What are Generics in Java?
Generics in Java allow you to create classes, interfaces, and methods that operate on types as parameters, enabling you to write code that is type-safe and reusable. Generics enable you to define classes or methods that can work with different types while providing compile-time type checking.

Why Use Generics?

Type Safety: Generics help catch type-related errors at compile time rather than at runtime. This ensures that you are working with the correct data types, reducing the likelihood of bugs.

Code Reusability: Generics enable you to write code that works with various data types, promoting code reusability and reducing redundancy.

Flexibility: With generics, you can write more flexible and generic code that adapts to different data types, making your code more adaptable and versatile.

Readability: Generics make your code more readable and self-explanatory by specifying the data types used in a class, method, or interface.

When to Use Generics?

Use generics when:
You want to create a data structure or class that can work with multiple data types (e.g., ArrayList<T>).
You want to create a method that can be used with different data types without code duplication.
You need type safety and compile-time checks for your code.
Where to Use Generics?
Generics can be applied in various parts of your Java code, such as:
Collections: Generics are commonly used in Java's collection framework, e.g., ArrayList<T>, HashMap<K, V>.

Custom data structures: You can create your own data structures that use generics.
Utility classes: Generic utility classes for sorting, searching, etc.
Methods: Writing methods that work with generic types.

How to Use Generics in Java
Let's explore how to use generics with examples:
Defining a Generic Class
javaCopy code

```
public class Box<T> {
   private T value;

   public Box(T value) {
      this.value = value;
   }

   public T getValue() {
      return value;
   }
}
```

Using a Generic Class
javaCopy code

```
Box<Integer> intBox = new Box<>(42);int value = intBox.getValue(); //
```

No casting needed
Defining a Generic Method
javaCopy code

```
public <T> T findMax(T[] array) {
   if (array == null || array.length == 0)
      return null;

   T max = array[0];
   for (T item : array) {
      if (item.compareTo(max) > 0) {
         max = item;
      }
   }
   return max;
}
```

Using a Generic Method
javaCopy code

Integer[] intArray = {1, 5, 2, 8, 3};Integer maxInt = findMax(intArray); // Automatically infers the type

String[] stringArray = {"apple", "banana", "cherry"};String maxString = findMax(stringArray);
These examples demonstrate the use of generics in defining generic classes and methods. Generics are particularly valuable when you want to create flexible, type-safe, and reusable code.