Preguntas de la semana 2

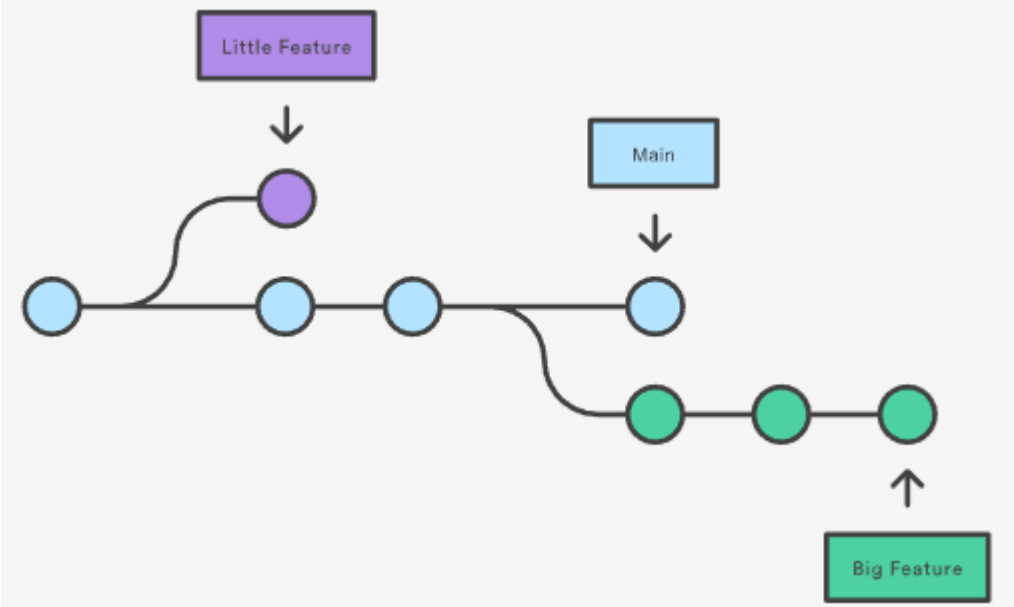
Clever Armando León Valdez

Índice

Git Hub

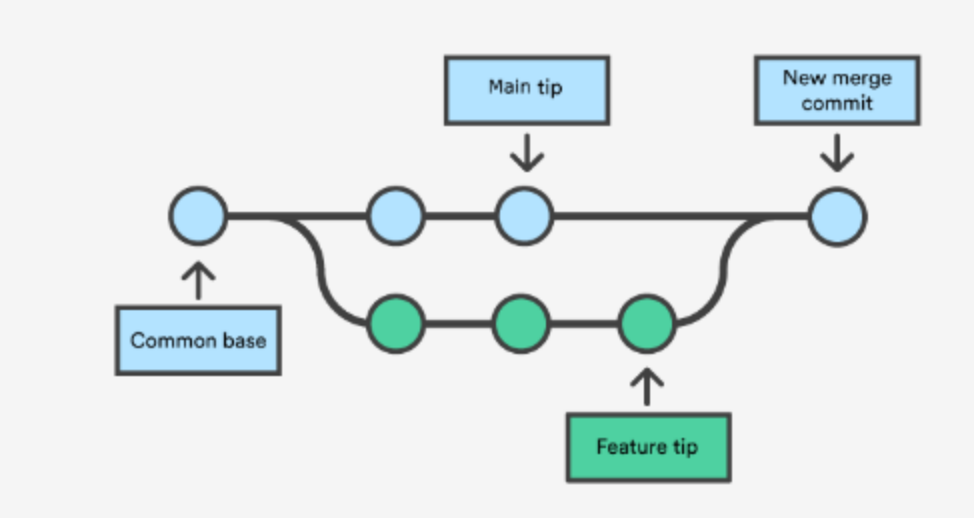
Branch

- Una rama representa una línea de desarrollo independiente.
- Todos los commits se hacen sobre una rama, que por defecto será la MASTER.
- Su principal función es crear diferentes versiones de un proyecto, para que así se pueda trabajar sin afectar el flujo principal de trabajo.
- Git branch “Nombre de la rama” => Crear rama y copia el commit a esta rama.
- Git checkout “Nombre de la rama” => Nos cambia de rama
- Git Branch => Nos muestra las ramas que existen y con un * la en la que esta.
- Git push origin --delete “Nombre de la rama” => elimina rama remota



Merge

- Va a ocurrir en la rama ubicada generando que el ultimo commit se esa rama sea el principal
- Frecuentemente usado para fusionar 2 ramas.
- Ofrece la posibilidad de realizar fusiones parciales o seleccionar solo ciertos cambios para ser aplicados.
- Puede detectar y resolver automáticamente conflictos entre las diferentes versiones de los archivos.



