Lecture 5
summary of Java SE

presentation

Java Programming – Software App Development

Cristian Toma

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Agenda for Lecture 5 – Summary of JSE

Java annotations – annotation type, meta-annotation and Reflection mechanism

Java Annotations & Reflection

1.1 Java Reflection

- **Java Reflection** is an “introspective technique” that allows a computer program to examine and modify the structure and behavior (specifically the values, meta-data, properties and functions) of an object at runtime. [WIKI]

- **Java Reflection** is an advanced technique and should be used by experienced programmers that have good knowledge of Java and JVM.

- **Java Reflection** is a technique which allows different applications to do various operations that are quit impossible otherwise. It is a common approach for high-level programming languages as Java or C#.

1.1 Java Reflection

Samples for objects and objects arrays in **Java Reflection**:

- Operator ***instanceof***
- Displaying class methods
- Obtaining info about constructors methods
- Obtaining info about class fields
- Invoking methods by name
- Creation of new objects
- Changing the value from various field
- Using the arrays/vectors in Java Reflection context

1.1 Java Reflection

What types of applications are using **Java Reflection**?
Class Browser
Debugger
Test Tool
Dynamic Proxy

What are the disadvantages / issues of the **Java Reflection** techniques?
Performance – the reflection acts at "byte-code" level, but some optimizations of JVM could not be applied.
Security constraints – almost impossible to be applied at Java Applet – Security Manager Module.
Exposing the internal items of a class – it is not recommended but it is possible to access private fields and methods.

Is this technique used within these lectures?
YES – FTP server sample
YES – together with annotations at EJB 3.0 and Web Services

1.2 Java Annotations

- Java Annotation “is the meta-tags that you will use in your code to give it some life.”

- There are two types: “***annotation type***” and “***annotation***”

- Define annotation – ***“annotation type”*** :

```
public @interface MyAnnotation {
   String doSomething();
}
```

- Use annotation – “***annotation***”:

```
@MyAnnotation (doSomething="What to do")
public void mymethod() { .... }
```

1.2 Java Annotations

Three kind of ***"annotation type"*** :

- 1. Marker – does NOT have internal elements

  Sample:

```
public @interface MyAnnotation { }
```

  Usage:

```
@MyAnnotation
 public void mymethod() { .... }
```

- 2. Single Element – has a single element represented by key=value

  Sample:

```
public @interface MyAnnotation {
    String doSomething();
}
```

  Usage:

```
@MyAnnotation ("What to do")
public void mymethod() { .... }
```

1.2 Java Annotations

Kind of ***"annotation type"*** :

- 3. Full-Value / Multi-Value – has multiple internal elements

  Sample:

```
public @interface MyAnnotation {
    String doSomething();
    int count;
    String date();
}
```

  Usage:

```
@MyAnnotation (doSomething="What to do", count=1, date="09-09-
  2005")
public void mymethod() { .... }
```

1.2 Java Annotations

Rules for defining – ***"annotation type"*** :

1. The defining of the annotation should start with '@interface' keyword.
2. The declared methods has no parameters.
3. The declared methods has no "throw exception" statements.
4. The data types of the method are:
    * primitive – byte, char, int, float, double, etc.
    * String
    * Class
    * enum
    * arrays of one of the types from above – int[], float[], etc.

In JDK 5.0 there are predefined / simple – ***"annotation"*** :

1. @Override
2. @Deprecated
3. @SupressWarnings

1.2 Java Annotations

Starting with JDK 5.0 there are *"meta-annotation"* that can be applied only to the *"annotation type"*:

1. Target

```
@Target(ElementType.TYPE)
@Target(ElementType.FIELD)
@Target(ElementType.METHOD)
@Target(ElementType.PARAMETER)
@Target(ElementType.CONSTRUCTOR)
@Target(ElementType.LOCAL_VARIABLE)
@Target(ElementType.ANNOTATION_TYPE)
```

2. Retention
   - @Retention(RetentionPolicy.SOURCE) – retinute la nivel cod sursa si sunt ignorate de compilator
   - @Retention(RetentionPolicy.CLASS) – retinute la nivel de compilare dar ignorate de VM la run-time
   - @Retention(RetentionPolicy.RUNTIME) – sunt retinute si utilizate doar la run-time

3. Documented – @Documented

4. Inherited – @Inherited

Section Conclusion

Fact: **Java Annotations & Reflection**

In few **samples** it is simple to remember: Java annotations, annotations types, reflection and sample for combining annotation with reflection. The combining approach is used in behind by major web/app JEE servers for technologies like EJB or Web-Services plus XML parsing in JAXB and J-Unit for QA.

Java Libraries - JAR, Input / Output & JNI – Java Native Interface on Linux & Windows

Java I/O & JNI

2.1 Java Library

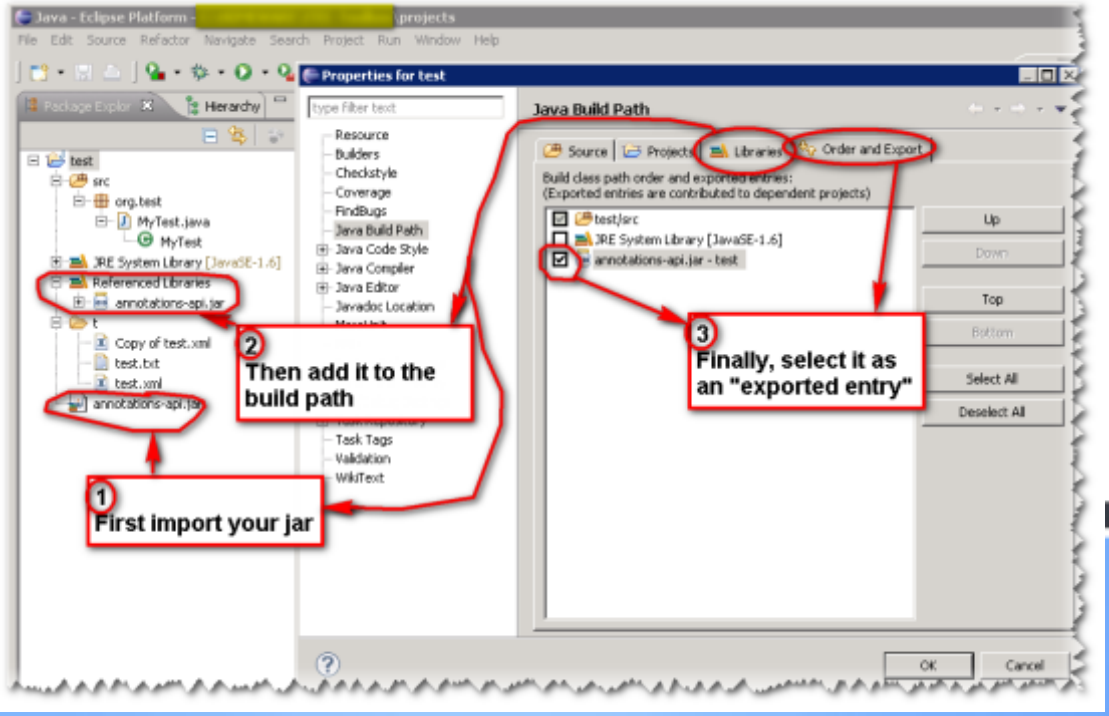
What is a **Java library**?

