1. تفاوت **ArrayList** و **Vector**:

در اینجا به تفکیک تفاوت های کلیدی بین ArrayList و Vector در جاوا آمده است:

1. **Synchronization**

ArrayList: غیر همگام است، به این معنی که چندین رشته می توانند به طور همزمان به آن دسترسی داشته باشند و آن را تغییر دهند. این منجر به عملکرد سریع‌تر در محیط‌های تک رشته‌ای می‌شود، اما نیاز به همگام‌سازی دستی در سناریوهای چند رشته‌ای دارد.

Vector: همگام است، به این معنی که فقط یک رشته می تواند در یک زمان به آن دسترسی داشته باشد و آن را تغییر دهد. این امر ایمنی thread را تضمین می کند، اما در مقایسه با ArrayList عملکرد کندتری را به همراه دارد.

1. **Resizing**

ArrayList: وقتی تعداد عناصر آن بیشتر شود، اندازه خود را 50 درصد از ظرفیت فعلی خود افزایش می دهد. این امر فضای تلف شده را به حداقل می رساند اما ممکن است منجر به تغییر اندازه مکرر با مجموعه داده های بزرگ شود.

Vector: با رسیدن به حد ظرفیت، اندازه آن دو برابر می شود (100٪ افزایش). این تغییر اندازه را کاهش می دهد اما می تواند در ابتدا حافظه بیشتری مصرف کند.

3. **Legacy Status**

ArrayList: در جاوا 1.2 معرفی شد و یک انتخاب مدرن برای آرایه های داینامیک می باشد.

Vector: بخشی از چارچوب مجموعه های قدیمی (قبل از جاوا 1.2) و یک کلاس قدیمی در نظر گرفته شده است.

4. **Performance**

ArrayList: به طور کلی به دلیل عدم هماهنگی و افزایش کوچکتر تغییر اندازه، "سریعتر" است.

Vector: به دلیل همگام سازی و افزایش تغییر اندازه بزرگتر، " کندتر" است.

5. **Other Differences**

**Iteration** : ArrayList فقط از رابط Iterator استفاده می کند، در حالی که Vector از Iterator و Enumeration پشتیبانی می کند.

**Inheritance** : ArrayList قابل طبقه بندی نیست، در حالی که Vector به طور مستقیم Object را گسترش می دهد.

**Framework Integration** : ArrayList به طور کامل با فریم ورک های کالکشن ها یکپارچه شده است، در حالی که Vector دارای برخی روش های قدیمی خارج از فریم ورک است.

6. **When to use which**

از ArrayList استفاده کنید: در بیشتر موارد، به خصوص برای برنامه های تک رشته ای یا زمانی که عملکرد بسیار مهم است.

از Vector استفاده کنید: فقط زمانی که ایمنی نخ ضروری است و سایر مکانیسم های همگام سازی غیرعملی هستند. با این حال، می توان با استفاده از متدهای از قبل آماده شده مانند «CopyOnWriteArrayList» برای توسعه امکان همگام سازی در ArrayListها استفاده کرد و به این ترتیب تنها مزیت vector نیز توسط ArrayListها قابل دستیابی خواهد بود.

به یاد داشته باشید که Vector یک کلاس قدیمی است و به دلیل پیامدهای عملکردی و عدم یکپارچگی کامل چارچوب، معمولاً برای توسعه جدید باید از آن اجتناب کرد.

در توسعه مدرن جاوا، به شدت توصیه می شود که "ArrayList" را به "Vector" در تقریباً همه موارد ترجیح دهید. در اینجا دلیل آن است:

\*\*کارایی:\*\*«ArrayList» به طور قابل توجهی سریعتر است، زیرا از نظر رشته ای ایمن نیست. این بدان معنی است که از سربار مکانیزم های همگام سازی استفاده شده توسط "Vector" جلوگیری می کند.

\*\*بهره وری:\*\* "ArrayList" در صورت نیاز، اندازه آرایه داخلی خود را تا 50٪ تغییر می دهد، که در نتیجه تغییر اندازه کمتر و کوچکتر در مقایسه با 100٪ دو برابر شدن "Vector"، منجر به استفاده بهتر از حافظه در اکثر سناریوها می شود.

\*\*مدرنیت:\*\* 'Vector' یک کلاس قدیمی است که قبل از Java Collections Framework است و ادغام محدودی با آن دارد. «ArrayList» بخش اصلی چارچوب است و از ویژگی‌ها و توسعه مداوم آن سود می‌برد.

\*\*گزینه های جایگزین برای Safe Threading:\*\*\* اگر ایمنی thread بسیار مهم است، استفاده از جایگزین‌های امن Thread از Frameworkها مانند «CopyOnWriteArrayList» یا «ConcurrentLinkedQueue» را در نظر بگیرید که عملکرد و ویژگی‌های همزمانی بهتری را در مقایسه با «Vector» ارائه می‌دهند.

بنابراین، مگر اینکه \*\*به شدت\*\* به ایمنی رشته در یک محیط چند رشته ای نیاز داشته باشید و \*\*نتوانید\*\* از جایگزین های مدرن استفاده کنید، تقریباً هیچ دلیلی برای استفاده از "Vector" در توسعه جاوا معاصر وجود ندارد. در اینجا چند نکته اضافی وجود دارد که باید در نظر گرفته شود:

\* اگر از کدهای قدیمی استفاده می‌کنید که از «Vector» استفاده می‌کند، به‌طور کلی جایگزین آن با «ArrayList» در بیشتر موارد بی‌خطر است، البته تا زمانی که همگام‌سازی مناسب در صورت لزوم اجرا شود.

