13

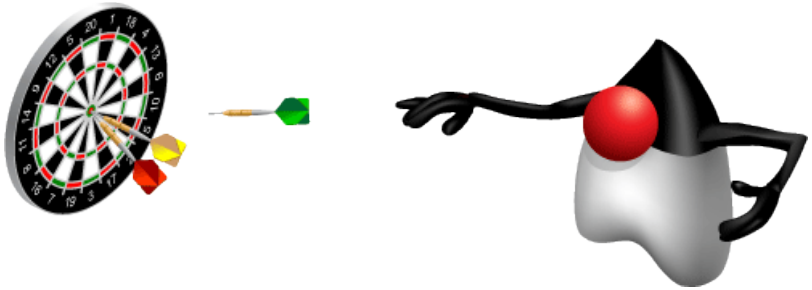
Java I/O Fundamentals

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use I/O streams to read and write files
- Read and write objects by using serialization



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java I/O Basics

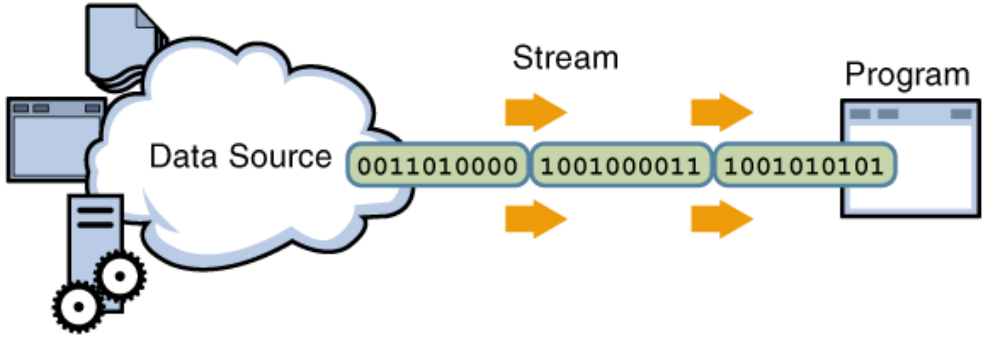
The Java programming language provides a comprehensive set of libraries to perform input/output (I/O) functions.

- Java defines an I/O channel as a stream.
- An I/O stream represents an input source or an output destination.
- An I/O stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- I/O streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

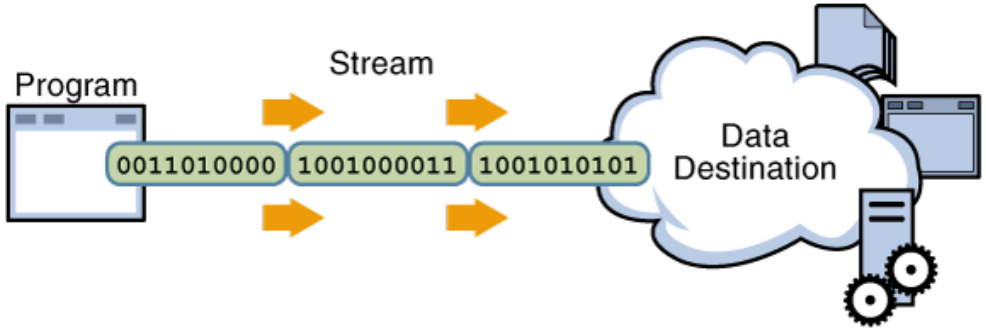
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

I/O Streams

- A program uses an input stream to read data from a source, one item at a time.