What are the advantages and disadvantages of Java libraries?

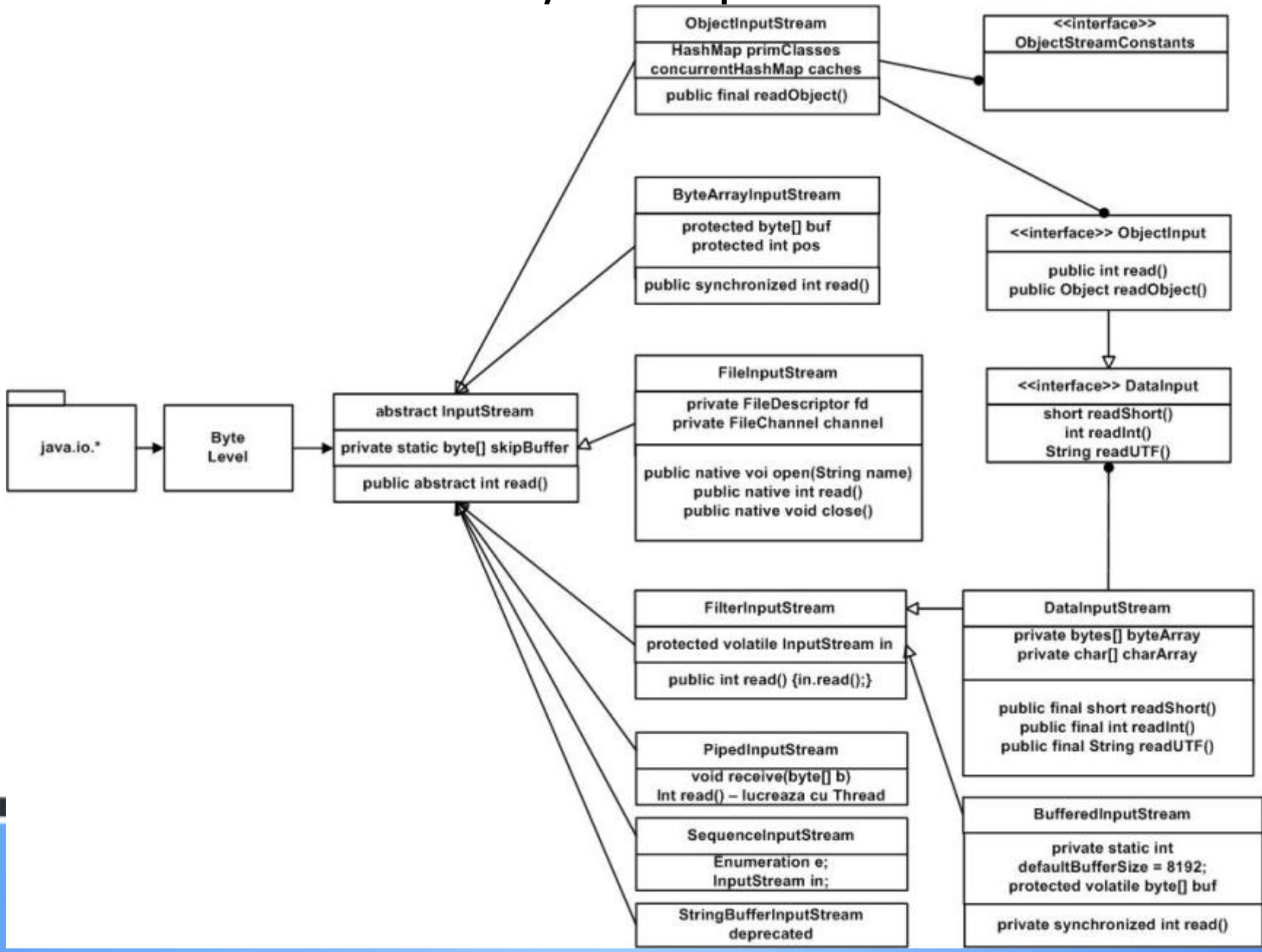
How can be solved in Java multiple dependencies or inclusions of the same class in the compilation phase? How were solved these problems in C/C++?

How should be created a Java library and how should be used – command line vs. IDE?

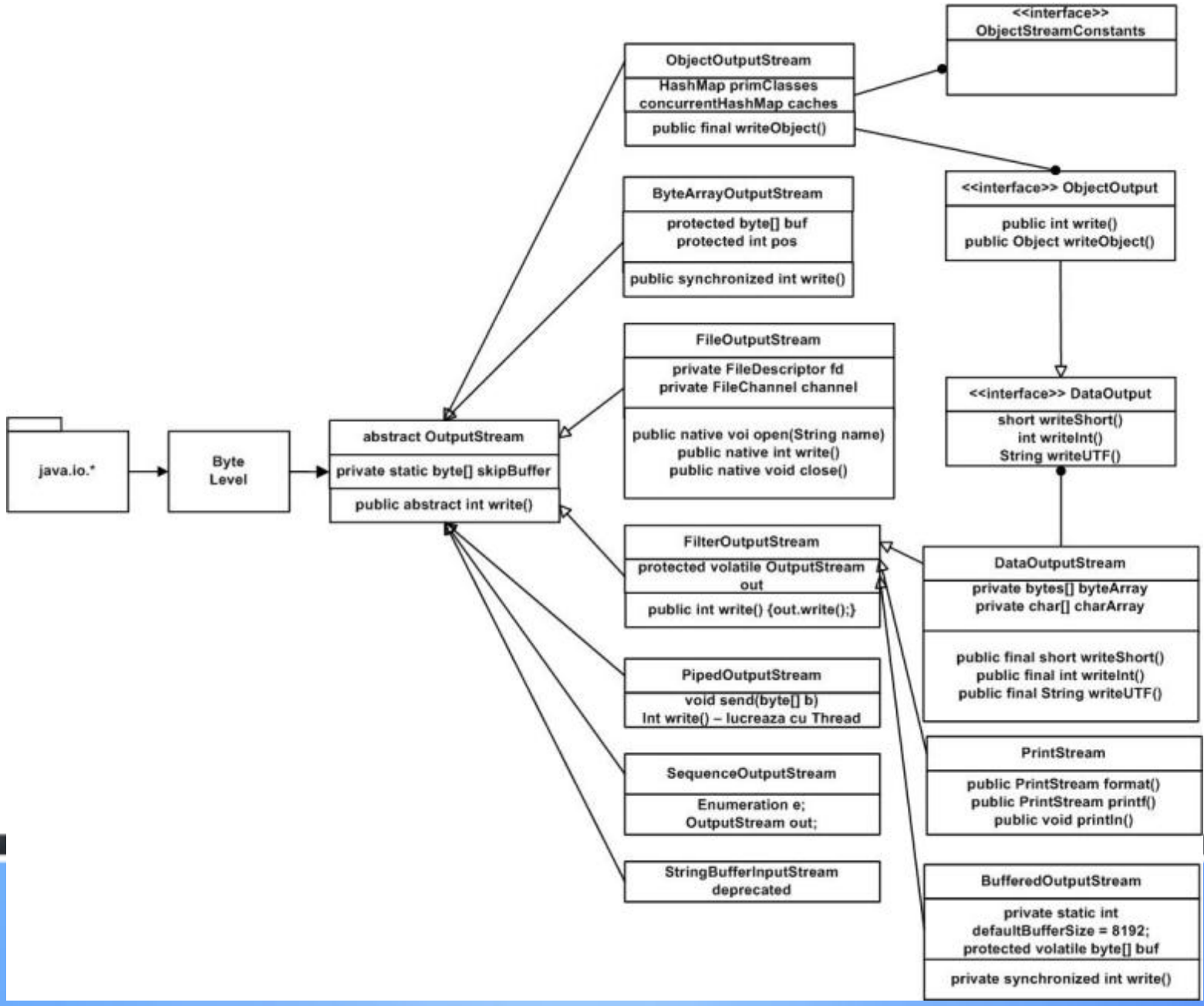
>jar -cvf archive_name.jar files_names_to_compress
>javac -classpath .:archieve_name.jar *.java
>java –classpath .:archive_name.jar file_with_main_class