با پیروی از این دستورالعمل ها، می توانید مطمئن شوید که از کارآمدترین و مناسب ترین ساختارهای داده برای برنامه های جاوا خود استفاده می کنید.



تفاوت HashSet و TreeSet:

در اینجا به تفکیک تفاوت های کلیدی بین HashSet و TreeSet در جاوا آمده است:

\*\*ساختار داده ای:\*\*

HashSet: از یک جدول هش برای حافظه داخلی استفاده می کند که زمان ثابتی را برای عملیاتی مانند «افزودن»، «حذف» و «حاوی» ارائه می دهد. هش کردن دسترسی سریعی را فراهم می کند اما هیچ نظمی را برای المنت های آن در نظر نمی گیرد.

TreeSet: از درخت جستجوی باینری برای جستجو در آن استفاده می کند. در حالی که عملیات جستجو، درج و حذف زمان لگاریتمی دارند (O(log n))، عناصر همیشه به ترتیب \*\*مرتب شده\*\* نگهداری می شوند.

\*\*سفارش و موارد تکراری:\*\*

HashSet: هیچ ترتیبی را برای عناصر حفظ نمی کند و امکان تکرار (عناصر با مقدار یکسان اما ارجاعات شیء متفاوت) را می دهد.

TreeSet: \*\*ترتیب طبیعی\*\* (صعودی) را برای عناصر به طور پیش فرض حفظ می کند. همچنین می توانید یک «مقایسه کننده» سفارشی برای تعریف معیارهای مرتب سازی خود ارائه دهید. \*\*تکراری مجاز نمی باشد\*\*.

\*\*مقادیر نال:\*\*

HashSet: به مقادیر "null" به عنوان عناصر اجازه می دهد.

TreeSet: به مقادیر "null" اجازه نمی دهد، اگر بخواهید آنها را اضافه کنید، "NullPointerException" پرتاب می شود.

\*\* Thread Safety:\*\*

HashSet: Thread-safe نیست، به این معنی که دسترسی همزمان از چندین ترد بدون همگام سازی مناسب می تواند منجر به رفتار غیرمنتظره شود.

TreeSet: بصورت پیش فرضThread-safeنیست اما می توانید از متدهایی مانند «Collections.synchronizedSet(TreeSet)» یا «ConcurrentSkipListSet» برای ایمنی رشته استفاده کنید.

\*\* Use Cases:\*\*

HashSet: بهترین گزینه برای زمانی که جستجوهای سریع و بررسی عضویت بسیار مهم است و ترتیب مهم نیست. ایده آل برای مجموعه ای از اشیاء منحصر به فرد بر اساس متدهای 'hashCode()' و 'equals()'.

TreeSet: زمانی مفید است که به عناصر مرتب شده نیاز دارید و می خواهید بر اساس سفارش یا مقادیر خاص بازیابی کارآمدی داشته باشید. مناسب برای سناریوهایی که در آن تکرار از طریق عناصر به ترتیب خاص ضروری است.

\*\* Performance:\*\*

HashSet: به دلیل پیچیدگی میانگین زمانی ثابت، معمولاً برای عملیات اساسی "سریعتر" است.

TreeSet: از نظر عملیات اساسی به دلیل مرتب سازی سربار "کندتر" است، اما اگر دسترسی مرتب شده مهمتر از سرعت باشد، ممکن است ترجیح داده شود.

**به طور خلاصه، HashSet را برای مجموعه‌های سریع و نامرتب با موارد تکراری احتمالی، و TreeSet را برای مجموعه‌های مرتب‌شده با عناصر منحصربه‌فرد که نظم مهم است، انتخاب کنید. به یاد داشته باشید که هنگام استفاده از هر یک از مجموعه ها، الزامات ایمنی نخ را در نظر بگیرید.**

مثال از HashSet:

این مثال اضافه کردن نام‌های دانش‌آموز منحصربه‌فرد به HashSet و تکرار آن‌ها را نشان می‌دهد:

import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

// Create a HashSet to store student names

HashSet<String> studentNames = new HashSet<>();

// Add some student names

studentNames.add("Alice");

studentNames.add("Bob");

studentNames.add("Charlie");

studentNames.add("David"); // Duplicate is ignored

// Iterate over the student names

System.out.println("Students:");

for (String name : studentNames) {

System.out.println(name);

}

}

}

```

output:

Students:

Alice

Bob

Charlie

David

همانطور که می بینید، نام تکراری "دیوید" اضافه نشده است و عناصر به ترتیب خاصی چاپ نمی شوند.

مثال TreeSet:

این مثال اضافه کردن اعداد منحصر به فرد به یک TreeSet و تکرار آنها را به ترتیب صعودی نشان می دهد:

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

// Create a TreeSet to store numbers

TreeSet<Integer> numbers = new TreeSet<>();

// Add some numbers

numbers.add(5);

numbers.add(2);

numbers.add(8);

numbers.add(1); // Duplicate is ignored

// Iterate over the numbers (ascending order)

System.out.println("Numbers:");

for (Integer number : numbers) {

System.out.println(number);

}

}

}

output:

Numbers:

1

2

5

8

عناصر به ترتیب صعودی چاپ می شوند زیرا TreeSet ساختار مرتب شده ای را بر اساس ترتیب طبیعی عناصر (یا یک مقایسه کننده سفارشی در صورت ارائه) حفظ می کند.

تفاوت HashMap و HashTable:

