Academia Java by Xideral

Examen Segunda Semana

Asael Marcial Grajales

Tabla de contenido

SCRUM

Que es SCRUM

Scrum es un marco de trabajo ágil para la gestión de proyectos. Fue desarrollado originalmente para el desarrollo de software, pero se ha utilizado con éxito en una variedad de ámbitos, incluyendo marketing, finanzas y operaciones. Scrum se basa en la colaboración, la flexibilidad y la adaptabilidad para alcanzar metas y objetivos de proyectos de manera eficiente. Scrum se divide en tres roles principales: el Product Owner, el Scrum Master y el equipo de desarrollo. Estos roles trabajan juntos para planificar, ejecutar y evaluar el progreso del proyecto en Sprints cortos, generalmente de 1 a 4 semanas de duración.

Pasos para implementar SCRUM

1. **Elige un responsable del producto.** Este individuo es quien posee la visión de lo que vas a hacer, producir o lograr. Toma en cuenta los riesgos y recompensas, qué es posible, qué puede hacerse y qué es lo que le apasiona.

2. **Selecciona un equipo**. Este equipo debe contar con todas las habilidades necesarias para tomar la visión de los responsables del producto y hacerla realidad. Los equipos deben ser pequeños, de tres a nueve personas por regla general.

3. **Elige un Scrum Master**. Ésta es la persona que capacitará al resto del equipo en el enfoque Scrum y que ayudará al equipo a eliminar todo lo que lo atrasa.

4. **Crea y prioriza una bitácora del producto**. Se trata de una lista de alto nivel de todo lo que debe hacerse para volver realidad la visión. Esta bitácora existe y evoluciona durante el periodo de vida del producto; es la guía de caminos hacia éste. El responsable del producto debe consultar tanto a todos los interesados como al equipo para cerciorarse de que representa lo que la gente necesita y lo que se puede hacer.

5. **Afina y estima la bitácora del producto**. Es crucial que la gente que realmente se hará cargo de los elementos de la bitácora del producto estime cuánto esfuerzo implicarán. El equipo debe examinar cada elemento de la bitácora y ver si, en efecto, es viable. Cada elemento debe poder mostrarse, demostrarse y (es de esperar) entregarse. No calcules la bitácora en horas, porque la gente es pésima para esto.

6. **Planeación del sprint**. Ésta es la primera de las reuniones de Scrum. El equipo, el Scrum Master y el responsable del producto se sientan a planear el sprint. Los sprints son siempre

de extensión fija, inferior a un mes. La mayoría de la gente ejecuta en la actualidad uno o dos sprints semanales. El equipo examina el inicio de la bitácora y pronostica cuánto puede llevar a cabo en este sprint. Si el equipo ha pasado por varios sprints, debe considerar el número de puntos que acumuló en el más reciente. Este número se conoce como velocidad del equipo. El Scrum Master y el equipo deben tratar de aumentar ese número en cada sprint. Ésta es otra oportunidad para que el equipo y el responsable del producto confirmen que todos comprenden a la perfección cómo esos elementos cumplirán la visión. Durante esta reunión todos deben acordar asimismo una meta de sprint, que todos han de cumplir en este sprint. Uno de los pilares de Scrum es que una vez que el equipo se compromete con lo que cree que puede terminar en un sprint, eso se queda ahí. No puede cambiar ni crecer. El equipo debe ser capaz de trabajar en forma autónoma a lo largo del sprint para terminar lo que pronosticó que podía hacer.

7. **Vuelve visible el trabajo.** La forma más común de hacerlo en Scrum es crear una tabla de Scrum con tres columnas: Pendiente, En proceso y Terminado. Notas adhesivas representan los elementos por llevar a cabo y el equipo avanza por la tabla conforme los va concluyendo, uno por uno. Otra manera de volver visible el trabajo es crear un diagrama de finalización. En un eje aparece el número de puntos que el equipo introdujo en el sprint y en el otro el número de días. Cada día, el Scrum Master suma el número de puntos completados y los grafica en el diagrama de finalización. Idealmente, habrá una pendiente descendente que conduzca a cero puntos para el último día del sprint.

8. **Parada diaria o Scrum diario**. Éste es el pulso de Scrum. Cada día, a la misma hora, durante no más de quince minutos, el equipo y el Scrum Master se reúnen y contestan tres preguntas:

1. ¿Qué hiciste ayer para ayudar al equipo a terminar el sprint?
2. ¿Qué harás hoy para ayudar al equipo a terminar el sprint?
3. ¿Algún obstáculo te impide o impide al equipo cumplir la meta del sprint?

