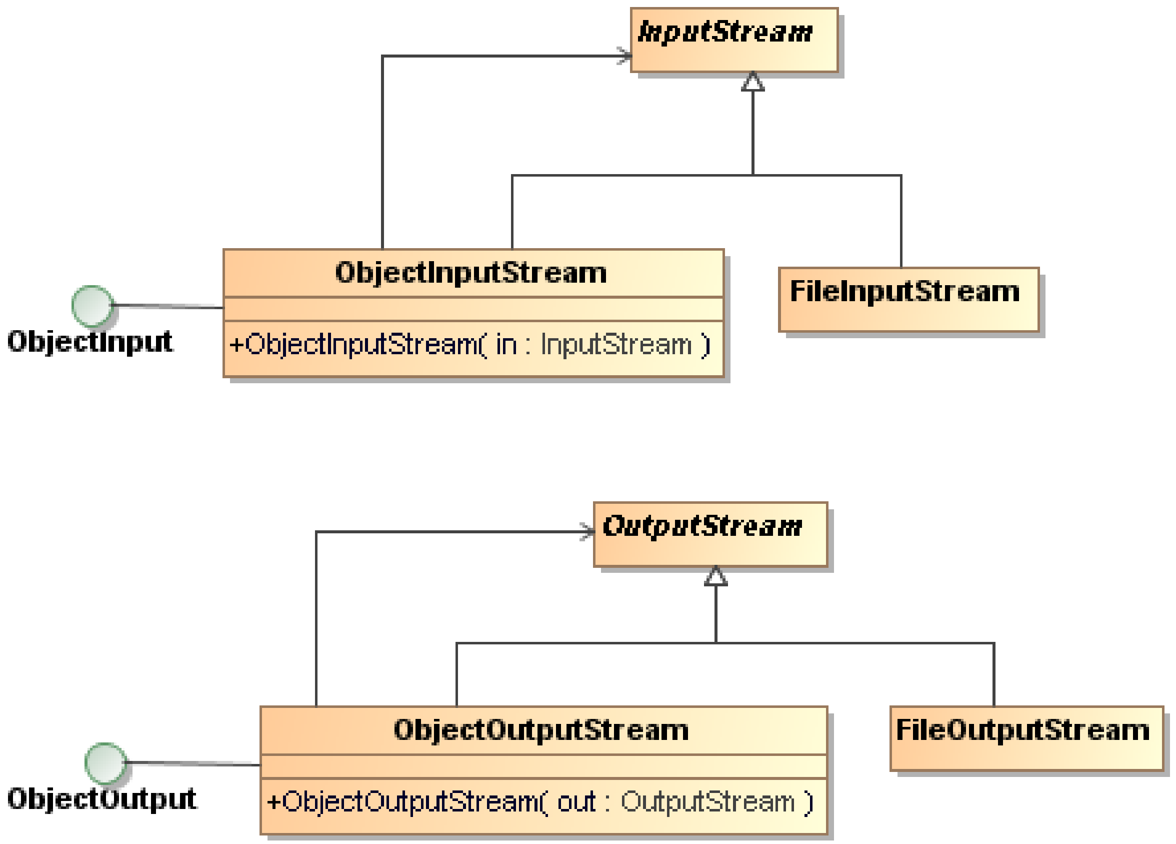
- salary is a transient variable and it’s value was not saved to file
- Similarly static variable values are also not serialized since they belongs to class and not object.

Java Objects Serialization

- The process of writing/reading objects from/to a file/external support.
- An object is persistent (serializable) if it can be written into a file/external support and can be read from a file/external support



Objects Serialization



Class Refactoring with Serialization and serialVersionUID

Serialization in java permits some changes in the java class if they can be ignored.Some of the changes in class that will not affect the deserialization process are:

- Adding new variables to the class
- Changing the variables from transient to non-transient, for serialization it's like having a new field.
- Changing the variable from static to non-static, for serialization it's like having a new field.

For all these changes to work, the java class should have **serialVersionUID** defined for the class.

Class Refactoring with Serialization and serialVersionUID

- If we change the Employee class as follows:

```
public class Employee implements Serializable {
…
private String password;
…
public String getPassword() { return password;}
public void setPassword(String password) { this.password = password;}
}
```

Class Refactoring with Serialization and serialVersionUID

- And run the following:

```
public class DeserializationTest {
public static void main(String[] args) {
String fileName="employee.ser";
Employee empNew = null;
try {
empNew = (Employee) SerializationUtil.deserialize(fileName);
} catch (ClassNotFoundException | IOException e) { e.printStackTrace();}
System.out.println("empNew Object::"+empNew);
}}
```

- We got an error!! The reason is that serialVersionUID of the previous class and new class are different.
- if the class doesn’t define serialVersionUID, it’s getting calculated automatically and assigned to the class. Java uses class variables, methods, class name, package etc to generate this unique long number.

Class Refactoring with Serialization and serialVersionUID

- In order to avoid the error we have to add the following field to the original Employee class

```
private static final long serialVersionUID = -6470090944414208496L;
```

- Now we will serialize it and then will add the new filed password to Employee class and will deserialize it again we will not get any error.
    - the object stream is deserialized successfully because the change in Employee class is compatible with serialization process.

Java Externalizable Interface

- the java serialization process is done automatically.
- Sometimes we want to obscure the object data to maintain it’s integrity.
- We can do this by implementing java.io.Externalizable interface and provide implementation of writeExternal() and readExternal() methods to be used in serialization process.

Java Externalizable Interface

```
 package seminar12.externalization;

import java.io.Externalizable;

import java.io.IOException;

import java.io.ObjectInput;

import java.io.ObjectOutput;


public class Person implements Externalizable{

private int id;

private String name;

private String gender;

@Override

public void writeExternal(ObjectOutput out) throws IOException {

out.writeInt(id);

out.writeObject(name+"xyz");

out.writeObject("abc"+gender);

}

@Override

public String toString(){ return "Person{id="+id+",name="+name+",gender="+gender+"}";}
```

Java Externalizable Interface

```
@Override

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

id=in.readInt();

//read in the same order as written

name=(String) in.readObject();

if(!name.endsWith("xyz")) throw new IOException("corrupted data");

name=name.substring(0, name.length()-3);

gender=(String) in.readObject();

if(!gender.startsWith("abc")) throw new IOException("corrupted data");

gender=gender.substring(3);

}

public int getId() { return id;}

public void setId(int id) {this.id = id;}

public String getName() {return name;}

public void setName(String name) {this.name = name;}

public String getGender() {return gender;}

public void setGender(String gender) {this.gender = gender;}

}
```

Java Externalizable Interface

```
package  seminar12.externalization;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;


public class ExternalizationTest {

public static void main(String[] args) {

String fileName = "person.ser";

Person person = new Person();

person.setId(1);

person.setName("ABC");

person.setGender("Male");
```

Java Externalizable Interface

```
try {
FileOutputStream fos = new FileOutputStream(fileName);
ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(person);
    oos.close();
} catch (IOException e) {e.printStackTrace();}


FileInputStream fis;
try {
fis = new FileInputStream(fileName);
ObjectInputStream ois = new ObjectInputStream(fis);
    Person p = (Person)ois.readObject();
    ois.close();
    System.out.println("Person Object Read="+p);
} catch (IOException | ClassNotFoundException e) { e.printStackTrace();}
 }}
```

Java Serialization Methods

- serialization in java is automatic and all we need is implementing Serializable interface
- there are four methods that we can provide in the class to change the serialization behavior:
    - readObject(ObjectInputStream ois): If this method is present in the class, ObjectInputStream readObject() method will use this method for reading the object from stream.
    - writeObject(ObjectOutputStream oos): If this method is present in the class, ObjectOutputStream writeObject() method will use this method for writing the object to stream. One of the common usage is to obscure the object variables to maintain data integrity.
    - Object writeReplace(): If this method is present, then after serialization process this method is called and the object returned is serialized to the stream.
    - Object readResolve(): If this method is present, then after deserialization process, this method is called to return the final object to the caller program.

Serialization with Inheritance

- Sometimes we need to extend a class that doesn’t implement Serializable interface.
- If we rely on the automatic serialization behavior and the superclass has some state, then they will not be converted to stream and hence not retrieved later on.
- This is one place, where readObject() and writeObject() methods really help. By providing their implementation, we can save the super class state to the stream and then retrieve it later on.
- See the following example:

Serialization with Inheritance

- Superclass does not implement Serializable

```
package seminar12.serialization.inheritance;


public class SuperClass {

private int id;

private String value;

public int getId() {

return id;}

public void setId(int id) {

this.id = id;}

public String getValue() {

return value;}

public void setValue(String value) {

this.value = value;}

}
```

Serialization with Inheritance

```
package seminar12.serialization.inheritance;


import java.io.IOException;

import java.io.InvalidObjectException;

import java.io.ObjectInputStream;

import java.io.ObjectInputValidation;

import java.io.ObjectOutputStream;

import java.io.Serializable;


public class SubClass extends SuperClass implements Serializable, ObjectInputValidation{

private static final long serialVersionUID = -1322322139926390329L;

private String name;


public String getName() {return name;}

public void setName(String name) {this.name = name;}

@Override

public String toString(){return "SubClass{id="+getId()+",value="+getValue()+",name="+getName()+"}";}
```

Serialization with Inheritance

```
//adding helper method for serialization to save/initialize super class state

private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException{

ois.defaultReadObject();

//notice the order of read and write should be same

setId(ois.readInt());

setValue((String) ois.readObject());        }


private void writeObject(ObjectOutputStream oos) throws IOException{

oos.defaultWriteObject();

oos.writeInt(getId());

oos.writeObject(getValue());}


@Override

public void validateObject() throws InvalidObjectException {

//validate the object here

if(name == null || "".equals(name)) throw new InvalidObjectException("name can't be null or empty");

if(getId() <=0) throw new InvalidObjectException("ID can't be negative or zero");

}     }
```

Serialization with Inheritance

```
package seminar12.serialization.inheritance;

import java.io.IOException;

import seminar12.serialization.SerializationUtil;

public class InheritanceSerializationTest {

public static void main(String[] args) {

String fileName = "subclass.ser";

SubClass subClass = new SubClass();

subClass.setId(10);

subClass.setValue("Data");

subClass.setName("ABC");

try {

SerializationUtil.serialize(subClass, fileName);

} catch (IOException e) {e.printStackTrace();return;}

try {

SubClass subNew = (SubClass) SerializationUtil.deserialize(fileName);

System.out.println("SubClass read = "+subNew);

} catch (ClassNotFoundException | IOException e) {e.printStackTrace();}}

}
```

Serialization Proxy Pattern

- Java Serialization pitfalls:
  - The class structure can't be changed a lot without breaking the java serialization process. So even though we don't need some variables later on, we need to keep them just for backward compatibility.
  - Serialization causes huge security risks, an attacker can change the stream sequence and cause harm to the system. For example, user role is serialized and an attacker change the stream value to make it admin and run malicious code.

Serialization Proxy Pattern

- Serialization Proxy pattern is a way to achieve greater security with Serialization.

  - an inner private static class is used as a proxy class for serialization purpose. This class is designed in the way to maintain the state of the main class.

  - This pattern is implemented by properly implementing readResolve() and writeReplace() methods.

  - See the following example

Serialization Proxy Pattern

```
package seminar12.serialization.proxy;

import java.io.InvalidObjectException;

import java.io.ObjectInputStream;

import java.io.Serializable;


public class Data implements Serializable{

private static final long serialVersionUID = 2087368867376448459L;

private String data;


public Data(String d){ this.data=d;}


public String getData() {return data;}


public void setData(String data) {this.data = data;}


@Override

public String toString(){return "Data{data="+data+"}";

}
```

Serialization Proxy Pattern

```
//serialization proxy class

private static class DataProxy implements Serializable{

private static final long serialVersionUID = 8333905273185436744L;

private String dataProxy;

private static final String PREFIX = "ABC";

private static final String SUFFIX = "DEFG";


public DataProxy(Data d){

//obscuring data for security

this.dataProxy = PREFIX + d.data + SUFFIX;}


private Object readResolve() throws InvalidObjectException {

if(dataProxy.startsWith(PREFIX) && dataProxy.endsWith(SUFFIX)){

return new Data(dataProxy.substring(3, dataProxy.length() -4));

}else throw new InvalidObjectException("data corrupted");

}

}
```

Serialization Proxy Pattern

```
//replacing serialized object to DataProxy object

private Object writeReplace(){

return new DataProxy(this);

}


private void readObject(ObjectInputStream ois) throws InvalidObjectException{

throw new InvalidObjectException("Proxy is not used, something fishy");

}

}
```

Serialization Proxy Pattern

- Data and DataProxy class should implement Serializable interface.
- DataProxy should be able to maintain the state of Data object.
- DataProxy is inner private static class, so that other classes can't access it.
- DataProxy should have a single constructor that takes Data as argument.

- Data class should provide writeReplace() method returning DataProxy instance. So when Data object is serialized, the returned stream is of DataProxy class. However DataProxy class is not visible outside, so it can't be used directly.
- DataProxy class should implement readResolve() method returning Data object. So when Data class is deserialized, internally DataProxy is deserialized and when it's readResolve() method is called, we get Data object.
- Finally implement readObject() method in Data class and throw InvalidObjectException to avoid hackers attack trying to fabricate Data object stream and parse it.

Serialization Proxy Pattern

```
package seminar12.serialization.proxy;

import java.io.IOException;

import seminar12.serialization.SerializationUtil;


public class SerializationProxyTest {

public static void main(String[] args) {

String fileName = "data.ser";

Data data = new Data("ABC");

try {

SerializationUtil.serialize(data, fileName);

} catch (IOException e) {e.printStackTrace();}

try {

Data newData = (Data) SerializationUtil.deserialize(fileName);

System.out.println(newData);

} catch (ClassNotFoundException | IOException e) { e.printStackTrace();}

}

}
```

Serializable objects

- All the reachable objects (the objects that can be reach using the references) are saved into the file only once.

```
class CircularList implements Serializable{
  private class Node implements Serializable{
      Node urm;
      //...
  }
   private Node head; //last node of the list refers to the head of the list
   //...
}
```

- The objects which are referred by a serializable object must be also serializable.

Serializable data structures

```
public class Stack implements Serializable{
   private class Node implements Serializable{
   //...
   }
    private Node top;
   //...
 }
//...
Stack s=new Stack();
 s.push("ana");
 s.push(new Produs("Paine", 2.3));
                  //class Produs must be serializable
//...
 ObjectOuputStream out=...
  out.writeObject(s);
```