Simuladores

Evidencia “Java Basics”

1.

Name	Taken on - 17 jul, '24 11:53 AM	**Status**	Passed 71%
Correct Answers	12	**Total Questions**	17
Time Taken	00:44:18	**Total Time**	00:36:16
Start Time	17 jul 24 10:53	**Finish/Pause Time**	17 jul 24 11:40

Test Details | Performance Report

S ...	Marked	Atte...	Result	Exam Objective	Difficulty Le...	Problem Statement	Note
1		✓	✗	01 - Java Basics - O...	Easy	Which of the following are correct about "enc...	
2		✓	✓	01 - Java Basics - O...	Easy	Which of the following are benefits of polymo...	
3		✓	✗	01 - Java Basics - O...	Tough	`short firstValue = 5;`	
4		✓	✓	01 - Java Basics	Very Easy	Which of the following are valid declarations o...	
5		✓	✓	01 - Java Basics - O...	Easy	}	
6		✓	✓	01 - Java Basics - O...	Easy	given the following requirements - 1. Implement three classes - Car, SUV, and	
7		✓	✓	01 - Java Basics	Very Easy		
8		✓	✓	01 - Java Basics - O...	Easy	Which of the following are features of Java?S...	
9		✓	✗	01 - Java Basics	Tough	contents of a file.	
10		✓	✓	01 - Java Basics	Very Easy		
11		✓	✓	01 - Java Basics	Easy	value = Integer.parseInt(globa	
12		✓	✓	01 - Java Basics	Very Easy	`if (args.length == 0 ){`	
13		✓	✗	01 - Java Basics - O...	Easy	`... other irrelevant code`	
14		✓	✓	01 - Java Basics	Tough	`public static long main(String[] args){`	
15		✓	✓	01 - Java Basics	Very Easy	`System.out.println("Hello");`	

2.

Test Overview Time Left - OVER LIMIT 00:02:32

Name	Taken on - 19 jul, '24 10:50 AM	**Status**	Passed 78%
Correct Answers	14	**Total Questions**	18
Time Taken	00:40:56	**Total Time**	00:38:24
Start Time	19 jul 24 10:50	**Finish/Pause Time**	19 jul 24 11:31

Test Details | Performance Report

S ...	Marked	Atte...	Result	Exam Objective	Difficulty Le...	Problem Statement	Note
1		✓	✓	01 - Java Basics	Very Easy		
2		✓	✗	01 - Java Basics - O...	Easy	Which of the following are benefits of polymo...	
3		✓	✗	01 - Java Basics	Tough	and	
4		✓	✓	01 - Java Basics	Easy	}	
5		✓	✓	01 - Java Basics	Very Easy	`System.out.println(harry);`	
6		✓	✓	01 - Java Basics	Very Easy	`if (args.length == 0 ){`	
7		✓	✓	01 - Java Basics	Very Easy	You have written some Java code in MyFirstClass.java file. Which of the following	
8		✓	✗	01 - Java Basics	Very Easy		
9		✓	✓	01 - Java Basics	Very Easy	`System.out.println(args[1]);`	
10		✓	✗	01 - Java Basics - O...	Very Tough	Identify correct option(s)	
11		✓	✓	01 - Java Basics - O...	Easy	Which of the following are correct about "enc...	
12		✓	✓	01 - Java Basics - O...	Easy	`public void setRadius(int r){`	
13		✓	✓	01 - Java Basics - O...	Easy	given the following requirements - 1. Implement three classes - Car, SUV, and	
14		✓	✓	01 - Java Basics - O...	Easy	Which of the following are features of Java?S...	
15		✓	✓	01 - Java Basics	Tough	`static A a1 = new A(1);`	

View Questions | Pause Test | Evaluate Test

Evidencia “Working with Java Data Types”

1.

Name Taken on - 20 jul, '24 05:39 PM
Correct Answers 14
Time Taken 00:19:33
Start Time 20 jul 24 17:39

Status Passed 78%
Total Questions 18
Total Time 00:38:24
Finish/Pause Time 20 jul 24 17:59

Test Details | Performance Report

S ...	Marked	Atte...	Result	Exam Objective	Difficulty Le...	Problem Statement	Note
1		✓	✓	02 - Working with J...	Very Easy	8. s.grade();	
2		✓	✓	02 - Working with J...	Tough	public static void main(String[] args){	
3		✓	✓	02 - Working with J...	Very Easy	Which of these are NOT legal declarations wit...	
4		✓	✓	02 - Working with J...	Very Easy	}	
5		✓	✓	02 - Working with J...	Real Brainer	int i1 = 1, i2 = 2, i3 = 3;	
6		✓	✗	02 - Working with J...	Tough	public static void main(String[] args){	
7		✓	✗	02 - Working with J...	Tough	Which of the following are valid code snippets...	
8		✓	✓	02 - Working with J...	Very Tough		
9		✓	✓	02 - Working with J...	Very Tough		
10		✓	✓	02 - Working with J...	Tough		
11		✓	✓	02 - Working with J...	Very Easy	public void addMore(int more){	
12		✓	✓	02 - Working with J...	Easy	Which of the following statements will print tr...	
13		✓	✓	02 - Working with J...	Real Brainer	return y;	
14		✓	✓	02 - Working with J...	Real Brainer	class Two extends Super{	
15		✓	✓	02 - Working with J...	Tough	static final String CLASS_GUID; // 2	
16		✓	✗	02 - Working with J...	Easy	int coupon, offset, base; //3	
17		✓	✗	02 - Working with J...	Very Tough	String get(){	
18		✓	✓	02 - Working with J...	Easy	int i = Integer.parseInt(args[i]);	

2.

Test Overview Time Left - 00:17:41

Name Taken on - 20 jul, '24 06:34 PM
Correct Answers 13
Time Taken 00:20:43
Start Time 20 jul 24 18:34

Status Passed 72%
Total Questions 18
Total Time 00:38:24
Finish/Pause Time 20 jul 24 18:59

Test Details | Performance Report

S ...	Marked	Atte...	Result	Exam Objective	Difficulty Le...	Problem Statement	Note
1		✓	✓	02 - Working with J...	Very Easy	public static void main(String[] args)	
2		✓	✓	02 - Working with J...	Tough	String("aaaaa"); //1	
3		✓	✓	02 - Working with J...	Very Easy	void doSomething(Object s){ o = s; }	
4		✓	✓	02 - Working with J...	Very Easy	public static void main(String args[ ]	
5		✓	✓	02 - Working with J...	Tough	Long ln = new Long(42);	
6		✓	✓	02 - Working with J...	Tough		
7		✓	✓	02 - Working with J...	Very Easy	8. s.grade();	
8		✓	✓	02 - Working with J...	Tough	Identify the valid code fragments when occur...	
9		✓	✗	02 - Working with J...	Very Tough	Which of the following options will yield a Boo...	
10		✓	✓	02 - Working with J...	Easy	Which of the following statements will print tr...	
11		✓	✗	02 - Working with J...	Tough	Which of the following are valid code snippets...	
12		✓	✗	02 - Working with J...	Easy	static int i2 ;	
13		✓	✓	02 - Working with J...	Very Tough	Assume that a, b, and c refer to instances of...	
14		✓	✓	02 - Working with J...	Real Brainer	return y;	
15		✓	✗	02 - Working with J...	Easy	int coupon, offset, base; //3	
16		✓	✓	02 - Working with J...	Easy	}	
17		✓	✗	02 - Working with J...	Real Brainer	Which of the following are valid classes?	
18		✓	✓	02 - Working with J...	Real Brainer	int i1 = 1, i2 = 2, i3 = 3;	

Git & GitHub

Introducción

Hoy en día la colaboración es fundamental en el desarrollo de software, sitios web, y entre otros tipos de proyectos, por lo que se tiene la necesidad de trabajar sobre un mismo proyecto con agilidad y eficiencia. Nosotros podemos tener archivos en nuestro equipo y llevar un control sobre sus cambios, sin embargo, podríamos errar al entrar a la última versión, además se le suma la necesidad de introducir los cambios realizados por todo el equipo. Derivado de esta problemática surge lo que se presenta a continuación.

Conceptos clave

Sistema local de control de versiones.	Tiene una base de datos que mantiene todos los cambios en los archivos bajo control de revisión. Uno de estos sistemas es RCS (Revision Control System)que gestiona múltiples revisiones de archivos. RCS automatiza el almacenamiento, recuperación, registro, identificación y combinación de revisiones.
Sistema de control de versiones centralizado.	Los sistemas de control de versiones centralizados contienen solo un repositorio a nivel mundial y cada usuario debe comprometerse a reflejar sus cambios en el repositorio. Es posible que otros vean sus cambios mediante la actualización. Una ventaja es que todos saben hasta cierto punto lo que están haciendo todos los demás en el proyecto. Los administradores tienen un control detallado sobre quién puede hacer qué, y es mucho más fácil administrar un CVCS que tratar con bases de datos de cada cliente.
Sistemas de control de versiones distribuidos.	Los sistemas de control de versiones distribuidos contienen múltiples repositorios. Cada usuario tiene su propio repositorio, así como una copia de trabajo. El solo hecho de confirmar sus cambios no dará acceso a otros colaboradores a estos. Esto se debe a que la confirmación reflejará esos cambios en el repositorio local y deberá enviarlos para que sean visibles en el repositorio central. De manera similar, cuando actualiza, no obtiene los cambios de otros a menos que primero haya ingresado esos cambios en su repositorio. Los sistemas más populares de Sistemas de control de versiones distribuidos es Git y Mercuarial
Git	Git, que presenta una arquitectura distribuida, es un ejemplo de DVCS (sistema de control de versiones distribuido, por sus siglas en inglés). En lugar de tener un único espacio para todo el historial de versiones del software, como sucede de manera habitual en los sistemas de control de versiones antaño populares, como CVS o Subversion (también conocido como SVN), en Git, la copia de trabajo del código de cada desarrollador es también un repositorio que puede albergar el historial completo de todos los cambios.

Funcionamiento Git

Configuración de un repositorio comandos

1. **Git init**

`$ git init`	crea un nuevo repositorio de Git. Puede utilizarse para convertir un proyecto existente y sin versión en un repositorio de Git, o para inicializar un nuevo repositorio vacío.
`$ git clone <repo url>`	se emplea para crear una copia de un repositorio ya existente, git clone llama primero a git init par generar un nuevo repositorio, luego copia los datos del repositorio existente y extrae un nuevo conjunto de archivos de trabajo
`$ git init –bare <directory>`	Inicializa un repositorio de git vacío, pero omite el directorio de trabajo. Los repositorios compartidos deberían crearse con la marca –bare.
`$ git init <directory> --template=<template_directory>`	Las plantillas te permiten inicializar un nuevo repositorio con un subdirectorio de .git predefinido. Puedes configurar una plantilla para que tenga los directorios y archivos predeterminados que se copiarán en el subdirectorio de .git del nuevo repositorio.

2. ***Git clone***

$git clone	git clone es una utilidad de línea de comandos de Git que se utiliza para fijar como objetivo un repositorio existente con el fin de clonarlo o copiarlo. Una vez que un desarrollador ha obtenido una copia de trabajo, todas las operaciones de control de versiones se gestionan por medio de su repositorio local. La clonación crea automáticamente una conexión remota llamada "origin" que apunta al repositorio original.
`$ git clone <repo> <directory>`	Clona el repositorio ubicado en ＜repo＞ en la carpeta llamada ~ ＜directory＞! en la máquina local.
`$ git clone --branch <tag> <repo>`	Clona el repositorio ubicado en ＜repo＞ y clona solamente la referencia para ＜tag＞
`git clone -depth=1 <repo>`	Clona el repositorio ubicado en < repo > y clona solamente el historial de confirmaciones especificado por la opción depth=1. En este ejemplo, se realiza una clonación de < repo > y solo se incluye la confirmación más reciente en el nuevo repositorio clonado. La clonación superficial es muy útil cuando se trabaja con repositorios que tienen un largo historial de confirmaciones.
`$ git clone –branch`	El argumento -branch permite especificar una rama concreta para clonarla en vez de la rama a la que apunta el HEAD remoto, normalmente la rama principal. Asimismo, puedes incluir una etiqueta en vez de una rama con el mismo efecto.

3. **Git config**
    a. El caso práctico más básico de git config es invocarlo con un nombre de configuración, que mostrará el valor definido con ese nombre. Los nombres de configuración son cadenas delimitadas por puntos que se componen de una "sección" y una "clave" en función de su jerarquía. Por ejemplo: user.email.
    b. Niveles y archivos de git config
        - --local – aplica al repositorio de contexto en el que se invoca git config
        - --global – aplica al usuario de un sistema operativo
        - --system – afecta a todos los usuarios de un sistema operativo y a todos los respositorios.
        - Ejemplo:
          `$ git config --global user.email "your_email@example.com"`

    c. Editor de git config: core.editor
        - `$ git config --global core.editor "nano -w"`
4. **Git commit**
    a. El comando git commit captura una instantánea de los cambios preparados en ese momento del proyecto. Las instantáneas confirmadas pueden considerarse como versiones "seguras" de un proyecto: Git no las cambiará nunca a no ser que se lo pidas expresamente. Antes de ejecutar git commit, se utiliza el comando git add para pasar o "preparar" los cambios en el proyecto que se almacenarán en una confirmación. Estos dos comandos, git commit y git add, son dos de los que se utilizan más frecuentemente.
    b. `$ git commit` - Confirma la instantánea preparada. El comando abrirá un editor de texto que te pedirá un mensaje para la confirmación. Una vez escrito el mensaje, guarda el archivo y cierra el editor para crear la confirmación
    c. `$ git commit` –a - Confirma una instantánea de todos los cambios del directorio de trabajo. Esta acción solo incluye las modificaciones a los archivos con seguimiento (los que se han añadido con git add en algún punto de su historial).
    d. `$ git commit` –m "mensaje" - Un comando de atajo que crea inmediatamente una confirmación con un mensaje de confirmación usado. De manera predeterminada, git commit abrirá el editor de texto configurado localmente y solicitará que se introduzca un mensaje de confirmación. Si se usa la opción -m, se omitirá la solicitud de editor de texto a favor de un mensaje insertado.
    e. `$ git commit` -am "mensaje" - Un comando de atajo para usuarios avanzados que combina las opciones -a y -m. Esta combinación crea inmediatamente una confirmación de todos los cambios preparados y aplica un mensaje de confirmación insertado.
    f. `$ git commit –amend` - Esta opción añade otro nivel de funcionalidad al comando confirmado. Al pasar esta opción, se modificará la última confirmación. En vez de crear una nueva confirmación, los cambios preparados se añadirán a la confirmación anterior. Este comando abrirá el editor de texto configurado del sistema y te pedirá que cambies el mensaje de confirmación especificado anteriormente.

5. **Git add**
    a. Guarda los cambios para una confirmación.
    b. `$ git add <file>` - guarda el archivo editado
    c. `$ git status` – nos ayuda a examinar el resultado de esta acción.

6. **Git stash**
    a. El comando git stash almacena temporalmente (o guarda en un stash) los cambios que hayas efectuado en el código en el que estás trabajando para que puedas trabajar en otra cosa y, más tarde, regresar y volver a aplicar los cambios más tarde. Guardar los cambios en stashes resulta práctico si tienes que cambiar rápidamente de contexto y ponerte con otra cosa, pero estás en medio de un cambio en el código y no lo tienes todo listo para confirmar los cambios.
    b. `$ git stash pop` – Al hacer pop se eliminan los cambios de este y se vuelven a aplicar en el código en el que estás trabajando.
    c. `$ git stash apply` – Es un opción para volver a aplicar los cambios en el código en el que estas trabajando y conservarlos
    d. `$ git stash branch` - Para crear una rama nueva en la que aplicar los cambios del stash
    e. `$ git stash drop` – para eliminar un stash
7. **.gitignore**
    a. Los archivos ignorados suelen ser artefactos de compilación y archivos generados por el equipo que pueden derivarse de tu fuente de repositorios o que no deberían confirmarse por algún otro motivo. Estos son algunos ejemplos habituales:

        - Cachés de dependencias, como es el caso del contenido de /node_modules o /packages.
        - Código compilado como, por ejemplo, los archivos .o, .pyc y .class.
        - Directorios de salida de compilación, como es el caso de /bin, /out o /target.
        - Archivos generados en tiempo de ejecución como, por ejemplo, .log, .lock o .tmp.
        - Archivos ocultos del sistema, como es el caso de .DS_Store o Thumbs.db.
        - Archivos personales de configuración de IDE como, por ejemplo, .idea/workspace.xml.
8. **Examinar un repositorio**
    a. `$ git log` – Muestra el historial de confirmaciones completo con el formato predeterminado, puedes usar Espacio para desplazarte y q para salir.
    b. `$ git log –n <limit>` -Esta opción limita el número de confirmaciones.
    c. `$ git log --oneline` – permite agrupa cada confirmación en una sola línea.
    d. `$ git log –autor="<pattern>"` – busca las confirmaciones hechas por un autor en particular.
    e. `$ git log --grep="<pattern>"` – busca las confirmaciones con un mensaje.
    f. `$ git log <file>` - para ver el historial de un archivo en concreto.
9. **Trabajo colaborativo**

a. `$ git remote` – te permite crear, ver y eliminar conexiones con otros repositorios, en esencia es un interfaz para gestionar una lista de entradas remotas almacenadas en el archivo ./.git/config del repositorio.
b. `$ git remote -v` – enumera las conexiones remotas incluyendo la URL de cada conexión.
c. `$ git remote add <name> <url>` - crea una nueva conexión a un repositorio remoto.
d. `$git remote rm <name>` - elimina la conexión con el repositorio remoto que lleva el nombre <name>.
e. `$ git remote rename <old-name> <new-name>` - cambia el nombre de una conexión remota
f. `$ git push <remote-name> <branch-name>` - se cargará el estado local de < branch-name> en el repositorio remoto especificado por <remote-name>
g. `$ gitfetch` – descarga confirmaciones, archivos y referencias de un repositorio remoto a tu repositorio local, pero no te obliga a fusionar los cambios en tu repositorio. Git aísla el contenido recuperado del contenido local existente sin tener ningún tipo de repercusión sobre el desarrollo local de tu trabajo. El contenido recuperado debe extraerse específicamente con el comando git checkout. Esto permite que la recuperación constituya una forma segura de revisar confirmaciones antes de integrarlas en tu repositorio local.
h. $ git fetch <remote> - Recupera todas las ramas del repositorio. También descarga todos los commits y archivos requeridos del otro repositorio.
i. $ git branch –r – nos ayuda a mostrar las ramas locales y remotas.
j. $ git fetch <remote> <branch> -Solo recupera la rama especificada.
k. $ git fetch –all – Recupera todos los repositorios remotos registrados y sus ramas.
l. $ git merge origin/main - Las ramas origin/main y principal ahora apuntarán a la misma confirmación y estarán sincronizadas con los desarrollos de nivel superior.
m. $ git push - se usa para cargar contenido del repositorio local a un repositorio remoto. Se usa sobre todo para publicar y cargar cambios locales a un repositorio central. Después de modificar el repositorio local, se ejecuta un envío para compartir las modificaciones con los miembros remotos del equipo
n. $ git pull se emplea para extraer y descargar contenido desde un repositorio remoto y actualizar al instante el repositorio local para reflejar ese contenido. La fusión de cambios remotos de nivel superior en tu repositorio local es una tarea habitual de los flujos de trabajo de colaboración basados en Git. El comando git pull es, en realidad, una combinación de dos comandos, git fetch seguido de git merge. En la primera etapa de la operación git pull ejecutará un git fetch en la rama local a la que apunta HEAD. Una vez descargado el contenido, git pull entrará en un flujo de trabajo de fusión. Se creará una nueva confirmación de fusión y se actualizará HEAD para que apunte a la nueva confirmación.

Singleton: ChocolateBoiler

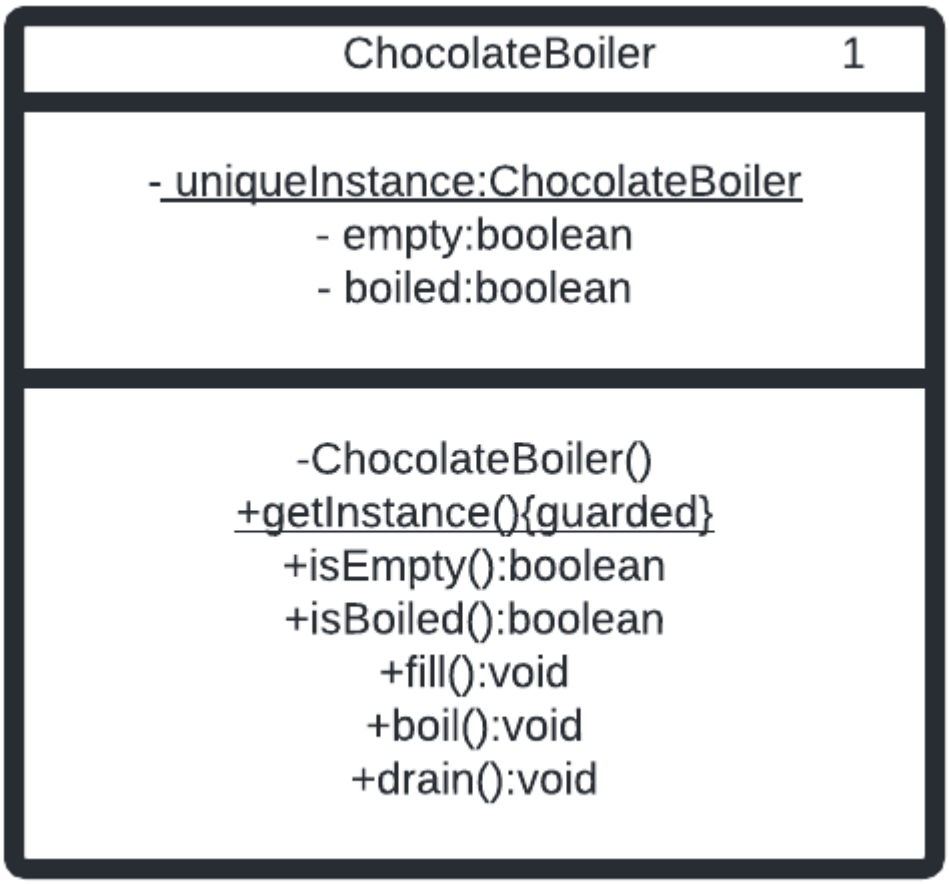
¿A qué da solución este Singleton?

En una fábrica de chocolate se requiere una caldera que tome el chocolate y la leche, llevarlos a ebullición y vaciar la mezcla para que pasen a la siguiente fase de elaboración.

Por lo que es importante considerar que:

- Debe tener tres funciones clave de llenado, calentamiento a ebullición y drenado.
- Debe ser único, no se debe repetir ni en otro hilo

De acuerdo con los puntos anteriores se definió que la clase ChocolateBoiler, debía seguir el patrón Singleton, por lo que tendría 1 atributo de clase y 2 variables de instancia, para la instancia única y sus estados de vacío y hervido, como banderas para el proceso. Un constructor privado



Herencia y Polimorfismo