Intermediate Software Development

Essay One

Define APIE by describing each of the acronym's letters in detail as well as their relationships to one another. Provide details of how these casting techniques are used with the terms superclasses, subclasses, abstract classes and interfaces. Why are these techniques used? What does the keyword final have to do with inheritance? Describe static and dynamic polymorphism for object-oriented programming languages. Why is each important? Describe upcasting and downcasting. What is the difference between 'overloading' and overriding' OO methods? How does upcasting and downcasting relate to polymorphism? How does an object-oriented programming language differ from a procedural/imperative programming language? What can be the advantage of using an OOP language over a procedural language for application design?

Answer:

APIE – Abstraction, Polymorphism, Inheritance and Encapsulation.

Abstraction:

Abstraction is referred or considered as an extremely powerful feature of OOP which is used to make a class abstract. The concept can be applied to classes or methods. Abstraction can be achieved using Interface and abstract class. Abstract classes are used when we have to declare class that provide part of implementation where extended classes of this will provide implementation for some or all the methods declared in the class. When we declare a class as abstract, all methods declared but not implemented in the class should be made abstract. Similarly, when a method is made abstract then the class which contains that method must be abstract. We have to use ***abstract*** keyword to declare a class or method as abstract.

When we declare a class as an abstract, we cannot create the instance of the class i.e. it cannot be instantiated and it serves as super class. This super class is used to derive other sub classes (also called concrete classes) which can be instantiated at runtime. If a method is declared as abstract in super class then sub class must provide an implementation for that method. Abstraction is normally used when we know something needs to be there but not sure how exactly it should look like. For e.g. when I am creating a class called Animals, there should be methods like *breed*() and *noOfLegs*() but don't know breed and noOfLegs of every animals since they could have different breed and noOfLegs eg some can be retriever breed and may have 4 legs whereas some others can be of eider breed and may have 2 legs. So implementation of those breed () and noOfLegs () methods should be left to there concrete implementation e.g. Dog , Bird , Cat etc. Constructors and static methods cannot be made abstract as constructors cannot be inherited and static methods cannot be overridden in sub class.

Interface is another way of providing abstraction. Interface by default is an abstract class, and can implement any number of abstract methods. Since the interface class are abstract it cannot be instantiated and do not provides implementation for the methods. The class which implements interface should provide implementation for all abstract methods declared in it. If the class that implements interface is an abstract super

class and no implementation is provided then all derived sub classes must provide an implementation for each of the interface methods.

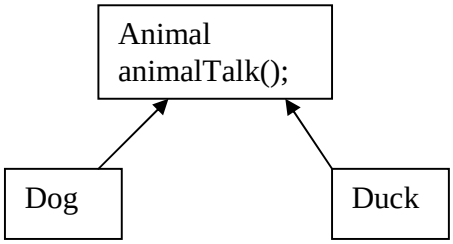
The main difference between Interface and Abstract class are:

Interface	Abstract class
An interface can only provide the definition of the method not implementation or behavior	An abstract class can provide complete code or just the methods that have to be overridden
An interface cannot have access modifiers, By default it is public.	An abstract class can contain access modifiers.
For execution, It takes more time to find the actual method in the respective class	Execution is Fast.
If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method	If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.
No fields can be defined in interfaces	An abstract class can have fields and constraints defined
A class may inherit several interfaces.	A class may inherit only one abstract class.
An Interface can be used for objects of different types.	An abstract class is used for objects of the same type.

Finally, If we know only name of the methods that has common behavior without specifying any details about how that behavior works, then we have to use *interface* for the implementation and if we know some of the behavior while designing class and if that would remain common across all subclasses then we have to use *abstract* class.

Polymorphism:

Polymorphism is another important OOP concept and widely used in Java. Polymorphism in java is supported along with other concept like Abstraction, Encapsulation and Inheritance. Polymorphism is a mechanism that is often summarized as one interface and multiple implementations that means the ability of the object belonging to different types with the specific behavior of each type. By using polymorphism, one object can be treated like another and in this way we can also create and define multiple level of interface. For eg. If I consider Animal super class which have *animalTalk*() method but I don't know what would be the sound of the animal as different animal can make different sounds such as dog barks and birds quack. Thus by creating Dog and Duck sub class of animal super class I can inherit animalTalk() method, which makes different sounds depending on the type of animal. The same can be represented diagrammatically as follows which depicts that the class Animal has sub classes Dog and Duck Which inherits animalTalk () method and represents different sounds:



Polymorphism in Java has been divided into two types: Compile time polymorphism (**static binding**) and Runtime polymorphism **(dynamic binding**). Static polymorphism is also known as method overloading, which dynamic polymorphism also known as method overriding.

1. ***Static Polymorphism*** :

Static Polymorphism or method overloading means there are several methods present in a class having the same name but has different parameters. At compile time, Java knows which method to invoke by checking the method signatures. Based on the parameters the method to be invoked is decided at compile time it is also called compile time polymorphism.

The importances of static polymorphism are:

- □ The cost of resolving the symbol to an address is done once- at compile time.
- □ Since everything about the program is known at compile-time, the compiler is able to do some very sophisticated optimizations.
- □ Tools can know everything about the code, making it easier for tool builders to provide very nice features like code completion and static analysis.

2. ***Dynamic Polymorphism***:

Consider a sub class overrides a particular method of the super class i.e. in the program we create an object of the subclass and assign it to the super class reference. Now, if we call the overridden method on the super class reference then the sub class version of the method will be called. Since the method to be executed is decided at run time, it is also called run time polymorphism.

The importances of dynamic polymorphism are:

- □ The cost of compiling is much lower, allowing for much faster development.
- □ The runtime linker will know more about the current state of the program, it can reorder the memory and resources more easily.
- □ Allows sharing of common resources rather than duplicating it in each executable.