Eso es todo, en eso consiste la reunión. Si se prolonga más de quince minutos lo estás haciendo mal. Lo que esto hace es ayudar al equipo a saber exactamente dónde se encuentra todo en el curso de un sprint. El Scrum Master se encarga de eliminar los obstáculos, o impedimentos, contra el progreso del equipo.

9. **Revisión del sprint o demostración del sprint.** Ésta es la reunión en la que el equipo muestra lo que hizo durante el sprint. Todos pueden asistir, no sólo el responsable del producto, el Scrum Master y el equipo, sino también los demás interesados, la dirección, clientes, quien sea. Ésta es una reunión abierta en la que el equipo hace una demostración de lo que pudo llevar a Terminado durante el sprint. El equipo debe mostrar únicamente lo que satisface la definición de Terminado, lo total y completamente concluido y que puede entregarse sin trabajo adicional. Esto puede no ser un producto terminado, pero sí una función concluida de uno de ellos.

10. **Retrospectiva del sprint.** Una vez que el equipo ha mostrado lo que logró en el sprint más reciente, piensa en qué marchó bien, qué pudo haber marchado mejor y qué puede mejorar en el siguiente sprint. Para ser eficaz, esta reunión requiere cierto grado de madurez emocional y una atmósfera de confianza. La clave es que no se trata de buscar a quién culpar; lo que se juzga es el proceso. Es crucial que la gente, como equipo, asuma la responsabilidad de su proceso y de sus resultados y busque soluciones también como

equipo. Al mismo tiempo, debe tener fortaleza para tocar los temas que le incomodan de un modo orientado a la solución, no acusatorio. Y el resto del equipo ha de tener la madurez de oír la realimentación, aceptarla y buscar una solución, no ponerse a la defensiva. Al final de la reunión, el equipo y el Scrum Master deben acordar una mejora al proceso que implementarán en el siguiente sprint. Esa mejora al proceso, también llamada kaizen, debe incorporarse en la bitácora del sprint siguiente, con pruebas de aceptación. De esta manera, el equipo podrá ver fácilmente si en verdad implementó la mejora y qué efecto tuvo ésta en la velocidad.

11. Comienza de inmediato el ciclo del siguiente sprint, tomando en cuenta la experiencia del equipo con los impedimentos y mejoras del proceso.

Conceptos Básicos de GIT

¿Qué es un branch?

El objetivo de utilizar ramas es identificar tareas de desarrollo sin afectar otras ramas en el repositorio, permitiendo desarrollar características, corregir errores, o experimentar nuevas ideas en un área contenida de tu repositorio.

Al crear un repositorio, GitHub automáticamente crea una rama. Esta es la rama preterminada a la cual se le asigna el nombre *main.* Cada repositorio tiene una rama preterminada y puede tener otras ramas, las cuales se pueden crear a partir de la rama *main* o de otra rama existente, como se puede ver en la siguiente imagen.

El comando git branch te permite crear, enumerar y eliminar ramas, así como cambiar su nombre. No te permite cambiar entre ramas o volver a unir un historial bifurcado. Por este motivo, git branch está estrechamente integrado con los comandos git checkout y git merge.

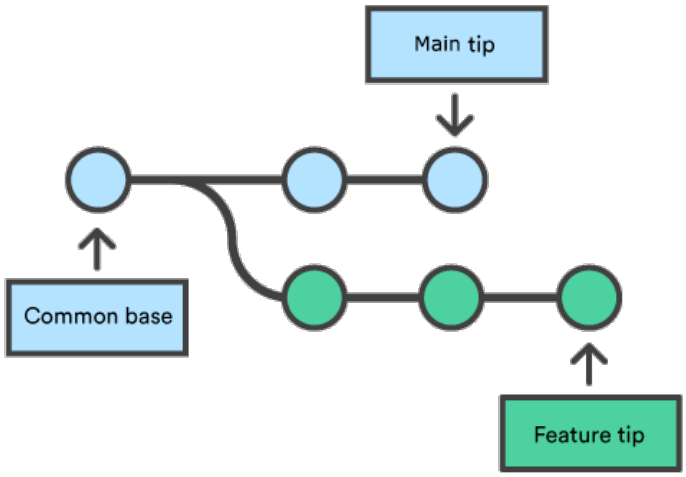
Se puede abrir una solicitud de incorporación de cambios para combinar los cambios de la rama actual a otra.

