Java8

Resumen

Java 8 :: Resumen

Resumen

Variables

- Las variables de nivel de clase siempre están inicializadas a su valor `default`.
- Las variables de nivel de método no están inicializadas.
- Los identificadores deben comenzar con una letra, un carácter de moneda ($) o un carácter de conexión como el guión bajo (_). ¡Los identificadores no pueden comenzar con un dígito!

Resumen

keywords

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

Resumen

Identificadores

- Los identificadores deben comenzar con
  - una letra,
  - un carácter de moneda ($)
  - un carácter de conexión como el guión bajo (_).
- Los identificadores no pueden comenzar con un dígito.
- Son case-sensitive

Ilegales

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

Legales

```
int _a;
int $c;
int ______2_w;
int _$;
int this_is_a_very_detailed_name_for_an_identifier;
```

Resumen

Imports

- El único trabajo de una sentencia `import` es ahorrar escritura.
- Puede utilizar un asterisco (*) para buscar en el contenido de un solo paquete.
- "`static import`", se aplica para es de importación estática de identificadores y métodos.
- Puede importar clases de API y / o clases personalizadas.

Resumen

Ejecutable

- Puede compilar y ejecutar programas Java utilizando los programas de línea de comandos `javac` y `java`, respectivamente. Ambos programas soportan una variedad de opciones de línea de comandos.
- Las únicas versiones de métodos main () con poderes especiales son aquellas versiones con método firmas equivalentes a `public static void main (String [] args)`.
- `main()` puede estar sobrecargado.

Resumen

Source File Declaration Rules (I)

- Las declaraciones de importación (si las hay) deben venir después del paquete y antes de la declaración de clase.
- Si no hay ninguna instrucción package, las sentencias import deben ser las primeras (noncomment) declaraciones en el archivo de origen.
- Las instrucciones package e import se aplican a todas las clases del archivo.
- Un archivo puede tener más de una clase no pública.
- Los archivos sin clases public no tienen restricciones de nombres.

Resumen

Source File Declaration Rules (II)

- Las declaraciones de importación (si las hay) deben venir después del paquete y antes de la declaración de clase. Si no hay ninguna instrucción package, las sentencias import deben ser las primeras (noncomment) declaraciones en el archivo de origen.
- Las instrucciones package e import se aplican a todas las clases del archivo.
- Un archivo puede tener más de una clase no pública.
- Los archivos sin clases public no tienen restricciones de nombres.

Resumen

Class Access Modifiers

- Hay tres modificadores de acceso: `public`, `protected` y `private`.
- Existen 4 tipos de niveles de acceso: `public`, `protected`, `default` y `private`.
- Las clases únicamente pueden tener acceso `public` o `default`.
- Una clase con acceso `default` sólo puede ser vista por clases dentro del mismo `package`.
- Una clase con acceso `public` puede ser vista por todas las clases de todos los paquetes.
- La visibilidad de la clase gira en torno a si el código de una clase puede
  - Crear una instancia de otra clase
  - Extender (o subclase) otra clase
  - Métodos de acceso y variables de otra clase.

Resumen

Class Modifiers (Nonaccess)

- Las clases también pueden ser modificadas con `final`, `abstract`, o `strictfp`. Una clase no puede ser final y abstracta.
- Una clase final no puede ser sub clasificada (heredada).
- Una clase `abstract` no se puede instanciar.
- Un solo método `abstract` en una clase, obliga a que toda la clase debe ser `abstract`.
- Una clase `abstract` puede tener tanto métodos abstractos como no abstractos.
- La primera clase concreta para extender una clase abstracta debe implementar todos sus métodos abstractos.

Resumen

Interface Implementation (I)

- Las interfaces son contratos para lo que una clase puede hacer, pero no dicen nada sobre la forma en que la clase debe hacerlo.
- Las interfaces pueden ser implementados por cualquier clase desde cualquier árbol de herencia.
- Interface methods are by default public and abstract—explicit declaration of these modifiers is optional.
- Una interfaz es como una clase abstracta de 100 por ciento y es implícitamente abstracta si escribe el modificador abstracto en la declaración o no.
- Una interfaz sólo puede tener métodos abstractos, no se permiten métodos concretos.
- Los métodos de interfaz son por defecto público y la declaración abstracta-explícita de estos modificadores es opcional.

Resumen

Interface Implementation (II)

- Las interfaces pueden tener constantes, que siempre son implícitamente `public`, `static` y `final`.
- Las declaraciones constantes de interfaces `public`, `static` y `final` son opcionales en cualquier combinación.
- Una clase de implementación legal no abstracta tiene las siguientes propiedades:
  - Proporciona implementaciones concretas para los métodos de la interfaz.
  - Debe seguir todas las reglas legales de anulación para los métodos que implementa.
  - No debe declarar ninguna nueva excepción comprobada para un método de implementación.
  - No debe declarar excepciones comprobadas que sean más amplias que las excepciones declaradas en el método de interfaz.
  - Puede declarar excepciones de tiempo de ejecución en cualquier implementación de método de interfaz independientemente de la declaración de interfaz.

Resumen

Interface Implementation (II)

- Una clase de implementación legal no abstracta tiene las siguientes propiedades (cont):
  - Debe mantener la firma exacta (que permite los retornos covariantes) y el tipo de retorno de los métodos que implementa (pero no tiene que declarar las excepciones de la interfaz).

```
public interface IFace1 {
        void method1() throws Exception;
        void method2() throws Exception;
        void method3() throws MyException;
        void method4() throws MyException;
        void method5() throws MyException;
}
```

```
public class MyException extends Exception {}

public class MySubException extends MyException {}
```

```
public class Class1 implements IFace1 {
        @Override  public void method1()  {}   <- Ok
        @Override  public void method2() throws MyException { } <- Ok
        @Override  public void method3() throws Exception { }  <- Error
        @Override  public void method4() throws MyException, Exception { }  <- Error
        @Override  public void method5() throws MyException, MySubException { }  <- Ok
}
```

Resumen

Interface Implementation (III)

- Una clase que implementa una interfaz puede ser abstract.
- Una clase de implementación abstracta no tiene que implementar los métodos de interfaz (pero la primera subclase concreta debe).
- Una clase puede extender sólo una clase (no hay herencia múltiple), pero puede implementar muchas interfaces.
- Las interfaces pueden extender una o más interfaces.
- Las interfaces no pueden extender una clase o implementar una clase o interfaz.
- Al realizar el examen, verifique que las declaraciones de interfaz y clase sean legales antes de verificar otra lógica de código.

Resumen

Access Levels

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Resumen

Member Access Modifiers (I)

- Los métodos y variables de instancia (no locales) se conocen como "miembros" (member).
- Los miembros pueden utilizar los cuatro niveles de acceso: `public`, `protected`, `default`, y `private`. El acceso a los “miembros” tiene dos formas:
    - El código de una clase puede acceder a un miembro de otra clase.
    - Una subclase puede heredar un miembro de su superclase.
- Si no se puede acceder a una clase, no se puede acceder a sus miembros.
- Determine la visibilidad de la clase antes de determinar la visibilidad del miembro.
- Los miembros públicos pueden ser accedidos por todas las otras clases, incluso en otros paquetes.
- Si un miembro de superclase es público, la subclase lo hereda, independientemente del paquete.

Resumen

Member Access Modifiers (II)

- Los miembros a los que se accede sin el operador punto (.) Deben pertenecer a la misma clase. (this). Siempre hace referencia al objeto que se está ejecutando actualmente.
    - `this.aMethod()` es lo mismo que invocar `aMethod()`.
- Se puede acceder a miembros privados sólo por código en la misma clase.
- Los miembros privados no son visibles para las subclases, por lo que los miembros privados no pueden ser heredados.

Resumen

Member Access Modifiers (III)

- Los miembros default y protected difieren sólo cuando están involucradas subclases::
  - Los miembros `default` solo pueden ser accedidos por clases del mismo `package`.
  - Los miembros `protected` pueden ser accedidos por otras clases en el mismo paquete, más subclases independientemente del paquete.
  - protected = package + kids (kids = subclasses).
  - Para las subclases fuera del paquete, el miembro protegido sólo se puede acceder mediante herencia; Una subclase fuera del paquete no puede tener acceso a un miembro protegido utilizando una referencia a una instancia de superclase. (En otras palabras, la herencia es el único mecanismo para que una subclase fuera del paquete tenga acceso a un miembro protegido de su superclase).
  - Un miembro protegido heredado por una subclase de otro paquete no es accesible a ninguna otra clase en el paquete de subclase, excepto para las subclases propias de la subclase.

Resumen

Class Access Modifiers

- Hay tres modificadores de acceso: `public`, `protected` y `private`.
- Existen 4 tipos de niveles de acceso: `public`, `protected`, `default` y `private`.
- Las clases únicamente pueden tener acceso `public` o `default`.
- Una clase con acceso `default` sólo puede ser vista por clases dentro del mismo `package`.
- Una clase con acceso `public` puede ser vista por todas las clases de todos los paquetes.
- La visibilidad de la clase gira en torno a si el código de una clase puede
  - Crear una instancia de otra clase
  - Extender (o subclase) otra clase
  - Métodos de acceso y variables de otra clase.

Resumen

Other Modifiers (I)

- Los métodos `final` methods no pueden ser sobrescritas en las subclases.
- Los métodos `abstract` son declaradas con la firma, tipo de retorno, y opcionalmente la sentencia `throws`, pero nunca implementados.
- Los métodos `abstract` se cierran con “;”, no con llaves. Tres maneras de detectar un método no abstracto:
  - El método no está marcado como `abstract`.
  - El método tiene llaves.
  - El método PUEDE tener código entre llaves.
- La primera clase no abstracta (concreta) para extender una clase abstracta debe implementar todos los métodos abstractos de la clase abstracta.

Resumen

Other Modifiers (II)

- El modificador `synchronized` sólo se aplica a los métodos y bloques de código.
- Los métodos `synchronized` pueden tener cualquier control de acceso y también pueden ser marcados como finales.
- Los métodos `abstract` deben ser implementados por una subclase, por lo que deben ser heredables. Por esta razón:
  - los métodos `abstract` no pueden ser private.
  - los métodos `abstract` no pueden ser final.
  - el modificador native se aplica solo a métodos.
  - el modificador strictfp se aplica solo a métodos y clases.

Resumen

Methods with var-args

- A partir de Java 5, los métodos pueden declarar un parámetro que acepta de cero a muchos argumentos, el llamado método var-arg.
- Un parámetro var-arg parameter es declarado con la sintaxis `type... name`; por ejemplo

```
void doStuff(int... x) { }
```

- Los métodos var-arg solo pueden tener un único parámetro de este tipo.
- Los métodos que combinan parámetros normales con var-arg, este debe ser el último.

Resumen

- Las variables de instancia pueden
  - Tener control de acceso
  - Marcadas como `final` o `transient`
- Las variables de instancia no pueden ser `abstract`, `synchronized`, `native`, o `strictfp`.
- Es legal declarar una variable local con el mismo nombre que una variable de instancia; se llama
  “shadowing.”

Resumen

Variable Declarations (II)

- Las variables `final` poseen las siguientes propiedades:
  - Las variables `final` no se pueden reasignar una vez asignado un valor.
  - Las variables de referencia `final` no pueden referirse a un objeto diferente una vez que el objeto ha sido asignado a la variable final.
  - Las variables de `final` deben ser inicializado antes de que el constructor se complete.
- No existe tal cosa como un objeto `final`. Una referencia de objeto marcada como `final` NO significa que el objeto en sí no pueda cambiar.
- El modificador `transient` sólo se aplica a las variables de instancia.
- El modificador `volatile ("being stored in main memory and not from the CPU cache.")` sólo se aplica a las variables de instancia.

Resumen

Array Declarations

- Los arrays pueden contener primitivas u objetos, pero la propia matriz es siempre un objeto.
- Cuando se declara un array, los corchetes pueden estar a la izquierda oa la derecha del nombre de la variable.
- Nunca es legal incluir el tamaño de una matriz en la declaración.

Resumen

Static Variables and Methods

- No están vinculados a ningún caso particular de una clase.
- No se necesitan instancias de clase para usar miembros estáticos de la clase.
- Existe una sola copia de las variables/clases `static` y todas las instancias la comparten.
- Métodos `static` no tienen acceso directo a miembros no `static`.

Resumen

enums (I)

- Un `enum` especifica una lista de valores constantes asignados a un tipo.
- Un `enum` NO ES un `String` o un `int`; una constante `enum` es un de un tipo `enum`. Por ejemplo, SUMMER y FALL son tipos `enum` de Season.
- Un `enum` puede ser declarado dentro o fuera de una clase, pero NUNCA en un método.
- Un `enum` declarado fuera de una clase, NO puede ser marcada como `static`, `final`, `abstract`, `protected` o `private`.
- `Los enum` pueden contener, constructores, métodos, variables, y cuerpos de clase constantes-específicos.
- `enum` constants can send arguments to the enum constructor, using the syntax BIG(8), where the int literal 8 is passed to the enum constructor.`s values.`

Resumen

enums (II)

- Los constructores de las `enum` pueden tener argumentos y pueden ser sobrecargados.
- Los constructores de las `enum` NUNCA pueden ser invocado directamente desde el código. Son llamadas siempre de manera automática cuando `enum` es inicializado.
- El punto y coma al final de la declaración de un `enum` es opcional:
  - `enum Foo { ONE, TWO, THREE} enum Foo { ONE, TWO, THREE};`
  - `MyEnum.values() returns an array of MyEnum's values.`

Resumen

Encapsulation, IS-A, HAS-A

- El encapsulado ayuda a ocultar la implementación detrás de una interfaz (API).
- El código encapsulado tiene dos características:
  - Las variables de instancia se mantienen protegidas (usualmente con el modificador `private`).
  - Los métodos getter y setter, proveen acceso a la instancia de las variables.
- IS-A es indicativo de herencia (`extends`) o implementación (`implements`).
- IS-A, “hereda de,” y “es subtipo de”.
- HAS-A nos indicativo una instancia de una clase "tiene una" referencia a una instancia de otra clase u otra instancia de la misma clase.

Resumen

Inheritance

- La herencia permite que una clase sea una subclase de una superclase y por lo tanto hereda variables protegidas y métodos de la superclase.
- La herencia es un concepto clave que subyace al IS-A, el polimorfismo, la sobrecarga, la sobrecarga y el “casting”.
- Todas las clases (excepto la clase Object) son subclases de tipo Object, y por lo tanto heredan los métodos de “Object”

Resumen

Polymorphism

- Polimorfismo significa "muchas formas".
- Una variable de referencia es siempre de un tipo único, no cambiable, pero puede referirse a un objeto de subtipo.
- Un objeto único puede ser referido por variables de referencia de muchos tipos diferentes, siempre y cuando sean del mismo tipo o un supertipo del objeto.
- El tipo de la variable de referencia (no el tipo del objeto) determina qué métodos se pueden llamar.
- Las invocaciones de métodos polimórficos se aplican sólo a métodos de instancia sobrescritos

Resumen

Overriding and Overloading (I)

- Los métodos pueden ser sobreescritos o sobrecargados; los constructores pueden estar sobrecargados pero no anulados. Con respecto al método que sobreescribe :
    - Debe tener la misma lista de argumentos
    - Debe tener el mismo tipo de retorno, excepto que, a partir de Java 5, el tipo de retorno puede ser una subclase, y esto se conoce como “Covariant return”
    - No debe tener un modificador de acceso más restrictivo
    - Puede tener un modificador de acceso menos restrictivo
    - No debe lanzar excepciones verificadas nuevas o ampliadas
    - Puede lanzar menos y menos ampliadas o cualquier excepción no verificada
- Los métodos final no pueden ser sobrescritos

Resumen

Overriding and Overloading (II)

- Sólo los métodos heredados pueden ser sobrescritos. Los métodos privados no son heredados.
- Una subclase utiliza s `super.overriddenMethodName()` para llamará al version del metodo de la superclase que a sido sobrescrita
- Sobrecarga significa reutilizar un nombre de método pero con argumentos diferentes.
- Métodos sobrecargados
    - Debe tener listas de argumentos diferentes
    - Puede tener diferentes tipos de devolución, si las listas de argumentos también son diferentes
    - Puede tener diferentes modificadores de acceso
    - Puede lanzar diferentes excepciones

Resumen

Overriding and Overloading (III)

- Los métodos de una superclase pueden sobrecargarse en una subclase.
- El polimorfismo se aplica a la sobrescritura, no a la sobrecarga.
- Tipo de objeto (no el tipo de la variable de referencia) determina qué método anulado se utiliza en tiempo de ejecución.
- El tipo de referencia determina qué método sobrecargado se utilizará durante la compilación.

Resumen

Reference Variable Casting

- Existen 2 tipos de casting de variables de referencia:
  - **Downcasting :** Si tiene una variable de referencia que hace referencia a un objeto de subtipo, se puede asignar a una variable de referencia del subtipo. Se debe hacer una conversión explícita, y el resultado es que puede acceder a los miembros del subtipo con esta nueva variable de referencia.

```
Object o = “a string”;
String s = (String) o;
```

  - **Upcasting :** asignar una variable de referencia a una variable de referencia de supertipo explícita o implícitamente. Esto es una operación inherentemente segura porque la asignación restringe las capacidades de acceso de la nueva variable.

```
Object o = new String(“a string”);
```

Resumen

Implementing an Interface

- Cuando implementamos una interfaz, está cumpliendo su contrato.

- Se implementa de manera correcta una interfaz mediante la implementación adecuada y concreta de todos los métodos definidos.
- Una sola clase puede implementar muchas interfaces.

Resumen

Return Types

- Los métodos sobrecargados pueden cambiar los tipos de retorno; Los métodos sobrescritos no pueden, excepto en el caso de los “covariant returns“.
- Los tipos de referencia de referencia de objeto pueden aceptar valores nulos como valores devueltos.
- Una matriz es un tipo de devolución legal, tanto para declarar como para devolver como un valor.
- Para los métodos con tipos de retorno primitivos, cualquier valor que se puede convertir implícitamente en el tipo de retorno se puede devolver.
- Los métodos con objetos reren
- Los métodos con un tipo de retorno de referencia de objeto pueden devolver un subtipo. Los métodos con un tipo de retorno de interfaz pueden devolver cualquier implementador.

Resumen

Constructors and Instantiation (I)

- El constructor es siempre invocado cuando un nuevo objeto se crea.
- Cada superclase en el árbol de herencia de un objeto tendrá un constructor llamado.
- Cada clase, incluso las abstractas, tienen al menos un constructor.
- Los constructores deben tener el mismo nombre que la clase.
- Los constructores no tienen retorno.
- La típica ejecución del constructor se produce de la siguiente manera:
    - El constructor llama a su constructor de superclase, que llama a su constructor de superclase, y así sucesivamente todo hasta el constructor de Object.
    - El constructor de Object ejecuta y luego regresa al constructor de llamada, que se ejecuta hasta completar y luego regresa a su constructor de llamada, y así sucesivamente hasta la finalización del constructor de la instancia real que se está creando.

Resumen

Constructors and Instantiation (II)

- Los constructores pueden tener cualquier modificador de acceso.
- El compilador creará un constructor predeterminado si no crea ningún constructor en su clase..
- El constructor `default` es un constructor no-arg con una llamada no-arg a `super()`.
- La primera declaración de cada constructor debe ser una llamada a `this()` (un constructor sobrecargado) o a `super()`.
- El compilador agrega una llamada a `super()` a menos que ya haya realizado una llamada a `this()` o `super()`.
- Los miembros de instancia sólo son accesibles después de ejecutarse el super constructor.
- Las clases `abstract` tienen constructores que se llaman cuando se crea una instancia de una subclase concreta.
- Las interfaces no tienen constructores.

Resumen

Constructors and Instantiation (III)

- Si su superclase no tiene un constructor vacío, debe crear un constructor e insertar una llamada a super () con argumentos que coinciden con los del constructor de la superclase
- Los constructores nunca se heredan, por lo tanto no se sobreescriben.
- Un constructor puede ser invocado de manera directa por otro constructor via `super()` y `this()`;
- Asuntos a tener encuentra con la llamada `this()`:
    - Debe aparecer como primera sentencia dentro de un constructor.
    - Las lista de argumentos determinar a que constructor sobrecargado se llama.
    - Los constructores pueden llamar a constructores, y así sucesivamente, pero tarde o temprano uno de ellos mejor llamará `super()` o la pila explotará (Stackoverflow.
    - Las llamadas a `super()` y `this()` no pueden estar en el mismo constructor. Usted puede tener uno o el otro, pero nunca ambos.

Resumen

Initialization Blocks

- Use `static init` blocks—`static { /* code here */ }`— para el código que desea ejecutar una vez, cuando se carga la clase por primera vez. Múltiples bloques se ejecutan desde arriba hacia abajo.
- Use normal `init` blocks—`{ /* code here }`— para el código que desea que se ejecute para cada nueva instancia, justo después de todos los super constructores han ejecutado. Una vez más, varios bloques se ejecutan desde la parte superior de la clase hacia abajo.

Resumen

Stack and Heap

- Las variables locales (las variables de métodos) viven / se alojan en el stack.
- Los objetos y sus variables de instancia viven / se alojan en el heap.

Resumen

Statics

- Use métodos estáticos cuando debe implementar comportamientos que no deba ser afectado por el estado de las instancias.
- Use variables estáticos para almacenar datos que son específicos de clase, no de instancia. Solo habr una copia de esta variable.
- Todos los miembros estáticos pertenecen a una clase no a su instancia.
- Un método estático no puede acceder directamente a una variable de instancia.
- Utilice el operador “.” para acceder a los miembros estáticos, pero recuerde que el uso de una variable de referencia con el operador “.” es realmente un truco de sintaxis y el compilador sustituirá el nombre de la clase por la variable de referencia; por ejemplo:

  `d.doStuff();` será `Dog.doStuff();`

- métodos estáticos no pueden ser sobreescritos pero pueden ser redefinidos.

Resumen

Literals and Primitive Casting

- Los literales `int` pueden ser binarios, decimales, octales (013), o hexadecimales (0x3d).
- Los literales para `long`s finalizan en L or l.
- Los literales `float` finalizan en F o f
- Los literales `double` literals finalizan en digito, D o d.
- Los literales `boolean` son true y false.
- Los literales `char`, son caracteres simples dentro de comillas simples: ‘d’.

Resumen

Handling Exceptions (I)

- Las excepciones se agrupan en dos tipos: **checked** y **unchecked**.
  Las **checked** incluyen todo las heredadas de Exception, excluyendo las que lo hacen de RuntimeException.
- Las excepciones de tipo **checked** están sujetas a la regla de “declarar y gestionamos”; declaramos mediante **throws** y gestionamos con el uso del **try/catch**.
- Los subtipos de **Error** o **RuntimeException** son **unchecked**. Si bien se pueden declarar o gestionar, aunque no es requerido.
- Si se utiliza, de manera opcional, el bloque finally, hay que tener en cuenta que siempre se invocará, independientemente de que se lance o no una excepción en el intento correspondiente, e independientemente de si se captura o no una excepción lanzada.

Resumen

- La única excepción a la regla finally-siempre-será-invocado es que cuando la JVM se cierra. Esto podría ocurrir si el código de los bloques `try` o `catch` llama a `System.exit()`.
- Sólo porque finalmente se invoca no significa que se complete. Código en el bloque finalmente podría plantear una excepción o emitir un System.exit ().
- Las excepciones no captadas se propagan de nuevo a través de la pila de llamadas, comenzando desde el método donde se genera la excepción y terminando con el primer método que tiene una captura correspondiente para ese tipo de excepción o un cierre de JVM.

Resumen

Handling Exceptions (III)

- Puede crear sus propias excepciones, normalmente extendiendo `Exception` o uno de sus subtipos. Su excepción se considerará una excepción **checked** (a menos que se extienda desde `RuntimeException`) y el compilador aplicará la regla **declarar y gestionar** para esa excepción.
- Todos los bloques de captura deben ser ordenados de la forma más específica a la más general. Si tiene una cláusula catch para `IOException` y `Exception`, debe poner el catch para `IOException` primero en su código. De lo contrario, el `IOException` sería capturado por `catch(Exception e)`, porque un argumento `catch` puede capturar la excepción especificada o cualquiera de sus subtipos. El compilador le impedirá definir cláusulas de captura que nunca se puedan alcanzar.
- Algunas excepciones son creadas por programadores, y algunas por la JVM.

Resumen

Handling Exceptions (IV)

Exception	Description	Typically Thrown
`ArrayIndexOutOfBoundsException`	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`	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
`IllegalArgumentException`	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
`IllegalStateException`	Thrown when the state of the environment doesn't match the operation being attempted—for example, using a scanner that's been closed.	Programmatically
`NullPointerException`	Thrown when attempting to invoke a method on, or access a property from, a reference variable whose current value is `null`.	By the JVM
`NumberFormatException`	Thrown when a method that converts a `String` to a number receives a `String` that it cannot convert.	Programmatically
`AssertionError`	Thrown when an `assert` statement's `boolean` test returns `false`.	Programmatically
`ExceptionInInitializerError`	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
`StackOverflowError`	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
`NoClassDefFoundError`	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing `.class` file.	By the JVM

Resumen

Scope

- El ámbito se refiere a la vida útil de una variable.
- Hay cuatro ámbitos básicos:
  - Las variables estáticas viven básicamente mientras su clase vive.
  - Las variables de instancia viven mientras su objeto vive.
  - Las variables locales viven mientras su método esté en la pila; sin embargo, si su método invoca otro método, están temporalmente no disponibles.
  - Bloquear variables (por ejemplo, en un for o un if) en vivo hasta que el bloque complete

Resumen

Passing Variables into Methods

- Los métodos pueden tomar primitivas y / o referencias de objetos como argumentos.
- Los argumentos de método son siempre copias.
- Los argumentos de método nunca son objetos reales (pueden ser referencias a objetos).
- Un argumento primitivo es una copia no unida (unattached) del primitivo original.
- Un argumento de referencia es otra copia de una referencia al objeto original.
- El shadowing ocurre cuando dos variables con diferentes alcances comparten el mismo nombre. Esto lleva a problemas difíciles de encontrar y difíciles de responder preguntas del examen.

Resumen

Basic Assignments (I)

- Los enteros literales son implícitamente int .
- Las expresiones enteras siempre dan como resultado un int-size, nunca menor.
- Los numeros Floating-point son implícitamente doubles (64 bits).
- Al estrechar (narrowing) una primitiva se trunca los bits de orden alto.
- Las asignaciones compuestas (como + =) realizan un cast automático.
- Una variable de referencia contiene los bits que se utilizan para referirse a un objeto.

Resumen

- Las variables de referencia pueden referirse a subclases del tipo declarado pero no a las superclases.
- Cuando crea un nuevo objeto, como `MyClass b = new MyClass () ;`, la JVM hace tres cosas:
  - Hace que una variable de referencia b, de tipo `MyClass`.
  - Crea un nuevo objeto `MyClass`.
  - Asigna el objeto `MyClass` a la variable de referencia b.

Resumen

Using a Variable or Array Element That Is Uninitialized and Unassigned

- Cuando se instancia un array de objetos, los objetos dentro del array no se instancian automáticamente, pero todas las referencias obtienen el valor predeterminado de null.
- Cuando se instancia un array de primitivas, los elementos obtienen valores predeterminados.
- Las variables de instancia siempre se inicializan con un valor predeterminado.
- Las variables locales / automáticas / método nunca reciben un valor predeterminado. Si intenta utilizar uno antes de inicializarlo, obtendrá un error de compilador.

Resumen

Garbage Collection (I)

- En Java, el garbage collection (GC) provee la gestión automática de memoria. El propósito de GC es eliminar los objetos que no se pueden alcanzar.
- Sólo la JVM decide cuándo ejecutar el GC; Sólo puede sugerirlo.
- No se puede saber el algoritmo del GC con seguridad.
- Los objetos deben ser considerados elegibles antes de que puedan ser recogidos basura.
- Un objeto es elegible cuando ningún hilo en directo puede alcanzarlo.
- Para llegar a un objeto, debe tener una referencia real y accesible a ese objeto. Las aplicaciones Java pueden quedarse sin memoria.

Resumen

instanceof Operator

- Instanceof es sólo para variables de referencia; Comprueba si el objeto es de un tipo particular.
- El operador instanceof sólo puede usarse para probar objetos (o nulos) en los tipos de clase que están en la misma jerarquía de clases.
- Para las interfaces, un objeto pasa la prueba instanceof si alguna de sus superclases implementa la interfaz en el lado derecho del operador instanceof.

Resumen

Arithmetic Operators

- Los cuatro operadores primarios de matemáticas son add (+), subtract (-), multiply (*) y divide (/).
- El operador restante (módulo) (%) devuelve el resto de una división.
- Las expresiones se evalúan de izquierda a derecha, a menos que añada paréntesis, o a menos que algunos operadores en la expresión tengan mayor precedencia que otros.
- **Los operadores *, /, y% tienen mayor precedencia que + y -.**

Resumen

String Concatenation Operator

- Si cualquiera de los operandos es un String, el operador + concatena los operandos.
- Si ambos operandos son numéricos, el operador +, suma operandos.

Resumen

Garbage Collection (II)

- “Islands of objects” Pueden ser recolectados de basura, aunque se refieran entre sí.
- Se puede solicitar la recolección por parte del `garbage collector` con System.gc ();
- La clase Object tiene un método finalize().
- El método finalize () está garantizado para ejecutarse una vez y sólo una vez antes de que el `garbage collector` elimine un objeto.
- El `garbage collector` no da garantías; finalize () puede que nunca se ejecutarse. Se puede hacer no elegible dentro del mismo método.

Resumen

Relational Operators

- Los operadores relacionales siempre dan como resultado un valor booleano (true or false).
- Hay seis operadores relacionales: >,> =, <, <=, ==, y !=. Los dos últimos (== y !=) a veces se denominan operadores de igualdad.
- Cuando se comparan char, Java utiliza el valor Unicode del carácter como el valor numérico.
- Operadores de igualdad
    - Hay dos operadores de igualdad: == y! =.
    - Se pueden probar cuatro tipos de cosas: números, caracteres, booleanos y variables de referencia.
- Al comparar las variables de referencia, == devuelve verdadero sólo si ambas referencias se refieren al mismo objeto.

Resumen

Increment/Decrement Operators

- Los operadores de `prefix` (por ejemplo, ++ x y -x) se ejecutan antes de que se utilice el valor en la expresión.
- Los operadores `postfix` (por ejemplo, x ++ y x--) se ejecutan después de que se utiliza el valor en la expresión.
- En cualquier expresión, ambos operandos se evalúan completamente antes de aplicar el operador.
- Las variables marcadas como `final` no se pueden incrementar ni disminuir.

Resumen

Ternary (Conditional) Operator

- Devuelve uno de los dos valores basándose en si su expresión booleana es true o false.
  - Devuelve el valor después del ? si la expresión es verdadera.
  - Devuelve el valor después de : si la expresión es falsa.

Resumen

Using String and StringBuilder (I)

- Los `String` son objetos inmutable, la variable de referencia no.
- Si creas un nuevo `String` sin asignar, se pierde.
- Si se redirecciona una referencia `String` a un nuevo `String`, se puede perder el `String` antiguo.
- methods use zero-based indexes, except for the second argument of substring().
- Los métodos de los `String` utilizan índices basados en cero, excepto el segundo argumento de substring ().
- La clase `String` es `final` .
- Cuando lae JVM encuentra literal `String`, este es sumado al pool de literales.
- Los `String` poseen un método llamado `length()`—los array un atributo llamado `length`.

Resumen

Using String and StringBuilder (II)

- Metodos a recordar en `Stringr`: `charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()`, y `trim()`.
- Los objetos `StringBuilder` son mutables — pueden cambiar sin crear un objeto nuevo.
- Los métodos del `StringBuilder` actúan sobre el objeto invocando, y los objetos pueden cambiar sin ninguna asignación.
- Recuerde que la cadena de metodos es evaluada de izquiera a derecha. Patron Builder
- Metodos a recordar en `StringBuilder`: `append()`, `delete()`, `insert()`, `reverse()`, y `toString()`.

Resumen

- La única expresión legal en una sentencia `if` es una expresión booleana, es decir, una expresión que se resuelve en una referencia `boolean` o `Boolean`.
- Cuidado con las asignaciones booleanas (=) que se pueden confundir con las pruebas de igualdad booleana (==):

```
boolean x = false;
if(x=true){} // las asignaciones son siempre true
```

- Las llaves son opcionales para si los bloques que tienen solamente una declaración condicional, **pero ten cuidado con las indentaciones engañosas**.
- Las sentencias `switch` sólo pueden evaluar `enums` o `byte, short, int, char`, y, a partir de Java 7, tipos de datos `String`:

```
long s = 30;
swtich(s){} ERROR
```

Resumen

- La constante `case` debe ser una variable literal o `final`, o una expresión constante, incluyendo un `enum` o `String`. No puede haber un caso que incluya una variable no `final` o un rango de valores.
- Si la condición en una instrucción `switch` coincide con una constante de caso, la ejecución se ejecutará a través de todo el código en el conmutador después de la instrucción de caso coincidente hasta que se encuentre una instrucción `break` o el final de la instrucción `switch`. En otras palabras, el caso coincidente es sólo el punto de entrada en el bloque de casos, pero a menos que haya una sentencia `break`, el caso coincidente no es el único código de caso que se ejecuta.
- La palabra clave `default` debe utilizarse en una instrucción `switch` si desea ejecutar algún código cuando ninguno de los valores de caso coincide con el valor condicional.
- El bloque `default` se puede ubicar en cualquier lugar del bloque de conmutación `switch`, por lo que si no hay coincidencias anteriores, se ingresará el bloque `default` y si el valor `default` no contiene una interrupción, el código continuará ejecutándose hasta el final del `switch` o hasta que se encuentre la instrucción `break`.

Resumen

Writing Code Using Loops (I)

- Una declaración básicade un `for` tiene tres partes: declaración y / o inicialización, evaluación booleana y la expresión de iteración.
- Si una variable se incrementa o se evalúa dentro de un bucle básico `for`, debe declararse antes del bucle o dentro de la declaración de bucle `for`.
- No se puede acceder a una variable declarada (no sólo inicializada) dentro de la declaración básica de bucle fuera del bucle `for`, es decir, el código por debajo del bucle `for` no podrá utilizar la variable.
- Puede inicializar más de una variable del mismo tipo en la primera parte de la declaración básica de bucle; cada inicialización debe estar separada por una coma.
- Un enunciado mejorado (nuevo a partir de Java 5) tiene dos partes: la declaración y la expresión. Se utiliza sólo para realizar bucle a través de `arrays` o `Collections`.

Resumen

Writing Code Using Loops (II)

- Con esta mejora `for`, la expresión es mediante `arrays` o `Collections` a través de la cual desea realizar bucle.
- Con un realzado para, la declaración es la variable de bloque, cuyo tipo es compatible con los elementos de `arrays` o `Collections`, y esa variable contiene el valor del elemento para la iteración dada.
- No puede utilizar un número (construcción de lenguaje de estilo C) o cualquier cosa que no evalúe un valor booleano como condición para una instrucción if o una construcción de bucle. No puede, por ejemplo, decir `if (x)`, a menos que `x` sea una variable booleana.
- El bucle `do` entrará al cuerpo del bucle al menos una vez, incluso si la condición de prueba no se cumple.

Resumen

Logical Operators

- El examen cubre seis operadores "lógicos": &, |, ^,!, &&, y ||.
- Los operadores lógicos trabajan con dos expresiones (excepto `for`!) que deben resolver a valores booleanos.
- Los operadores `&&` y `&` devuelven true sólo si ambos operandos son verdaderos.
- Los operadores || y | devuelven true si uno o ambos operandos son verdaderos.
- Los operadores `&&` y `||` son conocidos como operadores de cortocircuito.
- El operador `&&` no evalúa el operando derecho si el operando izquierdo es falso.
- El operador || no evalúa el operando derecho si el operando izquierdo es verdadero.
- Los operadores & y | Los operadores siempre evalúan ambos operandos.
- El operador ^ (llamado "XOR lógico") devuelve true si exactamente un operando es verdadero.
- El operador ! (llamado el operador de "inversión") devuelve el valor opuesto del operando booleano al que precede.

Resumen

Using Arrays (I)

- Los `arrays` pueden contener primitivas u objetos, pero el propio `arrays` es siempre un objeto. Cuando se declara un `array`, los corchetes pueden estar a la izquierda o a la derecha del nombre. Nunca es legal incluir el tamaño de un `arrays` en la declaración.
- Debe incluir el tamaño de una `array` cuando la construye (utilizando new) a menos que esté creando un `array` anónimo.
- Los elementos de un `array` de objetos no se crean automáticamente, aunque los elementos de un `array` primitivo reciben valores predeterminados.
- Obtendrá una `NullPointerException` si intentas utilizar un elemento de un `array` de objetos, si ese elemento no se refiere a un objeto real.
- Los `array` se indexan comenzando con cero.
- Se produce una excepción `ArrayIndexOutOfBoundsException` si utiliza un valor de índice incorrecto.

Resumen

Using Arrays (II)

- Los `arrays` tienen un atributo de longitud - `length` - cuyo valor es el número de elementos. El último índice al que puede acceder es siempre un menos que la longitud del mismo. Los `arrays` multidimensionales son sólo `arrays` de `arrays`.
- Las dimensiones en un `array` multidimensional pueden tener diferentes longitudes.
- Un `arrays` de primitivas puede aceptar cualquier valor que se puede promover de forma implícita al tipo declarado del `array`, por ejemplo, una variable de bytes puede ir en una `array` de `int`.
- Un `arrays` de objetos puede contener cualquier objeto que pasa la prueba IS-A (o instanceof) para el tipo declarado del `array`. Por ejemplo, si `Horse` extiende `Animal`, entonces un objeto `Horse` puede entrar en un `array` de `Animal`.

Resumen

Using Arrays (III)

- Si asignas a un `array` a una referencia de un `array` declarada anteriormente, el `array` que estás asignando debe tener la misma dimensión que la referencia a la que se está asignando.

```
int arr[][][] = new int[2][2][2];

arr = new int[7][8]; // Inválido
```

- Puede asignar una `array` de un tipo a una referencia de un `array` previamente declarada de uno de sus supertipos. Por ejemplo, una `array` de `Honda` se puede asignar a un `array` declarada como tipo `Car` (suponiendo que `Honda` extienda el coche).

Resumen

Using ArrayList

- El `ArrayList` le permite cambiar el tamaño de su lista y hacer inserciones y eliminaciones a su lista mucho más fácilmente que los `arrays`.
- Para el examen OCA 7, las únicas declaraciones de `ArrayList` que necesita saber son de esta forma:

```
ArrayList<type> myList = new ArrayList<type>();
List<type> myList2 = new ArrayList<type>(); //polimorfismo
```

- El `ArrayList` puede contener sólo objetos, no primitivos, pero recuerda que autoboxing puede hacer que parezca que estás agregando primitivas a un `ArrayList` cuando de hecho estás agregando una versión de wrapper de una primitiva.
- El índice de `ArrayList` empieza en 0.
- El `ArrayList` puede tener entradas duplicadas. Nota: Determinar si dos objetos son duplicados es más complicado de lo que parece y no aparece hasta el examen OCP 7.
- El `ArrayList` tiene los siguientes métodos que se recomienda recordar: `add(element)`, `add(index, element)`, `clear()`, `contains()`, `get(index)`, `indexOf()`, `remove(index)`, `remove(object)`, and `size()`.

Resumen

Assertions (I)

- Dado que se puede simular el efecto de aserciones utilizando otras construcciones de programación, se puede argumentar que el punto de agregar afirmaciones a `Java` es que son fáciles de escribir. Las declaraciones de aserción vienen en dos formas:.

```
assert boolean-expression;
assert boolean-expression: information-expression;
```

- Ambas declaraciones dicen "Afirmo que la expresión booleana producirá un valor verdadero". Si no es así, la aserción producirá una excepción `AssertionError`. Esta es una subclase `Throwable`, y como tal no requiere una especificación de excepción.

```
public class Assert2 {
     public static void main(String[] args) {
          assert false: "Here's a message saying what happened";
     }
}
```

Resumen

Assertions (II)

- Otro ejemplo es el método `Thread.holdsLock ()` introducido en JDK 1.4. Esto se utiliza para situaciones complejas de subprocesamiento (como iterar a través de una colección de una manera segura de subproceso) donde debe confiar en el programador cliente u otra clase en su sistema utilizando la biblioteca correctamente, en lugar de en la palabra clave sincronizada solo. Para asegurarse de que el código está siguiendo correctamente los dictados del diseño de su biblioteca, puede afirmar que el hilo actual en realidad tiene el bloqueo:

```
assert Thread.holdsLock (this); // Aserción de estado de bloqueo
```

Resumen

Assertions (III)

- Las instrucciones de verificación son una adición valiosa a su código. Dado que las aserciones pueden ser deshabilitadas, las instrucciones de verificación deben utilizarse siempre que tenga conocimiento no obvio sobre el estado de su objeto o programa.

```
static {
     boolean assertionsEnabled = false;
     assert assertionsEnabled = true;
     if (!assertionsEnabled) {
          throw new RuntimeException("Assertions disabled");
     }
}
```

Resumen

Assertions (IV)

- El siguiente conmutador permite afirmaciones en varias granularidades:

```
java [ -enableassertions | -ea  ] [:<package name>"..." | :<class name> ]
```

- Sin argumentos, el conmutador habilita aserciones de forma predeterminada. Con un argumento que termina en "...", las aserciones se activan en el paquete especificado y en los subpaquetes. Si el argumento es simplemente "...", las aserciones se habilitan en el paquete sin nombre en el directorio de trabajo actual. Con un argumento que no termina en "...", las aserciones están habilitadas en la clase especificada.:

```
java [ -disableassertions | -da ] [:<package name>"..." | :<class name> ]
```

- Si una sola línea de comandos contiene varias instancias de estos conmutadores, se procesan en orden antes de cargar cualquier clase. Por ejemplo, para ejecutar un programa con aserciones activadas sólo en el paquete com.wombat.fruitbat (y cualquier subpaquete), se podría utilizar el siguiente comando::

```
java -ea:com.wombat.fruitbat... java -ea:com.wombat.fruitbat... <Main class>
```

Resumen

Assertions (V)

- Para ejecutar un programa con aserciones activadas en el paquete `com.wombat.fruitbat` pero deshabilitado en la clase com.wombat.fruitbat.Brickbat, se podría utilizar el siguiente comando::

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat <class>
```

- Los conmutadores anteriores se aplican a todos los cargadores de clases ya las clases del sistema (que no tienen un cargador de clases). Hay una excepción a esta regla: en su forma sin argumento, los modificadores no se aplican a clases de sistema. Esto hace que sea fácil activar afirmaciones en todas las clases excepto en las clases del sistema.

```
java [ -enablesystemassertions | -esa ]
```

- Para permitir afirmaciones en todas las clases del sistema.:

```
java [ -disablesystemassertions | -dsa ]
```

Resumen

Assertions (VI)

- Hay otra forma de controlar las aserciones: mediante programación, enganchando al objeto `ClassLoader`. JDK 1.4 agregó varios métodos nuevos a `ClassLoader` que permiten la activación y desactivación dinámica de aserciones, incluyendo `setDefaultAssertionStatus ()`, que establece el estado de aserción para todas las clases cargadas posteriormente.:

```
public class LoaderAssertions {
  public static void main(String[] args) {
    ClassLoader.getSystemClassLoader().setDefaultAssertionStatus(true);
    new Loaded().go();
  }
}

class Loaded {
  public void go() {
    assert false: "Loaded.go()";
  }
}
```

Resumen

Using break and continue

- Una instrucción `break` no etiquetada hará que la iteración actual de la construcción de bucles más interna se detenga y `continue` en la línea de código que sigue al bucle para ejecutarse.
- Una instrucción continue no etiquetada, hará que la iteración actual del bucle más interno se detenga, la condición de ese bucle que se verifique y si se cumple la condición, el bucle se ejecute de nuevo.
- Si la sentencia `break` o la sentencia `continue` están etiquetadas, hará que se produzca una acción similar en el bucle etiquetado, no en el bucle interno mas cercano.