Between static and dynamic polymorphism if we have to consider ***which is important*** it depends, since when the best possible performance is essential static binding is the best way to use. If the developer's time is more important than the computer then we have to use dynamic polymorphism

Difference between Overloading and Overriding' OO methods is as follows:

Overloading	**Overriding**
Occurs during compile time.	Occurs during run time.
Overloading of methods occurs in same class.	Overriding of methods can only be done in sub class i.e. happens between super class

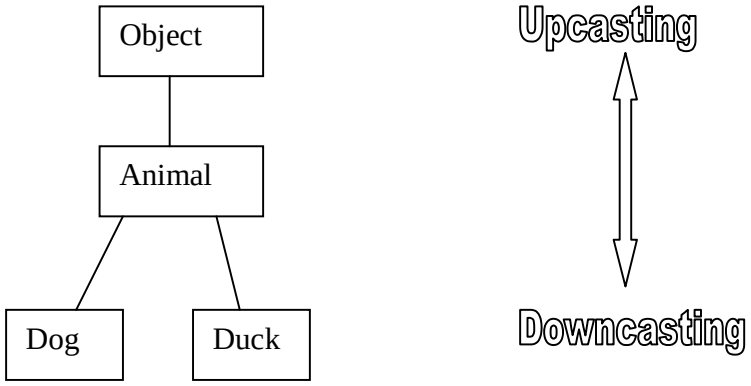
	and sub class.
Static, private and final methods can be overloaded.	Static, private and final methods cannot be overridden.
Overloaded methods are bonded using static binding i.e. type of reference variable used.	Overridden methods are bonded using dynamic bonding i.e. based upon actual Object.
Signatures i.e. parameters of the overloaded methods varies.	Signatures remain same.
Method can have any return type	Method return type must be same as super class method

□ **Upcasting and Downcasting:**

Upcasting is when an object of a derived class is casted to a variable of a base class (or any ancestor class). If the cast is invalid then ClassCastException exception is thrown by JVM. Casts, depending on how they are used, can help or hinder code flexibility. It is usually helpful to *upcast* - to cast from a subtype to a supertype and is done automatically.

Downcasting is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class). In contrast to upcasting, downcasting can be more troublesome to flexibility. We should try to use and make dynamic binding work for polymorphism instead of doing explicit downcasts. In order for dynamic casting to work the object needs to be polymorphic. Downcasts often are used in combination with the instanceof operator. instanceof is a runtime class information. When the JVM executes an instanceof operation, it consults the runtime class information associated with an object reference to determine whether the object is or is not an instance of the specified class.

Upcasting and Downcasting can be diagrammatically represented as shown below:



From the figure it clearly shows that, Duck and Dog sub class extends from Animal super class and in turn extends from Object.

Inheritance:
Inheritance is a compile-time mechanism in Java that allows extending a class (called the base class or superclass) with another class (called the derived class or subclass). In Java, inheritance is used for two purposes:
class inheritance - create a new class as an extension of another class, primarily for the purpose of code reuse. That is, the derived class with its own methods and data implementation inherits the public methods and public data of the base class. The methods of super class can be overridden in the sub class. Java only allows a class to have one immediate base class, i.e., single class inheritance.The two keywords, this and super helps to explicitly name the field or method. Using this and super we can decide whether to call a method or field present in the same class or to call from the immediate superclass. We use "this" keyword as a reference to the current object which is an instance of the current class. Whereas to extend a class constructor to reuse the superclass constructor and overridden superclass methods we use "super" keyword.
Interface inheritance - create a new class to implement the methods defined as part of an interface for the purpose of subtyping. That is a class that implements an interface "conforms to" (or is constrained by the type of) the interface. Java supports multiple interface inheritance.
For class inheritance, we have to use the keyword extends and for interface inheritance keyword implements should be used.

- **Keyword final usage with inheritance:**

If we want to prevent a class from being inherited then we have to precede the class declaration with keyword "final". This means a class declared as 'final' can never be sub classed. Declaring a class as final implicitly declares all of its methods as final, too.

But it is error to declare a class as both abstract and final since an abstract class is incomplete by itself and depends upon its subclasses to provide complete implementations.

'final' modifier can also be used to make methods from being inherited. Sometimes we may want some methods of super class not to be overridden by the sub class then we have to declare that method as final in super class.

'final' modifier can also be applied to variables. A final variable becomes a constant and its value cannot be changed.

Encapsulation:
The last important feature of OOP concept is encapsulation. Encapsulation is also known as data hiding, is to protect data from the user using the classes but still allowing the user to access data but not modify it. Through a public interface, the private data can be accessible to user class without worrying about messing with the private data. The data members can be accessed by the class functions only. In other words, Encapsulation is a process of binding or wrapping the data and the codes that operates on the data into a single entity. This keeps the data safe from outside interface and misuse.

Consider the following Program:
Animal abstract class:
package midtermexam_domain;

```
// @author Meghashree M Ramachandra
public abstract class Animal implements Relatable, Audible  {
    private String breed;
    protected int numberOfLegs;
    protected double loudness = 0.0;
    public Animal() {
    }

    public Animal(String breed, int numberOfLegs) {
        this.breed = breed;
        this.numberOfLegs = numberOfLegs;
    }

    public String getBreed() {
        return breed;
    }

    public void setBreed(String breed) {
        this.breed = breed;
    }

    public int getNumberOfLegs() {
        return numberOfLegs;
    }

    public void setNumberOfLegs(int numberOfLegs) {
        this.numberOfLegs = numberOfLegs;
    }

    public double getLoudness() {
        return loudness;
    }

    public void setLoudness(double loudness) {
        this.loudness = loudness;
    }

    @Override
    public String toString() {
        return "Animal{" + "breed=" + breed + ", numberOfLegs=" + numberOfLegs + ",
loudness=" + loudness + '}';
    }
}
```

Dog sub class:

```
package midtermexam_domain;
```

```
// @author Meghashree M Ramachandra
public class Dog extends Animal {

    public Dog() {
        this.loudness = 5.0;
    }

    public Dog(String breed, int numberOfLegs) {
        super(breed, numberOfLegs);
        this.loudness = 5.0;
    }
    @Override
    public String toString() {
        return "Dog{" + super.toString()+ '}';
    }
    @Override
    public String animalTalk() {
        return "Woof Woof!";
    }
    @Override
    public boolean isLouder(Object obj) {
        boolean result = false;
        if (obj instanceof Dog) {
            Dog downcastObj = (Dog) obj;
            if (this.loudness > downcastObj.loudness) {
                result = true;
            }
        } else if (obj instanceof Duck) {
            Duck downcastObj = (Duck) obj;
            if (this.loudness > downcastObj.loudness) {
                result = true;
            }
        }
        return result;
    }
}
```

Duck sub class:

```
package midtermexam_domain;
// @author Meghashree M Ramachandra
public class Duck extends Animal {
    public Duck() {
        this.loudness = 4.0;
    }
    public Duck(String breed, int numberOfLegs) {
        super(breed, numberOfLegs);
```

```
        this.loudness = 4.0;
    }
    @Override
    public String toString() {
        return "Duck{" + super.toString() +'}';
    }
@Override
    public String animalTalk() {
        return "Quack Quack!";    }
@Override
    public boolean isLouder(Object obj) {
        boolean result = false;
        if (obj instanceof Dog) {
            Dog downcastObj = (Dog) obj;
            if (this.loudness > downcastObj.loudness) {
                result = true;
            }
        } else if (obj instanceof Duck) {
            Duck downcastObj = (Duck) obj;
            if (this.loudness > downcastObj.loudness) {
                result = true;
            }
        }
        return result;
    }
}
```

