

Guia de estudio para OCPJSE7P

"Oracle Certified Professional Java SE 7 Programmer"

I. Cambios en este documento

Fecha	Cambios
2014-04-08	Primera publicación

II. Información del examen

Duración:	150 minutes
Nro. de preguntas:	90
Score para pasar:	65%
Código del examen:	1Z0-804
Nombre del examen:	Java SE 7 Programmer II
Costo:	\$ 150

Detalles completos del examen

https://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=5001&get_params=p_exam_id:1Z0-804

III. Temas del examen

1. Diseño de clases Java
2. Diseño avanzados de clases
3. Principios del diseño orientado a objetos
4. Genericos y colecciones
5. Procesamiento de String
6. Excepciones y aserciones
7. Fundamentos de Java I/O
8. Java File I/O (NIO.2)
9. Desarrollando aplicaciones con acceso a base de datos usando JDBC
10. Hilos
11. Concurrency
12. Localización

IV. Desarrollo de temas

1. Diseño de clases Java

1.1. Uso de los modificadores de acceso: **private, protected y public**

- ❑ Existen 3 modificadores de acceso: **public, protected** y **private**.
- ❑ Existen 4 niveles de acceso: **public, protected, default** y **private**.
- ❑ Las clases solo pueden tener el acceso **public** o default.
- ❑ Una clase con el acceso default solo puede ser vista por clases dentro del mismo paquete.
- ❑ Una clase con el acceso publico puede ser vista por cualquier clase de cualquier paquete.
- ❑ La visibilidad de una clase se refiera a que si el código de la clase puede:
 - ❑ Crear una instancia de otra clase.
 - ❑ Extender otra clase.
 - ❑ Acceder a miembros de otra clase.
- ❑ Métodos y variables de instancia (variables no locales) son conocidos como miembros.
- ❑ Los miembros pueden usar cualquier nivel de acceso: **public, protected, default, private**.
- ❑ El acceso a los miembros de una clase puede ser de 2 formas:
 - ❑ El código dentro de una clase puede acceder a los miembros de otra clase.
 - ❑ Una subclase puede heredar los miembros de la superclase.
- ❑ Si una clase no puede ser accedida tampoco se puede acceder a sus miembros.
- ❑ Se debe determinar la visibilidad de la clase antes de determinar la visibilidad de los miembros.
- ❑ Los miembros **public** se pueden acceder por cualquier clase de cualquier paquete.
- ❑ Si un miembro de la superclase es público la subclase heredo esto, sin importar en que paquete se encuentre.
- ❑ El acceso a miembros sin el punto (.) solo es posible dentro de la misma clase.
- ❑ **this** siempre se referencia al objeto actual cuando es creado.
- ❑ **this.unMetodo()** es lo mismo que invocar solamente **unMetodo()**.
- ❑ Los miembros **private** se pueden acceder únicamente dentro del código de la misma clase.
- ❑ Los miembros **private** no son visibles a las sub clases, por lo tanto los miembros privados no pueden ser heredados.

- ❑ El nivel de acceso **default** y **protected** son diferentes únicamente cuando hay subclases.
- ❑ Los miembros que tienen el nivel de acceso por defecto solo se pueden acceder desde las clases que están en el mismo paquete.
- ❑ El acceso a los miembros **protected** es para las clases del mismo paquete y para las subclases que pueden estar en otros paquetes.
- ❑ Para una subclase localizada en un paquete diferente a la superclase la única forma de acceder a los miembros **protected** de la superclase es por la herencia, no se puede obtener acceso por instanciar la superclase.
- ❑ Los miembros **protected** heredados por una subclase1 localizada en otro paquete no pueden ser accedidos por otras clases de ese paquete, pero si puede ser accedida por una subclase2 de la subclase1.

1.2. Sobre escritura de métodos

* El tipo de retorno de un metodo sobre escrito en una subclase no puede ser superclase del tipo de retorno en el metodo de la superclase, ejemplo:

```
public class AA {}  
public class A extends AA{}  
public class B extends A {}  
public class C extends B {}  
  
public class SuperTest {  
    public A callMe(){  
        return new A();  
    }  
}  
  
public class Test2 extends SuperTest {  
    //Genera un error de compilacion.  
    public AA callMe(){  
        return new C();  
    }  
}
```

El metodo callMe() de Test2 solo puede devolver A, B o C.

1.3. Sobre carga de constructores y métodos

Bla...bla

1.4. Uso del operador instanceof y conversiones

Bla...bla

1.5. Invocación de métodos virtuales

Bla...bla

1.6. Sobre escritura de los métodos hashCode, equals y toString de la clase Object

- ❑ Los métodos **equals()**, **hashCode()** y **toString()** son públicos.
- ❑ Cuando se sobre escribe **toString()** el metodo **System.out.println()** u otro metodo pueden ver algo útil del objeto, similar al estado de este.
- ❑ Use == para determinar si dos variables de referencia, se refieren al mismo objeto.
- ❑ Use **equals()** para determinar si dos objetos son significativamente equivalentes.
- ❑ Si no se sobre escribe **equals()**, diferentes objetos no pueden ser considerados iguales.
- ❑ Si no se sobre escribe **equals()**, los objetos no pueden hacer buenas llaves hash.
- ❑ **String** y las clases wrappers sobre escriben **equals()** y hace buenas llaves hash.
- ❑ Cuando se sobre escribe **equals()**, usar el operador **instanceof** para asegurar que se esta evaluando la clase apropiada.
- ❑ Cuando se sobre escribe **equals()**, se debe comparar los atributos significativos del objeto.
- ❑ Especificaciones del contrato de **equals()**:
 - ❑ Reflexiva: x.equals(x) es true.
 - ❑ Simétrica: si x.equals(y) es true, entonces y.equals(x) también es true.
 - ❑ Transitiva: Si x.equals(y) es true, y y.equals(z) es true, entonces z.equals(x) también es true.
 - ❑ Consistente: Multiples invocaciones a x.equals(y) siempre retorna el mismo resultado.
 - ❑ Null: Si x no es **null**, entonces x.equals(**null**) es false.
 - ❑ Si se sobre escribe el metodo **equals()**, también se debe sobre escribir el metodo **hashCode()**.
- ❑ HashMap, HashSet, Hashtable, LinkedHashMap, y LinkedHashSet usan hashing.
- ❑ Una apropiada sobre escritura de **hashCode()** se pega al contrato del **hashCode()**.
- ❑ Una eficiente sobre escritura de **hashCode()** distribuye las llaves uniformemente a través de las múltiples entradas.
- ❑ Es legal que el metodo **hashCode()** retorne el mismo valor para todas las instancias, pero en la practica es algo muy ineficiente.

- Especificaciones del contrato de **hashCode()**:
 - Consistent: multiple calls to x.hashCode() return the same integer.
 - Si x.equals(y) es true, entonces x.hashCode()==y.hashCode() es true.
 - Si x.equals(y) es false, entonces x.hashCode()==y.hashCode() puede ser false o true, pero false tenderá a crear una mejor eficiencia.
- Las variables **transient** o valores aleatorios no son apropiadas para **equals()** y **hashCode()**. Debido a que estos pueden retornar valores distintos en varias invocaciones.

1.7. Uso de la sentencia package e import

- * Un archivo fuente puede tener únicamente una sentencia **package** pero puede tener múltiples sentencias **import**.
- * Si existe la sentencia **package** esta debe ser la primera línea no comentada en el archivo fuente.
- * Si existe la sentencia **import** esta debe ser después de la sentencia **package** y antes de la declaración **class**.
- * Si no existe la sentencia **package**, y existe la sentencia **import** esta debe ser la primera línea no comentada.
- * La sentencia **package** e **import** aplican a todas las clases contenidas en el archivo fuente.
- * Se puede especificar un **import** estático así: **import static**.
- * Se puede usar un **import** estático para crear accesos directos a miembros **static** de cualquier clase.