Otros comandos

- Git init – “Initialize” Inicializar un control de versiones.
- Git clone ¨dir del repositorio¨ - clona un repositorio a tu disco local
- Git status – ver el estado de nuestros archivos
- Git add – añade el archivo a staging
- Git commit – guarda los archivos en el repositorio local
    - Agrega un mensaje con -m ¨mensaje¨
- Git reset head – quita archivos de staged
- Git show – Muestra los cambios que han existido en un archivo
- Git diff - muestra las diferencias entre una versión y otra

Resolver conflictos

Los merge conflict se producen cuando otra persona edita en la misma rama las mismas líneas de código que tú editas. Esto pasa por no bajar los últimos cambios, una mala comunicación o mala gestión del proyecto.

- Asegúrate de estar en la rama correcta antes de comenzar a resolver conflictos. Es recomendable trabajar en una rama de desarrollo antes de fusionar con la rama principal.
- Usa el comando "git pull" para sincronizar tus cambios con la rama remota. Si hay conflictos, Git te lo notificará.
- Utiliza "git status" para ver qué archivos tienen conflictos.

- Abre los archivos en conflicto en un editor de texto y busca las líneas con marcas especiales, como <<<<<<<. Estas marcas indican qué cambios de tu rama local y qué cambios de la rama remota están en conflicto.
- Decide qué cambios quieres conservar y elimina las marcas especiales y las líneas de conflicto que no quieres mantener. Asegúrate de dejar solo una versión del código.
- Usa "git add" para agregar los archivos editados a la stage area.
- Usa "git commit" para crear un commit con los cambios resueltos.
- Usa "git push" para enviar los cambios resueltos a la rama remota.
- Si es necesario, repite los pasos anteriores para resolver cualquier conflicto adicional.

Pasos para implementar Scrum

1. Elige un responsable del producto
   Es la persona que tiene una visión clara del producto, que se va a hacer, y lograr, además de que toma en cuenta los riesgos y la recompensa.

2. Selecciona un equipo
   Es el equipo de se encargará del desarrollo del proyecto, este equipo debe ser de entre 3 a 9 personas. Además, dicho equipo deberá contar con todas las habilidades necesarias para tomar la visión de los responsables del producto y hacerla realidad.

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.
   Elabora y establece un registro del ciclo de vida del producto.
   Se trata de una guia de todo lo necesario para alcanzar la visión. Este registro se mantiene y se desarrolla a lo largo del ciclo de vida del producto, y es la guía para alcanzarlo. Dicha guía es la visión final de "todo lo que el equipo puede hacer, en orden de importancia".
   Esta quia es única por lo que la persona responsable del producto debe tomar decisiones de priorización en todo el espectro.

5. Afina y estima la bitácora del producto
   Es esencial que las personas encargadas de llevar a cabo las tareas de la lista estimen cuánto trabajo implicará cada una además el equipo debe revisar cada tarea y determinar si es viable ¿Hay información suficiente para llevar a cabo el elemento? ¿Éste es lo bastante pequeño para estimarse? ¿Existe una definición de “terminado”; es decir, todos están de acuerdo en los criterios que deben cumplirse para poder decir que algo está “terminado”?
   ¿Esto crea valor visible?, estas son algunas preguntas para determinar la viabilidad de cada tarea

6. Planeación del sprint
   La primera reunión Scrum, todos se sientan y planean el sprint con duración menor a un mes. El equipo examina el inicio de la bitácora y pronostica cuanto llevara el sprint, además de que todos deberán acordar una meta a cumplir en el sprint.
   El equipo se compromete con lo que cree que puede terminar en el sprint, no puede cambiar ni aumentar la meta, además de que el equipo trabajara de forma autónoma en dicha meta.