Interface Relatable:

```
package midtermexam_domain;
// @author Meghashree M Ramachandra
public interface Relatable {
    boolean isLouder(Object obj);
}
```

Interface Audible:

```
package midtermexam_domain;
// @author Meghashree M Ramachandra
interface Audible {
    public String animalTalk(); // abstract methods only inside interfaces...
}
```

Driver Class:

```
package midtermexam_abstraction;

import midtermexam_domain.Animal;
import midtermexam_domain.Dog;
```

```
import midtermexam_domain.Duck;
// @author Meghashree M ramachandra
public class MidTermExam_Program1 {

   public static void main(String[] args) {
      //Animal animal = new Animal(); // cannot instantiate since abstract!
//Method Overloading
      Dog dog = new Dog("Retriever",4);
      Dog d = new Dog();
      Duck duck = new Duck("Eider",2);
      Duck b = new Duck();

      // Upcast to Animal abstract class...
      // Polymorphism uses toString overrides at subclass level even though
      // upcasted to Animal level = superclass.
      Animal [] animals = {dog,duck};
       System.out.println("Polymorphism and Upcasting - Upcasted sub class(dog, duck)
                                  to super class(Animal abstract class)");
      for (Animal anim : animals) {
         System.out.println(anim);
      }

      // Upcast to Object class...
      // Polymorphism uses toString overrides at subclass level even though
      // upcasted to Object level = super-superclass.
      Object [] objects = {dog,duck};
      System.out.println("Polymorphism and Upcasting- Upcasted sub class(dog, duck) to
super-superclass(Object level)");
      for (Object obj : objects) {
         System.out.println(obj);
      }

      Animal duckUpcast = duck; // Can be done implicitly without casting syntax.
      Object dogUpcast = dog;

      if (duckUpcast instanceof Duck) {
         System.out.println("Downcasting from superclass(Animal abstract class)to sub
class(dog, duck)");
         Duck duckDowncast = (Duck)duckUpcast;
         System.out.println("Downcasting duckUpcast to Duck success");
      }

      if (dogUpcast instanceof Dog) {
         System.out.println("Downcasting from super-superclass(Object level to sub
class(dog, duck)");
         Dog dogDowncast = (Dog)dogUpcast;
```

```
            System.out.println("Downcasting dogUpcast to Dog success");
        }

        // Polymorphically obtain correct animal sound from array of Animal...
        System.out.println("Interface methods result.");
        for (Animal anim : animals) {
            System.out.println(anim.animalTalk());
        }

        if (duck.isLouder(dog)) {
            System.out.println("A duck is louder than a dog!");
        }
        else System.out.println("A dog is louder than a duck!");

    }
}
```

The above program clearly explains all the concepts explained above:
Inheritance was demonstrated by the subclass, which inherited the functionality from the super class(Animal abstract) and the interface (Relatable, Audible).
The 'Dog' and 'Duck' class demonstrated the characteristics of the 'Animal' when it is instantiated with the subclass type. Thus, demonstrating the concept of **Polymorphism**.
Overloading and **Overriding** methods are demonstrated in driver class where instance of sub class Dog and Duck is created which is of constructor with no parameters and full arg parameter thus implementing Overloading, and the toString method declared in Dog and Duck sub class overrides the toString method declared in Animal super class.
Upcasting and **Downcasting** methods are implemented in driver class where Dog and Duck is upcasted to Animal and Object class, and again downcasting from Animal and Object class to Dog and Duck sub class.
The concept of '**Abstraction**' was demonstrated as follows:
The super class 'Animal' was made '**abstract class**', which means that the class cannot be instantiated and it is incomplete. However, the functionality of this class will be used by its subclass Dog and Duck. Similarly, **Interface** concept was also shown by using Relatable and Audible interfaces.
Encapsulation is demonstrated is class Animal where type is declared as private. The data of type field can only be retrieved and used by sub class using respective getter and setter methods but cannot be modified.

OOP language differs from a procedural/Imperative Language:

- In OOP program, unit of program is object, which is nothing but combination of data and code. In procedural program, programming logic follows certain procedures and the instructions are executed one after another.
- In OOP's program, it is accessible within the object and which in turn assures the security of the code where as in procedural program, data is exposed to the whole program.

- OOP is concerned to develop an application based on real time while Procedural Programming Languages are more concerned with the processing of procedures and functions.
- In OOP, the Objects communicate with each other via Functions while there is no communication in PPL rather it's simply a passing values to the Arguments to the Functions and / or procedures.
- OOP follows Bottom up Approach of Program Execution while in PPL its Top Down approach.
- OOP concepts include Inheritance, Encapsulation and Data Abstraction, Late Binding, Polymorphism, Multithreading, and Message Passing while PPL is simply a programming in a traditional way of calling functions and returning values.

Advantages of using an OOP language over procedural language for application design:

- OOPS provides a clear modular structure for programs which makes it good for defining abstract data types where implementation details are hidden and the unit has clearly defined interface.
- OOPS removes writing lot of code through reusable components and frameworks and designs.
- OOPS makes it easy to maintain and modify the existing code as new objects can be created with small differences.
- OOP methods make code more maintainable. Identifying the source of errors becomes easier because objects are self-contained (encapsulation). The principles of good OOP design contribute to an application's maintainability.
- Fits the way the real world works. It is easy to map a real world problem to a solution in OOP code.

References:

http://www.tutorialspoint.com/java/java_abstraction.htm
http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html
http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html
http://www.codeproject.com/Articles/11155/Abstract-Class-versus-Interface
http://www.sitepoint.com/quick-guide-to-polymorphism-in-java/
http://java67.blogspot.com/2012/09/difference-between-overloading-vs-overriding-in-java.html
http://www.authorstream.com/Presentation/vachas236-1380206-final/
http://www.artima.com/designtechniques/rtciP.html
http://docs.oracle.com/javase/tutorial/java/concepts/inheritance.html
http://www.opengurukul.com/vlc/mod/page/view.php?id=379
http://wiki.answers.com/Q/What_is_the_difference_between_object-oriented_and_procedural_programming_languages
http://www.tutorialspoint.com/java/java_encapsulation.htm
The Java Programming Language, 4th Edition 2005, By Ken Arnold, James Gosling, David Holmes.
Prof. Kimont Videos.

Essay Two:

Describe the two fundamental types of stream IO. What is a cascaded stream? What are buffered streams? Why are these types of streams useful? How are primitive data types streamed? What complications can arise when using such streams? What is object serialization/deserialization? How is it accomplished with streams and what is necessary when coding for it? What advantages does it provide? How does the term persistence relate to serialization capabilities? How does the keyword transient relate to serialization capabilities?

Answer:

The two fundamental types of stream IO are:

1. Byte Stream
2. Character Stream

They are present in java.io package.