Es importante mencionar que, para crear una rama, abrir una solicitud de extracción o eliminar y restablecer ramas en una solicitud de extracción, se debe tener acceso de escritura.

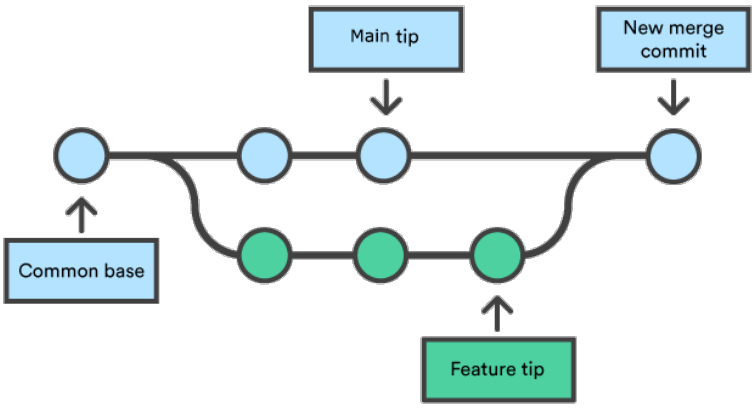
¿Qué es un merge?

El comando git merge permite tomar las diferentes líneas de desarrollo creadas por git branch e integrarlas en una sola rama, combinando varias secuencias de commits en un historial unificado.

Como se puede ver en la siguiente imagen, git merge toma dos punteros de commit, normalmente los extremos de las ramas, y encuentra un commit base común entre ellos, una vez encontrado este commit base, crea un nuevo "merge commit" que combina los cambios de cada secuencia de confirmación de fusión puesta en cola.



Al invocar este comando, la rama de fusión especificada se fusionará con la rama actual, la cual asumiremos que es la main, resultando en la siguiente imagen.



¿Qué es un conflict?

Un conflicto surge cuando dos ramas independientes han editado la misma línea de un archivo o cuando un archivo se ha eliminado en una rama, pero editado en la otra debido a que Git no puede determinar automáticamente cuál es la correcta.

Git tiene muchas herramientas de línea de comandos para resolver los conflictos de fusión, como git log, git reset, git status, git checkout y git reset. Además, existen herramientas de terceros que ofrecen funciones de soporte agilizadas para conflictos de fusión.

Modelo – Vista – Controlador (MVC)

¿Qué es MVC?

MVC (Modelo-Vista-Controlador) es un patrón de arquitectura de software que se utiliza para separar la lógica de negocio de una aplicación de su interfaz de usuario y su mecanismo de control.

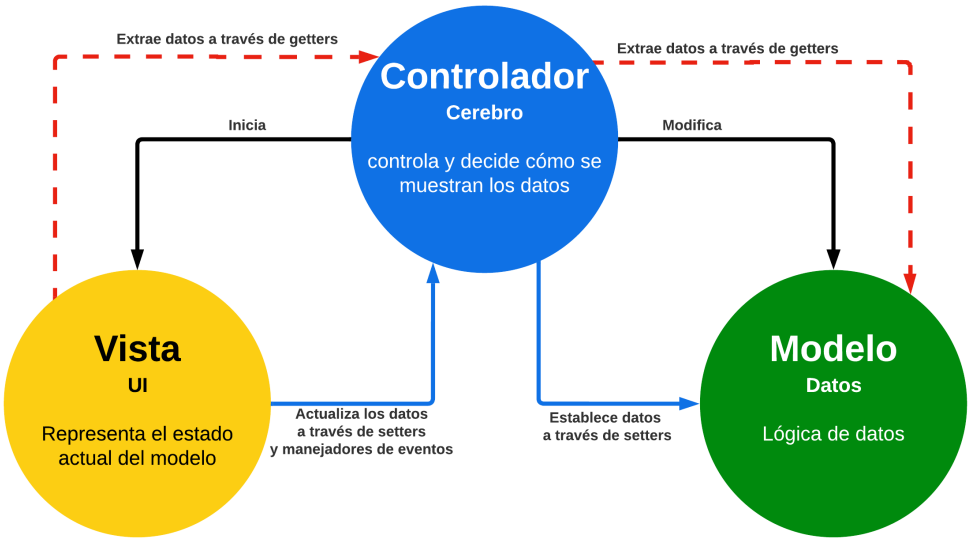
Modelo: Representa la lógica de negocio y la información de la aplicación. Es la fuente de datos y se encarga de actualizar la vista cuando cambian los datos.

