Chapter 10.

Serialization and File I/O

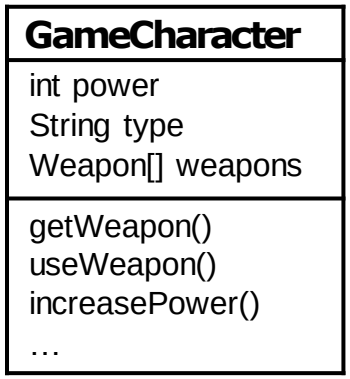
2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

Saving and Restoring State

- Ocean was asked to develop a fantasy adventure game that takes more than 1 session to complete
- As the game progresses, characters in the game become stronger, smarter, etc., and gather and use weapons
- Obviously, players of the game would not want to start from scratch each time they launch the game (it takes them ages to get their characters in top shape for a spectacular battle)
- Ocean needed a way to save the state of the characters and a way to restore the state when the game resumes

Saving and Restoring State

- Ocean had 2 options
  - Option 1: Create a file and write the serialized objects to the files
  - Option 2: Create a file and write 1 line of text per character, separating the pieces of state with delimiters (e.g., commas)
- A serialized file is much harder for humans to read, but is much easier and safer for your program to restore the objects
- A plain text file is easier for humans (and other non-Java programs) to read, but care must be taken when reading in the objects' variable values (order matters!)



Writing Objects to a File

- The 4 simple steps in serializing an object

1. Make a FileOutputStream

```
FileOutputStream fileStream = new FileOutputStream("MyGame.sav");
```

2. Make an ObjectOutputStream

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

3. Write the objects

```
os.writeObject(elf);
os.writeObject(troll);
os.writeObject(magician);
```

If the file "MyGame.sav" does not exist, it will be created automatically

4. Close the ObjectOutputStream

```
os.close();
```

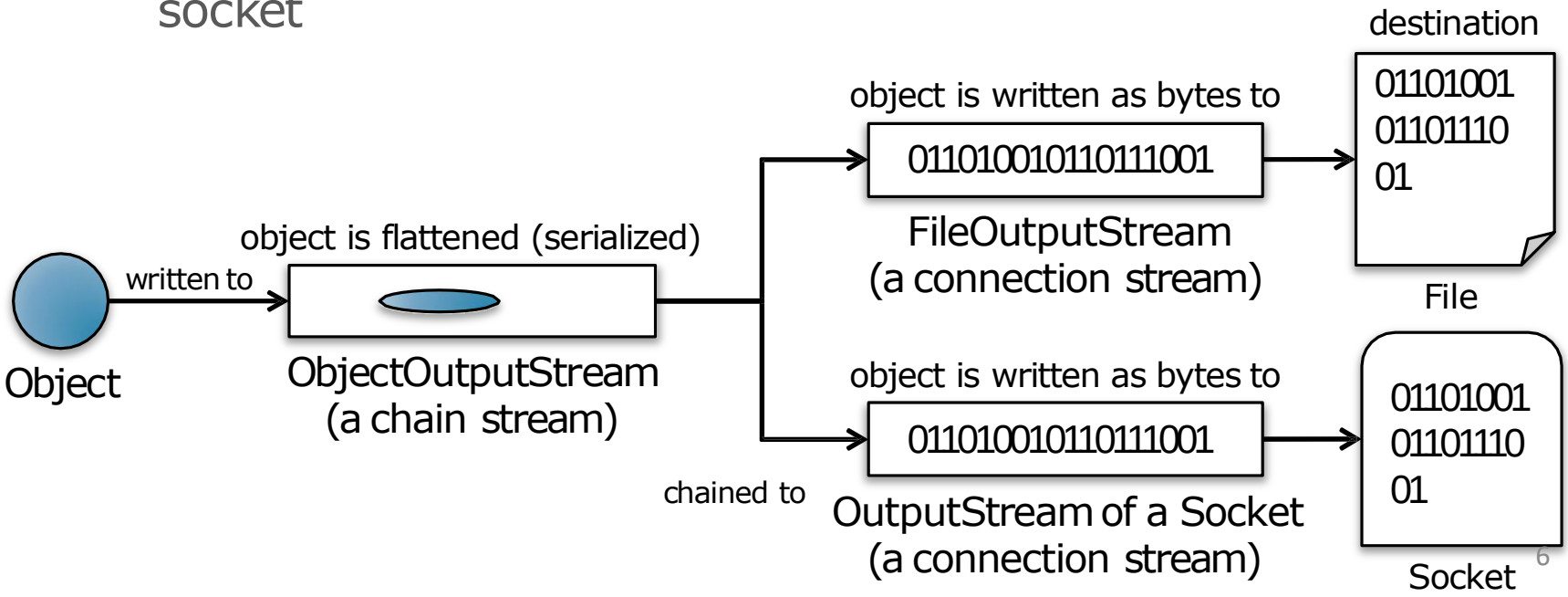
Closing the stream at the top closes the ones underneath, so FileOutputStream will close automatically