2. Diseño avanzados de clases

2.1. Identify when and how to apply abstract classes

Bla...bla

2.2. Construct abstract Java classes and subclasses

- ❑ Las interfaces son contratos respecto a que debe hacer una clase, pero no especifica el como se debe realizar.
- ❑ Las interfaces pueden ser implementadas por cualquier clase de cualquier jerarquía.
- ❑ Una interface es similar a una clase 100% abstracta. La interface puede tener el modificador **abstract**, el cual es implícito.
- ❑ Una interface puede tener únicamente métodos abstractos, no esta permitido los métodos concretos.
- ❑ Por defecto los métodos de una interface son **public** y **abstract**, la declaración explícita de estos modificadores es opcional.
- ❑ Las interfaces pueden tener constantes las cuales son siempre implícitamente **public**, **static** y **final**, la declaración de cualquiera de estos modificadores es opcional, no es valido cualquier otro modificador.
- ❑ Una clase no abstracta de implementación debe seguir las siguientes reglas:
 - ❑ Debe proveer implementación para todos los métodos de la interface.
 - ❑ Deberá seguir todas las reglas de sobre escritura para los métodos que implementa.
 - ❑ No deberá declarar ninguna nueva excepción chequeada para el método que implementa.
 - ❑ No deberá declarar ninguna excepción chequeada que se más amplia (superclase) que la excepción declarada en el método de la interface.
 - ❑ Si puede declara cualquier excepción de tipo runtime (RuntimeException) en la implementación de cualquier método de interface sin importar la declaración en la interface.
 - ❑ Se debe mantener la firma exacta del método (es permitido retornos covariant) y el tipo de retorno del método que se implementa, no es necesario que se declare las excepciones del método que esta en la interface.
- ❑ Una clase que implementa una interface puede ser **abstract**.
- ❑ Una clase **abstract** que implementa una interface no tiene que implementar todos los métodos de la interface, pero la primera clase concreta que la extienda si debe hacerlo.

- Una clase solo puede extender una clase (No es valida la herencia múltiple), pero puede implementar varias interfaces a la vez.
- Las interfaces pueden extender una o varios interfaces a la vez.
- Las interfaces no pueden extender una clase o implementar un interface.

2.3. Use the static and final keywords

- Una clase también puede tener los modificadores **final**, **abstract** o **strictfp**.
- Una clase no puede ser **final** y **abstract** a la vez.
- Una clase **final** no puede ser heredada.
- Una clase **abstract** no puede ser instanciada.
- Si la clase tiene un simple método **abstract** entonces la clase debe tener el modificador **abstract**.
- Una clase abstract puede tener métodos abstract y no abstract (concretos) a la vez.
- La primera clase concreta que extienda una clase **abstract** debe implementar todos los métodos **abstract**.

2.4. Create top-level and nested classes

Clases internas (Inner).

- Una clase interna regular es declarada dentro de otra clase, pero fuera de cualquier metodo o bloque de código.

Ejemplo:

```
class MyOuter {  
    private int x = 7;  
  
    // inner class definition  
    class MyInner {  
        public void seeOuter() {  
            System.out.println("Outer x is " + x);  
        }  
    } // close inner class definition  
} // close outer class
```

- Una clase interna tiene los mismo derechos que cualquier otro miembro de la clase externa (outer), entonces puede ser marcada con los mismos modificadores de acceso como **abstract** o **final**.
- Las instancia de la clase interna comparte una especial relación con la instancia de la clase outer. La relación da a la clase interna accesos a todos los miembros de la clase outer, incluyendo los miembros privados.
- Para instanciar una clase interna se debe tener referencia a una instancia de la clase outer. Ejemplo:
MyOuter mo = new MyOuter();

```
MyOuter.MyInner inner = mo.new MyInner();
```

- Dentro del código de la clase outer se puede instanciar a la clase inner simplemente así:

```
MyInner mi = new MyInner();
```

- Dentro del código de la clase inner se usa **this** para referenciar a la instancia de inner. Para referenciar a la instancia de outer se debe hacer: **MyOuterClass.this**.

Clases internas Method-Local

- Una clase interna method-local es definida dentro de un método.

Ejemplo:

```
class MyOuter2 {
    private String x = "Outer2";
    void doStuff() {
        class MyInner {
            public void seeOuter() {
                System.out.println("Outer x is " + x);
            } // close inner class method
        } // close inner class definition
    } // close outer class method doStuff()
} // close outer class
```

- Para que la clase inner sea utilizada, esta debe ser instanciada, la instanciación debe ser dentro del mismo método, pero después del código que define la clase.
- Las clases internas method-local no pueden usar las variables que están declaradas en el método, incluyendo parámetros, al menos que estas sean **final**.
- El único modificador que se puede aplicar a estas clases son **abstract** o **final**.

Clases internas anónimas.

- Las clases internas anónimas no tienen nombre, y su tipo deberá ser una subclase de un tipo nombrado o una implementación de una interfaz nombrada. Veamos el siguiente ejemplo para entender lo mencionado.

Ejemplo:

```
class Popcorn {
    public void pop() {
        System.out.println("popcorn");
    }
}

class Food {
    Popcorn p = new Popcorn() {
        public void pop() {
            System.out.println("anonymous popcorn");
        }
    };
}
```

Se observa a la clase Popcorn, el cual tiene el método `pop()`. En la clase Food se declara una variable de tipo Popcorn, pero el objeto que se asigna no es directamente de tipo Popcorn, observar que se está redefiniendo la clase Popcorn, se está sobre escribiendo el método `pop()`, es decir se está creando una sub clase

de Popcorn(todo el código en amarillo corresponde a la sub clase), pero la sub clase que se esta creando no tiene nombre, a esto denominamos clase interna anónima.

- Una clase interna anónima siempre es creada como parte de una sentencia; se debe cerrar la sentencia después de definir la clase, es decir "};".
- Una clase interna anónima puede extender una clase o implementar una interface. No ocurre lo mismo con clases no anónimas (inner u otra), una clase anónima interna no puede ambos a la vez. Es decir no puede extender una clase e implementar una interface, no puede implementar mas de una interface.
- Un clase interna argument-defined es declarado, definido, y automáticamente instancia como parte de una invocación al método. La clave para recordar es que la clase se define dentro de un argumento del método, por lo que la sintaxis se pondrá fin a la definición de la clase con una llave, seguido de un paréntesis de cierre para terminar la llamada al método, seguido por un punto y coma al final de la declaración: });

Clases estáticas anidadas.

- La clase estática anidada son clases inner que se marcan con el modificador **static**. Ejemplo:

```
class MyOuter {
    private int x = 7;

    // inner class definition
    static class MyInner {
        public void seeOuter() {
            System.out.println("Outer x is " + x);
        }
    } // close inner class definition
} // close outer class
```

- Una clase estática anidada no es propiamente una clase inner, es una clase anidada superior, la forma de declaración puede generar confusión.
- Porque la clase anidada es estática, no se comparte ninguna relación especial con la instancia de la clase outer. No se requiere una instancia de la clase outer para instanciar una clase estática anidada.
- Para instanciar una clase estática anidada se requiere el nombre de la clase outer y de la clase anidada. Ejemplo:
BigOuter.Nested n = new BigOuter.Nested();
- Estas clases no pueden acceder a miembros no estáticos de la clase outer, es decir las instancias de la clase estática anidada no obtienen un referencia **this** de la clase outer.

2.5. Use enumerated types

- Un **enum** especifica una lista de valores constantes que se pueden asignar a un tipo.

Ejemplo:

```
public enum TipoDocumento {
    DNI, CARNET, RUC, PASAPORTE
}
```

- Las constantes del **enum** no son **String** o **int**, el tipo de la constante es del tipo de **enum**.

Por ejemplo DNI y CARNET son del tipo **enum** TipoDocumento.

- Un **enum** puede ser declarado fuera o dentro de una clase, pero no dentro de un método.

- Un **enum** declarado fuera de una clase no se puede marcar como **static**, **final**, **abstract**, **protected** o **private**.

- Un **enum** también puede contener constructores, métodos, variables y constantes del enum.

- Una constante **enum** puede enviar argumentos al constructor del **enum**.

Ejemplo:

```
public enum EstadolImagen{  
    VACIO(0,0), PEQUENA(320,200), MEDIANA(800,600), GRANDE(1024,768);  
  
    private final int ancho;  
    private final int alto;  
  
    EstadolImagen(int ancho, int alto){  
        this.ancho=ancho;  
        this.alto=alto;  
    }  
    ...  
}
```

- El constructor de un **enum** puede tener argumentos y puede ser sobrecargado.

- El constructor de un **enum** nunca se puede invocar directamente dentro del código.

Ellos se invocan automáticamente cuando el **enum** es inicializado.

- El punto y coma (;) al final de la declaración del **enum** es opcional. Estas 2 declaraciones son legales:

enum TipoDocumentos{*DNI, CARNET, RUC, PASAPORTE*};
enum TipoDocumentos{*DNI, CARNET, RUC, PASAPORTE*}

- **TipoDocumentos.values()** retorna un array de valores de **TipoDocumentos**.

Ejemplo:

```
System.out.println(TipoDocumentos.values()[1]);
```

Muestra en consola: CARNET

- Cada valor del enum puede tener un cuerpo similar a una clase. Ejemplo:

enum TipoDocumentos{

DNI{

```
    int getDigitoCheck(){  
        return ...;  
    }
```

}

, CARNET, RUC, PASAPORTE}

- ❑ Un enum y su valor se pueden comparar con == o equals ambos darán true.

3. Principios del diseño orientado a objetos

3.1. Write code that declares, implements and/or extends interfaces

- Los métodos pueden ser sobre escritos y sobre cargados; los constructores pueden ser sobre cargados pero no sobre escritos.
- Con respecto al metodo que es sobre escrito, el metodo que sobre escribe:
 - Deberá tener la misma lista de argumentos.
 - Deberá tener el mismo tipo de retorno, pero desde Java 5, el tipo de retorno puede ser una sub clase, a esto se conoce como retorno covariant.
 - No deberá tener un modificador de acceso mas restrictivo.
 - Puede tener un modificador de acceso menos restrictivo.
 - No deberá lanzar nuevas o más amplias excepciones checked.
 - Puede lanzar algunas o más estrechas excepciones checked o cualquier excepción no checked.
- Una sub clase usa super.overriddenMethodName() para invocar a la versión del metodo en la súper clase.
- Sobre carga significa reusar el nombre del metodo pero con diferentes argumentos.
- Los métodos sobre cargados
 - Deberá tener una lista diferente de argumentos.
 - Puede tener un diferente tipo de retorno, si la lista de argumento son diferentes.
 - Puede tener diferente modificado de acceso.
 - Puede lanzar excepciones diferentes.
- Los métodos de una súper clase pueden ser sobre cargados en una sub clase.
- Polimorfismo aplica para sobre escritura y no para sobre carga.
- Object type (not the reference variable's type), determines which overridden method is used at runtime.
El tipo del objeto (Cuando la variable es diferente tipo), determina cual

- Reference type determines which overloaded method will be used at compile time.

3.2. Choose between interface inheritance and class inheritance

- Polimorfismo significa varias formas.
- Un objeto puede ser referenciado por variables de diferentes tipos, siempre y cuando todos esos tipos sean supertipos (super clases o interfaces) del tipo original del objeto.

- El tipo de la variable de referencia (cuando es diferente al tipo del objeto) determina que miembros se pueden llamar.

- Hay dos tipo de casting de variables de referencia: downcasting y upcasting.

- Downcasting: Ocurre en el siguiente escenario.
Si: Clase1 extends Clase2

//Tenemos el siguiente caso:

```
Clase2 x=new Clase1();
```

```
Clase1 y;
```

```
y=(Clase1)x
```

- Upcasting: Ocurre en el siguiente escenario.

Si: Clase1 extends Clase2

//Tenemos el siguiente caso:

```
Clase2 x;
```

```
Clase1 y=new Clase1();
```

//De forma implícita.

```
x=y;
```

//De forma explícita.

```
x=(Clase2)y;
```

3.3. Apply cohesion, low-coupling, IS-A, and HAS-A principles

- La herencia permite a una clase ser una sub clase de una súper clase y de esa forma heredar los miembros de la súper clase.
- La herencia es un concepto clave que esta relacionado a IS-A, polimorfismo, sobre escritura, sobre carga y casting.
- Todas las clases(excepto la clase Object) son sub clases de Object.
- El encapsulamiento ayuda a ocultar la implementación detrás de una interface (API).
- El código encapsulado tiene 2 características:
 - Las variables de instancia son protegidas(usualmente con el modificador **private**)
 - Los métodos get y set son proporcionados para acceder a las variables de instancia.
- IS-A se refiere a herencia o implementación.
- IS-A es expresado con el palabra reservada **extends**
- IS-A “hereda de” y “es un sub tipo de” son las expresiones equivalentes.
- HAS-A significa que una instancia de una clase “tiene una” referencia a una instancia de otra clase u otra instancia de la misma clase.

- ❑ Acoplamiento se refiere al grado con el cual una clase conoce acerca o usa miembros de otra clase.
- ❑ Bajo acoplamiento es estado deseable de tener clases que son bien encapsuladas, minimizando las referencias entre estas y militando el uso del "API".
- ❑ Alto acoplamiento es el estado no deseado de tener clases que rompen las reglas del bajo acoplamiento.
- ❑ Cohesión se refiere al grado en el cual una clase tiene un simple, bien definido rol o responsabilidad.
- ❑ Cohesión alta es estado deseable de una clase cuyos miembros soportan un simple bien definido rol o responsabilidad.
- ❑ Cohesión baja es estado no deseable de una clase cuyos miembros soportan múltiples no definidos roles o responsabilidades.

3.4. Apply object composition principles (including has-a relationships)

3.5. Design a class using a Singleton design pattern

3.6. Write code to implement the Data Access Object (DAO) pattern

3.7. Design and create objects using a factory pattern

4. Genericos y colecciones

4.1. Create a generic class

- ❑ Generic deja cumplir en tiempo de compilación el tipo correcto en Colecciones u otras clases y declaraciones de métodos usando parametros de tipo genérico.
- ❑ Un ArrayList<Animal> puede aceptar referencias de tipo Dog, Cat o cualquier sub tipo de Animal (Sub clase o implementación en caso Animal sea una interface).
- ❑ Cuando se usa colecciones genéricas, no se requiere un casteo al momento de obtener los elementos según el tipo de objeto. Si no se usa colecciones genéricas si se requiere un cast.