1. ***Byte Stream*:**

Byte Streams are used for input and output operation of 8-bit bytes in the program. Byte streams are used for data-based I/O (Data-based I/O works with streams of binary data, such as the bit pattern for an image). Byte streams are considered to be descended from Inputstream and Outputstream. The java.io package defines abstract classes for basic byte input and output streams. These abstract classes are then extended to provide several useful stream types. Stream types are almost always paired: For example, where there is a FileInputStream to read from a file, there is usually a FileOutputStream to write to a file.

Inputstream: The abstract class InputStream declares methods to read bytes from a particular source. InputStream is the superclass of most byte input streams in java.io. specialized data types. An input stream is used to read data in to a program's memory.

Outputstream:

The abstract class OutputStream is analogous to InputStream; it provides an abstraction for writing bytes to a destination i.e. to write data out to some external form of memory such as a text or xml file.

Below Example demonstrates the usage of *FileInputStream* and *FileOutputStream*, which uses byte streams to copy inputdata.txt to outputdata.txt, one byte at a time.

```
package midtermexam_pragram2bytestreams;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.logging.Level;
import java.util.logging.Logger;
// @author Meghashree M Ramachandra
public class MidTermExam_Pragram2ByteStreams {

    public static void main(String[] args) {
```

```
FileInputStream in = null;
FileOutputStream out = null;

try {
   //instance of FileInputStream and FileOutputStream.
   in = new FileInputStream("data/inputdata.txt");
   out = new FileOutputStream("data/outputdata.txt");
   int c;
   //Read and Write data
   while ((c = in.read()) != -1) {
      out.write(c);
   }
} catch (IOException ex) {

Logger.getLogger(MidTermExam_Pragram2ByteStreams.class.getName()).log(Level.SE
VERE, null, ex);
   } finally {
//Close file pointer after job is done
      if (in != null) {
         try {
            in.close();
         } catch (IOException ex) {

Logger.getLogger(MidTermExam_Pragram2ByteStreams.class.getName()).log(Level.SE
VERE, null, ex);
         }
      }
      if (out != null) {
         try {
            out.close();
         } catch (IOException ex) {

Logger.getLogger(MidTermExam_Pragram2ByteStreams.class.getName()).log(Level.SE
VERE, null, ex);
         }
      }      }    }}
```

2. *Character Stream*:

The abstract classes for reading and writing streams of characters are Reader and Writer. Each supports methods similar to those of its byte stream InputStream and OutputStream, respectively.For example, InputStream has a read method that returns a byte as the lowest 8 bits of an int, and Reader has a read method that returns a char as the lowest 16 bits of an int. And where OutputStream has methods that write byte arrays, Writer has methods that write char arrays. Basically,character stream are actually built on top of a byte stream.

Reader:

The abstract class Reader provides a character stream analogous to the byte stream InputStream.

Writer:

The abstract class Writer provides a stream analogous to OutputStream but designed for use with characters instead of bytes.

Below Example demonstrates the usage of *FileReader* and *FileWriter*,

```
package midtermexam_program2characterstreams;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
//@author Meghashree M Ramachandra

public class MidTermExam_Program2CharacterStreams {

   public static void main(String[] args) {

      FileReader inputStream = null;
      FileWriter outputStream = null;

      try {
         //instance of FileReader and FileWriter.
         inputStream = new FileReader("data/inputdata.txt");
         outputStream = new FileWriter("data/outputdata.txt");

         int c;
         //Read data from file and write data into another file
         while ((c = inputStream.read()) != -1) {
            outputStream.write(c);
         }
      } catch (IOException ex) {

Logger.getLogger(MidTermExam_Program2CharacterStreams.class.getName()).log(Lev
el.SEVERE, null, ex);
      } finally {
         if (inputStream != null) {
            try {
               inputStream.close();
            } catch (IOException ex) {

Logger.getLogger(MidTermExam_Program2CharacterStreams.class.getName()).log(Lev
el.SEVERE, null, ex);
            }
```

```
        }
        if (outputStream != null) {
           try {
              outputStream.close();
           } catch (IOException ex) {

Logger.getLogger(MidTermExam_Program2CharacterStreams.class.getName()).log(Lev
el.SEVERE, null, ex);
           }
        }
     }
  }}
```

The most important difference between above character stream example and Byte Stream example is that character stream example uses and FileWriter for input and output in place of FileInputStream and FileOutputStream. As explained above, character stream example holds character value in its last 16 bits, whereas in byte stream example holds byte value in its last 8 bits.

Cascaded stream:
Cascaded with respect to stream construction is defined as providing the output of a stream as an input to another stream.
FileInputStream fis = new FileInputStream("data/daylight-record.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
In the above example, the declaration of the IO stream begins with an instance of the java.io.FileInputStream class, which has the physical access to daylight-record.ser file. This class is then used to create an instance of the secondary java.io.ObjectInputStream filter class. This dependent declaration shows the meaning of casacaded streams as fis is provided as input to ois which ObjectInputStream.

Buffered Streams:
Java.io.Bufferedstreams are employed in java to overcome the shortcomings when Unbuffered streams are used. The Buffered stream classes BufferedInputStream, BufferedOutputStream,BufferedReader, and BufferedWriter buffer their data to avoid every read or write going directly to the next stream.

Each of the Buffered streams supports two constructors:
Have to take a reference to the wrapped stream and the size of the buffer to use, while the other only takes a reference to the wrapped stream and uses a default buffer size.
When *Buffered input streams* read is invoked on an empty Buffered input stream, it fills the buffer with as much data as is available in source stream and returns the requested data from that buffer.
Example:

```
//For Byte Streams
// A FileInputStream object is created
 FileInputStream fileis = new FileInputStream (pathname);
```

// FileInputStream object reference is passed to BufferedInputStream constructor
BufferedInputStream bufferedis = new BufferedInputStream (fileis);
Buffered output streams do the same. When a write fills the buffer, the destination stream's write is invoked to empty the buffer. This buffering can turn many small write requests on the Buffered stream into a single write request on the underlying destination.
Example:
//***For Byte Streams***
// A FileOutputStream object is created
FileOutputStream fileos = new FileOutputStream (pathname);
// FileOutputStream object reference is passed to BufferedOutputStream constructor
BufferedOutputStream bufferedos = new BufferedOutputStream (fileos);

The BufferedReader and BufferedWriter classes work just like their byte-based counterparts, but operate on characters instead of bytes. The Buffered character streams also understand lines of text. The newLine method of BufferedWriter writes a line separator to the stream..The method readLine in BufferedReader returns a line of text as a String. The method readLine accepts any of the standard set of line separators. If the end of stream is encountered before a line separator, then the text read to that point is returned. If only the end of stream is encountered readLine returns null.
Example:
//***For Character Streams***
// A FileReader object is created
FileReader fileis = new FileReader (pathname);
// FileReader object reference is passed to BufferedReader constructor
BufferedReader bufferedis = new BufferedReader (fileis);

//***For Character Streams***
// A FileWriter object is created
FileWriter fileos = new FileWriter (pathname);
// FileOutputStream object reference is passed to BufferedWriter constructor
BufferedWriter bufferedos = new BufferedWriter (fileos);

These Buffered streams are useful because:
It makes program efficient as it reduces the disk access, network activity, or some other operation that is relatively expensive and maximizes the I/O performance.
It adds functionality to streams namely the ability to buffer and to support the mark and reset methods.

