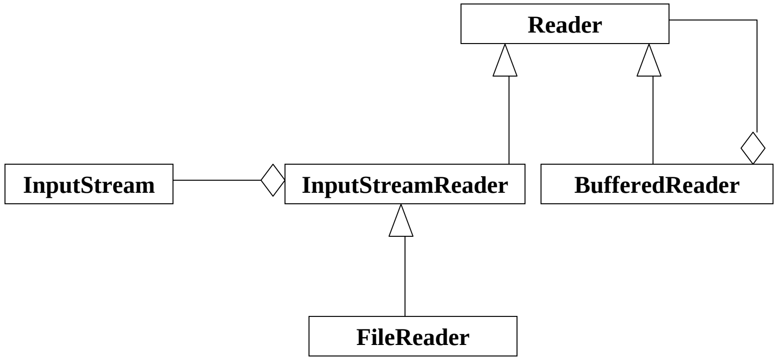
Topic Coverage for Today

1. **Character File Classes**
   - **Classes and Inheritance Hierarchy**
   - **Examples**
2. **Console IO**
   - **WordFrequency and Book Examples**
   - **StringTokernizer**
3. **More on Serialization**
   - **Serialization when classes have class instances as member variables.**
   - **Employee Example**

Inheritance Hierarchy for Character Files (Input)



Reader

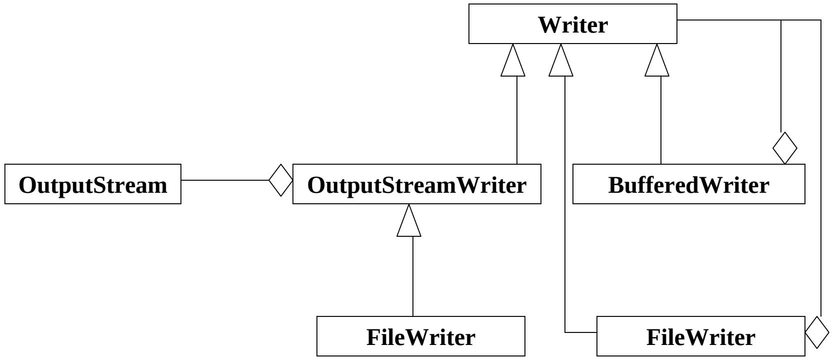
InputStream

InputStreamReader

BufferedReader

FileReader

Inheritance Hierarchy for Character Files (Output)



Writer

OutputStream

OutputStreamWriter

BufferedWriter

FileWriter

FileWriter

public abstract class Reader extends Object

Abstract class for reading character streams. The only methods that a subclass must implement are read(char[], int, int) and close(). Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

public abstract class Writer extends Object

Abstract class for writing to character streams. The only methods that a subclass must implement are write(char[], int, int), flush(), and close(). Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

public class InputStreamReader extends Reader

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and translates them into characters according to a specified character encoding. The encoding that it uses may be specified by name, or the platform’s default encoding may be accepted.

Each invocation of one of an InputStreamReader’s read() methods may cause one or more bytes to be read from the underlying byte-input stream. For top efficiency, consider wrapping an InputStreamReader within a BufferedReader; for example,

```
 BufferedReader in
   = new BufferedReader(new InputStreamReader(System.in));
```

public class FileWriter extends OutputStreamWriter

Convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream

---

public class BufferedReader extends Reader

Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

```
 BufferedReader in
   = new BufferedReader(new FileReader("foo.in"));
```