Ejemplo:

```
List<String> gList = new ArrayList<String>();  
List list = new ArrayList();  
// more code  
String s = gList.get(0); // No se require casteo  
String s = (String)list.get(0); // Se require casteo.
```

- ❑ Se puede pasar una colección generica en un metodo que toma una colección no generica, pero el resultado puede ser errado. El compilador no podrá saber el tipo correcto de la colección.
- ❑ El compilador puede reconocer código que puede ser inseguro.
Por ejemplo si se pasa un List<String> en el siguiente metodo:
void foo (List aList) { aList.add(anInteger);}
Se obtiene una advertencia porque el add() es potencialmente "inseguro".
- ❑ "Compilar sin errores" no es lo mismo que "compilar sin advertencias"
compilar con advertencias no se consideran errores de compilación o fallas.
- ❑ La información de tipo genérico no existe en tiempo de ejecución- esta es para tiempo de compilación únicamente. Mezclar código genérico con código legado puede crear código compilado que genere una excepción en tiempo de ejecución.
- ❑ Las reglas de asignación del polimorfismo solo aplican a los tipos base, no a parametros de tipos genéricos. Ejemplo:

```
List<Animal> aList = new ArrayList<Animal>(); // Correcto  
List<Animal> aList = new ArrayList<Dog>(); // Incorrecto  
void foo(List<Animal> aList) { } // No se le puede pasar un List<Dog>  
List<Animal> bar() { } // No puede retornar un List<Dog>
```

- ❑ Usar wilcards en las declaraciones de métodos genéricos permite aceptar subtipos o supertipos del tipo declarado en el metodo.
void addD(List<Dog> d) {} // Solo acepta tipos <Dog>
void addD(List<? extends Dog>) {} // Puede aceptar un a <Dog> o un <Beagle>
- ❑ La palabra **extends** que acompaña al wildcard se usa para especificar un extends o implements. Entonces en <? extends Dog>, Dog puede ser una clase o interface.

- Cuando se usa un wildcard <? extends Dog>, la colección puede ser accedida pero no se puede agregar nuevos elementos, si se puede eliminar elementos.
- Cuando se usa un wildcard List<?>, cualquier tipo genérico se puede asignar a la referencia, la colección puede ser accedida pero no se puede agregar nuevos elementos, si se puede eliminar elementos.
- List<Object> se refiere únicamente a List<Object>, mientras que List<?> o List<? extends Object> puede contener cualquier tipo de objeto.

□ El siguiente tipo de declaración solo permite almacenar nulls.

```
List<? extends Object> lista= new ArrayList<String>();
```

```
List<? > lista= new ArrayList<String>();
```

```
lista.add(null);
```

□ Por convenciones de declaración de genéricos se usa:

T para tipos (no colecciones).

E para elementos de colecciones.

K para llaves dentro de Map.

V es un valor, ejemplo un valor de retorno o un valor en un Map.

Ejemplos:

```
public interface List<E> // Declaración de la API List.
```

```
boolean add(E o) // Declaración de List.add().
```

```
class Foo<T> { } // Una clase.
```

□ El identificador de tipo genérico puede ser usado en una declaración de clase, metodo o variable.

```
class Foo<T> { } // Una clase.
```

```
T anInstance; // Una variable de instancia.
```

```
Foo(T aRef) {} // El argumento de un constructor.
```

```
void bar(T aRef) {} // El argumento de un metodo.
```

```
T baz() {} // Un tipo de retorno.
```

El compilador sustituirá el tipo correcto.

□ Se puede ser mas de un tipo parametrizado dentro de una declaración:

```
public class UseTwo<T, X>{}
```

□ Se puede declarar un metodo genérico usando un tipo no definido dentro de la clase:

```
public <T> void makeList(T t) { }
```

T no es el tipo de retorno, porque el metodo es **void**, pero para usar T dentro del argumento del metodo se debe declarar <T>, el cual pasa antes que retorne el metodo.

4.2. Use the diamond for type inference

Bla...bla...

4.3. Analyze the interoperability of collections that use raw types and generic types

Bla...bla...

4.4. Use wrapper classes, autoboxing and unboxing

Bla...bla...

4.5. Create and use List, Set and Deque implementations

- Las actividades comunes con colecciones incluye agregar objetos, remover objetos, verificar que se incluya el objeto, retornar un objeto e iterar sobre la colección.
- Tres significados para **Collection**:
 - **Collection** representa una estructura de datos en la cual se almacena objetos.
 - **Collection** es una interface de java.util de la cual extiende Set y List.
 - **Collections** es una clase donde todos sus métodos son static y son útiles para trabajar colecciones.
- List, Set, Map y Queue son las cuatro colecciones básicas.
- **List, Set y Queue** extienden Collection. **Map** no extiende de Collection pero si es parte del **Java Collection Framework**.
- **List** mantiene los elementos en Ordered (Quedan en el orden en el que se agregan), permite duplicado y se busca por un índice.
- **Set** puede ser ordered o sorted (La colección ordenará los objetos de forma automática), no se permite duplicados.
- **Map** almacena los objetos con una llave; puede ser ordered o sorted, no se permiten llaves duplicadas.
- **Queue** almacena objetos en ordered por FIFO (El primero en ingresar es el primero en salir) o por prioridad.
- También se puede clasificar a las colecciones según: Sorted, Unsorted, Ordered, Unordered.
- Cuando es **ordered** se itera a través de la colección en un orden especificado no aleatorio.
- Cuando es **sorted** se itera a través de la colección mediante un ordenamiento especificado.
- **Sorted** puede aplicar según el orden alfabético, numérico o definido por programación.
- **ArrayList**: Rápida iteración y rápido acceso aleatorio.

- **Vector:** Es similar a un ArrayList lento, debido a que tiene sus métodos sincronizados.
- **LinkedList:** Es similar a una ArrayList, pero se implementa utilizando una lista doblemente enlazada, es decir tienes diferentes implementaciones.
Usos frecuentes de LinkedList:
 - Cuando se requiere eficiencia al remover o agregar elementos que están entre otros elementos o al inicio/final.
 - Cuando no se requiere eficiencia en el acceso aleatorio a los elementos y se desea iterar sobre estos uno a uno.
 - Se usa para implementar pilas y colas.
- **Set:** no puede contener elementos duplicados, y no se tiene un explicito ordered de los elementos.
- **HashSet:** Asegura un tiempo constante para agregar, remover, ver el contenido o tamaño, no provee ordering, ni sorted. El Ordered es impredecible.
- **LinkedHashSet:** es una ordered versión de HashSet que mantiene una lista doblemente enlazada, se debe usar esta clase en vez de HashSet cuando se desea iterar por los elementos en ordered.
- **TreeSet:** Se itera sobre los elementos considerando sorted.
- **HashMap:** Actualización rápida (llave/valor); permite un **null** como llave y varios **null** como valores.
- **Hashtable:** Similar a un HashMap lento, debido a que sus métodos son sincronizados. No esta permito null en la llave o valor. En tiempo de ejecución sale NullPointerException al momento del put().
- **LinkedHashMap:** Iteracion rapida, la iteracion es por elementos en ordered. Permite una llave null y varios valores null.
- **TreeMap:** Es un map sorted según la llave.
- **PriorityQueue:** Para hacer una lista en ordered por prioridad de elementos.
 - La interface Queue tiene los siguientes métodos:
add(), agrega un elemento a la cola, retorna un **IllegalStateException** si la cola no tiene capacidad.
element(), retorna un elemento sin removerlo, retorna **NoSuchElementException** si la cola esta vacía.
offer(), agrega un elemento a la cola, similar a add(), pero retorna **false** si la cola no tiene capacidad.
peek(),retorna un elemento sin removerlo. Similar a element(), pero retorna **null** si la cola esta vacía.
poll(), retorna un elemento y lo remueve. Retorna null si la cola esta vacia.
remove(), similar a poll(), pero retorna **NoSuchElementException** si la cola esta vacía.

