# Java Generics Questions

### 1. What is Java Generics?

A. Java Generics is a feature in Java that allows for type safety and improved code reusability. It provides a way to specify the type of data that can be stored in a collection or passed as an argument to a method. This improves type safety by preventing incorrect type assignments, reducing the need for typecasting, and allowing for more efficient code.

### 2. What is the purpose of generics in Java?

A. The purpose of generics in Java is to provide type safety and improved code reusability. They allow developers to specify the type of data that can be stored in a collection or passed as an argument to a method, improving type safety and reducing the need for typecasting. This results in more efficient and maintainable code.

### 3. How does Java Generics improve type safety?

A. Java Generics improve type safety by allowing developers to specify the type of data that can be stored in a collection or passed as an argument to a method. This prevents incorrect type assignments, reducing the need for type casting and making it easier to detect and prevent bugs. When an incorrect type is assigned, a compile-time error is generated, allowing the programmer to quickly fix the issue before it becomes a problem at runtime. By improving type safety, Java Generics also makes the code more maintainable, as it is easier to understand and modify.

### 4. What is type parameterization in Java?

A. Type parameterization in Java refers to the process of defining a generic type by specifying the type parameter that will be used. This allows for a single class, method, or interface definition to be reused for different types of data. When a generic type is instantiated, the programmer specifies the actual type to be used in place of the type parameter. This allows for type safety and improved code reusability, as the same code can be used for multiple types of data, reducing the need for duplication and making the code easier to maintain.

### 5. What is type erasure in Java Generics?

A. Type erasure in Java Generics is the process by which generic type information is removed at compile-time, leaving only the raw type information. This means that at runtime, generic types are treated as their raw type, and all type parameter information is lost. For example, when the generic class List<T> is compiled, it is treated as if it were a raw type List, without any information about the type parameter T. This allows Java Generics to maintain backwards compatibility with older versions of the Java language, as generic type information is not available at runtime. However, it also means that some type information is lost, and certain operations, such as reflection, may not work as expected with generic types.

### 6. How do you create a generic class in Java?

A. To create a generic class in Java, you need to declare a type parameter in angle brackets < > after the class name. For example:

* **public** **class** MyClass<T> {
* T value;
* // ...
* }

Here, 'T' is the type parameter, which can be any type specified by the user when the class is instantiated. The type parameter can then be used within the class definition to specify the type of variables, return types of methods, and so on. To instantiate a generic class, you simply provide the actual type to be used in place of the type parameter:

* MyClass<Integer> myObject = **new** MyClass<Integer>();

This creates an instance of 'MyClass' where the type parameter T is replaced by 'Integer'.

### 7. How do you create a generic method in Java?

A. To create a generic method in Java, you declare a type parameter within angle brackets < > before the return type of the method. For example:

* **public** **static** <T> **void** myMethod(T parameter) {
* // ...
* }

Here, 'T' is the type parameter, which can be any type specified by the user when the method is called. The type parameter can then be used within the method to specify the type of variables and return types. To call a generic method, you simply provide the actual type to be used in place of the type parameter:

* myMethod(123);

This calls the method 'myMethod' with the type parameter 'T' replaced by 'Integer'. Note that the type parameter can also be inferred from the argument passed to the method, so it is not necessary to explicitly specify it.

### 8. How do you use wildcards in Java Generics?

A. Wildcards in Java Generics are used to specify unknown or generic types in a more flexible way. There are two types of wildcards in Java Generics: upper bounded wildcards and lower bounded wildcards.

Upper bounded wildcards are specified using the '?' symbol and the 'extends' keyword. For example:

* **public** **void** myMethod(List<? **extends** Number> list) {
* // ...
* }

This specifies that the method 'myMethod' accepts a 'List' of any type that extends the 'Number' class.

Lower bounded wildcards are specified using the '?' symbol and the 'super' keyword. For example:

* **public** **void** myMethod(List<? **super** Integer> list) {
* // ...
* }

This specifies that the method 'myMethod' accepts a 'List' of any type that is a superclass of 'Integer'.