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.

```
import java.io.*;
public class WriteMatrix4 {
  static double[][] data = {
    { Math.exp(2.0), Math.exp(3.0), Math.exp(4.0) },
    { Math.exp(-2.0), Math.exp(-3.0), Math.exp(-4.0) },
  };
  public static void main(String[] args) {
    int row = data.length;
    int col = data[0].length;
    int i, j;
    for (i = 0; i < row; i++) {
      for (j = 0; j < col; j++) {
            System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
      }
    }
```

```
if (args.length > 0) {
    try {
            PrintWriter out =
               new PrintWriter(new BufferedWriter(new FileWriter(args[0])));
            out.println(row);
            out.println(col);
            for (i = 0; i < row; i++) {
              for (j = 0; j < col; j++) {
                out.println(data[i][j]);
              }
            }
            out.close();
    } catch (IOException e) {}
  }
 }
}
```

```
import java.io.*;
public class ReadMatrix4 {
  static double[][] data;
  public static void main(String[] args) {
    if (args.length > 0) {
      try { BufferedReader in =
              new BufferedReader(new FileReader(args[0]));
                String line;
                line = in.readLine();
                int row = Integer.parseInt(line);
                System.out.println("row = " + row);
                line = in.readLine();
                int col = Integer.parseInt(line);
                System.out.println("col = " + col);
                data = new double[row][col];
                for (int i = 0; i < row; i++) {
                  for (int j = 0; j < col; j++) {
                    line = in.readLine();
                    data[i][j] = Double.valueOf(line).doubleValue();
                    System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
                  }
                }
      } catch (IOException e) {}
    }
  }
}
```

Everything is input as a Character String and must be converted to the desired types.

Console IO

Console IO is a special case of character stream IO where the “file” is System.in. It is otherwise about the same as character stream IO.

The thing we need to take a look at is multiple symbols on a single line of text.

We have seen but not explained the StringTokenizer class and its use.

a) You can avoid use of StringTokernizer by always writing out your files with one symbol per line.

b) But if you need to use it here is how it works.

- **Create an instance of StringTokenizer including specification of the delimiters.**
- **Read in a line of text or a Stream of chars.**
- **Apply the operations of the StringTokenizer to extract the String which defines a symbol.**
- **Convert the String to the desired type for input.**

```
import java.util.*;
import java.io.*;
public class CountWords {
  static public void main(String[] args) {
    HashSet words = new HashSet();
    BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
    String delim = " \t\n.,:;?!-/()[]\"\’";
    String line;
    int count = 0;
    try {
      while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line, delim);
        while (st.hasMoreTokens()) {
          count++;
          words.add(st.nextToken().toLowerCase());
        }
      }
    } catch (IOException e) {}
    System.out.println("Total number of words: " + count);
    System.out.println("Number of different words: " + words.size());
  }
}
```

```
import java.util.*;
import java.io.*;
public class WordFrequency {
  static public void main(String[] args) {
    HashMap words = new HashMap();
    Count count;
    String delim = " \t\n.,:;?!-/()[]\"\’";
    try  {
   BufferedReader in = new BufferedReader(new FileReader(args[0]));
    String line, word;
      while ((line = in.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(line, delim);
      while (st.hasMoreTokens()) {
        word = st.nextToken().toLowerCase();
        count = (Count) words.get(word);
        if (count == null) {
          words.put(word, new Count(word, 1));
        } else { count.i++; }
     }
    }
   }
```

Java IO - Part 2

```
import java.io.*;
import java.util.*;
public class Book
{ private String title;
  private String author;
  private int year;
  private double price;
 public Book()
 { title = "Unknown";
   author = "Unknown";
   year = 0;
   price = 0.0;
 }
 public Book(String title, String author, int year, double price)
 { this.title = title;
   this.author = author;
   this.year = year;
   this.price = price;
 }
```

The Book class is used in the outside assignments on Collections and on JavaIO.

It is a second illustration of console IO and the use of StringTokenizer.

Java IO - Part 2

```
public static Book[] ReadConsole(int n)
/* ReadConsole invokes the read method of Book to read in an array of Book
instances and then writes the data back to the console.*/
{   Book[] books = new Book[n];
    Book book = new Book();
    InputStreamReader bookInfo = null;
    BufferedReader bookStream = null;
    bookInfo   = new InputStreamReader(System.in);
    bookStream = new BufferedReader(bookInfo);
    try {
         Book.read(bookStream,book);
        }
             catch (Exception ioe)
             {ioe.printStackTrace();
         System.exit(1);
             }
        book.write();
             books[0] = book;
             return books;
}
```

Java IO - Part 2

```
public static void read(BufferedReader bookStream, Book book)
/* This version of read expects each instance variable of Book to be on a separate line
             when input.  */
  { System.out.println(" Enter Title");
    try { book.title = bookStream.readLine(); }
    catch (IOException ioe) { book.title = "Unknown"; }
    System.out.println(" Enter Author");
    try { book.author = bookStream.readLine(); }
    catch (IOException ioe) { book.title = "Unknown"; }
    System.out.println(" Enter Year");
    try {book.year = Integer.valueOf(bookStream.readLine()).intValue(); }
    catch (IOException ioe) { System.out.println(" ioe ");book.year = 0;}
    catch (NumberFormatException nfe) {System.out.println(" nfe "); book.year = 0; }
    System.out.println(" Enter Price");
    try {book.price = Double.valueOf(bookStream.readLine()).doubleValue(); }
    catch (IOException ioe) { book.price = 0.0; }
    catch (NumberFormatException nfe) { book.price = 0.0; }
}
```

Note the type casting operations in the highlighted lines of code and the NumberFormatException.

Java IO - Part 2

```
public void write()
{ System.out.println(title);
  System.out.println(author);
  System.out.println(year);
  System.out.println(price);
}
}
```

```
public static void read(BufferedReader bookStream, Book book)
  /* Read expects to have the title on one line, the author on one line
and the year and the price as two numbers on the same line separated by a
space.*/
  { System.out.println(" Enter Title");
    try { book.title = bookStream.readLine(); }
    catch (IOException ioe) { title = "Unknown"; }
    System.out.println(" Enter Author");
    try { book.author = bookStream.readLine(); }
    catch (IOException ioe) { title = "Unknown"; }
    String yearpriceLine;
    System.out.println(" Enter Year and Price on one line");
    try { yearpriceLine = bookStream.readLine(); }
    catch (IOException ioe) { yearpriceLine = "";}
    StringTokenizer tokenizer = new StringTokenizer(yearpriceLine);
    try { book.year = Integer.valueOf(tokenizer.nextToken()).intValue(); }
    catch (NumberFormatException nfe) { year = 0; }
    try { book.price = Double.valueOf(tokenizer.nextToken()).doubleValue(); }
    catch (NumberFormatException nfe) { price = 0.0; }
}
```

Note the use of StringTokenizer to extract year and price.

```
import java.io.*;
import java.util.*;
import Book;
/* Test program for class Book and its IO methods. */
public class BookIOTest
        { public static void main(String[] args)
                {int n = 10;
                 Book[] books = new Book[n];
                 books = Book.ReadConsole(n);
                }
        }
```

Serialization when classes have instances of other classes as instance variables.

1. **Layout of object instances in storage.**
2. **Desirable properties for a serialization mechanism.**
3. **The Java serialization mechanism.**
4. **Some examples of serialization.**

An example program to demonstrate memory layout for object instances.

```
class ObjectFileTest
{  public static void main(String[] args)
      { Employee[] staff = new Employee[3];
        Employee harry = new Employee("Harry Hacker",
             35000, new Day(1989,10,1));
        Manager carl = new Manager("Carl Cracker",
             75000, new Day(1987,12,15));
        Manager tony = new Employee("Tony Tester",
             38000, new Day(1990,3,15));
         Staff[0] = harry;
         Staff[1] = carl;
         Staff[2] = tony;
         int i;
         for (i = 0; i < Staff.length; i++)
            Staff[i].raiseSalary(100);
         for (i = 0; i < Staff.length; i++)
            Staff[i].print();

       }
```

Java IO - Part 2

```
class Employee implements Serializable
{  public Employee(String n, double s, Day d)
   {  name = n;
      salary = s;
      hireDay = d;
   }
   public Employee() {}
   public void print()
   {  System.out.println(name + " " + salary
         + " " + hireYear());
   }
   public void raiseSalary(double byPercent)
   {  salary *= 1 + byPercent / 100;
   }
   public int hireYear()
   {  return hireDay.getYear();
   }
```

Java IO - Part 2

```
private String name;
   private double salary;
   private Day hireDay;
}
class Manager extends Employee
{  public Manager(String n, double s, Day d)
   {  super(n, s, d);
      secretaryName = "";
   }
   public Manager() {}
   public void raiseSalary(double byPercent)
   {  // add 1/2% bonus for every year of service
      Day today = new Day();
      double bonus = 0.5 * (today.getYear() -
             hireYear());
      super.raiseSalary(byPercent + bonus);
   }
```

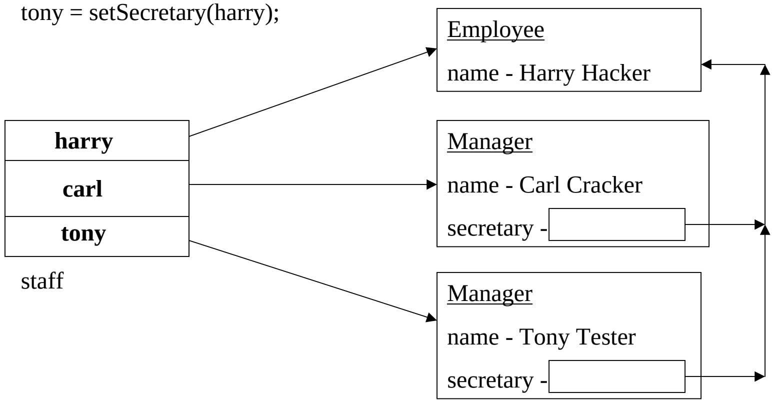
Java IO - Part 2

```
    public void setSecretaryName(String n)
    {  secretaryName = n;
    }

    public String getSecretaryName()
    {  return secretaryName;
    }

    private String secretaryName;
 }
```

Java IO - Part 2

```
harry = new Employee(“Harry Hacker”,…..);
Manager carl = new Manager(“Carl Cracker”,….);
carl.setSecretary(harry);
Manager tony = new Manager(“Tony Tester”,….);
tony = setSecretary(harry);
```

Heap memory Layout for Staff, Employee and Manager objects.



harry

carl

tony

staff

Employee

name - Harry Hacker

Manager

name - Carl Cracker

secretary -

Manager

name - Tony Tester

secretary -

Ideally the output functions of an I/O package would:

a) Allow recreation of memory layout when the data is input.

b) Store only the minimum amount of data.

Let us consider the case where we output the Employee and Manager objects using a DataOutputStream.

a) Output the Employee object

b) Output the Carl Cracker Manager object

- **output the primitive types**
- **output the instance of secretary associated with this Manager Object.**

c) Output the Tony Tester Manager object.

- **output the primitive types**
- **output the instance of secretary associated with this Manager Object.**

1. The result will be that we have made three copies of the Employee object which has a secretary relationship with the two manager objects.

2. There is no reasonable way to recreate the original memory layout when the data is input.

Java has a simple concept of Serialization which is used for objects written to an ObjectOutputStream.

1. Each object which is saved to the disk is given a serial number.

2. When an object which contains references to other objects is written to the disk a search is made to determine if the referenced objects have already been stored. If so, then a reference to the already stored copy of the object is inserted in the stored record for the containing object. If the referenced object has not yet been stored then it is given a serial number and stored.

3. When the objects are input the ObjectInputStream automatically searches for referenced objects and recreates the memory image from which the objects were stored.