Primitive Data Types Streamed:
Primitive data types (boolean, char, byte, short, int, long, float, and double) streamed using Data streams. Data streams are filtered streams that perform binary I/O operation. These streams filter an existing byte stream so that each primitive data types can be read from or written to the stream directly.
These Data streams are as follows:

1. DataInputStream
2. DataOutputStream

1. ***DataInputStream*:**

This class is derived from **FilterInputStream** class that reads binary represented data of Java primitive data types from an underlying input stream in a machine-independent way. It reads only Java primitive data types and doesn't read the object values. The constructor of DataInputStream is written as:

DataInputStream (java.io.InputStream in);

The argument that is to be filtered should be an existing input stream (**buffered input stream** or **file input stream**). The read () method is used to read the data according to its types.

2. ***DataOutputStream*:**

This class is derived from **FilterOutputStream** class that writes binary represented data of Java primitive data types reading from an underlying output stream in a machine-independent way. It writes only Java primitive data types and doesn't write the object values. The constructor of DataOutputStream is written as:

DataOutputStream (java.io.OutputStream out);

The argument that is to be filtered should be an existing output stream (**buffered output stream** or **file output stream**).The **write( )** method is used to write the data according to its types.

Below Example demonstrates the implementation of reading and writing operations through the Data I/O streams:

```
package midtermexam_progrm2datastreams;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
 // @author Meghashree M Ramachandra
public class MidTermExam_Progrm2DataStreams {
   public static void main(String[] args) {
      int integerValue = 1;
      String StrinValue = "Hello";
      float floatValue = 10.50f;
      try {
         // Create an instance of FileOutputStream with primitivedata.dat
         // as the file name to be created. Then we pass the input
         // stream object in the DataOutputStream constructor.
         FileOutputStream fos = new FileOutputStream("data/primitivedata.dat");
         DataOutputStream dos = new DataOutputStream(fos);
         //  write  data to the primitivedata.dat.
         // using dataoutputstream methods writeInt(), writeFloat(), writeUTF().
         dos.writeInt(integerValue);
         dos.writeUTF(StrinValue);
         dos.writeFloat(floatValue);
         dos.flush();
```

```
    dos.close();
   //read back data and display it.
    // Using DataInputStream methods readInt(), readFloat(),readUTF(), etc.
    FileInputStream fis = new FileInputStream("data/primitivedata.dat");
    DataInputStream dis = new DataInputStream(fis);
    // Read the data
    int integerValueResult = dis.readInt();
    System.out.println("integer: " + integerValueResult);
    String StringValueResult = dis.readUTF();
    System.out.println("string: " + StringValueResult);
    float floatValueResult = dis.readFloat();
    System.out.println("float: " + floatValueResult);
  } catch (IOException e) {
    e.printStackTrace();
  }   }}
```

Complications with data streams:
The main complication while using data streams arises when using it for floating point numbers to represent financial values i.e. precise values. It is particularly bad for decimal fractions, because common values (ex 0.1) do not have a binary representation.

Object Serialization/Deserialization:
***Serialization*:**
It can be defined as conversion of an object to a series of bytes, so that the object can be easily saved to persistent storage (such as file or a memory buffer) or to transmit it over a network connection in binary form and later can be retrieved and deserialized.
***Deserialization*:**
It is the opposite process of serialization where we retrieve the object back from the converted series of bytes which is stored as persistent storage (such as file or a memory buffer) or network connection.

The necessary thing while coding for serialization/deserialization: an abstract class has to implement the interface ***Serializable*** and it is present in the package java.io.Serializable.
The serialization and deserialization can be done for byte stream using ObjectOutputStream and ObjectInputStream respectively.

The constructor for ObjectOutputStream for serialization is as shown below:
ObjectOutput objOut = new ObjectOutputStream(new FileOutputStream(f));

The constructor for ObjectInputStream for deserialization is as shown below:
ObjectInputobjOut = new ObjectInputStream(new FileOutputStream(f));

Below instance example shows the serialization and deserialization to be done using ObjectInputStream and ObjectOutputStream:
// Serialize an object of type MidtermExamClass

```
MidtermExamClass myObject = new MidtermExamClass ();
FileOutputStream fos = new FileOutputStream("data/myObject.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(myObject);
oos.flush();
oos.close();

//Deserialize the object persisted in "myObject.ser"
FileInputStream fis = new FileInputStream("data/myObject.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
MidtermExamClass myDeserializedObject = (MidtermExamClass)ois.readObject();
ois.close();
```

We should be sure to make MidtermExamClass implements the java.io.Serializable interface.

Advantages of Using Serialization/Deserialization:

- It is easy to use and can be customized.
- The serialized stream can be encrypted, authenticated and compressed, supporting the needs of secure Java computing.
- Serialized classes can support coherent versioning and are flexible enough to allow gradual evolution of application's object schema.
- A method of persisting objects which is more convenient than writing their properties to a text file on disk, and re-assembling them by reading this back in.
- Reducing time taken to write code for save and restoration of object or application state
- Making it easier for objects to travel over a network connection.

Persistence with serialization:

Persistence means having an object's life independent from the life time of the application in which it is running. Object persistence refers to the possibility of objects running across application. Persistence can be implemented by storing objects and then retrieving them.

Constructor for creating a Persistent Object:

```
PersistentObject po = new PersistentObject(Args);
```

Below instance example demonstrates the serialization and deserialization of persistent object:

```
//Serializing the Persistent Object
PersistentObject pObj = new PersistentObject(new dayLightRecords);
try {
        ObjectOutputStream oostream = new ObjectOutputStream(new
        FileOutputStream("data/daylightrecord.ser"));
        oostream.writeObject(pObj);
} catch (IOException ex){
}
```

```
//DeSerializing the Persistent Object
try {
        ObjectInputStream oistream = new ObjectInputStream(new
        FileInputStream("data/ daylightrecord.ser”));
try {
        PersistentObject pObjOutput = (PersistentObject)oistream.readObject();
} catch (IOException ex){
}
```

Here also we should make sure to make dayLightRecords class implements the java.io.Serializable interface.

