CN208 Introduction to Computer Programming

Lecture #11

Streams (Continued)

Pimarn Apipattanamontre

Email: ***pimarn@pimarn.com***

The Object Class

- The *Object* class is the direct or indirect superclass of every class in Java.
- There are three useful methods in the *Object* class:
  - *String toString( )*

    returns a string representation of the object.
  - *boolean equals(Object other)*

    tests whether the object equals another object.
  - *Object clone( )*

    makes a full copy of an object.

Overriding the toString( ) Method

```
BankAccount account = new BankAccount(5000);
String s = account.toString();
System.out.println(s);  // prints something like “BankAccount@d24606bf”
```

- If we call the *toString( )* method inherited from the *Object* class, all it prints is the name of the class, followed by the address of the allocated object in memory.
- To print the contents of the object, we have to override the *toString( )* method in the *BankAccount* class.

```
public class BankAccount
{  . . .
  public String toString() { return “BankAccount[balance = ” + balance + “]”; }
}
```

```
BankAccount account = new BankAccount(5000);
System.out.println(account); // prints “BankAccount[balance = 5000.0]”
```

- The *toString( )* method (when overridden) is useful for debugging the program when we want to check the current state of the object.

Overriding the toString( ) Method

```
public class BankAccount
{  . . .
  public String toString() { return “BankAccount[balance = ” + balance + “]”; }
}
public class SavingsAccount extends BankAccount
{ . . .
  public String toString()
  { return super.toString() + “[interestRate = ” + interestRate + “]”; }
}
```

```
SavingsAccount momsSavings = new SavingsAccount(5000, 2);
System.out.println(momsSavings); // prints “BankAccount[balance = 5000.0][interstRate = 2.0]”
```

Overriding the toString( ) Method

- Instead of hard-coding the class name, we can call the *getClass( )* method of the *Object* class to obtain an object of the *Class* class. Then call the *getName( )* method of the *Class* object to get the name of the class.

```
public class BankAccount
{  . . .
   public final Class getClass() { . . . }
   public String toString()
   { return    this.getClass().getName() + "[balance = " + balance + "]"; }
}
public class SavingsAccount extends BankAccount
{  . . .
    public String toString()
   { return   super.toString() + "[interestRate = " + interestRate + "]"; }
}
```

⇦ Inherited from the *Object* class

```
SavingsAccount momsSavings = new SavingsAccount(5000, 2);
System.out.println(momsSavings);// prints "SavingsAccount[balance = 5000.0][interstRate = 2.0]"
```

Overriding the equals( ) Method

- The *equals( )* method is used for comparing whether two objects have the same contents.
- Since the *Object* class has no idea about the structure of its subclasses, the *equals( )* method defined in the *Object* class is not useful.
- To make use of the *equals( )* method, we need to override it in our own defined subclasses.

```
public class BankAccount
{ . . .
  public boolean equals(Object otherObject)
  {
      BankAccount other = (BankAccount)otherObject;
      return balance == other.balance;
  }
}
```

We can assign a supertype reference to the subtype reference as long as we are sure that *otherObject* refers to a *BankAccount* object.

If we are wrong, the program will throw an exception and terminate.

```
BankAccount account = new BankAccount(1000);
BankAccount x = new BankAccount(1000);
account.equals(x)  ⟶  returns true
```

- What if *account.equals(x)* is called when *x* is not a *BankAccount* object reference?

The instanceof Operator

object instanceof ClassName

The operator returns `true` if the *object* is an instance of *ClassName* (or one of its subclasses) and `false` otherwise.

```
public class BankAccount
{ . . .
  public boolean equals(Object otherObject)
  {   if (otherObject instanceof BankAccount)
      { BankAccount other = (BankAccount)otherObject;
        return balance == other.balance;
      }
      else
        return false;
  }
}
```

```
BankAccount account = new BankAccount(1000);
```

- What if *account.equals(x)* is called when *x* is a subclass of the *BankAccount* class such as a SavingsAccount object reference?