Connection Streams and Chain Streams

- The Java I/O API has 2 types of streams, namely connection streams and chain streams
- A connection stream (e.g., FileOutputStream) represents a connection to a source or destination (e.g., file, network socket, etc.). It has low-level methods for serialization (e.g., writing bytes to a destination)
- A chain stream (e.g., ObjectOutputStream) works only if chained to another stream. It has higher-level methods for serialization (e.g., writing objects to another stream)
- Often, it takes at least 2 streams hooked together to do something useful
- The ability to mix and match different combinations of connection and chain streams gives you tremendous flexibility

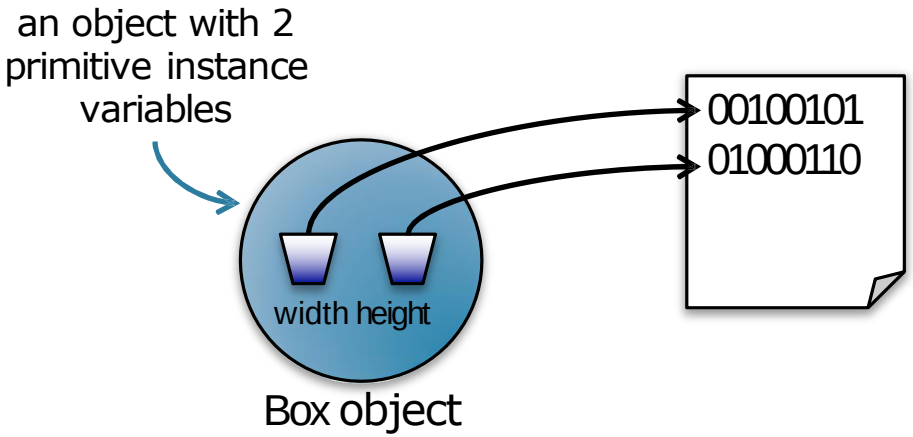
Connection Streams and Chain Streams

- Example
  - The ObjectOutputStream turns objects into data that can be written to a connection stream
  - The FileOutputStream represents a connection to a file and has methods for writing bytes to the file
  - The OutputStream of a Socket represents a connection to a network socket and has methods for writing bytes to the socket



Serializing an Object

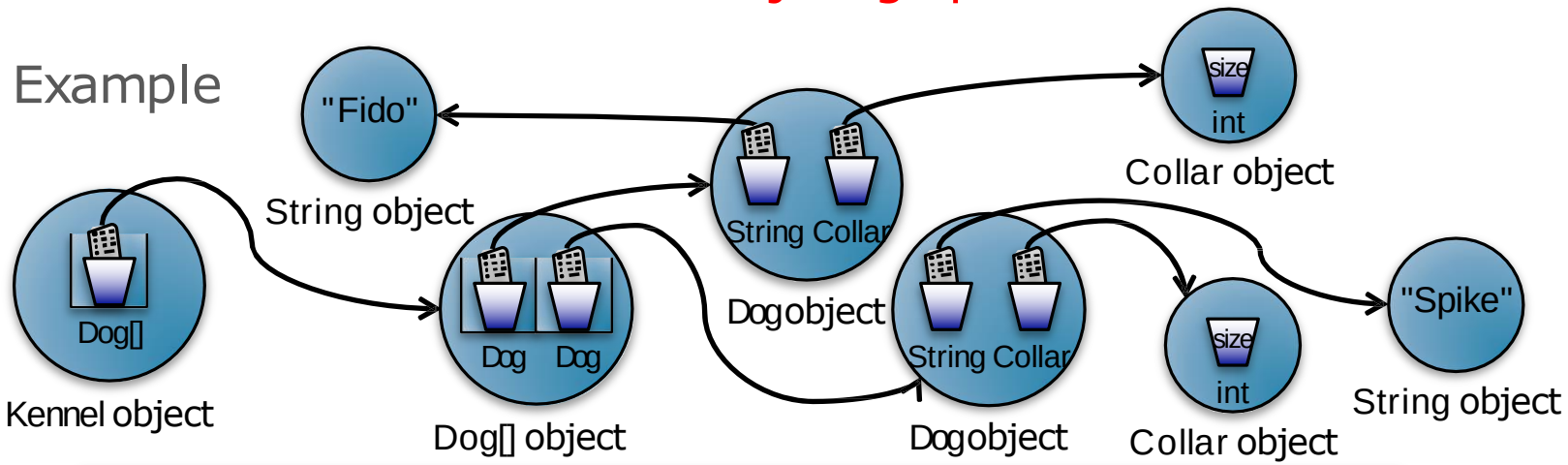
- Objects on the heap have state represented by the values of their instance variables. These values make one instance of a class different from another instance of the same class
- Serialized objects save the values of the instance variables, so that an identical instance (object) can be brought back to life on the heap