7. Vuelve visible el trabajo
   Para mostrar de manera evidente el progreso y el estado del trabajo se creara una tabla con 3 columnas, Pendiente, En proceso y Terminado. Por medio de notas adhesivas se representan los elementos por llevar a cabo, y van avanzando en base a su estado.

8. Parada o Scrum diarios
   Es una reunión que se realiza todos los días, a la misma hora, y dura no más de 15 minutos. En esta reunión, el equipo y el Scrum Master se reúnen para responder tres preguntas clave:

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

   La idea es mantener un seguimiento diario y conciso del progreso del equipo y identificar problemas temprano. Si la reunión dura más de 15 minutos es un indicador de que algo no está funcionando correctamente. El equipo es autónomo y las tareas no son asignadas por alguien, sino el mismo equipo lo hace.

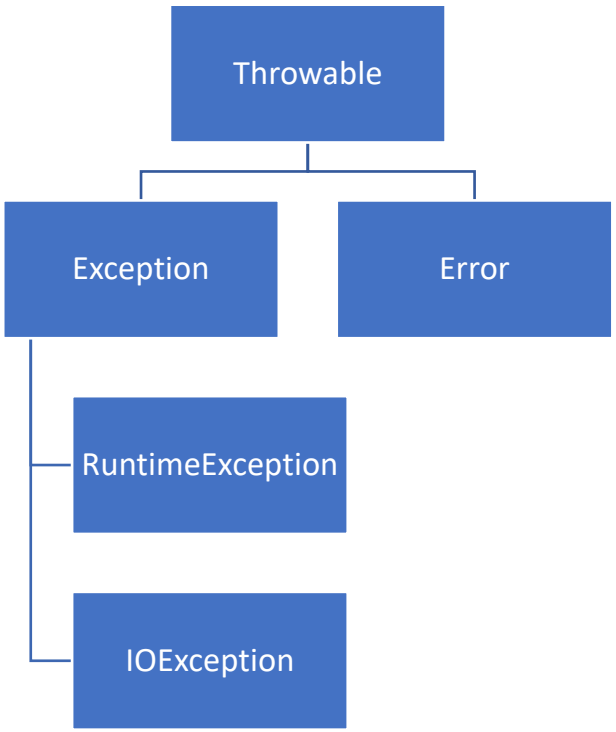
9. Revisión del sprint o demostración del sprint
   Es una reunión en la que el equipo muestra lo que ha logrado durante el sprint, esta reunión es abierta y pueden asistir todas las personas interesadas.

Es una oportunidad para que el equipo muestre lo que ha logrado durante el sprint, solo lo que se considera terminado, lo que está completo y listo para entregar. No se refiere al producto, sino de una función de este.

10. Retrospectiva del sprint
    Se debe analizar que marcho bien, que pudo haber marchado mejor, y que se va a mejorar para el siguiente sprint.
    Esta reunión necesita una atmosfera de confianza, sin juzgar o buscar culpables, algunas preguntas de ayuda son ¿Por qué tal cosa ocurrió de tal manera? ¿Por qué pasamos por alto tal otra?, ¿Qué podríamos hacer más rápido?
    Se busca una cooperación de equipo, para que este asuma la responsabilidad y busque soluciones.
    Al final, el equipo y el Scrum Master deben acordar una mejora al proceso para el siguiente sprint.

