1) What are Generics in Java ?

Ans:

1. Generics allow you to create classes, interfaces, and methods that can operate on various data types by using placeholders called type parameters (`<T>`). This promotes code reusability and flexibility.

2. Generics provide compile-time type checking, helping to catch type-related errors at compile time rather than runtime. This ensures that the code is less error-prone and more reliable.

3. Java's Collections Framework extensively utilizes generics to create type-safe data structures like lists, sets, and maps that hold elements of specific types. This enhances both type safety and code readability.

4. Bounded Type Parameters: You can restrict the range of acceptable types for a generic parameter using bounded type parameters. This ensures that the types passed to the generic component meet specific requirements.

5. Type Erasure and Backward Compatibility: Generics use type erasure to remove most generic type information during compilation, making the code compatible with older non-generic Java code. However, the compiler enforces type safety during compilation.

2) What are the benefits of using Generics in Java?

Ans:

1. Type Safety: Generics enable compile-time type checking, preventing type-related errors at runtime. This ensures that data types are used correctly, leading to more robust and reliable code.

2. Code Reusability: By creating generic classes, methods, and interfaces, you can write code that works with multiple data types. This reduces code duplication and promotes reusability across different scenarios.

3. Collections Framework: Generics are integral to Java's Collections Framework, ensuring type-safe storage and retrieval of elements in data structures like lists, sets, and maps. This prevents errors caused by incompatible data types.

4. Readability and Expressiveness: Generics make code more readable and self-explanatory. Type parameters provide clear documentation about the expected data types, improving code comprehension for developers.

5. Compile-Time Error Detection: Generics help catch errors at compile time rather than runtime. This early detection reduces debugging efforts and minimizes the risk of unexpected issues in deployed code.

6. Elimination of Type Casting: With generics, the need for manual type casting is reduced or eliminated, leading to cleaner and more concise code. This also enhances performance by avoiding unnecessary type conversions.

3) What is a Generic class in Java ?

Ans:

1. Type Parameterization: Generic classes are defined with type parameters enclosed in angle brackets (`<>`). These placeholders represent the actual data types that the class will work with.

2. Data Type Flexibility: Generic classes can be instantiated with different data types, enabling a single class to work with various types of data.

3. Code Reusability: Generic classes promote code reusability by allowing you to create a single class that can handle multiple data types, eliminating the need for duplicated code.

4. Type Safety: Type parameters ensure that instances of the generic class work with the correct data types, preventing type-related errors and enhancing code reliability.

5. Collections Framework Integration: Java's Collections Framework extensively utilizes generic classes to create type-safe data structures like lists, sets, and maps. This enforces type consistency within collections.

6. Instance Customization: When creating an instance of a generic class, you specify the specific data type(s) the instance will work with, adapting the behavior to match those types.

4) What is a Type parameter in Java Generics?

Ans:

1. Placeholder for Data Types: Type parameters are placeholders represented by names (e.g., `T`, `E`) within angle brackets (`<>`) in generic classes, methods, or interfaces.

2. Flexible Data Handling: They allow you to create code that works with different data types while ensuring type safety and avoiding code duplication.

3. Usage in Definitions: Type parameters are used in the class, method, or interface definitions to indicate where the actual data types will be substituted when instances are created or methods are called.

4. Actual Type Arguments: When using a generic class or method, you provide actual data types (e.g., `Integer`, `String`) as arguments for the type parameters, tailoring the behavior to those types.

5. Type Erasure: Java employs type erasure to remove most type parameter information during compilation, ensuring compatibility with non-generic code and retaining type safety checks.

5) What is a Generic method in Java?

Ans:

A generic method in Java is a method that can operate on different data types while maintaining type safety.

It allows you to create methods that can work with various types of input, and the actual data types are determined when the method is called.

1. Type Parameter Declaration: In a generic method, you declare a type parameter by specifying it before the return type of the method. Common type parameter names include `T`, `E`, and so on.

2. Method Signature: The declaration of the generic method includes the type parameter and the regular method parameters. The type parameter can be used in the method's return type and its parameter types.

3. Flexible Data Handling: Generic methods allow you to write a single method implementation that can be used with different data types, enhancing code reusability.

4. Type Inference: In many cases, Java's compiler can infer the actual type argument based on the context in which the method is called, reducing the need for explicit type specification.

5. Actual Type Argument: When calling a generic method, you provide the actual data type for the type parameter in angle brackets. For example, `myGenericMethod<Integer>(value)` specifies `Integer` as the type argument.

Example Syntax:

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

6) What is the difference between ArrayList and ArrayList<T> ?

Ans:

ArrayList:

- Non-generic version of the ArrayList class.
- Introduced in earlier versions of Java before generics.
- Does not enforce type safety; any type of object can be added.
- Type information is not checked at compile time but can lead to runtime errors.
- Limited expressiveness and potential for type-related bugs.
- Not recommended for modern Java programming due to lack of type safety.

ArrayList<T>:

- Generic version of the ArrayList class.
- Introduced with Java generics to provide type safety.
- Enforces type safety by allowing only elements of the specified data type.
- Compiler checks type information at compile time, reducing runtime errors.
- Increases code expressiveness and minimizes type-related bugs.
- Recommended for modern Java programming to ensure type safety and reliable code.