Wildcards are useful in situations where you want to write generic code that can accept a wide range of types, but still maintain some type safety and constraints on the types that can be used.

### 9. What are the different types of wildcards in Java Generics?

A. There are two types of wildcards in Java Generics: upper bounded wildcards and lower bounded wildcards.

Upper bounded wildcards: Upper bounded wildcards are specified using the '?' symbol and the 'extends' keyword. For example: 'List<? extends Number>'. This specifies that the list can hold elements of type 'Number' or any subtype of 'Number'.

Lower bounded wildcards: Lower bounded wildcards are specified using the '?' symbol and the 'super' keyword. For example: 'List<? super Integer>'. This specifies that the list can hold elements of type 'Integer' or any supertype of 'Integer'.

### 10. What is a bounded type parameter in Java Generics?

A.A bounded type parameter in Java Generics is a type parameter that is restricted to a specific type or range of types. Bounding a type parameter helps to ensure that the code that uses the generic class or method is type safe and will only work with the specified types.

Bounds are specified using the 'extends' keyword in Java. For example:

* **public** **class** MyClass<T **extends** Number> {
* T value;
* // ...
* }

Here, the type parameter 'T' is bounded to the type 'Number', meaning that it can only be instantiated with types that are 'Number' or a subtype of 'Number'. This helps to ensure that the code that uses the 'MyClass' class will only work with objects of type 'Number' or its subtypes, and will not work with other types.

Bounded type parameters can also be used with wildcards in generic methods:

* **public** **static** <T **extends** Number> **void** myMethod(T parameter) {
* // ...
* }

This specifies that the type parameter 'T' in the method 'myMethod' is bounded to the type 'Number', meaning that it can only be called with arguments of type 'Number' or its subtypes.

### 11. Can a generic class extend another class?

A. Yes, a generic class in Java can extend another class. The generic class can either be a subtype of the superclass or can have the same type parameters as the superclass. For example:

* yaha par add kardena codeclass SuperClass<T> {
* // ...
* }
* **class** MyClass<T> **extends** SuperClass<T> {
* // ...
* }

Here, 'MyClass' extends the generic class 'SuperClass' and has the same type parameter 'T'. When 'MyClass' is instantiated, the type parameter can be specified, just as with any other generic class:

* MyClass<Integer> myObject = **new** MyClass<Integer>();

#### *Note that if the superclass is not generic, then the subclass can extend it just like any other non-generic class.*

### 12. Can a generic class implement an interface?

A. Yes, a generic class in Java can implement an interface. The generic class can either implement the interface with the same type parameters as the interface or with its own type parameters. For example:

* **interface** MyInterface<T> {
* // ...
* }
* **class** MyClass<T> **implements** MyInterface<T> {
* // ...
* }

Here, 'MyClass' implements the generic interface 'MyInterface' and has the same type parameter T. When 'MyClass' is instantiated, the type parameter can be specified, just as with any other generic class:

* MyClass<Integer> myObject = **new** MyClass<Integer>();

If the interface is not generic, then the class can implement it just like any other non-generic interface.

### 13. Can you use primitives as type arguments in Java Generics?

A. No, you cannot use primitives as type arguments in Java Generics. Primitive types in Java (e.g. 'int', 'float', 'char') are not objects and do not inherit from the 'Object' class. Java Generics are based on the concept of type parameterization, which requires that the type argument be a reference type.

If you need to use primitive types with Java Generics, you can use their corresponding wrapper classes (e.g. 'Integer', Float', Character') instead. These wrapper classes are reference types and can be used as type arguments in Java Generics. For example:

* List<Integer> list = **new** ArrayList<Integer>();
* list.add(1);
* list.add(2);
* list.add(3);

### 14. What is type inference in Java Generics?

A. Type inference in Java Generics is a feature introduced in Java 7 that allows the compiler to determine the type argument for a generic type based on the context in which the type is used. This means that in some cases, you do not need to explicitly specify the type argument when creating an instance of a generic type.