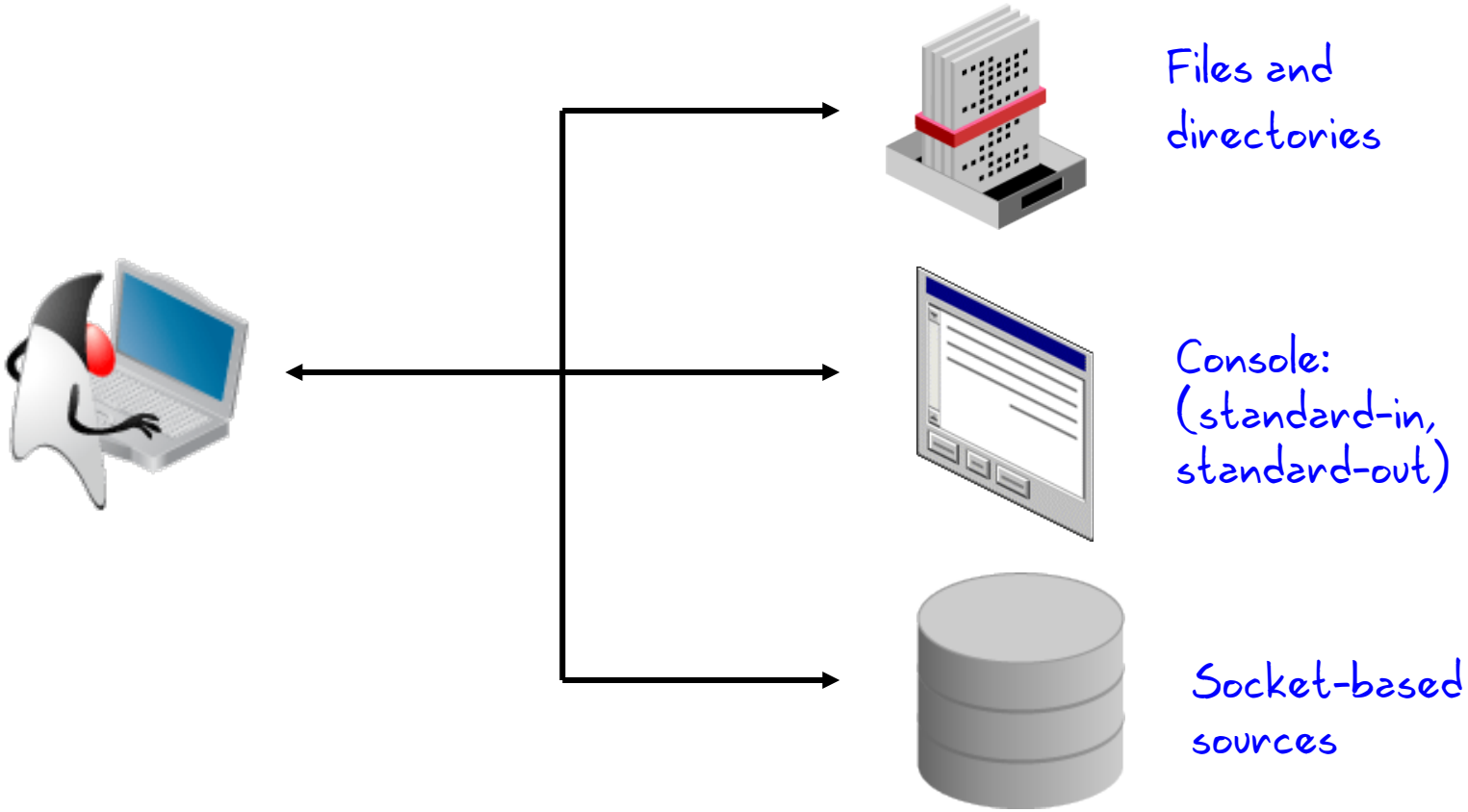
- A program uses an output stream to write data to a destination (sink), one item at time.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

I/O Application

Typically, a developer uses input and output in three ways:



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Data Within Streams

- Java technology supports two types of streams: character and byte.
- Input and output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
  - Normally, the term *stream* refers to a byte stream.
  - The terms *reader* and *writer* refer to character streams.