Transient Keyword:

The main purpose of transient keyword is to avoid serialization. When an object is serialized, all the variable in the object is converted to persistent state. Sometimes, we don’t want certain variables in the persistent state as they are not necessary to store or transfer across network. So, we can do this by using transient keyword. The variable declared with “transient” keyword i.e. preceded with transient will not be serialized or persisted.

References:

http://www.javacoffeebreak.com/articles/serialization/index.html
http://www.javabeat.net/2009/02/what-is-transient-keyword-in-java/
http://www.devx.com/tips/Tip/12932
http://www.roseindia.net/java/example/java/io/SerializingObject.shtml
http://www.roseindia.net/java/example/java/io/DataStreams.shtml
http://docs.oracle.com/javase/tutorial/essential/io/datastreams.html
http://docs.oracle.com/javase/tutorial/essential/io/charstreams.html
http://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html
The Java Programming Language, 4th Edition 2005, By Ken Arnold, James Gosling, David Holmes.
Prof. Kimont Videos.

Essay Three:
Discuss the differences between a process and a thread. Describe two ways to create a thread object. Which technique is better and why? Discuss a thread's lifecycle. Define the Producer/Consumer parallel pattern. What two ways are there to synchronize a multi-threaded application? What impact do shared resources have on multi-threaded programs?

Answer:
Differences between process and thread are as follows:

- Both process and Thread is independent path of execution but one process can have multiple Threads.
- Every process has its own memory space, executable code and a unique process identifier (PID) while every thread has its own stack in Java but it uses process main memory and shares it with other threads.
- A process is also referred as "heavyweight", while threads are referred as "light weight" in operating system
- Processes are heavily dependent on system resources available while threads require minimal amounts of resource.
- Modifying a main thread may affect subsequent threads while changes on a parent process will not necessarily affect child processes. Threads within a process communicate directly while processes do not communicate so easily.
- It's easy to create Thread as compared to Process which requires duplication of parent process.
- All Threads which is part of same process share system resource like file descriptors , Heap Memory and other resource but each Thread has its own Exception handler and own stack in Java.

Two ways to create a thread object:

1. Implement Runnable Interface
2. Create subclass for java.lang.Thread.

1. ***Implement Runnable Interface:***

One way to create a thread in java is to implement the Runnable Interface and then instantiate an object of the class. We need to overload the run method of Runnable interface with the code that has to be executed by the thread. The Runnable object is passed to the Thread constructor as shown below example:

```
public class HelloWorldRunnable implements Runnable {
   public void run() {
      System.out.println("Hello World from a Runnable Interface!");
   }
   public static void main(String args[]) {
      (new Thread(new HelloWorldRunnable())).start();
   }}
```

2. ***Create subclass for java.lang.Thread:***

Another way to create a thread object is to create a subclass or derived class of java.lang.Thread class and instantiate it. Here, we need to overload the run() method of Thread class with the code that has to be executed by the thread. An application can subclass Thread, providing its own implementation of run, as shown in below example:

```
public class HelloWorldThread extends Thread {
   public void run() {
      System.out.println("Hello World from sub class of a thread!");
   }
   public static void main(String args[]) {
      (new HelloWorldThread ()).start();
   }}
```

We need to execute Thread.start for both above example to start the thread.

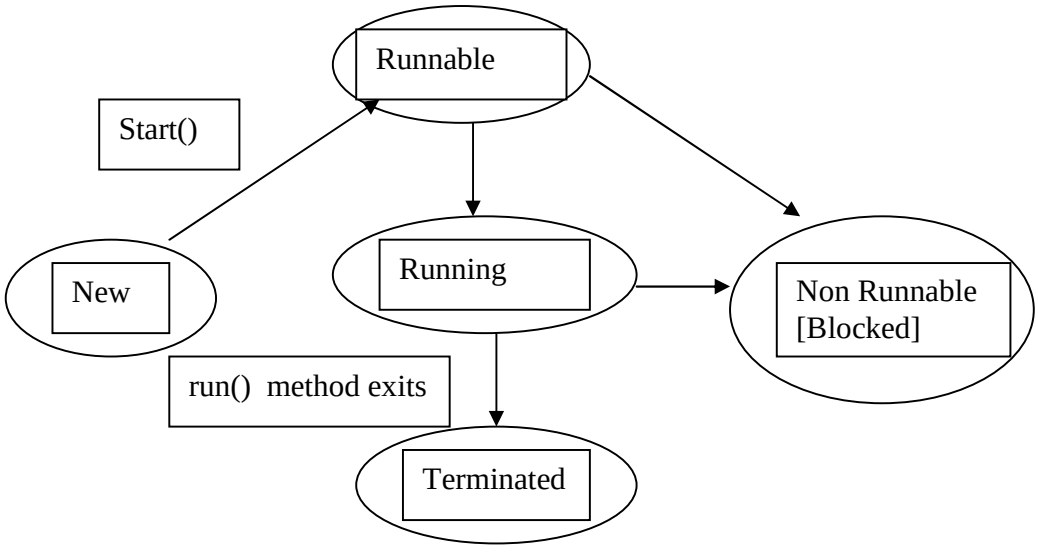
Among two techniques Runnable Interface is considered preferable for following reasons:

- Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

Thread Lifecycle:

A thread can be in one of the below five states:

1. New
2. Runnable
3. Running
4. Non Runnable [Blocked]
5. Terminated



1. New:

After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.

2. Runnable:

A thread is in Runnable state after invocation of start() method, but the thread scheduler has not yet selected it to be the running thread.

3. Running:

A thread is in running state that means the thread is currently executing. The thread enters into the running state after the thread scheduler has selected it.

4. Non Runnable [Blocked]:

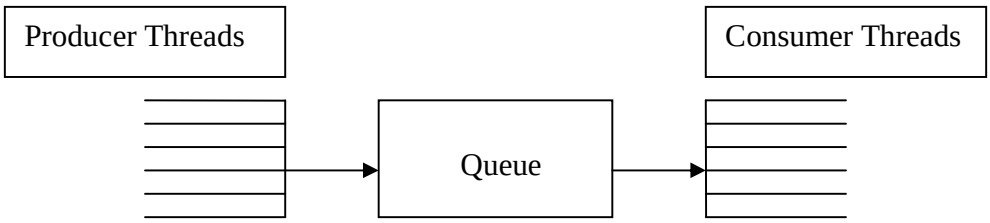
This is the state where thread is still alive but not currently eligible to run i.e. it may be waiting for the resources that are hold by another thread.

5. Terminated:

A thread is terminated or is in a dead state when run() method exits.

