CIS 182 Final Exam Review – Guidelines

For all of the chapters, I have copied the chapter reviews from the Deitel website. I also took the liberty of highlighting important concepts and in some instances added code snippets to demonstrate a concept being summarized.

You may use the Final Exam Review when taking the final exam.

 Chapters 9 and 10 cover Inheritance, Polymorphism, and the use of Interfaces. You are responsible for everything except the “optional” sections at the end of the chapters.

In Chapter 11 "Exception Handling", you are responsible for sections 1 - 6, and section 9.

In chapter 14 "GUI Components: Part I". There was a lot of material and examples in this chapter. You are responsible for the following sections: 1 - 10, and section 18. I distributed several handouts in class pertaining to GUI concepts. You may use these handouts on the final exam.

In chapter 17 "Files and Streams" you are responsible for sections 1, 2, 3 (File class - information only), 4, and 5. Focus on working with binary files. Be able to create a binary stream object (input file or output file).

Chapter 9 Inheritance - Summary

- Software reuse reduces program-development time.
- The direct superclass of a subclass (specified by the keyword extends in the first line of a class declaration) is the superclass from which the subclass inherits. An indirect superclass of a subclass is two or more levels up the class hierarchy from that subclass.

```java
class HourlyEmployee extends Employee
```

- In single inheritance, a class is derived from one direct superclass. In multiple inheritance, a class is derived from more than one direct superclass. Java does not support multiple inheritance.
- A subclass is more specific than its superclass and represents a smaller group of objects.
- Every object of a subclass is also an object of that class's superclass. However, a superclass object is not an object of its class's subclasses.

  HourlyEmployee “is-a” Employee is ok. However, Employee “is-a” HourlyEmployee is NOT ok.

- An "is-a" relationship represents inheritance. In an "is-a" relationship, an object of a subclass also can be treated as an object of its superclass.
- A "has-a" relationship represents composition. In a "has-a" relationship, a class object contains references to objects of other classes.

  An example of composition would be an Employee class containing String class objects. We would say that Employee "has-a" String

- A subclass cannot access or inherit the private members of its superclass—allowing this would violate the encapsulation of the superclass. A subclass can, however, inherit the non-private members of its superclass.

  Some say that with inheritance, everything is inherited but the way you can access what you have inherited depends on the access specifier used in the declaration of the member.

  Public members are directly accessible. Private members that are “inherited” can only be accessed using a public method that was also inherited.
• A superclass method can be **overridden** in a subclass to declare an appropriate implementation for the subclass.

  **Overriding a method** is done when a method is declared in a subclass with the same method header as a method in a superclass.

• Single-inheritance relationships form tree-like hierarchical structures—a superclass exists in a hierarchical relationship with its subclasses.

• A superclass's **public** members are accessible wherever the program has a reference to an object of that superclass or one of its subclasses.

• A superclass's **private** members are accessible only within the declaration of that superclass.

• A superclass's **protected** members have an intermediate level of protection between public and private access. They can be accessed by members of the superclass, by members of its subclasses and by members of other classes in the same package.

• The first task of any subclass constructor is to call its direct superclass's constructor, either explicitly (manually) or implicitly (automatically), to ensure that the instance variables inherited from the superclass are initialized properly.

• A subclass can explicitly invoke a constructor of its superclass by using the superclass constructor call syntax—keyword **super**, followed by a set of parentheses containing the superclass constructor arguments.  
  ```java
  super(arg1, arg2, ...);
  ```

• When a subclass method overrides a superclass method, the superclass method can be accessed from the subclass if the superclass method name is preceded by the keyword **super** and a dot (.) separator.  
  ```java
  super.toString();
  ```

• Declaring instance variables **private**, while providing non-private methods to manipulate and perform validation, helps enforce good software engineering.

• Method **toString** takes no arguments and returns a String. The **Object class's toString method is normally overridden by a subclass.**

• When an object is output using the **%s** format specifier, the object's **toString** method is called implicitly to obtain its string representation.
Access to Class Members
[ From the Java Tutorial ]

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>no modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>[ package ]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members.

The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member.

The third column indicates whether subclasses of the class — declared outside this package — have access to the member.

The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use.

Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Chapter 10 Polymorphism - Summary

- With polymorphism, it is possible to design and implement systems that are more easily extensible. Programs can be written to process objects of types that may not exist when the program is under development.

- There are many situations in which it is useful to declare abstract classes for which the programmer never intends to create objects. These are used only as superclasses, so we sometimes refer to them as abstract superclasses. You cannot instantiate objects of an abstract class.

- Classes from which objects can be created are called concrete classes.

- A class must be declared abstract if one or more of its methods are abstract. An abstract method is one with the keyword abstract to the left of the return type in its declaration.

  ```java
  public abstract double getEarnings();
  ```

- If a class extends a class with an abstract method and does not provide a concrete implementation of that method, then that method remains abstract in the subclass. Consequently, the subclass is also an abstract class and must be declared abstract.

- Java enables polymorphism—the ability for objects of different classes related by inheritance or interface implementation to respond differently to the same method call.

- When a request is made through a superclass reference to a subclass object to use an abstract method, Java executes the implemented version of the method found in the subclass.
• Although we cannot instantiate objects of abstract classes, we can declare variables of abstract-class types. Such variables can be used to reference subclass objects.

  Suppose class Employee is an abstract class:  
  ```java
  public abstract class Employee
  ```
  Inside a different class, we could declare a reference variable of type Employee:
  ```java
  Employee e;
  ```

• Due to dynamic binding (also called late binding), the specific type of a subclass object need not be known at compile time for a method call of a superclass variable to be compiled. At execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

• Operator `instanceof` checks the type of the object to which its left operand refers and determines whether this type has an "is-a" relationship with the type specified as its right operand. If the two have an "is-a" relationship, the `instanceof` expression is true. If not, the `instanceof` expression is false.

  ```java
  HourlyEmployee empX;  empX instanceof HourlyEmployee  would return True.
  ```

• Every object in Java knows its own class and can access this information through method `getClass`, which all classes inherit from class `Object`. Method `getClass` returns an object of type `Class` (package `java.lang`), which contains information about the object's type that can be accessed using `Class`'s public methods. `Class` method `getName`, for example, returns the name of the class.

  Given x is an object reference variable  
  ```java
  x.getClass().getName()  would return the name of the class as a String.
  ```

• An interface declaration begins with the keyword `interface` and contains a set of `public abstract` methods. Interfaces may also contain `public static final` fields.

  ```java
  public interface Comparable
  ```

• To use an interface, a class must specify that it implements the interface and must either declare every method in the interface with the signatures specified in the interface declaration or be declared abstract.

  ```java
  public class Employee implements Comparable
  ```

• An `interface` is typically used when disparate (i.e., unrelated) classes need to provide common functionality (i.e., methods) or use common constants.

• An `interface` is often used in place of an `abstract class` when there is no default implementation to inherit.

• When a class implements an interface, it establishes an "is-a" relationship with the interface type, as do all its subclasses. To implement more than one interface, simply provide a comma-separated list of interface names after keyword `implements` in the class declaration.

  ```java
  public class Employee extends Object implements Comparable, Serializable
  ```

---

**Chapter 11 Exception Handling - Summary**

• An exception is an indication of a problem that occurs during a program's execution.

• Exception handling enables programmers to create applications that can resolve exceptions.

• Exception handling enables programmers to remove error-handling code from the "main line" of the program's execution, improving program clarity and enhancing modifiability.
• Exceptions are **thrown** when a method detects a problem and is unable to handle it.

• An exception's **stack trace** includes the name of the exception in a descriptive message that indicates the problem that occurred and the complete method-call stack (i.e., the call chain) at the time the exception occurred.

• The point in the program at which an exception occurs is called the **throw point**.

• A **try block** encloses the code that might throw an exception and the code that should not execute if that exception occurs.

• Exceptions may surface through:
  o **explicitly** mentioned code in a try block,
  o through calls to other methods or even
  o through deeply nested method calls initiated by code in the try block.

• A **catch block** begins with the keyword catch and an exception parameter followed by a block of code that catches (i.e., receives) and handles the exception. This code executes when the try block detects the exception.

• An **uncaught exception** is an exception that occurs for which there are no matching catch blocks.