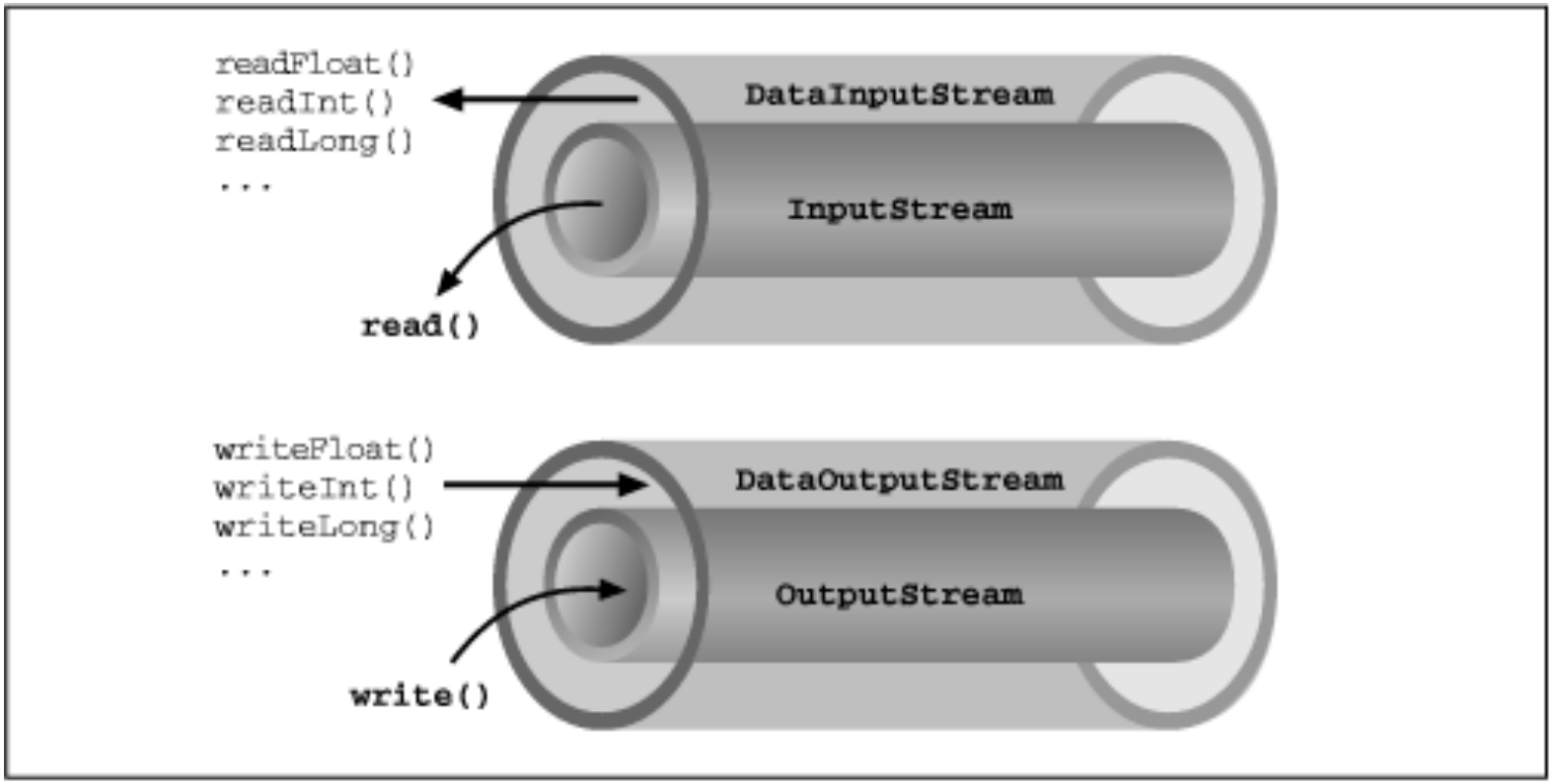
2.1 Java I/O – Input Stream



2.1 Java I/O – Output Stream

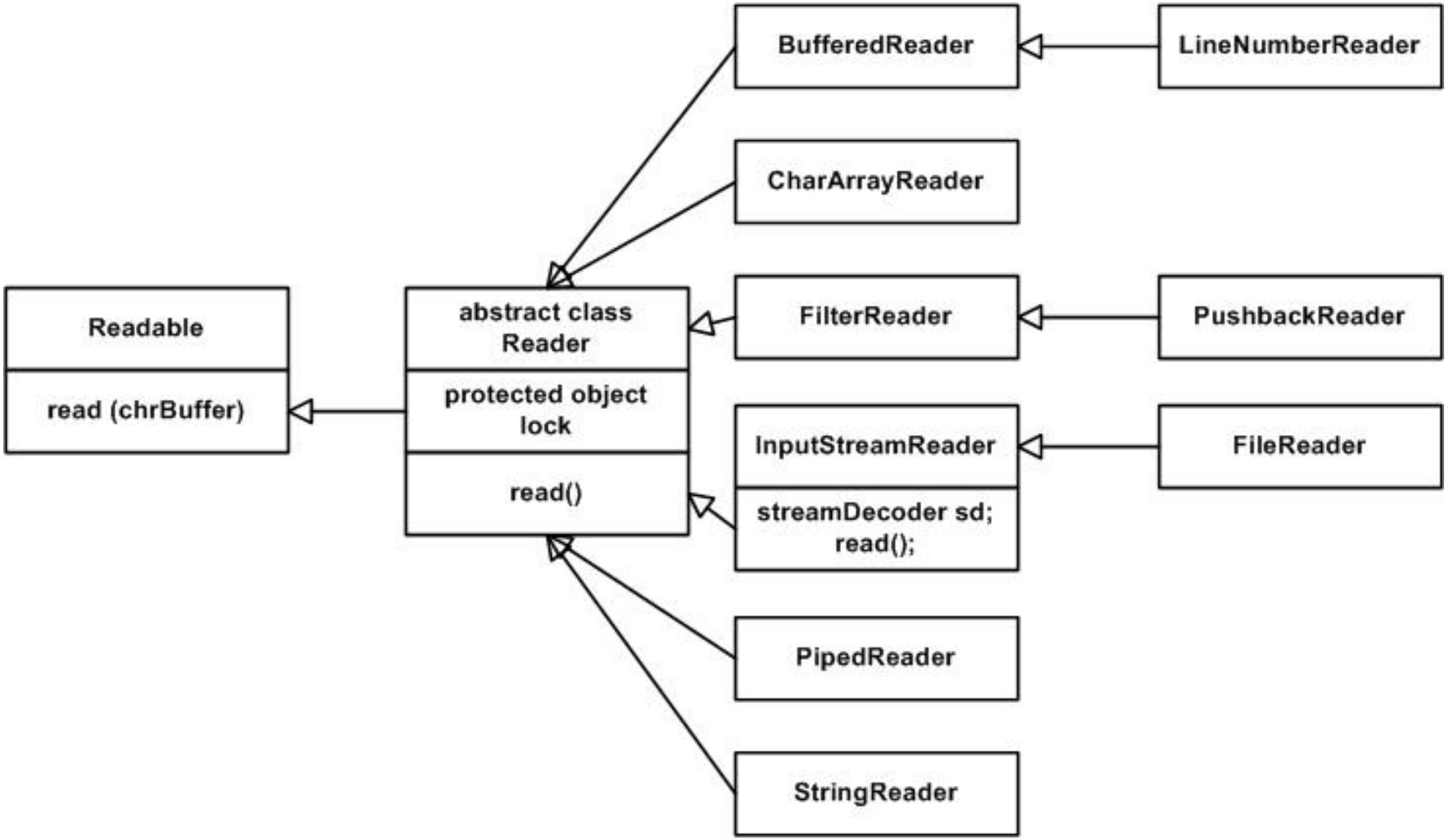


2.1 Java I/O – Streams Encapsulation

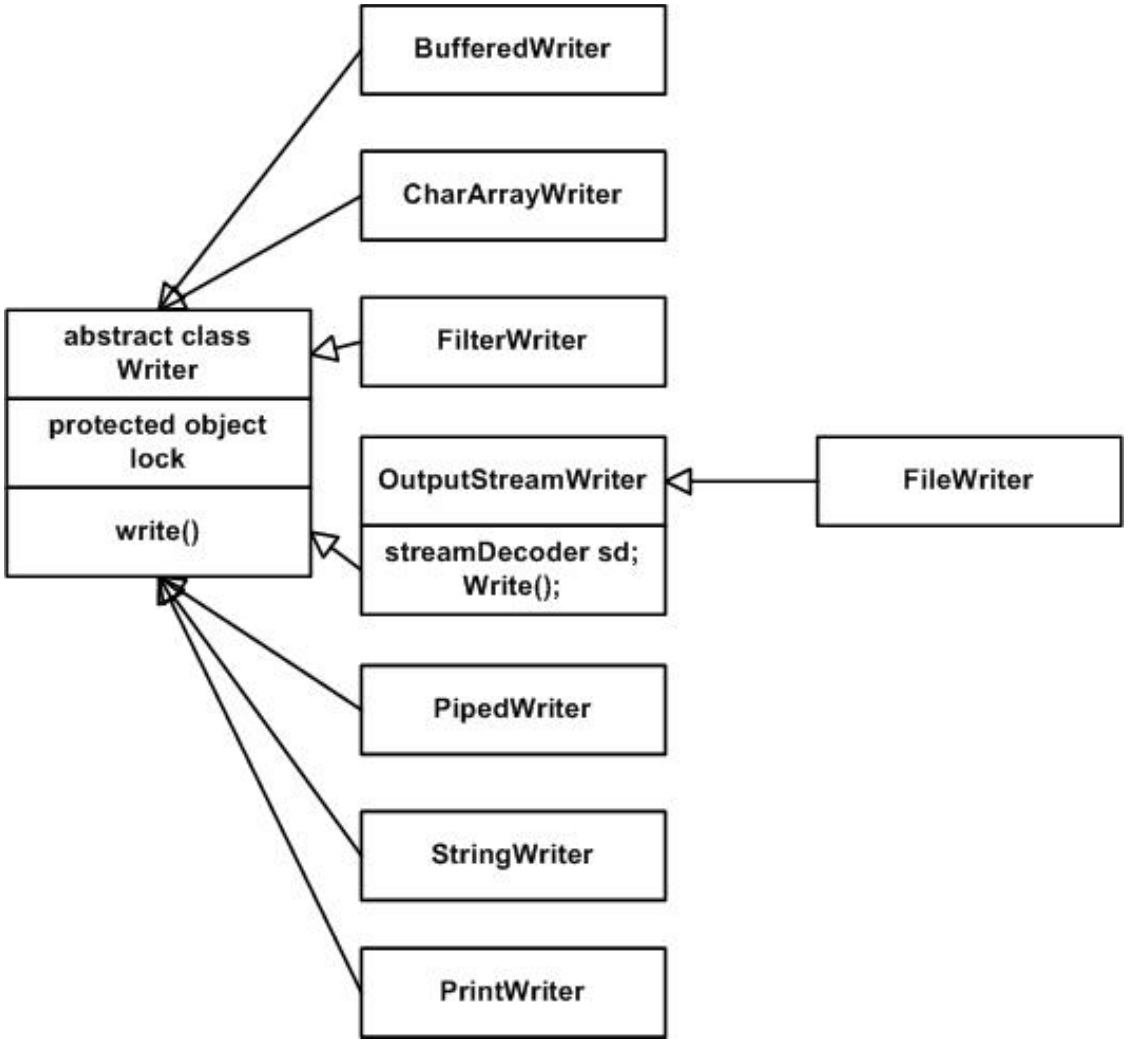


http://doc.sumy.ua/prog/java/exp/ch10_01.htm

