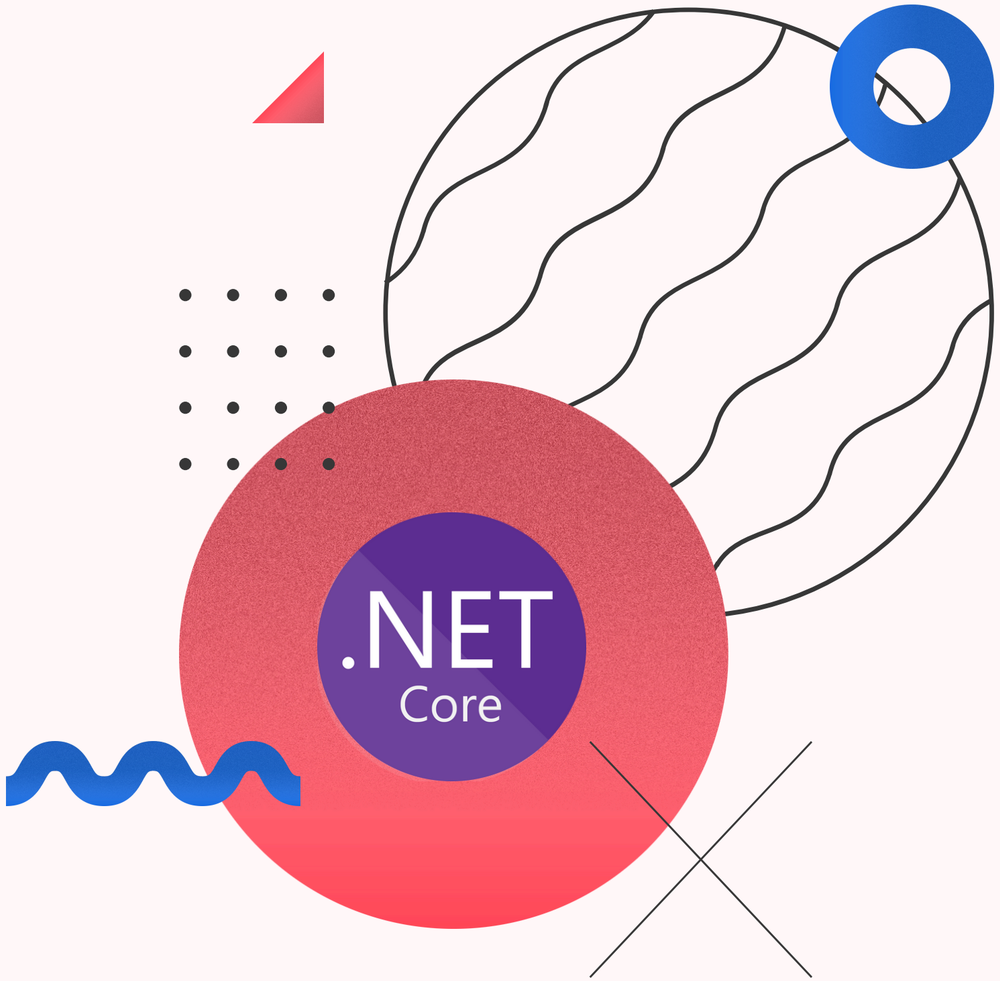
STARTUP HAKK

ArrayList

VS

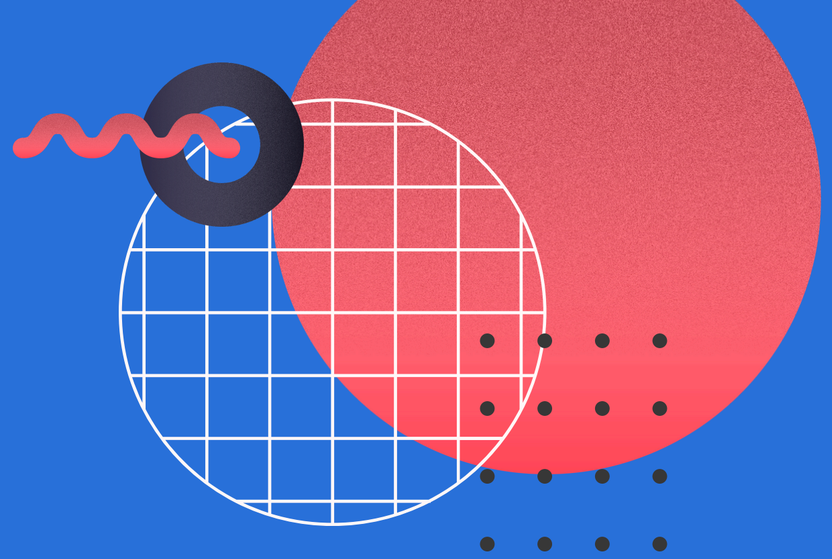
List



Spencer Thomason

Twitter: @StartupHakk
Facebook: @StartupHakk

ArrayList: Pros

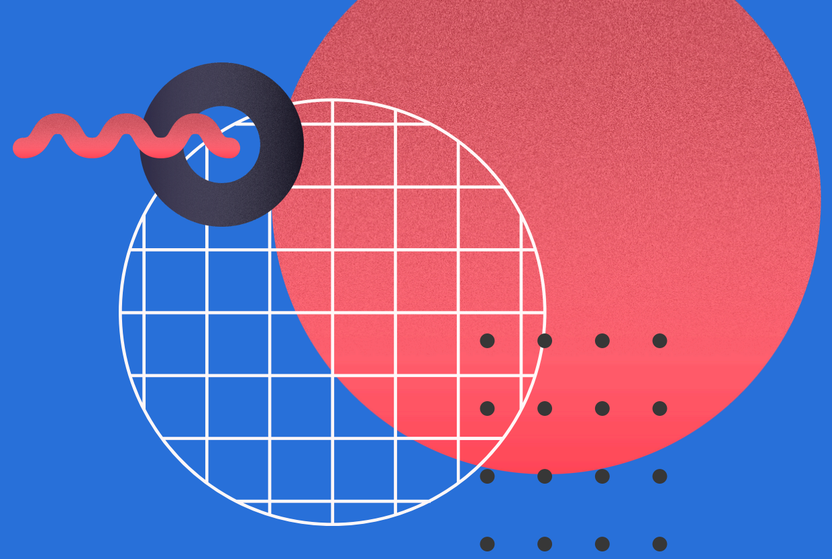


Dynamic Size:
ArrayList automatically adjusts its size when elements are added or removed, which makes it convenient for scenarios where the size of the collection is not known beforehand.

No Type Constraint:
ArrayList can hold elements of any type since it stores objects, which can be advantageous when dealing with heterogeneous collections.

Direct Access by Index:
Provides fast indexed access to elements, allowing for efficient random access retrieval.