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

Explica los tipos de exections



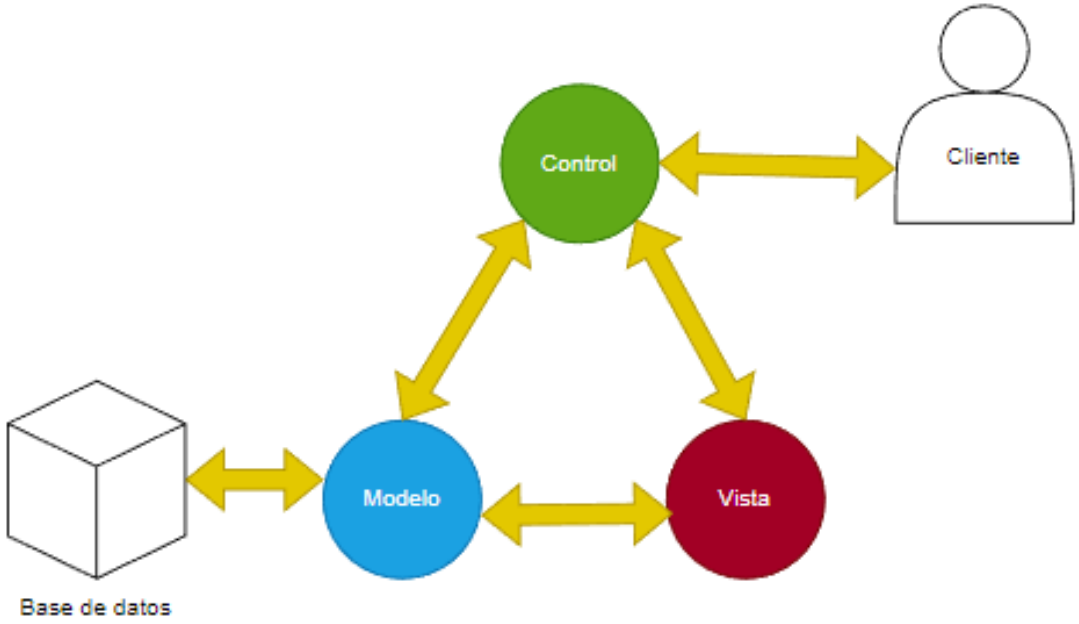
Throwable Clase base que representa todo lo que se puede “lanzar” en Java Contiene una instantánea del estado de la pila en el momento en el que se creó el objeto. Almacena un mensaje que podemos utilizar para detallar qué error se produjo. Puede tener una causa, también de tipo Throwable, que permite representar el error que causó este error.

Exception Exception y sus subclases indican situaciones que una aplicación debería tratar de forma razonable. Los dos tipos principales de excepciones son:

- RuntimeException - errores del programador, como una división por cero o el acceso fuera de los límites de un array
- IOException - errores que no puede evitar el programador, generalmente relacionados con la entrada/salida del programa.

Error Subclase de Throwable que indica problemas graves que una aplicación no debería intentar solucionar. Por ejemplo, memoria agotada, error interno de la JVM…

