

Resumen de objetivos para el SCJP 6.0

Autor:

- Mayer Horna García

Con el fin de aprender todos, cualquier aporte a este documento reenviarlo a:

mayer.horna@gmail.com

Copyright © 2010

<CAP1: Declarations and Access Control>

<CERTIFICATION OBJECTIVES>

<OBJECTIVES>

<OBJECTIVE 1.3 AND 1.4: Identifiers & JavaBeans>

<SUMMARY>

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

1.4 Develop code that declares both static and non-static methods, and—if appropriate—use method names that adhere to the JavaBeans naming standards. Also develop code that declares and uses a variable-length argument list.

</SUMMARY>

1. Los **identificadores** únicamente pueden empezar por letras, guión bajo(_) o símbolo de dólar(\$).

Ejemplo de identificadores legales:

```
int _a;
int $c;
int _____2_w;
int _$;
int this_is_a_very_detailed_name_for_an_identifier;
```

Ejemplo de identificadores ilegales:

```
int :b;
int -d;
int e#;
int .f;
int 7g;

int double;
```

2. **Palabras reservadas:** Son todas las palabras propias del lenguaje y que no se pueden utilizar para declarar atributos, métodos, clases, etc. Son escritas en minúsculas:

abstract	continue	for	new	switch
assert ***	default	goto *	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum ****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp **	volatile
const *	float	native	super	while
* not used				
** added in 1.2				
*** added in 1.4				
**** added in 5.0				

</OBJECTIVE 1.3 AND 1.4: Identifiers & JavaBeans>

<OBJECTIVE 1.1: Declare Classes>

<SUMMARY>

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports)..

</SUMMARY>

3. Únicamente **puede** haber una clase pública por fichero java. El nombre del fichero debe coincidir con el de la clase pública. Puede haber más de una clase default en el mismo fichero.
4. En un fichero sin clases públicas, las clases no tienen restricción de nombre, no es necesario que coincida el nombre de una clase con el nombre del fichero.
5. Los comentarios pueden aparecer en cualquier parte del fichero java.
6. Las clases únicamente pueden ser **public** o **default** (no poner nada delante del class).
7. Niveles de acceso: (de mayor a menor restricción) -> **private** (acceso únicamente desde la propia clase), **default** (acceso desde la propia clase y desde el mismo paquete), **protected** (acceso desde la propia clase, paquete y subclase), **public** (acceso desde cualquier paquete). Nota: En Java, hay 4 niveles de acceso, pero solo 3 modificadores de acceso (**Ver punto 22**).
8. Las clases también pueden ser declaradas como **final** (no se puede extender de ella), **abstract** (no puede ser instanciada, solo extender de ella) y **strictfp** (los métodos de la clase seguirán las reglas del IEEE 754 para los números en coma flotante, solo se utiliza para clases o métodos, no para variables). Estos 3, son considerados como modificadores de clase (no de acceso). Pueden acompañar a cualquiera de los modificadores de acceso, p.e: **public final...**, pero no pueden combinarse entre ellos. Por ejemplo esto es incorrecto: **final abstract**. Sólo combinar **strictfp** con cualquier de éstos dos, Por ejemplo, esto es correcto : **final strictfp** o **public final strictfp**.
9. Métodos y atributos (miembros de una clase): Pueden tener los 4 niveles de acceso. Las variables de instancia son siempre inicializados a sus valores por defecto. Los métodos pueden tener **final** o **abstract**. Si tiene **final** significa que no va a poder ser sobreescrito por alguna subclase. Si es **abstract** significa que es un método abstracto (sin cuerpo, solo declaración), recordar que solo las clases abstractas pueden tener métodos **abstractos**. No existen variables de instancia, ni locales que puedan ser **abstractos**, esto solo se aplica a métodos. Las variables de instancia y locales si pueden tener **final**.

Ver imagen del punto 24 para que veas que modificador puede ser aplicado a variables y/o métodos.

10. El modificador **static** sirve para que un miembro (atributo o método) sea de clase, es decir, todos las instancias de esa clase compartirán el mismo valor de la variable (en caso de ser un atributo) o el mismo método.

En el caso de las variables es como si las consideráramos variables globales.

Puede estar antes o después del modificador de acceso, por ejemplo:

```
static public void main(String[] a)      es lo mismo que decir  
public static void main(String[] a)
```

11. Una clase únicamente puede heredar de otra (herencia simple) e implementar múltiples interfaces.

12. Siempre que se herede de una clase, se tendrá acceso sólo a todos los miembros declarados como **public**, **protected** y también se tendrá acceso a los miembros **default** si la superclase de la que extendemos está en el mismo paquete de la que se hereda. Si una clase hija en diferente paquete que el padre, instancia un objeto de la clase padre, no va a poder acceder a los miembros (variables de instancia y métodos) de la clase padre si estas están con modificador de acceso: **protected**

13. Todas las clases, incluyendo las abstractas, tienen al menos un **constructor**. Aunque no lo especifiquemos directamente, el compilador insertará el constructor por defecto en caso de que nosotros no insertemos uno. El constructor por defecto es de modificador de acceso **públic**, pero el programador puede poner cualquier otro modificador de acceso(private,protected,<default>).

14. Las clases abstractas no se pueden instanciar, es decir no se pueden crear objetos de esa clase con “new”.

15. Cuando se crea un objeto de una clase, se llamará a uno de sus constructores, con o sin argumentos en función de la instancia del objeto. En la primera línea de cada constructor el compilador insertará una llamada al constructor (**super()**) por defecto de la superclase (la superclase de todas las clases es Object). Únicamente se puede hacer una llamada a un constructor dentro de otro. **Si creamos una clase con un constructor que recibe algún argumento, el compilador ya no nos insertará el constructor por defecto.** Los constructores no se heredan.

</OBJECTIVE 1.1: Declare Classes>

<OBJECTIVE 1.1 AND 1.2: Declare Interfaces>

<SUMMARY>

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports).

1.2 Develop code that declares an interface. Develop code that implements or extends one or more interfaces. Develop code that declares an abstract class. Develop code that extends an abstract class.

</SUMMARY>

16. Las **interfaces** únicamente declaran sus métodos (no los implementan). Las variables(constantes) son implícitamente **public**, **final** y **static**. Los métodos son implícitamente **public abstract**. Las interfaces pueden heredar de otras interfaces, tienen herencia múltiple (entre interfaces **extends** nunca **implements**). Pensar en las interfaces como clases 100 % abstractas.

Una **interface** solo puede extender de una o muchas interfaces.

Las **interfaces** pueden ser utilizadas polimórficamente.

Las interfaces son implícitamente abstractas, osea, el siguiente código es legal:

```
public abstract interface Rollable
```

17. Si desde una **clase no abstracta** se implementa una **interface**, es obligatorio implementar todos los métodos de esa interfaz.

En cambio las **clases abstractas** no tienen la obligación de implementar los métodos de la **interface**, eso sí, la primera **clase no abstracta** que extienda de ella, deberá implementar todos sus métodos declarados como abstractos y también los métodos de la interfaz que implementó.

</OBJECTIVE 1.1 AND 1.2: Declare Interfaces>

<OBJECTIVE 1.3 AND 1.4: Declare Class Members>

<SUMMARY>

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

1.4 Develop code that declares both static and non-static methods, and—if appropriate—use method names that adhere to the JavaBeans naming standards. Also develop code that declares and uses a variable-length argument list.

</SUMMARY>

- 18.** **native**, solo se utiliza para métodos y no tiene cuerpo(igual que los métodos abstractos):

```
public native void prestarSaludo();
```

- 19.** Sobre los métodos que aceptan **argumentos variables(VAR-ARGS)**: Estos métodos aceptan cero o más argumentos en su llamada, su sintaxis es: nombreMetodo(int ... x) ó nombreMetodo(long x, float y, int ... z). Si el método debe aceptar argumentos normales, **los argumentos variables siempre deben ir al final y solo se puede tener un var-args en un método.**

- 20.** Es legal que un método se llame como el constructor, pero es una falta grave llamar un método así.

- 21.** Las **variables** dentro de los métodos deben ser siempre inicializadas explícitamente, en este caso a diferencia de los atributos, el compilador dará un error a la hora que se quieren utilizar estas variables no inicializadas. Las **constantes** se indican con el modificador **final**.

- 22.** Tabla de accesos a miembros de clase

TABLE I-2 Determining Access to Class Members

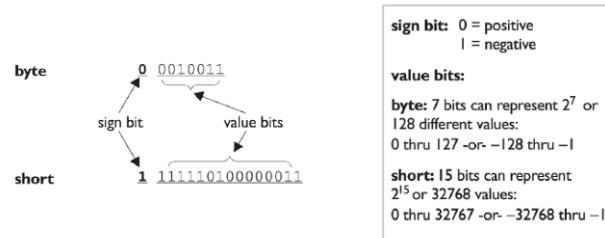
Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, through inheritance	No	No
From any non-subclass class outside the package	Yes	No	No	No

23. Sobre las variables primitivas:

Pueden ser de 8 tipos: **char**, **boolean**, **byte**, **short**, **int**, **long**, **double**, or **float**.

Los 6 tipos numéricos, están compuestos por bytes de 8 bits y pueden ser positivos o negativos. El primer bit es usado para representar el signo, donde un 1 significa negativo y un 0 significa positivo

FIGURE I-6 The Sign bit for a byte



La siguiente tabla muestra los tamaños y rangos de los tipos primitivos numéricos, estos rangos ya no son tomados en los exámenes actuales, solo son tomados los tamaños.

TABLE I-3 Ranges of Numeric Primitives

Type	Bits	Bytes	Minimum Range	Maximum Range
byte	8	1	-2^7	$2^7 - 1$
short	16	2	-2^{15}	$2^{15} - 1$
int	32	4	-2^{31}	$2^{31} - 1$
long	64	8	-2^{63}	$2^{63} - 1$
float	32	4	n/a	n/a
double	64	8	n/a	n/a

La cantidad de negativos es igual a la cantidad de positivos, el cero "0" es considerado como positivo, es por esa razón que se resta -1 en el rango máximo.

La fórmula para hallar el rango mínimo es $-2^{(\text{bits}-1)}$ y para el rango máximo es $2^{(\text{bits}-1)} - 1$. En ambos se resta **1 bits** que es el que representa el signo (positivo o negativo).

Los de tipo **boolean** no tienen rango, solo aceptan **true** o **false**.

Los de tipo **char** (1 carácter), es de 16 bits Unicode character. puede tener 2^{16} posibles valores, el rango es de 0 a 65535, osea es un poco más grande que un **short**. Al ser de tipo char, no cuenta con signo de positivo o negativo, por tal razón no se le resta -1 a la potencia.

24. Sobre las variables de instancia:

Son definidas dentro de las clases, pero fuera de los métodos y son inicializadas cuando la clase es intanciada.

Para el examen necesitas saber sobre las variables de instancia que:

- Pueden usar los cuatro niveles de acceso (lo que significa que puede ser marcado con cualquiera de los 3 modificadores de acceso).
- Pueden ser marcadas como **final**
- Pueden ser marcadas como **transient**
- No pueden ser marcadas como **abstract**
- No pueden ser marcadas como **synchronized**
- No pueden ser marcadas como **strictfp**
- No pueden ser marcadas como **native**
- No pueden ser marcadas como static, porque entonces se habrían convertido en variables de clase.

En el siguiente cuadro se muestra que modificadores pueden ser aplicados tanto a métodos como variables.

FIGURE I-7 Comparison of modifiers on variables vs. methods

Local Variables	Variables (non-local)	Methods
final	final public protected private static transient volatile	final public protected private static abstract synchronized strictfp native

Las variables locales, se almacenan en el **stack**.

Las variables de instancia, se almacenan en el **heap**

Las variables locales, siempre tienen que ser inicializadas (inicialización explícita). Pueden ser declaradas sin inicializar (el compilador no marca error), pero al momento de querer utilizarla en alguna parte del método, es ahí donde el compilador indica que la variable debe ser inicializada.

Las variables de instancia no requieren ser inicializadas (inicialización implícita). Estos son los valores por defecto que se les asignan a las **variables de instancia** de acuerdo a su tipo:

Variable Type	Default Value
Object reference	null (not referencing any object)
byte, short, int, long	0
float, double	0.0
boolean	false
char	'\u0000'

Una variable local puede ser llamada exactamente igual a una variable de instancia, ha esto se le conoce como **shadowing**. Por ejemplo:

```
class Persona {  
    private String nombre;  
    private String apellidos; // Line 7  
    public mostrarSaludo(){  
        String nombre = this.nombre; // Shadowing  
        System.out.println(nombre);  
    }  
    public mostrarSaludo(String apellidos){ //Shadowing  
        //apellidos = apellidos; // no marca error, no surgiría ningún efecto  
        // porque se esta tratando de actualizarse a si  
        // misma, ante esto se debe utilizar:::::::::::::  
        //  
        this.apellidos = apellidos;  
        System.out.println(this.nombre + apellidos);  
    }  
}
```

25. Sobre los arrays:

La declaración de array de primitivos:

```
int[] key; // Este tipo de declaración es la recomendada  
int key []; // Este tipo de declaración es válida pero no es recomendada
```

La declaración de array de Objetos:

```
Thread[] threads; // Este tipo de declaración es la recomendada  
Thread threads [];// Este tipo de declaración es válida pero no es recomendada
```

No es una correcta declaración:

```
int[5] scores; // La dimensión se asigna a la hora de instanciar.
```

Construyendo e inicializando un array anónimo:

```
int[] testScores;  
testScores = new int[] {4,7,2};
```

Recuerda que tu no debes especificar el tamaño cuando se utiliza un array anónimo, el tamaño es derivado del número de elementos separados por, y dentro de {}.

Una declaración incorrecta de un array anónimo sería:

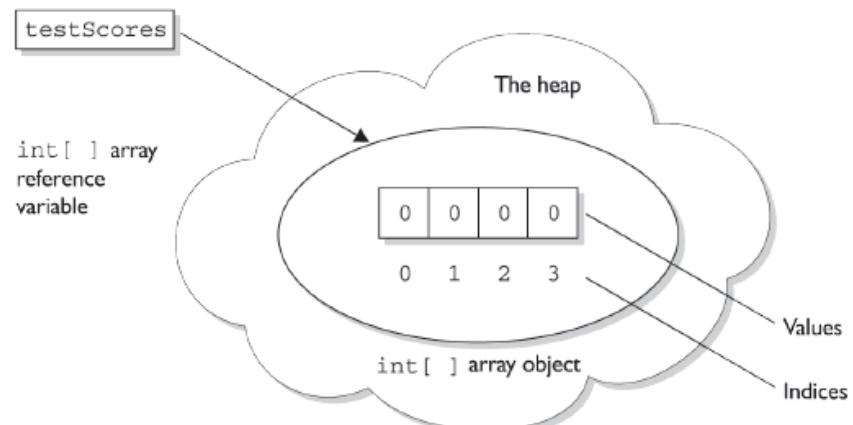
```
Object objArray = new Object[3] {null, new Object(), new Object()};  
// not legal;size must not be specified
```

Construcción de Arrays:

```
int[] testScores; // Declares the array of ints  
testScores = new int[4]; // constructs an array and assigns it  
// to the testScores variable
```

FIGURE 3-2

A one-dimensional array on the Heap



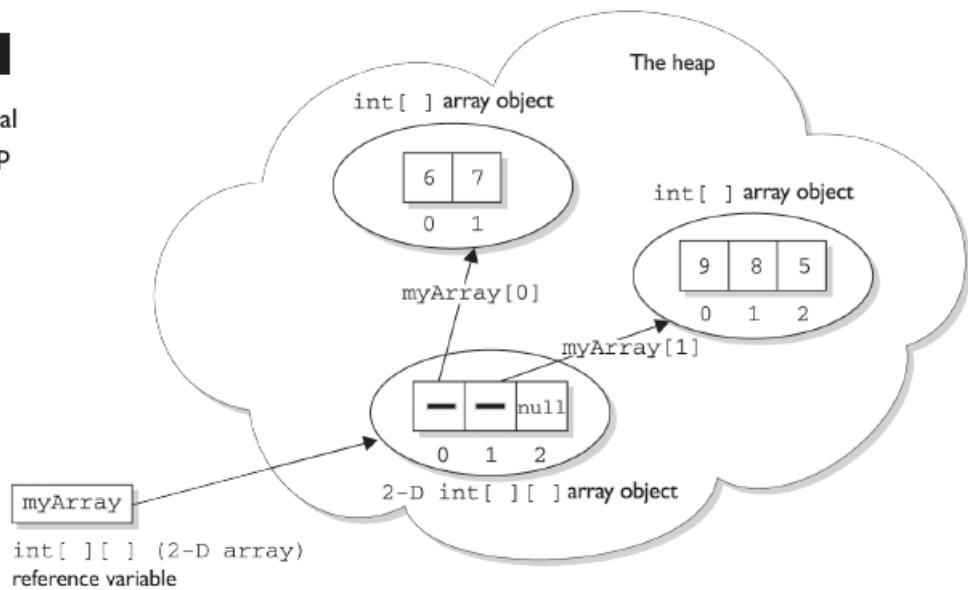
```
int[] testScores = new int[4];  
Thread[] threads = new Thread[5];
```

Construcción de array multidimensional:

```
int [ ] [ ] myArray = new int [3] [ ] ;
```

FIGURE 3-3

A two-dimensional array on the Heap



Picture demonstrates the result of the following code:

```
int [ ][ ] myArray = new int[3][ ] ;
myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```

Mas info de Arrays:

http://java.sun.com/docs/books/jls/third_edition/html/arrays.html

26. Sobre los bloques de inicialización de una clase:

Analizando las siguientes clases:

```
class ClasePadre {
    ClasePadre() {
        System.out.println("inicio clase padre");
    }
}

public class InitMembersOfClass extends ClasePadre {

    InitMembersOfClass(int x) {
        System.out.println("1-arg const");
    }

    InitMembersOfClass() {
        System.out.println("no-arg const");
    }

    static {
        System.out.println("1st static init");
    }

    {
        // los bloques son los segundos en ser ejecutados y
        // se ejecutan de acuerdo al orden en que se encuentran en la clase.
        System.out.println("1st instance init::");
    }

    {
        System.out.println("2nd instance init");
    }

    static {
        // los bloques static son los primeros en ser ejecutados y
        // se ejecutan de acuerdo al orden en que se encuentran en la clase.
        System.out.println("2nd static init");
    }

    public static void main(String[] args) {
        new InitMembersOfClass();
        new InitMembersOfClass(7);
    }
}
```

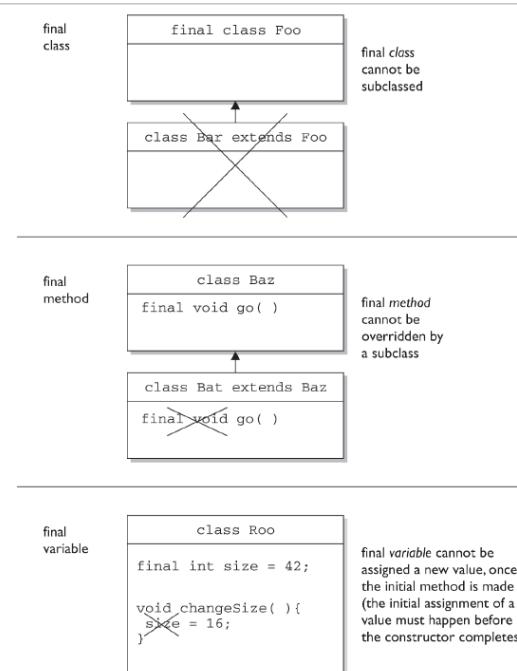
El resultado sería:

```
1st static init
2nd static init
inicio clase padre
1st instance init
2nd instance init
no-arg const
inicio clase padre
1st instance init
2nd instance init
1-arg const
```

Recordar las siguientes reglas:

- Un bloque de inicialización se ejecutan en el orden en que aparecen.
- Los bloques estáticos se ejecutan cuando la clase es llamada por primera vez y se ejecutan en el orden en que aparecen.
- Los bloques de instancia se ejecutan siempre que la clase es instanciada.
- Los bloques de instancia se ejecutan después de que el constructor llame al super() y antes de que el constructor llame a las líneas que le siguen al super().

27. Sobre las variables **final**:



Se aplica a clases, métodos y variables locales y de instancia, si es aplicado a variables de instancia éstas tienen que inicializarse sino el compilador marca error, lo mismo pasa con las variables locales.

28. Sobre las variables **transient**

Si marcas como **transient** a una variable de instancia, tu estás diciendo a la JVM que ignore esta variable cuando el objeto que la contiene es **serializado**.

Solo es aplicado a variables de instancia.

29. Sobre las variables **volatile**

Volatile se utiliza con variables modificadas asincrónicamente por objetos en diferentes *threads* (literalmente "hilos", tareas que se ejecutan en paralelo); básicamente esto implica que distintas tareas pueden intentar modificar la variable simultáneamente, y **volatile** asegura que se vuelva a leer la variable (por si fue modificada) cada vez que se la va a usar (esto es, en lugar de usar registros de almacenamiento como buffer).

Solo es aplicado a variables de instancia.

30. Los tipos **enumerados** se definen usando la palabra **enum**. únicamente pueden ser **public** o **default** (Así como las clases). Ejemplo: enum Baraja {Oros, Bastos, Copas, Espadas}. Implícitamente extienden de la clase **java.lang.Enum**. Los enumerados pueden contener métodos, variables y constantes. No se pueden declarar dentro de los métodos de la clase, pero sí como miembro de ésta.

Ejemplos de uso:

- Cuando el enum está fuera de la clase

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING } // this cannot be private or protected
class Coffee {
    CoffeeSize size;
}
public class CoffeeTest1 {
    public static void main(String[] args) {
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG; // enum outside class
    }
}
```

- Cuando el enum está dentro de la clase

```
class Coffee2 {
    enum CoffeeSize {BIG, HUGE, OVERWHELMING }
    CoffeeSize size;
}
public class CoffeeTest2 {
    public static void main(String[] args) {
        Coffee2 drink = new Coffee2();
        drink.size = Coffee2.CoffeeSize.BIG; // enclosing class
        // name required
    }
}
```

Notas:

- No puede haber un enum dentro de un método.
- El punto y coma al final del **enum** {...}; es opcional ponerlo.
- Un enum puede estar solo dentro de un archivo .java sin necesidad de que exista una clase. Pero el nombre del enum debe coincidir con el nombre del fichero .java.
- Los enums no son Strings o ints. Las variables que se declaran son tipos de CoffeeSize y que cuando uno accede a estos, por ejemplo : CoffeeSize.BIG , puede utilizar los métodos que la clase **java.lang.Enum** provee, por ejemplo si queremos saber el ordinal de esa enumeración haríamos: CoffeeSize.BIG.ordinal(), lo cual devolvería 0.

Declarando constructores, métodos y variables en un enum

enum en realidad es una pequeña clase, por tal tu puedes hacer mas que una lista de valores constantes enumerados, puedes agregar constructores, variables de instancia, métodos.

Con esta idea podemos por ejemplo hacer como una tabla de búsqueda, donde diríamos que BIG es 8 onzas, HUG es 10 onzas, OVERWHELMING es 16 onzas.

En código sería:

```
enum CoffeeSize {
    BIG(8), HUGE(10), OVERWHELMING(16);
    // the arguments after the enum value are "passed"
    // as values to the constructor
    CoffeeSize(int ounces) {
        this.ounces = ounces;    // assign the value to
                                // an instance variable
    }
    private int ounces; // an instance variable each enum
    // value has
    public int getOunces() {
        return ounces;
    }
}

class Coffee {
    CoffeeSize size; // each instance of Coffee has-a
    // CoffeeSize enum
    public static void main(String[] args) {
        Coffee drink1 = new Coffee();
        drink1.size = CoffeeSize.BIG;
        Coffee drink2 = new Coffee();
        drink2.size = CoffeeSize.OVERWHELMING;
        System.out.println(drink1.size.getOunces()); // prints 8
        System.out.println(drink2.size.getOunces()); // prints 16
    }
}
```

Las constantes de enumeración pueden sobre escribir métodos del propio **enum**

En código sería:

```
enum CoffeeSize {
    BIG(8),
    HUGE(10),
    OVERWHELMING(16) {      // start a code block that defines
                           // the "body" for this constant
        public String getLidCode() {      // override the method
                                         // defined in CoffeeSize
            return "A";
        }
    }; // <-- the semicolon is REQUIRED when you have a body

    CoffeeSize(int ounces) {
        this.ounces = ounces;
    }
    private int ounces;
    public int getOunces() {
        return ounces;
    }
    public String getLidCode() { // this method is overridden
                               // by the OVERWHELMING constant
        return "B";      // the default value we want to return for
                          // CoffeeSize constants
    }
}
```

Mas info sobre enums:

<http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html>

31. Importaciones estáticas: se utilizan para importar una clase estática dentro de otra sin necesidad de anteponer el nombre de la clase cuando se accede a alguno de sus miembros estáticos. Se indica con **import static paquete.nombreClase**.

Mas info de importaciones estáticas:

http://java.sun.com/docs/books/jls/third_edition/html/packages.html#7.5.3

<http://java.sun.com/j2se/1.5.0/docs/guide/language/static-import.html>

</OBJECTIVE 1.3 AND 1.4: Declare Class Members>

</OBJECTIVES>

</CAP1: Declarations and Access Control>

<CAP2: Object Orientation>

<CERTIFICATION OBJECTIVES>

<OBJECTIVES>

<OBJECTIVE 5.1: Encapsulation>

<SUMMARY>

5.1 Develop code that implements tight encapsulation, loose coupling, and high cohesion in classes, and describe the benefits.

</SUMMARY>

32. Cuando se habla de encapsulamiento, se habla de proteger nuestras variables de cualquier otro programa que quiera acceder a ellas. Esto por buenas prácticas, por no tener en público variables que nada tienen que hacer fuera, mas aún, no tienen que ser vistas en el api cuando otro programador, instancie tu clase, etc.

33. Problema típico:

```
public class Account {  
    public double amount;  
    public Date lastAccess;  
    // ...  
}  
  
public class Application {  
    public static void main(String[] args) {  
        Account a = new Account();  
        a.amount = 30000; // Legal but bad!  
    }  
}
```

`amount` es accedido desde el void `main()`, es posible ya que es público. Esto es legal, pero es correcto?, en una aplicación bien pensada, no debería ocurrir, ya que `amount` es un campo calculado y debería ser actualizado desde un método interno de cálculo.

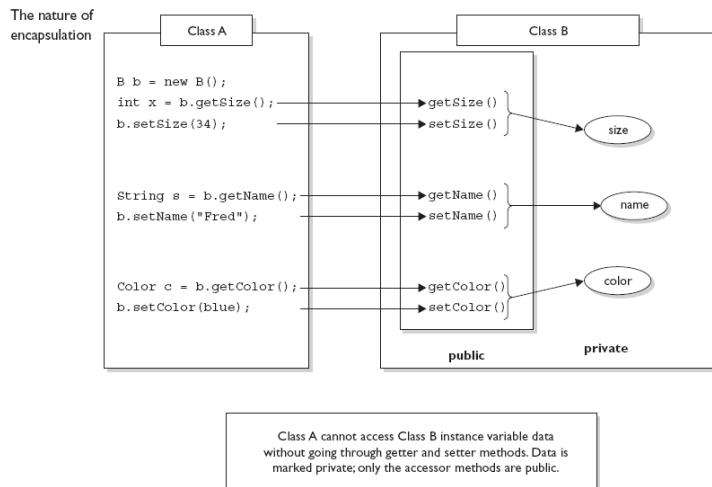
Si deseas que tu aplicación sea flexible, extensible, y de fácil mantenimiento, entonces tu diseño deberá incluir encapsulación. ¿Cómo se hace eso?:

- Mantener las variables de instancia protegidas (con un modificador de acceso, por lo general **private**).
- Hacer público los métodos de acceso, para utilizar éstos, en lugar de acceder directamente a las variables.
- Para los métodos, usar la convención JavaBean:

`set<someProperty> and get<someProperty>.`

34. Ejemplo de encapsulación:

FIGURE 2-1



</OBJECTIVE 5.1: Encapsulation>

<OBJECTIVE 5.5: Inheritance, Is-A, Has-A>

<SUMMARY>

5.5 Develop code that implements "is-a" and/or "has-a" relationships.

</SUMMARY>

35. Sobre las relaciones Is-A y Has-A:

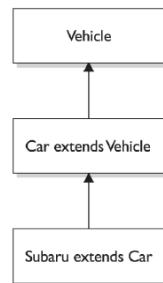
IS-A

Hace referencia a la herencia de clases o implementación de interfaces.

"Car **extends** Vehicle" means "Car IS-A Vehicle"
"Subaru **extends** Car" means "Subaru IS-A Car"

FIGURE 2-2

Inheritance tree
for Vehicle, Car,
Subaru



HAS-A

Está basada en el uso. En otras palabras Clase A **HAS-A** B si código en la clase A tiene una referencia a una instancia de la clase B, por ejemplo:

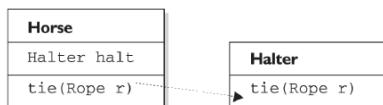
A Horse IS-A Animal. A Horse HAS-A Halter.

The code might look like this:

```
public class Animal {  
}  
  
public class Horse extends Animal {  
    private Halter myHalter;  
}
```

FIGURE 2-3

HAS-A
relationship
between Horse
and Halter



Horse class has a Halter; because Horse declares an instance variable of type Halter.
When code invokes tie() on a Horse instance, the Horse invokes tie() on the Horse object's Halter instance variable.

</OBJECTIVE 5.5: Inheritance, Is-A, Has-A>

<OBJECTIVE 5.2: Polymorphism>

<SUMMARY>

5.2 Given a scenario, develop code that demonstrates the use of polymorphism. Further, determine when casting will be necessary and recognize compiler vs. runtime errors related to object reference casting.

</SUMMARY>

36. Recuerda, cualquier objeto Java que puede pasar mas de un **test IS-A**, puede ser considerado polimórfico. Todos sabes que el padre de todas las clases es la clase **Object**. Por tal podemos decir que todos los objetos son polimórficos para Object.

37. Recuerda que la única forma de acceder a un objeto es através de una variable **de referencia**. Entonces es bueno que recuerdes lo siguiente:

- Una variable de referencia puede ser de un solo tipo, y una vez declarada, ese tipo nunca puede ser modificado(aunque el objeto al que hace referencia puede cambiar).
- Una referencia es una variable, por lo que pueden ser reasignados a otros objetos, (a menos que la variable se declare final).
- El tipo de la variable de referencia, determina que métodos pueden ser invocados en el objeto al que referencia.
- Una variable de referencia puede referenciar a cualquier objeto del mismo tipo o un subtipo.
- Una variable de referencia puede ser declarada como un tipo **clase** o un tipo **interface**. Si la variable se declara como un tipo **interface**, esta puede referenciar a cualquier objeto que implemente la **interface**.