- Las colecciones solo pueden almacenar objetos, pero los primitivos pueden ser autoboxed.
- Se puede iterar con el **for** mejorado o con **hasNext()** y **next()**.
- **hasNext()** determina si existen mas elementos, entonces el **Iterator** no se moverá.
- **next()** retorna el siguiente elemento y mueve el **Iterator** hacia adelante.
- Para trabajar correctamente las llaves de un **Map** deben sobre escribir **equals()** y **hashCode()**.
- Queues usan **offer()** para agregar un elemento, **poll()** para remover la cabecera de la queue y **peek()** para revisar la cabecera de la queue.
- Desde Java 6, **TreeSet** y **TreeMap** tiene nuevos métodos de navegación similar a **floor()** y **higher()**.
- NavigableSet es una sub interface de SortedSet y provee métodos para buscar elementos. java.util.TreeSet es una implementación de NavigableSet.
- Ejemplo de uso de NavigableSet:

```
list<Integer> list= Arrays.asList(3,2,4,1,5);
NavigableSet<Integer> ns= new TreeSet<Integer>(list);
System.out.println("Por defecto es en orden ascendente:" + ns);
Iterator<Integer> descendingIterator = ns.descendingIterator();
StringBuilder sb= new StringBuilder("Orden descendiente: ");
while(descendingIterator.hasNext()){
    int m=descendingIterator.next();
    sb.append(m+ " ");
}
System.out.println(sb);

int greatest = ns.lower(3);
System.out.println("Menor de 3="+greatest);

int smallest=ns.higher(3);
System.out.println("Mayor de 3="+smallest);
```

Salida:

Por defecto es en orden ascendente:[1, 2, 3, 4, 5]
Orden descendiente: 5 4 3 2 1
Menor de 3=2
Mayor de 3=4

- NavigableMap es una sub interface de SortedMap y provee métodos para buscar elementos. java.util.TreeMap es una implementación de NavigableMap.
- Desde Java 5 se agregó el paquete **java.util.concurrent**, el cual trae un conjunto de interfaces y clases que facilitan el trabajo con colecciones que están siendo trabajadas por varios hilos a la vez, una interface importante de este paquete es **ConcurrentMap**.

- La interface **ConcurrentMap** hereda de **Map**, es cual es capaz de manejar accesos concurrentes (get y put) en este.
- **ConcurrentMap** provee los siguientes métodos:
putIfAbsent(key, value), Si la llave no existe agrega la relación en el mapa, si existe obtiene el valor.
remove(key, value), remueve la relación si existe la llave y el valor en el mapa.
replace(key, value), reemplaza la relación en el mapa.
- ConcurrentHashMap es una implementación de ConcurrentMap, es similar a Hashtable, pero se diferencia de esta en los bloqueos, Hashtable bloque todo el mapa cuando se esta leyendo, ConcurrentHashMap solo bloque la fila que esta escribiendo en ese momento.
- ConcurrentNavigableMap es similar al NavigableMap.

4.6. Create and use Map implementations

Bla...bla...

4.7. Use java.util.Comparator and java.lang.Comparable

- Sorting puede ser en el orden natural de los elementos, también puede ser via Comparable o varios Comparator.
- Implementar **Comparable** usando el metodo **compareTo()**, provee únicamente un ordenamiento.
- Crear varios **Comparator** para implementar varios ordenamientos de clase, implementar **compare()**.
- Para poder ordenar o buscar en una lista de elementos esta debe ser **Comparable**.
- Para poder buscar en un array o lista esta debe ser primero ordenada.
- Las clases **Collections** y **Arrays** del java.util tienen los siguientes métodos:
 - **A sort()** ordena usando Comparator o un orden natural.
 - **binarySearch()** busca sobre un pre ordenado array o List.
- Las búsquedas binarias se pueden usar según:
`Collections.binarySearch(List<? extends Comparable<? super T>> list, T key)`
`Collections.binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`
- **Arrays.asList()** Crea un List de los elementos del array y vincula a estas.
- **Collections.reverse()** revierte el orden de los elementos en el List.
- **Collections.reverseOrder()** retorna un Comparator en orden inverso.

- List y Set tienen el metodo toArray() para crear arrays.

4.8. Sort and search arrays and lists

Bla...bla...

5. Procesamiento de String

5.1. Search, parse and build strings (including Scanner, StringTokenizer, StringBuilder, String and Formatter)

- ❑ Los objetos String son inmutables (No se pueden modificar), pero las variables de referencia a un objeto String no lo son.
- ❑ Si se crea un String nuevo sin ser asignado a alguna variable, este se perderá del programa.
- ❑ Si se direcciona la referencia de un String a un nuevo String, el antiguo String puede ser perdido.
- ❑ Los métodos de la clase String basan sus índices iniciando en cero, excepto el segundo parámetro de substring().
- ❑ La clase String es final, sus métodos no se puede sobre escribir.
- ❑ Cuando la JVM encuentra un String literal, este se agregar al pool de String literal.
- ❑ String tiene un metodo length(); y los arrays tienen un atributo length.
- ❑ El API StringBuffer es lo mismo que el nuevo API StringBuilder, excepto que los métodos de StringBuilder no son sincronizados para thread safety.
- ❑ Los métodos de StringBuilder deben correr mas rápidos que los métodos de StringBuffer.
- ❑ Todos los siguiente puntos aplican a StringBuffer y StringBuilder:
 - ❑ Ambos son mutables, ellos pueden cambiar sin tener que crear un objeto nuevo.
 - ❑ Los métodos de StringBuffer actúan directo sobre el objeto que los invoca, y pueden cambiar al objeto, sin necesidad de una asignación explícita en la sentencia.
 - ❑ equals() de StringBuffer() no esta sobre escrito, este no compara valores.
- ❑ Se debe recordar que una cadena de métodos se evalúan de izquierda a derecha.
- ❑ Los siguiente métodos de String son los mas importantes: charAt(), concat(), equalsIgnoreCase(), length(), replace(), substring(), toLowerCase(), toString(), toUpperCase() y trim().
- ❑ Los siguiente métodos de StringBuffer son los mas importantes: append(), delete(), insert(), reverse() y toString()

5.2. Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to: . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ()

- ❑ regex significa expresiones regulares, los cuales son patrones usados para buscar algunos datos dentro de algún dato largo.
- ❑ regex es un sub lenguaje que existe en Java y otros lenguajes, por Ejemplo Perl.

- ❑ regex deja buscar para patrones usando cadenas de caracteres o metacaracteres. Metacaracteres permiten buscar de una forma mas abstracta por ejemplo "un digito" o "un espacio en blanco".

- ❑ Algunos Metacaracteres son:

\d	Un digito [0-9]
\s	Un espacio en blanco [\t\n\x0B\f\r]
\w	Un carácter [a-zA-Z_0-9]
.(punto)	Cualquier carácter

- ❑ Regex permite especificar cuantificadores, entonces es posible especificar situaciones como: "Considerar 1 o más dígitos en una fila".

- ❑ Algunos Cuantificadores son:

?	Uno o nada
*	Cero o varios
+	Uno o varios

- ❑ No se puede mezclar Metacaracteres y cadenas al menos que se usen caracteres de escape. Por ejemplo String s="\d".

- ❑ Las clases **Pattern** y **Matcher** de Java son muy potentes para las capacidades de regex. Están en el paquete **java.util.regex**.

- ❑ Métodos importantes de Pattern
compile(), Compila la regex a un Pattern.

- ❑ Métodos importantes de Matcher
matches(), Intenta comparar la región entera otra vez con el pattern.
pattern(), Retorna el Pattern que fue interpretado por este Matcher.
find(), Intenta encontrar la siguiente secuencia que cumple con el pattern dentro de la cadena original.
start(), Retorna el índice de inicio de una comparación previa.
group(), Retorna la secuencia que coincide con la comparación previa.

- ❑ Se puede usar la clase **java.util.Scanner** para hacer búsquedas simples de regex, pero su intención principal es para usar tokens.

- ❑ Tokenizing es el proceso de dividir data delimitada en partes pequeñas.

- ❑ En tokenizing la data que se desea es llamada token y la cadena de separación de los tokens se llama delimitador.

- ❑ Tokenizing puede hacer con la clase **Scanner** o con el metodo **String.split()**.

- ❑ Los delimitadores pueden ser caracteres simples como comas o expresiones complejas regex.

- ❑ La clase Scanner permite navegar por los tokens mediante un loop, el cual permite parar donde uno desee.