Explica el modelo MVC

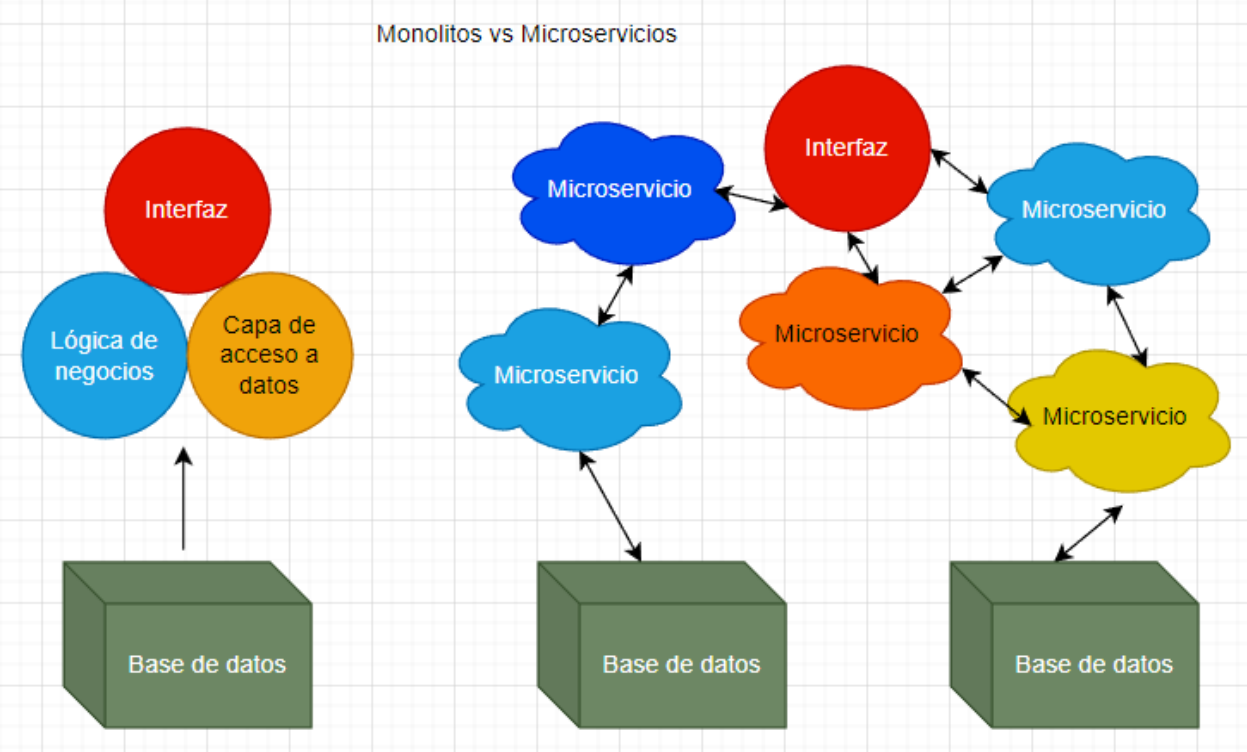


MVC es un patrón de diseño utilizado para separar el software en diferentes partes, la interfaz de usuario y la lógica de control en componentes distintos y separados.

Este patrón de diseño se utiliza para mejorar la división del trabajo y el mantenimiento del software, y existen algunas variantes como MVVM, MVP o MVW.

1. El modelo es el componente encargado de manejar los datos, responsable de acceder a la capa de almacenamiento de datos, Si el modelo es activo, también notificará a las vistas los cambios en los datos que puedan producirse a través de agentes externos.
2. El controlador es el componente encargado de recibir los eventos de entrada y contiene reglas para gestionar estos eventos. Recibe los eventos y realiza acciones en consecuencia, como llamar al modelo para obtener información o actualizar las vistas.
3. Las vistas son los componentes encargados de mostrar los datos al usuario y tienen un registro de su controlador asociado. Reciben los datos del modelo y los presentan al usuario, y también pueden ofrecer el servicio de actualización para ser invocado por el controlador o por el modelo.

Microservicios vs monolitos



Monolitos

Es una arquitectura tradicional de un software, que es independiente de otras aplicaciones. Una arquitectura monolítica es una red informática grande y única, con una base de código que aúna todos los intereses empresariales

En resumen, las ventajas de un sistema monolítico incluyen una implementación sencilla, un desarrollo más fácil, un mejor rendimiento, pruebas simplificadas y una depuración sencilla. Sin embargo, las desventajas incluyen una velocidad de desarrollo más lenta, dificultades para escalar, menor fiabilidad, barreras para la adopción de tecnología, falta de flexibilidad y dificultad para implementar cambios.

Microservicios

La arquitectura de microservicios es un método que se basa en una serie de servicios que se pueden implementar de forma independiente.

Los microservicios separan las tareas en procesos más pequeños que funcionan de manera independiente entre sí y contribuyen al conjunto global. Por ejemplo, la actualización, las pruebas, la implementación y el escalado se llevan a cabo dentro de cada servicio.

Los microservicios ofrecen una mayor agilidad en el desarrollo, escalabilidad flexible, implementación continua, fácil mantenimiento y pruebas, independencia en la implementación, flexibilidad tecnológica, alta fiabilidad y mayor satisfacción en los equipos de desarrollo. Las desventajas de los microservicios incluyen un desarrollo descontrolado, costes exponenciales de infraestructura, sobrecarga organizativa, desafíos para la depuración, falta de estandarización y dificultad en la clarificación de la propiedad de los servicios.

En resumen, un sistema monolítico es una aplicación que se construye como una sola unidad, mientras que los microservicios son una aplicación construida como un conjunto de servicios independientes que trabajan juntos.

Multicatch y trywithresources

Multicatch

Un bloque try puede ir seguido de uno o más bloques catch. Cada bloque debe contener un controlador de excepción diferente. Por lo tanto, si tiene que realizar diferentes tareas cuando ocurren diferentes excepciones, use el bloque de captura múltiple de Java.