</OBJECTIVE 5.2: Polymorphism>

<OBJECTIVE 1.5 and 5.4: Overriding / Overloading>

38. Reglas para los métodos sobreescritos(overriding):

- Los métodos pueden ser sobreescritos sólo si son heredados por la subclases.
- La lista de argumentos debe coincidir exactamente(número de argumentos y tipo) con la de los método sobreescritos. Si no coinciden lo que se estaría haciendo es un sobrecarga(overload).
- El tipo de retorno debe ser el mismo o de un subtipo del tipo de retorno declarado en el método sobreescrito(covariant return). P.e.: Si el método sobreescrito retorna Object, el método que sobreescribe puede retornar un String.
- El nivel de acceso de los métodos que sobreescriben no puede ser más restrictivo que los métodos sobreescritos, por el contrario si pueden ser menos restrictivos. P.e: Si el método sobreescrito tiene un nivel de acceso **protected** entonces el método que sobreescribe no puede ser **private**, PERO si puede ser **public**.
- El método que sobreescribe puede lanzar(throw) cualquier excepción de tiempo de ejecución (unchecked(runtime) exception). Independientemente de si el método sobreescrito declara la excepción.

En otras palabras al método que sobreescribe se le puede asignar throws RuntimeException y/o sus subclases de RuntimeException, tales como :

ArithmetricException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMException, ConcurrentModificationException, DOMException, EmptyStackException, IllegalArgumentExeption, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, ImagingOpException, IndexOutOfBoundsException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NullPointerException, ProfileDataException, ProviderException, RasterFormatException, SecurityException, SystemException, UndeclaredThrowableException, UnmodifiableSetException, UnsupportedOperationException.

- El método que sobreescriben no puede lanzar(throw) cualquier excepción que sea de mayor jerarquía o diferente que la declarada por el método sobreescrito. Por ejemplo si un método declara un throw IOException el método que sobreescribe no puede declarar un throw SQLException o un throw Exception, ya que la primera no es igual y la segunda está por encima de la jerarquía de IOException. En cambio sí podría sobreescribirse con un método que declare un FileNotFoundException, ya que ésta esta por debajo de la jerarquía de IOException.
- Los métodos que sobreescriben pueden o no volver a declarar el throws declarado por el método sobreescrito.
- No se puede sobreescribir un método final.

- No se puede sobreescribir un método static.
- Si un método no puede ser heredado, tu no puedes sobreescribirlo. Si por ejemplo la clase padre tiene un método **private String eat()** y una subclase de esta tiene un método **public String eat()**, aparentemente se está sobreescribiendo porque un método que sobreescribe si puede tener como nivel de acceso a uno menos restrictivo que el método sobreescrito, pero en realidad no se está sobreescribiendo. En esta situación el compilador no marca error, pero tener en claro que no se está sobreescribiendo, solo se está creando un método nuevo con el mismo nombre dentro de la clase.

39. Reglas para los métodos sobrecargados (overloaded methods)

- Deben cambiar la lista de argumentos (el tipo o el número de argumentos).
- Pueden cambiar el tipo de retorno.
- Pueden cambiar el modificador de acceso.
- Pueden declarar excepciones nuevas o más amplias.
- Un método puede ser sobrecargado en la misma clase o en una subclase.
- Cuando ocurre el siguiente caso:

```
class EasyOver {

    static void go(long x) {
        System.out.print("long ");
    }

    static void go(int x) {
        System.out.print("int ");
    }

    static void go(double x) {
        System.out.print("double ");
    }

    public static void main(String[] args) {
        byte b = 5;
        short s = 5;
        long l = 5;
        float f = 5.0f;
        go(b);
        go(s);
        go(l);
        go(f);
    }
}
```

La salida es: int int long double

El orden de ejecución siempre es de **menor a mayor** tipo de dato. Como se ve, cuando llamamos a go(b) y se manda un byte, el método sobrecargado que llama es `static void go(int x)`.

- Cuando ocurre el siguiente caso:

```
class AddBoxing {
    static void go(Integer x) {
        System.out.println("Integer");
    }

    static void go(long x) {
        System.out.println("long");
    }

    public static void main(String[] args) {
        int i = 5;
        go(i); // which go() will be invoked?
    }
}
```

La salida es: long

El compilador siempre busca el primitivo con menor tipo de dato, luego si no hubiese buscado en los Wrappers(Ver capítulo 3: **Assignments** , ahí se habla de los wrappers).

- Cuando ocurre el siguiente caso:

```
class AddVarargs {
    static void go(int x, int y) {
        System.out.println("int,int");
    }

    static void go(byte... x) {
        System.out.println("byte... ");
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b, b); // which go() will be invoked?
    }
}
```

La salida es: int,int

El compilador siempre prefiere a los primitivos ante un var-args.

- Cuando ocurre el siguiente caso:

```
class BoxOrVararg {
    static void go(Byte x, Byte y) {
        System.out.println("Byte, Byte");
    }

    static void go(byte... x) {
        System.out.println("byte... ");
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b, b); // which go() will be invoked?
    }
}
```

La salida es: Byte, Byte

El compilador siempre prefiere a los wrappers ante un var-args.

- Caso en que no funciona el overloading:

```
class Dog4 {
    public static void main(String[] args) {
        Dog4 d = new Dog4();
        d.test(new Integer(5)); // can't widen an Integer
        // to a Long
    }

    void test(Long x) {
    }
}
```

Uno puede pensar que un Integer puede ser pasado a un Long por ser de menor rango, pero **no es así**. En este ejemplo el compilador marca error. Cuando se trata de parametros de tipo Wrapper solo se debe enviar valores del mismo tipo o primitivos que pueda aceptarse.

- Caso en que no funciona el overloading:

```
class WidenAndBox {
    static void go(Byte x) {
        System.out.println("Long");
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b); // must widen then box - illegal
    }
}
```

Cuando un primitivo es enviado, el parámetro receptor debe ser del mismo tipo primitivo, un primitivo de mayor rango o su respectivo Wrapper (Byte) o su superclase (Number) u (Object). Es por eso que el compilador marca error.

40. Cuadro de diferencias entre los métodos sobrecargados y sobreescritos

TABLE 2-3 Differences Between Overloaded and Overridden Methods		
	Overloaded Method	Overridden Method
Argument(s)	Must change.	Must not change.
Return type	Can change.	Can't change except for covariant returns.
Exceptions	Can change.	Can reduce or eliminate. Must not throw new or broader checked exceptions.
Access	Can change.	Must not make more restrictive (can be less restrictive).
Invocation	Reference type determines which overloaded version (based on declared argument types) is selected. Happens at <i>compile</i> time. The actual <i>method</i> that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the <i>signature</i> of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the <i>class</i> in which the method lives.	Object type (in other words, <i>the type of the actual instance on the heap</i>) determines which method is selected. Happens at <i>runtime</i> .

41. Sobre el **upcasting** y **downcasting**:

El **upcasting** trabaja implícitamente, tu no tienes que especificar el tipo a castear, porque cuando uno aplica upcasting, implicitamente está restringiendo el número de métodos a invocar, por ejemplo(Horse **IS-A** Animal):

```
Horse horse = new Horse();
Animal animal = (Animal)horse; // Explícita (Válida)
Animal animal = horse; // Implícita (Válida)
// "horse es de tipo Animal"
```

El **downcasting** si se quiere aplicar de la siguiente manera

```
Animal animal = new Animal();
Horse horse = (Horse)animal; // Compila pero falla.
```

fallaría, debido a que por ejemplo si horse tuviera un método que no tiene animal y se quiere ejecutar saldría error ya que la referencia animal que es la que verdaderamente estaría en horse no cuenta con ese método. Además habría que preguntarse: “animal es de tipo horse????”. Sería diferente si fuera:

```
Animal animal = new Horse();
Horse animal = (Horse)animal;//Compila y no falla.
```

Ó:

```
Object miObject = new String("casa");
String miString = (String)miObject;
```

En ambos casos no falla, debido a que en tiempo de ejecución la JVM va a saber que animal referencia a un tipo horse y miObject referencia a un tipo String.

</OBJECTIVE 1.5 and 5.4: Overriding / Overloading>

<OBJECTIVE 1.6, 5.3, and 5.4 Constructors and Instantiation>

<SUMMARY>

1.6 Given a set of classes and superclasses, develop constructors for one or more of the classes. Given a class declaration, determine if a default constructor will be created, and if so, determine the behavior of that constructor. Given a nested or nonnested class listing, write code to instantiate the class.

5.3 Explain the effect of modifiers on inheritance with respect to constructors, instance or static variables, and instance or static methods.

5.4 Given a scenario, develop code that declares and/or invokes overridden or overloaded methods and code that declares and/or invokes superclass, overridden, or overloaded constructors.

</SUMMARY>

42. Constructor Básico:

```
class Foo {  
    Foo() {  
        } // The constructor for the Foo class  
}
```

Recuerda esto:

- Los constructores no retornan nada (no llevan void ni cualquier tipo, si lo tuvieran, se convertiría en método y dejaría de ser constructor).
- Se llaman igual a la clase.

43. Constructor con parámetros:

```
class Foo {  
    int size;  
    String name;  
  
    Foo(String name, int size) {  
        this.name = name;  
        this.size = size;  
    }  
}
```

Este ejemplo no tiene constructor sin argumentos (no-arg constructor). Lo que significa que la siguiente linea de código arrojaría error:

```
Foo f = new Foo(); // Won't compile, no matching constructor
```

Pero si se instancia de la siguiente manera no arrojaría error:

```
Foo f = new Foo("Warner Penkaps", 26);
```

Recuerda:

- Cuando se declara uno o muchos constructores que tenga por lo menos un parámetro, el compilador ya no va crear el constructor por defecto(el que no tiene parámetros), por eso marca error en: `Foo f = new Foo();`; Entonces, lo único que queda por hacer para que no marque error es colocarle explícitamente el constructor por defecto(sin argumentos):

```
class Foo {  
    int size;  
    String name;  
  
    Foo() {  
    }  
  
    Foo(String name, int size) {  
        this.name = name;  
        this.size = size;  
    }  
}
```

- El único lugar dentro de la clase que puedes llamar a un constructor, es dentro de la llamada de otro constructor, por ejemplo, esto es legal:

```
class Foo {  
    int size;  
    String name;  
  
    Foo() {  
        Foo("nombre por defecto", 18)  
    }  
  
    Foo(String name, int size) {  
        this.name = name;  
        this.size = size;  
    }  
}
```

44. Como mencioné el compilador genera código para los constructores. En la siguiente tabla se muestra en que casos, hace esto:

TABLE 2-4 Compiler-Generated Constructor Code

Class Code (What You Type)	Compiler Generated Constructor Code (in Bold)
class Foo { }	class Foo { Foo() { super(); } }
class Foo { Foo() { } }	class Foo { Foo() { super(); } }
public class Foo { }	public class Foo { public Foo() { super(); } }
class Foo { Foo(String s) { } }	class Foo { Foo(String s) { super(); } }
class Foo { Foo(String s) { super(); } }	Nothing, compiler doesn't need to insert anything.
class Foo { void Foo() { } }	class Foo { void Foo() { Foo() { super(); } } } <i>(void Foo() is a method, not a constructor.)</i>

45. Ten mucho cuidado con el manejo de constructores en la herencia. Observa el siguiente código:

```
class Animal {
    Animal(String name) {
    }
}

class Horse extends Animal {
```

Aparéntemente está bien, PERO no es así.

Si nos basamos en el cuadro anterior, el compilador generaría el siguiente código:

```
class Animal {
    Animal(String name) {
    }
}

class Horse extends Animal {
    Horse() {
        super();
    }
}
```

Yyyyy, super(); está buscando el constructor por defecto(sin parámetros). En conclusión, este código marca error de compilación.

</OBJECTIVE 1.6, 5.3, and 5.4 Constructors and Instantiation>

<OBJECTIVE 5.1 Coupling and Cohesion >

- 46.** Sobre el bajo acoplamiento y alta cohesion de clases(**Coupling and Cohesion**).
- Acoplamiento se refiere al grado en que una clase conoce o utiliza los miembros de otra clase.
 - Bajo acoplamiento es el estado deseable de tener las clases.
 - o Trata de reducir al mínimo referencias del uno al otro, y limite la amplitud del uso del API.
 - Un situación de bajo acoplamiento es cuando la variable de referencia es declarada por un tipo de interface, no una clase. La interface proporciona un pequeño número de métodos.
 - Cuando hay bajo acoplamiento es poco probable que los cambios realizados a una clase requiera hacer cambios en otro lado.
 - Cohesión se refiere al grado en que una clase tiene un único y bien definido papel o responsabilidad.
 - Alta cohesión es el estado deseable de una clase cuyos miembros tienen un apoyo único, bien centrado en su papel o responsabilidad. Por ejemplo si queremos hacer una clase que imprima reportes, podemos tener una clase que se encargue de la conexión a la base de datos, otra de escoger la salida, y así más clases que pueden ser reutilizables por la clase principal, todo esto para cumplir un único objetivo que es emitir el reporte.
 - La cohesión es el principio OO el cual está asociado con asegurar que una clase esté diseñada con un único objetivo bien enfocado.

</OBJECTIVE 5.1 Coupling and Cohesion>

</OBJECTIVES>

</CERTIFICATION OBJETIVES>

</CAP2: Object Orientation>

<CAP3: Assignments>

47. Sobre las clases wrapper:

Primitive	Wrapper Class	Constructor Arguments
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
double	Double	double or String
float	Float	float, double, or String
int	Integer	int or String
long	Long	long or String
short	Short	short or String

48. Sobre el boxing, =, equals() en Wrappers

```
Integer int1 = 1000;
Integer int2 = 1000;
if(int1 != int2) System.out.println("different objects");
if(int1 == int2) System.out.println("== objects");
if(int1.equals(int2)) System.out.println("meaningfully equal");
```

//la salida es :

```
different objects
meaningfully equal
```

Que es lo que está sucediendo?

Lo que pasa que cuando se compara con equals, se está comparando el valor y el tipo de dato, si coinciden estos dos criterios entonces entra al if. En cambio el ==, !=, funciona solo cuando los valores primitivos son:

- Boolean
- Byte
- Character from \u0000 to \u007f (7f is 127 in decimal).
- Short and Integer from -128 to 127

Es por eso que la segunda condicional (`if(int1 == int2)`) no da true debido que los valores de las variables int1 e int2 son 1000, pero si fueran menores o iguales que 127, si darían true.

49. Sobre el Garbage Collection:

- Puede ser llamado con la sentencia `System.gc()` o `Runtime.getRuntime().gc()`.
- Solo instancias de clases son sujetas al GC, no primitivos.
- El llamar adrede al recolector de basura de la JVM no nos asegura que se va a ejecutar. La JVM primero verifica si realmente hay un exceso de memoria antes de ejecutar su recolector de basura.
- Si se quiere ejecutar algo de código antes de que el objeto sea llamado por el GC, se debe sobreescribir el método `finalize()` que es de la clase Object. En términos generales se recomienda que no se sobreescriba este método.
- Un objeto se convierte en elegible para el GC cuando es inalcanzable(**Unreachability**) por cualquier código. Hay **dos** maneras para que esto ocurra:

1) Cuando explicitamente el objeto es seteado NULL.

Ejemplo:

```
class GarbageTruck {  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("hello");  
        System.out.println(sb);  
        // El objeto StringBuffer no es elegible por colector  
        sb = null;  
        // Ahora el objeto StringBuffer es elegible por colector  
    }  
}
```

2) Cuando la referencia que indicaba a este objeto, apunta hacia cualquier otro

Ejemplo:

```
class GarbageTruck {  
    public static void main(String[] args) {  
        StringBuffer s1 = new StringBuffer("hello");  
        StringBuffer s2 = new StringBuffer("goodbye");  
        System.out.println(s1);  
        // En este punto el StringBuffer "hello" no es elegible por el gc  
        s1 = s2; // Redirects s1 to refer to the "goodbye" object  
        // Ahora el StringBuffer "hello" es elegible por el gc  
    }  
}
```

Consideraciones:

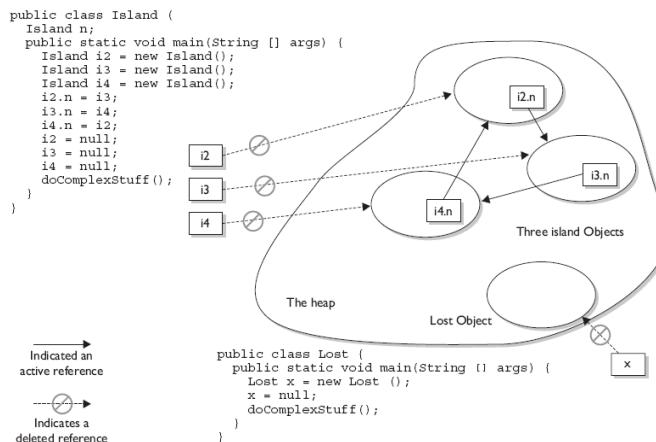
- a) **Los objetos que son creados dentro de un método**, también son considerados por el gc. Cuando un método es invocado, cualquier variable local creada existe solo para la duración del método. **Pero si un objeto** creado dentro del método es retornado, entonces esta referencia es asignada en una variable, por lo tanto este **objeto** no es elegible por el gc.

```
class GarbageFactory {  
    public static void main(String[] args) {  
        Date d = getDate();  
        doComplicatedStuff();  
        System.out.println("d = " + d);  
    }  
    public static Date getDate() {  
        Date d2 = new Date();  
        StringBuffer now = new StringBuffer(d2.toString());  
        System.out.println(now);  
        return d2;  
    }  
}
```

En el ejemplo anterior, el método `getDate()`, crea un objeto que será asignado a la variable `d`, y este objeto no será elegible recién al finalizar el `main()`. Sin embargo el objeto `StringBuffer` creado dentro del método `getDate()`, si lo será al finalizar este ya que implícitamente lo que va hacer es asignarle `null` a la variable `now`.

- b) Hay otra manera en que los objetos pueden ser elegibles por el GC, incluso si aún tienen referencias válidas. **Llamamos a este escenario “islands of isolation”(islas de aislamiento)**. Un ejemplo:

FIGURE 3-7 “Island” objects eligible for garbage collection



Explicación: Island es una clase que tiene una variable de instancia que es una variable de referencia a otra instancia de la misma clase. Se está creando 3 objetos Island en **i2, i3, i4**, y estas se están vinculando mediante la propiedad de cada una de ellas, luego **i2, i3, i4** se hacen null, por tanto los objetos a los que apuntaban quedan disponibles para el **GC**, a pesar de que aun existan referencias entre ellos, esto debido a que no hay forma de que un hilo directo pueda acceder a cualquiera de estos objetos.

50. Ejemplo de uso System.gc() y el Runtime.getRuntime().gc()

Ambos métodos se utilizan para llamar al GC(**Peeeero el sabrá si es necesario efectuar esta operación o no**).

Pruebas:

```
class CheckGC {  
    public static void main(String[] args) {  
        Runtime rt = Runtime.getRuntime();  
        System.out.println("Total JVM memory: " + rt.totalMemory());  
        System.out.println("Before Memory = " + rt.freeMemory());  
        Date d = null;  
        for (int i = 0; i < 600000; i++) {  
            d = new Date();  
            d = null;  
        }  
        System.out.println("After Memory = " + rt.freeMemory());  
        rt.gc(); // an alternate to System.gc()  
        System.out.println("After GC Memory = " + rt.freeMemory());  
    }  
}
```

51. Sobre el método finalize():

Java proporciona un mecanismo para ejecutar un código justo antes de que se elimine el objeto por el GC. Este código se encuentra en un método llamado **finalize()** de la clase Object que heredan todas las clases. Esto suena super OK, y te puede dar una idea de: “Es el método perfecto para sobreescribirlo y dentro, cerrar todos los recursos abiertos”. El problema es que, como ya sabes no puedes contar con el recolector de basura para eliminar un objeto cada vez que quieras. Por lo tanto, cualquier código puesto dentro del método **finalize()** no está garantizado que funcione. No se puede confiar en él. De hecho, se recomienda no sobreescribir **finalize()** en absoluto.

Quizz del capítulo 3(Assignments)

- 1) In the following code, which is the first statement, where the object originally held in *m*, may be garbage collected?

(Assume that MyObject is a properly defined class.)

```
public class Test //1.  
{ //2.  
    public static void main(String[] args) //3.  
    {  
        MyObject m = new MyObject("Sun", 48); //5.  
        m.myMethod(); //6.  
        System.out.println("Working"); //7.  
        m = null; //8.  
        m = new MyObject ("Moon", 36); //9.  
        m.yourMethod(); //10.  
        System.out.println("Testing"); //11.  
    }  
}
```

Choose one answer.

- a) Line 12
- b) Line 13
- c) Line 9
- d) Line 10

</CAP3: Assignments>

<CAP4: Operators>

<CERTIFICATION OBJETIVES>

<OBJECTIVE 7.6 : Java Operators>

52. Sobre los Operadores:

- **Operadores compuestos de asignación:**

`-=, +=, /=`

Ejemplo:

```
y -= 6;  
x += 2 * 5;  
x *= 2 + 5;
```

Equivale a decir:

```
y = y - 6;  
x = x + (2 * 5);  
x = x * (2 + 5);
```

- **Operadores relacionales**, siempre su resultado es un boolean(true o false):

`<, <=, >, >=, ==, !=.`

Hay cuatro cosas que se pueden testear:

- números.
- caractéres.
- booleanos primitivos.
- variables que refieren a Objetos.

Que es lo que hace `==`? Evalua el valor en la variable, en otras palabras, el **bit pattern**.

- Cuando se utiliza el operador `instanceof`, p.e: `objA instanceof ClassX`, devuelve true cuando `ClassA IS-A ClassX`, ó `objA` es una instancia de `ClassX` o implementa `InterfaceX`, esto si suponemos que en vez de clase fuera un interface.
- Usted no puede usar el operador `instanceof` para evaluar dos diferentes jerarquías de clase. Por ejemplo el siguiente código no compila:

```
class Cat {}  
class Dog {  
    public static void main(String [] args) {  
        Dog d = new Dog();  
        System.out.println(d instanceof Cat);  
    }  
}
```

- Nota:

exam

watch

Remember that arrays are objects, even if the array is an array of primitives. Watch for questions that look something like this:

```
int [] nums = new int[3];
if (nums instanceof Object) { } // result is true
```

An array is always an instance of Object. Any array.

Table 4-1 summarizes the use of the `instanceof` operator given the following:

```
interface Face { }
class Bar implements Face{ }
class Foo extends Bar { }
```

TABLE 4-1 Operands and Results Using `instanceof` Operator.

First Operand (Reference Being Tested)	instanceof Operand (Type We're Comparing the Reference Against)	Result
null	Any class or interface type	false
Foo instance	Foo, Bar, Face, Object	true
Bar instance	Bar, Face, Object	true
Bar instance	Foo	false
Foo []	Foo, Bar, Face	false
Foo []	Object	true
Foo [1]	Foo, Bar, Face, Object	true

- Operadores Aritméticos:

`+, -, *, /, %`

Recordar que por defecto las expresiones son evaluadas de izquierda a derecha. Si se quiere condicionar se debe utilizar los paréntesis “()”. Recordar también que el `*`, `/` y `%` tienen mayor precedencia que el `+` y `-`.

- Operadores de incremento y decremento

`++(prefijo y postfijo), --(prefijo y postfijo)`

Ejemplo de uso:

```
1. class MathTest {
2.     static int players = 0;
3.     public static void main (String [] args) {
4.         System.out.println("players online: " + players++);
5.         System.out.println("The value of players is " + players);
6.         System.out.println("The value of players is now " + ++players);
7.     }
8. }
```

La salida es:

```
players online: 0
The value of players is 1
The value of players is now 2
```

Observaciones:

Cuando el operador está después de la variables(postfijo), se va a operar la variable después de ser utilizada.

Cuando el operador está antes de la variables(prefijo), se va a operar la variable antes de ser utilizada.

- Operadores Lógicos:

&, |, ^, !, &&, y ||

Siendo **operadores binarios**: &, | y ^ (No están incluidos en el examen)

Pero si son utilizados para comparar booleanos, entonces:

- & a diferencia de && evalúa todas las expresiones que están siendo operadas, a pesar de que una de ellas sea false.
- | a diferencia de || evalúa todas las expresiones que están siendo operadas, a pesar de que una de ellas sea true.
- ^ (XOR, OR exclusivo) sólo debe haber un true en toda la expresión.

53. Ejemplos de uso de operadores:

- Evaluando igualdad entre Enums:

```
class EnumEqual {  
    enum Color {  
        RED, BLUE  
    } // ; is optional  
  
    public static void main(String[] args) {  
        Color c1 = Color.RED;  
        Color c2 = Color.RED;  
        if (c1 == c2) {  
            System.out.println("==");  
        }  
        if (c1.equals(c2)) {  
            System.out.println("dot equals");  
        }  
    }  
}
```

La salida es:

```
==  
dot equals
```

- Uso de **instanceof**:

a)

```
public static void main(String[] args) {  
    String s = new String("foo");  
    if (s instanceof String) {  
        System.out.print("s is a String");  
    }  
}
```

La salida es:

```
s is a String
```

b)

```
class B extends A {
    public static void main(String[] args) {
        A myA = new B();
        m2(myA);
    }

    public static void m2(A a) {
        if (a instanceof B)
            ((B) a).doBstuff(); // downcasting an A reference
        // to a B reference
    }

    public static void doBstuff() {
        System.out.println("'a' refers to a B");
    }
}
```

La salida es:

```
'a' refers to a B
```

c) Esto marca error de compilación;

```
class Cat {
}

class Dog {
    public static void main(String[] args) {
        Dog d = new Dog();
        System.out.println(d instanceof Cat);
    }
}
```

- Operadores condicional:

a)

```
class Salary {
    public static void main(String[] args) {
        int numOfPets = 3;
        String status = (numOfPets < 4) ? "Pet limit not exceeded"
            : "too many pets";
        System.out.println("This pet status is " + status);
    }
}
```

b) anidado

```
class AssignmentOps {
    public static void main(String[] args) {
        int sizeOfYard = 10;
        int numOfPets = 3;
        String status = (numOfPets < 4) ?
            "Pet count OK" : (sizeOfYard > 8) ?
                "Pet limit on the edge" : "too many pets";
        System.out.println("Pet status is " + status);
    }
}
```

- Operadores Lógicos

- a) Operadores a nivel de bit(**No viene en el examen**):

```
public static void main(String[] args) {  
    byte b1 = 6 & 8;  
    byte b2 = 7 | 9;  
    byte b3 = 5 ^ 4;  
    byte b4 = 5 >> 4;  
    byte b5 = 5 << 4;  
    System.out.println(b1 + " " + b2 + " " + b3 + " " + b4 + " " + b5);  
}
```

Salida:

0 15 1 0 80

- b) Operadores a nivel de bit pero usados para evaluar operaciones booleanas (**si viene en el examen**)

- 1)** Uso de | y ||

```
int z = 5;  
if(++z > 5 || ++z > 6) z++; // z = 7 after this code
```

versus:

```
int z = 5;  
if(++z > 5 | ++z > 6) z++; // z = 8 after this code
```

Que sucede?, el | evalua las dos expresiones si o si, en cambio el || evalua la segunda, siempre y cuando la primera sea falsa, si la primera fuese verdadera, entonces la segunda ya no es evaluada.

</OBJECTIVE 7.6 : JAVA_OPERATORS>

</CERTIFICATION_OBJETIVES>

</CAP4:Operators>

<CAP5: Flow Control, Exceptions, and Assertions>

<CERTIFICATION_OBJECTIVES>

<OBJECTIVE 2.1 : if and switch Statements>

54. Sobre if:

Un if básico:

```
int x = 3  
if (x==5) {  
    System.out.println("Dentro de una sentencia if");  
}
```

booleanExpression solo debe ser una **variable** o una invocación a un **método** que devuelva un **boolean** o **Boolean**.

si booleanExpression es **false** entonces puedes utilizar un if anidado:

```
if (x > 3) {  
    System.out.println("x es mayor que 3");  
} else {  
    System.out.println("x no es mayor que 3");  
}
```

Solo puede haber un else por cada if. Por ejemplo, el siguiente código no compilaría:

```
if (x > 3) {  
    System.out.println("x es mayor que 3");  
} else {  
    System.out.println("x no es mayor que 3");  
} else {  
    System.out.println("si no es ninguno");  
}
```

Si quieres hacer **else** anidado, entonces deberías utilizar **else if**:

```
if (x > 3) {  
    System.out.println("x es mayor que 3");  
} else if(x<3){  
    System.out.println("x es menor que 3");  
} else if(x==3){  
    System.out.println("x es igual a 3");  
} else {  
    System.out.println("si no es ninguno");  
}
```

Si la sentencia a ejecutar dentro del **if**, **else** o del **else if** es una línea, entonces puedes obviar las llaves {}. *Mala práctica (no es legible), Pero viene en el examen:*

```
if (x > 3)  
    System.out.println("x es mayor que 3");  
else if(x<3)  
    System.out.println("x es menor que 3");  
else if(x==3)  
    System.out.println("x es igual a 3");  
else  
    System.out.println("si no es ninguno");
```

Y para cerrar, ten cuidado con estooo: (Suele venir en el examen)

¿Imprime o no imprime Hola?

```
if (false);  
{  
    System.out.println("Hola");  
}
```

Respuesta: SI, debido a que el punto y coma();, es una sentencia vacía, entonces esta es la que se está obviando su ejecución con el **false**, y no el bloque de código que imprime **Hola**

55. Sobre switch and case:

```
switch (expression) {  
    case constant1: code block  
    case constant2: code block  
    default: code block  
}  
  
expression debe ser un char, byte, short, int o un enum (a partir de java 5), o la  
llamada a un método que te devuelva cualquiera de éstos tipos.  
  
constant1, puede ser un valor directo por ejemplo: 1, o puede ser una varibale, pero  
ésta debe ser final.
```

Errores de compilación:

```
1  byte g = 2;  
2  switch(g) {  
3      case 23:  
4      case 128:  
5      case 23:  
6      default:  
7  }
```

El compilador marca error en:

Línea 4, debido a que 128 no está dentro del rango de byte. Esto quiere decir que los cases deben estar en el mismo rango que el switch.

Línea 5, debido a que no puede haber dos cases iguales.

Observar bien la sintaxis del **switch case**, los dos puntos, el code block puede o no estar dentro de llaves.

Cuando el **switch** encuentra una coincidencia ejecuta todos los bloques de código a partir de ahí, es por eso que debe haber un **break** en cada bloque de código para que no se ejecute los siguientes cases.

Si el **switch** está dentro de un método que espera un Valor de retorno, entonces, todos los cases y el default deberían retornar un valor. Al menos que aya un retorno por defecto al finalizar el **switch**

Ejemplo de **switch case** con un **enum**:

```
enum Color {
    red, green, blue
}

class SwitchEnum {
    public static void main(String[] args) {
        Color c = Color.green;
        switch (c) {
            case red:
                System.out.print("red ");
            case green:
                System.out.print("green ");
            case blue:
                System.out.print("blue ");
            default:
                System.out.println("done");
        }
    }
}
```

El default, puede estar en cualquier orden, no necesariamente tiene que estar al final, y funciona como si fuera otro case, por ejemplo si :

```
int x = 2;
switch (x) {
    case 2:
        System.out.println("2");
    default:
        System.out.println("default");
    case 3:
        System.out.println("3");
    case 4:
        System.out.println("4");
}
```

La salida sería:

```
2
default
3
4
```

Y si

```
int x = 7;
switch (x) {
    case 2:
        System.out.println("2");
    default:
        System.out.println("default");
    case 3:
        System.out.println("3");
    case 4:
        System.out.println("4");
}
```

La salida sería:

```
default
3
4
```

</OBJECTIVE 2.1 : if and switch Statements >

<OBJECTIVE 2.2 : Loops and Iterators>

56. Los Java loops son 3 : while, do , y for (**a partir de Java 5 el for tiene dos variaciones**)

57. Usando el bucle **while**:

while es bueno para escenarios donde tu no conoces cuantas veces un bloque de código se repetirá, pero tu quieras que continue iterando mientras la condición es **true**.