Overriding the equals( ) Method

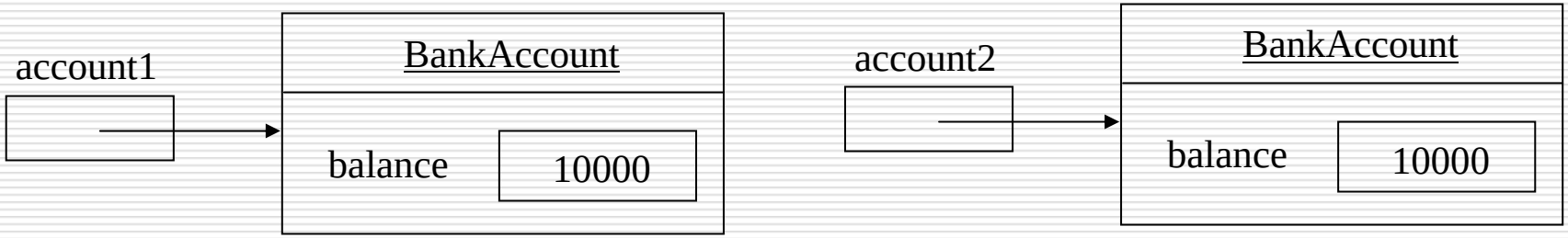
```
public class BankAccount
{ . . .
  public boolean equals(Object otherObject)
  {   if (otherObject == null) return false;
      if (getClass() != otherObject.getClass()) return false;

      BankAccount other = (BankAccount)otherObject;
      return balance == other.balance;
  }
}
```

Overriding the clone( ) Method

- The *clone( )* method makes a copy of an object. It returns a new object that has identical state to the existing object.

```
public class BankAccount
{ . . .
  public Object clone()
  {
      BankAccount clonedAccount = new BankAccount(balance);
      return clonedAccount;
  }
}
```

```
BankAccount account2 = (BankAccount) account1.clone();
```



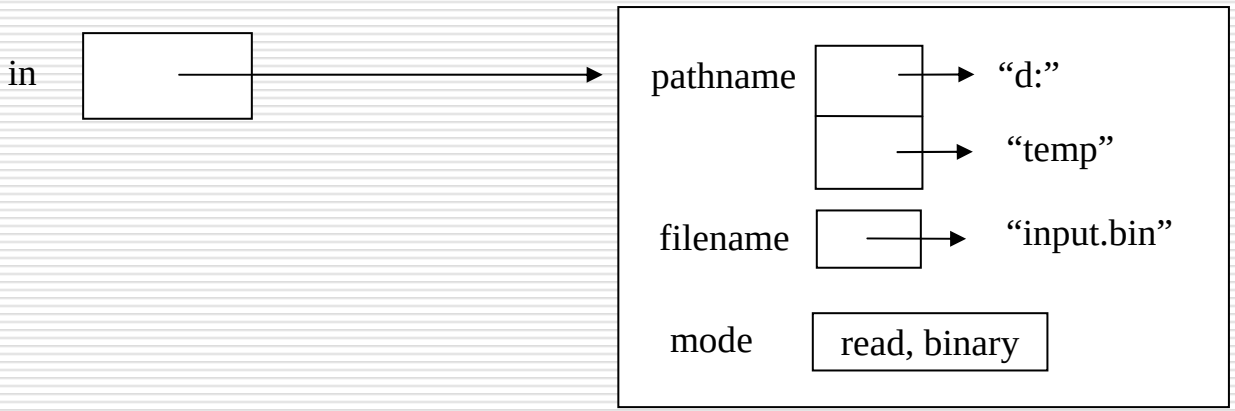
Streams

- There are two types of file formats:
  - Text File Format
    - Data in text files is represented as a sequence of characters.
  - Binary Format
    - Data in binary files is represented as a sequence of bytes and it is not in the human-readable form.
- All of the classes relating to files are defined in *java.io* package.

Read and Write Data with a Binary File

- To read data from a binary file, we create a *FileInputStream* object:

```
FileInputStream in = new FileInputStream("d:\\temp\\input.bin");
```



Read and Write Data with a Binary File

- To read data from a binary file, we create a *FileInputStream* object:

```
FileInputStream in = new FileInputStream("d:\\temp\\input.bin");
```

- To write data to a binary file, we create a *FileOutputStream* object:

```
FileOutputStream out = new FileOutputStream("d:\\temp\\output.bin");
```

- Use the *read( )* method of the *FileInputStream* class to read a byte of the file.

```
int next = in.read();
byte b;
if (next != -1)
  b = (byte) next;
```

- The *read( )* method returns an *int*, either the byte that was read or the integer -1 if the end of the input stream has been reached.
- We should test the returned value first. If it is not -1 then we can cast it to a byte.
- Use the *write( )* method of the *FileOutputStream* class to write the specified byte *b*.

```
out.write(b);
```

Read and Write Data with a Text File

- To read data from a text file, we first create a *FileReader* object:

```
FileReader  reader = new FileReader("input.txt");
```

- To write data to a text file, create a *FileWriter* object:

```
FileWriter writer = new FileWriter("output.txt");
```

- Use the *read( )* method of the *FileReader* class to read a single character from a text file.

```
int next = reader.read();
char c;
if (next != -1)
  c = (char) next;
```

- The *read( )* method returns an *int*, either the character that was read or the integer -1 if the end of the input stream has been reached.
- We should test the returned value first. If it is not -1 then we can cast it to a char.
- Use the *write( )* method of the *FileWriter* class to write the specified character *c*.