- A la vez solo ocurre una excepción y a la vez solo se ejecuta un bloque catch.
- Todos los bloques catch deben ordenarse del más específico al más general, es decir, catch para ArithmeticException debe ir antes que catch para Exception.

```
prueba {
    int  a[]= nuevo int [ 5 ];
    a[ 5 ]= 30 / 0 ;
   }
   captura (Excepción Aritmética e)
     {
      System.out.println( "Ocurre una excepción aritmética" );
     }
   captura (ArrayIndexOutOfBoundsException e)
     {
      System.out.println( "Ocurre una excepción ArrayIndexOutOfBounds" );
     }
   captura (Excepción e)
     {
      System.out.println( "Ocurre una excepción principal" );
     }
   System.out.println( "resto del código" );
```

También se puede especificar varias excepciones separadas por el operador "|" dentro de un bloque catch. Esto permite que el código maneje de manera diferente varias excepciones diferentes en un solo bloque en lugar de tener varios bloques catch independientes.

```
try {
   // Excepción
} catch (Excepcion1 | Excepcion2 | Excepcion3 e) {
   // manejo de la excepción
}
```

Try with resources

Permite crear un recurso, como un archivo, dentro del bloque try y asegurarse de que se cierra automáticamente después de usarlo, independientemente de si se produce

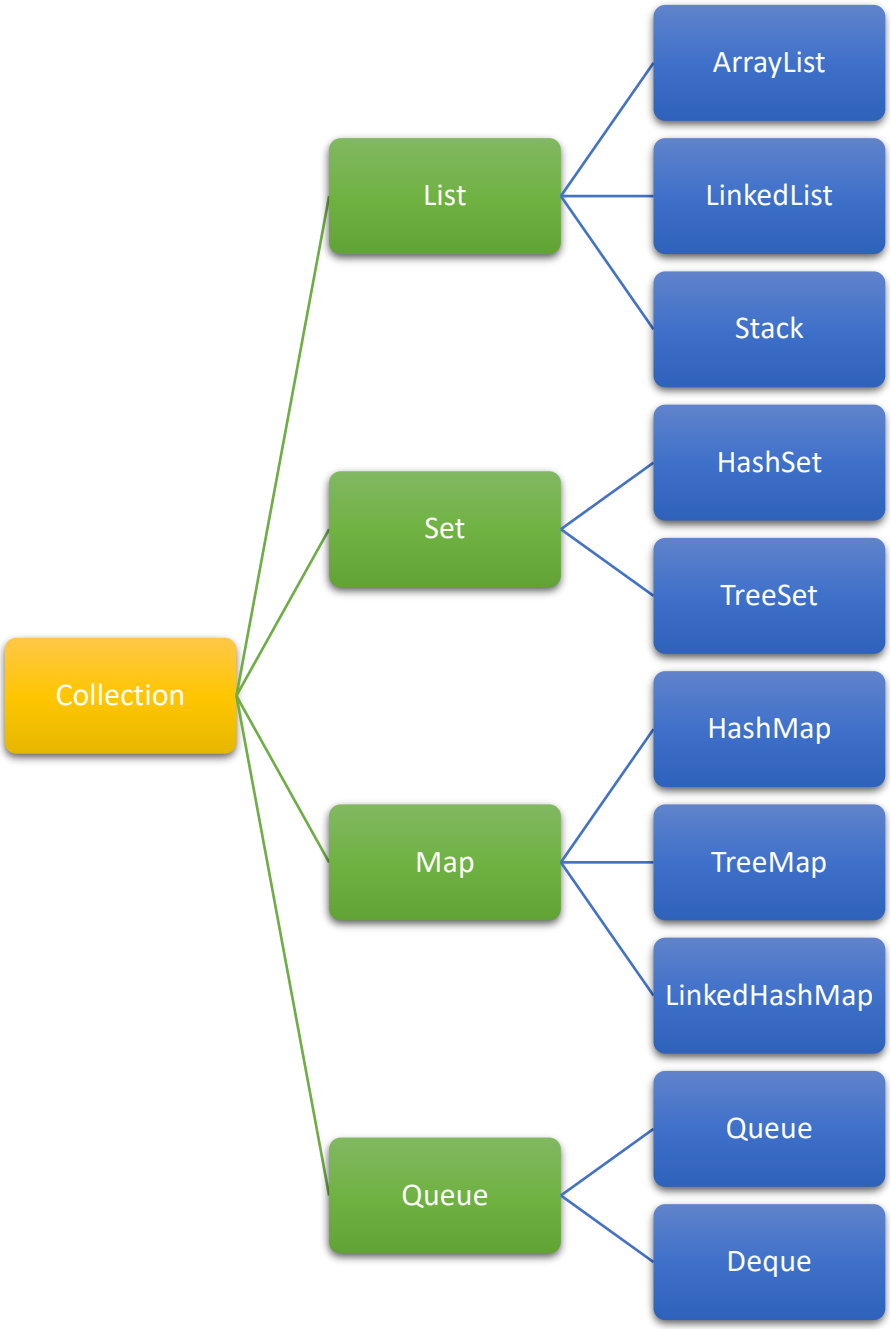
una excepción o no. Esto ayuda a evitar fugas de recursos y a mantener el código limpio y legible.

```
14          try(FileReader fr = new FileReader("pop.txt")){
15          System.out.println("Reading from file");
16          int c1 = fr.read();
17          while (c1 != -1) {
18              System.out.print((char) c1);
19              c1 = fr.read();
20          }
21      } catch (FileNotFoundException e1) {
22          e1.printStackTrace();
23      }  catch (IOException e) {
24          e.printStackTrace();
25      }
26
```