Ejemplo:

```
while (expression) {  
    // do stuff  
}
```

or

```
int x = 2;  
while (x == 2) {  
    System.out.println(x);  
    ++x;  
}
```

58. Usando el bucle **do**:

do es similar al **while**, excepto que la expresión no es evaluada hasta después de que el código dentro del do es ejecutado. Así se garantiza que el código dentro del doo es ejecutado por lo menos una vez.

Ejemplo:

```
do {  
    System.out.println("Inside loop");  
} while (false);
```

Nota: El punto y coma (;) es obligatorio, ojo con eso.

59. Usando el bucle **for**:

A partir de java 5, se puede utilizar el famoso **for(:){...}** muy usado para iterar colecciones, mas adelante hablaré de esto.

El básico bucle for:

```
for /*Initialization*/ ; /*Condition*/ ; /* Iteration */) {  
    /* loop body */  
}
```

Este consta de 3 partes:

- Declaración e inicialización de variables
- La expresión booleana (condición)
- La expresión de iteración.

Las 3 partes están separadas por el punto y coma(;)

Ejemplos:

```
for (int i = 0; i<10; i++) {  
    System.out.println("i is " + i);  
}  
  
for (int x = 10, y = 3; y > 3; y++) {  
}  
  
for (int x = 0, y=0; (((x < 10) && (y-- > 2)) | x == 3)); x++) {  
}  
  
for ( ; ; ){ // Bucle infinito  
}  
  
int i = 0;  
for (; i < 10;){  
    i++;  
    // mas codigo  
}  
  
for (int i = 0, j = 0; (i < 10) && (j < 10); i++, j++) {  
    System.out.println("i is " + i + " j is " + j);  
}
```

Ejemplo de error de compilación:

```
for (int x = 1; x < 2; x++) {  
    System.out.println(x); // Legal  
}  
System.out.println(x); // No Legal! x es ahora fuera de este  
                      // ámbito y no puede ser accedido  
  
for (int x = 0; (x > 5), (y < 2); x++) { } // mas de 1 expresion boolean
```

60. Causas para interrumpir un for

TABLE 5-1 Causes of Early Loop Termination

Code in Loop	What Happens
break	Execution jumps immediately to the 1st statement after the for loop.
return	Execution jumps immediately back to the calling method.
System.exit()	All program execution stops; the VM shuts down.

61. Los for-each tienen la siguiente sintaxis:

```
for(declaration : expression)
```

Este consta de 2 partes:

- Declaración: La nueva variable del tipo compatible con los elementos del arreglo que tu estás accediendo. Esta variable estará disponible dentro del bloque **for** y será la misma que el elemento actual del arreglo.
- La expresión: variable o método que devuelva un arreglo. El arreglo puede ser de tipo primitivos, objetos, arrays.

62. Son declaraciones legales e ilegales para los for-each:

```
// Variables
int x;
long x2;
Long [] La = {4L, 5L, 6L};
long [] la = {7L, 8L, 9L};
int [][] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};
String [] sNums = {"one", "two", "three"};
Animal [] animals = {new Dog(), new Cat()};

// DECLARACIONES LEGALES DE for
for(long y : la) ; // loop thru an array of longs
for(long lp : La) ; // autoboxing the Long objects into longs
for(int[] n : twoDee) ; // loop thru the array of arrays
for(int n2 : twoDee[2]) ; // loop thru the 3rd sub-array
for(String s : sNums) ; // loop thru the array of Strings
for(Object o : sNums) ; // set an Object reference to
// each String
for(Animal a : animals) ; // set an Animal reference to each
// element

// DECLARACIONES ILEGALES DE for
for(x2 : la) ; // x2 is already declared
for(int x2 : twoDee) ; // can't stuff an array into an int
for(int x3 : la) ; // can't stuff a long into an int
for(Dog d : animals) ; // you might get a Cat!
```

63. Los labeled statements(declaración).

Muchas declaraciones en Java pueden ser etiquetadas. Los labeled son mas comunes utilizarlos con **for** o **while**.

Un labeled statement debe estar justo antes de la declaración y consiste en una palabra terminada con dos puntos ":".Por ejemplo:

```
foo:
    for (int x = 3; x < 20; x++) {
        while(y > 7) {
            y--;
        }
    }
```

64. Ahora debes conocer que **break** puede ser: **labeled** y **unlabeled break**. Y lo mismo ocurre con **continue**: **labeled** y **unlabeled continue**.

- Un ejemplo de uso de **labeled break**:

```
boolean isTrue = true;
outer:
for (int i = 0; i < 5; i++) {
    while (isTrue) {
        System.out.println("Hello");
        break outer; // Sale del outer y continua la
                      // siguiente linea System.out.println("Good-Bye");
    } // end of inner while loop
    System.out.println("Outer loop."); // Won't print
} // end of outer for loop
System.out.println("Good-Bye");
```

La salida es:

```
Hello
Good-Bye
```

- El siguiente ejemplo es similar al anterior pero en vez de **break** se utiliza **continue**.
- labeled continue:**

```
outer:
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        System.out.println("Hello");
        continue outer;
    } // end of inner loop
    System.out.println("outer"); // Never prints
}
System.out.println("Good-Bye");
```

La salida es:

```
Hello
Hello
Hello
Hello
Hello
Good-Bye
```

</OBJECTIVE 2.2 : Loops and Iterators>

<OBJECTIVE 2.4 AND OBJECTIVE 2.5 : Handling Exceptions>

65. El **finally** de los **try catch** siempre se va a ejecutar, por más que dentro del **try{} exista un return**. Solo no ejecutará si dentro del **try{} hay un System.exit(0);**

66. Puede no haber ningún un **catch(...){}**, pero si es así, necesariamente debe haber un **finally** (el **try{}** no puede estar sólo siempre debe haber de uno a muchos **catch o un finally o ambas**).

67. Puede haber mas de un **catch(...){}**.

68. Solo puede haber un **finally(){}**.

69. Stack de métodos:

FIGURE 5-1

The Java method call stack

1) The call stack while method3() is running.

4	method3()	method2 invokes method3
3	method2()	method1 invokes method2
2	method1()	main invokes method1
1	main()	main begins

The order in which methods are put on the call stack

2) The call stack after method3() completes

Execution returns to method2()

1	method2()	method2() will complete
2	method1()	method1() will complete
3	main()	main() will complete and the JVM will exit

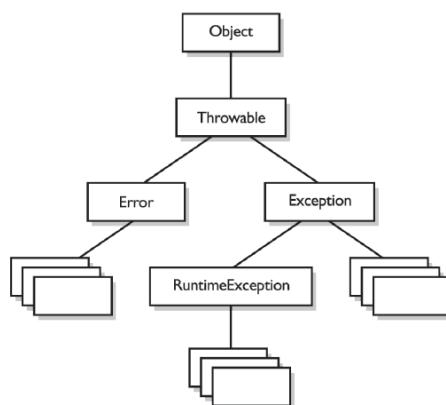
The order in which methods complete

70. Las excepciones de tipo aritméticas (**ArithmaticException**) como una división entre 0 son de tipo **unchecked exception**, que quiere decir que son detectadas en tiempo de ejecución. Recordar que **ArithmaticException** extiende de **RuntimeException**.

71. La jerarquía de excepciones:

FIGURE 5-2

Exception class hierarchy



Las clases que derivan de Error, representan situaciones inusuales que no son causadas por errores de programa, e indican cosas que normalmente no ocurren durante la ejecución de un programa, cosas como “JVM running out of memory”. Generalmente tu aplicación no será capaz de recuperarse de un Error. Los Errores no son manejados como excepciones ya que ellos no derivan de la clase Exception.

72. Para el examen, no es necesario conocer cualquiera de los métodos que figuran en la clase **Throwable**, incluidas las de **Exception** y **Error**. Se espera saber que tipo de Excepción, Error, RuntimeException, y Throwable pueden ser lanzados utilizando la palabras reservada **throw**, y saber que todas puedan ser capturados (aunque rara vez se capture algo más que los subtipos de **Exception**).
73. No es recomendable utilizar un solo catch y que éste pueda capturar cualquier excepción, me refiero a no hacer esto:

```
try {
    // some code
} catch (Exception e) {
    e.printStackTrace();
}
```

Este código capture cualquier tipo de excepción que genera. Por supuesto, ningún handler **Exception** puede manejar adecuadamente cada excepción, y la programación de esta forma va en contra del **objetivo del diseño**. Los Manejadores de excepciones(handler exception) que atrapan muchos errores a la vez probablemente reducirá la fiabilidad de su programa, porque es probable que una excepción será capturada y el handler no sabrá cómo manejarla.

74. Si hay mas de un catch, el primero no puede ser de mayor jerarquía que el segundo, por ejemplo el siguiente código marca error de compilación:

```
try {
    // do risky IO things
} catch (IOException e) {
    // handle general IOExceptions
} catch (FileNotFoundException ex) { //Compilation faild
    // handle just FileNotFoundException
}
```

Esto debido a que FileNotFoundException es subtipo de IOException.

75. La palabra reservada **throws**, tal como se usa en el siguiente código...:

```
void myMethod () throws IOException, FileNotFoundException {
    // code for the method here
}
```

...es para indicar que tipo de excepción puede ser lanzada por el método `myMethod()`.

Si por ejemplo existiese otro método :

```
void otherMethod () {  
    myMethod() // Marca error de compilación  
}
```

Y éste llama a `myMethod`, el compilador marcaría error debido a que `otherMethod` debe controlar la excepción que puede ocurrir al ejecutar el método `myMethod()`, para controlarla se logra usando un `try catch`:

```
void otherMethod () {  
    try{  
        myMethod()  
    }catch(FileNotFoundException e){  
  
    }catch(IOException e){  
  
    }  
}
```

Nota: El `catch` deben capturar todas las posibles excepciones que `myMethod()` pueda lanzar. Esto quiere decir que el siguiente código también es válido:

```
void otherMethod () {  
    try{  
        myMethod()  
    }catch(IOException e){  
  
    }  
}
```

Es válido, debido a que `IOException` puede capturar un `FileNotFoundException` y `IOException`.

Sino se desea controlar la excepción, lo único que queda por hacer es declarar con `throws` en el método `otherMethod()`:

```
void otherMethod() throws IOException, FileNotFoundException {  
    myMethod()  
}
```

Haciendo esto la responsabilidad de controlar la excepción quedaría en manos del método que llame a `otherMethod()`.

OJO CON LO SIGUIENTE:

El compilador marca error y obliga a controlar las excepciones solo con los checked exception, no con los unchecked exception(runtime exception). Si por ejemplo

```
void myMethod () throws IOException, FileNotFoundException {
    // code for the method here
}
```

Hubiera sido:

```
void myMethod () throws RuntimeException, ArithmeticException{
    // code for the method here
}
```

El compilador no hubiera marcado error y no hubiera obligado a declarar un try catch en el método otherMethod().

La explicación es: que todas las clases que estén por debajo de RuntimeException son excepciones lanzadas en tiempo de ejecución y no en tiempo de compilación.

76. La palabra reservada **throw**, se utiliza para lanzar excepciones manualmente y se puede declarar dentro de métodos, por ejemplo:

```
void doStuff() {
    doMore();
}
void doMore() {
    throw new IOException();
}
void doMore2() throws IOException {
try{
    throw new IOException();
}catch(IOException ex){
    throw ex;    //es válido porque ex es un objeto
}
}
```

Pero el código anterior marcaría error de compilación debido a que cuando se ejecute doMore() se va a lanzar una IOException y esta no va ser controlada por nadie, entonces el compilador obliga a que doMore defina la excepción, me refiero a:

```
void doMore() throws IOException{
    throw new IOException();
}
```

Nota: Puede ser el mismo IOException o una Clase Superior.

Pero al hacer esto, igual seguiría marcando error de compilación debido a que doStuff() debe controlar la excepción, y sucede todo lo explicado en el punto anterior, es todo una cadena =).

77. Los objetos de tipo `Error` no son objetos `Exception`, pero pueden ser representadas como tal en ocasiones excepcionales. `Exception` y `Error`, comparten el mismo parente y ambos pueden ser utilizados con la palabra reservada `throw`. Cuando un `Error` o subclase de `Error` es lanzado, esto es un `unchecked`, lo que significa que no es necesario capturarlo con un `catch`.

El siguiente código compila bien:

```
class TestEx {
    public static void main (String [] args) {
        badMethod();
    }
    static void badMethod() { // No need to declare an Error
        doStuff();
    }
    static void doStuff() { //No need to declare an Error
        try {
            throw new Error();
        }
        catch(Error me) {
            throw me; // We catch it, but then rethrow it
        }
    }
}
```

</ OBJECTIVE 2.4 AND OBJECTIVE 2.5 : Handling Exceptions >

<OBJECTIVE OBJECTIVE 2.6 : Common Exceptions and Errors>

<SUMMARY>

*Recognize situations that will result in any of the following being thrown:
ArrayIndexOutOfBoundsException, ClassCastException, IllegalArgumentException,
IllegalStateException, NullPointerException, NumberFormatException, AssertionError,
ExceptionInInitializerError, StackOverflowError, or NoClassDefFoundError. Understand
which of these are thrown by the virtual machine and recognize situations in which others
should be thrown programmatically*

</SUMMARY>

78. Para el propósito del examen, se define los exceptions y errors, en dos categorías:

- JVM exceptions:

Excepciones, lógicamente, lanzadas por la JVM.

- Programmatic Exceptions: Excepciones, explícitamente lanzadas por la aplicación y/o por los programadores de la API.

79. JVM Exceptions:

Solo la JVM conoce cuando este momento ocurre, y la JVM será el que origina esta excepción.

Una de las excepciones más comunes de este tipo es el **NullPointerException**. Sucede cuando queremos utilizar una variable de referencia con valor actual null. Ejemplo:

```
class NPE {  
    static String s;  
    public static void main(String [] args) {  
        System.out.println(s.length()); // NullPointerException  
    }  
}
```

Otra clase de este tipo es **StackOverflowError**. El cual es lanzado cuando la pila de la memoria esta llena. Si se ejecuta un método recursivo sin parar, entonces se provocará este error(**NO PROBAR EN CASA**) :

```
void go() { // recursion gone bad  
    go();  
}
```

80. Programmatic Exceptions

Muchas clases en el API de JAVA, tienen métodos que toman argumentos String, y convierten estos String en números. Un buen ejemplo de estas clases son las llamadas clases Wrapper es por eso que cuando uno utiliza el método **parseInt** de la clase **Integer** y le manda como parámetro un **String** esta operación requiere ser controlada con un **NumberFormatException**. Ejemplo:

```
int parseInt(String s) throws NumberFormatException {  
    boolean parseSuccess = false;  
    int result = 0;  
    // do complicated parsing  
    if (!parseSuccess) // if the parsing failed  
        throw new NumberFormatException();  
    return result;  
}
```

Otros ejemplos de excepciones programadas son el **AssertionError** y **IllegalArgumentException**, podríamos utilizar **IllegalArgumentException**, para controlar el **parseInt()**, debido a que **NumberFormatException** extiende de **IllegalArgumentException**, pero por explicaciones anteriores sabemos que se debe utilizar el más preciso, para este caso es **NumberFormatException**.

81. Tabla de excepciones mas comunes:

TABLE 5-2 Descriptions and Sources of Common Exceptions.

Exception (Chapter Location)	Description	Typically Thrown
ArrayIndexOutOfBoundsException (Chapter 3, "Assignments")	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
ClassCastException (Chapter 2, "Object Orientation")	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
IllegalArgumentException (Chapter 3, "Assignments")	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
IllegalStateException (Chapter 6, "Formatting")	Thrown when the state of the environment doesn't match the operation being attempted, e.g., using a Scanner that's been closed.	Programmatically
NullPointerException (Chapter 3, "Assignments")	Thrown when attempting to access an object with a reference variable whose current value is <code>null</code> .	By the JVM
NumberFormatException (Chapter 6, "Formatting")	Thrown when a method that converts a String to a number receives a String that it cannot convert.	Programmatically
AssertionError (This chapter)	Thrown when a statement's boolean test returns <code>false</code> .	Programmatically
ExceptionInInitializerError (Chapter 3, "Assignments")	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
StackOverflowError (This chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
NoClassDefFoundError (Chapter 10, "Development")	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing <code>.class</code> file.	By the JVM

</OBJECTIVE OBJECTIVE 2.6 : Common Exceptions and Errors>

<OBJECTIVE OBJECTIVE 2.3 : Working with the Assertion Mechanism>

<SUMMARY>

2.3 Develop code that makes use of assertions, and distinguish appropriate from inappropriate uses of assertions.

</SUMMARY>

82. Las aserciones les permite probar sus afirmaciones durante el desarrollo, veamos los siguientes códigos:

Ejemplo 01:

```
private void methodA(int x) {  
    if (x > 2) {  
        // código  
    } else if (x < 2) {  
        // código  
    } else {  
        // x debe ser 2  
        // código  
    }  
  
    while (true) {  
        if (x > 2) { //Si x no es mayor a 2, tendrás problemas  
            break;  
        }  
        System.out.print("Si entra aquí, YA fuiste");  
    }  
}
```

Ejemplo 02:

Supongamos que el número pasado al método `methodA` nunca debe ser negativo. Mientras haces testing y debugging, deseas validar esta afirmación pero lo que no quieres es utilizar los `if/else`, ni capturar excepciones, ni tampoco imprimir el valor del parámetro para saber si es positivo. Entonces la solución es utilizando `Asserts`.

Observa el siguiente código:

```
//problema:  
  
private void methodA(int num) {  
    if (num >= 0) {  
        useNum(num + x);  
    } else {  
        // num must be < 0  
        // This code should never be reached!  
        System.out.println("Yikes! num is a negative number! " + num);  
    }  
}  
//solución:  
private void methodA(int num) {  
    assert (num >= 0); // throws an AssertionError  
    // if this test isn't true  
    useNum(num + x);  
}
```

Las aceraciones las puedes hacer de 2 maneras:

Really simple:

```
private void doStuff() {  
    assert (y > x);  
    // more code assuming y is greater than x  
}
```

Simple:

```
private void doStuff() {  
    assert (y > x): "y is " + y + " x is " + x;  
    // more code assuming y is greater than x  
}
```

La forma Simple es para pasar un mensaje para mostrar en el stack trace(consola).

Las aserciones deben funcionar durante el desarrollo(-ea), pero cuando la aplicación es deploydada estas no deberían ser reconocidas.

83. Formas legales e ilegales de declarar aserciones:

```
void noReturn() {}  
int aReturn() { return 1; }  
void go() {  
    int x = 1;  
    boolean b = true;  
  
    // LEGALES:  
    assert(x == 1);  
    assert(b);  
    assert true;  
    assert(x == 1) : x;  
    assert(x == 1) : aReturn();  
    assert(x == 1) : new ValidAssert(); //any object  
  
    // ILEGALES:  
    assert(x = 1); // none of these are booleans  
    assert(x);  
    assert 0;  
    assert(x == 1) : ; // none of these return a value  
    assert(x == 1) : noReturn();  
    assert(x == 1) : ValidAssert va;  
}
```

84. Para que a la hora de ejecutar el programa, se reconosca las aserciones se necesita enviar este argumento en la JVM. VM arguments:

-enableassertions

O simplemente:

-ea

Y para desabilitarlos:

-disableassertions

O simplemente:

-da

Algunos comandos para habilitar y/o desabilitar asserciones:

Para habilitar aserciones menos en la clase Foo

```
java -ea -da:com.geeksanonymous.Foo
```

Para habilitar aserciones menos en el paquete com.geeksanonymous y todos sus subpaquetes.

```
java -ea -da:com.geeksanonymous...
```

TABLE 5-4 Assertion Command-Line Switches

Command-Line Example	What It Means
<code>java -ea</code> <code>java -enableassertions</code>	Enable assertions.
<code>java -da</code> <code>java -disableassertions</code>	Disable assertions (the default behavior of 1.5).
<code>java -ea:com.foo.Bar</code>	Enable assertions in class <code>com.foo.Bar</code> .
<code>java -ea:com.foo...</code>	Enable assertions in package <code>com.foo</code> and any of its subpackages.
<code>java -ea -dsa</code>	Enable assertions in general, but disable assertions in system classes.
<code>java -ea -da:com.foo...</code>	Enable assertions in general, but disable assertions in package <code>com.foo</code> and any of its subpackages.

85. El paquete que aloja la clase `AssertionError` es `java.lang`

86. Usando Aserciones apropiadamente:

- No uses try catch para capturar una aserción. Aunque lo podrías capturar con la clase `AssertionError` ya que esta es una subclase de `Throwable`, PERO no lo hagas.
- No uses aserciones para validar argumentos de un método público.

Don't Use Assertions to Validate Arguments to a Public Method

The following is an inappropriate use of assertions:

```
public void doStuff(int x) {  
    assert (x > 0);           // inappropriate!  
    // do things with x  
}
```

- Usa aserciones para validar argumentos de un método privado.
- No use aserciones para validar Command-Line Arguments (Argumentos de línea de comando).

- Usar aserciones, incluso en métodos públicos, para verificar **cases** que tu sabes que nunca van a ocurrir. Ejemplo:

```

switch (x) {
    case 1:
        y = 3;
        break;
    case 2:
        y = 9;
        break;
    case 3:
        y = 27;
        break;
    default:
        assert false; //suponemos que nunca debe entrar aquí
                        //porque si entra lanzaría un AssertionError
}

```

</OBJECTIVE OBJECTIVE 2.3 : Working with the Assertion Mechanism>

</CERTIFICATION_OBJETIVES>

</CAP5: Flow Control, Exceptions, and Assertions>

<CAP6: Strings, I/O, Formatting, and Parsing Certification Objectives>

<CERTIFICATION_OBJETIVES>

<OBJECTIVE OBJECTIVE 3.1 : Strings, I/O, Formatting, and Parsing>

<SUMMARY>

3.1 Discuss the differences between the String, StringBuilder, and StringBuffer classes.

</SUMMARY>

87. La clase *StringBuilder* fue añadida en la versión 5. es más veloz que el *StringBuffer*, pero es no-synchronized.

88. Hay que entender que un objeto **String** es inmutable y nunca puede ser cambiado.

89. En Java cada carácter de un String es un carácter Unicode de 16 bit.

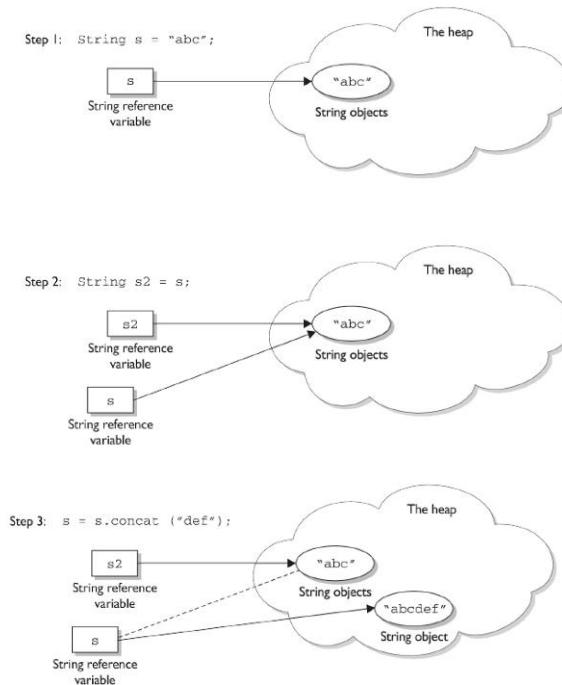
90. El siguiente ejemplo va mostrando como se van creando los **String**:

```
String s = "abcdef";    // create a new String object, with
                       // value "abcdef", refer s to it
String s2 = s;          // create a 2nd reference variable
                       // referring to the same String

// create a new String object, with value "abcdef more stuff",
// refer s to it. (Change s's reference from the old String
// to the new String.) ( Remember s2 is still referring to
// the original "abcdef" String.)

s = s.concat(" more stuff");
```

FIGURE 6-1 String objects and their reference variables



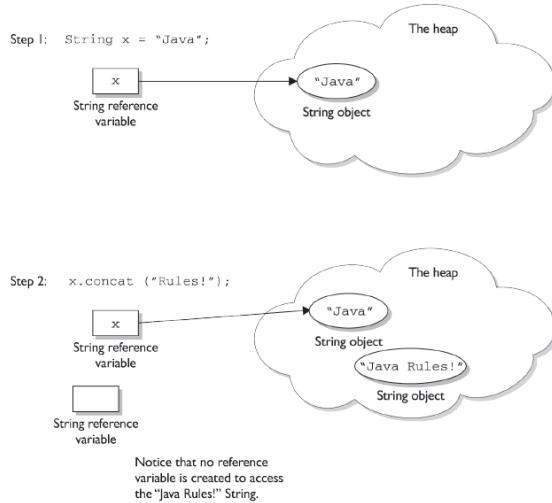
Otro ejemplo:

Let's look at another example:

```
String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x); // the output is "x = Java"
```

The first line is straightforward: create a new String object, give it the value "Java", and refer x to it. Next the VM creates a second String object with the value "Java Rules!" but nothing refers to it. The second String object is instantly lost; you can't get to it. The reference variable x still refers to the original String with the value "Java". Figure 6-2 shows creating a String without assigning a reference to it.

FIGURE 6-2 A String object is abandoned upon creation



91. La JVM reserva un espacio especial llamada **String Constant Pool**. Cuando el compilador encuentra un String Literal, este checa en el pool si existe un String idéntico, si lo encuentra la referencia del nuevo literal es directamente al **String** existente y ya no crea un nuevo **String**.

92. La clase **String** es **final**, por tal motivo no se puede extender de ella.

93. Creando nuevos String:

```
String s = "abc"; // creates one String object and one
// reference variable
```

In this simple case, "abc" will go in the pool and s will refer to it.

```
String s = new String("abc"); // creates two objects,
// and one reference variable
```

In this case, because we used the new keyword, Java will create a new String object in normal (non-pool) memory; and s will refer to it. In addition, the literal "abc" will be placed in the pool.

Ambos objetos van a ser referencia al literal "abc" que se encuentra en el String Pool.

94. Métodos importantes de la clase String.

- **charAt()** Devuelve el carácter situado en el índice especificado.
- **concat()** Añade un String al final, puede utilizarse también "+", para concatenar. Y devuelve un nuevo String.
- **equalsIgnoreCase()** Determinamos la igualdad de dos String, ignorando si son mayúsculas o minúsculas.

```
String x = "Exit";
System.out.println( x.equalsIgnoreCase("EXIT")); // is "true"
System.out.println( x.equalsIgnoreCase("tixe")); // is "false"
```

- **length()** Retorna el número de caracteres de un String.

```
String x = "01234567";
System.out.println( x.length() ); // returns "8"
```

e x a m

watch

Arrays have an attribute (not a method), called `length`. You may encounter questions in the exam that attempt to use the `length()` method on an array, or that attempt to use the `length` attribute on a String. Both cause compiler errors—for example,

```
String x = "test";
System.out.println( x.length ); // compiler error
```

or

```
String[] x = new String[3];
System.out.println( x.length ); // compiler error
```

- **replace()** Reemplaza la ocurrencia de un carácter con un nuevo carácter.

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') ); // output is "oXoXoXoX"
```

- **toLowerCase()** Retorna un String con los caracteres en minúscula.
- **toString()** Retorna el valor de un String.
- **toUpperCase()** Retorna un String con los caracteres en mayúscula.
- **trim()** Remueve los caracteres en blanco del inicio y del final un String.

```
String x = "    hi    ";
System.out.println(x + "x"); // result is " hi x"
System.out.println(x.trim() + "x"); // result is "hix"
```

- **substring(int begin), substring(int begin, int end)** Retorna un String que es una parte del String original.

Ojo: **begin**, indica índice(basado desde 0), y **end** indica posición (no basado desde 0).

95. Sobre las clases **java.lang.StringBuffer** y **java.lang.StringBuilder**.

Deben ser utilizadas cuando se tiene que hacer muchas modificaciones en los String, por ejemplo, cuando uno quiere hacer muchas **concatenaciones**.

Lo bueno de estas clases que trabajan a nivel de streams en memoria y no recargan al String Pool, lo que quiere decir que su valores son mutables.

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb ); // output is "fed---cba"
```

96. **StringBuilder** fue añadido en Java 5, cuenta con los mismos métodos que **StringBuffer**, pero la diferencia es que **StringBuilder** es **not thread safe**, en otras palabras, no tiene métodos sincronizados (Más acerca de thread safe en el *capítulo 9*).

SUN, recomienda utilizar **StringBuilder** siempre que sea posible, ya que es más rápido.

97. Métodos importantes de **StringBuilder** y **StringBuffer**:

- public synchronized StringBuffer **append**(String s).

```
StringBuffer sb = new StringBuffer("set ");
sb.append("point");
System.out.println(sb); // output is "set point"
StringBuffer sb2 = new StringBuffer("pi = ");
sb2.append(3.14159f);
System.out.println(sb2); // output is "pi = 3.14159"
```

- public StringBuilder **delete**(int start, int end). El primer argumento es basado desde 0, el segundo argumento no es basado desde 0. Ejemplo:

```
StringBuilder sb = new StringBuilder("0123456789");
System.out.println(sb.delete(4,6)); // output is "01236789"
```

- public StringBuilder **insert**(int offset, String s). El primer parámetro es basado en 0.

```
StringBuilder sb = new StringBuilder("01234567");
sb.insert(4, "---");
System.out.println( sb ); // output is "0123---4567"
```

- public synchronized StringBuffer **reverse**().

```
StringBuffer s = new StringBuffer("A man a plan a canal Panama");
s.reverse();
System.out.println(s); // output: "amanaP lanac a nalp a nam A"
```

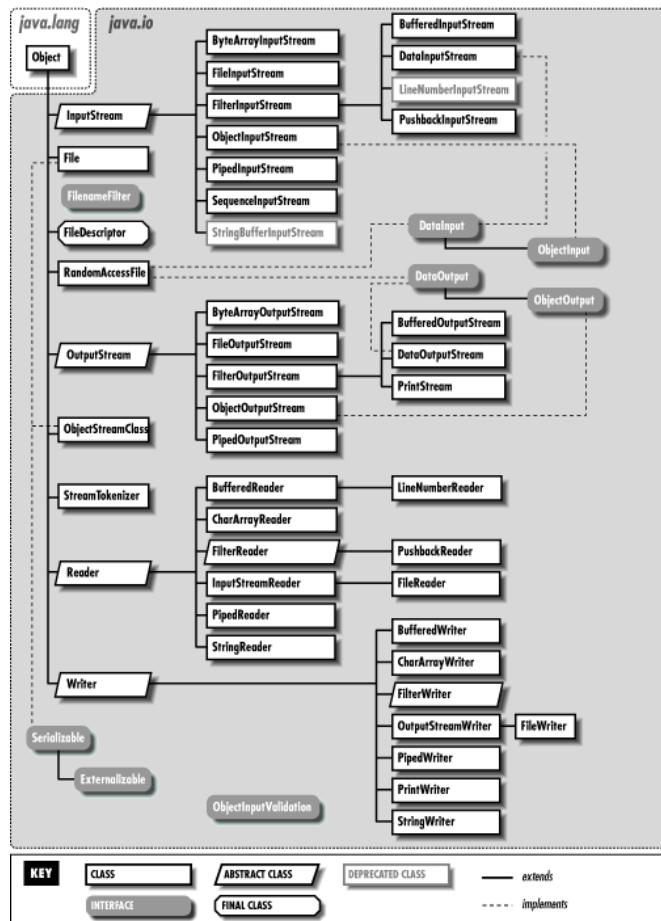
- public String **toString**()

```
StringBuffer sb = new StringBuffer("test string");
System.out.println( sb.toString() ); // output is "test string"
```

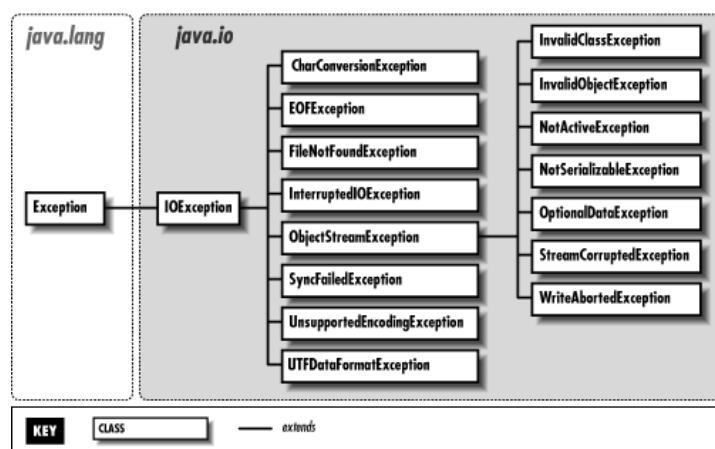
<OBJECTIVE 3.2 : File Navigation and I/O >

3.2 Given a scenario involving navigating file systems, reading from files, or writing to files, develop the correct solution using the following classes (sometimes in combination), from java.io: BufferedReader, BufferedWriter, File, FileReader, FileWriter, and PrintWriter.