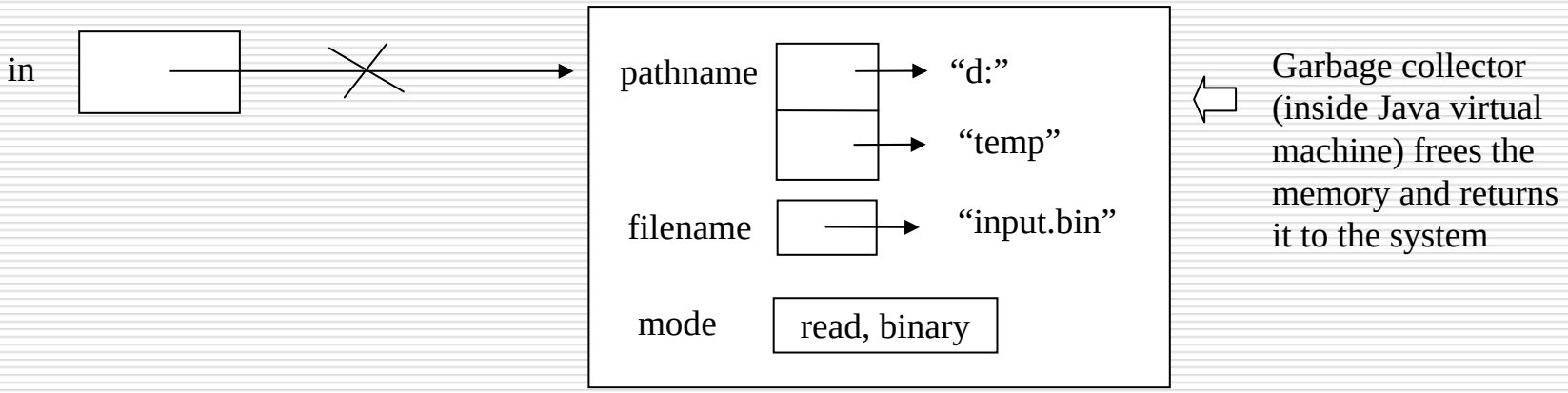
```
writer.write(c);
```

Closing Binary or Text Files

- Use the *close( )* method to close all files we no longer need.

```
in.close();
out.close();
reader.close();
writer.close();
```

- The *close( )* method releases any system resources associated with the file and make sure that all changes are committed to the disk file.



Example of File Processing

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

public class FileCopier
{ public void copyFile(String inFile,String outFile)
    throws IOException
  { FileReader in = null;
    FileWriter out = null;
    try
    { in = new FileReader(inFile);
      out = new FileWriter(outFile);
      boolean done = false;
      while (!done)
      { int next = in.read();
        if (next == -1)
          done = true;
        else
        { char c = (char) next;
          out.write(c);
        }
      }
    }
    finally
    { if (in != null) in.close();
      if (out != null) out.close();
    }
  }
}
```

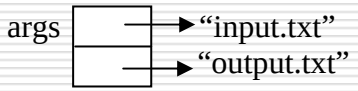
```
import java.io.IOException;

public class FileCopyTest
{
  public static void main(String[] args)
  {
    String inputFile = "input.txt";
    String outputFile = "output.txt";
    try
    {
      FileCopier fc = new FileCopier();
      fc.copyFile(inputFile, outputFile);
    }
    catch (IOException exception)
    {
      System.out.println("Error processing file");
    }
  }
}
```

Example of File Processing

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

public class FileCopier
{ public void copyFile(String inFile,String outFile)
    throws IOException
  { FileReader in = null;
    FileWriter out = null;
    try
    { in = new FileReader(inFile);
      out = new FileWriter(outFile);
      boolean done = false;
      while (!done)
      { int next = in.read();
        if (next == -1)
          done = true;
        else
        { char c = (char) next;
          out.write(c);
        }
      }
    }
    finally
    { if (in != null) in.close();
      if (out != null) out.close();
    }
  }
}
```