• An uncaught exception will cause a program to terminate early if that program contains only one thread. If the program contains more than one thread, only the thread where the exception occurred will terminate. The rest of the program will run, but may yield adverse effects.

• **At least one catch block or a finally block must immediately follow the try block.**

• Each catch block specifies in parentheses an **exception parameter** that identifies the exception type the handler can process. The exception parameter's name enables the catch block to interact with a caught exception object.

• If an exception occurs in a try block, the try block terminates immediately and program control transfers to the first of the following catch blocks whose exception parameter type matches the type of the thrown exception.

• After an exception is handled, **program control does not return to the throw point because the try block has expired**. This is known as the termination model of exception handling.

• If there are multiple matching catch blocks when an exception occurs, only the first is executed.

• After executing a catch block, this program's flow of control proceeds to the first statement after the last catch block.

• A **throws clause** specifies the exceptions the method throws, and appears after the method's parameter list and before the method body.

  ```java
  public void readRecords( ) throws FileNotFoundException
  ```

• The throws clause contains a **comma-separated list** of exceptions that the method will throw if a problem occurs when the method executes.

• Exception handling is designed to process synchronous errors, which occur when a statement executes.

• Exception handling is not designed to process problems associated with asynchronous events, which occur in parallel with, and independent of, the program's flow of control.
All Java exception classes inherit, either directly or indirectly, from class Exception. Because of this fact, Java's exception classes form a hierarchy. Programmers can extend this hierarchy to create their own exception classes.

Class Throwable is the superclass of class Exception, and is therefore also the superclass of all exceptions. Only Throwable objects can be used with the exception-handling mechanism.

Class Throwable has two subclasses: Exception and Error.

Class Exception and its subclasses represent exceptional situations that could occur in a Java program and be caught by the application.

Class Error and its subclasses represent exceptional situations that could happen in the Java runtime system. Errors happen infrequently, and typically should not be caught by an application.

Java distinguishes between two categories of exceptions: checked and unchecked.

Unlike checked exceptions, the Java compiler does not check the code to determine whether an unchecked exception is caught or declared. Unchecked exceptions typically can be prevented by proper coding.

An exception's type determines whether the exception is checked or unchecked.

All exception types that are direct or indirect subclasses of class RuntimeException are unchecked exceptions.

All exception types that inherit from class Exception but not from RuntimeException are checked.

Various exception classes can be derived from a common superclass.

If a catch block is written to catch exception objects of a superclass type, it can also catch all objects of that class's subclasses. This allows for polymorphic processing of related exceptions.

Programs that obtain certain types of resources must return them to the system explicitly to avoid so-called resource leaks. Resource-release code typically is placed in a finally block.

The finally block is optional. If it is present, it is placed after the last catch block.

Java guarantees that a provided finally block will execute whether or not an exception is thrown in the corresponding try block or any of its corresponding catch blocks. Java also guarantees that a finally block executes if a try block exits by using a return, break or continue statement.

If an exception that occurs in the try block cannot be caught by one of that try block's associated catch handlers, the program skips the rest of the try block and control proceeds to the finally block, which releases the resource. Then the program passes to the next outer try block—normally in the calling method.

If a catch block throws an exception, the finally block still executes. Then the exception is passed to the next outer try block—normally in the calling method.

Programmers can throw exceptions by using the throw statement.

throw new Exception( );

A throw statement specifies an object to be thrown. The operand of a throw can be of any class derived from class Throwable.

A new exception class must extend an existing exception class to ensure that the class can be used with the exception-handling mechanism.
A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an application. A GUI gives an application a distinctive "look" and "feel."

Providing different applications with consistent, intuitive user interface components allows users to be somewhat familiar with an application, so that they can learn it more quickly.

GUIs are built from GUI components—sometimes called controls or widgets.

Most applications use windows or dialog boxes (also called dialogs) to interact with the user.

Class JOptionPane (package javax.swing) provides prepackaged dialog boxes for both input and output. JOptionPane static method showInputDialog displays an input dialog.

```java
String response = JOptionPane.showInputDialog();
```

A prompt typically uses sentence-style capitalization—a style that capitalizes only the first letter of the first word in the text unless the word is a proper noun.

An input dialog can only input Strings. This is typical of most GUI components.