98. Jerarquía de clases del paquete IO:



99. Jerarquía de clases para control de Excepciones de tipo IO:



100. Acá un resumen de todas las clases que necesitas saber para el examen:

File, FileReader, BufferedReader, FileWriter, BufferedWriter, PrintWriter, Console.

- **File:** El API dice que la clase **File** es una representación abstracta de un archivo y rutas de directorio. La clase **File** no es utilizada para escribir o leer data, es usada para trabajar en alto nivel, puedes crear nuevos archivos, buscar archivos, borrar archivos, crear directorios, y trabajar con rutas.

Ejemplo de cómo crear un archivo:

```
import java.io.*;

class Writer1 {
    public static void main(String[] args) {
        try { // warning: exceptions possible
            boolean newFile = false;
            File file = new File("fileWrite1.txt"); // it's only an object
            System.out.println(file.exists()); // look for a real file
            newFile = file.createNewFile(); // maybe create a file!
            System.out.println(newFile); // already there?
            System.out.println(file.exists()); // look again
        } catch (IOException e) {
        }
    }
}
```

Nota: cuando trabajes con rutas como `"directory1/file3.txt"`, en la creacion del objeto `File`, no te preocunes si lo haces de esa manera o de esta: `"directory1\\file3.txt"`, automáticamente el API lo transformará en un path correcto según el sistema operativo en el que está corriendo la aplicación.

Si quieres quieres armar paths, con el separador correcto, podrías utilizar `File.separator`.

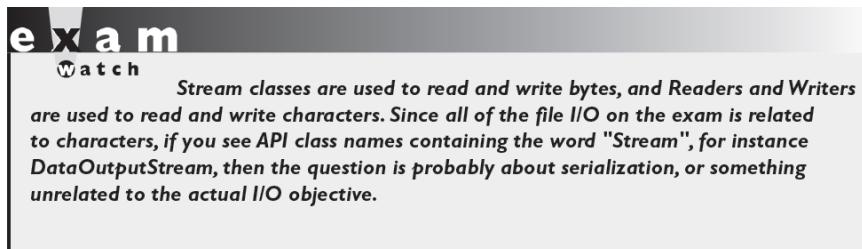
- **FileReader:** Esta clase es usada para leer archivos de caracteres. Su método **read()** relativamente es de bajo nivel, permitiendo que tú, leas todo el flujo de caracteres, o un número fijo de caracteres.

- **BufferedReader:** Esta clase es utilizada para hacer las clases **Reader** que son de bajo nivel, más eficientes y fáciles de usar. Comparando con los **FileReader**, **BufferedReader** puede leer trozos relativamente grandes de datos de un archivo a la vez.

BufferedReader provee de métodos más convenientes como el **readLine()**, que permite que tu obtengas la siguiente línea de caracteres de un archivo.

Nota: Su constructor solo espera como parámetro un objeto que esté por debajo de la jerarquía de la clase **Reader**.

- **FileWriter:** Esta clase es usada para escribir archivos de caractes, su método **write()**, permite escribir caracteres o cadenas(Strings) a un archivo. Usualmente son utilizados por **BufferedWriters** y **PrintWriters**, quienes proveen un mejor rendimiento y métodos mas flexibles para escribir data.
- **BufferedWriter:** Esta clase es usada para hacer las clases de bajo nivel como **FileWriters** mas eficientes y fáciles de usar. Comparando con los **FileWriter**, **BufferedWriter** puede escribir trozos relativamente grandes de datos en un archivo a la vez.
BufferedWriter provee de un método **newLine()** que hace mas facil la escritura por líneas en un archivo.
Nota: Su constructor solo espera como parámetro un objeto que esté por debajo de la jerarquía de la clase Writer.
- **PrintWriter:** Esta clase se ha mejorado significativamente en Java 5. Puedes utilizarlo en lugares donde previamente has utilizado **FileWriter** y/o **BufferedWriter**.
Cuenta con nuevos métodos como: **format()**, **append()**, **printf()** haciendo de esta clase potente y facil de usar.
- **Console:** Esta clase es nueva de Java SE 6, proporciona métodos para leer desde la consola y escribir el formato de salida a la consola.



101. Usando FileWriter y FileReader

En la práctica, tu probablemente no usarás las clases **FileWriter** y **FileReader** sin un wrapping como **BufferedWriter** y **BufferedReader** (De esto, mas adelante).

Un ejemplo de su uso:

```
class Writer2 {
    public static void main(String[] args) {
        char[] in = new char[50]; // para almacenar la entrada
        int size = 0;
        try {
            File file = new File("fileWrite2.txt"); // solo es un objeto
            FileWriter fw = new FileWriter(file); // crea un archivo actual
            fw.write("howdy\nfolks\n");
            fw.flush(); // confirma los cambios en el archivo
            fw.close(); // cerrar el archivo

            FileReader fr = new FileReader(file); // crea un objeto FileReader
            size = fr.read(in); // leer todo el archivo!
            System.out.print(size + " ");
            for (char c : in) // imprime el array
                System.out.print(c);
            fr.close(); // de nuevo, se cierra.
        } catch (IOException e) {
        }
    }
}
```

Lo que produce:

```
12 howdy
folks
```

Esto es lo que ocurre:

- 1) `FileWriter fw = new FileWriter (archivo)` hizo tres cosas:
 - a. Se creó una variable de referencia `FileWriter`, `fw`.
 - b. Se creó un objeto `FileWriter`, y le asignó a `fw`.
 - c. Se creó un verdadero archivo vacío en el disco.
- 2) Se escribió 12 caracteres en el archivo con el método `write()`, y se hace un `flush()` y un `close()`.
- 3) Se Hace un nuevo objeto `FileReader`, el cual también es abierto en disco para su lectura.
- 4) El método `read()` lee todo el archivo, un carácter por carácter a la vez, y lo pone dentro del arreglo de `char[]`.
- 5) Se imprime el numero de caracteres que lee `size`, y recorre por el array `in`, imprimiendo cada carácter que lee, luego cierra el archivo.

Nota: Como puedes observar, por ejemplo en la escritura, para hacer un salto de linea tienes que hacer `\n` dentro de la cadena a imprimir, y en la lectura, tienes que tener un array de bytes previamente dimensionado para poder cargar los valores del archivo. Todo esto son limitaciones, por lo que en la mayoría de veces querrás usar más alto nivel de clases I/O como `BufferedWriter` `BufferedReader` o en combinación con o `FileReader` `FileWriter`

102. Sobre el `flush()` y `close()`:

Cuando se escriben los datos en un stream, tu nunca sabrás con exactitud cuando el último de los datos realmente se enviará.

Invoca al método `flush()`, para garantizar que el último de los datos va a estar realmente en el archivo. Cada vez que hayas terminado con un archivo, ya sea leyendo o escribiendo en el mismo, debes invocar al método `close()`, para liberar recursos del sistema operativo.

103. Mini API de java.io

TABLE 6-1 java.io Mini API			
java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format()*, printf()*, print(), println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()

*Discussed later

En la clase `File`, el método `exists()` verifica si existe tanto archivos como carpetas, sin embargo el método `isFile()`, solo verifica si existe el archivo y el método `isDirectory()`, si es un directorio.

104. Hay dos formas de crear un archivo usando la clase **File**:

- 1) Invocando al método **createNewFile()**, del objeto File

```
File file = new File("foo"); // no file yet
    file.createNewFile(); // make a file, "foo" which
        // is assigned to 'file'
```

- 2) Creando un objeto PrintWriter o FileWriter a partir del objeto File, al hacer esto, automáticamente se crea el archivo:

```
File file = new File("foo"); // no file yet
PrintWriter pw =
new PrintWriter(file); // make a PrintWriter object AND
// make a file, "foo" to which
// 'file' is assigned, AND assign
// 'pw' to the PrintWriter
```

105. Ejemplo de escritura y lectura de un archivo:

```
File file = new File("filewriter2.txt");
FileWriter fw = new FileWriter(file);
PrintWriter pw = new PrintWriter(fw);
pw.println("holá");
pw.println("mundo");

// Leemos la data así:
FileReader fr = new FileReader(file);
BufferedReader br = new BufferedReader(fr);
String data = br.readLine();
```

106. Ejemplos de trabajo con directorios:

- 1) Creación de un directorio y un archivo dentro de él:

```
File myDir = new File("myDir");
myDir.mkdir();

// una vez creado el directorio, ya podemos poner archivos en él
File myFile = new File(myDir, "myFile.txt");
myFile.createNewFile();
// escribiendo data
PrintWriter pw = new PrintWriter(myFile);
pw.println("new example");
pw.flush();
pw.close();
```

- 2) Borrar un directorio o renombrarlo:

```
File myDir = new File("myDir");
myDir.delete(); //Falla si es que hay archivos dentro del directorio
```

3) Borrar un directorio con archivos en su interior(Función recursiva):

```
class DeleteDir {
    public static void main(String args[]) {
        deleteDirectory(new File("myDir"));
    }

    static public boolean deleteDirectory(File path) {
        if (path.exists()) {
            File[] files = path.listFiles();
            for (int i = 0; i < files.length; i++) {
                if (files[i].isDirectory()) {
                    deleteDirectory(files[i]);
                } else {
                    files[i].delete();
                }
            }
        }
        return (path.delete());
    }
}
```

4) Renombrar directorios, Mover archivos:

```
String oldDir = "C:\\oldDir\\";
String oldName = "oldName.txt";

String newDir = "C:\\oldDir\\subDir\\";
String newName = "newName.txt";

File fOld = new File (oldDir, oldName);
File fNew = new File (newDir, newName);

fOld.renameTo(fNew);
```

5) Listar Archivos:

```
File existingDir = new File("existingDir");

String[] files = existingDir.list();

for(String sn: files){
    System.out.println(sn);
}
```

107. Busqueda de archivo e impresión de contenido:

```
public static void main(String[] args) {
    try {
        File existingDir = new File("existingDir");
        System.out.println(existingDir.isDirectory());
        File existingDirFile = new File(existingDir, "archivo.txt");
        System.out.println(existingDirFile.isFile());
        FileReader fr = new FileReader(existingDirFile);
        BufferedReader br = new BufferedReader(fr);
        String s;
        while ((s = br.readLine()) != null) {
            System.out.println(s);
        }
        br.close();
    } catch (IOException e) { // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

108. Sobre la clase **Console** (Nuevo de Java 6)

Si estás utilizando Java SE 6, desde línea de comandos, por lo general necesitarás tener acceso a una consola de objeto, y lo puedes hacer invocando `System.console()`. Pero este método devuelve null si tu programa se ejecuta en un entorno que no tiene acceso a una consola de objetos, cuidado con este punto.

La clase `Console`, hace fácil aceptar la entrada desde línea de comandos, tanto de forma `echoed` y `nonechoed`(como una contraseña), y hace que sea fácil de escribir el formato de salida a la línea de comandos. Es una manera práctica de escribir los motores de pruebas para hacer pruebas unitarias sin necesidad de que se interactúe con una interfaz gráfica.

Métodos importantes:

public String `readLine()`

Lee un sola línea de texto de la consola.

public String `readLine(String fmt, Object ... args)`

Proporciona un formato de sistema, y a continuación, lee una sola línea de texto desde la consola.

public char[] `readPassword()`

Lee un password de la consola(no se ven los caracteres que se van digitando en la consola).

public char[] `readPassword(String fmt, Object ... args)`

Proporciona un formato de sistema, y a continuación, lee un password de la consola.

Ejemplos de uso:

1)

```
import java.io.Console;
class Main {

    public static void main(String[] args) {
        Console console = System.console();
        String username = console.readLine("Username: ");
        char[] password = console.readPassword("Password: ");

        if (username.equals("admin") && String.valueOf(password).equals("secret")) {
            console.printf("Bienvenido");
        } else {
            console.printf("Error en login o password");
        }
    }
}
```

2)

```
class NewConsole {
    public static void main(String[] args) {
        String name;
        Console c = System.console(); // #1: get a Console
        char[] pw;
```

```

pw = c.readPassword("%s", "pw: "); // #2: return a char[]
for (char ch : pw)
    c.format("%c ", ch); // #3: format output
c.format("\n");
MyUtility mu = new MyUtility();
while (true) {
    name = c.readLine("%s", "input?: ");
    c.format("output: %s \n", mu.doStuff(name));
}
}

class MyUtility { // #5: class to test
String doStuff(String arg1) {
    // stub code
    return "result is " + arg1;
}
public String readLine(String fmt, Object ... args){
    return null;
}
}

```

</OBJECTIVE 3.2 : File Navigation and I/O >

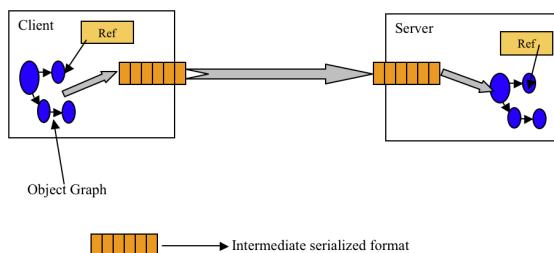
<OBJECTIVE 3.3: Serialization >

<SUMMARY>

3.3 Develop code that serializes and/or de-serializes objects using the following APIs from `java.io`: `DataInputStream`, `DataOutputStream`, `FileInputStream`, `FileOutputStream`, `ObjectInputStream`, `ObjectOutputStream`, and `Serializable`.

</SUMMARY>

Imagina que tienes dos programas java corriendo en dos máquinas diferentes en una misma red, y quieres desde el programa 1, crear un objeto de la clase Cliente y quieres pasar esa información al programa 2. ¿Como lo harías?. La respuesta es serializando el objeto Cliente. ¿Y como lo serializo?, ¿y como lo paso por la red?, ¿Si no quiero transmitir todas las propiedades del objeto Cliente, que tendría que hacer?.



109. Trabajando con `ObjectOutputStream` y `ObjectInputStream`

La magia de la serialización sucede con dos métodos, uno que serializa los objetos y los escribe en un **Stream** y otro que lee un **Stream** y deserializa los objetos.

```
ObjectOutputStream.writeObject() // serializa y escribe  
ObjectInputStream.readObject() // lee y deserializa
```

Ejemplo de un programa que serializa un objeto Cat y lo deserealiza:

1) Caso 1, serializa un objeto Cat sin propiedades :

```
import java.io.*;  
  
class Cat implements Serializable {  
} // 1  
  
class SerializeCat {  
    public static void main(String[] args) {  
        Cat c = new Cat(); // 2  
        try {  
            FileOutputStream fs = new FileOutputStream("testSer.ser");  
            ObjectOutputStream os = new ObjectOutputStream(fs);  
            os.writeObject(c); // 3  
            os.close();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        try {  
            FileInputStream fis = new FileInputStream("testSer.ser");  
            ObjectInputStream ois = new ObjectInputStream(fis);  
            c = (Cat) ois.readObject(); // 4  
            ois.close();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

- 2) Caso 2, serializa un objeto Cat con 2 propiedades, nombre y descripcion, esta última marcada como **transient**:

```
import java.io.*;

class Cat implements Serializable {
    String nombre;
    transient String descripcion;
} // 1

class SerializeCat {
    public static void main(String[] args) {
        Cat c = new Cat(); // 2
        c.nombre = "michifu";
        c.descripcion = "esta es una descripcion del gato michifu";
        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(c); // 3
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            c = (Cat) ois.readObject(); // 4
            System.out.println("nombre: " + c.nombre);
            System.out.println("descripcion: " + c.descripcion);
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

La salida es :

```
nombre: michifu
descripcion: null
```

Y que pasó con la descripción????.

La respuesta es que no se consideró en la serialización debido a que se marcó como **transient**.

- 3) Caso 3, serializa un objeto Cat con 1 propiedad Collar:

```
import java.io.*;

class Collar {
    String largo;
    public Collar(String largo) {
        this.largo = largo;
    }
}
class Cat implements Serializable {
    Collar collar;
} // 1

class SerializeCat {
    public static void main(String[] args) {
        Cat c = new Cat(); // 2
        c.collar = new Collar("50cm");
        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(c); // 3
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            c = (Cat) ois.readObject(); // 4
            System.out.println("largo del collar: " + c.collar.largo);
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Salida:

java.io.NotSerializableException: api_contents.Collar

Esto debido a que se quiere serializar un objeto **Cat** que tiene una propiedad **Collar** que no puede serializarse (NO IMPLEMENTA Serializable). Solución: Que **Collar** implemente la interface Serializable.

- 4) Caso 4, Clase Hija: Cat es serializable y clase padre:Animal no lo es.

```

class Animal {
    String grupo;
}

class Cat extends Animal implements Serializable {
    String nombre;
}

class SerializeCat {
    public static void main(String[] args) {
        Cat c = new Cat(); // 2
        c.nombre = "Michifu";
        c.grupo = "Mamifero";
        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(c); // 3
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            c = (Cat) ois.readObject(); // 4
            System.out.println("grupo: " + c.grupo);
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Salida:

grupo: null

Muestra null porque La clase padre: Animal, no es serializable. Por tal sus propiedades que son pasadas al hijo tampoco, y son consideradas en la serialización, PERO con sus valores **iniciales**.

¿Valores iniciales..?

Si hubieras puesto la clase Animal de la siguiente manera:

```

class Animal {
    String grupo = "grupo inicial";
}

```

Entonces la salida hubiera sido :

grupo: grupo inicial

¿Pero como sabe Cat que Animal debe tener los valores iniciales?

RESPUESTA::: En el momento de deserializar Cat(clase hija), si Animal(clase padre) no es serializable, entonces se crea una instancia de Animal (llama al constructor por defecto de la clase Padre) **OJO CON ESTO**.

En cambio si Animal fuera serializable, no se crearía ninguna instancia de Animal.

5) Caso 5, Clase **Cat** tiene una propiedad de tipo **static** y se quiere serializar.

¿Qué va a suceder?, La respuesta es que siempre va a mostrar el último valor estático asignado. Debido a que:

Las variables staticas no son almacenadas dentro del estado del objeto, ya que no son parte del objeto sino de la clase. Así que cuando se deserealice un objeto con una variable statica, esta variable tendrá el valor actual que tenga en la clase.

////////// Ejemplo de como grabar un objeto serializado en tabla(array de bytes)

```
public void saveRowDeleted(ObjetoBase entity) { //entity puede ser Customer u otro bean
    Auditoria auditoria = new Auditoria(new Usuario(new Long(1)));
    try {
        ByteArrayOutputStream ba = new ByteArrayOutputStream();
        ObjectOutputStream os = new ObjectOutputStream(ba);
        os.writeObject(entity); // 3
        os.close();
        byte[] binario = ba.toByteArray();
        auditoria.setFormato_serializado(binario);
        auditoria.setFormato_json(new JSONObject(entity).toString());
        auditoria.setOperacion("DELETE");
        auditoria.setQuery(entity.getClass().getName());
        guardar(auditoria);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

110. Diagrama de Clases de Streams de Bytes

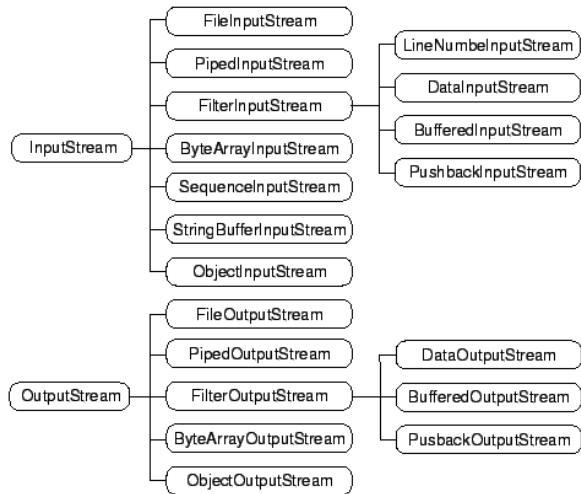


Figura: Diagrama de las Clases de Streams de Bytes

e x a m
watch

If you serialize a collection or an array, every element must be serializable! A single non-serializable element will cause serialization to fail. Note also that while the collection interfaces are not serializable, the concrete collection classes in the Java API are.

111. Preguntas sobre **DataInputStream** y **DataOutputStream**, no son consideradas en el examen.

112. La versión del .class de la clase cuando se serializa debe ser igual a la version del .class de la clase cuando se deserializa. Esto quiere decir que si serializas un objeto con la versión 4 de java y la quieres deserializar con la versión 5 de java, te arrojaría error.

113. NOTAA: la interface **java.io.Externalizable** es una interface que hereda de **java.io.Serializable**, pero si una clase la implementa, **hace que esta clase tenga un completo control sobre su serialización**. Esta interface obliga a sobreescribir los siguientes métodos:

```
public void readExternal(ObjectInput in)      y  
public void writeExternal(ObjectOutput out)
```

</OBJECTIVE 3.3 : Serialization >

<OBJECTIVE 3.4 : Dates, Numbers, and Currency >
<SUMMARY>

3.4 Use standard J2SE APIs in the java.text package to correctly format or parse dates, numbers and currency values for a specific locale; and, given a scenario, determine the appropriate methods to use if you want to use the default locale or a specific locale.
Describe the purpose and use of the java.util.Locale class.

</SUMMARY>

114. Trabajando con Dates,Numbers y Currencies:

Para esto, primero hay que familiarizarse con las clases de java.text y java.util.

Aquí hay cuatro clases relacionadas con la fecha que necesitas comprender:

- **java.util.Date.** Tu puedes usar esta clase como puente entre la clase Calendar y la clase DateFormat.

Una instancia de Date representa una fecha y hora de acuerdo milisegundos.

La clase Date internamente en un long que almacena la hora y la fecha.

Desde el 1 enero de 1970 a la fecha actual

Ejemplo de cambio de fecha:

```
public static void main(String[] args) {  
    Date d1 = new Date(); // un trillon  
    System.out.println("fecha actual: " + d1.toString());  
    d1.setTime(d1.getTime() + 1*24*60*60*1000); // Se suma un dia  
    System.out.println("nueva fecha: " + d1.toString());  
}
```

- **java.util.Calendar.** Esta clase provee una enorme variedad de métodos que te ayudan a convertir y manipular fechas y horas. Por ejemplo, si se deseas añadir un mes a una determina fecha o quieres saber que día de la semana cae el 1ero de Enero en el año 3000, los métodos de **Calendar** te van a ser de mucha ayuda.

Una subclase de Calendar es: **java.util.GregorianCalendar**

Consulta en www.icu-project.org para trabajar con otras clases concretas como Judaismo, Budismo, Islamico, Hindue, etc

- **java.text.DateFormat.** Esta clase es usada para dar formato a fechas, no solo para dar formatos como "01/01/70" o "January 1, 1970," sino tambien dar formato de acuerdo a numerosos lugares alrededor del mundo.
- **java.text.NumberFormat.** Esta clase es usada para dar formato a números y monedas para localidades alrededor del mundo.

- **java.util.Locale** Esta clase es usada en conjunción con **DateFormat** y **NumberFormat** para dar formato a fechas, números y monedas para una localidad específica.

Con la ayuda de **Locale** podrás convertir fechas como "10/08/2005" a "Segunda-feira, 8 de Outubro de 2005"(Fecha en portugués). Si tu quieres manipular fechas sin producir formatos de salida, tu puedes utilizar la clase **Locale** directamente con **Calendar**.

115. Usos comunes

TABLE 6-2 Common Use Cases When Working with Dates and Numbers	
Use Case	Steps
Get the current date and time.	1. Create a Date: <code>Date d = new Date();</code> 2. Get its value: <code>String s = d.toString();</code>
Get an object that lets you perform date and time calculations in your locale.	1. Create a Calendar: <code>Calendar c = Calendar.getInstance();</code> 2. Use <code>c.add(...)</code> and <code>c.roll(...)</code> to perform date and time manipulations.
Get an object that lets you perform date and time calculations in a different locale.	1. Create a Locale: <code>Locale loc = new Locale(language);</code> or <code>Locale loc = new Locale(language, country);</code> 2. Create a Calendar for that locale: <code>Calendar c = Calendar.getInstance(loc);</code> 3. Use <code>c.add(...)</code> and <code>c.roll(...)</code> to perform date and time manipulations.
Get an object that lets you perform date and time calculations, and then format it for output in different locales with different date styles.	1. Create a Calendar: <code>Calendar c = Calendar.getInstance();</code> 2. Create a Locale for each location: <code>Locale loc = new Locale(...);</code> 3. Convert your Calendar to a Date: <code>Date d = c.getTime();</code> 4. Create a DateFormat for each Locale: <code>DateFormat df = DateFormat.getDateInstance(style, loc);</code> 5. Use the <code>format()</code> method to create formatted dates: <code>String s = df.format(d);</code>
Get an object that lets you format numbers or currencies across many different locales.	1. Create a Locale for each location: <code>Locale loc = new Locale(...);</code> 2. Create a NumberFormat: <code>NumberFormat nf = NumberFormat.getInstance(loc);</code> -or- <code>NumberFormat nf = NumberFormat.getCurrencyInstance(loc);</code> 3. Use the <code>format()</code> method to create formatted output: <code>String s = nf.format(someNumber);</code>

116. Cortitas:

- `DateFormat.getDateInstance()` devuelve un `DateFormat` de acuerdo al locale por defecto.
- `DateFormat.getInstance()` devuelve un `DateFormat` por defecto que usa el estilo SHORT para la fecha y el tiempo.
- Uno de las sobrecargas del método `Calendar.set([year],[month],[day])` sirve para especificar una fecha de acuerdo a los parámetros pasados. Tener cuidado que [month] es basado en 0, lo que quiere decir que si se indica un 11 quiere decir el mes diciembre.
- Los métodos `getDisplayCountry()` y `getDisplayLanguage()` obtienen el país y el lenguaje actual y lo muestra en el idioma por defecto, pero si se utilizarán de la siguiente manera `getDisplayCountry(locUs)` y `getDisplayLanguage(locUs)`, obtienen el país y el lenguaje actual pero esta vez lo muestra en el idioma que se indica como parámetro.
- **NumberFormat.getInstance()** obtiene el formato de coma flotante actual. **NumberFormat.getInstance(locUs)** obtiene el formato de coma flotante de Estados Unidos. **NumberFormat.getCurrencyInstance()** obtiene el formato de coma flotante

mas el símbolo de moneda actual. `NumberFormat.getInstance(locUS)` obtiene el formato de coma flotante más el simbolo de moneda de Estados Unidos.

- En este ejemplo se muestra el uso de los métodos `getMaximumFractionDigits()`, `setMaximumFractionDigits()`, `parse()`, and `setParseIntegerOnly()`:

```
float f1 = 123.45678f;
NumberFormat nf = NumberFormat.getInstance();
System.out.println(nf.getMaximumFractionDigits() + " ");
System.out.println(nf.format(f1) + " ");
nf.setMaximumFractionDigits(5);
System.out.println(nf.format(f1) + " ");
try {
    System.out.println(nf.parse("1234,567"));
    nf.setParseIntegerOnly(true);
    System.out.println(nf.parse("1234,567"));
} catch (ParseException pe) {
    System.out.println("parse exc");
}
```

This, on our JVM, produces

```
3
123,457
123,45678
1234.567
1234
```

- Formas de instanciar las clases de `java.util` y `java.text`

TABLE 6-3 Instance Creation for Key `java.text` and `java.util` Classes

Class	Key Instance Creation Options
<code>util.Date</code>	<code>new Date();</code> <code>new Date(long millisecondsSince010170);</code>
<code>util.Calendar</code>	<code>Calendar.getInstance();</code> <code>Calendar.getInstance(Locale);</code>
<code>util.Locale</code>	<code>Locale.getDefault();</code> <code>new Locale(String language);</code> <code>new Locale(String language, String country);</code>
<code>text.DateFormat</code>	<code>DateFormat.getInstance();</code> <code>DateFormat.getDateInstance();</code> <code>DateFormat.getDateInstance(style);</code> <code>DateFormat.getDateInstance(style, Locale);</code>
<code>text.NumberFormat</code>	<code>NumberFormat.getInstance()</code> <code>NumberFormat.getInstance(Locale)</code> <code>NumberFormat.getNumberInstance()</code> <code>NumberFormat.getNumberInstance(Locale)</code> <code>NumberFormat.getCurrencyInstance()</code> <code>NumberFormat.getCurrencyInstance(Locale)</code>

- Ejemplos de Fechas:

1) Uso de `DateFormat`:

```
public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);
    DateFormat[] dfa = new DateFormat[6];
    dfa[0] = DateFormat.getInstance();
    dfa[1] = DateFormat.getDateInstance();
    dfa[2] = DateFormat.getDateInstance(DateFormat.SHORT);
    dfa[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
    dfa[4] = DateFormat.getDateInstance(DateFormat.LONG);
    dfa[5] = DateFormat.getDateInstance(DateFormat.FULL);
    for (DateFormat df : dfa)
        System.out.println(df.format(d1));
}
```

</OBJECTIVE 3.4 : Dates, Numbers, and Currency >

<OBJECTIVE 3.5 : Parsing, Tokenizing, and Formatting>

<SUMMARY>

3.5 Write code that uses standard J2SE APIs in the java.util and java.util.regex packages to format or parse strings or streams. For strings, write code that uses the Pattern and Matcher classes and the String.split method. Recognize and use regular expression patterns for matching (limited to: .(dot), *(star), +(plus), ?, \d, \s, \w, [], ()). The use of *, +, and ? will be limited to greedy quantifiers, and the parenthesis operator will only be used as a grouping mechanism, not for capturing content during matching. For streams, write code using the Formatter and Scanner classes and the PrintWriter.format/printf methods. Recognize and use formatting parameters (limited to: %b, %c, %d, %f, %s) in format Strings.