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

public class FileCopyTest
{ public static void main(String[] args)
  { if (args.length != 2) usage();
    try
    { FileCopier fc = new FileCopier();
      fc.copyFile(args[0], args[1]);
    }
    catch (FileNotFoundException exception)
    { System.out.println("Cannot open file");
      System.out.println(exception);
    }
    catch (IOException exception)
    { System.out.println("Error processing file");
      System.out.println(exception);
    }
  }
  public static void usage()
  { System.out.println(
      "Usage: java FileCopyTest infile outfile");
    System.exit(1);
  }
}
```

```
% javac FileCopyTest.java
% java FileCopyTest input.txt output.txt
```



Writing Text Files

- The *write( )* method of the *FileWriter* class sends output to a file one character at a time.

```
FileWriter writer = new FileWriter("output.txt");
writer.write('A');
```

- To print numbers, objects, or strings, we can construct a *PrintWriter* from a *FileWriter* object.

```
PrintWriter out = new PrintWriter(writer);
```

- Then use the *print( )* or *println( )* method to send the output to the file.

```
out.println(29.95);
out.println("Hello, World!");
BankAccount anAcc = new BankAccount(100);
out.println(anAcc);
```

- The *print( )* and *println( )* method:

  (a) Convert a number to the string representation or use the *toString( )* method to convert an object to the string.

  (b) Break up the string into individual characters and then give each character to the *FileWriter* object through its *write( )* method.

Reading Text Files

- To read data (more than a character at a time) from text files, we construct an object of the *BufferedReader* class:

```
FileReader reader = new FileReader("input.txt");
BufferedReader in = new BufferedReader(reader);
```

- Then use the *readLine( )* method to get a line of the text file at a time.

```
String inputLine = in.readLine( );
```

- The *readLine( )* method keeps calling the *read( )* method of the *FileReader* object (supplied in the constructor) until the entire input line has been read.
- The *readLine( )* method returns *null* when trying to read data after the last line.

Reading Text Files

- After a line of the input file has been read, we can use the *Integer.parseInt( )* or *Double.parseDouble( )* method to convert the string to a number.

```
double x = Double.parseDouble(inputLine);
```

- If there are several data items in a single line, use the methods of the StringTokenizer class to break up the line into multiple tokens.

```
StringTokenizer tokenizer = new StringTokenizer(inputLine);
while (tokenizer.hasMoreToken( ))
{
    String token = tokenizer.nextToken();
    double x = Double.parseDouble(token);
    ...
}
```

Object Streams

- One application of using the *FileInputStream* and *FileOutputStream* classes is object streams.
- We use the *ObjectOutputStream* class to write an entire object to a binary file.
- First, we construct an object of the *ObjectOutputStream* class from a *FileOutputStream* object.

```
BankAccount anAcc = new BankAccount(100);
FileOutputStream out = new FileOutputStream(“bankacc.dat”);
ObjectOutputStream outObj = new ObjectOutputStream(out);
```

- Then we use the *writeObject( )* method of the *ObjectOutputStream* class to write the object into the file in a binary format.

```
outObj.writeObject(anAcc);
```

- The *writeObject( )* method gives each byte of *anAcc* object to the *FileOutputStream* object through its *write( )* method.

Object Streams

- We use the *ObjectInputStream* class to read objects back from a binary file.
- First, we construct an object of the *ObjectInputStream* class from a *FileInputStream* object.

```
FileInputStream in = new FileInputStream("bankacc.dat");
ObjectInputStream inObj = new ObjectInputStream(in);
```

- Then we use the *readObject( )* method of the *ObjectInputStream* class to read an object from the binary file.
- The *readObject( )* method returns an object reference of type *Object*, we need to cast them the original class of the object and use a cast.

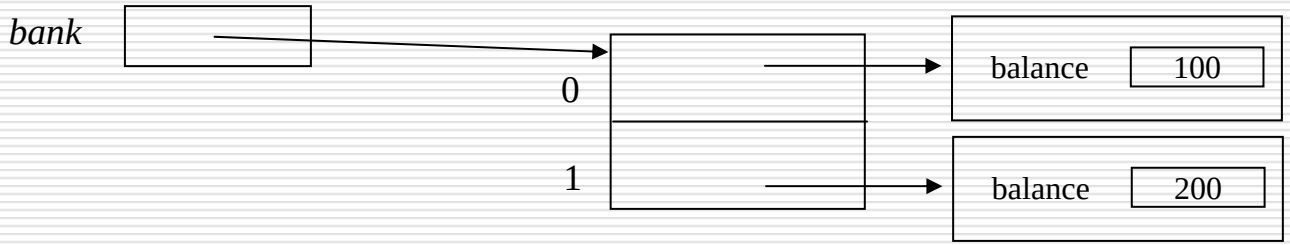
```
BankAccount anAcc = (BankAccount) inObj.readObject();
```

- The *readObject( )* method can throw two checked exceptions: *IOException* and *ClassNotFoundException*. Thus, we need to throw (by using *throws* specifier) or catch (by using *catch block*) them.

Object Streams

- We can save a sequence of objects in an array list with one instruction:

```
ArrayList bank = new ArrayList();
bank.add(new BankAccount(100));
bank.add(new BankAccount(200));
FileOutputStream out = new FileOutputStream("bank.dat");
ObjectOutputStream outObj = new ObjectOutputStream(out);
```




```
outObj.writeObject(bank);
```

- The array list and all the *BankAccount* objects that it references are saved into the file.

Object Streams

- We can also read the structure of the array list with one instruction:

```
FileInputStream in = new FileInputStream(“bank.dat”);
ObjectInputStream inObj = new ObjectInputStream(in);
```

```
ArrayList aBankAccSet = (ArrayList) inObj.readObject(); ⇦
```

Object Streams

- To save or retrieve objects of a particular class into an object stream by using the *readObject( )* and *writeObject( )* methods, that class must implement the *Serializable* interface.
- The Serializable interface (located in *java.io* package) has no method so there is no effort involved with implementing it.

```
import java.io.Serializable;
public class BankAccount implements Serializable
{
  . . .
}
```

- Why don't all classes implement Serializable interface automatically?
  - The answer is that it is for security purposes.
  - Java wants the programmers defining the class definition to open a permission allowing objects of that class read/written from/into a file EXPLICITLY.
  - The way to provide the permission is that the programmers must implement the Serializable interface.

The File Class

- An object of the *File* class stores information about file and directory pathnames in a platform independent way.
- To construct an object of the File class:

```
File inputFile = new File("C:\\datafiles\\input.txt");
```

- The File class contains the *exists( )* method to test whether the file denoted by the pathname exists in the disk.

```
inputFile.exists();
```

  tests whether the filename "*input.txt*" is currently in the "*C:\datafiles*" directory.

- We cannot use a *File* object for reading and writing. We need to pass it to a constructor of the following *FileReader*, *FileWriter*, *FileInputStream*, *FileOutputStream* classes.

```
FileReader in = new FileReader(inputFile);
int next = in.read();
```

Example of File Processing

File BankAccount.java

```
import java.io.Serializable;

public class BankAccount implements Serializable
{  // instance variable
   private double balance;
   // methods
   public void deposit(double amount)
   {  balance = balance + amount; }
   public void withdraw(double amount)
   {  if (amount < balance)
      {  IllegalArgumentException  exception
           = new IllegalArgumentException("Amount exceeds balance.");
         throw exception;
      }
      balance = balance - amount;
   }
   public double getBalance() { return balance; }
   // constructors
   public BankAccount(double initBalance) { balance = initBalance; }
   public BankAccount() { this(0); }
}
```

File BankAccountTest.java

```
import java.io.*;
import java.util.ArrayList;

public class BankAccountTest
{   public static void main(String[] args)
          throws IOException, ClassNotFoundException
    {  ArrayList aBankAccSet;
       File f = new File("bank.dat");
       if (f.exists())
       {  FileInputStream in = new FileInputStream(f);
          ObjectInputStream inObj = new ObjectInputStream(in);
          aBankAccSet = (ArrayList) inObj.readObject();
          in.close();
       }
       else aBankAccSet = new ArrayList();

       aBankAccSet.add(new BankAccount(100));
       aBankAccSet.add(new BankAccount(200));

       double sum = 0;
       for (int i=0; i<aBankAccSet.size(); i++)
       {  BankAccount anAcc = (BankAccount) aBankAccSet.get(i);
          sum = sum + anAcc.getBalance();
       }
       System.out.println("Total balance = " + sum);
       FileOutputStream out = new FileOutputStream(f);
       ObjectOutputStream outObj = new ObjectOutputStream(out);
        outObj.writeObject(aBankAccSet);
        out.close();
    }
}
```

Assume that there is no "bank.dat" file.

% java BankAccountTest

Total balance = 300.0

% java BankAccountTest

Total balance = 600.0