Vista: Es la interfaz gráfica de usuario (GUI) que muestra la información al usuario y recolecta la entrada del usuario.

Controlador: Es el intermediario entre el modelo y la vista. Recibe las acciones del usuario desde la vista y actualiza el modelo con esa información. También actualiza la vista cuando el modelo cambia.

El objetivo de MVC es separar los diferentes aspectos de una aplicación para hacerla más fácil de mantener y escalar. Al separar la lógica de negocio del diseño de interfaz, es más fácil hacer cambios en una parte de la aplicación sin afectar al resto de la misma.

Diagrama

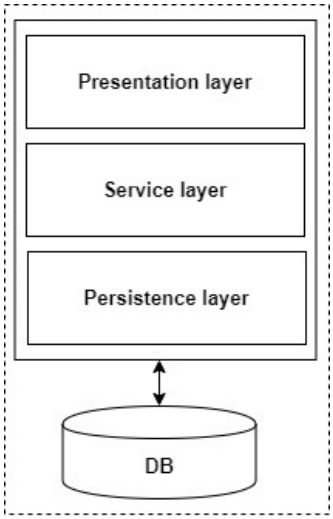


Arquitectura Monolítica y Microservicios

Arquitectura monolítica

Una aplicación monolítica es una aplicación que se ejecuta en un único proceso y que tiene todas las funcionalidades y componentes necesarios para funcionar en un único paquete. Estas aplicaciones son más fáciles de desarrollar y probar, pero suelen ser menos escalables y flexibles que las aplicaciones basadas en microservicios.

Diagrama

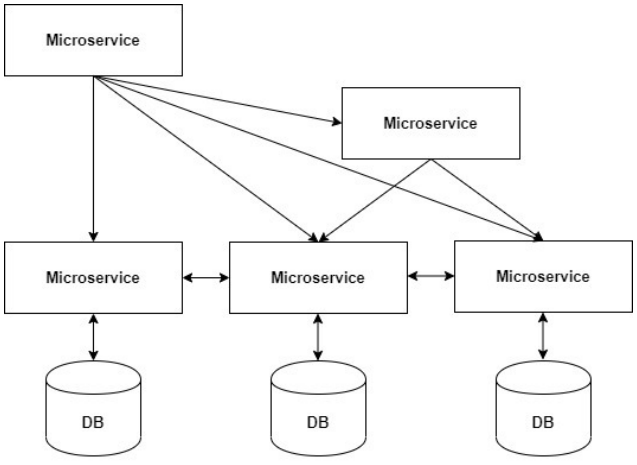


Monolithic architecture

Arquitectura de microservicios

Los microservicios son una arquitectura de software en la que una aplicación se divide en pequeños servicios independientes que se comunican entre sí mediante una interfaz establecida. Cada servicio es responsable de una funcionalidad específica y se despliega y actualiza de forma independiente. Esto permite que el equipo de desarrollo trabaje de forma más colaborativa y aumenta la escalabilidad y la flexibilidad de la aplicación.

Diagrama



Microservices architecture

Excepciones

¿Qué son?

Una excepción es la indicación de que se produjo un error en el programa. Las excepciones, como su nombre lo indica, se producen cuando la ejecución de un método no termina correctamente, sino que termina de manera excepcional como consecuencia de una situación no esperada.

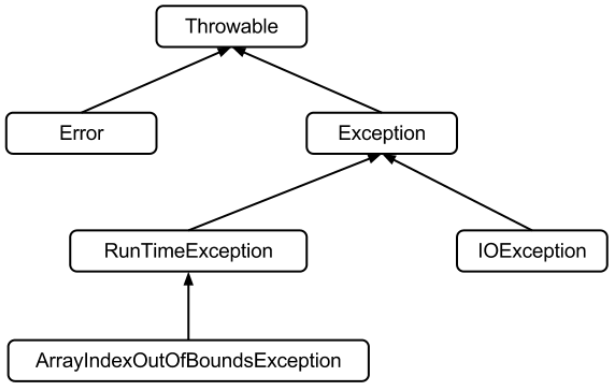
Ocurren cuando se produce una situación anormal durante la ejecución de un programa (por ejemplo, se accede a un objeto que no ha sido inicializado o tratamos de acceder a una posición inválida en un vector), si no manejamos de manera adecuada el error que se produce, el programa va a terminar abruptamente su ejecución.