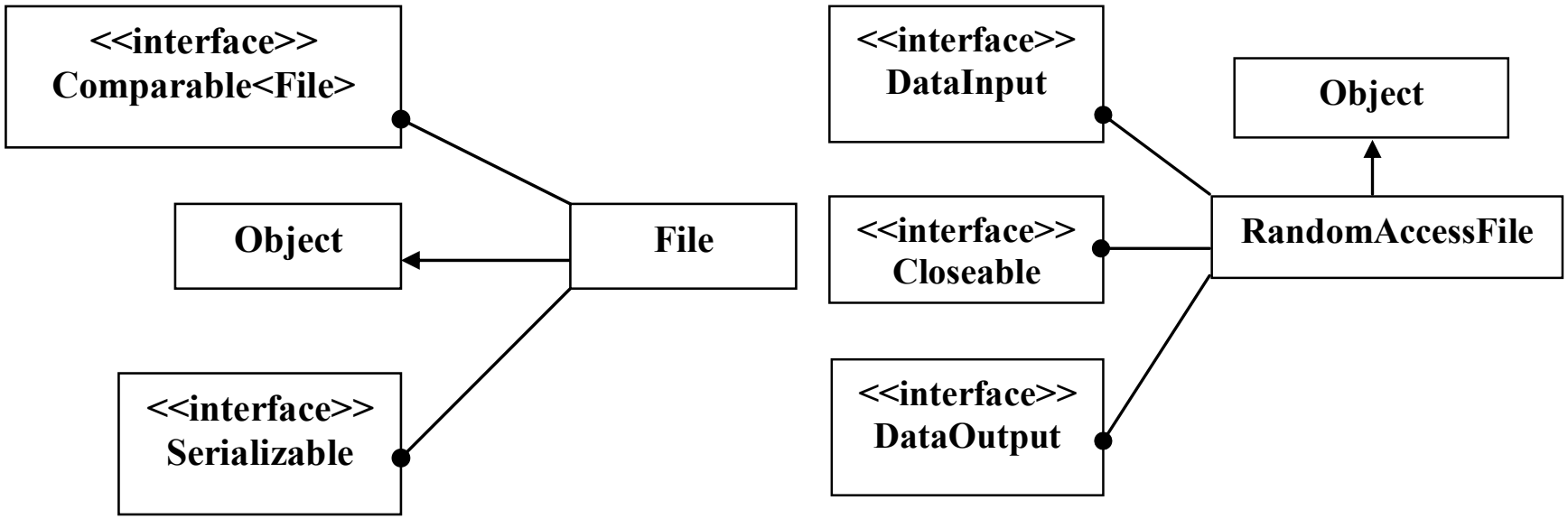
2.1 Java I/O – char level reading



2.1 Java I/O – char level writing

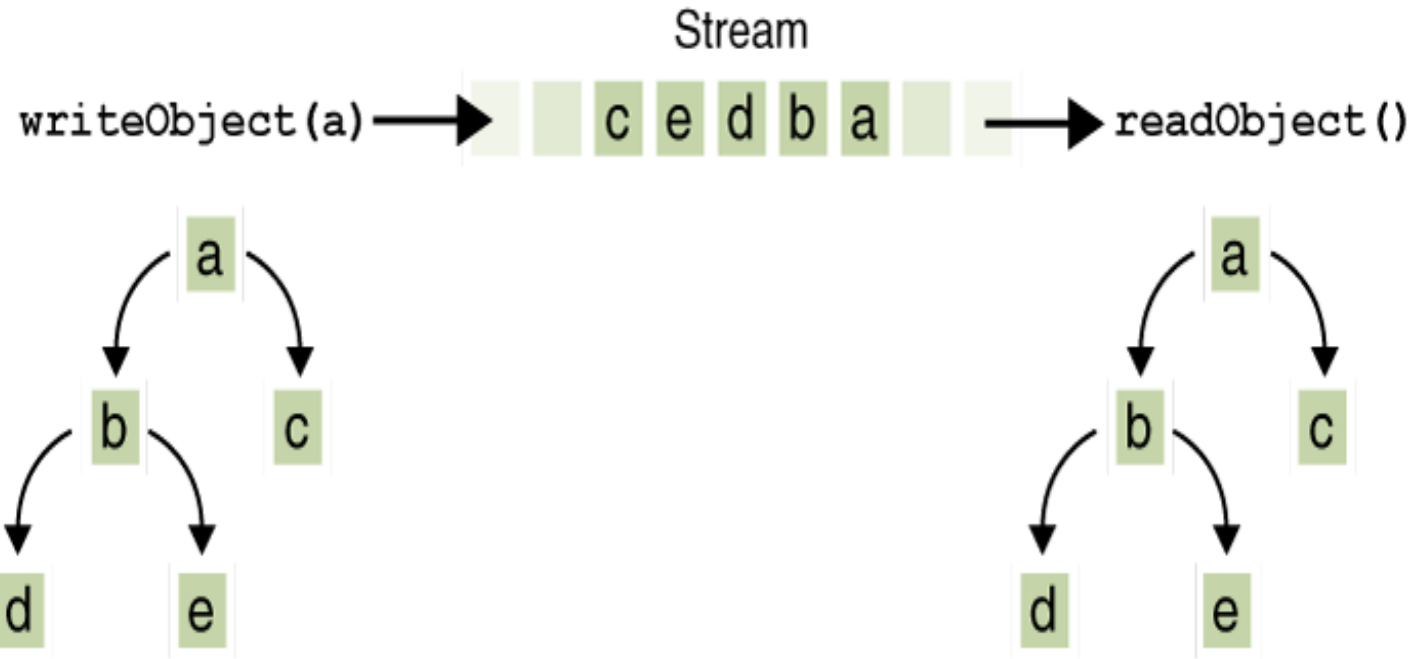


2.1 Java I/O – File Access



2.1 Java I/O – Serialization

This is demonstrated in the following figure, where `writeObject` is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b** contains references to **d** and **e**. Invoking `writeobject(a)` writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by `readObject`, the other four objects are read back as well, and all the original object references are preserved.



