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 – 7 (only through the finally block in section 7).

In chapter 17 "Files and Streams" you are responsible for sections 1, 2, 4, and 5. Focus on working with binary files. Be able to create a binary stream object (ObjectInputStream and ObjectOutputStream) and use the readObject and writeObject methods.

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. This is why you can create a superclass reference variable and actually store a reference to a subclass object (covered in detail in chapter 10).

- 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 with a String class reference variable storing a reference to a String. 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 but a modified method body.

- 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. `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. `super.toString( ), super.earnings( )`
- 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.

**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. `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. The keywords `public` and `abstract` are not required in the method header as Java will make these methods implicitly public abstract. Interfaces may also contain `public static final` fields. The keywords `public static final` may be omitted from the field declaration because Java will implicitly declare fields in an interface to be `public static final`.

```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.

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:

- explicitly mentioned code in a try block,
- through calls to other methods or even
- 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.
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.

- A record is a group of related fields.

- A file is a group of related records.

- 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) Files are opened by creating objects of these stream classes.

- A Scanner object is “wrapped” around a File object to allow input from a text file
  
  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).

- 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.
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`.

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 AccountRecordSerializable 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` 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();
```