El delimitador por defecto es el espacio en blanco.

Ejemplo:

Lee cada una de las palabras de un archivo de texto.

```
File file = new File(fileName);
Scanner scanner=new Scanner(file);
while(scanner.hasNext()){
    System.out.println(scanner.next());
}
scanner.close();
```

Para entrada de datos enteros de la consola.

```
Scanner sc=new Scanner(System.in);
int i=sc.nextInt();
```

Lee solo longs de un archivo de texto.

```
Scanner sc = new Scanner(fileName);
while(sc.hasNextLong()){
    long aLong=sc.nextLong();
}
```

- La clase Scanner permite que se tokenize Strings, streams o archivos.

- The `String.split()` method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.

El metodo `String.split()` tokenizes.

- Desde Java 5, hay dos métodos que se pueden usar para formatear salidas a consola, estos son `format()` y `printf()`. Estos son métodos de la clase `PrintStream`, una instancia de esta clase es el **out** en `System.out`.

- La especificación de formatos generales, caracteres y números, tiene la siguiente sintaxis:

`%[argument_index$][flags][width][.precision]conversion`

- El metodo **format()** y **printf()** son idénticos funcionalmente.

- Formatear data con **printf()** (o **format()**) se lleva a cabo utilizando cadenas de formatos que están asociadas con argumentos primitivos o de cadena.

- El metodo **format()** permite mezclar literales dentro de la cadena de formato.

- Algunos valores para formatear cadenas:

- Flags:

-	El resultado se justificara a la izquierda.
+	El resultado siempre incluirá un signo
0	El resultado usara al 0 para padded (Rellenar)
,	El resultado incluirá especificación locale
(El resultado encerrara el numero negativo en paréntesis.

□ Conversiones: b, c, d, f, y s

b, B	Si el argumento es null entonces el resultado es “ false ”. Si el argumento es boolean o Boolean entonces el resultado es generado por String.valueOf(). En cualquier otro caso el resultado es “ true ”
c, C	El resultado es un carácter Unicode
d,	El resultado es formateado como entero decimal.
f	El resultado es formateado como numero decimal.
js, S	Si el argumento es null entonces el resultado es “ null ”. Si el argumento implementa Formattable , entonces se invoca a argumento.formatTo. En cualquier otro caso el resultado se obtiene por invocar a argumento.toString().

- Si los caracteres de conversión no concuerdan con el tipo de argumento, se genera una excepción.

- Índices de los argumentos.

Se pueden especificar según:

* **Índice explícito**, hace referencia a la posición del argumento, se inicia por 1\$, un argumento se puede referenciar más de una vez.

Ejemplo:

```
formatter.format("%4$s %3$s %2$s %1$s %4$s %3$s %2$s %1$s","a", "b", "c", "d");
//Salida: "d c b a d c b a"
```

* **Índice ordinario**, Se usa según el orden de los argumentos.

```
formatter.format("%s %s %s %s", "a", "b", "c", "d")
//Salida: "a b c d"
```

5.3. Format strings using the formatting parameters: %b, %c, %d, %f, and %s in format strings.

6. Excepciones y aserciones

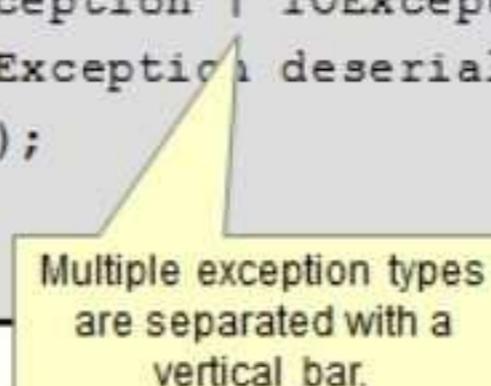
6.1. Uso de la sentencia throw y throws

- Con JSE 7 se pueden lanzar excepciones precisas, observar el siguiente código, en JSE 6 da error de compilación

```
public static int readByteFromFile() throws IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    } catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
}
```

6.2. Desarrollando código que maneja múltiples tipos de excepciones en un solo bloque catch

```
ShoppingCart cart = null;
try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new ObjectInputStream(is)) {
    cart = (ShoppingCart)in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " + cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

 Multiple exception types are separated with a vertical bar.

- Las excepciones que se colocan dentro del catch no pueden tener relación de herencia

6.3 Desarrollar código que use la sentencia try-con-recursos (Incluido el uso de que clases que implementen la interface AutoCloseable)

* Con la sentencia try-con-recursos ya no se necesita incluir la sentencia **finally**

```
System.out.println("About to open a file");
try (InputStream in =
        new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

- * Puedo instanciar varios recursos separados por ";" al momento de cerrarse se hacen en orden inverso al que fueron declarados
- * Si ocurre una excepción mientras se crea el recurso AutoCloseable el control pasa el bloque catch
- * Si ocurre una excepción en el cuerpo del bloque try, todos los recursos serán cerrados y luego pasa el control al bloque catch y si al momento de cerrar esos recursos ocurre una excepción, estas excepciones serán suprimidas, puedo obtener la excepciones suprimidas, ver ejemplo

```
} catch(Exception e) {
    System.out.println(e.getMessage());
    for(Throwable t : e.getSuppressed()) {
        System.out.println(t.getMessage());
    }
}
```

- * Si no ocurre una excepción en el bloque try, pero al momento de cerrar los recursos ocurre un error, el control salta al bloque catch
- * Es necesario que la clase del recurso implemente java.lang.AutoCloseable o java.io.Closeable

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

- * **Closeable** extiende a **AutoCloseable**
- * En **Closeable** el metodo close() por definición es idempotente, en **AutoCloseable** no es necesario

6.4. Creando excepciones personalizadas

- * Se crean extendiendo la clase **Exception** o alguna de sus sub clases

```
public class DAOException extends Exception {  
  
    public DAOException() {  
        super();  
    }  
  
    public DAOException(String message) {  
        super(message);  
    }  
}
```

6.5. Probando invariantes usando aserciones

```
assert <boolean_expression>;  
  
assert <boolean_expression> : <detail_expression>;  
  
* Invariantes internas
```

```
1  if (x > 0) {  
2      // do this  
3  } else {  
4      assert (x == 0);  
5      // do that, unless x is negative  
6  }
```

* Control de flujos invariantes

```
1  switch (suit) {  
2      case Suit.CLUBS: // ...  
3          break;  
4      case Suit.DIAMONDS: // ...  
5          break;  
6      case Suit.HEARTS: // ...  
7          break;  
8      case Suit.SPADES: // ...  
9          break;  
10     default: assert false : "Unknown playing card suit";  
11         break;  
12 }
```

* Post condiciones y clases invariantes

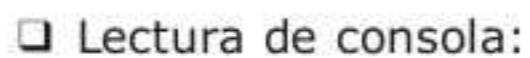
```
1 public Object pop() {
2     int size = this.getElementCount();
3     if (size == 0) {
4         throw new RuntimeException("Attempt to pop from empty stack");
5     }
6
7     Object result = /* code to retrieve the popped element */ ;
8
9     // test the postcondition
10    assert (this.getElementCount() == size - 1);
11
12    return result;
13 }
```

7. Fundamentos de Java I/O

7.1. Lectura y escritura de datos de consola



- Salida estándar : System.out es una instancia de PrintStream
 Entrada estándar : System.in es una instancia de InputStream
 Salida de error estándar : System.err es una instancia de PrintStream



```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 public class KeyboardInput {
5     public static void main(String[] args) {
6         try (BufferedReader in =
7             new BufferedReader (new InputStreamReader (System.in))) {
8             String s = "";
9             // Read each input line and echo it to the screen.
10            while (s != null) {
11                System.out.print("Type xyz to exit: ");
12                s = in.readLine();
13                if (s != null) s = s.trim();
14                System.out.println("Read: " + s);
15                if (s.equals ("xyz")) System.exit(0);
16            }
17        } catch (IOException e) {
18            System.out.println ("Exception: " + e);
19        }
20    }
  
```

Chain a buffered reader to an input stream that takes the console input.

- Un objeto de la clase java.io.Console, le cual se puede obtener mediante System.console(), permite capturar entradas desde la consola con echo o sin echo (echo se refiere a mostrar en pantalla lo que se esta ingresando por teclado), en el caso de sin echo puede ser para capturar algún valor secreto por ejemplo un Password, para esto se puede usar el metodo console.readPassword().

```

10 Console cons = System.console();
11 if (cons != null) {
12     String userTyped; String pwdTyped;
13     do {
14         userTyped = cons.readLine("%s", "User name: ");
15         pwdTyped = new String(cons.readPassword("%s", "Password: "));
16         if (userTyped.equals("oracle") && pwdTyped.equals("tiger")) {
17             userValid = true;
18         } else {
19             System.out.println("Wrong user name/password. Try again.\n");
20         }
21     } while (!userValid);
22 }
  
```

readPassword does not echo the characters typed to the console.

7.2. Uso de las clases del paquete java.io incluido: BufferedReader, BufferedWriter, File, FileReader, FileWriter, DataInputStream, DataOutputStream, ObjectOutputStream, ObjectInputStream, y PrintWriter

- Un nuevo objeto **File** no significa que exista un nuevo archivo en el sistema de archivos.
- Un objeto **File** puede representar a un archivo o a un directorio.
- La clase **File** deja manejar (Crear, renombrar y eliminar) archivos y directorios.
- Los métodos `createNewFile()` y `mkdir()` crean entradas en el sistema de archivos.
- Las clases `FileWriter` y `FileReader` son de bajo nivel para I/O. Se puede usar estas para escribir y leer archivos, pero ellas deberán ser utilizadas usualmente mediante clase wrapper.
- Las clases del paquete `java.io` están diseñadas para ser encadenadas o usadas mediante clases wrappers (Esto es un común uso del patrón de diseño Decorator)
- Es muy común usar una clase `BufferedReader` como wrapper de la clase `FileReader` y `BufferedWriter` como wrapper de la clase `FileWriter`, para acceder a métodos de alto nivel, los cuales son mas prácticos.
- PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.
- La clase **PrintWriter** tiene métodos: `append()`, `format()` y `printf()`.
- Lectura y escritura de objetos:

```

1 public static void main(String[] args) {
2     Stock s1 = new Stock("ORCL", 100, 32.50);
3     Stock s2 = new Stock("APPL", 100, 245);
4     Stock s3 = new Stock("GOGL", 100, 54.67);
5     Portfolio p = new Portfolio(s1, s2, s3);
6     try (FileOutputStream fos = new FileOutputStream(args[0])) {
7         ObjectOutputStream out = new ObjectOutputStream(fos)) {
8             out.writeObject(p);           // The writeObject method writes the
9         } catch (IOException i) {      // object graph of p to the file stream.
10            System.out.println("Exception writing out Portfolio: " + i);
11        }
12        try (FileInputStream fis = new FileInputStream(args[0])) {
13            ObjectInputStream in = new ObjectInputStream(fis)) {
14                Portfolio newP = (Portfolio)in.readObject(); // The readObject method
15            } catch (ClassNotFoundException | IOException i) { // restores the object from
16                System.out.println("Exception reading in Portfolio: " + i);
17            }

```

Portfolio is the root object.

The writeObject method writes the object graph of p to the file stream.

The readObject method restores the object from the file stream.

8. Java File I/O (NIO.2)

- 8.1. Operate on file and directory paths with the Path class
- 8.2. Check, delete, copy, or move a file or directory with the Files class
- 8.3. Read and change file and directory attributes, focusing on the BasicFileAttributes, DosFileAttributes, and PosixFileAttributes interfaces
- 8.4. Recursively access a directory tree using the DirectoryStream and FileVisitor interfaces
- 8.5. Find a file with the PathMatcher interface
- 8.6. Watch a directory for changes with the WatchService interface

9. Desarrollando aplicaciones con acceso a base de datos usando JDBC

- 9.1. Describe the interfaces that make up the core of the JDBC API (including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations)
- 9.2. Identify the components required to connect to a database using the DriverManager class (including the jdbc URL)
- 9.3. Submit queries and read results from the database (including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections)

9.4. Uso de transacciones en JDBC (Incluido el desactivado del modo auto-commit, commit y rollback de transacciones)

- * Por defecto cada operación SQL en JDBC es una transacción independiente y al terminar se realiza un commit
- * Desactivando el auto-commit: `con.setAutoCommit(false)`
- * Invocando al commit: `con.commit()`
- * Invocando al rollback: `con.rollback()`

- 9.5. Construct and use RowSet objects using the RowSetProvider class and the RowSetFactory interface

9.6. Creando y usando objetos PreparedStatement y CallableStatement

- * PreparedStatement

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

- * CallableStatement

```
CallableStatement cStmt
    = con.prepareCall("{CALL EmplAgeCount (?, ?)}");
int age = 50;
cStmt.setInt (1, age);
ResultSet rs = cStmt.executeQuery();
cStmt.registerOutParameter(2, Types.INTEGER);
boolean result = cStmt.execute();
int count = cStmt.getInt(2);
System.out.println("There are " + count +
    " Employees over the age of " + age);
```


10. Hilos

10.1. Create and use the Thread class and the Runnable interface

- ❑ Los hilos pueden ser creados por extender la clase **Thread** y sobre escribir el metodo **public void run()**.
- ❑ Los hilos también pueden ser creado por llamar al constructor de **Thread** que tiene como argumento a un **Runnable**. El objeto **Runnable** se dice que es el objetivo del hilo.
- ❑ Se puede llamar a **start()** en un objeto **Thread** solamente una vez. Si se invoca más de una vez se lanza **RuntimeException**.
- ❑ Es legal crear varios objetos **Thread** utilizando el mismo objeto **Runnable** como objetivo.
- ❑ Cuando un objeto **Thread** es creado no se inicia de forma automática, es necesario invocar al metodo **start()** para que se ejecute. Cuando un objeto **Thread** existe pero no se ha iniciado, este se encuentra en el estado **new**, el cual es diferente al estado **alive**.

10.2. Manage and control thread lifecycle

- ❑ Una vez que es iniciado una thread nueva, esta siempre entra al estado **runnable**.
- ❑ El thread scheduler puede mover una thread entre el estado **runnable** y el estado **running**.
- ❑ Para un maquina típica de un solo procesador, únicamente una thread puede estar en estado **running** en un tiempo, aunque varias threads pueden estar en estado **runnable**.
- ❑ No hay garantía en el orden en el cual las threads que han sido iniciadas, determine el orden en la cual ellas se ejecuten.
- ❑ No hay garantía que los threads se turnen de forma equivalente. Esto depende del thread scheduler y según lo determinado por cada implementación de JVM.
Si se desea una garantía que los threads se turnen de forma independiente a la JVM, se puede utilizar el metodo **sleep()**, esto impide que el thread acapare el proceso de ejecución mientras que otra thread no tiene asignación de proceso. Aunque en la mayoría de los casos basta con invocar al metodo **yield()** para que los threads trabajen bien juntos.
- ❑ Una thread que está en estado **running** puede entrar al estado **blocked/waiting** por invocación al metodo **wait()**, **sleep()** o **join()**.
- ❑ Una thread en estado **running** puede entrar al estado **blocked/wating** porque no puede adquirir el control de bloqueo de un bloqueo de código sincronizado.
- ❑ Cuando el **sleep()** o **wait()** a terminado, o un objeto bloqueado se convierte en disponible, el thread puede únicamente reingresar al estado **runnable**.
- ❑ Una thread en estado **dead** no puede ser iniciada otra vez.
- ❑ Se manda a dormir a la thread para demorar su ejecución por un periodo de tiempo y los bloqueos no son liberados durante este tiempo.