Producer/Consumer Pattern:

Producer/Consumer pattern idea is to process data asynchronously by partitioning requests among different groups of threads. The producer is a thread (or group of threads) that generates requests (or data) to be processed. The consumer is a thread (or group of threads) that takes those requests (or data) and acts upon them. This pattern provides a clean separation that allows for better thread design and makes development and debugging easier. The patter can be shown diagrammatically as follows:



There are two main types of synchronization that Java supports for multithreaded application:

1. Monitors
2. Mutexes.

1. Monitors:

A fundamental responsibility of an operating system is to protect system resources from unrestricted access, like a monitor protects the internal data and methods of an object from unrestricted access by other threads.Each Java object has a monitor and only one

thread at a time has access to that monitor. When more than one thread wants access to an object's monitor, they must wait until that monitor is released. The Thread object itself has nothing to do with implementing monitors. Monitors are inherent in every Java object, and every Java object has its own independent monitor.

Java defines the keyword "synchronized" to gain access to an object's monitor. There are two ways to use synchronize: either by method or by block. To allow only one thread at a time to access an object's method, use the "synchronized" keyword in the definition of the method.To allow only one thread at a time access to a portion of an object's method, use the block form of synchronized within the method .

2. Mutexes:

Mutexes are used when two or more threads can't interleave certain types of operations. Thus, one sequence must be completed before the other is started. The generic mutex methods are: wait(), notify() and notifyAll() are available to all Java objects because they are declared in the Object Java class. These methods allow any thread to wait for any other thread to complete some activity. When the activity is complete, the thread notifies one of the (or all if notifyAll is called) waiting threads.

Shared resources on multithreaded programs:

Threading sometimes may become very hard to control, especially when it is accessing shared resources. In such cases, the threads have to be coordinated; otherwise it may leads to race condition (It is a parallel execution of your program by multiple threads at same time) problem. For example, a printer cannot be used to print two documents at the same time and if there are multiple printing requests, threads managing these printing operations needs to be coordinated.

The problem related to shared resources on multiple threading can be defined in two ways: read/write problem and producer-consumer problem

Read/Write problem:

If one thread tries to read the data and another thread tries to update the same data, it is known as read/write problem, and it leads to inconsistent state for the shared data. This can be solved by synchronizing access to the data via Java synchronized keyword. For example,

```
public synchronized void update() {
…
}
```

Producer-Consumer Problem:

The producer-consumer problem (also known as the bounded-buffer problem) is another example of a multithread synchronization or shared resources problem. The problem describes two threads, the producer and the consumer, who share a common, fixed-size buffer. The producer’s job is to generate a piece of data and put it into the buffer. The consumer is consuming the data from the same buffer simultaneously. The problem arises when the producer try to add data into the buffer if it is full or the consumer will try to remove data from an empty buffer.

The solution for this problem involves two parts. The producer should wait when it tries to put the newly created product into the buffer until there is at least one free slot in the buffer. The consumer, on the other hand, should stop consuming if the buffer is empty.
Basically, problems of shared resources using multiple threading can be solved by writing the programs which are thread safe one example is as shown below:

```
public class Counter {
   private int count;
   AtomicInteger atomicCount = new AtomicInteger( 0 );
    //This method thread-safe because of locking and synchornization
   public synchronized int getCount(){
      return count++;
   }
    //This method is thread-safe because count is incremented atomically
   public int getCountAtomically(){
      return atomicCount.incrementAndGet();
   }}
```

References:

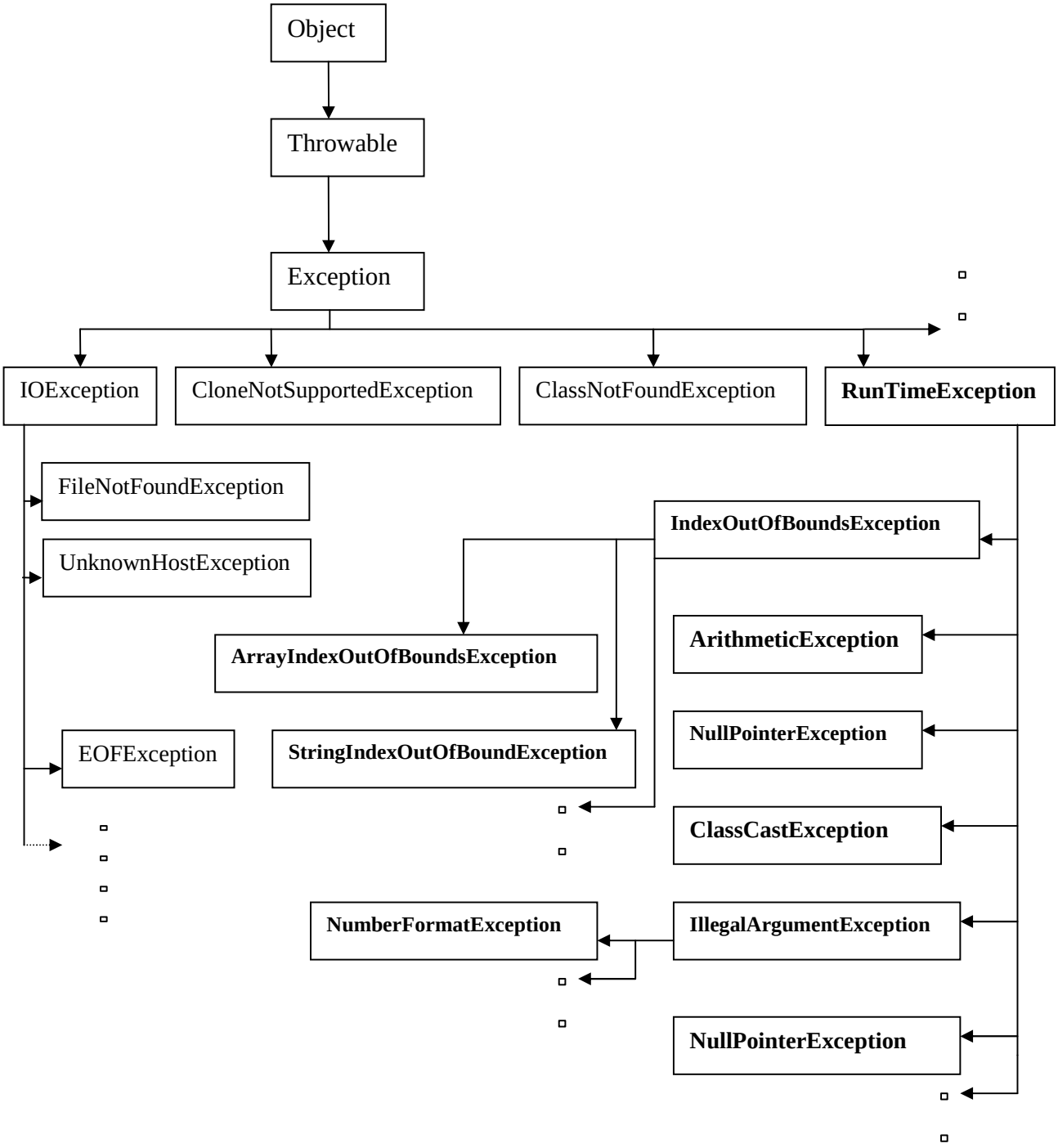
http://www2.sys-con.com/itsg/virtualcd/java/archives/0301/cunninham/index.html
http://book.javanb.com/java-threads-3rd/jthreads3-CHP-8-SECT-3.html
http://www.javatpoint.com/life-cycle-of-a-thread
http://www.javabeginner.com/learn-java/java-threads-tutorial
http://www.differencebetween.net/miscellaneous/difference-between-thread-and-process/
http://www.buyya.com/java/Chapter14.pdf
The Java Programming Language, 4th Edition 2005, By Ken Arnold, James Gosling, David Holmes.
Prof. Kimont Videos.

