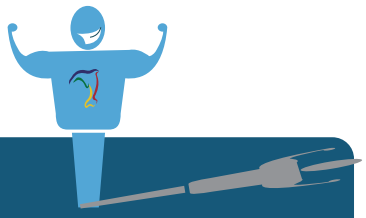
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

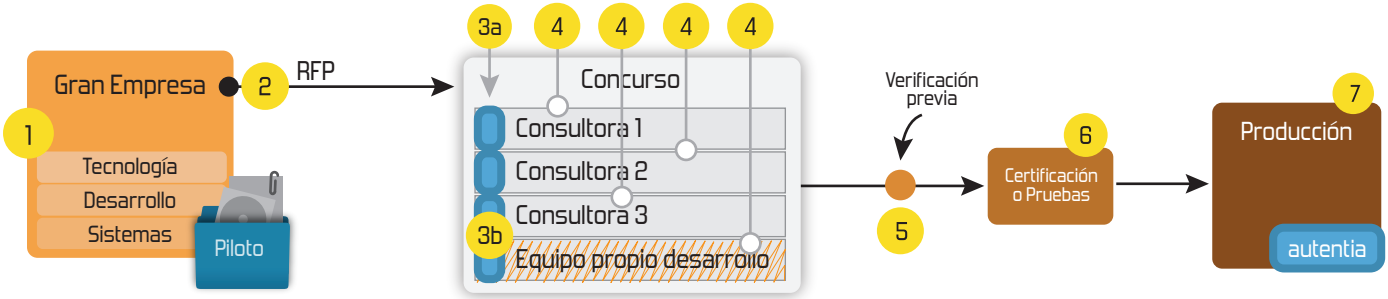
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Tutorial desarrollado por:
Juan Alonso Ramos (Autentia) es consultor tecnológico de desarrollo de proyectos informáticos.
Contacta en:
jalonso@autentia.com

Descargar este documento en formato PDF JavaSCJP5.pdf

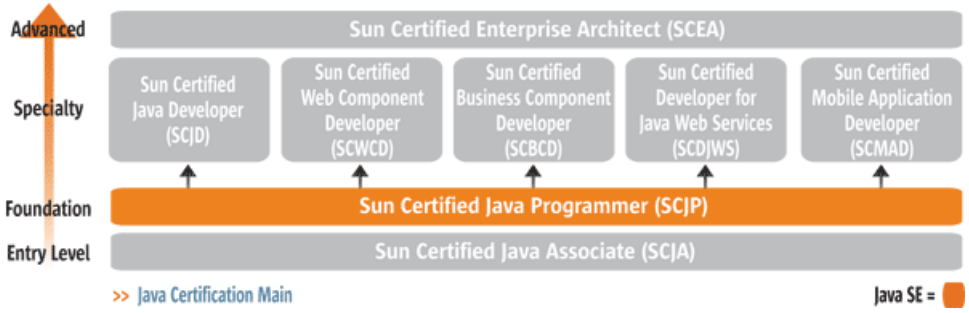
Fecha de creación del tutorial: 2007-08-13

Certificación Java SCJP 5.0

Introducción

El examen de certificación *Sun Certified Java Programmer* (SCJP) es una prueba donde Sun Microsystems evalúa los conocimientos sobre Java en su versión estándar para la versión 5.0. Es un test de 72 preguntas (tendrás que acertar al menos 43 lo que supone el 59%) donde deberás demostrar tu capacidad para detectar errores de compilación o de ejecución de un trozo de código, salida esperada de un programa, trozos de código que faltan a un programa para que resulten determinados valores de salida, conceptos de la API, conceptos de Orientación a Objetos, Threads, Colecciones, Genéricos, etc.

Para hacer este examen no se necesita ningún requisito en especial. Para optar a los siguientes sí es obligatorio aprobar el SCJP. El siguiente gráfico muestra las distintas certificaciones que hay. La fuente es de Sun Microsystems, Inc.



Todo lo que puede entrar en el examen está explicado detalladamente en la web de Sun.

Temario

Debido a que el temario es amplio y hay mucha información en otras páginas web y libros, en este tutorial únicamente veremos un resumen de lo más importante. Lo primero que debes aprender es lo nuevo que incluye Java 5.0, un tutorial explicándolo con más detalle lo encontrarás aquí.

Sección 1: Declaraciones, inicialización y alcance

1. 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.

2. Las clases únicamente pueden ser **public** o **default** (no poner nada delante del class).

3. Modificadores 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).

4. Las clases también pueden ser declaradas como **final** (no se puede extender de ella), **abstract** (no implementa alguno de sus métodos) y **strictfp** (los métodos de la clase seguirán las reglas del IEEE 754 para los números en coma flotante).

5. Métodos y atributos (miembros de una clase): Tienen los 4 modificadores. Los atributos son siempre inicializados a sus valores por defecto.

6. 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.

7. Las **interfaces** únicamente declaran sus métodos (no los implementan). éstos son implícitamente public, final y static. También se pueden declarar constantes en las interfaces (public, static y final). Las interfaces pueden heredar de otras interfaces, tienen herencia múltiple (entre interfaces extends nunca implements).

8. Una clase únicamente puede heredar de otra (herencia simple) e implementar múltiples interfaces. Siempre que se herede de una clase, se tendrá acceso a todos los miembros declarados como public y protected de la clase. 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.

9. Si desde una clase no abstracta se implementa una interfaz, es obligatorio implementar todos los métodos de esa interfaz. Las **clases abstractas** no tienen la obligación de implementar los métodos de la interfaz, 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.

10. Las clases abstractas no se pueden instanciar, es decir no se pueden crear objetos de esa clase.

11. 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 ninguno.

12. Cuando se crea un objeto de una clase, se llamará a uno de sus constructores, con o sin argumentos en función de la instanciación del objeto. En la primera línea de cada constructor el compilador insertará una llamada al constructor 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.

13. Los **identificadores** únicamente pueden empezar por letras, guión bajo o símbolo de dólar.

14. Palabras reservadas: Son todas las palabras propias del lenguaje y que no se pueden utilizar para declarar atributos, métodos, clases, etc. Tienes una lista *aquí*.

15. Métodos que aceptan **argumentos variables**. 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.

16. Las **variables** dentro de los métodos deben ser siempre inicializas expresamente, en este caso a diferencia de los atributos, el compilador dará un error si no son inicializas a algún valor. Las **constantes** se indican con el modificador final.

17. Los tipos **enumerados** se definen usando la palabra *enum*. únicamente pueden ser public o default. 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.

18. 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.*;

19. Las **clases internas** son clases que se declaran dentro de otras clases. Para acceder a ellas desde otra clase que no sea la propia externa, en la declaración se debe indicar el nombre de la externa. Ejemplo: ClaseExterna.ClaseInterna ci = new ClaseExterna.ClaseInterna(); Desde 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: las **clases internas locales a un método** (se definen dentro de un método de la clase externa y no pueden utilizar las variables de éstos excepto las finales), **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.

Sección 2: Control de flujo

1. El **nuevo bucle for** acepta cualquier colección (que extienda de java.util.Collection) o arrays pero nunca iteraciones (java.lang.Iterable). Su sintaxis es for(Tipo identificador : expresión) {}. No se pueden eliminar elementos de la colección mientras se está iterando. Tampoco se pueden modificar los tipos simples ni el tipo String en una colección mientras se está iterando, es decir el valor que toma el identificador no puede ser modificado dentro del for. Esto no es así para los objetos que sí se pueden modificar.

2. Los argumentos de un **switch** pueden ser de tipo byte, short, char, int o un tipo enumerado. Los argumentos del case únicamente pueden ser literales o variables finales.

3. Dentro de cualquier bucle se puede salir del mismo con un **break**. Para saltar una iteración del bucle se hace mediante **continue**.

4. Las **aserciones** son expresiones condicionales que se espera se evalúen a true para que el programa pueda seguir su ejecución. Si su resultado fuera false se produciría una AsertionError y el programa se pararía. Se indican con la palabra *assert* seguido de la expresión condicional. Por ejemplo si un objeto no puede venir nunca a null se podría indicar en un assert, por ejemplo: *assert obj != null : "El objeto 'obj' no puede ser null"*. Si obj no fuera null seguiría ejecutándose correctamente el programa y si fuera null se produciría una AsertionError. El control de las aserciones están deshabilitadas por defecto por lo que habría que habilitarlas en el momento de la ejecución. Esto se hace con *java -ea Programa.* También se puede utilizar *java -enableassertions NombreClase.* Para deshabilitarlas de nuevo bastaría con poner *java -da NombreClase* o *java -disableassertions.*

5. Todas las **excepciones** son subclases de java.lang.Throwable. únicamente las excepciones que se deberían controlar son las subclases de java.lang.RuntimeException que a su vez extienden de *java.lang.Exception*. Estas excepciones pueden ser capturadas dentro de un bloque **try-catch** o bien lanzadas hacia arriba (al método que llamó al que lanza la excepción) mediante ***throw*** (el método debe indicar que lanza excepciones mediante ***throws***). En el bloque *try* también se puede indicar un bloque *finally* que se garantiza su ejecución aunque se produzca cualquier excepción (a menos que se haga un *System.exit()*). También se puede indicar un bloque *try-finally* sin *catch*. Los errores no controlados son subclases de *java.lang.Error*.

6. Las excepciones que no debes olvidar son: **ArrayIndexOutOfBoundsException** (se produce cuando se intenta acceder a una posición de un array que no existe), **ClassCastException** (se produce al intentar hacer casting a una clase que no es una subclase del objeto), **NullPointerException** (se produce cuando se intenta acceder a algún miembro de un objeto que es nulo), **IllegalArgumentException** (se lanza para indicar que un método ha recibido algún argumento no esperado), **IllegalStateException** (la llamada a un método se ha hecho en un momento inapropiado), **NumberFormatException** (se lanza cuando se intenta formatear un String a un tipo numérico pero el String contiene caracteres inadecuados), **AsertionError** (cuando una aserción se evalúa a false), **ExceptionInInitializerError** (se lanza ante una excepción inesperada cuando se intenta inicializar algún atributo estático, siempre

dentro de bloques static), **StackOverflowError** (se lanza cuando se produce un desbordamiento de pila), **NoClassDefFoundError** (lanzada por el ClassLoader al intentar acceder a una clase que no existe).

Sección 3: Contenidos de la API

1. Los **tipos primitivos** tienen una clase asociada denominada *wrapper clase*, int -> Integer, long -> Long, char -> Character... Cada clase, excepto Character, tiene dos constructores, uno que acepta el valor primitivo y otro que acepta un String que representa el valor.

2. Autoboxing: Es la conversión automática entre el valor primitivo y su wrapper clase asociada (de int a Integer). **Unboxing**: Proceso contrario al autoboxing, se pasa de un tipo wrapper a su valor primitivo (por ejemplo de Integer a int).

3. Las instancias de la clase **String** una vez que toman un valor son inmutables, su valor no cambia. Si queremos concatenar varios Strings lo mejor es utilizar la clase **StringBuffer** o **StringBuilder**. Estas dos clases son similares, la diferencia es que StringBuffer tiene el acceso a su contenido sincronizado.

4. La clase **File** representa un archivo o un directorio. La clase **FileReader** se utiliza para leer streams de caracteres de un fichero. La clase **FileInputStream** se utiliza para leer streams de bytes desde un fichero. La clase **BufferedReader** lee líneas de texto desde un stream de caracteres. Para leer desde un fichero de texto bastaría con hacer lo siguiente: BufferedReader br = new BufferedReader(new FileReader(new File("fichero.txt"))); La clase BufferedReader permite leer línea a línea el fichero mediante el método readLine. Estas clases se utilizan para leer, para realizar operaciones de escritura a un fichero las clases son: **FileWriter, FileOutputStream, BufferedWriter**.

5. Implementando la interfaz **Serializable** se puede almacenar en un fichero el contenido completo de un objeto (excepto los atributos marcados como **transient** o **static**). La interfaz **Externalizable** también permite que el objeto sea serializado a un archivo pero es responsabilidad suya el proceso de lectura y escritura de su contenido. La interfaz *Externalizable* extiende de *Serializable*.

6. La clase **Locale** representa un valor específico dependiendo de una región. Se utiliza para formatear fechas, usos horarios, monedas, etc.

7. La clase **DateFormat** se utiliza para formatear fechas dependiendo del Locale. Es una clase abstracta que a través del método getDateInstance() se obtiene una referencia a un objeto de tipo DateFormat.

8. La clase **NumberFormat** se utiliza para formatear y parsear números dependiendo, al igual que DateFormat, del Locale de la máquina donde se ejecuta el programa.

9. La clase **Pattern** representa una expresión regular. A través del método compile() se compila la expresión regular y mediante el método matcher se le pasa la expresión donde buscar las ocurrencias que marca la expresión regular. Este método devuelve un objeto de la clase **Matcher** que proporciona métodos para buscar dentro de la expresión.

10. La clase **Scanner** representa una cadena de texto que puede ser parseada utilizando expresiones regulares.

11. A través de **System.out.printf** al igual que se hace en C podemos formatear argumentos mediante reglas especiales: **%b** -> indica que si el argumento a evaluar es *null*, el resultado devuelto será *false y* si el argumento es un boolean (o Boolean), el resultado será su String.valueOf(), **%c** -> el resultado será un carácter Unicode, **%d** -> el resultado se formateará como un entero decimal, **%f** -> el resultado se formateará como un número decimal, **%s** -> si el argumento es *null*, el resultado mostrado será null. Si el argumento arg implementa **Formattable**, se invocará arg.formatTo, si no el resultado devuelto se obtendrá invocando a arg.toString().

Sección 4: Concurrencia

1. Los *threads* se pueden crear de dos formas: extendiendo de la clase **Thread** o implementando la interfaz **Runnable** (e implementando el método public void run()).

2. Cuando se crea el Thread está en el estado ***New***. Se debe invocar al método *start()* para que pase al estado ***Runnable*** aunque no empezará su ejecución mientras no se invoque el método *run*() y pase al estado *Runnning.* Muchos threads pueden estar en el estado *Runnable* pero sólo uno puede estar en el estado ***Running*** en máquinas con un único procesador.

3. No se puede llamar más de una vez al método **start()** ya que lanzará un **IllegalThreadStateException** si el thread ya ha sido iniciado. El método start() llama al método **run()** para que el thread se ejecute.

4. El thread se considera terminado cuando termina su ejecución el método run(). Una vez muerto el thread no se puede arrancar de nuevo.

5. El método **sleep()** detiene el thread durante un tiempo determinado indicado en milisegundos.

6. El método **yield()** mueve el thread del estado Running a Runnable, es decir detiene su ejecución y lo pone en la cola de procesos listos para ejecutar.

7. Cuando un thread llama al método **join()** de otro thread, el actual thread en ejecución esperará hasta que el thread que invocó al join() termine.

8. El método **isAlive()** devuelve true si el thread ya ha sido iniciado.

9. Cuando se crea un thread se puede indicar su prioridad de ejecución mediante el método **setPriority()** que va de 1 (mínima prioridad) a 10 (máxima prioridad).

10. Para garantizar que únicamente un thread acceda concurrentemente a un bloque de código o a un método, se debe marcar con **synchronized**.

11. Los métodos *wait()*, *notify()* y *notifyAll()* son métodos definidos en la clase *java.lang.Object* y deben ser incluidos dentro un contexto sincronizado.

12. El método **wait()** pone al thread que lo invoque en estado de espera hasta que otro thread llame al *notify()* o al *notifyAll()*.

13. El método **notify()** se utiliza para poner de nuevo un thread que está detenido al estado Running.

14. El método **notifyAll()** realiza el mismo trabajo que el notify() pero notifica a todos los threads en estado de espera.

Sección 5: Conceptos OO

1. La **encapsulación** ayuda a que las aplicaciones sean reutilizables y robustas. Los atributos de una clase siempre deberán tener acceso privado con métodos get y set para su acceso y modificación.

2. Normalmente se utiliza el concepto **es-un** para referirnos a una relación de herencia o implementación expresado con *extends* o *implements*. Para referirnos a una asociación entre clases se utiliza el concepto **tiene-un** e indica que en una clase, alguno de sus atributos son instancias de otras clases.

3. La **herencia** es el mecanismo que permite a una clase ser subclase o superclase de otras. Todas las clases (excepto Object) son subclases de Object y por lo tanto heredan sus métodos.

4. El **polimorfismo** permite que un objeto puede ser referenciado con alguno de los tipos de las clases padre. Se consigue mediante la herencia o la implementación. El tipo de la variable de referencia es el que determina qué métodos pueden ser utilizados. Por ejemplo las clases *ArrayList* y *HashSet* implementan las interfaces *List* y *Set* respectivamente. Estas dos interfaces a su vez implementan la interfaz *Collection*. Es perfectamente correcto lo siguiente: *Collection c1 = new ArrayList();* y *Collection c2 = new HashSet()*;

5. Los métodos que su tipo de retorno sea un tipo primitivo, pueden retornar cualquier valor que pueda ser implícitamente convertido al tipo de retorno, por ejemplo de short a int o de long a double. Esto también es equivalente a los objetos y se denomina **retorno covariante**. Si un método debe devolver un objeto de tipo Collection, es perfectamente válido devolver una subclase de Collection, por ejemplo ArrayList o HashSet.

6. Los métodos pueden ser **sobrescritos** y **sobrecargados**. Los **constructores** no se pueden sobrescribir ya que no se heredan. Deben tener el mismo nombre que la clase y no tienen tipo de retorno. Los constructores tienen los cuatro modificadores. La primera línea del constructor debe ser una llamada a this() o a super(). Si no se indica expresamente el compilador lo insertará por nosotros en el momento de la compilación. No se puede hacer en el mismo constructor una llamada a this() y a super().

7. Para que un método sea sobrescrito se debe considerar lo siguiente: debe tener el mismo nombre, lista de argumentos, tipo devuelto (excepto en tipos de retorno covariante), no debe tener un acceso más restrictivo ni lanzar nuevas excepciones. De esta manera el método de la superclase quedará sobrescrito. Para acceder al método de la superclase se ha de hacer con *super.nombreMetodo*.

8. Para sobrecargar un método: debe tener el mismo nombre, debe tener diferente lista de argumentos, puede tener diferente tipo de retorno, puede usar diferente modificador o lanzar diferentes excepciones.

9. Los métodos marcados con **final** no se pueden sobrescribir.

10. Siempre que se llama al constructor de una clase, se llama también al de su superclase y así sucesivamente hasta Object.

Sección 6: Colecciones / Genéricos

1. Una colección es una estructura de datos que se utiliza para almacenar objetos.

2. Deberás aprender y saber diferenciar las características de varias colecciones, entre ellas están las interfaces List y Set, ambas implementan la interfaz **Collection** y por otro lado la interfaz Map.

3. La interfaz **List** es una colección que puede contener elementos duplicados y donde importa el orden de sus elementos. Algunas clases que implementan List son: **ArrayList** (permite rápidas iteraciones y acceso aleatorio), **Vector** (similar a ArrayList pero con sus métodos sincronizados), **LinkedList** (aÃ±ade sus elementos al principio o al final, utilizado para implementar pilas y colas).

4. La interfaz **Set** no permite objetos duplicados y éstos son insertados de forma aleatoria y sin ordenar. Algunas clases que implementan este interfaz son: **HashSet** (asegura que no haya objetos duplicados, no mantiene la ordenación de los elementos y su acceso es rápido), **LinkedHashSet** (no permite objetos duplicados pero mantiene un orden en la inserción), **TreeSet** (no permite objetos duplicados y los inserta ordenados, por orden alfabético, numérico o configurable).

5. La interfaz **Map** inserta elementos asociándoles una clave para su posterior acceso, sin ningún orden de inserción ni ordenación. Algunas clases que implementan Map son: **HashMap** (almacena objetos y claves de rápido acceso, permite elementos y claves nulas), **Hashtable** (igual que HashMap pero con sus métodos sincronizados, no permite claves ni valores nulos), **LinkedHashMap** (mismas características que HashMap pero los elementos se almacenan por orden de inserción), **TreeMap** (se insertan los elementos ordenados).

6. Para que los elementos de una colección puedan ser considerados iguales (en el caso de la interfaz Set) o la búsqueda por claves sea eficiente (en el caso de Map), cuando definimos la clase de los objetos a almacenar deberíamos sobrescribir los métodos **equals** y **hashCode**: public boolean equals(Object o) y public int hashCode().

7. El operador **"=="** determina si dos variables se refieren al mismo objeto. El método equals determina si dos objetos son exactamente el mismo.

8. Si dos objetos son iguales (el método equals de uno de ellos pasándole el otro objeto como argumento devuelve true), sus hashcodes deberían también ser iguales. Por tanto cuando se implementa el equals también se debería implementar el hashcode. La implementación por defecto del equals() devuelve true únicamente si los dos objetos que intervienen en la comparación son el mismo. Se debe utilizar para la implementación de estos dos métodos los atributos que hagan único al objeto y que lo diferencie de los demás.

9. Cuando se comparan variables de referencia utilizando el operador "==" el resultado de la comparación es true únicamente si las variables de referencias se refieren al mismo objeto.

10. Las clase String y las wrapper classes sobrescriben el método equals(). Dos cadenas de texto se pueden comparar con el equals para determinar si son la misma. En las wrapper clases, el método equals devolverá true si dos objetos tienen el mismo tipo y el mismo valor. Si comparamos instancias de la misma clase wrapper con el operador "==", únicamente se consideran iguales las instancias de Boolean con el mismo valor (true o false), las instancias de Byte, las instancias de Character desde '\u0000' a '\u007f' y las instancias de Short e Integer en el rango -128 al 127.

11. La clase **Collections** proporciona métodos de utilidades para utilizar con colecciones.

12. La clase **Arrays** proporciona métodos útiles para trabajar con arrays.

13. La interfaz **Comparable** define el método *compareTo(Object o)* para comparar el objeto con los demás elementos de una colección para ordenarlo.

14. La interfaz **Comparator** define el método *compare(Object o1, Object o2)* que compara entre dos objetos para su ordenación en estructuras de tipo TreeSet o TreeMap donde la inserción de los elementos se realiza ordenada.

15. Con el uso de los **genéricos** se evitan las excepciones en tiempo de ejecución por culpa de la inserción de objetos de distinto tipo en las colecciones. Los tipos genéricos únicamente son evaluados en tiempo de compilación, en ejecución no existen aunque nos aseguramos que si el código compila correctamente, no nos encontraremos con problemas de casting de objetos en la ejecución del código. Por ejemplo si declaramos la siguiente lista: List<Integer> myList = new ArrayList<Integer>() el compilador no dejará que se inserte un objeto que no sea de tipo Integer. El compilador sí aceptará subclases del tipo indicado, por ejemplo si se declara una List<Map> se podrá insertar un tipo HashMap, TreeMap, etc, es decir admite el polimorfismo.

16. No se puede instanciar una colección de un tipo genérico diferente al de la declaración del objeto. Lo siguiente no es correcto: ArrayList<Map> list = new ArrayList<HashMap>(). En el momento de la declaración (tipo Map) e instanciación (tipo HashMap) del objeto colección no se admite el polimorfismo.

17. La superclase de todos los genéricos es **<?>**, indica que cualquier genérico puede ser insertado en la colección. También se puede indicar el polimorfismo en los genéricos mediante **<? extends Map>**. Todo lo que extienda de Map se puede insertar en la colección.

18. Los genéricos también se pueden utilizar en la declaración de clases (class Foo<T> { ... }), atributos de la clase (T instancia), constructores (Foo (T ref)), métodos y tipos de retorno (public T bar (T ref) {}). El compilador sustituirá el tipo por el que se cree cuando se instancie un objeto de la clase Foo.

Sección 7: Conceptos fundamentales

1. Se puede invocar al **recolector de basura** mediante System.gc() pero no se asegura su ejecución.

2. El método **finalize()** está definido en la clase Object y se sobrescribe para que sea llamado justo antes de que el recolector de basura elimine el objeto y éste pueda liberar los recursos que tenga asociados.

3. Cuando se pasa un objeto a un método lo que realmente se le pasa es la referencia al objeto en memoria. Tanto el código que llama al método como el método llamado comparten el mismo objeto por lo que si el método lo modifica, estará modificando el mismo objeto que se le pasó.

4. El operador **instanceof** se utiliza para determinar si un objeto es una instancia particular de una clase

5. El operador condicional **&** devuelve true si ambos operandos son true.

6. El operador condicional **&&** devuelve true si ambos operandos son true. Si la evaluación del primero no devuelve true, el segundo no se evalúa.

7. El operador **|** devulve true si al menos un operando es true.

8. El operador **||** devuelve true si al menos un operando de la comparación es true. Si la evaluación del primero es true, el segundo no se evalúa.

9. El operador **^** (XOR) devuelve true si únicamente uno de los dos operandos es true.

10. El operador **!** devuelve el valor opuesto del operando booleano que le sucede.

11. Los **operadores ++** y **--** si se ponen delante de la variable, se ejecutan antes de que el valor sea utilizado en la expresión y si se indican después de la variable se ejecutan después de que el valor se utilice en la expresión.

12. El **operador ternario condicional** evalúa la expresión booleana y asigna el valor a la variable que le sigue al operador **:** si el resultado es true, si no le asigna el valor que sigue después del operador **?**. Ejemplo: int i = (0 == 0f) ? 1 : 2; El resultado de la expresion (0 == 0f) es true por lo que a la variable i se le asignará el valor 1.

13. Longitudes tipos primitivos: enteros: byte (8 bits), short (16 bits), int (32 bits), long (64 bits); punto flotante: float (32 bits),

double (64 bits).

Solicitar un examen

Para solicitar un examen debes entrar en la web de Sun y en la parte de exámenes de certificación solicitar el "*Sun Certified Programmer for the Java 2 Platform, Standard Edition 5.0 (CX-310-055)*". Lo primero es dar tus datos personales y hacer el pago. Por correo electrónico te enviarán un código (lo denominan voucher) que deberás dar cuando te pongas en contacto con alguno de los centros prometric que es donde se realizan los exámenes. En la web de prometric puedes solicitar el centro que mejor te pille y también el día y hora de la prueba.

Conclusiones

Espero que te haya sido de ayuda todo lo que se ha explicado en este tutorial, si no es para dar el paso y probar suerte con el examen seguro te ha ayudado a recordar algunas cosas del lenguaje que estaban olvidadas.

Los contenidos del examen de certificación no tratan aspectos diferentes a los que utilizamos día a día los programadores por lo que no es complicado aprobarlo si estás familiarizado con el lenguaje. Si te planteas sacarte la certificación en Java debes tener en cuenta que las tecnologías cambian rápido y no es algo que te vaya a durar toda la vida pero siempre es un reto.

Si quieres más información te recomiendo que visites otras páginas web que tratan de forma más específica las certificaciones en Java.

http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
http://faq.javaranch.com/view?ScjpFaq
http://www.examulator.com/phezam/login.php
http://www.wickedlysmart.com/SCJPStudyGuide/Java_5_SCJPquestions.html
http://www.javabeat.net/javabeat/scjp5/mocks/index.php
http://www.javacoffeebreak.com/articles/toptenerrors.html

Puedes opinar sobre este tutorial aquí

Mensajes multi-idioma en Java	Locate, ResourceBundle, MessageFormat, etc. Fundamental para un correcto diseño ...
Decompilar Java	Os mostramos como recuperar el fuente de vuestro código a partir de los ficheros compilados .class
Documentar código Java con JavaDoc	Os mostramos como utilizar los comentarios y etiquetas de JavaDoc para documentar programas Java.
Introducción a JavaHelp	En este tutorial os mostramos los primeros pasos para utilizar el API JavaHelp de cara a la generación de ayudas con Java
Optimización Java con Eclipse Profiler Plugin	Alejandro Pérez nos enseña como analizar el rendimiento de nuestras aplicaciones con Eclipse Profiler Plugin.
Ordenar Listas en Java	En este tutorial se pretende mostrar una forma sencilla de ordenar listas en java usando dos métodos estáticos de la clase java.util.Collections e implementando dos interfaces Comparable y Comparator
Técnicas básicas y poco comentadas en Java	Os mostramos como realizar algunas cosas simples en Java: Formateo de decimales y enteros, gestión de preferencias y comparación entre objetos de nuevas clases
Construir un Servidor Web en Java	En este tutorial os enseñamos los principios de las aplicaciones multi-hilo a través de la creación de un servidor web básico en Java. Podremos ver en un ejemplo real el uso de sockets, threads, excepciones, etc.