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){` `System.out.println("Hello");`	
15		✓	✓	01 - Java Basics	Very Easy		

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 s1 = new A(1);`	

View Questions | Pause Test | Evaluate Test

Evidencia “Working with Java Data Types”

1.

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.

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**
    a. `$ 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.
    b. `$ 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
    c. `$ 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.
    d. `$ 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**
    a. 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.
    b. `$ git clone <repo> <directory>` - Clona el repositorio ubicado en ＜repo＞ en la carpeta llamada ~＜directory＞! en la máquina local.
    c. `$ git clone --branch <tag> <repo>` - Clona el repositorio ubicado en ＜repo ＞ y clona solamente la referencia para ＜tag＞
    d. `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.

e. `$ 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.

f.

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 diff
    a.
6. Git stash
    a.
7. .gitignore