Essay Four:
Discuss the importance of exceptions and exception handling in your applications. What is the class hierarchy of an Exception? an Error? What is the difference between these data types?

Answer:
Importance of exceptions and exception handling:

- Exception handling helps us catch or identify abnormal scenarios in our code and handle them appropriately instead of throwing up a random error on the front-end (User Interface) of the application.
- It helps is grouping and Differentiating Error Types i.e. since all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.
- One of the significance of this mechanism is that it throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
- Exception handling allows developers to detect errors easily without writing special code to test return values. Even better, it lets us keep exception-handling code cleanly separated from the exception-generating code. It also lets us use the same exception-handling code to deal with a range of possible exceptions.

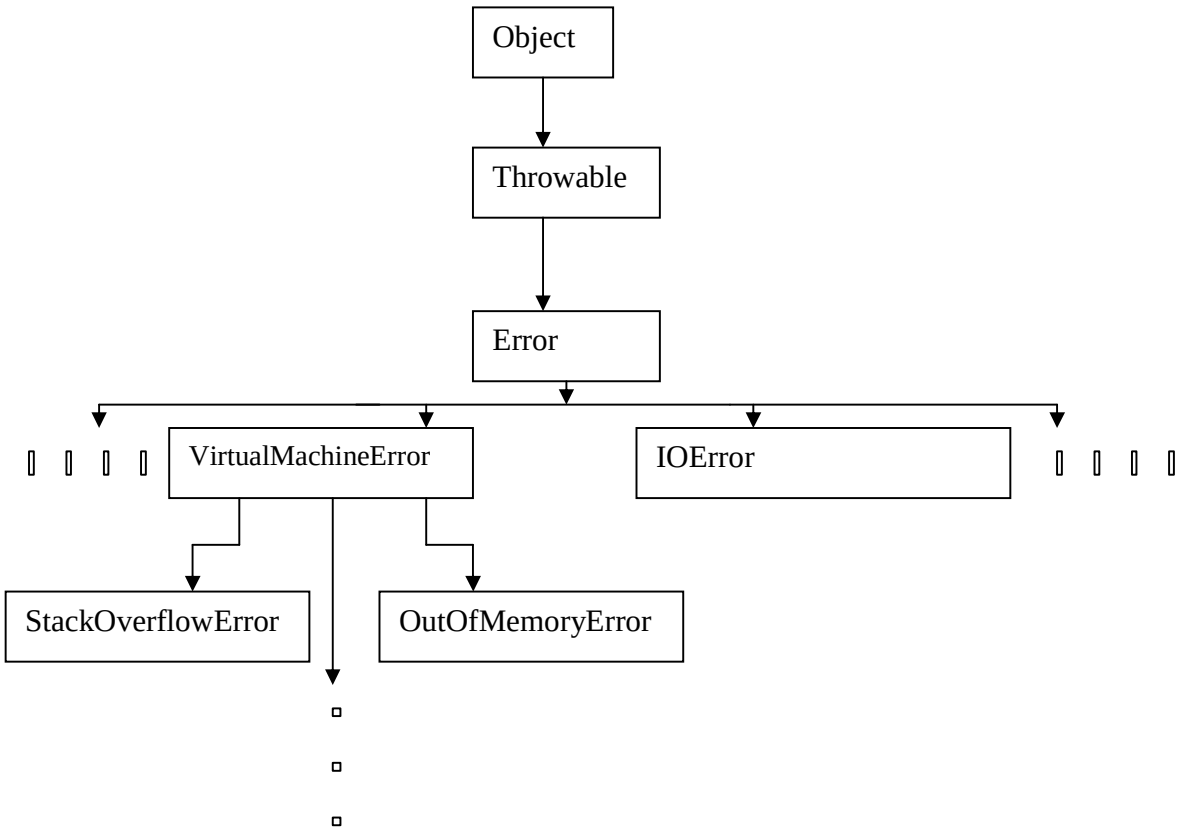
Class Hierarchy of exception:
Exception and its entire sub classes can be handled within the program. Exception are categorized as checked exception and Unchecked exception.
Checked exception: Exception that must be declared in throws clause of a method signature and is caught at compile time. All exceptions other than RuntimeException and its subclasses are checked exception.
Unchecked Exception: Exception that they need not be declared using throws in method signature. RuntimeException and all its sub classes are called unchecked exceptions.
The Exception hierarchy (only important exceptions are shown) can be represented as shown below, unchecked exception is marked as bold:



Class Hierarchy of Error:

Error and all its sub classes are expected to be handled by the program. Error and its sub type are used by Java's run time system to indicate problems related to run time environment like out of memory or stack overflow. For example, out of memory problem (OutOfMemoryError) cannot be fixed using program code and is not expected to be handled by the program.

The Error hierarchy (only important errors are shown) can be represented as shown below:



Difference between Exception and Error:

Exception	Error
Exception can be caught and handled properly. Ex FileNotFoundException if file not found.	Error cannot be caught neither handled. Ex: OutOfMemoryError if no memory available.
Exception which may not be fatal in all cases.	Error are often fatal in nature and recovery from Error is not possible.
Exception is generally divided into two categories: checked and unchecked Exceptions.	Error is not divided.
Exceptions can be system defined or can be user defined.	Run time errors are thrown by system only.
Exception doesn't terminate the program (except unckecked exception i.e. RuntimeException)since it's handled in try-catch block.	Error causes the program to terminate.

References:
http://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html
http://wiki.answers.com/Q/What_is_the_difference_between_exception_and_error_in_java
http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html
http://javapapers.com/core-java/exception-handling-2/
The Java Programming Language, 4th Edition 2005, By Ken Arnold, James Gosling, David Holmes.
Prof. Kimont Videos.