Cuando durante la ejecución de un método el computador detecta un error, crea un objeto de una clase especial para representarlo (de la clase Exception en Java), el cual incluye toda la información del problema, tal como el punto del programa donde se produjo, la causa del error, etc. Luego, "dispara" o "lanza" dicho objeto (throw en inglés), con la esperanza de que alguien lo atrape y decida como recuperarse del error. Si nadie lo atrapa, el programa termina, y en la consola de ejecución aparecerá toda la información contenida en el objeto que representaba el error.

Tipos de excepciones

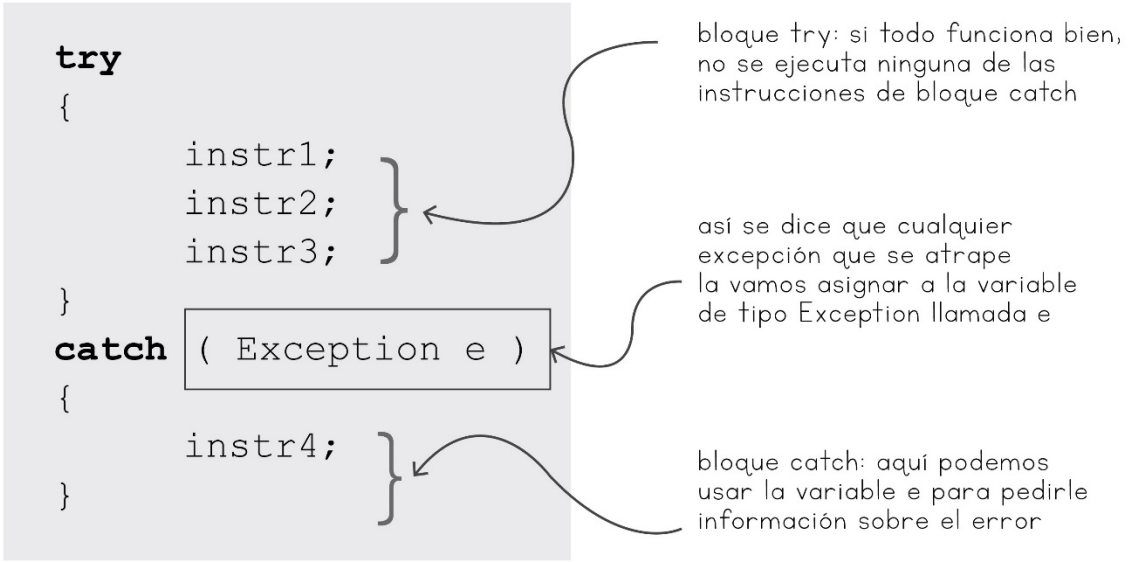
Existe una jerarquía de clases que representan errores en Java. En la siguiente imagen podemos ver:

- Excepciones irrecuperables: Hijas de Error. Son errores de la propia máquina virtual de Java.
- Excepciones que NO es necesario gestionar: Hijas de RunTimeException. Son excepciones muy comunes, por ejemplo NullPointerException, ArrayIndexOutOfBoundsException.
- Excepciones que es necesario gestionar: Hijas de Exception. Todas las demás, por ejemplo IOException.



Try catch

La instrucción try-catch de Java tiene la estructura que se muestra en la siguiente imagen.



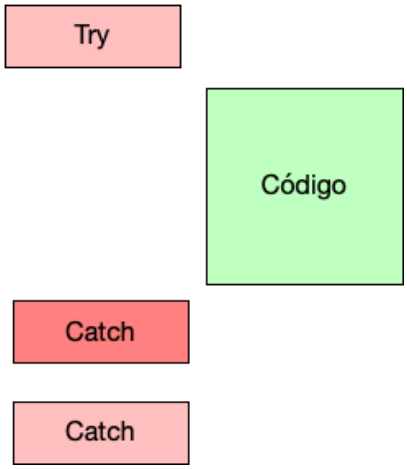
En la instrucción try-catch hay dos bloques de instrucciones, con los siguientes objetivos:

1. Delimitar la porción de código dentro de un método en el que necesitamos desviar el control si una excepción ocurre allí (la parte try). Si se dispara una excepción en alguna de las instrucciones del bloque try, la ejecución del programa pasa inmediatamente a las instrucciones del bloque catch. Si no se dispara ninguna excepción en las instrucciones del bloque try, la ejecución continúa después del bloque catch.
2. Definir el código que manejará el error o atrapará la excepción (la parte catch).