- ❑ Se garantiza que el tiempo para dormir de un thread sea por lo menos el tiempo que se indique en el argumento de `sleep()`, al menos que sea interrumpido. Pero no se garantiza en que momento esa thread volverá a estar en estado running.
- ❑ El metodo **sleep()** es **static** el cual provoca que la thread duerma, estando en estado running. Una thread no puede invocar el metodo **sleep()** de otro thread, la invocación siempre es desde la misma clase.
- ❑ El metodo **setPriority()** es utilizado en un objeto **Thread** para dar una prioridad de 1(Bajo) a 10(Alto), aunque las prioridades no son garantizadas y no todos los JVMs reconocen estos 10 niveles de prioridad, algunos niveles pueden ser tratados como iguales.
- ❑ Si no se modifica de forma explicita la prioridad de una thread, es igual a la prioridad de la thread que la creo.
- ❑ El metodo **yield()** puede causar que una thread deje de correr, si hay threads en estado runnable de la misma prioridad, no hay garantía de que esto pase, y no hay garantía que cuando el thread deje de correr y regrese al estado runnable, sea otra thread la que se coloque en estado running (Puede volver a ser la misma). Una thread puede ejecutar el `yield()` e inmediatamente reingrese al estado running.
- ❑ Lo más parecido a una garantía es que en cualquier tiempo dado cuando una thread esta corriendo usualmente no tiene una prioridad mas bajo que los threads que están en estado runnable. Si una thread de prioridad baja esta corriendo cuando una thread de prioridad alta entra al estado runnable, el JVM usualmente se adelanta a la ejecución de baja prioridad y colocara al thread de alta prioridad en running.
- ❑ Cuando una thread1 invoca al metodo **join()** de otra thread2, la thread1 que esta corriendo espera hasta que la thread2 se haya completado. Es decir el metodo **join()** le dice a la thread2 que queremos juntarnos al finalizar esta y que nos avise, entonces por el momento la thread1 puede entrar al estado runnable.

10.3. Synchronize thread access to shared data

- ❑ Los métodos **synchronized** permiten que solo una thread esta trabajando con ese metodo en un determinado momento.
- ❑ **synchronized** se utiliza para marcar metodos y bloques de código (es decir solo una parte del cuerpo de un metodo).
- ❑ Para sincronizar un bloque de código (Es decir solo una pequeña porción del cuerpo del metodo), se debe especificar un argumento al **synchronized**, el cual debe ser el objeto cuyo bloqueo queremos sincronizar.
- ❑ Si un objeto tiene partes que no están marcadas con **synchronized** esas si pueden ser accedidas por varias threads de forma simultanea.
- ❑ Cuando una thread es enviada a dormir, lo que tenga bloqueado no estará disponible para otras threads.
- ❑ Los métodos **static** pueden ser **synchronized**, utilizando el bloqueo de una instancia de **java.lang.Class** representando la clase.

- ❑ Un deadlock es cuando la ejecución de una thread se queda a la mitad y el código esta esperando que el bloqueo sea removido de algún objeto.
- ❑ Deadlocking puede ocurrir cuando un objeto bloqueado intenta acceder a otro objeto bloqueado el cual intenta accede al primer objeto bloqueada. En otras palabras, ambos threads están esperando que el otro libere el objeto bloqueado, de esta forma el bloqueo nunca se libera.
- ❑ Un Deadlocking es algo incorrecto, se debe evitar esto.
 - ❑ El metodo **wait()** permite decir a un hilo:
No hay nada que yo pueda hacer por ahora, entonces colócame en un pool de espera y notifícame cuando algo pase y yo este involucrado.
Básicamente un **wait()** significa:
Mantenme en espera en el pool o agrégame en la lista de espera.
 - ❑ El metodo **notify()** es usado para enviar una señal a una y solo a una de las threads que están esperando por el mismo objeto bloqueado.
 - ❑ El metodo **notify()** no puede especificar cual thread que espera será notificada.
 - ❑ El metodo **notifyAll()** trabaja igual que **notify()**, pero envía la señal a todos las threads que esperan sobre el objeto.
 - ❑ Los tres métodos: **wait()**, **notify()** y **notifyAll()** deberán ser invocados dentro de un contexto **synchronized**. Una thread invoca a un **wait()** o **notify()** sobre una objeto particular, el thread deberá sostener el bloqueo sobre el objeto.

10.4. Identify code that may not execute correctly in a multi-threaded environment.

Bla...bla

11. Concurrencia

- 11.1. Use collections from the `java.util.concurrent` package with a focus on the advantages over and differences from the traditional `java.util` collections.
- 11.2. Use `Lock`, `ReadWriteLock`, and `ReentrantLock` classes in the `java.util.concurrent.locks` package to support lock-free thread-safe programming on single variables.
- 11.3. Use `Executor`, `ExecutorService`, `Executors`, `Callable`, and `Future` to execute tasks using thread pools.
- 11.4. Use the parallel Fork/Join Framework

12. Localización

12.1. Read and set the locale by using the Locale object

12.2. Build a resource bundle for each locale

12.3. Call a resource bundle from an application

12.4. Format dates, numbers, and currency values for localization with the NumberFormat and DateFormat classes (including number format patterns)

- Las clases mas importante son:

java.util.Date, Su intención principal es representar fechas y tiempos.

java.util.Calendar, Es una clase abstracta que provee métodos que permiten trabajar con fechas y tiempos.

java.text.DateFormat, Es una clase abstracta que permite el formateo de fecha/tiempo.

java.text.NumberFormat, Es una clase abstracta que permite el formateo de números.

java.util.Locale, Clase para especificar una localización (Idioma y País).

- La mayoría de métodos de la clase Date no están vigentes.

□ Una fecha es almacenada como un long, el numero de milisegundos desde el 1 de Enero de 1970.

- Date objects are go-betweens the Calendar and Locale classes.

□ La clase **Calendar** provee con conjunto de métodos muy potentes para manipular fechas y realizar tareas tales como obtener días de la semana o agregar una cantidad de meses al año de una fecha.

□ Se crea instancias de **Calendar** usando el metodo estático de factoría **getInstance()**.

- Algunos metodo importantes de la clase **Calendar**

add(), permite agregar o substraer segundos, minutos, días, etc. a una fecha.

roll(), Similar a **add()**, pero no incrementa la parte grande de la fecha (Ejemplo: agregar 10 meses a una fecha que tiene como mes a Octubre la nueva fecha tendrá a Agosto como nueva fecha pero no incrementara el año)

□ Las instancias de **DateFormat** son creadas usando los métodos estáticos de factoría **getInstance()** y **getDateInstance()**.

□ Existen diversos formatos "estilos" disponibles en la clase DateFormat.

□ DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.

□ El metodo **DateFormat.format()** se usa para crear un String conteniendo la fecha formateada.

Ejemplo:

```
DateFormat formatter = DateFormat.getInstance();
String dateStr=formatter.format(new Date());
```

- La clase Locale es usado en conjunto con DateFormat y NumberFormat.

- ❑ Objetos de tipo DateFormat o NumberFormat pueden ser construidos con un específico Locale el cual es inmutable.

- ❑ Ejemplo de la creación de Locale:

```
// Un locale genérico para el idioma Ingles.  
Locale locale1 = new Locale("en");  
  
// Un locale para el idioma inglés en Canadá.  
Locale locale2 = new Locale("en", "CA");  
  
// Un locale para el idioma inglés en USA,  
// para Silicon Valley.  
Locale locale3 = new Locale("en", "US", "SiliconValley");
```

12.5. Describe the advantages of localizing an application

12.6. Define a locale using language and country codes