Stream	Byte Streams	Character Streams
Source streams	`InputStream`	`Reader`
Sink streams	`OutputStream`	`Writer`

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Byte Stream `InputStream` Methods

- The three basic `read` methods are:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close();                // Close an open stream
int available();             // Number of bytes available
long skip(long n);           // Discard n bytes from stream
```


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Byte Stream `OutputStream` Methods

- The three basic `write` methods are:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close(); // Automatically closed in try-with-resources
void flush(); // Force a write to the stream
```


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Byte Stream: Example

```
import java.io.FileInputStream; import java.io.FileOutputStream;
import java.io.FileNotFoundException; import java.io.IOException;

public class ByteStreamCopyTest {
    public static void main(String[] args) {
        byte[] b = new byte[128];
        // Example use of InputStream methods
        try (FileInputStream fis = new FileInputStream (args[0]);
             FileOutputStream fos = new FileOutputStream (args[1])) {
             System.out.println ("Bytes available: " + fis.available());
            int count = 0; int read = 0;
            while ((read = fis.read(b)) != -1) {
                 fos.write(b);
                count += read;
             }
             System.out.println ("Wrote: " + count);
         } catch (FileNotFoundException f) {
             System.out.println ("File not found: " + f);
         } catch (IOException e) {
             System.out.println ("IOException: " + e);
         }
     }
}
```

Note that you must keep track of how many bytes are read into the byte array each time.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Character Stream `Reader` Methods

- The three basic `read` methods are:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- Other methods include:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Character Stream `Writer` Methods

- The basic `write` methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Other methods include:

```
void close()
void flush()
```