For example, consider the following code:

* List<Integer> list = **new** ArrayList<>();
* list.add(1);
* list.add(2);
* list.add(3);

Here, the type argument 'Integer' for the generic type 'ArrayList' can be inferred by the compiler based on the type of the variable 'list', which is 'List<Integer>'. This means that you do not need to explicitly specify the type argument 'Integer' when creating an instance of 'ArrayList'.

Type inference can also be used with generic methods. For example:

* **class** MyClass {
* **public** **static** <T> **void** printList(List<T> list) {
* **for** (T item : list) {
* System.out.println(item);
* }
* }
* }
* List<Integer> list = Arrays.asList(1, 2, 3);
* MyClass.printList(list);

Here, the type argument 'T' for the generic method 'printList' can be inferred by the compiler based on the type of the argument passed to the method, which is 'List<Integer>'. This means that you do not need to explicitly specify the type argument 'T' when calling the method.

### 15. How do you use the diamond operator with generics in Java?

A. The diamond operator ('< >') is used in Java to infer the type argument for a generic type. The diamond operator was introduced in Java 7 and can be used with any generic type to simplify the syntax for creating instances of the type.

For example, consider the following code:

* List<Integer> list = **new** ArrayList<Integer>();
* list.add(1);
* list.add(2);
* list.add(3);

This code creates a list of 'Integers' using the 'ArrayList' class and explicitly specifies the type argument 'Integer'. With the diamond operator, the same code can be written as follows:

* List<Integer> list = **new** ArrayList<>();
* list.add(1);
* list.add(2);
* list.add(3);

Here, the diamond operator '< >' is used to infer the type argument 'Integer' based on the type of the variable 'list', which is 'List<Integer>'. The compiler will determine the type argument based on the context in which the generic type is used, and this eliminates the need to explicitly specify the type argument.

The diamond operator can be used with any generic type, including generic classes, generic methods, and generic interfaces. However, it is important to note that the type argument must be able to be inferred by the compiler, otherwise a compile-time error will occur.

### 16. What is the difference between a raw type and a parameterized type in Java?

A. A raw type in Java is a generic type without its type argument specified. A raw type can be thought of as a generic type in its original, unparameterized form. Raw types were used in Java prior to the introduction of generics in Java 5, and they still exist today for backward compatibility.

For example, consider the following generic class:

* **class** MyClass<T> {
* **private** T data;
* // ...
* }

The raw type of 'MyClass' is 'MyClass', and it can be used as follows:

* MyClass rawObj = **new** MyClass();

A parameterized type, on the other hand, is a generic type with its type argument specified. A parameterized type is created by providing the type argument in angle brackets (<>) after the generic type.

For example, consider the following code:

* MyClass<Integer> intObj = **new** MyClass<Integer>();

Here, 'MyClass<Integer>' is a parameterized type, and the type argument 'Integer' is specified in the angle brackets.

It is important to note that the use of raw types in Java is discouraged, as it can lead to unsafe type casts and potential runtime errors. In general, it is recommended to use parameterized types whenever possible to take advantage of the type safety guarantees provided by generics.

### 17. How do you create a generic constructor in Java?

A. A generic constructor in Java is created in the same way as a generic class or method. The constructor's type parameters are declared in angle brackets ('< >') before the constructor's name, just like a generic class or method.

Here is an example of a generic class with a generic constructor:

* **class** MyClass<T> {
* **private** T data;
* **public** <U> MyClass(U data) {
* **this**.data = (T) data;
* }
* // ...
* }

In this example, the generic constructor 'MyClass' has a type parameter 'U', which is used to specify the type of the 'data' argument. The constructor casts the 'data' argument to the type 'T', which is the type parameter of the containing class 'MyClass'.

It is important to note that the use of type casts in a generic constructor can result in runtime errors if the cast is invalid. Care should be taken to ensure that the type argument is appropriate and the cast will succeed. In general, it is recommended to use type bounds or other mechanisms to ensure type safety when using generic constructors.