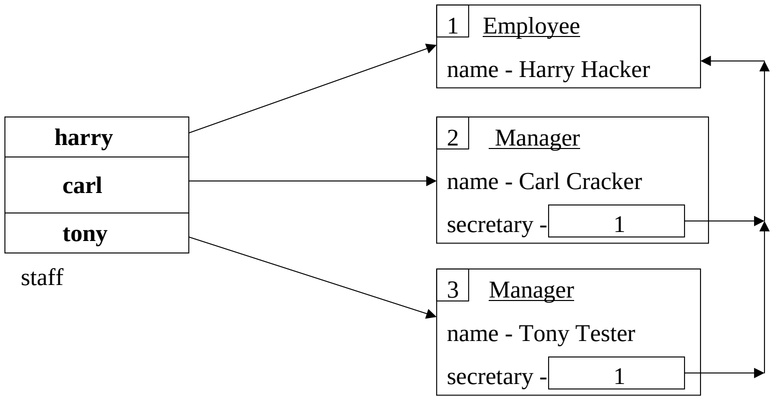
4. The objects need not be stored in any particular order.

```
class ObjectRefTest
{  public static void main(String[] args)
   {  try
      {  Employee[] staff = new Employee[3];
         Employee harry = new Employee("Harry Hacker",
             35000,new Day(1989,10,1));
         staff[0] = harry;
         staff[1] = new Manager("Carl Cracker", 75000,
            new Day(1987,12,15), harry);
         staff[2] = new Manager("Tony Tester", 38000,
            new Day(1990,3,15), harry);

         ObjectOutputStream out = new
                    ObjectOutputStream(new
            FileOutputStream("employee.dat"));
         out.writeObject(staff);
         out.close();
```

```
         ObjectInputStream in =  new
            ObjectInputStream(new
            FileInputStream("employee.dat"));
         Employee[] newStaff =
              (Employee[])in.readObject();

         for (int i = 0; i < newStaff.length; i++)
            newStaff[i].raiseSalary(100);
         for (int i = 0; i < newStaff.length; i++)
            newStaff[i].print();
      }
      catch(Exception e)
      {  e.printStackTrace();
         System.exit(1);
      }
   }
}
```

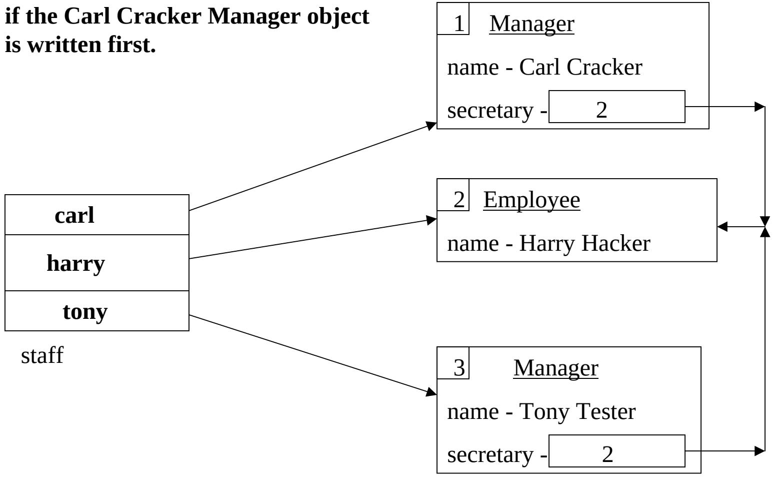
Java IO - Part 2

The disk file contents if the Employee object is stored first.



Java IO - Part 2

This is the storage map on the disk if the Carl Cracker Manager object is written first.



Java IO - Part 2

This is a simplified explanation. There are many possible complexities.

a) What happens if you change the definition of an object after writing the object instances to disk?

Mechanisms are provided for reading the object instances back using the new definition but these are beyond the scope of our needs.

b) You can override the default mechanisms provided by the Java Virtual Machine if you need to do so.

c) What happens with static or class variables? Don’t use them for now.