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

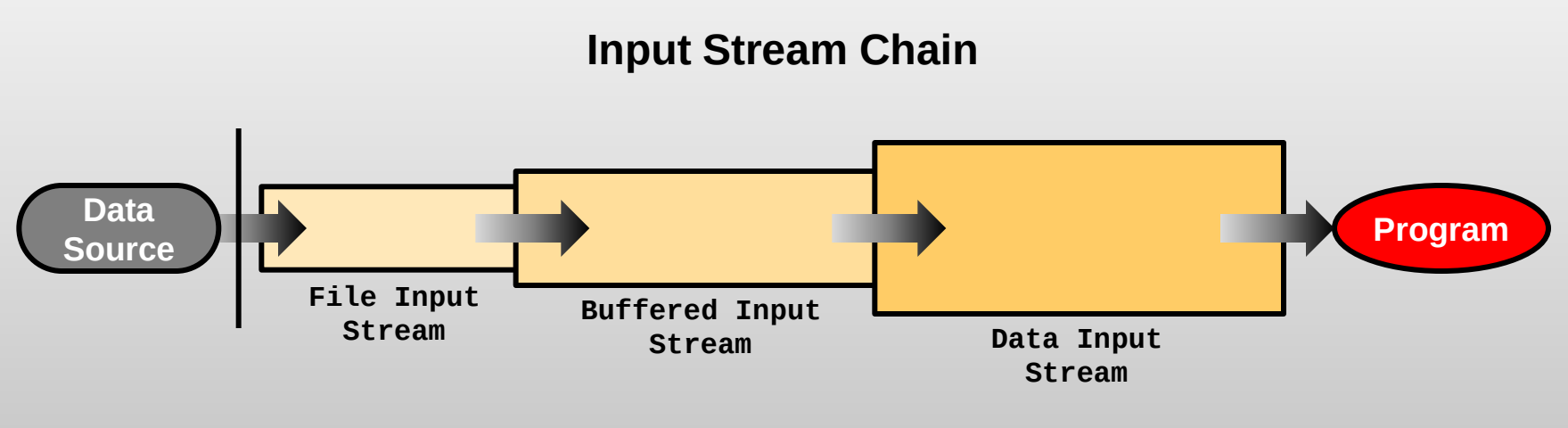
Character Stream: Example

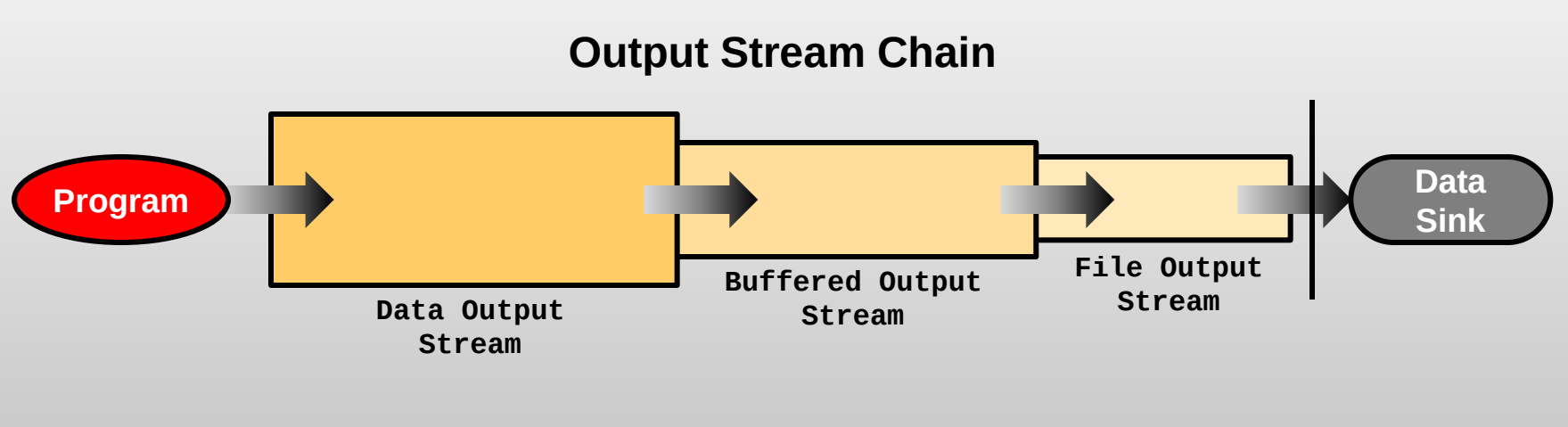
```
import java.io.FileReader; import java.io.FileWriter;
import java.io.IOException; import java.io.FileNotFoundException;

public class CharStreamCopyTest {
    public static void main(String[] args) {
        char[] c = new char[128];
        // Example use of InputStream methods
        try (FileReader fr = new FileReader(args[0]);
             FileWriter fw = new FileWriter(args[1])) {
             int count = 0;
             int read = 0;
             while ((read = fr.read(c)) != -1) {
                  fw.write(c);
                 count += read;
             }
             System.out.println("Wrote: " + count + " characters.");
         } catch (FileNotFoundException f) {
             System.out.println("File " + args[0] + " not found.");
         } catch (IOException e) {
             System.out.println("IOException: " + e);
         }
     }
}
```

Now, rather than a byte array, this version uses a character array.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

I/O Stream Chaining





Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Chained Streams: Example

```
import java.io.BufferedReader; import java.io.BufferedWriter;
import java.io.FileReader; import java.io.FileWriter;
import java.io.FileNotFoundException; import java.io.IOException;

public class BufferedStreamCopyTest {
   public static void main(String[] args) {
       try (BufferedReader bufInput
                   = new BufferedReader(new FileReader(args[0]));
            BufferedWriter bufOutput
                 = new BufferedWriter(new FileWriter(args[1]))) {
            String line = "";
            while ((line = bufInput.readLine()) != null) {
                 bufOutput.write(line);
                 bufOutput.newLine();
            }
        } catch (FileNotFoundException f) {
            System.out.println("File not found: " + f);
        } catch (IOException e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

A `FileReader` chained to a `BufferedFileReader`: This allows you to use a method that reads a String.

The character buffer replaced by a `String`. Note that `readLine()` uses the newline character as a terminator. Therefore, you must add that back to the output file.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Console I/O

The `System` class in the `java.lang` package has three static instance fields: `out`, `in`, and `err`.

- The `System.out` field is a static instance of a `PrintStream` object that enables you to write to *standard output.*
- The `System.in` field is a static instance of an `InputStream` object that enables you to read from *standard input.*
- The `System.err` field is a static instance of a `PrintStream` object that enables you to write to *standard error.*

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Writing to Standard Output

- The `println` and `print` methods are part of the `java.io.PrintStream` class.
- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- The `print` and `println` methods are overloaded for most primitive types (`boolean`, `char`, `int`, `long`, `float`, and `double`) and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reading from Standard Input