ArrayList: Cons

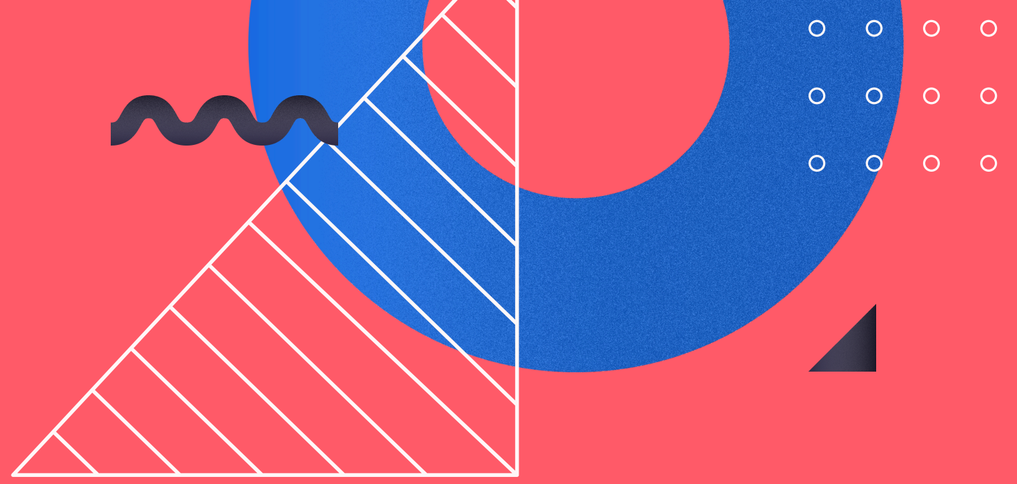


Boxing/Unboxing Overhead:
Since ArrayList stores objects, value types need to be boxed when added to the collection and unboxed when retrieved, which can lead to performance overhead, especially in large collections.

Lack of Type Safety:
The absence of generic typing (<T>) means there's no compile-time type safety. This can result in runtime errors if elements of incorrect types are added and not caught during compilation.

Type Casting Required:
Retrieving elements from an ArrayList requires explicit casting, which can lead to runtime errors if the wrong type is cast.