Class JOptionPane static method showMessageDialog displays a message dialog.

```java
JOptionPane.showMessageDialog();
```

Most Swing GUI components are located in package javax.swing. They are part of the Java Foundation Classes (JFC)—Java's libraries for cross-platform GUI development.

Together, the appearance and the way in which the user interacts with the application are known as that application's look-and-feel. Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform's custom look-and-feel.

Lightweight Swing components are not tied to actual GUI components supported by the underlying platform on which an application executes.

Several Swing components are heavyweight components that require direct interaction with the local windowing system, which may restrict their appearance and functionality.

Class Component (package java.awt) declares many of the attributes and behaviors common to the GUI components in packages java.awt and javax.swing.

Class Container (package java.awt) is a subclass of Component. Components are attached to Containers so the Components can be organized and displayed on the screen. Also a Container "is-a" Component.

Class JComponent (package javax.swing) is a subclass of Container. JComponent is the superclass of all lightweight Swing components and declares their common attributes and behaviors.

Some common JComponent features include a pluggable look-and-feel, shortcut keys called mnemonics, tool tips, support for assistive technologies and support for user interface localization.

Most windows are instances of class JFrame or a subclass of JFrame. JFrame provides the basic attributes and behaviors of a window.

A JLabel displays a single line of read-only text, an image, or both text and an image. Text in a JLabel normally uses sentence-style capitalization.
When building a GUI, each GUI component must be added to a container, such as a window created with a JFrame.

Many IDEs provide GUI design tools in which you can specify the exact size and location of a component by using the mouse, then the IDE will generate the GUI code for you.

This would be *explicitly* "laying out" an object rather than allowing this to be done using a layout manager.

JComponent method `setToolTipText` specifies the tooltip that is displayed when the user positions the mouse cursor over a lightweight component.

Container method `add` inserts a GUI component into a Container.

Class `ImageIcon` (package `javax.swing`) supports several image formats, including Graphics Interchange Format (GIF), Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG).

Interface `SwingConstants` (package `javax.swing`) declares a set of common integer constants that are used with many Swing components.

GUls are *event driven*—when the user interacts with a GUI component, events drive the program to perform tasks.

The code that performs a task in response to an event is called an *event handler* and the overall process of responding to events is known as event handling.

Class `JTextField` extends class `JTextComponent` (package `javax.swing.text`), which provides many features common to Swing’s text-based components. Class `JPasswordField` extends `JTextField` and adds several methods that are specific to processing passwords.

A component receives the *focus* when the user clicks the component.

Before an application can respond to an event for a particular GUI component, you must perform several coding steps:

1. Create a class that represents the event handler.
   ```java
   private class ButtonHandler
   ```

2. Implement an appropriate interface, known as an *event-listener interface*, in the class from *Step 1*.
   ```java
   private class ButtonHandler implements ActionListener
   ```

3. Indicate that an object of the class from *Steps 1 and 2* should be notified when the event occurs. This is known as *registering the event handler*.
   ```java
   buttonCalc.addActionListener( new ButtonHandler() );
   ```

Java allows you to declare *nested classes* inside other classes. Nested classes can be *static* or *non-static*.

*Non-static nested classes* are called *inner classes* and are *frequently used for event handling*.

An inner-class object is allowed to directly access all the instance variables and methods of its top-level class.
When the user presses Enter in a JTextField or JPasswordField, the GUI component generates an ActionEvent (package java.awt.event). Such an event is processed by an object that implements the interface ActionListener (package java.awt.event).

JTextField method addActionListener registers the event handler for a component text field. This method receives as its argument an ActionListener object.

The GUI component with which the user interacts is the event source.

An ActionEvent object contains information about the event that just occurred, such as the event source (a reference) and the text in the text field.

ActionEvent method getSource returns a reference to the event source. ActionEvent method getActionCommand returns the text the user typed in a text field or the label on a JButton.

JPasswordField method getPassword returns the password the user typed.

For each event-object type, there is typically a corresponding event-listener interface. Each event-listener interface specifies one or more event-handling methods that must be declared in the class that implements the interface.

When an event occurs, the GUI component with which the user interacted notifies its registered listeners by calling each listener’s appropriate event-handling method.