The values of the instance variables width and height are saved to the file, along with a little more info the JVM needs to restore the object (like what its class type is)

Serializing an Object

- When an object is serialized, all the objects it refers to from its instance variables are also serialized
- Serialization saves the entire object graph!
- Example



- Serialization is smart enough to know when 2 objects in the graph are the same and save only one of the 2 objects

Serializable Interface

- A class should implement the Serializable interface if it needs to be serializable

```
public class Box implements Serializable { … }
```

- The Serializable interface is known as a marker or tag interface because it does not have any methods to implement
- Its sole purpose is to announce that the class implementing it is serializable
- If any superclass of a class is serializable, the subclass is automatically serializable even if it does not explicitly declare it implements Serializable

Serializable Interface

Example

```
import java.io.*;

public class Box implements Serializable {
   private int width;
   private int height;

   public void setWidth(int w) { width = w; }
   public void setHeight(int h) { height = h; }

   public int getWidth() { return width; }
   public int getHeight() { return height; }
}
```

Serializable is in the java.io package

Just declare the class implements Serializable, no methods to implement

The values of all the instance variables will be saved

Serializable Interface

- Example

```
import java.io.*;

public class BoxTestDrive {
   public static void main(String[] args) {
      Box box = new Box();
      box.setWidth(70);
      box.setHeight(30);

      try {
         FileOutputStream fs = new FileOutputStream("Box.sav");
         ObjectOutputStream os = new ObjectOutputStream(fs);
         os.writeObject(box);  os.close();
      } catch (Exception ex) {
         ex.printStackTrace();
      }

   }
}
```

I/O operations can throw exceptions

Connect to a file named "Box.sav", it exists. If it does not exist, make a new file named "Box.sav"

Make an ObjectOutputStream chained to the connection stream

Serialization is All or Nothing

- Either the entire object graph is serialized correctly or serialization fails
- An object cannot be serialized if any of the objects it refers to from its instance variables is not serializable
- Example

```
import java.io.*;

public class Pond implements Serializable {
   private Duck duck = new Duck();
   // ...
}
```

```
public class Duck  {
   // duck code here
}
```

Pond implements Serializable. However, when you try to serialize a Pond object, it fails because the Duck object referenced by an instance variable of the Pond object is not serializable. This will result in a NotSerializableException

Transient Instance Variables

- Some objects cannot be saved because they are instances of classes that do not implement the Serializable interface
- Some objects depend on runtime-specific information that simply should not be saved (e.g., network connections, threads, file objects, etc.). They should be created from scratch each time
- Mark an instance variable as transient if it cannot or should not be saved, and it will be skipped by the serialization process
- Example:

```
import java.io.*;

public class Chat implements Serializable {
    transient String currentID;
    String userName;
    // ...
}
```

currentID is marked as transient and will be skipped in the serialization process (i.e., it will not be saved)

Restoring Objects from a File

- The 4 simple steps in deserializing an object

1. Make a FileInputStream

```
FileInputStream fileStream = new FileInputStream("MyGame.sav");
```

2. Make an ObjectInputStream

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

3. Read and cast the objects

```
GameCharacter elf = (GameCharacter) os.readObject();
GameCharacter troll = (GameCharacter) os.readObject();
GameCharacter magician = (GameCharacter) os.readObject();
```