List<T>: Pros

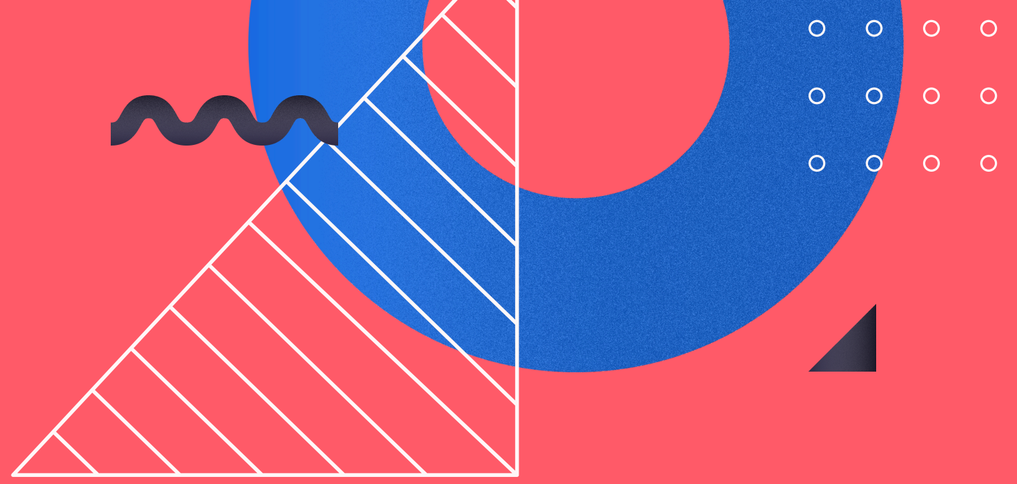


Type Safety:
List<T> is strongly typed, providing compile-time type safety. This ensures that only elements of the specified type can be added to the list, reducing the risk of runtime errors.

Performance:
List<T> generally performs better than ArrayList for value types since it doesn't require boxing/unboxing. It's also optimized for specific types of access, like iterating through elements.

Generic Support:
List<T> supports generics, allowing for cleaner code and improved readability. It also enables better code maintenance and easier debugging.