### 18. How do you create a generic enum in Java?

A. Enumerations in Java do not support generic type parameters. This means that it is not possible to create a generic enum in Java. However, you can use a regular enum and create generic methods or classes that use the enum as a type argument. For example:

* **enum** Color { RED, GREEN, BLUE }
* **class** EnumWrapper<T **extends** Enum<T>> {
* **private** T enumValue;
* **public** EnumWrapper(T enumValue) {
* **this**.enumValue = enumValue;
* }
* **public** T getEnumValue() {
* **return** enumValue;
* }
* }
* **public** **class** Main {
* **public** **static** **void** main(String[] args) {
* EnumWrapper<Color> wrapper = **new** EnumWrapper<>(Color.RED);
* System.out.println(wrapper.getEnumValue());
* }
* }

In this example, the 'EnumWrapper' class is a generic class that takes an enumeration type as its type argument. The 'EnumWrapper' class holds a value of the enumeration type, and provides a method to access it. The main method creates an instance of 'EnumWrapper' for the 'Color enum', and prints its value.

### 19. How do you use generics with arrays in Java?

A. Arrays in Java do not support generic types, so you cannot create an array of a generic type. However, you can use an array of Object type, and use type casts to assign and retrieve values.

Here is an example of using an array of Object type with generics:

* **class** MyClass<T> {
* **private** T[] dataArray;
* @SuppressWarnings("unchecked")
* **public** MyClass(T... dataArray) {
* **this**.dataArray = (T[]) **new** Object[dataArray.length];
* System.arraycopy(dataArray, 0, **this**.dataArray, 0, dataArray.length);
* }
* **public** T getDataAtIndex(**int** index) {
* **return** dataArray[index];
* }
* }
* **public** **class** Main {
* **public** **static** **void** main(String[] args) {
* MyClass<String> stringClass = **new** MyClass<>("Hello", "World");
* System.out.println(stringClass.getDataAtIndex(0));
* System.out.println(stringClass.getDataAtIndex(1));
* }
* }

In this example, the 'MyClass' class takes an array of generic type 'T' as an argument to its constructor. The constructor uses 'System.arraycopy' to copy the values of the input array to a new array of type 'Object[]'. The type cast to 'T[]' generates an unchecked warning, which is suppressed using the '@SuppressWarnings annotation'.

It is important to note that using an array of Object type and type casts can result in runtime errors if the type argument is not appropriate or the cast is invalid. Care should be taken to ensure that the type argument is appropriate and the type casts will succeed. In general, it is recommended to use other mechanisms, such as generic classes or methods, to ensure type safety when working with arrays and generics in Java.

### 20. What are some common pitfalls to avoid with Java Generics?

* Raw Types: Using raw types instead of parameterized types can result in loss of type safety and exceptions at runtime.
* Unbounded Wildcards: Using unbounded wildcards can result in loss of type safety, as they allow any type to be used as the type argument.
* Incorrect Use of Wildcards: Incorrect use of wildcards, such as using a lower-bounded wildcard where an upper-bounded wildcard is required, can result in type mismatches and compile-time errors.
* Incorrect Use of Type Inference: Incorrect use of type inference, such as not providing enough information to infer the type argument, can result in type mismatches and compile-time errors.
* Incorrect Use of Generics with Arrays: Java arrays do not support generics, and using a generic type as the component type of an array can result in a compile-time error.
* Inconsistent Use of Generics: Inconsistent use of generics, such as using a parameterized type in one part of the code and a raw type in another part, can result in type mismatches and unexpected behavior.
* Incorrect Use of Generics with Legacy Code: Incorrect use of generics with legacy code, such as attempting to use a generic type where the legacy code expects a raw type, can result in type mismatches and unexpected behavior.
* Ignoring Unchecked Warnings: Ignoring unchecked warnings can result in unexpected behavior and exceptions at runtime.
* Incorrect Use of Constructors with Generics: Incorrect use of constructors with generics, such as attempting to create an instance of a generic type with an incompatible type argument, can result in type mismatches and unexpected behavior.

It is important to understand these pitfalls and take care to avoid them when using generics in Java. By doing so, you can ensure that your code is safe, efficient, and easy to maintain.