When an event occurs, the event is dispatched only to the event listeners of the appropriate type. The GUI component receives a unique event ID specifying the event type, which it uses to decide the listener type to which the event should be dispatched and which method to call on each listener object.

Command buttons are created with class JButton.

A JButton can display an Icon. To provide the user with an extra level of visual interaction with the GUI, a JButton can also have a rollover Icon—an Icon that is displayed when the user positions the mouse over the button.

The Swing GUI components contain three types of state buttons—JToggleButton, JCheckBox and JRadioButton.

Classes JCheckBox and JRadioButton are subclasses of JToggleButton.

A JRadioButton is different from a JCheckBox in that normally several JRadioButtons are grouped together, and only one in the group can be selected at any time.

Method setFont (of class Component) sets the font of a component to a new object of class Font (package java.awt).

When the user clicks a JCheckBox, an ItemEvent occurs. This event can be handled by an ItemListener object, which must implement method itemStateChanged. Method addItemListener registers the listener for a JCheckBox or JRadioButton object.

checkBoxBold.addItemListener( new CheckBoxHandler() );

JCheckBox method isSelected determines if a JCheckBox is selected.

JRadioButtons are similar to JCheckboxes in that they have two states—selected and not selected. However, radio buttons normally appear as a group in which only one button can be selected at a time. Selecting a different radio button forces all others to be deselected.

JRadioButtons are used to represent mutually exclusive options.
• The logical relationship between JRadioButtons is maintained by a ButtonGroup object (package javax.swing).

• ButtonGroup method add associates each a JRadioButton with a ButtonGroup. If more than one selected JRadioButton object is added to a group, the selected one that was added first will be selected when the GUI is displayed.

• JRadioButtons generate ItemEvents when they are clicked.

• Many event-listener interfaces contain multiple methods. For many of these interfaces, packages java.awt.event and javax.swing.event provide event-listener adapter classes.

• An adapter class implements an interface and provides a default implementation of each method in the interface (an empty method body... { }).

• You can extend an adapter class to inherit the default implementation of every method and subsequently override only the method(s) you need for event handling.

• Class Graphics is used to draw.

• Layout managers arrange GUI components in a container for presentation purposes.

• All layout managers implement the interface LayoutManager (package java.awt).

• Container method setLayout specifies the layout of a container.

```
setLayout( new FlowLayout() );
```

• FlowLayout is the simplest layout manager. GUI components are placed on a container from left to right in the order in which they are added to the container. When the edge of the container is reached, components continue to display on the next line. Class FlowLayout allows GUI components to be left aligned, centered (the default) and right aligned.

• The BorderLayout layout manager (the default for a JFrame) arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER. NORTH corresponds to the top of the container.

• A BorderLayout limits a Container to containing at most five components—one in each region. Remember, a container "is-a" component.

• The GridLayout layout manager divides the container into a grid so that components can be placed in rows and columns.

A sample test harness class

```
// Fig. 14.20: RadioButtonTest.java
// Testing RadioButtonFrame.

import javax.swing.JFrame;

public class RadioButtonTest
{
    public static void main( String args[] )
    { 
        RadioButtonFrame rbFrame = new RadioButtonFrame();
        rbFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        rbFrame.setSize( 300, 100 ); // set frame size
        rbFrame.setVisible( true ); // display frame
    }
}
```
Chapter 17 Files and Streams - Summary

- Data stored in variables and arrays is temporary—the data is lost when a local variable goes out of scope or when the program terminates. **Computers use files for long-term retention of large amounts of data**, even after the programs that created the data terminate.

- Persistent data maintained in files exists beyond the duration of program execution.

- Computers store files on secondary storage devices such as hard disks.

- The smallest data item in a computer can assume the value 0 or the value 1 and is called a bit. Ultimately, a computer processes all data items as combinations of zeros and ones.

- The computer's character set is the set of all characters used to write programs and represent data.

- **Characters in Java are Unicode characters composed of two bytes**, each composed of eight bits.

- Just as characters are composed of bits, fields are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning.

- Data items processed by computers form a data hierarchy that becomes larger and more complex in structure as we progress from bits to characters to fields, and so on.