```
public class KeyboardInput {

    public static void main(String[] args) {
        String s = "";
        try (BufferedReader in = new BufferedReader(new InputStreamReader(System.in))) {
            System.out.print("Type xyz to exit: ");
            s = in.readLine();
            while (s != null) {
                System.out.println("Read: " + s.trim());
                if (s.equals("xyz")) {
                    System.exit(0);
                }
                System.out.print("Type xyz to exit: ");
                s = in.readLine();
            }
        } catch (IOException e) { // Catch any IO exceptions.
            System.out.println("Exception: " + e);
        }
    }
}
```

Chain a buffered reader to an input stream that takes the console input.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Channel I/O

Introduced in JDK 1.4, a channel reads bytes and characters in blocks, rather than one byte or character at a time.

```
import java.io.FileInputStream; import java.io.FileOutputStream;
import java.nio.channels.FileChannel; import java.nio.ByteBuffer;
import java.io.FileNotFoundException; import java.io.IOException;

public class ByteChannelCopyTest {
    public static void main(String[] args) {
        try (FileChannel fcIn = new FileInputStream(args[0]).getChannel();
             FileChannel fcOut = new FileOutputStream(args[1]).getChannel()) {
            ByteBuffer buff = ByteBuffer.allocate((int) fcIn.size());
             fcIn.read(buff);
            buff.position(0);
            fcOut.write(buff);
        } catch (FileNotFoundException f) {
            System.out.println("File not found: " + f);
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
    }
}
```

Create a buffer sized the same as the file size, and then read and write the file in a single operation.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Persistence

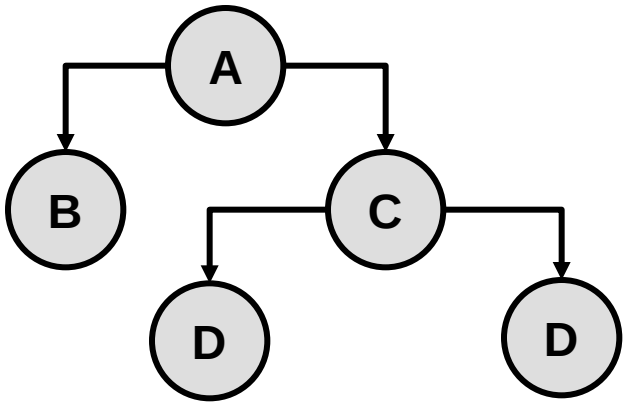
Saving data to some type of permanent storage is called persistence. An object that is persistent-capable can be stored on disk (or any other storage device), or sent to another machine to be stored there.

- A non-persisted object exists only as long as the Java Virtual Machine is running.
- Java serialization is the standard mechanism for saving an object as a sequence of bytes that can later be rebuilt into a copy of the object.
- To serialize an object of a specific class, the class must implement the `java.io.Serializable` interface.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Serialization and Object Graphs

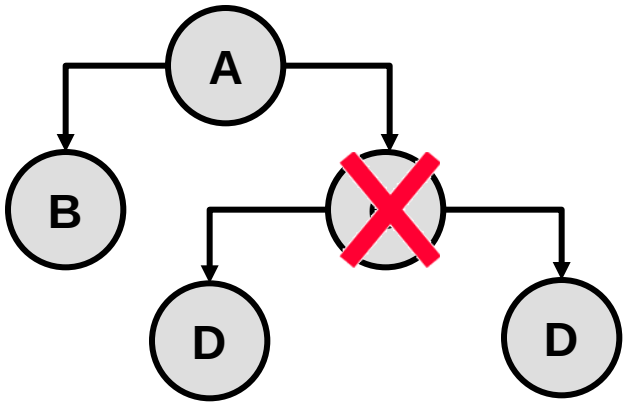
- When an object is serialized, only the fields of the object are preserved.
- When a field references an object, the fields of the referenced object are also serialized, if that object's class is also serializable.
- The tree of an object’s fields constitutes the *object graph*.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Transient Fields and Objects

- Some object classes are not serializable because they represent transient operating system–specific information.
- If the object graph contains a non-serializable reference, a `NotSerializableException` is thrown and the serialization operation fails.
- Fields that should not be serialized or that do not need to be serialized can be marked with the keyword `transient`.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Transient: Example