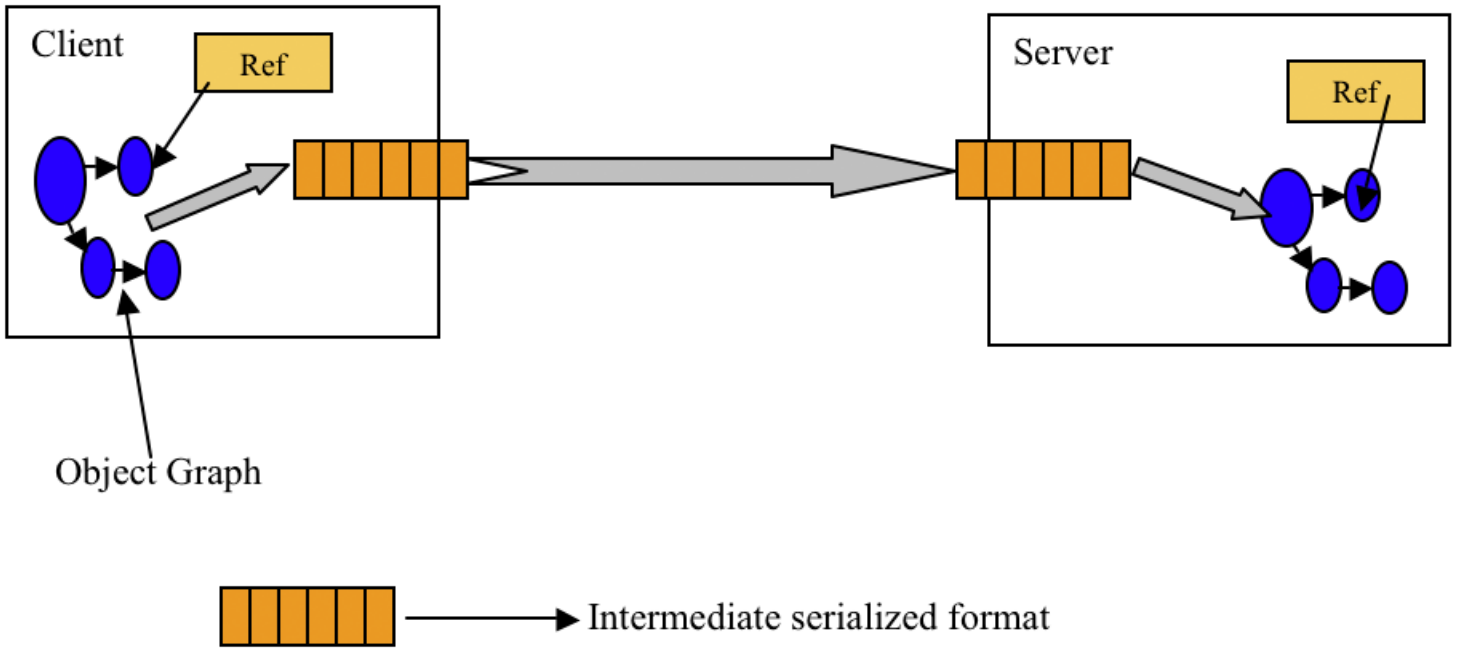
I/O of multiple referred-to objects

2.1 Java I/O – Serialization

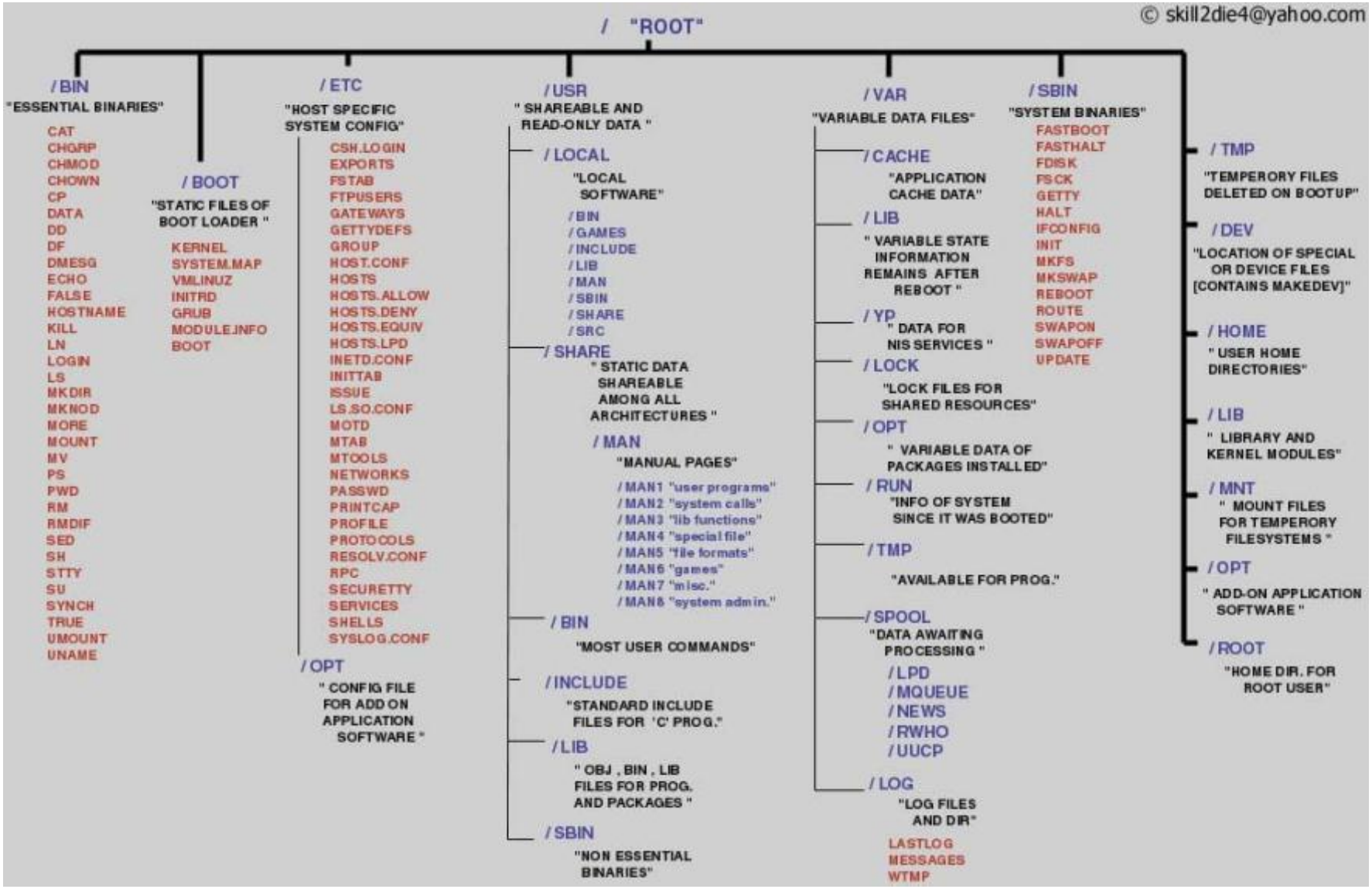
What is going to be saved and restored by serialization in Java?