</SUMMARY>

Puntos a ver:

- Escribir código que use las APIs de java.util, java.util.regex para formatear, o parsear strings o streams.
- Veremos String.split
- Usar regular expressions
- Para streams: Usar Formatter, Scanner y PrintWriter.format /printf
- Reconocer parámetros de formateo como (%b, %c, %d, %f, %s) en formateo de Strings

117. Patterns:

Lo que viene a continuación no es más que la traducción del la documentación de una parte de la clase Pattern. Para una referencia completa puede consultar la versión en inglés de Sun inc. sobre la clase Pattern en:

<http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html>

Ejemplo de uso de Pattern:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class RegexSmall {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("ab"); // the expression
        Matcher m = p.matcher("abaaaba"); // the source
        while (m.find()) {
            System.out.print(m.start() + " ");
        }
    }
}
```

La salida es:

0 4

find() , Busca de izq a der. Y una vez que encuentra un patrón este no es considerado mas.

Cuadro de expresiones:

Expresión	Encaja con
X	El carácter x
\	El carácter \
\On	El carácter con valor octal 0n (0 <= n <= 7)
\Onn	El carácter con valor octal 0nn (0 <= n <= 7)
\Omn	El carácter con valor octal 0mnn (0 <= m <= 3, 0 <= n <= 7)
\xhh	El carácter con valor hexadecimal 0xhh
\uhhhh	El carácter con valor hexadecimal 0xhhhh
\t	El tabulador ('\u0009')
\n	Nueva linea (line feed) ('\u000A')
\r	Retorno de carro ('\u000D')
\f	Nueva pagina ('\u000C')
\a	Un beep de alerta (bell) ('\u0007')
\e	Escape ('\u001B')
\cx	El carácter de control que corresponde a x
Intervalos de caracteres	
[abc]	a, b, o c
[^abc]	Cualquier carácter excepto a, b, o c (negación)
[a-zA-Z]	Desde la a a la z o desde la A hasta la Z, incluidos
[a-d[m-p]]	Desde la a hasta la d, o desde la m a la p: [a-dm-p] (unión)
[a-z&&[def]]	La d, la e, o la f (intersección)
[a-z&&[^bc]]	Desde la a hasta la z, excepto la b y la c: [ad-z] (resta)
[a-z&&[^m-p]]	Desde la a hasta la z, excepto desde la m hasta la p: [a-lq-z](resta)
Intervalos de caracteres predefinidos	
.	Cualquier carácter (puede que no se incluyan los terminadores de línea)
\d	Un número: [0-9]
\D	Todo menos un número: [^0-9]
\s	Un espacio en blanco: [\t\n\x0B\f\r]
\S	Todo menos un espacio en blanco: [^\s]
\w	Una letra: [a-zA-Z_0-9]
\W	Todo menos letras: [^w]
Intervalos de caracteres POSIX (solo para US-ASCII)	
\p{Lower}	Letras minúsculas: [a-z]
\p {Upper}	Letras mayúsculas: [A-Z]
\p {Alpha}	Letras: [{lower}{upper}]
\p {Digit}	Número decimal: [0-9]
\p {Alnum}	Carácter alfanumérico: [{alpha}{digit}]
\p {Punct}	Signos de puntuación: uno de !#\$%&!'()*+,-./:;<=>?@[\]^_`{ }~
\p {Graph}	Los caracteres visibles: [{alnum}{punct}]
\p {Print}	Los caracteres imprimibles: [{graph}]
\p {Blank}	Un espacio o un tabulador: [\t]
\p {Cntrl}	Un carácter de control: [\x00-\x1F\x7F]
\p {XDigit}	Un número hexadecimal: [0-9a-fA-F]
\p {Space}	Un espacio: [\t\n\x0B\f\r]

Limites	
^	Comienzo de una linea
\$	Fin de una linea
\b	Fin de palabra
\B	No es fin de palabra
\A	El principio de la cadena de entrada
\G	El final del ultimo patron encajado
\Z	El final de la entrada pero el terminador final, si existe
\z	El final de la cadena de entrada
Cuantificadores de cantidad	
X?	X, cero o una ocurrencia
X*	X, cero o mas ocurrencias
X+	X, una o mas ocurrencias
X{n}	X, exactamente n veces
X{n,}	X, por lo menos n veces
X{n,m}	X, por lo menos n veces pero no mas de m veces
Operadores logicos	
XY	X seguido de Y
X Y	X o Y
(X)	X, como un grupo
Referencias hacia atrás	
\n	Lo que haya encajado el n^{esimo} grupo

Ejemplo de uso:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Regex {
    public static void main(String[] args) {
        Pattern p = Pattern.compile(args[0]);
        Matcher m = p.matcher(args[1]);
        boolean b = false;
        while (b = m.find()) {
            System.out.print(m.start() + m.group());
        }
    }
}
```

Métodos importantes:

- **Pattern.compile(regex)** carga la expresión regular enviada como parámetro.
- **Pattern.matcher(strToEval)** crea un objeto que va a operar la expresión regular contra la secuencia de caracteres a evaluar (strToEval).
- **Matcher.find()** busca una coincidencia.
- **Matcher.start()** obtiene el índice de la anterior coincidencia.
- **Matcher.group()** obtiene la secuencia de caracteres de la anterior coincidencia.
- **Matcher.end()** obtiene el índice del carácter que empieza después del último match que devuelva **true** en la comparación. Ejemplo:
“a54a5” aplicando el “d” m.start() -> 1 m.end() -> 2 en cambio si aplicando el “d*” m.start() -> 0 m.end() -> 0 , lo que pasa es que estamos diciendo que buscamos de 0 a muchas coincidencias, y en este caso es 0 coincidencia debido a que “a” no es un dígito, entonces lo toma como match pero no lo considera como dígito por lo tanto su end() siempre va a dar su mismo índice, que en este caso es 0
y el **group()** vendría hacer como un “a54a5”.substring(m.start(), m.end()), lo cual hubiera devuelto:
aplicando el “\d” m.start() -> 1 , m.end() -> 2 , m.group() -> 5
aplicando el “\d*” m.start() -> 0 , m.end() -> 0 , m.group() -> “” (vacío debido a que sería un substring(0,0) y lo substring que tienen igual inicio y final siempre devuelven vacío).

118. Ejemplos:

1)

```
public static void main(String[] args) {
    Pattern p = Pattern.compile("0[xX]([0-9a-fA-F])"); // expression
    Matcher m = p.matcher("12 0x 0x12 0xf 0xh"); // source
    boolean b = false;
    while (b = m.find()) {
        System.out.println(m.start() + " ");
    }
}
```

Cual es la salida?

6

11

2)

```
public static void main(String[] args) {
    Pattern p = Pattern.compile("proj1([,]*)"); // expression
    Matcher m = p.matcher("proj3.txt,projlsched.pdf,proj1,proj2,proj1.java"); // source
    boolean b = false;
    while (b = m.find()) {
        System.out.println(m.group());
    }
}
// Otro ejemplo: Para teléfonos 1234567, 123 4567, 123-4567
// Pattern p = Pattern.compile("\d\d\d\d\d\d\d([-]\s)?\d\d\d\d\d\d\d")
```

Cual es la salida?

projlsched.pdf
proj1
proj1.java

3)

```
public static void main(String[] args) {
    Pattern p = Pattern.compile("a.c"); // expression
    Matcher m = p.matcher("ac abc a c"); // source
    boolean b = false;
    while (b = m.find()) {
        System.out.println(m.start() + ": " + m.group());
    }
}
```

Cual es la salida?

3: abc
7: a c

119. Tokenizing con String.split()

```
public static void main(String[] args) {
    String patron = "\\\\".";
    String[] tokens = "ab.cde.fg".split(patron);
    System.out.println("count " + tokens.length);
    for (String s : tokens) {
        System.out.println(">" + s + "<");
    }
}
```

Salida:

```
count 3
>ab<
>cde<
>fg<
```

120. Tokenizing con java.util.StringTokenizer

```
public static void main(String[] args) {
    StringTokenizer stringTokenizer = new StringTokenizer("ab cde fg"); // Por defecto el delimitador es " "
    //StringTokenizer stringTokenizer = new StringTokenizer("ab.cde.fg", "."); // Delimitador es "."
    int countTokens = stringTokenizer.countTokens();
    System.out.println("tokens: " + countTokens);
    for(int i = 0; i < countTokens ; i ++ ){
        System.out.println(stringTokenizer.nextToken());
    }
}
```

Salida:

```
tokens: 3
ab
cde
fg
```

121. Tokenizing con

java.util.Scanner :

```
public static void main(String[] args) {
    boolean booleano;
    int entero;
    String s, hits = " ";
    Scanner s1 = new Scanner("1 true 34 hi");
    //s1.useDelimiter("\\."); //Se puede cambiar el delimitador.
    Scanner s2 = new Scanner("1 true 34 hi");
    while (s1.hasNext()) {
        s = s1.next();
        System.out.println(">" + s + "<");
    }
    while (s2.hasNext()) {
        if (s2.hasNextInt()) {
            entero = s2.nextInt();
            hits += " entero";
        } else if (s2.hasNextBoolean()) {
            booleano = s2.nextBoolean();
            hits += " booleano";
        } else {
            s2.next();
            hits += " cadena";
        }
    }
    System.out.println("Encontro:" + hits);
}
```

The invocation and input

```
>1<
>true<
>34<
>hi<
Encontro: entero booleano entero cadena
```

Nota: el delimitador por defecto es espacio en blanco “ ”, pero si uno quiere especificar otro, debería utilizar el método `useDelimiter("%")` de la clase Scanner

122. Dando formatos con **printf()** y **format()**:

Los métodos **format()** y **printf()**, han sido añadidos a `java.io.PrintStream` en Java 5. Estos dos métodos se comportan de la misma manera y utilizan la clase `java.util.Formatter`

ver api: <http://java.sun.com/javase/6/docs/api/java/util/Formatter.html>

Ejemplos:

```
Uso de printf(string, Object ... args);  
System.out.printf("%2$d + %1$d", 123, 456);
```

Esto produce:

```
456 + 123
```

El formato del **string** es:

```
%[arg_index$] [flags] [width][.precision]conversion char  
% indica q apartir de ahí se va aplicar el formato  
[arg_index$] indica el argumento, en el ejemplo: 2$ indica el 456  
[flags] Muchos flags pueden ser aplicables, para el examen debes saber:  
* "-" Justifica este argumento a la izquierda.  
* "+" Incluye un signo (+ o -) con este argumento  
* "0" Pad este argumento con ceros.  
* "," Usa una especificación local para la separación de grupos (i.e., la coma en 123,456)  
* "(" Encierra números negativos en paréntesis  
[width] indica el número mínimo de caracteres a imprimir  
[precision] indica el número de dígitos a imprimir después del punto decimal, solo  
se indica cuando el argumento a formatear es de coma flotante.  
conversion char indica el tipo de argumento que tu necesitas formatear:  
* "b" boolean.  
* "c" char.  
* "d" integer.  
* "f" floating point.  
* "s" string.
```

123. Notas:

- `System.out.printf("%1$b", 123);`

La expresión anterior compila y ejecuta bien. “b”(boolean) retorna true si el parámetro es diferente de null o es un tipo diferente al boolean.

- `System.out.printf("%f", 123); System.out.printf("%d", 123.45);`

Las dos expresiones anteriores lanzarían una excepción en tiempo de ejecución debido a que f(punto flotante) y d(enteros) no promueven la conversión de un entero y un punto flotante respectivamente.

124. Ejemplos de printf:

```
public static void main(String[] args) {  
    int i1 = -123;  
    int i2 = 12345;  
    System.out.printf(">%1$(7d< \n", i1);  
    System.out.printf(">%0,7d< \n", i2);  
    System.out.format(">%-+7d< \n", i2);  
    System.out.printf(">%2$b + %1$5d< \n", i1, false);  
}
```

La salida es:

```
> (123)<  
>012.345<  
>+12345 <  
>false + -123<
```

</OBJECTIVE 3.4 : Parsing, Tokenizing, and Formatting>

</CERTIFICATION_OBJETIVES>

</CAP6: Strings, I/O, Formatting, and Parsing Certification Objectives>

<CAP7: Generics and Collections>

<CERTIFICATION_OBJECTIVES>

<OBJECTIVE 6.2 : Overriding hashCode() and equals() >

<SUMMARY>

6.2 Distinguish between correct and incorrect overrides of corresponding hashCode and equals methods, and explain the difference between == and the equals method.

</SUMMARY>

125. Para el examen no necesitas conocer todos los métodos de Object, pero si conocer los presentados en la siguiente tabla:

TABLE 7-1 Methods of Class Object Covered on the Exam

Method	Description
boolean equals (Object obj)	Decides whether two objects are meaningfully equivalent.
void finalize()	Called by garbage collector when the garbage collector sees that the object cannot be referenced.
int hashCode()	Returns a hashcode int value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet.
final void notify()	Wakes up a thread that is waiting for this object's lock.
final void notifyAll()	Wakes up all threads that are waiting for this object's lock.
final void wait()	Causes the current thread to wait until another thread calls notify () or notifyAll () on this subject.
String toString()	Returns a "text representation" of the object.

boolean equals(Object obj) : Decide si dos objetos son iguales significativamente.

int hashCode() : Retorna el hashCode(un int) de un objeto, de manera que el objeto pueda ser utilizado en las clases de colección que usan hash, incluyendo Hashtable, HashMap y HashSet.

String toString () : Retorna un String que representa el objeto.

126. Sobreescribiendo **toString()**

```
class Bob {  
    int shoeSize;  
    String nickName;  
  
    public Bob(String nickName, int shoeSize) {  
        this.shoeSize = shoeSize;  
        this.nickName = nickName;  
    }  
  
    public String toString() {  
        return ("Yo soy Bob, pero puedes llamarme " + nickName + ". Yo calzo " + shoeSize);  
    }  
}
```

127. Sobreescribiendo **equals()**

En los capítulos anteriores se vió cómo comparar dos referencias a objetos usando el operador == se evalúa como **verdadera** sólo cuando ambas referencias se refieren al mismo objeto (porque == simplemente mira a los bits de la variable, y son idénticos o no lo son).

La clase **String** y las clases **Wrapper** han sobreescrito el método **equals()** (heredado de la clase **Object**), de modo que se podría comparar dos objetos diferentes (del mismo tipo) para ver si sus contenidos son significativamente equivalentes.

Cuando realmente necesitas saber si dos referencias son idénticas, usa **==**.

Pero que ocurre si quieras definir como evaluar si un objeto es igual a otro. Por ejemplo, tenemos la clase **Persona**, y queremos definir que dos objetos **Persona** con el mismo atributo dni, son iguales. Entonces tendría que sobreescribir la clase **Persona**. Ejemplo:

```
public class Persona {  
    private String nombre;  
    private String dni;  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String getDni() {  
        return dni;  
    }  
  
    public void setDni(String dni) {  
        this.dni = dni;  
    }  
  
    public boolean equals(Object o){  
        Persona p = (Persona)o;  
        if(o instanceof Persona == true && ((Persona)o).getDni().equals(this.getDni())){  
            return true;  
        }  
        return false;  
    }  
}  
  
class Test {  
    public static void main(String...a){  
        Persona p1 = new Persona();  
        p1.setDni("41236456");  
  
        Persona p2 = new Persona();  
        p2.setDni("41236456");  
  
        System.out.println(p1.equals(p2));      //Salida : true  
    }  
}
```

128. Por mantener principios de Java, cuando dos objetos son considerados como iguales, el metodo **hashCode()** de ambos debe retornar el mismo valor. En la clase **Persona** sobreescribir el método hashCode:

```
public int hashCode () {  
    return Integer.parseInt(this.dni);  
}
```

129. Los **hashCode()** son usados típicamente para incrementar la performance de muchas colecciones de datos. Colecciones como **HashMap** y **HashSet** usan el valor del hashCode para determinar como el objeto deberá ser ordenado en la colección. También ayuda a localizar el objeto en la colección.

Condition	Required	Not Required (But Allowed)
<code>x.equals(y) == true</code>	<code>x.hashCode() == y.hashCode()</code>	
<code>x.hashCode() == y.hashCode()</code>		<code>x.equals(y) == true</code>
<code>x.equals(y) == false</code>		No <code>hashCode()</code> requirements
<code>x.hashCode() != y.hashCode()</code>	<code>x.equals(y) == false</code>	

</OBJECTIVE 6.2 : Overriding hashCode() and equals() >

<OBJECTIVE 6.1 : Collections >

<SUMMARY>

6.1 Given a design scenario, determine which collection classes and/or interfaces should be used to properly implement that design, including the use of the Comparable interface.

</SUMMARY>

130. Interfaces para manejo de Colecciones.

Las interfaces que debes conocer para el examen y para tu vida javística en general son 9:

Collection	Set	SortedSet
List	Map	SortedMap
Queue	NavigableSet	NavigableMap

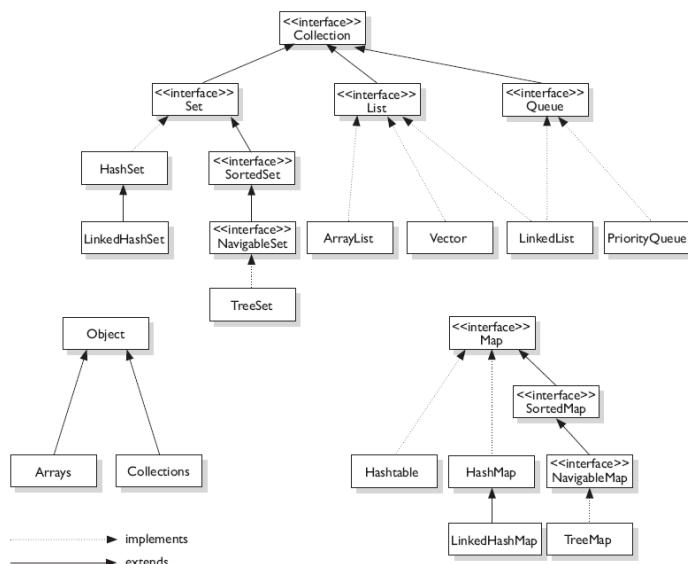
NavigableSet y **NavigableMap** son nuevas de java 6.

Las clases que implementan estas interfaces y que tu necesitas conocer para el examen son las 13 presentadas en la siguiente tabla.

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

Pero existen más.

131. Interfaces y jerarquía de clases para colecciones:



exam

Watch
You can so easily mistake "Collections" for "Collection"—be careful.
Keep in mind that **Collections** is a class, with static utility methods, while **Collection** is an interface with declarations of the methods common to most collections including `add()`, `remove()`, `contains()`, `size()`, and `iterator()`.

List: Lista de objetos (clases que implementan List).

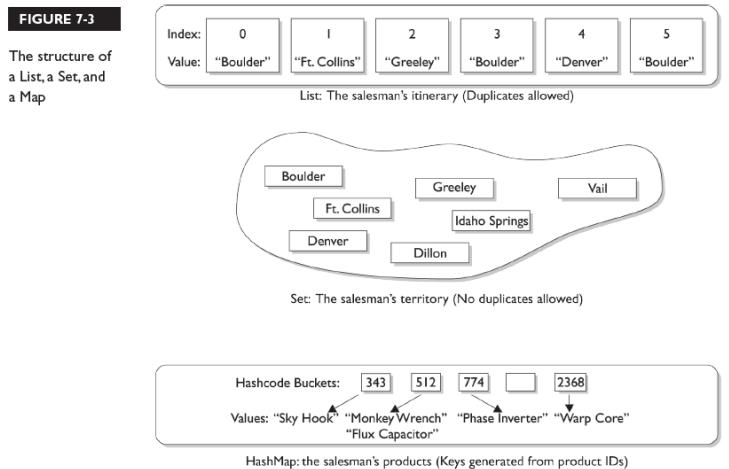
Set: Únicos objetos (clases que implementan Set).

Maps: Objetos con un único ID (clases que implementan Map).

Queues: Objetos organizados por el orden en que se van a procesar

132. Ilustración de cómo es un Set, List y un Map

Figure 7-3 illustrates the structure of a List, a Set, and a Map.



But there are sub-flavors within those four flavors of collections:



133. Antes de entender bien sobre **Sorted**, **Unsorted**, **Ordered**, **Unordered**, primero tenemos que discutir la idea de iteración. Cuando uno piensa en iteración, se puede pensar en recorrer una matriz usando, por ejemplo, un bucle de acceso a cada elemento en la matriz en orden ([0], [1], [2], y así sucesivamente). Iterando a través de una colección a menudo significa ir a través de los elementos, uno tras otro, comenzando por el primer elemento.

Sin embargo, a veces hasta el concepto de primer es un poco extraño, ya que en un **Hashtable** no existe el concepto de **primero**, **segundo**, **tercero**, y así sucesivamente(**unordered**). En un **Hashtable**, los elementos se colocan en un orden basado en la hashcode de la key. Pero algo tiene que ir primero al iterar, por lo que cuando iteras sobre un **Hashtable** es incierto cual será el primero, ya que puede cambiar de manera aparentemente aleatoria, ya que no tiene una regla(s) para conocer como se va a ordenar (**unsorted**).

Ejemplo de cómo recorrer un hash:

```
public static void main(String[] args) {
    Hashtable h = new Hashtable();
    h.put("1", "hugo");//No existe indice
    h.put("2", "paco");
    h.put("3", "luis");

    Set keySet = h.keySet();
    Iterator i = keySet.iterator();
    String key = null;
    while(i.hasNext() == true && (key = (String)i.next()) != null) {
        System.out.println(h.get(key));
    }
}
```

134. Cortitas:

- **Ordered(Ordenada):** Significa que cuando una colección es **ordered**, tu puedes recorrer la colección en un específico(no random) orden. Un **HashTable** no es ordered. Aunque el Hashtable tiene su propia lógica interna para determinar el orden(basado en hashcodes y la aplicación de la misma colección). Un **ArrayList** sin embargo mantiene el orden establecido por los índices de los elementos (como un array, entonces es **ordered**).
 - Hay algunas colecciones que mantienen un **orden natural**, y las colecciones son entonces no sólo son **ordered** sino también **sorted**. Como **orden natural** se refiere a por ejemplo si almacena strings, los almacena por orden alfabético A...,B...,C... si almacena enteros, los almacena de menor a mayor 1, 2, 3... etc.

```
ArrayList arrayList = new ArrayList();
arrayList.add("hugo"); //0
arrayList.add("paco"); //1
arrayList.add("luis"); //2

for (Object object : arrayList) {
    System.out.println(object.toString());
}
```

- **Sorted:** Significa que el orden en la colección es determinada de acuerdo a alguna regla o reglas para conocer como se va a ordenar. Sorted es basado en las propiedades de los mismos objetos. Las colecciones que se basan en un orden **natural** son consideradas **sorted**. Pero si quieras hacer que una colección sea **Sorted** y no de orden **natural**, entonces tus clases de tu colección deben hacer uso de las interfaces **Comparable** o **Comparator**.

El siguiente código ordena una colección de Personas, usando la interface Comparable:

```
//Para poder ordenar Personas debe implementar Comparable
public class Persona implements Comparable{
    private String nombre;
    private String dni;

    public Persona(String dni, String nombre) {
        this.nombre = nombre;
        this.dni = dni;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }

    //Retorna:
    //Un número negativo si el objeto(this) es menor.
    //0, Si los objetos son iguales.
    //Un número positivo si el objeto(this) es mayor.
    //NOTA: Collections, siempre ordena de menor a mayor
    public int compareTo(Object o) {
        return this.dni.compareTo(((Persona)o).getDni());
    }
}

//Ordena una lista de Personas
public static void main(String[] args) {
    Persona p1 = new Persona("41236452", "Hugo");
    Persona p2 = new Persona("41236454", "Paco");
    Persona p3 = new Persona("41236453", "Luis");
    Persona p4 = new Persona("41236451", "Luisa");

    ArrayList<Persona> list = new ArrayList<Persona>();
    list.add(p1);
    list.add(p2);
    list.add(p3);
    list.add(p4);

    //Lista sin ordenar
    System.out.println("Lista sin ordenar:");
    for (Persona persona : list) {
        System.out.print(persona.getDni() + ", ");
        System.out.println(persona.getNombre());
    }

    //Lista ordenada
    System.out.println("Lista ordenada:");
    Collections.sort(list);
    for (Persona persona : list) {
        System.out.print(persona.getDni() + ", ");
        System.out.println(persona.getNombre());
    }
}
```

Colecciones más comunes:

- **ArrayList:** Es de rápida iteración y rápido acceso aleatorio. Es una colección ordenada (por índice), pero no sorted (clasificada). A partir de la versión 1.4 implementa la nueva interface **RandomAccess**. Utiliza esta clase en vez de un **LinkedList** cuando quieras una iteración rápida y no hacer muchas operaciones de inserción o borrado.

Ejemplo de uso:

```
class Dog {  
    public String name;  
  
    Dog(String n) {  
        name = n;  
    }  
  
}  
  
class ItTest {  
    public static void main(String[] args) {  
        List<Dog> d = new ArrayList<Dog>();  
        Dog dog = new Dog("aiko");  
        d.add(dog);  
        d.add(new Dog("clover"));  
        d.add(new Dog("magnolia"));  
        Iterator<Dog> i3 = d.iterator(); // make an iterator  
        while (i3.hasNext()) {  
            Dog d2 = i3.next(); // cast not required  
            System.out.println(d2.name);  
        }  
        System.out.println("size " + d.size());  
        System.out.println("get1 " + d.get(1).name);  
        System.out.println("aiko " + d.indexOf(dog));  
  
        d.remove(2);  
        Object[] oa = d.toArray();  
        for (Object o : oa) {  
            Dog d2 = (Dog) o;  
            System.out.println("oa " + d2.name);  
        }  
    }  
}
```

- **Vector:** Existe desde los primeros días de Java, **Vector** y **Hashtable** son las dos colecciones originales, el resto fue añadido con Java 2 versión 1.2 y 1.4. Un vector es básicamente lo mismo que un **ArrayList**, pero los métodos del **Vector** son sincronizados por **thread safety**. Normalmente vas a preferir un **ArrayList** en vez de un **Vector** porque los métodos sincronizados añaden un rendimiento que tal vez no necesitarás. Y si necesitas thread safety, hay métodos de utilitarios en la clase **Collections** que pueden ayudar. Vector es la única clase que como **ArrayList** implementa **RandomAccess**.
- **LinkedList:** Es ordenado por índice, como el **ArrayList**, excepto que los elementos están doblemente vinculados el uno al otro, esto le da una mejor funcionalidad cuando se quiere añadir y eliminar desde el inicio o el final, lo que le convierte en una elección fácil para la elaboración de una pila o cola. Ten en cuenta que itera más lentamente que un **ArrayList**. Pero es una buena opción cuando se quiere rapidez de inserción o eliminación. A partir de java 5, se ha mejorado con la nueva

implementación de la interface **java.util.Queue**. Ya que tal, que ahora soporta métodos definidos por queue tales como : peek(), poll(), y offer().

- **Set Interface:** Tiene una singularidad – no permite duplicados. Tu buen amigo el método **equals()** determina si dos objetos son iguales(en cada caso solo uno puede estar en el Set).

Implementaciones de la interface Set:

- **HashSet:** Es unsorted y unordered. Usa el hashCode y equals() para evaluar a la hora de insertar el objeto. Use esta clase cuando no quiere una colección con duplicados ni tampoco quiere iterar sobre esta.
- **LinkedHashSet:** Es una versión ordered del **HashSet** que mantiene un doble enlace entre todos los elementos. Use esta clase en lugar de **HashSet** cuando vas aplicar interacciones. Cuando tu iteras con un **HashSet** el orden es impredecible, mientras un **LinkedHashSet** permite iterar los elementos en el orden que fueron insertados.

e x a m

Watch

When using HashSet or LinkedHashSet, the objects you add to them must override hashCode(). If they don't override hashCode(), the default hashCode() method will allow multiple objects that you might consider "meaningfully equal" to be added to your "no duplicates allowed" set.

- **TreeSet:** es una de las dos sorted collections (la otra es **TreeMap**). Utiliza una estructura **arbol rojo y negro**, y se tiene la garantía que los elementos estarán ordenados en orden ascendente, según el orden natural. Opcionalmente puede construir un TreeSet con un constructor que le permite dar a la colección sus propias reglas sobre el como se van a ordenar los elementos, utilizando **Comparable** or **Comparator**. A partir de java 6 **TreeSet** implementa la interface **NavigableSet**.

- **Map Interface**

Un Map se preocupa por los identificadores únicos. Se mapea un clave única (ID) a un valor, donde tanto la clave y el valor son, por supuesto, los objetos. Te permite hacer cosas como buscar un valor basado en la clave. Al igual que el **HashSet** se basa en el método **equals()** para determinar si dos claves son las mismas o diferentes.

Implementaciones de la interface Map:

- **HashMap:** Es unsorted, unordered Map. Cuando quieres un Map y no importa el orden (al iterar a través de él), entonces **HashMap** es la mejor elección. En caso de que el key sea un objeto, se basa en el **hashcode** y el **equals()**, para evaluar y no permitir claves duplicadas pero ojo: si se quiere insertar una clave que ya existe actualiza el valor de la clave existente con el que se quiere insertar.

Permite una clave nula y múltiples valores nulos en la colección.

Ejemplo de uso:

```

class Dog {
    public Dog(String n) {
        name = n;
    }
    public String name;
    public boolean equals(Object o) {
        if ((o instanceof Dog) && (((Dog) o).name == name)) {
            return true;
        } else {
            return false;
        }
    }
    public int hashCode() {
        return name.length();
    }
}

class Cat {}

enum Pets {
    DOG, CAT, HORSE
}

class MapTest {
    public static void main(String[] args) {
        Map<Object, Object> m = new HashMap<Object, Object>();
        m.put("k1", new Dog("aiko")); // add some key/value pairs
        m.put("k2", Pets.DOG);
        m.put(Pets.CAT, "CAT key");
        Dog dl = new Dog("clover"); // let's keep this reference
        m.put(dl, "Dog key");
        m.put(new Cat(), "Cat key");
        System.out.println(m.get("k1")); // #1
        String k2 = "k2";
        System.out.println(m.get(k2)); // #2
        Pets p = Pets.CAT;
        System.out.println(m.get(p)); // #3
        System.out.println(m.get(dl)); // #4
        System.out.println(m.get(new Cat())); // #5
        System.out.println(m.size()); // #6
    }
}

```

- **Hashtable:** Como **Vector**, Hashtable ha existido desde la prehistoria de **Java**. Al igual que **Vector**, también sus principales métodos son sincronizados(synchronized). Así como el **ArrayList** es la versión mas moderna del **Vector**, el **HashMap** es la versión más moderna del **Hashtable**, solo que el **HashMap** no tiene métodos sincronizados y otra cosa mas:: **HashMap** acepta **null** como key o **null** como valor, en cambio **Hashtable** no acepta nada que sea **null**.
- **LinkedHashMap:** Al igual que su colega **LinkedHashSet**, el **LinkedHashMap** mantiene un orden de inserción (o opcionalmente, para el acceso). A pesar de que será algo mas lento que el **HashMap** para añadir o remover elementos, es más rápido la iteración con **LinkedHahsMap**

Algunos ejemplos:

<http://www.ibm.com/developerworks/java/library/j-mer0821/>

- **TreeMap:** Es un **Map** sorted, y esto significa que por defecto es “ordenada por el orden natural”. Al igual que **TreeSet**, **TreeMap** permite definir un orden personalizado (A través de un **Comparator** o un **Comparable**). Cuando tu construyes un **TreeMap**, que especifica como los elementos deben compararse entre sí cuando están siendo ordenados.