- Typically, several fields compose a record (implemented as a class in Java).

- A record is a group of related fields.

- A file is a group of related records.

- To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record key. A record key identifies a record as belonging to a particular person or entity and is unique to each record.

- There are many ways to organize records in a file. The most common is called a sequential file, in which records are stored in order by the record-key field.

- A group of related files is often called a database. A collection of programs designed to create and manage databases is called a database management system (DBMS).

- Java views each file as a sequential stream of bytes.

- Every operating system provides a mechanism to determine the end of a file, such as an end-of-file marker or a count of the total bytes in the file that is recorded in a system-maintained administrative data structure.

- Byte-based streams represent data in **binary format**.

- Character-based streams represent data as **sequences of characters**.

- Files that are created using byte-based streams are **binary files**.

- Files created using character-based streams are **text files**. Text files can be read by text editors, whereas binary files are read by a program that converts the data to a human-readable format.

- Java also can associate streams with different devices. Three stream objects are associated with devices when a Java program begins executing—System.in, System.out and System.err.

- The java.io package includes definitions for stream classes, such as FileInputStream (for byte-based input from a file), FileOutputStream (for byte-based output to a file), FileReader (for character-based input from a file) and FileWriter (for character-based output to a file). Files are opened by creating objects of these stream classes.
• Class **File** is used to obtain information about files and directories and to construct File objects.

  A Scanner object is "wrapped" around a File object to allow input from a text file
  
  ```java
  private Scanner input = new Scanner( new File( "MyData.txt" );
  ```

• **Character-based input and output** can be performed with classes Scanner and Formatter (or my preference PrintWriter).

• **Class Formatter** enables formatted data to be output to the screen or to a file in a manner similar to System.out.printf but the method is called format().

• A separator character (`\`\`) is used to separate directories and files in the path.

• **Java imposes no structure on a file**—notions such as a record do not exist as part of the Java language. The programmer must structure files to meet an application's requirements.

• To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired information is found.

• Data in many sequential files cannot be modified without the risk of destroying other data in the file. Therefore, records in a sequential-access file are not usually updated in place. Instead, the entire file is usually rewritten.

• Java provides a mechanism called **object serialization** that enables entire objects to be written to or read from a binary stream.

• **Only classes that implement interface Serializable** can be serialized and deserialized with ObjectOutputStream and ObjectInputStream.

• For any class you wish to serialize (save in binary) must implement the tagging interface `Serializable`. There is nothing to implement but including implements `Serializable` "tags" the class as one which may be used to create objects that will be saved in binary form.

  ```java
  public class AccountRecord implements Serializable{
  ```

• A serialized object is an object represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

• After a **serialized** object has been written into a file, it can be read from the file and **deserialized**—that is, the type information and bytes that represent the object and its data can be used to re-create the object in memory.

• Classes **ObjectInputStream** and **ObjectOutputStream**, which respectively implement the `ObjectInput` and `ObjectOutput` interfaces, enable entire objects to be read from or written to a stream (possibly a file).

• To declare a reference to a binary file you want to **write** to you must "wrap" an ObjectOutputStream around a FileOutputStream.

  ```java
  private ObjectOutputStream output;
  output = new ObjectOutputStream( new FileOutputStream(“Account_Data.ser”) );
  ```

• To declare a reference to a binary file you want to **read** from you must "wrap" an ObjectInputStream around a FileInputStream. This is similar to “wrapping” a Scanner object around a File object.

  ```java
  private ObjectInputStream input;
  input = new ObjectInputStream( new FileInputStream(“Account_Data.ser”) );
  ```

• The **ObjectOutput** interface contains method **writeObject**, which takes an Object that implements interface Serializable as an argument and writes its information to an OutputStream.

  ```java
  AccountRecordSerializable record;
  record = new AccountRecordSerializable( … );
  output.writeObject( record ); //Write a serialized object to the file
  ```
The **ObjectInput** interface contains method **readObject**, which reads and returns a reference to an Object from an InputStream. After an object has been read, its reference can be cast to the object's actual type.

**Note:** When you read a serialized object it is treated as type **Object**. You must cast it to the correct class type before it can be used.

```java
record = ( AccountRecordSerializable ) input.readObject();
```