```
public class Portfolio implements Serializable {
    public transient FileInputStream inputFile;
    public static int BASE = 100;
    private transient int totalValue = 10;
    protected Stock[] stocks;
}
```

`static` fields are not serialized.

Serialization includes all of the members of the `stocks` array.

- The field access modifier has no effect on the data field being serialized.
- The values stored in static fields are not serialized.
- When an object is deserialized, the values of static fields are set to the values declared in the class. The value of non-static transient fields is set to the default value for the type.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Serial Version UID

- During serialization, a version number, serialVersionUID, is used to associate the serialized output with the class used in the serialization process.
- After deserialization, the serialVersionUID is checked to verify that the classes loaded are compatible with the object being deserialized.
- If the receiver of a serialized object has loaded classes for that object with different serialVersionUID, deserialization will result in an `InvalidClassException`.
- A serializable class can declare its own serialVersionUID by explicitly declaring a field named `serialVersionUID` as a static final and of type long:

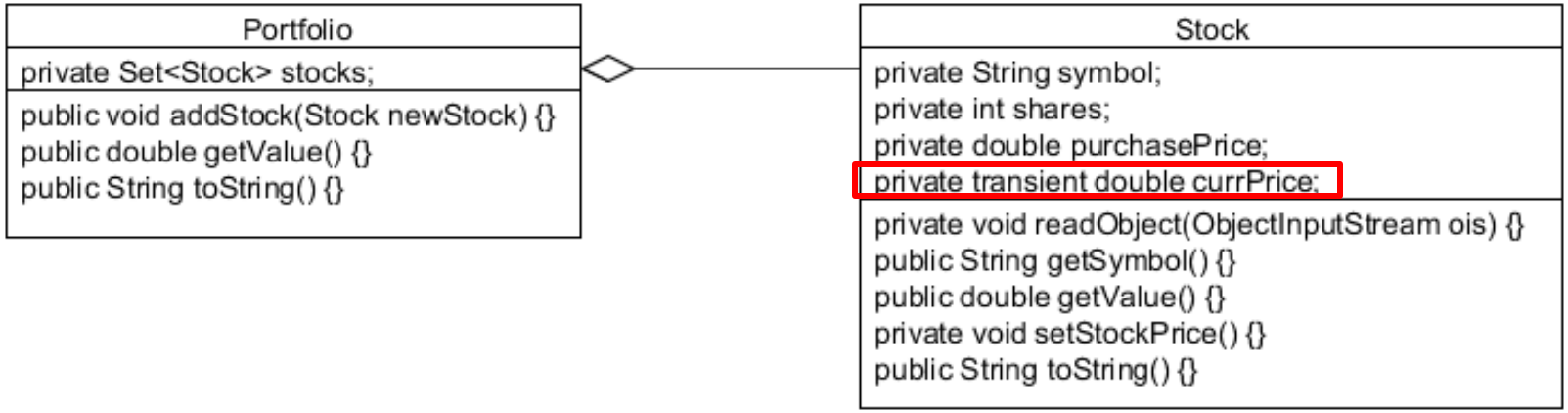
```
private static long serialVersionUID = 42L;
```


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Serialization: Example

In this example, a Portfolio is made up of a set of Stocks.

- During serialization, the current price is not serialized, and is, therefore, marked `transient`.
- However, the current value of the stock should be set to the current market price after deserialization.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Writing and Reading an Object Stream