- **Queue Interface:**

Es una cola, aunque son posibles otros órdenes, las colas son pensadas como FIFO (primero en entrar, primero en salir). Las colas soportan todo los métodos estandar de Collection y también añade métodos para agregar o quitar elmentos de la cola de revisión.

Implementaciones de la interface Queue

- **PriorityQueue:** Esta clase es nueva con Java 5. Desde que a la clase **LinkedList** se le hizo implementar la interface **Queue**, las colas pueden ser atendidas con un **LinkedList**. El propósito de un **PriorityQueue** es crear una “prioridad de entrada, prioridad de salida” frente a una típica cola FIFO. Los elementos de un **PriorityQueue** están ordenados ya sea por orden **natural**(en cuyo caso los elementos que se ordenan primero se acceden en primer lugar) o de acuerdo con un **Comparator**. En cualquier caso, los elementos ordenados representan su prioridad relativa.

Ejemplo de uso:

```
class PQ {
    static class PQsort implements Comparator<Integer> { // inverse sort
        public int compare(Integer one, Integer two) {
            return two - one; // unboxing
        }
    }

    public static void main(String[] args) {
        int[] ia = { 1, 5, 3, 7, 6, 9, 8 }; // unordered data
        PriorityQueue<Integer> pq1 = new PriorityQueue<Integer>(); // using natural order
        for (int x : ia){
            // load queue
            pq1.offer(x);
        }
        for (int x : ia){
            // review queue
            System.out.print(pq1.poll() + " ");
        }
        System.out.println("");
        PQsort pqs = new PQsort(); // get a Comparator
        PriorityQueue<Integer> pq2 = new PriorityQueue<Integer>(10, pqs); // using Comparator
        for (int x : ia){
            // load queue
            pq2.offer(x);
        }
        System.out.println("size " + pq2.size());
        System.out.println("peek " + pq2.peek());
        System.out.println("size " + pq2.size());
        System.out.println("poll " + pq2.poll());
        System.out.println("size " + pq2.size());
        for (int x : ia){
            // review queue
            System.out.print(pq2.poll() + " ");
        }
    }
}
```

e x a m

Watch

You can easily eliminate some answers right away if you recognize that, for example, a **Map** can't be the class to choose when you need a name/value pair collection, since **Map** is an interface and not a concrete implementation class. The wording on the exam is explicit when it matters, so if you're asked to choose an interface, choose an interface rather than a class that implements that interface. The reverse is also true—if you're asked to choose a class, don't choose an interface type.

- La siguiente tabla sumariza 11 de las 13 concretas clases orientadas a la colección y que tu necesitas saber para el examen:

TABLE 7-2 Collection Interface Concrete Implementation Classes

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
HashTable	x			No	No
TreeMap	x			Sorted	By <i>natural order</i> or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By <i>natural order</i> or custom comparison rules
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	By to-do order

</OBJECTIVE 6.1 : Collections >

<OBJECTIVE 6.5 : Using the Collections Framework>
<SUMMARY>

6.3 Write code that uses the NavigableSet and NavigableMap interfaces.

6.5 Use capabilities in the java.util package to write code to manipulate a list by sorting, performing a binary search, or converting the list to an array. Use capabilities in the java.util package to write code to manipulate an array by sorting, performing a binary search, or converting the array to a list. Use the java.util.Comparator and java.lang.Comparable interfaces to affect the sorting of lists and arrays. Furthermore, recognize the effect of the "natural ordering" of primitive wrapper classes and java.lang.String on sorting.

</SUMMARY>

135. Ya se vió la parte teórica ahora veremos como se trabaja con ellos.

- **ArrayList:**

En la práctica típicamente vas a querer instanciar un **ArrayList** polimórficamente:

```
List myList = new ArrayList();
```

Como hablamos de java 5y 6,tu querras decir:

```
List<String> myList = new ArrayList<String>();
```

Ejemplo de uso

```
import java.util.*;  
  
class TestArrayList {  
    public static void main(String[] args) {  
        List<String> test = new ArrayList<String>();  
        String s = "hi";  
        test.add("string");  
        test.add(s);  
        test.add(s + s);  
        System.out.println(test.size());  
        System.out.println(test.contains(42));  
        System.out.println(test.contains("hihi"));  
        test.remove("hi");  
        System.out.println(test.size());  
    }  
}
```

which produces:

```
3  
false  
true  
2
```

- **Sorting Collections:**

Empezamos con algo simple, un ordenamiento de una colección de Strings:

```
import java.util.*;

class TestSort1 {
    public static void main(String[] args) {
        ArrayList<String> stuff = new ArrayList<String>();
        stuff.add("Denver");
        stuff.add("Boulder");
        stuff.add("Vail");
        stuff.add("Aspen");
        stuff.add("Telluride");
        System.out.println("unsorted " + stuff);
        Collections.sort(stuff); // #2
        System.out.println("sorted " + stuff);
    }
}
```

This produces something like this:

```
unsorted [Denver, Boulder, Vail, Aspen, Telluride]
sorted [Aspen, Boulder, Denver, Telluride, Vail]
```

El problema está cuando queremos ordenar una colección de objetos como:

```
ArrayList<DVDInfo> dvdList = new ArrayList<DVDInfo>();
DVDInfo info1 = new DVDInfo();
DVDInfo info2 = new DVDInfo();
dvdList.add(info1);
dvdList.add(info2);
Collections.sort(dvdList);
```

El resultado de ejecutar el código anterior es:

```
TestDVD.java:13: cannot find symbol
symbol : method sort(java.util.ArrayList<DVDInfo>)
location: class java.util.Collections
    Collections.sort(dvdlist);
```

excepción, porqué? Y porque no sucedió con los Strings???, lo que pasa es que los String implementan la interface Comparable. Entonces ahí está lo que hay que hacer...::: hacer que la clase **DVDInfo** implemente la interface **Comparable**.

- La interface **Comparable**:

Esta interface es usada por el método **Collections.sort()** y **java.util.Array.sort()**.

La clase que implemente esta interface debe implementar un único método: **compareTo()** :

```
int x = thisObject.compareTo(anotherObject);
```

El método **compareTo()** retorna:

Negativo si `thisObject < anotherObject`

Cero si `thisObject == anotherObject`

Positivo si `thisObject > anotherObject`

Aplicando a nuestra clase DVDInfo:

```
class DVDInfo implements Comparable<DVDInfo> { // #1
    private String title;
    private String genre;

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getGenre() { return genre; }
    public void setGenre(String genre) { this.genre = genre; }

    public int compareTo(DVDInfo d) {
        return this.title.compareTo(d.getTitle());
    }
}
```

Otra forma:

```
class DVDInfo implements Comparable {
    private String title;
    private String genre;

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getGenre() { return genre; }
    public void setGenre(String genre) { this.genre = genre; }

    public int compareTo(Object o) {
        // than a specific type
        DVDInfo d = (DVDInfo)o;
        return title.compareTo(d.getTitle());
    }
}
```

La primera forma es la ideal (a partir de java 5 – Uso de Generics).

- Sorting with Comparator

La interface **Comparator** tiene la capacidad de ordenar la colección de maneras diferentes. En este caso, la clase DVDInfo no va a implementar **Comparator**. Ahora la clase llamada **GenreSort** se encargará de gestionar las comparaciones entre los objetos, a la hora de ordenar, veamos como trabaja **Comparator**:

```
class GenreSort implements Comparator<DVDInfo> {
    public int compare(DVDInfo one, DVDInfo two) {
        return one.getGenre().compareTo(two.getGenre());
    }
}
```

Haciendo uso de esta clase, el código quedaría:

```
import java.util.*;
import java.io.*; // populateList() needs this

class TestDVD {
    ArrayList<DVDInfo> dvdlist = new ArrayList<DVDInfo>();

    public static void main(String[] args) {
        new TestDVD().go();
    }

    public void go() {
        populateList();
        System.out.println(dvdlist); // output as read from file
        Collections.sort(dvdlist);
        System.out.println(dvdlist); // output sorted by title
        GenreSort gs = new GenreSort();
        Collections.sort(dvdlist, gs);
        System.out.println(dvdlist); // output sorted by genre
    }

    public void populateList() {
        // read the file, create DVDInfo instances, and
        // populate the ArrayList dvdlist with these instances
    }
}
```

Diferencias entre las interfaces Comparable y Comparator:

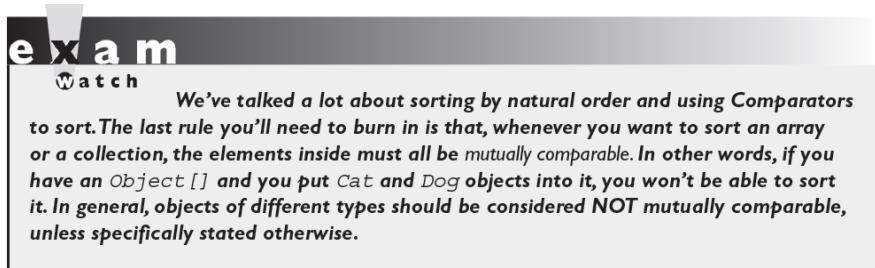
TABLE 7-3 Comparing Comparable to Comparator

java.lang.Comparable	java.util.Comparator
int objOne.compareTo(objTwo)	int compare(objOne, objTwo)
Returns negative if objOne < objTwo zero if objOne == objTwo positive if objOne > objTwo	Same as Comparable
You must modify the class whose instances you want to sort.	You build a class separate from the class whose instances you want to sort.
Only one sort sequence can be created	Many sort sequences can be created
Implemented frequently in the API by: String, Wrapper classes, Date, Calendar...	Meant to be implemented to sort instances of third-party classes.

- Sorting with the Arrays Class

De la misma manera que sucede con Collections:

```
* Arrays.sort(arrayToSort)  
* Arrays.sort(arrayToSort, Comparator)
```



- Searching Arrays and Collections

Collections y Arrays tienen métodos que te permitirán buscar un específico elemento. Al buscar a través de colecciones o conjuntos, la siguientes reglas se aplican:

- Las búsquedas se hacen con el método **binarySearch()**.
- Una búsqueda satisfactoria retorna un entero que representa el índice del elemento encontrado.
- Una búsqueda no satisfactoria retorna un entero que representa el punto de inserción (**insertion point**). El punto de inserción es el lugar donde el elemento sería insertado para que la colección se mantenga ordenada. El valor returnedo es negativo y está representado como: **(-(insertion point) -1)**. Esto con el fin de que no retorne positivo ya que si fuera así indicaría que la búsqueda fue existosa.
- Antes de empezar la búsqueda, la **Colección** debe estar sorteada, sino no va a buscar mal. Si se aplica un **Comparator** este debe ser pasado como parámetro, pero si el orden de la colección va a ser por orden natural, entonces no es necesario enviar el **Comparator**:

```
binarySearch(list, key, comparator)  
binarySearch(list, key)
```

Ejemplo de uso:

```
package session07;

import java.util.Arrays;
import java.util.Comparator;

class SearchObjArray {
    public static void main(String[] args) {
        String[] sa = { "one", "two", "three", "four" };
        Arrays.sort(sa); // #1
        for (String s : sa)
            System.out.print(s + " ");
        System.out.println("\none = " + Arrays.binarySearch(sa, "one")); // #2
        System.out.println("now reverse sort");
        ReSortComparator rs = new ReSortComparator(); // #3
        Arrays.sort(sa, rs);
        for (String s : sa)
            System.out.print(s + " ");
        System.out.println("\none = " + Arrays.binarySearch(sa, "one")); // #4
        System.out.println("one = " + Arrays.binarySearch(sa, "one", rs)); // #5
    }

    static class ReSortComparator implements Comparator<String> { // #6
        public int compare(String a, String b) {
            return b.compareTo(a); // #7
        }
    }
}

/*
which produces something like this:
four one three two
one = 1
now reverse sort
two three one four
one = -1
one = 2
*/

```

- Convirtiendo Arrays a List de Arrays

Las clases de **List** y **Set** tienen el método **toArray()**, y los **Arrays** tienen un método llamado **asList()**.

El método **Arrays.asList()** copia un array dentro de un **List**. Cuando uno actualiza un **Array** que ha sido pasado a **Lista**, automáticamente cambia en la **Lista** y viceversa. Ejemplo:

```
String[] sa = { "one", "two", "three", "four" };
List sList = Arrays.asList(sa); // make a List
System.out.println("size " + sList.size());
System.out.println("idx2 " + sList.get(2));
sList.set(3, "six"); // change List
sa[1] = "five"; // change array
for (String s : sa)
    System.out.print(s + " ");
System.out.println("\nsl[1] " + sList.get(1));
```

This produces

```
size 4
idx2 three
one five three six
sl[1] five
```

Ahora hechemos un vistazo al método **toArray()**, éste metodo puede ser usado de dos maneras, una que devuelve un nuevo objeto array y otra que pasa las referencias al array destino:

```
List<Integer> iL = new ArrayList<Integer>();
for (int x = 0; x < 3; x++)
    iL.add(x);
Object[] oa = iL.toArray(); // create an Object array
Integer[] ia2 = new Integer[3];
ia2 = iL.toArray(ia2); // create an Integer array
```

- **Usando Listas**

La manera más común de examinar los elementos de un **List** es con el uso de un **Iterator**. Un Iterator es un objeto que está asociado con una colección específica. Te permite recorrer la colección paso a paso. Hay dos métodos de **Iterator** que tu necesitas entender para el examen, estos son:

- boolean **hasNext()** Retorna true si hay por lo menos hay un elemento más que se pueda recorrer de la colección.
- object **next()** Este método retorna el siguiente objeto en la colección y se mueve hacia delante después que el elemento ha sido retornado.

Ejemplo de uso:

```
package sesion07;
import java.util.*;

class Dog {
    public String name;
    Dog(String n) {
        name = n;
    }
}

class ItTest {
    public static void main(String[] args) {
        List<Dog> d = new ArrayList<Dog>();
        Dog dog = new Dog("aiko");
        d.add(dog);
        d.add(new Dog("clover"));
        d.add(new Dog("magnolia"));
        Iterator<Dog> i3 = d.iterator(); // make an iterator
        while (i3.hasNext()) {
            Dog d2 = i3.next(); // cast not required
            System.out.println(d2.name);
        }
        System.out.println("size " + d.size());
        System.out.println("get1 " + d.get(1).name);
        System.out.println("aiko " + d.indexOf(dog));
        d.remove(2);
        Object[] oa = d.toArray();
        for (Object o : oa) {
            Dog d2 = (Dog) o;
            System.out.println("oa " + d2.name);
        }
    }
}
/* This produces
 * aiko
 * clover
 * magnolia
 * size 3
 * get1 clover
 * aiko 0
 * oa aiko
 * oa clover */
```

- **Using Sets**

Recuerda que **Sets** son usados cuando no quieres ningún duplicado en la colección.

Si tu quieres añadir algun elemento que ya existe en el **Set**, el método **add()** te devuelve **false**.

Mucho cuidado cuando usas un **TreeSet**, acá el porqué:

```
import java.util.*;  
  
class SetTest {  
    public static void main(String[] args) {  
        boolean[] ba = new boolean[5];  
        // (1)insert code here  
        ba[0] = s.add("a");  
        ba[1] = s.add(new Integer(42));  
        ba[2] = s.add("b");  
        ba[3] = s.add("a");  
        ba[4] = s.add(new Object());  
        for (int x = 0; x < ba.length; x++)  
            System.out.print(ba[x] + " ");  
        System.out.println("\n");  
        for (Object o : s)  
            System.out.print(o + " ");  
    }  
}
```

Si tu insertaras la siguiente línea de código

```
Set s = new HashSet(); // (1)insert code here
```

La salida sería :

```
true true true false true  
a java.lang.Object@e09713 42 b
```

Es importante saber que el orden de los objetos impresos en el segundo for, no es predecible. **HashSets** no garantizan un orden.

Ahora si cambiamos el código por este otro:

```
Set s = new TreeSet(); // (1)insert code here
```

La salida sería:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String  
at java.lang.Integer.compareTo(Integer.java:35)  
at java.util.TreeMap.compare(TreeMap.java:1093)  
at java.util.TreeMap.put(TreeMap.java:465)  
at java.util.TreeSet.add(TreeSet.java:210)
```

Esto es debido a que el set está añadiendo objetos de distintos tipos. Lo que ocurre es lo siguiente:

Siempre que se desee tener una colección ordenada, sus elementos deben ser mútuamente comparables. Recuerde que, a menos que se especifique lo contrario, los objetos de diferentes tipos no son comparables, pero si son del mismo tipo esta clase debe implementar **Comparable**.

- **Navegando TreeSet y TreeMap**

En java 6 ha introducido 2 nuevas interfaces:

`java.util.NavigableSet` y `java.util.NavigableMap`. Y para propósitos de navegación las clases `TreeSet` y `TreeMap` las implementa.

Imagina que el tren Limeñito tiene un programa de salidas basadas en horas militares.

Se quiere saber:

1. El último tren que sale antes de las 4 (1600 hours)
2. El primer tren que sale después de las 8 (2000 hours)

```
class Ferry {
    public static void main(String[] args) {
        TreeSet<Integer> times = new TreeSet<Integer>();
        times.add(1205); // add some departure times
        times.add(1505);
        times.add(1545);
        times.add(1830);
        times.add(2010);
        times.add(2100);

        // Java 5 version
        TreeSet<Integer> subset = new TreeSet<Integer>();
        subset = (TreeSet) times.headSet(1600);
        System.out.println("J5 - last before 4pm is: " + subset.last());
        TreeSet<Integer> sub2 = new TreeSet<Integer>();
        sub2 = (TreeSet) times.tailSet(2000);
        System.out.println("J5 - first after 8pm is: " + sub2.first());

        // Java 6 version using the new lower() and higher() methods
        System.out.println("J6 - last before 4pm is: " + times.lower(1600));
        System.out.println("J6 - first after 8pm is: " + times.higher(2000));
    }
}
```

El programa produce una salida de:

```
J5 - last before 4pm is: 1545
J5 - first after 8pm is: 2010
J6 - last before 4pm is: 1545
J6 - first after 8pm is: 2010
```

Importante:

Otros métodos relacionados a la navegación

TABLE 7-4 Important "Navigation" Related Methods

Method	Description
<code>TreeSet.ceiling(e)</code>	Returns the lowest element $\geq e$
<code>TreeMap.ceilingKey(key)</code>	Returns the lowest key $\geq \text{key}$
<code>TreeSet.higher(e)</code>	Returns the lowest element $> e$
<code>TreeMap.higherKey(key)</code>	Returns the lowest key $> \text{key}$
<code>TreeSet.floor(e)</code>	Returns the highest element $\leq e$
<code>TreeMap.floorKey(key)</code>	Returns the highest key $\leq \text{key}$
<code>TreeSet.lower(e)</code>	Returns the highest element $< e$
<code>TreeMap.lowerKey(key)</code>	Returns the highest key $< \text{key}$
<code>TreeSet.pollFirst()</code>	Returns and removes the first entry
<code>TreeMap.pollFirstEntry()</code>	Returns and removes the first key-value pair
<code>TreeSet.pollLast()</code>	Returns and removes the last entry
<code>TreeMap.pollLastEntry()</code>	Returns and removes the last key-value pair
<code>TreeSet.descendingSet()</code>	Returns a NavigableSet in reverse order
<code>TreeMap.descendingMap()</code>	Returns a NavigableMap in reverse order

- **Using Maps**

Recuerda que cuando usas las clases que implementan Map, cualquier clase que uses como llave(key) debe sobreescribir los métodos hashCode() y equals().

Ejemplo de cómo se usa un HashMap:

```

class Dog {
    public Dog(String n) {
        name = n;
    }

    public String name;

    public boolean equals(Object o) {
        if ((o instanceof Dog) && (((Dog) o).name == name)) {
            return true;
        } else {
            return false;
        }
    }

    public int hashCode() {
        return name.length();
    }
}

class Cat {

enum Pets {
    DOG, CAT, HORSE
}

class MapTest {
    public static void main(String[] args) {
        Map<Object, Object> m = new HashMap<Object, Object>();
        m.put("k1", new Dog("aiko")); // add some key/value pairs
        m.put("k2", Pets.DOG);
        m.put(Pets.CAT, "CAT key");
        Dog d1 = new Dog("clover"); // let's keep this reference
        m.put(d1, "Dog key");
        m.put(new Cat(), "Cat key");
        System.out.println(m.get("k1")); // #1
        String k2 = "k2";
        System.out.println(m.get(k2)); // #2
        Pets p = Pets.CAT;
        System.out.println(m.get(p)); // #3
        System.out.println(m.get(d1)); // #4
        System.out.println(m.get(new Cat())); // #5
        System.out.println(m.size()); // #6
    }
}

```

which produces something like this:

```

Dog@1c
DOG
CAT key
Dog key
null
5

```

Recordar que ::::: cuando queremos poner instancias de nuestros propios objetos como keys, la validación que hace el map es sobre si ese objeto ya existe: **utiliza su equals() y su hashCode()**, ya q por regla en Java si dos objetos son iguales, éstos deben tener el mismo **hashCode**.

- **Usando la clase PriorityQueue**

La última colección que tu necesitas entender para el examen es la clase PriorityQueue.

A diferencia de las estructuras que son primero en entrar, primero en salir, un PriorityQueue ordena sus elementos usando una definición de prioridad. La prioridad puede ser tan simple como un orden natural (en la que, por ejemplo, una entrada de 1 sería una prioridad más alta que un ingreso de 2). Además un Priority Queue puede ser ordenado mediante un comparador, que le permite definir cualquier orden que desee. Las colas tienen una serie de métodos que no se encuentran en otras interfaces de colecciones, tales como: **peek()**, **poll()**, y **offer()**.

Ejemplo de Uso:

```
class PQ {
    static class PQsort implements Comparator<Integer> { // inverse sort
        public int compare(Integer one, Integer two) {
            return two - one; // unboxing
        }
    }

    public static void main(String[] args) {
        int[] ia = { 1, 5, 3, 7, 6, 9, 8 }; // unordered data
        PriorityQueue<Integer> pq1 = new PriorityQueue<Integer>(); // use
        // natural
        // order
        for (int x : ia) // load queue
            pq1.offer(x);
        for (int x : ia) // review queue
            System.out.print(pq1.poll() + " ");
        System.out.println("");
        PQsort pqs = new PQsort(); // get a Comparator
        PriorityQueue<Integer> pq2 = new PriorityQueue<Integer>(10, pqs); // use Comparator
        for (int x : ia) // load queue
            pq2.offer(x);
        System.out.println("size " + pq2.size());
        System.out.println("peek " + pq2.peek());
        System.out.println("size " + pq2.size());
        System.out.println("poll " + pq2.poll());
        System.out.println("size " + pq2.size());
        for (int x : ia) // review queue
            System.out.print(pq2.poll() + " ");
    }
}
```

This code produces something like this:

```
1 3 5 6 7 8 9
size 7
peek 9
size 7
poll 9
size 6
8 7 6 5 3 1 null
```

Para el examen, los métodos de PriorityQueue son importantes que los entiendas:

offer() que es igual a **add()**.

peek() que recupera el elemento de la cabecera de la cola y no lo borra.

poll() que recupera el elemento de la cabecera de la cola y lo borra.

exam

watch

It's important to know some of the details of natural ordering. The following code will help you understand the relative positions of uppercase characters, lowercase characters, and spaces in a natural ordering:

```
String[] sa = {">ff<", "> f<", ">f <", ">FF<"}; // ordered?
PriorityQueue<String> pq3 = new PriorityQueue<String>();
for(String s : sa)
    pq3.offer(s);
for(String s : sa)
    System.out.print(pq3.poll() + " ");
```

This produces:

```
> f< >FF< >f < >ff<
```

If you remember that spaces sort before characters and that uppercase letters sort before lowercase characters, you should be good to go for the exam.

Recordar que los espacios en blanco tienen mayor prioridad que las letras en mayúscula y que las mayúsculas tienen mas prioridad que las minúscula.

- **Métodos en Arrays y Collections.** “T[]” Indica cualquier array que no sea un array de primitivos.

TABLE 7-4 Key Methods in Arrays and Collections

Key Methods in <code>java.util.Arrays</code>	Descriptions
<code>static List asList(T[])</code>	Convert an array to a List, (and bind them).
<code>static int binarySearch(Object[], key)</code> <code>static int binarySearch(primitive[], key)</code>	Search a sorted array for a given value, return an index or insertion point.
<code>static int binarySearch(T[], key, Comparator)</code>	Search a Comparator-sorted array for a value.
<code>static boolean equals(Object[], Object[])</code> <code>static boolean equals(primitive[], primitive[])</code>	Compare two arrays to determine if their contents are equal.
<code>public static void sort(Object[])</code> <code>public static void sort(primitive[])</code>	Sort the elements of an array by natural order.
<code>public static void sort(T[], Comparator)</code>	Sort the elements of an array using a Comparator.
<code>public static String toString(Object[])</code> <code>public static String toString(primitive[])</code>	Create a String containing the contents of an array.
Key Methods in <code>java.util.Collections</code>	Descriptions
<code>static int binarySearch(List, key)</code> <code>static int binarySearch(List, key, Comparator)</code>	Search a "sorted" List for a given value, return an index or insertion point.
<code>static void reverse(List)</code>	Reverse the order of elements in a List.
<code>static Comparator reverseOrder()</code> <code>static Comparator reverseOrder(Comparator)</code>	Return a Comparator that sorts the reverse of the collection's current sort sequence.
<code>static void sort(List)</code> <code>static void sort(List, Comparator)</code>	Sort a List either by natural order or by a Comparator.

- **Métodos en List, Set y Map**

TABLE 7-5 Key Methods in List, Set, and Map

Key Interface Methods	List	Set	Map	Descriptions
<code>boolean add(element)</code> <code>boolean add(index, element)</code>	X X	X		Add an element. For Lists, optionally add the element at an index point.
<code>boolean contains(object)</code> <code>boolean containsKey(object key)</code> <code>boolean containsValue(object value)</code>	X	X	X X	Search a collection for an object (or, optionally for Maps a key), return the result as a boolean.
<code>object get(index)</code> <code>object get(key)</code>	X		X	Get an object from a collection, via an index or a key.
<code>int indexOf(object)</code>	X			Get the location of an object in a List.
<code>Iterator iterator()</code>	X	X		Get an Iterator for a List or a Set.
<code>Set keySet()</code>			X	Return a Set containing a Map's keys.
<code>put(key, value)</code>			X	Add a key/value pair to a Map.
<code>remove(index)</code> <code>remove(object)</code> <code>remove(key)</code>	X X	X	X	Remove an element via an index, or via the element's value, or via a key.
<code>int size()</code>	X	X	X	Return the number of elements in a collection.
<code>Object[] toArray()</code> <code>T[] toArray(T[])</code>	X	X		Return an array containing the elements of the collection.

</OBJECTIVE 6.5 : Using the Collections Framework>

<OBJECTIVE 6.3 AND OBJECTIVE 6.4 : Generic Types>

<SUMARY>

6.3 Write code that uses the generic versions of the Collections API, in particular the Set, List, and Map interfaces and implementation classes. Recognize the limitations of the nongeneric

Collections API and how to refactor code to use the generic versions.

6.4 Develop code that makes proper use of type parameters in class/interface declarations, instance variables, method arguments, and return types; and write generic methods or methods that make use of wildcard types and understand the similarities and differences between these two approaches.

</SUMARY>

136. Se puede trabajar polimórficamente con las colecciones:

```
List<Integer> myList = new ArrayList<Integer>();
```

Pero que sucede cuando se quiere declarar de la siguiente manera:

```
class Parent { }
class Child extends Parent { }
List<Parent> myList = new ArrayList<Child>();
```

El código anterior **NO COMPILA**, porquéé??,,, en colecciones existe una regla muy simple: cuando tu declaras un tipo <Parent> la intanciacion también debe ser del mismo tipo <Parent> por mas que sea uno sea subtipo de otro, el compilador va a marcar error.

Ejemplos de casos similares:

```
List<Object> myList = new ArrayList<JButton>(); // NO!
List<Number> numbers = new ArrayList<Integer>(); // NO!
// recuerda que Integer es subtipo de Number
```

Pero esto es ok:

```
List<JButton> myList = new ArrayList<JButton>(); // yes
List<Object> myList = new ArrayList<Object>(); // yes
List<Integer> myList = new ArrayList<Integer>(); // yes
```

PEROO, si trabajas con Array si puedes alcanzar esto:

```
import java.util.*;
class Parent { }
class Child extends Parent { }
public class TestPoly {
    public static void main(String[] args) {
        Parent[] myArray = new Child[3]; // yes
    }
}
Object[] myArray = new JButton[3]; // yes
```

Pero esto no:

```
List<Object> list = new ArrayList<JButton>(); // NO!
```

137. Métodos Genéricos

Si por ejemplo tuvieramos el siguiente código:

```
abstract class Animal {  
    public abstract String accion();  
}  
  
class Dog extends Animal {  
    public String accion() {  
        return "I Dog";  
    }  
}  
  
public class CollectionTypeGenerics {  
    public static void main(String[] args) {  
        List<Dog> dogs = new ArrayList<Dog>();  
        dogs.add(new Dog());  
        dogs.add(new Dog());  
        dogs.add(new Dog());  
        addAll(dogs); // error de compilación  
    }  
  
    public static void addAll(List<Animal> animals){  
        animals.add(new Dog());  
    }  
}
```

Esto marcaría error de compilación, debido a que se quiere pasar una colección de **Dog** a una colección de **Animal**, y por lo visto anteriormente esto no se puede hacer.

Pero AHORA... que es lo que se puede hacer, si tengo una colección de **Cat** y quiero también hacer uso de ella????

Lo que se puede hacer es lo siguiente:

Cambiar el método:

```
public static void addAll(List<Animal> animals)
```

Por:

```
public static void addAll(List<? extends Animal> animals)
```

Con esto le decimos al método que acepte cualquier colección que tenga como elementos a un **subtipo** de **Animal**.

Pero ahora, el compilador marca error en:

```
public static void addAll(List<? extends Animal> animals){  
    animals.add(new Dog()); // NO! Can't add if we  
                           // use <? extends Animal>  
}
```

Cuando utilizamos esta forma no permite hacer un **add()**, pero si podríamos hacer un recorrido:

```
public static void addAll(List<? extends Animal> animals){  
    for (Animal animal : animals) {  
        System.out.println(animal.accion());  
    }  
}
```

Con esto solucionamos un problema, pero nos trae otro....que sucede realmente:

Primero, en `<? extends Animal>` la palabra **extends** se utiliza para representar **subtipos** o clases que implementan una **interface**.

Esto **ES INCORRECTO**: `<? implements Serializable>`. Si tu quieres indicar al método que tome cualquier tipo que implemente la interface Serializable, tu debes indicarlo con la palabra **extends**, así: `<? extends Serializable>`. Extraño pero es lo correcto

Para que no marque error de compilación, en nuestro código anterior, también lo que se puede hacer es :

```
public static void addAll(List<? super Dog> animals) {
    for (Animal animal : animals) {
        System.out.println(animal.accion());
    }
}
```

No marca error porque le estamos indicando que vamos a trabajar con instancias de Dog.

Después de esto surge una pregunta, hay alguna diferencia entre estos dos métodos:

```
public void foo(List<?> list) { }
public void foo(List<Object> list) { }
```

La respuesta es si. El `<?>` no indica ni extends ni super, lo que significa “cualquier Tipo”. Y puede aceptar `List<Dog>`, o `List<Cat>`. En cambio `<Object>` indica que solo acepta Object, no `List<Dog>`, o un `List<Cat>`.

Pregunta de examen:

Cual compila:

- 1) `List<?> list = new ArrayList<Dog>();`
- 2) `List<? extends Animal> aList = new ArrayList<Dog>();`
- 3) `List<?> foo = new ArrayList<? extends Animal>();`
- 4) `List<? extends Dog> cList = new ArrayList<Integer>();`
- 5) `List<? super Dog> bList = new ArrayList<Animal>();`
- 6) `List<? super Animal> dList = new ArrayList<Dog>();`

30 5

3

Repuesta:

Las que compilan son: la 1, 2 y 5

Las que no compilan son:

3) `List<?> foo = new ArrayList<? extends Animal>();` Tu no puedes poner cualquier comodín(?) en la creación del objeto, error en `new ArrayList<? extends Animal>();`

4) `List<? extends Dog> cList = new ArrayList<Integer>();` Tu no puedes asignar una lista de Integer a una lista que solo acepta Dog o subtipos de Dog

6) List<? super Animal> dList = new ArrayList<Dog>(); Tu no puedes asignar un Dog a un List<? super Animal> y también Dog esta por debajo de la jerarquia de Animal. Solo <Animal> u <Object> serían legales.

138. Declaraciones Genéricas

Ahora veremos como crear colecciones seguras y como declarar variables de referencia incluyendo argumentos y tipos de retorno genéricos.

Hechemos un vistoso al API, y veamos como esta declarada la interface List:

```
public interface List<E> extends Collection<E>
```

<E> indica un tipo, es por eso que cuando tu escribes tu código, puede cambiar de un generic List a List<Dog>, List<Integer>. Cualquier identificador de Java puede trabajar ahí, E (por “Element”) es usado cuando la plantilla es una colección. El otro conocido es T (“por Type”) es usado para designar cosas que no son colecciones.

Puedes tener códigos como:

```
public interface IGeneric<T> {
    public int add(T value);
}

public interface IGeneric<T extends Dog> {
    public int add(T value);
}
```

Lo que si no compilaría es:

```
public interface IGeneric {
    public int add(T value);      // T no ha sido definido como tipo
}
```

Ejemplo de uso:

```
class RentalGeneric<T> { // "T" is for the type
    // parameter
    private List<T> rentalPool; // Use the class type for the
                                // List type
    private int maxNum;

    public RentalGeneric(int maxNum, List<T> rentalPool) { // constructor takes
        // a
        // List of the class type
        this.maxNum = maxNum;
        this.rentalPool = rentalPool;
    }

    public T getRental() { // we rent out a T
        // blocks until there's something available
        return rentalPool.get(0);
    }

    public void returnRental(T returnedThing) { // and the renter
        // returns a T
        rentalPool.add(returnedThing);
    }
}
```

Cuando uno instancie la clase RentalGeneric va a especificar el tipo con el que se va a trabajar, Lo que va ayudar cuando por ejemplo invoquen su método **getRental()** a no estar casteando por cada invocación.

Haciendo uso de RentalGeneric:

```
class TestRental {
    public static void main(String[] args) {
        // make some Cars for the pool
        Car c1 = new Car();
        Car c2 = new Car();
        List<Car> carList = new ArrayList<Car>();
        carList.add(c1);
        carList.add(c2);
        RentalGeneric<Car> carRental = new RentalGeneric<Car>(2, carList);
        // now get a car out, and it won't need a cast
        Car carToRent = carRental.getRental();
        carRental.returnRental(carToRent);
        // can we stick something else in the original carList?
        // carList.add(new Cat("Fluffy")); // mararía error
    }
}
```

Otros ejemplos de uso de tipos genericos:

```
class TestGenerics<T> { // as the class type
    T anInstance; // as an instance variable type
    T[] anArrayOfTs; // as an array type

    TestGenerics(T anInstance) { // as an argument type
        this.anInstance = anInstance;
    }

    T getT() { // as a return type
        return anInstance;
    }
}

class UseTwo<T, X> {
    T one;
    X two;

    UseTwo(T one, X two) {
        this.one = one;
        this.two = two;
    }

    T getT() {
        return one;
    }

    X getX() {
        return two;
    }

    // test it by creating it with <String, Integer>
    public static void main(String[] args) {
        UseTwo<String, Integer> twos = new UseTwo<String, Integer>("foo", 42);
        String theT = twos.getT(); // returns a String
        int theX = twos.getX(); // returns Integer, unboxes to int
    }
}
```

```

class AnimalHolder<T extends Animal> { // Usar "T" en vez
    // de "?"
    T animal;

    public static void main(String[] args) {
        AnimalHolder<Dog> dogHolder = new AnimalHolder<Dog>(); // OK
        //AnimalHolder<Integer> x = new AnimalHolder<Integer>(); // NO!
    }
}

```

139. Creando métodos genéricos:

Usando un método genérico, nosotros podemos declarar un método genérico sin un tipo específico, y devolver información basada en el tipo del objeto pasado al método. Por ejemplo:

```

class CreateAnArrayList {
    public <T> void makeArrayList(T t) { // take an object of an
        // unknown type and use a
        // "T" to represent the type
        List<T> list = new ArrayList<T>(); // now we can create the
        // list using "T"
        list.add(t);
    }
}

```

Si quisieras restringir para que acepte un determinado tipo se tendría q hacer:

```

public <T extends Number> void makeArrayList(T t) {
}

```

Por regla el tipo `<T>` debe ser usado despues del nombre de la clase(`CreateAnArrayList<T extends Number>`) , o antes de indicar el tipo de retorno en un método(`public <T extends Number> void makeArrayList(T t)`).

e x a m

Watch One of the most common mistakes programmers make when creating generic classes or methods is to use a `<?>` in the wildcard syntax rather than a type variable `<T>`, `<E>`, and so on. This code might look right, but isn't:

```

public class NumberHolder<? extends Number> { }

While the question mark works when declaring a reference for a variable,
it does NOT work for generic class and method declarations. This code is not legal:

public class NumberHolder<?> { ? aNum; } // NO!

```

But if you replace the `<?>` with a legal identifier, you're good:

```

public class NumberHolder<T> { T aNum; } // Yes

```

</OBJECTIVE 6.3 AND OBJECTIVE 6.4 : Generic Types>

</CERTIFICATION_OBJECTIVES>

</CAP7: Generics and Collections >

<CAP8: Inner Classes>

<CERTIFICATION OBJECTIVES>

- Inner Classes
- Method-Local Inner Classes
- Anonymous Inner Classes
- Static Nested Classes

<OBJECTIVES>

140. Resumen Previo: Las **clases internas** son clases que se declaran dentro de otras clases. Para acceder a ellas, en la declaración se debe indicar el nombre de la externa.
Ejemplo:

```
ClaseExterna.ClaseInterna ci = new ClaseExterna().new ClaseInterna();
```

Desde el cuerpo de la clase externa se accede de forma normal, instanciando un objeto del tipo de la clase interna.

Desde la clase interna se tiene acceso a todos los miembros de la externa, incluyendo los privados.

Existen otros tres tipos de clases internas:

- **clases internas locales a un método** (se definen dentro de un método de la clase externa y no pueden utilizar las variables externas a la clase, pero internas al método, al menos, que las variables sean finales(**final**)),
- **clases internas anónimas** (se define su contenido en el momento de instanciarla, ej:
`Thread hilo = new Thread() {public void run() {}};`. Estas clases no pueden heredar al mismo tiempo de otra clase e implementar un interfaz, o uno u otro.
Por último las:
- **clases internas estáticas** se declaran con static y no tienen acceso al contenido que no sea estático de la clase externa.

<OBJECTIVE: Regular Inner Classes>

141. Codificando un “Regular” Inner Classes.

Cuando se utiliza el termino *regular* esta representa un inner class que no es:

- Static
- Method
- Anonymous

Pero en el libro dice: Que se utilizará el termino “**inner class**”, para referirse a todas las clases internas, el termino **Regular**, es para que sepas distinguirlas.

Tú defines un **inner class** dentro de los corchetes del **outer class**, pero fuera de cualquier método:

```
class MyOuter {
    class MyInner { }
    public void method1() {}
}
```

Cuando se compila esta clase : javac MyOuter.java

El resultado son dos .class:

```
MyOuter.class
MyOuter$MyInner.class
```

- Un **regular inner class** no puede tener declaraciones staticas de cualquier tipo.
- Solo se puede acceder a un **inner class** a través de la instancia de un **outer class**.

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

    public void makeInner() {
        MyInner in = new MyInner();
        in.seeOuter();
    }

    class MyInner {
        private String property1 = "hola";
        public void seeOuter() {
            System.out.println("Outer x es " + x);           //Outer x es 7
            System.out.println("Inner ref es " + this);      //Inner ref es
                                                       //demo.MyOuter$MyInner@92s3
            System.out.println("Outer ref es " + MyOuter.this); //Outer ref es
                                                       //demo.MyOuter@130c
        }
    }
}
```

- La forma de instanciar un inner class es:

```
public class MyOuterTester {
    public static void main(String[] args) {
        //como instanciar una clase inner fuera de su
        //clase outer o contenedora
        MyOuter mo = new MyOuter();
        MyOuter.MyInner inner = mo.new MyInner();
        inner.seeOuter();
        MyOuter.MyInner inner2 = new MyOuter().new MyInner();
        inner2.seeOuter();
    }
}
```

La salida es:

```
/*
Outer x is 7
Inner class ref is MyOuter$MyInner@113708
Outer class ref is MyOuter@33f1d7
*/
```

- Una regular **inner class** es un miembro de un **outer class** así como los son sus **variables y métodos de instancia**, por lo que los siguientes modificadores de acceso pueden ser aplicadas a una inner class:

- final
- abstract
- public
- private
- protected
- static—but static turns it into a static nested class not an inner class.
- strictfp

</OBJECTIVE: Regular Inner Classes>

<OBJECTIVE: Method – Local Inner Classes>

142. Una **regular inner class** esta dentro de otra clases, pero fuera de cualquier método (En otras palabras, al mismo nivel que una variable de instancia es declarada). PERO tu puedes definir un **inner class** dentro de un método:

```
class MyOuter2 {
    private String x = "Outer2";

    void doStuff() {
        final String s = "hola mundo";
        class MyInner {
            public void seeOuter() {
                System.out.println("Outer x is " + x);
                System.out.println("Outer x is " + s); //s ever must final
            } // close inner class method
        } // close inner class definition
    } // close outer class method doStuff()
} // close outer class
```

- Una **method-local inner class** no puede ser instanciado de ningún lado solo dentro del método, no funciona como un **regular inner class**.
- Un **method-local inner class** es tratado como una variable local de método, por lo tanto no puede ser **static**, ni **public**, ni **protected**, ni **transient**, los únicos modificadores que puedes aplicar son **abstract** y **final**, pero no ambas al mismo tiempo.
- Un **method-local inner class** solo puede acceder a variables externas, pero dentro del método, solo puede acceder si estas son de tipo **final**. Pero si puede acceder a variables que estan fuera del método.
- Sobre un **method-local inner class** dentro de un método estático:



Remember that a local class declared in a static method has access to only static members of the enclosing class, since there is no associated instance of the enclosing class. If you're in a static method there is no this, so an inner class in a static method is subject to the same restrictions as the static method. In other words, no access to instance variables.

</OBJECTIVE: Method – Local Inner Classes >

<OBJECTIVE: Anonymous Inner Classes >

143. Observa la sintaxis:

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

- **Food** tiene un variable p que no es una instancia de **Popcorn** sino es una instancia de un anonimo(sin nombre) subclase de **Popcorn**.
- Siempre la clase anónima debe cerrar con ":".

Ahora observa esta otra:

```
class Popcorn {  
    public void pop() {  
        System.out.println("popcorn");  
    }  
}  
  
class Food {  
    Popcorn p = new Popcorn() {  
        public void sizzle() {  
            System.out.println("anonymous sizzling popcorn");  
        }  
  
        public void pop() {  
            System.out.println("anonymous popcorn");  
        } //Si este método no estaría aquí , se llama al método que ésta en el parente  
    };  
  
    public void popIt() {  
        p.pop(); // OK, Popcorn has a pop() method  
        p.sizzle(); // Not Legal! Popcorn does not have sizzle()  
    }  
}
```

Que es lo q ocurre?. Si recordamos cuando aplicabamos polimorfismo:

```
Animal animal = new Horse();
```

Que ocurría si tratabamos de llamar a un método que solo esta en la clase Horse y que no ha sido heredado de animal, por ejemplo.

```
animal.methodOnlyInHorse()
```

Pues lo que sucedería es que el compilador marcaría error ya que la clase **Animal** no tiene el método **methodOnlyInHorse()**.

¿Por qué este recordatorio?, lo que pasa es que este mismo concepto se aplica a las clases anónimas, como observaran cuando invocamos al método **p.sizzle()**, el compilador marca error porque ese método no existe en la clase **Popcorn**, **mucho cuidado con esto.**

144. Pero ahí no acaba todo::::: “anonymous implementer of the specified interface type.”

Observemos lo siguiente:

```
interface Cookable {  
    public void cook();  
}  
  
class Food {  
    Cookable c = new Cookable() {  
        public void cook() {  
            System.out.println("anonymous cookable implementer");  
        }  
    };  
}
```

Te preguntarás: ¿Pero si Cookable es una interface y que hace el **new** ahí???

Lo que realmente está pasando, no es que se esté creando un objeto de una interface, ya que eso sería algo herrido, lo que se está haciendo es crear un objeto anónimo que está implementando la **interace Coockable**. Y este objeto anónimo al implementar Cookable puede ser seteado en la variable **C** (Se está aplicando polimorfismo).

145. Las dos formas anteriores pueden ser declaradas también dentro de métodos.

```
public static void a() {      //static or no static  
  
    Cookable c = new Cookable() {  
        public void cook() {  
            //insert code here  
        }  
    };  
}
```

146. Y para cerrar. El uso de instanciaciones de clases anónimas que implementan interfaces, pasadas como parámetros:

Observemos:

```
class MyWonderfulClass {  
    void go() {  
        Bar b = new Bar();  
        b.doStuff(ackWeDoNotHaveAFoo!); // No compilar esto en casa  
    }  
}  
  
interface Foo {  
    void foof();  
}  
  
class Bar {  
    void doStuff(Foo f) {  
    }  
}
```

Como podemos ver, ninguna clase implementa **Foo**, entonces ¿cómo podríamos pasar un parámetro al método `doStuff(...)`?

La respuesta essss:::: creando un objeto anónimo que implementa la interface **Foo**, de la siguiente manera:

```
class MyWonderfulClass {  
    void go() {  
        Bar b = new Bar();  
        b.doStuff(new Foo() {  
            public void foof() {  
                System.out.println("foofy");  
            } // end foof method  
        }); // end inner class def, arg, and b.doStuff stmt.  
    } // end go()  
} // end class
```

<OBJECTIVE: Anonymous Inner Classes >

<OBJECTIVE: Static Nested Classes>

147. Un ejemplo simple:

```
class BigOuter {  
    static class Nested {  
    }  
}
```

En realidad no es un “**static**”, no existe tal cosa como un **static class**. El modificador **static** en este caso indica que la clase interna es un miembro estático de la otra clase. Lo que significa que puede ser accedido, como cualquier otro miembro estático, sin tener una instancia de la clase externa.

148. Instanciando y usando Static Nested Classes:

Observemos un ejemplo:

```
class BigOuter {
    static class Nest {
        void go() {
            System.out.println("hi");
        }
    }
}

class Broom {
    static class B2 {
        void goB2() {
            System.out.println("hi 2");
        }
    }
}

public static void main(String[] args) {
    BigOuter.Nest n = new BigOuter.Nest(); // both class names
    n.go();
    B2 b2 = new B2(); // access the enclosed class
    b2.goB2();
}
```

Como se observa, solo con `new BigOuter.Nest()` es necesario para instanciar un static inner class (Esto porque `BigOuter` está en el exterior de la clase `Broom`). Pero en el caso de `new B2()` solo se instancia la clase debido a que está dentro de `Broom`, pero ojo::: `B2` tiene modificador static, sino no podría acceder, debido a que el método `void main` es static.

</OBJECTIVE: Static Nested Classes>
</OBJECTIVES>
</CERTIFICATION_OBJETIVES>
</CAP8: Inner Classes>

<CAP9: Threads >

<CERTIFICATION OBJECTIVES>

<OBJECTIVES>

<OBJECTIVE 4.1: Defining, Instantiating, and Starting Threads>

<SUMMARY>

4.1 Write code to define, instantiate, and start new threads using both `java.lang.Thread` and `java.lang.Runnable`.

</SUMMARY>

149. Que es un Thread

La Máquina Virtual Java (JVM) es un sistema multihilo. Es decir, es capaz de ejecutar varios hilos de ejecución simultáneamente. La JVM gestiona todos los detalles, asignación de tiempos de ejecución, prioridades, etc., de forma similar a como gestiona un Sistema Operativo múltiples procesos. La diferencia básica entre un proceso de Sistema Operativo y un `Thread` Java es que los hilos corren dentro de la JVM, que es un proceso del Sistema Operativo y por tanto comparten todos los recursos, incluida la memoria y las variables y objetos allí definidos.

Java da soporte al concepto de `Thread` desde el propio lenguaje, con algunas clases e interfaces definidas en el paquete `java.lang` y con métodos específicos para la manipulación de `Threads` en la clase `Object`.

Desde el punto de vista de las aplicaciones los hilos son útiles porque permiten que el flujo del programa sea dividido en dos o más partes, cada una ocupándose de alguna tarea de forma independiente. Por ejemplo un hilo puede encargarse de la comunicación con el usuario, mientras que otros actúan en segundo plano, realizando la transmisión de un fichero, accediendo a recursos del sistema (cargar sonidos, leer ficheros ...), etc. De hecho, todos los programas con interface gráfico (`AWT` o `Swing`) son multihilo porque los eventos y las rutinas de dibujado de las ventanas corren en un hilo distinto al principal.

150. En Java “thread” significa dos diferentes cosas:

- Una instancia de la clase `java.lang.Thread`
- Un hilo de ejecución(A thread of execution).

151. La siguiente tabla muestra los métodos para manejos de threads(hilos), lo métodos estáticos están en *italic*:

TABLE 9-2 Key Thread Methods

Class Object	Class Thread	Interface Runnable
<code>wait()</code>	<code>start()</code>	<code>run()</code>
<code>notify()</code>	<code>yield()</code>	
<code>notifyAll()</code>	<code>sleep()</code>	
	<code>join()</code>	

152. Making a Thread

Un hilo(thread) en Java es una instancia de **java.lang.Thread**. Para el examen tu necesitas conocer como mínimo, los siguientes métodos:

- start()
- yield()
- sleep()
- run()

La acción ocurre en el método **run()**.

```
public void run() {  
    // your job code goes here  
}
```

Siempre escriba código en el método **run()**, que quiera que se ejecute en un hilo separado.

Tu puedes definir e instanciar un hilo(thread) con una de estas dos maneras:

- Extendiendo de la clase **java.lang.Thread**.
- Implementando la interface **Runnable**.

Debes conocer acerca de ambos para el examen.

En el mundo real se prefiere más implementar la interface **Runnable** que extender de **Thread**.

Extender de la clase **Thread** es muy facil, pero no es una buena practica OO. Porque las subclases deberían ser reservas para versiones especiales de muchas **superclases**.

- Extendiendo de **java.lang.Thread**

- o Se logra extendiendo de la clase **java.lang.Thread** y
- o Sobreescribiendo el método **run()**.

Ejemplo:

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Important job running in MyThread");  
    }  
}
```

- Implementando la interface **java.lang.Runnable**

Ejemplo:

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Important job running in MyRunnable");  
    }  
}
```

Tu puedes pasar una sola instancia de Runnable a múltiples objetos Thread :

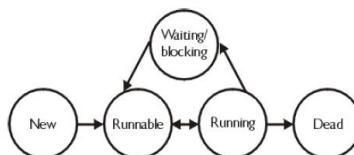
```
class TestThreads {  
    public static void main(String[] args) {  
        MyRunnable r = new MyRunnable();  
        Thread foo = new Thread(r);  
        Thread bar = new Thread(r);  
        Thread bat = new Thread(r);  
        foo.start();  
        bar.start();  
        bat.start();  
    }  
}
```

Esto significa que varios hilos de ejecución están corriendo la misma tarea.

- Los constructores de Thread, pueden ser:

- Thread()
- Thread(Runnable target)
- Thread(Runnable target, String name)
- Thread(String name)

-



Cuando un **Thread** ha sido instanciado pero no **iniciado** (en otras palabras, el método **start()** no ha sido invocado en la instancia del **Thread**), el thread está en un **nuevo estado (new state)**. En este estado el hilo(thread) no es aún considerado como **vivo(alive)**. Cuando el método **start()** es llamado, el hilo(thread) es considerado como **vivo(alive)** (Siempre que el método **run()** no esté en funcionamiento aún). Un hilo(thread) es considerado **muerto(dead)** después de que el método **run()** sea completado. El método **isAlive()** es la mejor manera de determinar si un **hilo(thread)** ha sido **iniciado(started)** pero que aún no ha completado el método **run()**. (Nota: El método **getState()** es muy usado para debugging, pero no tienes que conocerlo para el examen).

Ejemplo de uso:

```
class FooRunnable implements Runnable {  
    public void run() {  
        for (int x = 1; x < 6; x++) {  
            System.out.println("Runnable running");  
        }  
    }  
}  
  
class TestThreads {  
    public static void main(String[] args) {  
        FooRunnable r = new FooRunnable();  
        Thread t = new Thread(r);  
        t.start();  
    }  
}
```

La salida es:

```
Runnable running
Runnable running
Runnable running
Runnable running
Runnable running
```

e x a m

Watch

There's nothing special about the run() method as far as Java is concerned. Like main(), it just happens to be the name (and signature) of the method that the new thread knows to invoke. So if you see code that calls the run() method on a Runnable (or even on a Thread instance), that's perfectly legal. But it doesn't mean the run() method will run in a separate thread! Calling a run() method directly just means you're invoking a method from whatever thread is currently executing, and the run() method goes onto the current call stack rather than at the beginning of a new call stack. The following code does not start a new thread of execution:

```
Runnable r = new Runnable();
r.run(); // Legal, but does not start a separate thread
```

- Ejemplos de uso:

```
class NameRunnable implements Runnable {
    public void run() {
        System.out.println("NameRunnable running");
        System.out.println("Run by " + Thread.currentThread().getName());
    }
}

class NameThread {
    public static void main(String[] args) {
        NameRunnable nr = new NameRunnable();
        Thread t = new Thread(nr);
        t.setName("Fred");
        t.start();
    }
}

/*La Salida es :
   NameRunnable running
   Run by Fred
* */
```

Si se comenta t.setName("Fred") , la salida sería:

```
class NameThread {
    public static void main(String[] args) {
        NameRunnable nr = new NameRunnable();
        Thread t = new Thread(nr);
        //t.setName("Fred");
        t.start();
    }
}

/*La Salida es :
   NameRunnable running
   Run by Thread-0
* */
```

Y si :

```
class NameThread {  
    public static void main(String[] args) {  
        NameRunnable nr = new NameRunnable();  
        System.out.println("thread is "  
                           + Thread.currentThread().getName());  
    }  
}
```

La salida es:

```
/*La Salida es :  
 thread is main  
 * */
```

Y cuidado con esto :

```
class NameThread {  
    public static void main(String[] args) {  
        NameRunnable nr = new NameRunnable();  
        nr.run();  
    }  
}
```

La salida es:

```
/*La Salida es :  
 NameRunnable running  
 Run by main  
 * */
```

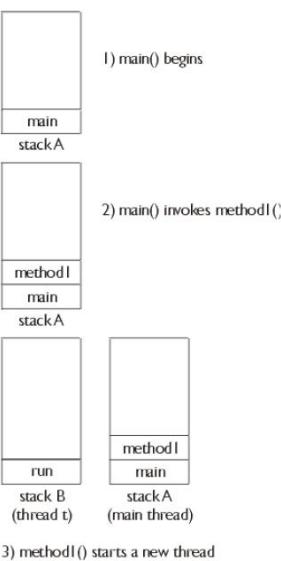
153. La siguiente imagen muestra el proceso de iniciación de un Thread.

FIGURE 9-1

Starting a thread

```
public static void main(String [] args) {  
    // running  
    // some code  
    // in main()  
    method1();  
    // running  
    // more code  
}
```

```
void method1() {  
    Runnable r = new MyRunnable();  
    Thread t = new Thread(r);  
    t.start();  
    // do more stuff  
}
```



154. Starting and Running Multiple Threads

El siguiente código crea un único **Runnable** y tres instancias de **Thread** que toman la misma instancia de **Runnable** y cada hilo(thread) es asignado un nombre. Por último se inicializan con el método **start()**.

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x <= 3; x++) {
            System.out.println("Run by "
                + Thread.currentThread().getName()
                + ", x is " + x);
        }
    }
}

class ManyNames {
    public static void main(String[] args) {
        // Make one Runnable
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr);
        Thread two = new Thread(nr);
        Thread three = new Thread(nr);
        one.setName("Fred");
        two.setName("Lucy");
        three.setName("Ricky");
        one.start();
        two.start();
        three.start();
    }
}
```

La salida es:

```
/*La Salida es:
Run by Fred, x is 1
Run by Fred, x is 2
Run by Fred, x is 3
Run by Lucy, x is 1
Run by Lucy, x is 2
Run by Lucy, x is 3
Run by Ricky, x is 1
Run by Ricky, x is 2
Run by Ricky, x is 3
*/

```

Pareciera que un hilo se ejecuta tras otro debido a que primero imprime Fred, luego Lucy y luego Ricky, pero esto no es garantizado. Peeeero, que pasaría si cambiamos el método **run()**, por este otro:

```
public void run() {
    for (int x = 1; x <= 400; x++) {
        System.out.println("Run by " + Thread.currentThread().getName()
            + ", x is " + x);
    }
}

/*
 * La salida es
 ...
Run by Fred, x is 345
Run by Lucy, x is 337
Run by Ricky, x is 310
Run by Lucy, x is 338
Run by Ricky, x is 311
Run by Lucy, x is 339
Run by Ricky, x is 312
Run by Fred, x is 346
Run by Lucy, x is 343
Run by Fred, x is 347
Run by Lucy, x is 344
...
*/

```

Ahora si se ve la diferencia, lo que pasa es que son hilos de ejecución en paralelo y no esperan a que el hilo anterior termine.

Si tu quieres llamar a **start()** por segunda vez, este causa una Excepción.

Solo un nuevo hilo(thread) puede ser empezado(started), **y solo una vez**. Un **Runnable thread** o un **thread muerto** no puede ser reiniciado(restarted).

155. The Thread Scheduler

The thread scheduler es la parte de la JVM (aunque la mayoría de los hilos de Java son subprocessos nativos en el sistema operativo subyacente), el decide que hilo debe ejecutarse en cualquier momento dado. Asumiendo un único procesador, sólo un hilo puede ejecutar a la vez. Solo un stack(pila) puede ser ejecutado a la vez. Solo un **hilo(thread)** en estado **ejecutable(runnable)** puede ser elegido por el **Thread Scheduler**

156. Métodos de la clase `java.lang.Object`.

Todos las clases en Java heredan estos 3 métodos relacionados al manejo de **Threads**

```
public final void wait() throws InterruptedException  
public final void notify()  
public final void notifyAll()
```

</OBJECTIVE 4.1: Defining, Instantiating, and Starting Threads>

<OBJECTIVE 4.2: Thread States and Transitions >

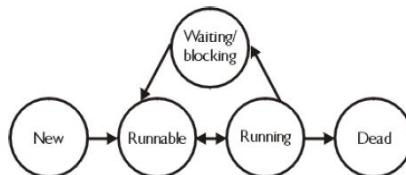
<SUMMARY>

4.2 Recognize the states in which a thread can exist, and identify ways in which a thread can transition from one state to another.

</SUMMARY>

157. Veremos tres estados del thread: **new**, **runnable**, **dead**. , pero hay mas

158. Thread States



- **New:** En este estado el **thread** es después de que el la instancia de Thread ha sido creada, pero el método **start()** no ha sido invocado on el thread. Indica que está vivo el hilo pero no en está en ejecución. En este punto el thread es considerado **not alive**.
- **Runnable:** Este estado es cuando el thread ha sido marcado como apto para ser ejecutado, pero no lo ejecuta (Ojo con esto). Un thread primero entra al estado **runnable** después puede pasar a **running**.
Cuando un thread está en este estado es considerado **alive**.
- **Running:** Este estado es cuando el **thread** es elegido (del **runnable pool**) por el scheduler.

- **Waiting/blocking/sleeping:** Son tres estados combinados en uno. Pero tienen algo en común: es considerado **alive**, pero no es elegible para ejecutar. En otras palabras no es **runnable**, pero podría retornar a un estado **runnable** si ocurre un evento en particular. Un **thread** puede ser bloqueado en espera de un recurso, en cuyo caso el es enviado al **runnable**, en este caso el evento que se envía a **runnable** es “**the availability of the resource**”.

Un **thread** puede estar en **sleeping** porque el thread de ejecución le dice que duerma por un periodo de tiempo, en este caso el evento que se envía a **runnable** es “**wakes up**” porque el tiempo de **sleep** ha expirado.

O el **thread** puede estar en **waiting**, porque el thread de ejecución hace que espere, en cuyo caso el evento que envía al **runnable** es que otro thread envíe una notificación de que ya no va a ser necesario esperar.

- **Dead:** Un **thread** es considerado **Dead** cuando su método **run()** ha sido completado. Una vez que un **thread** está **muerto**, este nunca puede volver a la vida. Si invocas al **start()** en un **thread muerto**, obtendrás un runtime exception.

159. Sleeping

El método **sleep()** simplemente le dice al hilo de ejecución que duerma durante los milisegundos especificados. Se debería utilizar **sleep()** cuando se pretenda retrasar la ejecución del hilo. **sleep()** no consume recursos del sistema mientras el hilo duerme. De esta forma otros hilos pueden seguir funcionando.

Ejemplo de uso:

Ejemplo 1:

```
try {
    Thread.sleep(5*60*1000); // Sleep for 5 minutes
} catch (InterruptedException ex) {

}
```

Ejemplo 2:

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x < 4; x++) {
            System.out.println("Run by(b) " + Thread.currentThread().getName());
            try {
                //Thread.sleep(2000);
                // System.out.println("Run by(a) " + Thread.currentThread().getName());
            } catch (Exception ex) {
            }
        }
    }
}

class ManyNames {
    public static void main(String[] args) {
        // Make one Runnable
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr);
        one.setName("Fred");
        Thread two = new Thread(nr);
        two.setName("Lucy");
        Thread three = new Thread(nr);
        three.setName("Ricky");
        one.start();
        two.start();
        try {
            two.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        three.start();
    }
}
```

Recuerda que **sleep()** es un método estático, no pienses que un hilo puede poner a dormir a otro. Puedes poner el método **sleep()**, en cualquier parte ya que todo el código está siendo ejecutado por un hilo. En pocas palabras, cuando llamas a **sleep()**, pones a dormir el hilo actual.

e x a m

W a t c h

Just because a thread's sleep() expires, and it wakes up, does not mean it will return to running! Remember, when a thread wakes up, it simply goes back to the runnable state. So the time specified in sleep() is the minimum duration in which the thread won't run, but it is not the exact duration in which the thread won't run. So you can't, for example, rely on the sleep() method to give you a perfectly accurate timer. Although in many applications using sleep() as a timer is certainly good enough, you must know that a sleep() time is not a guarantee that the thread will start running again as soon as the time expires and the thread wakes.