- **Non-static fields? Static fields?**
- **Transient fields?**
- **Private and public fields and/or methods?**
- **Prototype / signature of the methods and / or the implementation of the methods?**

http://www.javaworld.com/community/node/2915



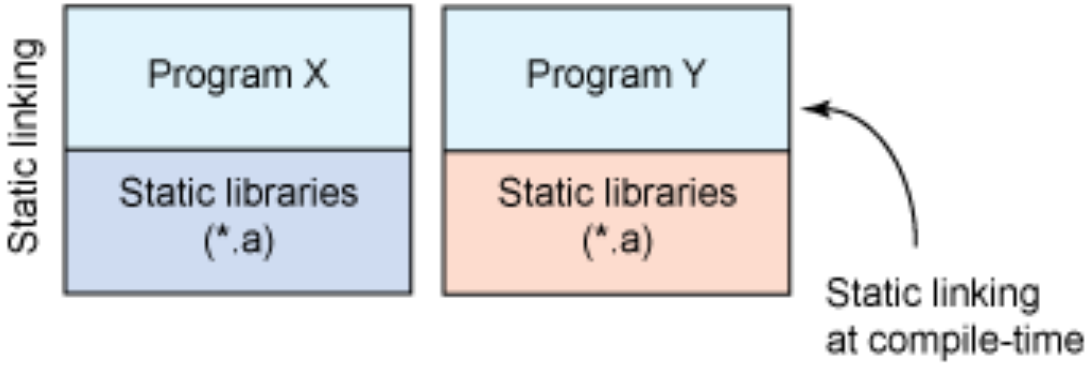
2.2 JNI – Linux Directories

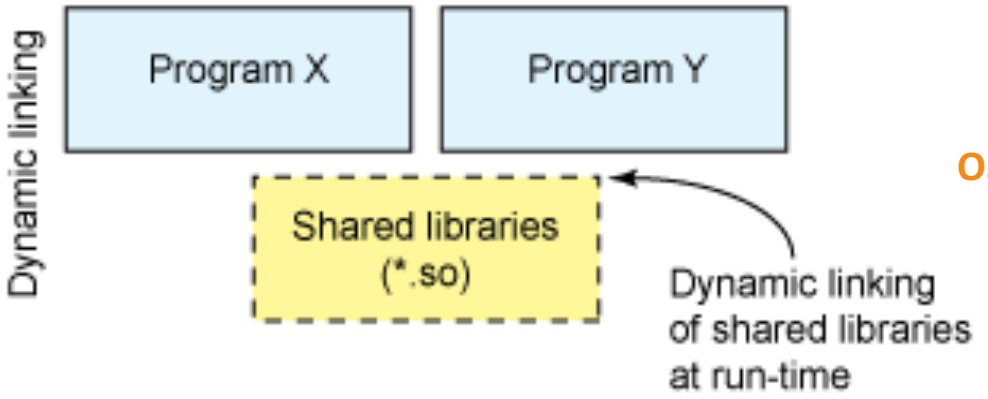


http://freshtutorial.com/file-structure-linux/

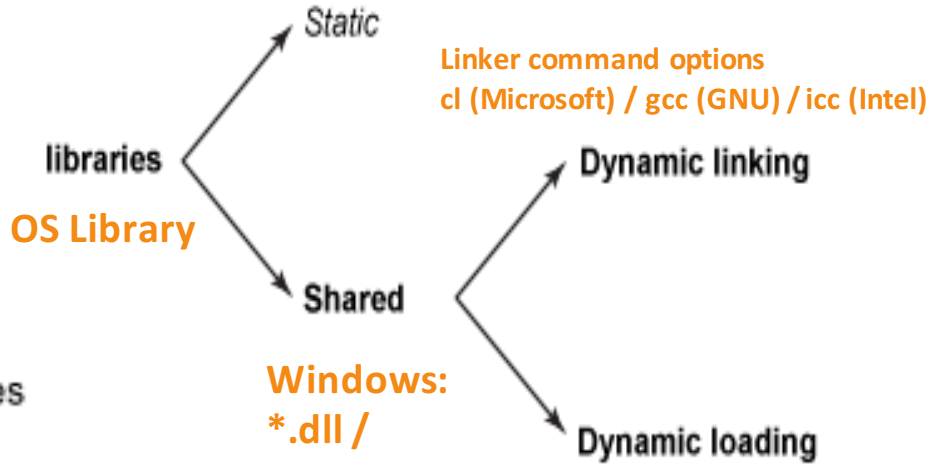
2.2 JNI – Linux vs. Win libraries

http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/





Windows: *.lib / Linux: *.a



Linker command options
cl (Microsoft) / gcc (GNU) / icc (Intel)

OS Library

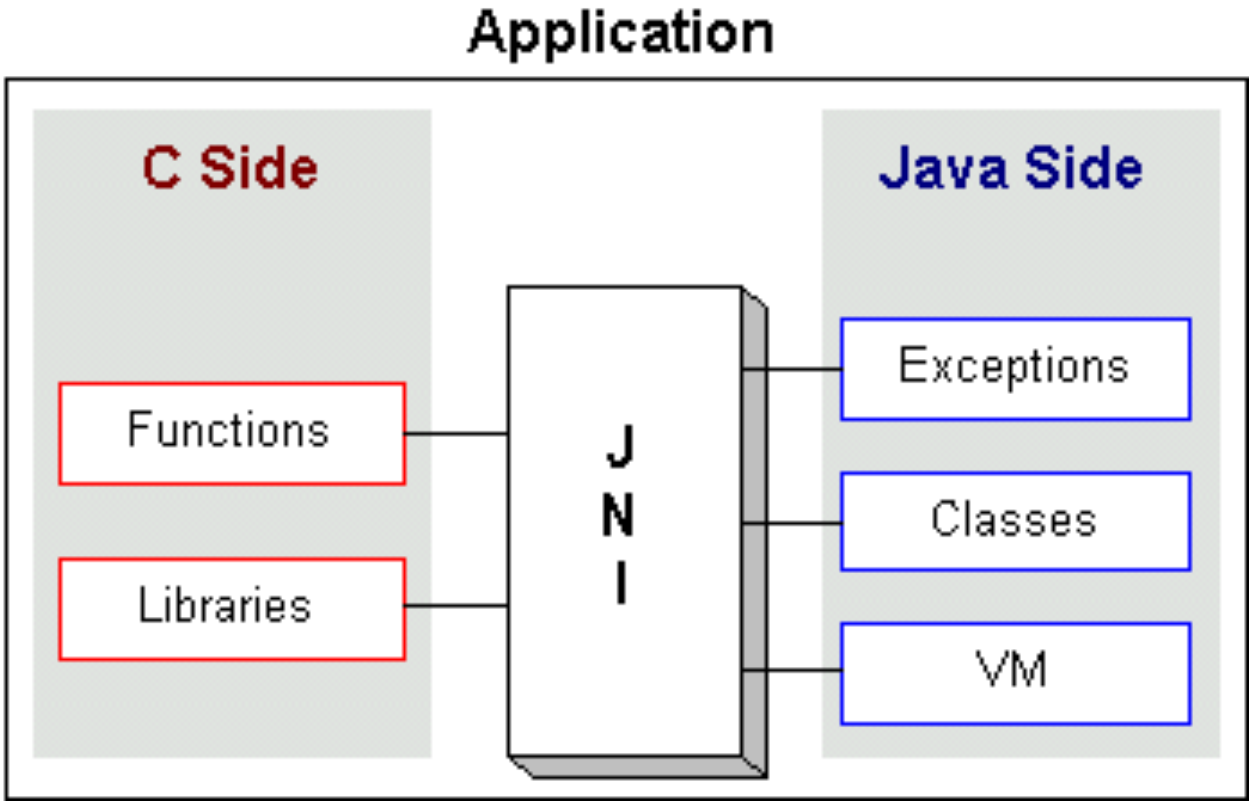
Windows: *.dll /

Linux: *.so

@ Run-time

```
/* Open the shared object */
dl_handle = dlopen( lib, RTLD_LAZY );
if (!dl_handle) {
  printf( "!!! %s\n", dlerror() );
  return;
}
```

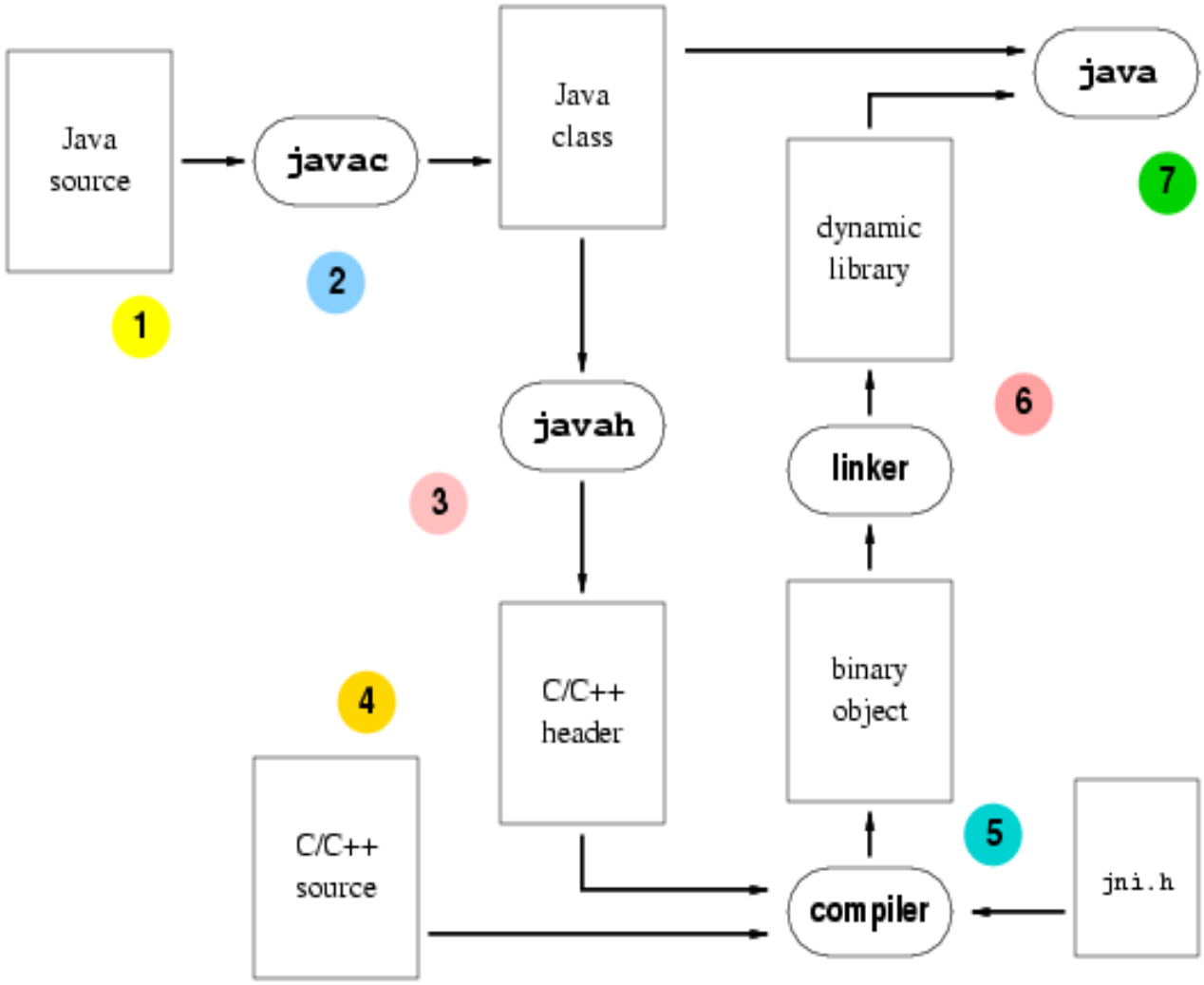
2.2 Java Native Interface



http://www.iam.ubc.ca/guides/javatut99/native1.1/concepts/index.html

2.2 Java Native Interface

http://cs.fit.edu/~ryan/java/language/jni.html



1. Create Java source code with native methods

native *return type* method (*arguments*);

2. Compile Java source code and obtain the class files
3. Generate C/C++ headers for the native methods; javah gets the info it needs from the class files
4. Write the C/C++ source code for the native method using the function prototype from the generated include file and the typedefs from include/jni.h
5. Compile the C/C++ with the right header files
6. Use the linker to create a dynamic library file
7. Execute a Java program that loads the dynamic library

static {
System.loadLibrary("*dynamic libary*"); }

Java I/O & Serialization + JNI
for easy sharing

Section Conclusions

Java I/O helps the Java programmers to save / restore data to / from HDD – Hard-disk, SAN – Storage Area Network or Network – it is even in behind of database communications – JDCB.

Java I/O is divided in input / output streams that operates over byte level and readers / writers at char level. Both streams and readers / writers may be specialized to work with primitive data and even objects.

Java Serialization mechanisms helps the developers to save / restore the objects / instances from HDD, Network, SAN or JDBC-BLOB.

JNI – Java Native Interface is the intermediary bridge between JVM – Java Virtual Machine and OS – Operating System.

JUnit Simple Example using Annotation, Full Intro in SQE/QA lectures

JUnit Sample

Questions & Answers!

Thanks!

Java Programming
End of Lecture 5 – summary of Java SE