### 21. How do you use upper bounded wildcards in Java Generics?

A. Upper bounded wildcards are used in Java Generics to specify that a type argument must be a subclass of a particular type. This allows for more flexible and generic type-safe code.

The syntax for an upper bounded wildcard is to use the wildcard character '?' followed by the keyword 'extends' and the upper bound type. For example:

* List<? **extends** Number> list = **new** ArrayList<Integer>();

In this example, the type argument of the ArrayList is Integer, which is a subclass of Number, so the list can be assigned to a reference of type 'List<? extends Number>'. This allows you to use the 'list' reference to access the elements in the list, as long as you only use methods that return 'Number' or its subclasses.

Upper bounded wildcards are useful in situations where you want to write generic methods or classes that can accept a variety of types that are subclasses of a particular type, while still maintaining type safety.

### 22. How do you use lower bounded wildcards in Java Generics?

A. Lower bounded wildcards are used in Java Generics to specify that a type argument must be a superclass of a particular type. This allows for more flexible and generic type-safe code.

The syntax for a lower bounded wildcard is to use the wildcard character '?' followed by the keyword 'super' and the lower bound type. For example:

* List<? **super** Integer> list = **new** ArrayList<Number>();

In this example, the type argument of the ArrayList is Number, which is a superclass of Integer, so the list can be assigned to a reference of type 'List<? super Integer>'. This allows you to use the 'list' reference to add elements of type 'Integer' to the list, as well as other types that are subclasses of 'Integer'.

Lower bounded wildcards are useful in situations where you want to write generic methods or classes that can accept a variety of types that are superclasses of a particular type, while still maintaining type safety. They are particularly useful when writing generic classes and methods that need to write elements to a collection, rather than just reading elements from it.

### 23. How do you create a generic interface in Java?

A. A generic interface in Java is defined by specifying type parameters within angle brackets ('< >') after the interface name. The type parameters are placeholders for the actual types that will be used when the interface is implemented or extended.

For example, to create a generic interface that defines a method to return a single value:

* **public** **interface** MyInterface<T> {
* T getValue();
* }

In this example, the type parameter 'T' is a placeholder for the actual type that will be used when the interface is implemented. The implementing class or subclass will provide the actual type argument for 'T'.

* **public** **class** MyClass **implements** MyInterface<String> {
* @Override
* **public** String getValue() {
* **return** "Hello, World!";
* }
* }

In this example, the class 'MyClass' implements the generic interface 'MyInterface', and provides the type argument 'String' for the type parameter 'T'. This means that the method 'getValue' will return a value of type String.

### 24. What is a generic type hierarchy in Java?

A. A generic type hierarchy in Java refers to a hierarchy of types that are related through inheritance and type parameterization. It is used to enforce type safety in generic code, and ensure that generic classes, interfaces, and methods can be used with a wide variety of types, while still providing compile-time type checking to ensure that only compatible types are used.

For example, consider the following hierarchy of types:

* **class** Animal {}
* **class** Mammal **extends** Animal {}
* **class** Giraffe **extends** Mammal {}

In this example, the class 'Animal' is the base type, and 'Mammal' and 'Giraffe' are subtypes. This hierarchy can be used in generic code by specifying type parameters that are bounded by the base type, such as:

* **class** MyClass<T **extends** Animal> {
* T value;
* **public** **void** setValue(T value) {
* **this**.value = value;
* }
* **public** T getValue() {
* **return** value;
* }
* }

In this example, the type parameter 'T' is bounded by the type 'Animal', which means that it can only be instantiated with types that are subtypes of 'Animal'. For example:

* MyClass<Giraffe> myClass = **new** MyClass<>();
* myClass.setValue(**new** Giraffe());
* Giraffe giraffe = myClass.getValue();

In this example, the type argument 'Giraffe' is a subtype of 'Animal', so it is compatible with the type parameter 'T', and the code will compile and run correctly. If an incompatible type were used, such as 'String', the code would not compile, and an error would be reported.