4. Close the ObjectInputStream

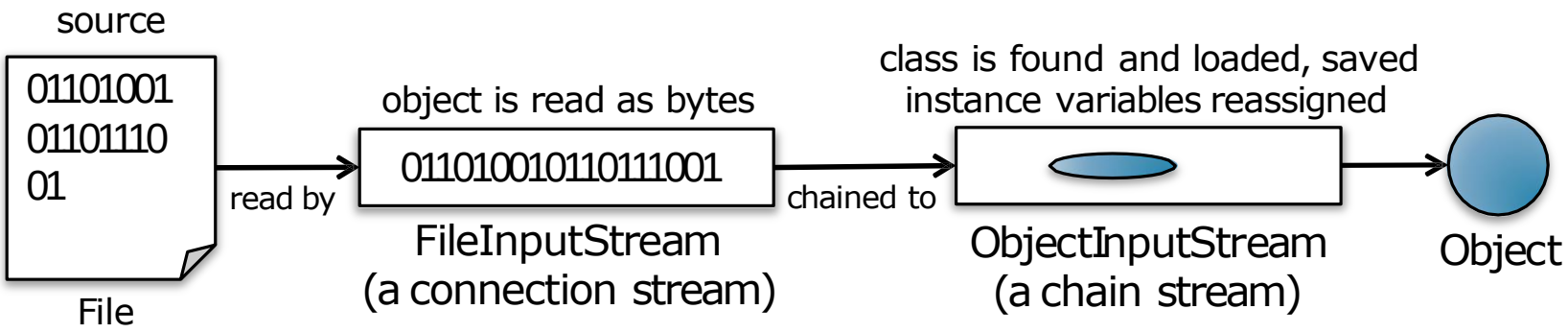
```
os.close();
```

You will get an exception if the file "MyGame.sav" doesn't exist

Read the objects in the same order in which they are written

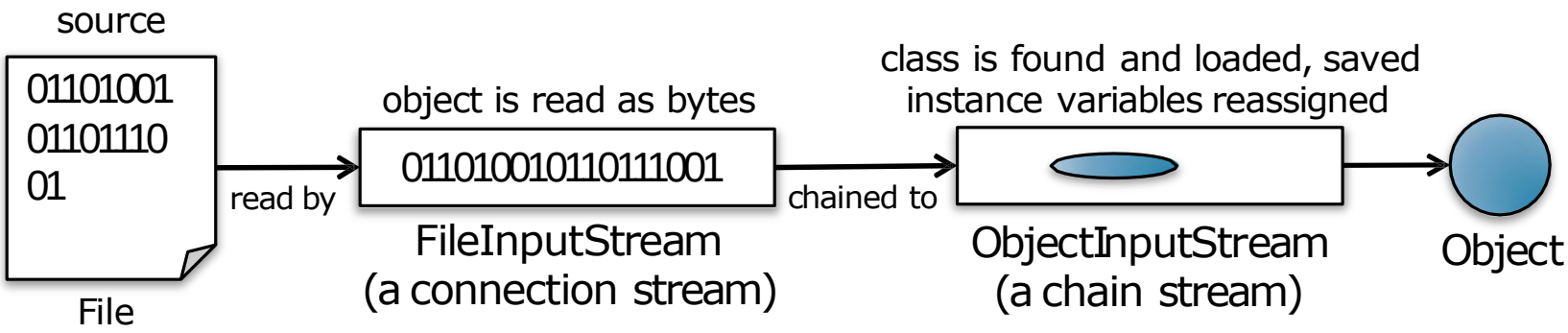
Closing the stream at the top closes the ones underneath, so FileInputStream will close automatically

Deserializing an Object



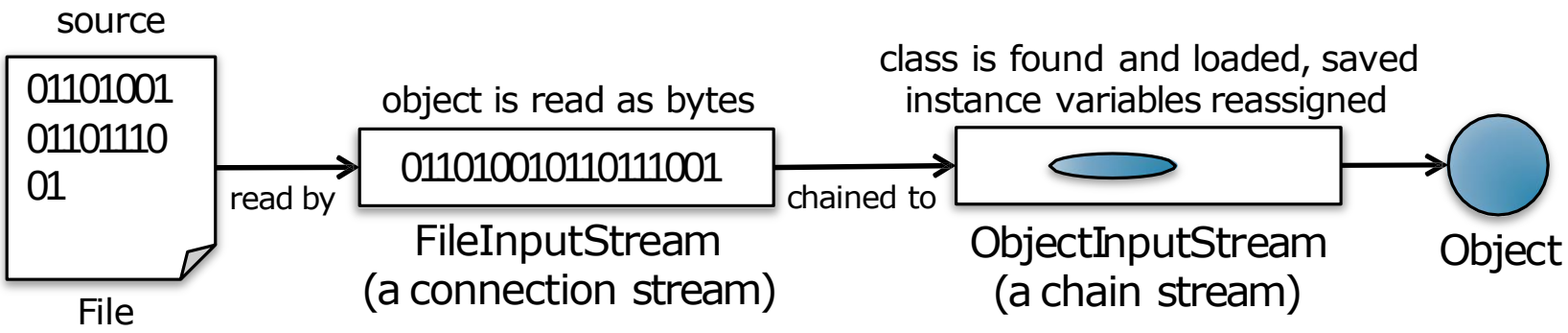
- When an object is deserialized, its data is read from the stream
- The JVM determines the object's class type through info stored with the serialized object, and attempts to find and load the object's class
- If the JVM cannot find and/or load the class, it throws an exception and deserialization fails