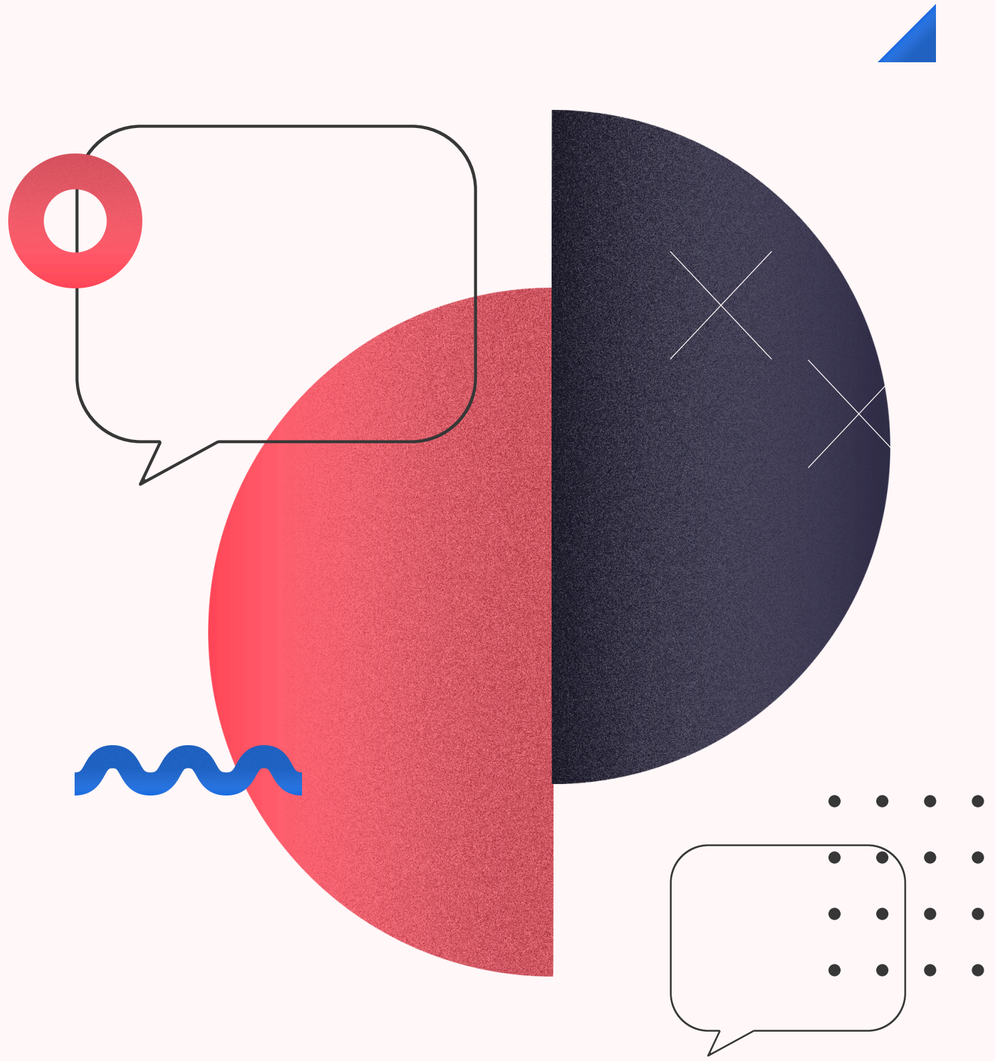
List<T>: Cons



Fixed Type:
Unlike ArrayList, List<T> is limited to storing elements of a single type specified by its generic type parameter <T>, which can be restrictive in scenarios where heterogeneous collections are needed.

No Automatic Upcasting:
Since List<T> is strongly typed, you can't implicitly upcast between different types of lists. This might lead to extra code when dealing with polymorphic behavior.

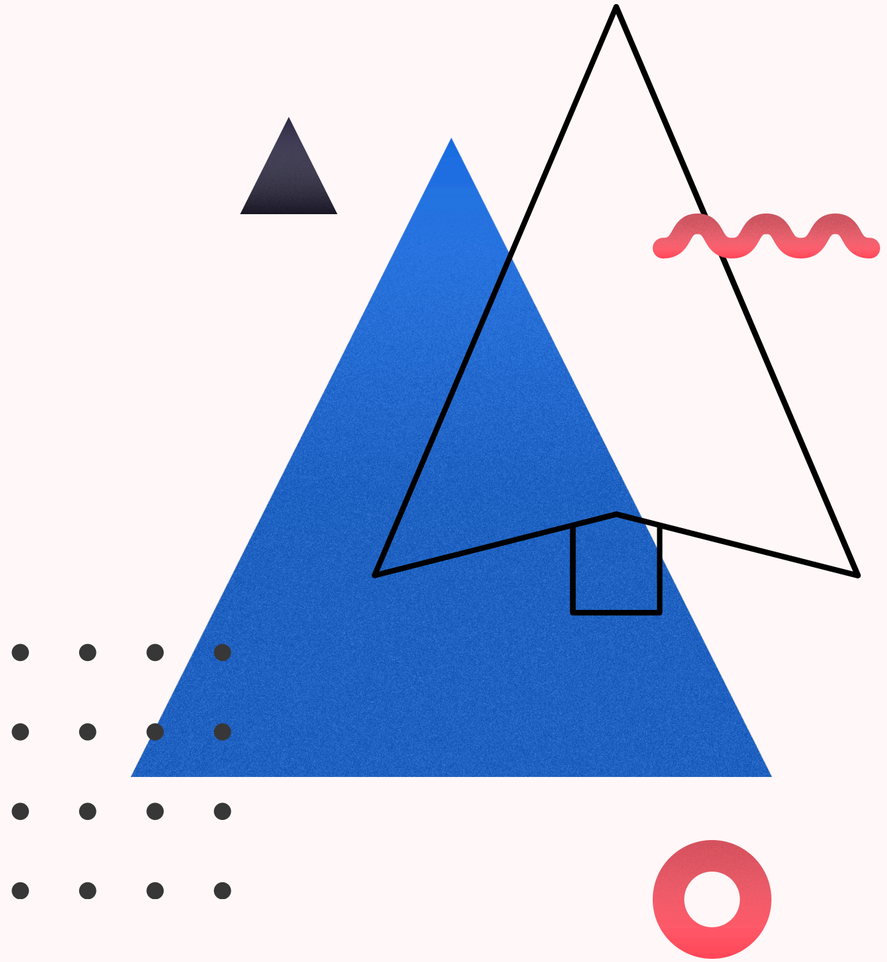
Resizing Overhead:
While List<T> automatically resizes internally, resizing can still incur performance overhead, especially when dealing with large collections or frequent resizing operations.



Code Samples

Summary

In summary, ArrayList offers flexibility with its ability to hold elements of any type but lacks type safety and can incur performance overhead due to boxing/unboxing. On the other hand, List<T> provides type safety, better performance for value types, and cleaner code with generics, but it's restricted to storing elements of a single type specified by its generic type parameter.



Thank you