### 25. Can you use multiple type parameters in Java Generics?

A. Yes, you can use multiple type parameters in Java Generics. To do so, you simply specify multiple type parameters separated by commas, within angle brackets, like this:

* **class** MyClass<T1, T2> {
* T1 value1;
* T2 value2;
* **public** **void** setValue1(T1 value1) {
* **this**.value1 = value1;
* }
* **public** **void** setValue2(T2 value2) {
* **this**.value2 = value2;
* }
* **public** T1 getValue1() {
* **return** value1;
* }
* **public** T2 getValue2() {
* **return** value2;
* }
* }

In this example, the class 'MyClass' has two type parameters, 'T1' and 'T2', which can be used to specify the types of the values 'value1' and 'value2', respectively. When you create an instance of 'MyClass', you specify the types to use for 'T1' and 'T2', like this:

* MyClass<String, Integer> myClass = **new** MyClass<>();
* myClass.setValue1("Hello");
* myClass.setValue2(42);
* String value1 = myClass.getValue1();
* Integer value2 = myClass.getValue2();

In this example, the type arguments 'String' and 'Integer' are used for 'T1' and 'T2', respectively. The code will compile and run correctly, and the values of 'value1' and 'value2' will be of type 'String' and 'Integer', respectively.

### 26. What is the difference between generic types and non-generic types in Java?

A.

| **Generic Types** | **Non-generic Types** |
| --- | --- |
| Parameterized with type parameters | No type parameters |
| Improved type safety | No type safety improvement |
| Reduced type casting | No reduction in type casting |
| More readable code | Harder to maintain code |
| Less prone to runtime errors | Prone to runtime errors |

Generic types and non-generic types are two different ways of defining classes and methods in Java.

A non-generic type, also known as a raw type, is a class or method that does not use any type parameters. For example:

* **class** MyClass {
* Object value;
* **public** **void** setValue(Object value) {
* **this**.value = value;
* }
* **public** Object getValue() {
* **return** value;
* }
* }

In this example, the class 'MyClass' is a non-generic type because it does not use any type parameters. The 'value' field and the 'setValue' and 'getValue' methods all use the raw type 'Object', which is a supertype of all classes in Java.

A generic type, on the other hand, uses type parameters to specify the types of values it can work with. For example:

* **class** MyClass<T> {
* T value;
* **public** **void** setValue(T value) {
* **this**.value = value;
* }
* **public** T getValue() {
* **return** value;
* }
* }

In this example, the class 'MyClass' is a generic type because it uses the type parameter 'T' to specify the type of the 'value' field and the type of the 'setValue' and 'getValue' methods. When you create an instance of 'MyClass', you specify the type to use for 'T', like this:

* MyClass<String> myClass = **new** MyClass<>();
* myClass.setValue("Hello");
* String value = myClass.getValue();

In this example, the type argument String is used for 'T', and the code will compile and run correctly, with the value of 'value' being of type 'String'.

The main advantage of using generic types is that they improve type safety by ensuring that the type of values being used with the class or method are compatible with the type parameters. This helps to catch errors at compile time, rather than at runtime.

### 27. How do you create a generic method that returns a type parameter in Java?

A. To create a generic method that returns a type parameter in Java, you can declare a type parameter in the method signature within angle brackets <>. Then, use the type parameter as the return type of the method.

Here's an example:

* **public** **static** <T> T genericMethod(T t) {
* **return** t;
* }

In this example, the type parameter 'T' is declared in the method signature, and is used as the return type of the method. The type parameter can be any reference type, such as an object or a custom class.

### 28. Can you overload a generic method in Java?

A. Yes, you can overload a generic method in Java. Overloading occurs when two or more methods in the same class have the same name but different parameter lists. The return type of the methods can be different or the same. In the case of generic methods, the type parameter list can be different, allowing for different implementations of the same method name.

Here's an example:

* **public** **static** <T> T genericMethod(T t) {
* **return** t;
* }
* **public** **static** <U> U genericMethod(U u) {
* **return** u;
* }

In this example, two methods with the same name 'genericMethod' are overloaded with different type parameters 'T' and 'U'. The methods are considered distinct and can be called with different arguments without any ambiguity.