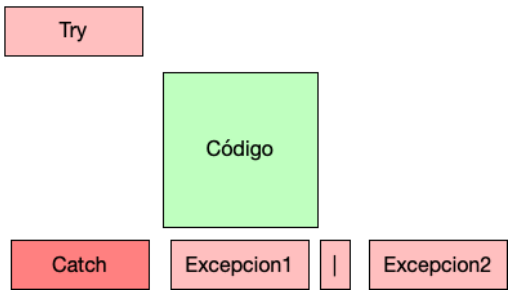
Además, existe la cláusula finally, la cual contiene sentencias a ejecutarse después de que las cláusulas try y catch se ejecuten, sin importar si una excepción es o no lanzada.

Multi-catch

El concepto de Java MultiCatch Block es uno de los conceptos más clásicos de manejo de excepciones. Es útil al tener la necesidad de capturar varias excepciones problemáticas y su estructura se puede observar en la siguiente imagen.



A partir de Java 7 se permite compactar este tipo de manejo de excepciones en una única línea agrupando múltiples excepciones de forma conjunta ya que asumimos que su tratamiento será idéntico.



Try-with-resources

Es una de las varias declaraciones try en Java, destinada a liberar a los desarrolladores de la obligación de liberar recursos utilizados en un bloque try.

Inicialmente se introdujo en Java 7 con la idea principal de que el desarrollador no tuviera que preocuparse por la administración de recursos, eliminando la necesidad de bloques finally, que los desarrolladores solo usaban para cerrar recursos en la práctica.

Además, el código que usa try-with-resources es a menudo más limpio y más legible, por lo tanto, hace que el código sea más fácil de administrar, especialmente cuando se trata de muchos bloques try.

Por ejemplo:

```
try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {

        writer.write(str); // do something with the file we've opened
```

```
}
catch (IOException e) {
    // handle the exception
}
```

Java entiende el código como “Los recursos abiertos entre paréntesis después de la instrucción try solo serán necesarios aquí y ahora. Llamará a sus métodos close() tan pronto como termine el trabajo en el bloque try. Si se lanza una excepción mientras está en el bloque try, cerrará esos recursos de todos modos”.

Collections

¿Qué son?

En Java, existen varios tipos de colecciones (también conocidas como "contenedores") que se pueden utilizar para almacenar y acceder a datos de manera eficiente. Algunos de los tipos más comunes de colecciones en Java son:

Tipos

ArrayList: Es una implementación de la interfaz List que utiliza un array interno para almacenar los datos. Es similar a un array dinámico, ya que permite agregar y eliminar elementos de forma dinámica.

LinkedList: Es una implementación de la interfaz List que utiliza una lista enlazada para almacenar los datos. Es similar a una lista doblemente enlazada, y es adecuada para insertar y eliminar elementos en cualquier posición.

HashSet: Es una implementación de la interfaz Set que utiliza una tabla hash para almacenar los datos. Es adecuada para almacenar conjuntos de elementos únicos y para realizar búsquedas rápidas.

TreeSet: Es una implementación de la interfaz SortedSet que utiliza un árbol rojo-negro para almacenar los datos. Es adecuada para almacenar conjuntos de elementos ordenados y para realizar búsquedas rápidas.

HashMap: Es una implementación de la interfaz Map que utiliza una tabla hash para almacenar los datos en forma de pares clave-valor. Es adecuada para almacenar colecciones de objetos y para realizar búsquedas rápidas.

TreeMap: Es una implementación de la interfaz SortedMap que utiliza un árbol rojo-negro para almacenar los datos en forma de pares clave-valor. Es adecuada para almacenar colecciones de objetos ordenadas y para realizar búsquedas rápidas.

Es importante mencionar que existen otras colecciones y estructuras de datos en Java como: Queue, Stack, Deque, Vector, entre otras.