```
public static void main(String[] args) {
    Stock s1 = new Stock("ORCL", 100, 32.50);
    Stock s2 = new Stock("APPL", 100, 245);
    Stock s3 = new Stock("GOOG", 100, 54.67);
    Portfolio p = new   Portfolio(s1, s2, s3);
    try (FileOutputStream fos = new FileOutputStream(args[0]);
         ObjectOutputStream out = new ObjectOutputStream(fos)) {
        out.writeObject(p);
    } catch (IOException i) {
         System.out.println("Exception writing out Portfolio: " + i);
    }
    try (FileInputStream fis = new FileInputStream(args[0]);
         ObjectInputStream in = new ObjectInputStream(fis)) {
        Portfolio newP = (Portfolio)in.readObject();
    } catch (ClassNotFoundException | IOException i) {
        System.out.println("Exception reading in Portfolio: " + i);
}
```

`Portfolio` is the root object.

The `writeObject` method writes the object graph of p to the file stream.

The `readObject` method restores the object from the file stream.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Serialization Methods

An object being serialized (and deserialized) can control the serialization of its own fields.

```
public class MyClass implements Serializable {
    // Fields
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        // Write/save additional fields
        oos.writeObject(new java.util.Date());
    }
}
```

`defaultWriteObject` is called to perform the serialization of this class' fields.

- For example, in this class, the current time is written into the object graph.
- During deserialization, a similar method is invoked:

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


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

readObject: Example

```
public class Stock implements Serializable {
    private static final long serialVersionUID = 100L;
    private String symbol;
    private int shares;
    private double purchasePrice;
    private transient double currPrice;

    public Stock(String symbol, int shares, double purchasePrice) {
        this.symbol = symbol;
        this.shares = shares;
        this.purchasePrice = purchasePrice;
        setStockPrice();
    }

    // This method is called post-serialization
    private void readObject(ObjectInputStream ois)
                            throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        // perform other initialization
        setStockPrice();
    }
}
```

Stock `currPrice` is set by the `setStockPrice` method during creation of the Stock object, but the constructor is not called during deserialization.

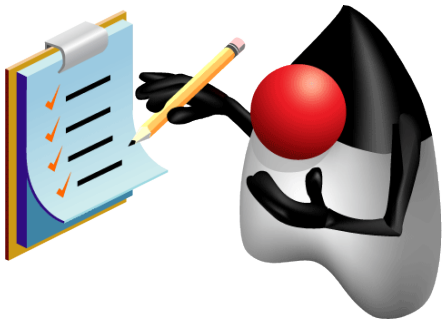
Stock `currPrice` is set after the other fields are deserialized.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have learned how to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use streams to read and write files
- Write and read objects by using serialization



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 13-1 Overview: Writing a Simple Console I/O Application

This practice covers the following topics:

- Writing a main class that accepts a file name as an argument
- Using `System` console I/O to read a search string
- Using stream chaining to use the appropriate method to search for the string in the file and report the number of occurrences
- Continuing to read from the console until an exit sequence is entered

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 13-2 Overview: Serializing and Deserializing a `ShoppingCart`

This practice covers the following topics:

- Creating an application that serializes a `ShoppingCart` object that is composed of an `ArrayList` of `Item` objects
- Using the `transient` keyword to prevent the serialization of the `ShoppingCart` total. This will allow items to vary their cost.
- Using the `writeObject` method to store today's date on the serialized stream
- Using the `readObject` method to recalculate the total cost of the cart after deserialization and print the date that the object was serialized

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

The purpose of chaining streams together is to:

a. Allow the streams to add functionality
b. Change the direction of the stream
c. Modify the access of the stream
d. Meet the requirements of JDK 7

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

To prevent the serialization of operating system–specific fields, you should mark the field:

a. `private`

b. `static`

c. `transient`

d. `final`

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Given the following fragments:

```
public MyClass implements Serializable {
    private String name;
    private static int id = 10;
    private transient String keyword;
    public MyClass(String name, String keyword) {
        this.name = name; this.keyword = keyword;
    }
}
```

```
MyClass mc = new MyClass ("Zim", "xyzzy");
```

Assuming no other changes to the data, what is the value of `name` and `keyword` fields after deserialization of the `mc` object instance?

a. `Zim, ""`

b. `Zim, null`

c. `Zim, xyzzy`

d. `"", null`

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.