160. Thread Priorities and yield

Para entender **yield()**, se debe entender el concepto de **thread priorities**.

Los Threads siempre se ejecutan con la misma prioridad, usualmente representada como un número entre el 1 y 10 (En algunos casos el rango es menor a 10). El **sheduler** en la mayoría de JVM usa preferentemente una prioridad basada en **scheduling**.

Para asignar una prioridad a un **thread**, se hace con el método **setPriority()**:

```
FooRunnable r = new FooRunnable();
Thread t = new Thread(r);
t.setPriority(8);
t.start();
```

Aunque la prioridad por defecto es 5, el hilo de la clase tiene tres constantes (static final variables) que definen la gama de prioridades de threads:

```
Thread.MIN_PRIORITY (1)
Thread.NORM_PRIORITY (5)
Thread.MAX_PRIORITY (10)
```

161. El método **yield()**

Este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que los hilos de menor prioridad no sufran inanición.

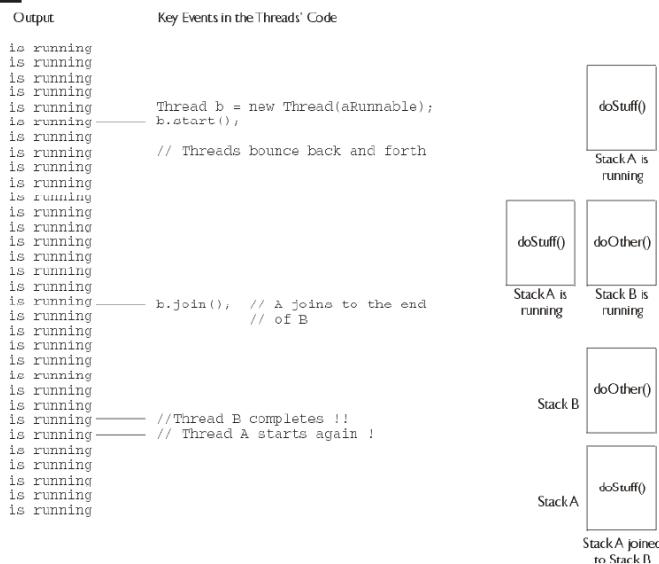
La intención de usar **yield()** es promover la toma correcta de threads basada en la prioridad. En realidad el método **yield()** no está garantizado para hacer lo que dice.

162. El método join()

Es un método no estático de la clase **Thread** permite a un hilo “unirse al final” de otro hilo.

Si tienes un hilo A, que no debe hacer nada hasta que el hilo B finalice su trabajo, entonces, tu dices que hilo B “join” hilo A. Esto significa que hilo A no será **Runnable** hasta que B aya **finalizado(dead)**.

FIGURE 9-3 The join() method



So far we've looked at three ways a running thread could leave the running state:

- **A call to sleep()** Guaranteed to cause the current thread to stop executing for at least the specified sleep duration (although it might be *interrupted* before its specified time).
- **A call to yield()** Not guaranteed to do much of anything, although typically it will cause the currently running thread to move back to runnable so that a thread of the same priority can have a chance.
- **A call to join()** Guaranteed to cause the current thread to stop executing until the thread it joins with (in other words, the thread it calls *join()* on) completes, or if the thread it's trying to join with is not alive, however, the current thread won't need to back out.

</OBJECTIVE 4.2: Thread States and Transitions >

<OBJECTIVE 4.3: Synchronizing Code>

<SUMARY>

4.3 Given a scenario, write code that makes appropriate use of object locking to protect static or instance variables from concurrent access problems.

</SUMARY>

163. ¿Te puedes imaginar que puede ocurrir cuando dos diferentes hilos tienen acceso a una sola instancia de una clase, y ambos hilos invocan métodos del objeto y estos métodos modifican el estado del objeto??.

Veamos:

El siguiente ejemplo trata de simular a dos personas Fred y Lucy que tienen acceso a la misma cuenta bancaria y hacen transacciones sobre ella.

```
class Cuenta {
    private int balance = 50;

    public int getBalance() {
        return balance;
    }

    public void retirar(int monto) {
        balance = balance - monto;
        if (balance < 0) {
            System.out.println("Cuenta está sobregirada!");
        }
    }
}

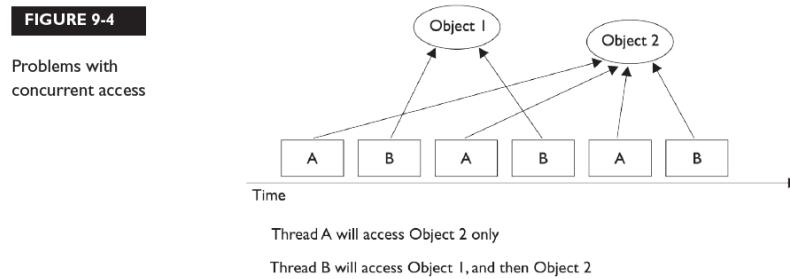
class GestorDeCuenta implements Runnable {
    private Cuenta cuenta = new Cuenta();

    public void run() {
        for (int x = 0; x < 5; x++) {
            hacerRetiro(10);
        }
    }

    private void hacerRetiro(int monto) {
        if (cuenta.getBalance() >= monto) {
            System.out.println(Thread.currentThread().getName() + " va a retirar " + monto);
            cuenta.retirar(monto);
            System.out.println(Thread.currentThread().getName() + " completa el retiro");
        } else {
            System.out.println("No hay fondo suficiente en la cuenta para "
                + Thread.currentThread().getName() + " balance: "
                + cuenta.getBalance());
        }
    }
}

class Principal {
    public static void main(String[] args) {
        GestorDeCuenta r = new GestorDeCuenta();
        Thread one = new Thread(r);
        Thread two = new Thread(r);
        one.setName("Fred");
        two.setName("Lucy");
        one.start();
        two.start();
    }
}
```

La siguiente imagen muestra que es lo que puede ocurrir si dos hilos tratan de acceder a un mismo objeto:



This problem is known as a "race condition," where multiple threads can access the same resource (typically an object's instance variables), and can produce corrupted data if one thread "races in" too quickly before an operation that should be "atomic" has completed.

Como protegemos nuestros datos, para que no ocurra esto...::??

Tu debes hacer dos cosas:

- Poner las variables como **private**.
- Sincronizar el código que modifica las variables.

Hacemos **synchronized** el método `hacerRetiro()`:

```
synchronized private void hacerRetiro(int monto) {  
    if (cuenta.getBalance() >= monto) {  
        System.out.println(Thread.currentThread().getName() + " va a retirar " + monto);  
        cuenta.retirar(monto);  
        System.out.println(Thread.currentThread().getName() + " completa el retiro");  
    } else {  
        System.out.println("No hay fondo suficiente en la cuenta para "  
            + Thread.currentThread().getName() + " balance: "  
            + cuenta.getBalance());  
    }  
}
```

Ahora nosotros garantizamos que un **thread** Lucy o Fred empiece el proceso (con el método `hacerRetiro()`) y que el otro thread no pueda acceder al método hasta que el primero complete el proceso (salir del método).

164. Synchronization and Locks(También podemos sincronizar bloques)

Ejemplos:

```
class SyncTest {  
    public void doStuff() {  
        System.out.println("not synchronized");  
        synchronized (this) {  
            System.out.println("synchronized");  
        }  
    }  
}
```

Otro:

```
public synchronized void doStuff() {  
    System.out.println("synchronized");  
}  
  
// Es equivalente a :  
  
public void doStuff() {  
    synchronized (this) {  
        System.out.println("synchronized");  
    }  
}
```

El método estático bloquea una instancia de clase mientras el método no estático bloquea sobre esta(**this**) instancia. Estas acciones no interfieren unos con otros en lo absoluto.

Nota: Solo se puede bloquear objetos y clases(Class), pero no variables de tipo de primitivo

165.

Table 9-1 lists the thread-related methods and whether the thread gives up its lock as a result of the call.

TABLE 9-1 Methods and Lock Status

Give Up Locks	Keep Locks	Class Defining the Method
wait ()	notify() (Although the thread will probably exit the synchronized code shortly after this call, and thus give up its locks.)	java.lang.Object
	join()	java.lang.Thread
	sleep()	java.lang.Thread
	yield()	java.lang.Thread

166. Thread Deadlock

El deadlock ocurre cuando dos **threads** son bloqueados. Ninguno puede correr hasta que el otro no deje su bloqueo, lo que puede ser para siempre.

Ejemplo de demostración de del **DeadLock** :

```
class DeadlockRisk {
    private static class Resource {
        public int value;
    }

    private Resource resourceA = new Resource();
    private Resource resourceB = new Resource();

    public int read() {
        synchronized (resourceA) { // May deadlock here
            synchronized (resourceB) {
                return resourceB.value + resourceA.value;
            }
        }
    }

    public void write(int a, int b) {
        synchronized (resourceB) { // May deadlock here
            synchronized (resourceA) {
                resourceA.value = a;
                resourceB.value = b;
            }
        }
    }
}
```

Asumiendo que **read()** es iniciado por un thread y **write()** es iniciado por otro, entonces hay **deadlock** en la línea 8 o 16. Ambos thread, el de lectura tendrá bloqueado a resourceA, el de escritura tendrá bloqueado a resourceB, y ambos estarán esperando el uno al otro.

</OBJECTIVE 4.3: Synchronizing Code>

<OBJECTIVE 4.4: Thread Interaction>

<SUMMARY>

4.4 Given a scenario, write code that makes appropriate use of wait, notify, or notifyAll.

</SUMMARY>

167. Lo último que tenemos que ver es cómo los hilos pueden interactuar el uno con el otro, para comunicar acerca de, entre otras cosas, su estado de bloqueo. La clase **Object** tiene tres métodos: **wait()**, **notify()**, y **notifyAll()** que ayudan a los hilos a comunicar acerca del estado de un evento.

168.

El método **wait()** pone al thread que lo invoque en estado de espera hasta que otro thread llame al *notify()* o al *notifyAll()*.

El método **notify()** se utiliza para poner de nuevo un thread que está detenido al estado Running. Si **notify()** se invoca en un objeto de un hilo que no es el propietario del bloqueo de ese objeto, se lanza **IllegalMonitorStateException**.

El método **notifyAll()** realiza el mismo trabajo que el **notify()** pero notifica a todos los threads en estado de espera.

169. Para el examen necesitas tener saber que...: **wait()**, **notify()**, y **notifyAll()** deben ser llamados desde **dentro de un contexto sincronizado!!.**

Ejemplo de uso:

```
class ThreadA {
    public static void main(String[] args) {
        ThreadB b = new ThreadB();
        b.start();

        synchronized (b) {
            try {
                System.out.println("Waiting for b to complete...");
                b.wait();
            } catch (InterruptedException e) {
            }
            System.out.println("Total is: " + b.total);
        }
    }
}

class ThreadB extends Thread {
    int total;

    public void run() {
        synchronized (this) {
            for (int i = 0; i < 100; i++) {
                total += i;
            }
            notify();
        }
    }
}
```

170.

e x a m
watch

When the `wait()` method is invoked on an object, the thread executing that code gives up its lock on the object immediately. However, when `notify()` is called, that doesn't mean the thread gives up its lock at that moment. If the thread is still completing synchronized code, the lock is not released until the thread moves out of synchronized code. So just because `notify()` is called doesn't mean the lock becomes available at that moment.

171. Ejemplo de uso de `notifyAll()`:

```
class Reader extends Thread {  
    Calculator c;  
  
    public Reader(Calculator calc) {  
        c = calc;  
    }  
  
    public void run() {  
        synchronized (c) {  
            try {  
                System.out.println("Waiting for calculation...");  
                c.wait();  
            } catch (InterruptedException e) {}  
            System.out.println("Total is: " + c.total);  
        }  
    }  
  
    public static void main(String[] args) {  
        Calculator calculator = new Calculator();  
        new Reader(calculator).start();  
        new Reader(calculator).start();  
        new Reader(calculator).start();  
        calculator.start();  
    }  
}  
  
class Calculator extends Thread {  
    int total;  
  
    public void run() {  
        synchronized (this) {  
            for (int i = 0; i < 100; i++) {  
                total += i;  
            }  
            notifyAll(); // Usar notifyAll() Cuando muchos Threads pueden estar esperando.  
        }  
    }  
}
```

</OBJECTIVE 4.4: Thread Interaction >

</OBJECTIVES>

</CERTIFICATION_OBJETIVES>

</CAP9: Threads >

<CAP10: Development>

<CERTIFICATION_OBJECTIVES>

<OBJECTIVES>

<OBJECTIVE 7.2: Using the javac and java Commands>

<SUMMARY>

7.1 Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.

7.2 Given an example of a class and a command-line, determine the expected runtime behavior.

7.5 Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully.

</SUMMARY>

172. Compiling with javac (Compilando con javac)

Este comando es utilizado para invocar el compilador de Java. En el capítulo 5 se hablo acerca del mecanismo de aserciones y cuando tu puedes usar las opciones `-source` cuando compillas un archivo. Hay muchas otras opciones que tu puedes especificar cuando ejecutas `javac`, opciones para generar debugging information o compiler warnings por ejemplo. Para el examen tu necesitas entender el `-classpath` y `-d` options. Aquí un vistoso del comando `javac`:

```
javac [options] [source files]
```

Hay un adicional command-line options llamado `@argfiles`, pero tu no necesitas estudiar esto para el examen. Ambos el `[options]` y el `[source files]` son partes opcionales del comando, y ambos permiten multiples entradas. El siguiente es un comando legal:

```
javac -help  
javac -classpath com:. -g Foo.java Bar.java
```

* La primera invocación no compila ningún archivo, pero imprime un resumen de las operaciones validas a realizar con el comando `javac`.

* La segunda invocación pasa al compilador dos opciones:

- classpath, que asu vez tiene un argumento de `com:.`
- g, para mostrar información de debugging

y pasa al compilador dos `.java` (`Foo.java` y `Bar.java`). Siempre que tu especifiques multiples opciones y/o archivos estos deben ser separados por espacios.

- **Compiling with `-d`**

Por defecto el compilador pone el archivo `.class` en el mismo directorio de origen del archivo `.java`. Esto esta bien para proyectos pequeños, pero no es lo recomendable. Lo recomendable es que mantengas tus archivos `.java` separados de tus archivos `.class` (Esto ayuda con el control de versiones, testeo,

despliegue...) La opción `-d` le dice al compilador en que directorio colocar el(los) archivo(s) .class generado(s) (`d` es su destino) . Digamos que usted tiene la siguiente estructura de documentos:

```
myProject
|
|--source
|   |
|   |-- MyClass.java
|
|-- classes
|   |
|   |--
```

El siguiente comando, compila `MyClass.java` y lo coloca el `MyClass.class` dentro del directorio `classes`.

```
cd myProject
javac -d classes source/MyClass.java
```

Ahora hechemos un vistazo a como se trabaja con la opción `-d` cuando se utiliza paquetes.

Supongamos que nosotros tenemos el archivo `.java` en la siguiente estructura de directorio:

```
package com.wickedlysmart;
public class MyClass { }

myProject
|
|--source
|   |
|   |--com
|       |
|       |--wickedlysmart
|           |
|           |--MyClass.java
|
|--classes
|   |
|   |--com
|       |
|       |--wickedlysmart
|           |
|           |-- (MyClass.class goes here)
```

Si te encuentras en el directorio `source` , tu deseas compilar `MyClass.java` y colocar el archivo `MyClass.class` dentro del directorio `classes/com/wickedlysmart` debes invocar el siguiente comando:

```
Javac -d ../classes com/wickedlysmart/MyClass.java
```

En este caso el `.class` sera colocado dentro del directorio que encaja justamente con el nombre del paquete (la ruta dentro de la carpeta clasess).

El comando `javac` puede ayudarlo a construir las carpetas que faltan, por ejemplo, si en vez de la estructura anterior hubiesemos tenido esta otra:

```
package com.wickedlysmart;
public class MyClass { }

myProject
|
|--source
|   |
|   |--com
|   |   |
|   |   |--wickedlysmart
|   |   |   |
|   |   |   |--MyClass.java
|
|--classes
|   |
|   |
```

Y el siguiente comando (el mismo que el anterior ejemplo):

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

En este caso, el compilador crea las dos carpetas llamadas `com` y `com/wickedlysmart`.

Lo último sobre `-d` que debes conocer para el examen es que si la carpeta destino que tu especificas no existe, tu obtendrás un error de compilación, si en el ejemplo anterior la carpeta `classes` no existiera, el compilador arrojaría:

```
java:5: error while writing MyClass: classes/MyClass.class (No
such file or directory)
```

173. Launching Applications with java

- El comando `java` es usado para invocar la Java Virtual Machine. En el capítulo 5 se habló acerca del mecanismo de aserciones y cuando tu debías usar los flags como `-ea` o `-da` a la hora de ejecutar una aplicación. Hay muchas otras opciones que tu puedes especificar cuando tu ejecutas el comando `java`, **para el examen tu necesitas entender las opciones `-classpath` (y su equivalente `-cp`) y `-D`, lo cual veremos en las próximas páginas.** La estructura de este comando es :

```
java [options] class [args]
```

El `[options]` y `[args]` son opcionales, y ambos pueden tener múltiples valores. Tu debes especificar exactamente un archivo `class` para ejecutar, y el comando `java` asume que tu le estás diciendo que es un `.class` file, es por eso que tu no especificas la extensión `.class` en la línea de comandos. Aquí un ejemplo:

```
java -DmyProp=myValue MyClass x 1
```

El comando se puede leer como “Crear una propiedad del sistema llamada myProp y se asigna un valor myValue”. Luego abre el archivo llamado MyClass.class y envia estos dos argumentos (String) : x y 1.

- **Using System Properties**

Java 5 tiene un clase llamada java.util.Properties que puede ser usada para acceder a imformación persistente del sistema tales como la versión del actual sistema operativo, del compilador de Java, y de la Java virtual machine. En adición tu puedes proveer información por defecto, tu puedes también añadir y obtener tus propias propiedades. Toma nota de lo siguiente:

```
public class TestProps {  
    public static void main(String[] args) {  
        Properties p = System.getProperties();  
        p.setProperty("myProp", "myValue");  
        p.list(System.out);  
    }  
}
```

Compila e invoca con el siguiente comando:

```
java -DcmdProp=cmdVal TestProps
```

La salida es la siguiente:

```
...  
os.name=Mac OS X  
myProp=myValue  
...  
java.specification.vendor=Sun Microsystems Inc.  
user.language=en  
java.version=1.5.0_02  
...  
cmdProp=cmdVal  
...
```

Como verás la propiedad cmdProp se añadio, al igual que la propiedad myProp.

Si deseas añadir una cadena por linea de comandos, tendrías que hacer:

```
java -DcmdProp="cmdVal take 2" TestProps
```

- **Handling Command-Line Arguments**

Nosotros podemos enviar argumentos a la aplicación por linea de comandos. El siguiente código demuestra como hacerlo:

```
class CmdArgs {  
    public static void main(String[] args) {  
        int x = 0;  
        for (String s : args)  
            System.out.println(x++ + " element = " + s);  
    }  
}
```

Compila y luego invocalo de la siguiente manera:

```
java CmdArgs x 1
```

La salida será:

```
0 element = x  
1 element = 1
```

Estos parámetros son capturados por el método main de la aplicación java. Los modificadores de main() pueden ser alterados un poco, el String array no tienen que llamarse args. Las siguientes son todas las declaraciones legales para main():

```
static public void main(String[] args)  
public static void main(String... x)  
static public void main(String bang_a_gong[])
```

- **Searching for Other Classes**

En la mayoría de los casos, cuando nosotros usemos el comando `java` y comando `javac`, vamos a querer que estos comandos busquen otras clases que sean necesarias para completar la operación. El caso mas obvio es cuando las clases que nosotros creamos usan clases que Sun provee con J2SE(ahora llamado Java SE). Para esto podemos agregar variables de entorno en nuestro sistema, como normalmente se suele hacer. Variables que apunten a la raiz de instalación de java, para que se pueda reconocer clases como `java.util.HashMap` y otros.

¿Pero cuando queremos reconocer otras clases que nosotros mismos hemos hecho?. Para esto podemos especificar un classpath para `java` o `javac` en la línea de comandos.

Nota: Un `classpath` declarado como línea de comandos sobreescribe el `classpath` declarado como variable de entorno, pero esto solo persiste solo lo que dure la invocación.

- **Declaring and Using Classpaths**

Classpaths consisten de un número de variable de ubicaciones de directorio, separados por delimitadores. Para sistemas basados en Unix, los slashes(/) son usados para construir ubicaciones de directorio, y el separador es el dos puntos (:).

Por ejemplo:

```
-classpath /com/foo/acct:/com/foo
```

- Especifica dos directorios en donde las clases pueden ser halladas: /com/foo/acct y /com/foo/. En ambos casos, estos directorios son absolutamente vinculadas a la raiz del sistema. Es importante recordar que cuando tu especificas un subdirectorio, tu no estas especificando los directorios que están por encima de estos. **Por eso en el ejemplo anterior el directorio /com no será buscado(OJO CON ESTO).**

exam watch

Most of the path-related questions on the exam will use Unix conventions. If you are a Windows user, your directories will be declared using backslashes (\) and the separator character you use will be a semicolon (;). But again, you will NOT need any shell-specific knowledge for the exam.

- Una situación común que sucede cuando invocamos `java` o `javac` es que no busca los archivos class en el directorio actual. Tu debes indicarle que lo haga de la siguiente manera:

```
-classpath /com/foo/acct:/com/foo:..
```

El punto (.) indica que considere también el directorio actual.

Recuerda los siguientes:::

El classpaths busca de izquierda a derecha, por lo tanto no es lo mismo decir :

```
-classpath /com:/foo:..
```

Que decir:

```
-classpath .:/foo:/com
```

Finalmente el comando `java` permite la abreviatura `-classpath` como `-cp`. La documentación es inconsistente acerca de que el comando `javac` permita la abreviación `-cp`. En la mayoría de maquinas lo hace, pero no es garantizado.

- **Packages and Searching**

Cuando tu comienzas a poner las clases en paquetes, y luego empiezas a usar **classpath**s para encontrar estas clases, las cosas pueden ponerse difíciles. Los creadores del examen sabían esto, y ellos intentaron crear una serie de preguntas diabólicas sobre paquetes / classpath con las que te puedes confundir. Vamos a empezar por la revisión de los paquetes. En el siguiente código:

```
package com.foo;
public class MyClass { public void hi() { } }
```

Nosotros decimos que `MyClass` es un miembro de el paquete `com.foo`. Esto significa que el nombre completo de la clase es ahora `com.foo.MyClass`. Cada vez que una clase esté dentro de un paquete, el paquete forma parte de su nombre completo, se vuelve **atómico**. Esto nunca puede ser dividido.

Ahora veremos como podemos usar `com.foo.MyClass` en otra clase:

```
package com.foo;
public class MyClass { public void hi() { } }
```

y en otro archivo:

```
import com.foo.MyClass; // either import will work
import com.foo.*;
public class Another {
    void go() {
        MyClass m1 = new MyClass(); // alias name
        com.foo.MyClass m2 = new com.foo.MyClass(); // full name
        m1.hi();
        m2.hi();
    }
}
```

- **Relative and Absolute Paths**

Un classpath es una colección de uno o mas paths. Cada ruta en un classpath es una ruta absoluta o ruta relativa. Una ruta absoluta en Unix empieza con el slash(/) (En window sería algo así como c:\). Una ruta relativa es aquel que no empieza con un slash.

</OBJECTIVE 7.2: Using the javac and java Commands>

<OBJECTIVE 7.5: JAR Files>

<SUMMARY>

7.5 Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully.

</SUMMARY>

174. JAR Files and Searching

Una vez que hayas construido y probado tu aplicación, es posible que desees enpaquetar tu aplicación de modo que sea fácil de distribuir e instalar. Uno de los mecanismos que Java proporciona para estos fines es un archivo JAR. JAR significa Java Archive. Y se utiliza para comprimir los datos (similares a los archivos en formato ZIP) y para almacenar los datos. Digamos que tienes una aplicación que utiliza diferentes clases que se encuentran en varios paquetes. Aquí hay un árbol de directorios.

```
test
  |
  |--UseStuff.java
  |--ws
    |
    |--(create MyJar.jar here)
    |--myApp
      |
      |--utils
        |
        |--Dates.class      (package myApp.utils; )
        |--Conversions.class      "      "
      |
      |--engine
        |
        |--rete.class      (package myApp.engine; )
        |--minmax.class      "      "
```

Tu puedes crear un solo JAR que contenga todas los archivos class que están en myApp, y también que el directorio myApp mantenga su estructura. Cada uno de estos JAR pueden ser trasladados de lugar a lugar, y de máquina a máquina, y todas estas clases dentro del JAR pueden ser accedidas vía classpath, y usadas por `java` y `javac`. Todo esto ocurre sin necesidad de estar descomprimiendo el JAR. Aunque tu no necesitas conocer como hacer un JAR para el examen, vamos hacerlo, para ir al directorio actual ws, y luego hacer un JAR llamado MyJar.jar, ejecutaremos la siguiente línea de comando:

```
cd ws
jar -cf MyJar.jar myApp
```

El comando `jar` creará el archivo JAR llamado MyJar.jar y almacenará todo lo que contiene el directorio `myApp` (carpetas y archivos). Tu puedes ver lo que contiene el archivo JAR, con el siguiente comando:

```
jar -tf MyJar.jar
```

Mostrará algo como lo siguiente:

```
META-INF/
META-INF/MANIFEST.MF
```

```
myApp/
myApp/.DS_Store
myApp/utils/
myApp/utils/Dates.class
myApp/utils/Conversions.class
myApp/engine/
myApp/engine/rete.class
myApp/engine/minmax.class
```

Muy bien ahora lo que se necesita saber para el examen. Para buscar un archivo JAR usando el classpath es similar a buscar un archivo de paquete en un classpath. La diferencia es que cuando tu especificas un path para un archivo JAR, tu necesitas incluir el nombre del archivo JAR al final del path.

Si por ejemplo decimos que queremos compilar `UseStuff.java` en el directorio test, y `UseStuff.java` necesita acceder a una clase contenida en `myApp.jar`. Al compilar `UseStuff.java` tu debes de decir:

```
cd test
javac -classpath ws/myApp.jar UseStuff.java
```

Compare el uso del archivo JAR usando un class en un paquete. Si `UseStuff.java` necesita usar clases en el paquete `myApp.utils` y la clase no esta en un JAR, tu deberás decir:

```
cd test
javac -classpath ws UseStuff.java
```

e x a m

watch

When you use an import statement you are declaring only one package. When you say `import java.util.;` you are saying "Use the short name for all of the classes in the `java.util` package." You're NOT getting the `java.util.jar` classes or `java.util.regex` packages! Those packages are totally independent of each other; the only thing they share is the same "root" directory, but they are not the same packages. As a corollary, you can't say `import java.*;` in the hopes of importing multiple packages —just remember, an import statement can import only a single package.*

- **Using .../jre/lib/ext with JAR files**

Cuando tu instalas Java, este viene con un conjunto de directorios y archivos, incluyendo los archivos JAR que vienen con las classes que trae el estandar J2SE. `java` y `javac` tienen una lista de lugares donde ellos pueden acceder cuando buscan archivos class. Dentro de su directorio, Java tiene un subdirectorio llamado `jre/lib/ext`. Si tu poner el archivo JAR dentro del subdirectorio `ext`, `java` y `javac` lo reconocerá y podrá usar sus clases que contiene. Tu no tiene que mencionar estos subdirectorios en un classpath.

Sun recomienda, hacer esto solo cuando estás haciendo pruebas de desarrollo, pero no cuando quieras distribuir el software.

e x a m

Watch

It's possible to create environment variables that provide an alias for long classpaths. The classpath for some of the JAR files in J2SE can be quite long, and so it's common for such an alias to be used when defining a classpath. If you see something like JAVA_HOME or \$JAVA_HOME in an exam question it just means "That part of the absolute classpath up to the directories we're specifying explicitly." You can assume that the JAVA_HOME literal means this, and is pre-pended to the partial classpath you see.

</OBJECTIVE 7.5: JAR Files>

<OBJECTIVE 7.1: Using Static Imports>

<SUMMARY>

7.1 Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.

</SUMMARY>

175. Static Imports

A partir de java 5, la sentencia `import` ha mejorado sobre el tema de importaciones de elementos estáticos. Esto se conoce como **static imports**. Static import se utiliza para cuando deseas utilizar un miembro estático de una clase. Usted puede usar este tipo de importaciones para las clases de la API y las suyas. Aquí un “antes y después de su uso”:

Antes del uso de static imports:

```
class TestStatic {
    public static void main(String[] args) {
        System.out.println(Integer.MAX_VALUE);
        System.out.println(Integer.toHexString(42));
    }
}
```

Después del uso de static imports:

```
import static java.lang.System.out; // 1
import static java.lang.Integer.*; // 2
class TestStaticImport {
    public static void main(String[] args) {
        out.println(MAX_VALUE); // 3
        out.println(toHexString(42)); // 4
    }
}
```

</OBJECTIVE 7.1: Using Static Imports >

Quizz del capítulo 10(Development)

1)

A class games.cards.Poker is correctly defined in the jar file Poker.jar. A user wants to execute the main method of Poker on a UNIX system using the command:

Java games.cards.Poker

What allows the user to do this?

- A- put Poker.jar in directory /stuff/java, and set the CLASSPATH to include /stuff/java
- B- put Poker.jar in directory /stuff/java, and set the CLASSPATH to include /stuff/java/*.jar
- C- put Poker.jar in directory /stuff/java, and set the CLASSPATH to include /stuff/java/Poker.jar
- D- put Poker.jar in directory /stuff/java/games/cards, and set the CLASSPATH to include /stuff/java
- E- put Poker.jar in directory /stuff/java/games/cards, and set the CLASSPATH to include /stuff/java/*.jar
- F- put Poker.jar in directory /stuff/java/games/cards, and set the CLASSPATH to include /stuff/java/Poker.jar

2)

A UNIX user named Bob wants to replace his chess program with a new one, but he is not sure where the old one is installed. Bob is currently able to run a Java chess program starting from his home directory /home/bob using the command:

Java -classpath /test:/home/bob/downloads/* jar games.Chess

Bob's CLASSPATH is set (at login time) to:

/usr/lib:/home/bob/classes:/opt/java/lib:/opt/java/lib/*.jar

What is a possible location for the Chess.class file?

- A- /test/Chess.class
- B- /home/bob/Chess.class
- C- /test/games/Chess.class
- D- /usr/lib/games/Chess.class
- E- /home/bob/games/Chess.class
- F- Inside jarfile /opt/java/lib/Games.jar (with a correct manifest)
- G- Inside jarfile /home/bob/downloads/Games.jar (with a correct manifest)

</OBJECTIVES>

</CERTIFICATION OBJETIVES>

</CAP10: Development>