Deserializing an Object



- A new object is given the space on the heap, but the constructor of the serialized object does not run (otherwise the object will be restored to its original new state)
- If the object has a non-serializable superclass somewhere up its inheritance tree, the no-argument constructor for that non-serializable class will run, along with any constructors above it

Deserializing an Object



- Once the constructor chaining begins, you cannot stop it, which means all superclasses, beginning with the first non-serializable one, will reinitialize their states
- The instance variables of the object are given the values from the serialized state
- Transient variables are given default values

Version Control is Crucial

- Each time an object is serialized, it is 'stamped' with a serialVersionUID which is computed based on information about the class structure
- If the class has been modified since the object was serialized, the class could have a different serialVersionUID
- When Java tries to deserialize an object, it compares the serialVersionUID of the serialized object with that of the class the JVM is using for deserializing the object
- If the 2 numbers do not match, the JVM assumes the class is not compatible with the previously-serialized object, this will result in an exception during deserialization

Using the serialVersionUID

- It is possible to keep the same serialVersionUID for a class even though it has actually been modified
- This can be done by putting a serialVersionUID in the class
  1. Use the serialver tool that ships with your Java development kit to get a serialVersionUID for a class

```
% serialver Dog
Dog: private static final long serialVersionUID = 6385147890911367760L;
```

  2. Copy and paste the output into the class

```
public class Dog {
   private static final long serialVersionUID = 6385147890911367760L;
   private String name;
   private int size;
   // method code here
}
```

Using the serialVersionUID

- When putting the serialVersionUID in a class, the programmer must take responsibility in ensuring the modified class is compatible with previously-serialized objects
- Changes to a class that can hurt deserialization
  - Deleting an instance variable
  - Changing the declared type of an instance variable
  - Changing a non-transient instance variable to transient
  - Moving a class up or down the inheritance hierarchy
  - Changing a class (anywhere in the object graph) from serializable to non-serializable
  - Changing an instance variable to static

Using the serialVersionUID

- Changes to a class that are usually OK
  - Adding new instance variable to the class (existing objects will deserialize with default values for the instance variables they didn't have when they were serialized)
  - Adding classes to the inheritance tree
  - Removing classes from the inheritance tree
  - Changing the access level of an instance variable
  - Changing an instance variable from transient to non-transient (previously-serialized objects will have a default value for the previously-transient variables)

Writing to a Text File

- Writing text data is similar to writing an object, except you write a String instead of an object, and you use a FileWriter instead of a FileOutputStream
- Example:

```
import java.io.*;

public class WriteFile {
   public static void main(String[] args) {
      try {
         FileWriter writer = new FileWriter("Test.txt");
         writer.write("This is a plain text file");
         writer.close();
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

FileWriter is in the java.io package

I/O operations can throw exceptions

If the file "Test.txt" does not exist, it will be created automatically

FileWriter writer = new FileWriter("Test.txt", true); ➔ append contents to the file

BufferedWriter

- FileWriter writes each and every thing you pass to the file each and every time
- This makes your application less efficient since every trip to the disk is a big deal compared to manipulating data in memory
- By chaining a BufferedWriter (a chain stream) to a FileWriter, the BufferedWriter will hold all the stuff you write to it until it is full
- Only when the buffer is full will the FileWriter actually be told to write to the file on disk
- It is also possible to tell the BufferedWriter to send data before its buffer is full by calling its `flush()` method

BufferedWriter

Example

```
import java.io.*;

public class BufferedWriteFile {
    public static void main(String[] args) {
        try {
            FileWriter fileWriter = new FileWriter("Test.txt");
            BufferedWriter writer = new BufferedWriter(fileWriter);
            writer.write("This is a plain text file");
            writer.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Use a BufferedWriter to improve the efficiency

Reading from a Text File

- Reading text from a file is also simple, and you will need a FileReader and a BufferedReader (for efficient reading)
- Example

```
import java.io.*;

public class BufferedReadFile {
   public static void main(String[] args) {
      try {
        FileReader fileReader = new FileReader("Test.txt");
        BufferedReader reader = new BufferedReader(fileReader);

        String line = null;
        while ((line = reader.readLine()) != null) {
           System.out.println(line);
        }
        reader.close();
      } catch (Exception ex) { ex.printStackTrace(); }
   }
}
```

Chapter 10.

End

2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong