Evolución de Java

Java es un lenguaje con características de programación orientado a objetos y procedurales, fue creado por James Gosling, Mike Sheridan y Patrick Naughton en Sun Microsystems en la década de 1990. Fue diseñado con la intención de ser una plataforma independiente, lo que significa que las aplicaciones escritas en Java podrían ejecutarse en cualquier sistema que tuviera una Máquina Virtual Java (JVM).

Java ha experimentado una evolución constante a lo largo de las décadas, pasando de un lenguaje inicialmente simple a una plataforma de desarrollo poderosa y versátil. La versión 8 marcó un cambio significativo con la introducción de características modernas que han influido en la forma en que se escriben aplicaciones Java. A continuación se muestran características importantes de las versiones de java a lo largo del tiempo

La versión inicial de Java, conocida como Java 1.0, se lanzó en 1996. Esta versión introdujo características básicas, como la gestión de objetos y la capacidad de ejecutar aplicaciones en múltiples plataformas. La verdadera innovación de Java radicaba en su capacidad para proporcionar portabilidad a través de la JVM.

Java 1.1, lanzada en 1997, trajo consigo mejoras significativas en rendimiento y estabilidad. También se introdujeron las "clases internas" en esta versión, lo que permitió una mejor organización del código.

Java 1.2, o Java 2, lanzada en 1998, fue un gran salto en la evolución de Java. Esta versión incluyó la famosa Java 2 Platform, que se dividió en tres ediciones: Java 2 Standard Edition (J2SE), Java 2 Enterprise Edition (J2EE) y Java 2 Micro Edition (J2ME). J2SE trajo consigo la API Swing para interfaces gráficas de usuario, lo que permitió un desarrollo de aplicaciones más atractivas visualmente.

Con Java 1.3 (también conocida como J2SE 1.3), lanzada en el año 2000, se introdujo la tecnología HotSpot, un compilador Just-In-Time que mejoró significativamente el rendimiento de las aplicaciones Java.

La versión 1.4 (J2SE 1.4) de Java, lanzada en 2002, introdujo numerosas mejoras en la plataforma. Entre las características destacadas se incluyen las assertions, la API de Logging y la integración de Java Web Start para la distribución de aplicaciones.

En 2004, se lanzó Java 5 (también conocida como J2SE 5 o Java 1.5) con una característica crucial: las anotaciones. Además, se introdujeron enumeraciones y mejoras en el manejo de excepciones.

Java 6, o J2SE 6, lanzada en 2006, se centró en el rendimiento y la optimización del lenguaje. Además, incluyó mejoras en la gestión de memoria y una mayor integración de XML.

La versión 7 de Java (también conocida como Java 7 o J2SE 7), lanzada en 2011, introdujo características como el manejo de múltiples excepciones y el soporte para tipos numéricos binarios.

Java 8 se lanzó en 2014. Esta versión fue un hito importante en la historia de Java debido a la introducción de las expresiones lambda, Streams API y la nueva fecha y hora API. Las expresiones lambda permitieron una programación más funcional, mientras que Streams API simplificó la manipulación de colecciones de datos.

GIT

Git es un sistema de control de versiones distribuido creado por Linus Torvalds en 2005. Se desarrolló para gestionar eficientemente el control de versiones de proyectos de software. Lo que distingue a Git de otros sistemas de control de versiones es su enfoque en la velocidad, la sencillez y la escalabilidad.

En Git, cada desarrollador tiene una copia completa del repositorio en su sistema, lo que significa que pueden trabajar de manera aislada en su propia rama de desarrollo y fusionar los cambios de manera controlada. Esto facilita la colaboración y permite el seguimiento de múltiples ramas de desarrollo al mismo tiempo.

Comandos básicos en GIT

Los siguientes comandos de git son un buen punto de partida para comenzar a trabajar con un controlador de versiones y colaboración de proyectos

1. git init: Este comando inicia un nuevo repositorio Git en un directorio local. Una vez ejecutado, se creará una carpeta oculta llamada ".git" que almacena toda la información relacionada con el repositorio.
2. git clone [URL]: Utiliza este comando para copiar un repositorio remoto en tu sistema local. Esto se hace generalmente al principio del desarrollo de un proyecto.
3. git add [archivo]: Agrega cambios en un archivo específico a la "zona de preparación", lo que significa que están listos para ser confirmados en el repositorio.
4. git commit -m "Mensaje del commit": Realiza un commit con los cambios en la zona de preparación. El mensaje es una descripción breve de los cambios realizados en este commit.
5. git status: Muestra el estado de los archivos en tu repositorio local. Indica los archivos modificados, agregados o no rastreados.
6. git log: Muestra un registro de todos los commits en el repositorio, incluyendo detalles como el autor, la fecha y el mensaje del commit.
7. git pull: Actualiza tu repositorio local con los cambios del repositorio remoto. Es útil para mantener sincronizados los cambios realizados por otros colaboradores.
8. git push: Sube tus commits locales al repositorio remoto. Esto comparte tus cambios con otros colaboradores.
9. git branch: Muestra una lista de las ramas en tu repositorio local. La rama actual se resalta con un asterisco (*).
10. git checkout [nombre de la rama]: Cambia a una rama diferente en tu repositorio local. Esto te permite trabajar en una rama específica.

Trabajando con repositorios en Github (Branches, Merge, Conflicts)

Trabajar con repositorios en GitHub (ramas, fusiones y conflictos) es esencial para colaborar en proyectos de desarrollo de software.

Branches en GitHub

Las ramas en GitHub son copias independientes de un proyecto en un estado específico. Se utilizan para trabajar en nuevas características, solucionar problemas o realizar modificaciones sin afectar la rama principal del proyecto, generalmente denominada "main" o "master". Algunos comandos y acciones relacionados con las ramas en GitHub incluyen:

- Crear una rama: Puedes crear una nueva rama a través de la interfaz web de GitHub o utilizando el comando git branch [nombre de la rama] en tu repositorio local.

- Cambiar a una rama: Puedes cambiar de rama localmente con git checkout [nombre de la rama]. En GitHub, puedes seleccionar la rama en la que deseas trabajar en el menú desplegable.

Merge en GitHub

Las fusiones en GitHub se utilizan para combinar los cambios realizados en una rama en otra, como unir una rama de desarrollo a la rama principal del proyecto. Algunos aspectos clave de las fusiones en GitHub incluyen:

-Fusionar ramas:En la interfaz de GitHub, puedes iniciar una solicitud de extracción (Pull Request) para fusionar cambios de una rama en otra. Esto permite a otros colaboradores revisar los cambios antes de la fusión.

-Resolución de conflictos: Si hay conflictos entre los cambios en la rama de destino y la rama que estás fusionando, GitHub te guiará para resolver estos conflictos. Debes editar manualmente los archivos en conflicto y luego confirmar los cambios.

-Eliminar una rama: Después de una fusión exitosa, puedes eliminar la rama que ya no necesitas. Esto ayuda a mantener un repositorio limpio.

Conflicts en GitHub

Los conflictos en GitHub pueden surgir cuando se intenta fusionar cambios en una rama que han afectado las mismas líneas de código que se han modificado en la rama de destino. Aquí hay algunos puntos clave sobre cómo manejar conflictos en GitHub:

-Resolver conflictos: GitHub te mostrará las áreas en conflicto en un archivo. Debes editar el archivo para conservar las partes que desees mantener y eliminar las que no necesitas. Luego, debes marcar el conflicto como resuelto y confirmar los cambios.

-Commit de resolución: Una vez que hayas resuelto los conflictos, debes realizar un commit para registrar las modificaciones que has realizado en la resolución.

-Continuar con la fusión: Después de resolver los conflictos y realizar un commit de resolución, puedes continuar con el proceso de fusión. En la interfaz de GitHub, puedes marcar que los conflictos se han resuelto y completar la fusión.

Como implementar scrum

Hacer planes es útil; seguirlos ciegamente, no. Resulta tentador trazar un sinfín de gráficas, exponer a la vista de todos lo que debe hacerse en un gran proyecto. Pero cuando los planes detallados se confrontan con la realidad se vienen abajo. En este contexto surge Scrum.

Scrum es un marco de trabajo ágil ampliamente utilizado en el desarrollo de software y proyectos, abraza esta idea fundamental. En lugar de adherirse rígidamente a planes a largo plazo, Scrum promueve la flexibilidad, la inspección, la adaptación y la entrega de valor constante.

Flexibilidad y Adaptación Continua

En Scrum, la flexibilidad es clave. En lugar de enfocarse en planes detallados y a largo plazo, los equipos Scrum trabajan en iteraciones más cortas llamadas "sprints", que suelen tener una duración de 2 a 4 semanas. Durante cada sprint, el equipo se enfoca en un conjunto específico de tareas, lo que permite una respuesta ágil a los cambios y nuevas ideas.

La clave es abrazar la idea de que el cambio es inevitable en cualquier proyecto. Scrum incorpora la flexibilidad como un supuesto fundamental, permitiendo a los equipos adaptarse a las necesidades cambiantes del negocio o los clientes. Esto significa que los planes pueden evolucionar a medida que se descubren nuevas ideas o surgen requerimientos inesperados.

Inspección y Adaptación

Scrum fomenta la inspección constante de los procesos y resultados. Al final de cada sprint, se realiza una revisión en la que se muestra el trabajo completado y se obtiene retroalimentación de los stakeholders. Esto permite ajustar el rumbo del proyecto según las necesidades y expectativas reales.

La inspección también se lleva a cabo durante las reuniones diarias de Scrum, donde el equipo evalúa su progreso y ajusta su enfoque según sea necesario. La adaptación constante garantiza que el proyecto se mantenga alineado con los objetivos y que se puedan tomar decisiones informadas.

Aprender de los Errores Temprano

Scrum promueve la idea de "equivocarse pronto" para corregir con anticipación. En lugar de perder tiempo en la creación de documentos y procedimientos detallados, se enfoca en entregar valor rápidamente y en incrementos pequeños. Esto permite identificar errores y ajustar el rumbo de manera temprana, lo que ahorra tiempo y recursos.

Los sprints cortos y la entrega de valor en cada iteración permiten obtener una retroalimentación valiosa de los usuarios. Esto no solo mejora la calidad del producto final, sino que también evita la inversión en esfuerzos inútiles.

Valor Constante

Scrum se centra en la entrega de valor constante. En lugar de esperar hasta el final del proyecto para ver resultados, Scrum permite que los stakeholders reciban valor tangible en cada sprint. Esto fortalece la confianza y la colaboración con los interesados y asegura que el proyecto siga siendo relevante y beneficioso.

Al seguir estos principios, los equipos Scrum pueden gestionar proyectos de manera más efectiva, responder a los cambios y entregar productos de alta calidad que satisfacen las necesidades reales de los usuarios. En un mundo empresarial en constante evolución, Scrum es una herramienta poderosa para el éxito.