En este ejemplo la primera línea del código representa la construcción try-with-resources. El objeto FileReader se declara y se crea una instancia dentro de los paréntesis después de la palabra clave try. Una vez que se complete el bloque de prueba, el objeto del lector se cerrará automáticamente.

Explica los tipos de Collections

En Java, las colecciones son un conjunto de elementos que se utilizan para almacenar y acceder a datos de forma eficiente. Existen varios tipos de colecciones en Java, cada uno con sus propias características y utilidades. Algunos de los tipos de colecciones más comunes en Java son:



List

- ArrayList: Es una implementación de la interfaz List que utiliza un array interno para almacenar los elementos. Es una colección dinámica que permite elementos duplicados y mantiene el orden de inserción, además aumenta su tamaño según crece la colección de elementos.
- LinkedList: Es otra implementación de la interfaz List que utiliza una lista enlazada para almacenar los elementos, similar a ArrayList, pero tiene un mejor rendimiento para operaciones de inserción y eliminación en el medio de la lista.
- Stack: Es una interfaz que representa una estructura de datos de pila, donde el último elemento en entrar es el primero en salir (LIFO). Puede ser implementado mediante un ArrayDeque o Vector.

Set

- HashSet: Es una implementación de la interfaz Set que utiliza una tabla hash para almacenar los elementos. No permite elementos duplicados y no mantiene el orden de inserción.
- TreeSet: Es una implementación de la interfaz SortedSet que utiliza un árbol binario para almacenar los elementos. No permite elementos duplicados y mantiene los elementos en orden natural o mediante un comparador.

Map

- Map: Es una interfaz que permite almacenar pares de clave-valor.
- HashMap: Los elementos que inserta en el map no tendrán un orden específico. No aceptan claves duplicadas ni valores nulos.
- TreeMap: El Mapa lo ordena de forma "natural". Por ejemplo, si la clave son valores enteros los ordena de menos a mayor.
- LinkedHashMap: Inserta en el Map los elementos en el orden en el que se van insertando; es decir, que no tiene una ordenación de los elementos como tal, por lo que esta clase realiza las búsquedas de los elementos de forma más lenta que las demás clases.

Queue

- Queue: Es una lista ordenada de objetos cuyo uso se limita a insertar elementos al final de la lista y eliminar elementos al principio de la lista, (es decir), sigue el principio FIFO o First-In-First-Out.
- Deque: está relacionado con la cola de dos extremos que admite la adición o eliminación de elementos de cualquier extremo de la estructura de datos. Puede usarse como cola (primero en entrar, primero en salir/FIFO) o como pila (último en entrar, primero en salir/LIFO) . Deque es el acrónimo de double-ended queue.