Without Generics

import java.util.ArrayList;  
  
public class Main {  
 public static void main(String[] args) {  
 ArrayList list = new ArrayList();  
 list.add("Hello");  
 list.add(123);  
 list.add(3.14);  
  
 String str = (String) list.get(0);  
 String str1 = (String) list.get(1);  
  
 }  
}

Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class java.lang.String (java.lang.Integer and java.lang.String are in module java.base of loader 'bootstrap')  
 at com.engineeringdigest.corejava.Main.main(Main.java:13)

Above code has 3 major issues

1. No Type safety
2. Manual casting
3. No Compile Time checking

These issues can be solved by Generics

import java.util.ArrayList;  
  
public class Main {  
 public static void main(String[] args) {  
 ArrayList<String> list = new ArrayList<>();  
 list.add("Hello");  
 list.add("World");  
 String s = list.get(0);  
 String s1 = list.get(1);  
  
 }  
}

Now let’s take another example

**Generic Types**

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

public class Main {  
 public static void main(String[] args) {  
 Box box = new Box();  
 box.setValue(1);  
 String i = (String) box.getValue(); // EXCEPTION !!!  
 System.out.println(i);  
 }  
}

Now we will make Box Generic class, but before that we will study Generic Types.

Generic types allow you to define a class, interface, or method with placeholders (type parameters) for the data types they will work with. This enables code reusability and type safety, as it allows you to create classes, interfaces, or methods that can operate on various types without needing to rewrite the code for each type.

A **generic type** is a class or interface that is parameterized over types. For example, a generic class can work with any type specified by the user, and that type can be enforced at compile time.

The syntax for a generic type is:

class ClassName<T> {  
 // Class body  
}

Where T is the type parameter, which can be any valid identifier. Conventionally, single-letter names are used for type parameters, such as T for Type, E for Element, K for Key, V for Value, etc.

public class Box<T> { // one or more type parameters  
// These type parameters are placeholders that are replaced with specific types when the class is instantiated.  
 private T value;  
  
 public T getValue() {  
 return value;  
 }  
  
 public void setValue(T value) {  
 this.value = value;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Box<Integer> box = new Box<>(); // Box is now type-safe  
 box.setValue(1); // No issue, it's an Integer  
 Integer i = box.getValue(); // No casting needed  
 System.out.println(i);  
 }  
}

Here, Box<T> is a generic class. The type parameter T will be replaced with a specific type when an object of Box is created. Now, the Box class is type-safe, and we will not encounter the ClassCastException because the types are enforced at compile time.

*So, In simpler terms,****generics allow you to write code that can work with any object type while ensuring type safety at compile time.***

Generics help us write more flexible and reusable code. For example, without generics, we would have to write multiple versions of the same class to handle different data types, leading to code duplication.

*A generic class can have more than one type parameter.*

class Pair<K, V> {  
 private K key;  
 private V value;  
  
 public Pair(K key, V value) {  
 this.key = key;  
 this.value = value;  
 }  
  
 public K getKey() {  
 return key;  
 }  
  
 public V getValue() {  
 return value;  
 }  
}

This class can be used to store pairs of related data (like key-value pairs).

public class Main {  
 public static void main(String[] args) {  
 Pair<String, Integer> pair = new Pair<>("Age", 30);  
 System.out.println("Key: " + pair.getKey()); // Prints: Key: Age  
 System.out.println("Value: " + pair.getValue()); // Prints: Value: 30  
 }  
}

Here, the Pair<K, V> class has two type parameters K and V, and you can specify the types when you create an instance of Pair.

**Type Parameter Naming Conventions**

While you can name type parameters anything, the convention is to use single letters that describe the purpose of the type parameter:

* T: Type
* E: Element (used in collections)
* K: Key (used in maps)
* V: Value (used in maps)
* N: Number

For example, in the java.util.Map<K, V> interface:

* K stands for the key type
* V stands for the value type

Map<String, Integer> map = new HashMap<>();  
map.put("One", 1);  
map.put("Two", 2);

**Generic Interface**

A **generic interface** in Java allows you to define an interface with type parameters. This means that the interface can work with any type specified at the time of implementation. Generic interfaces are commonly used when the type of the objects that the interface deals with is not known until runtime.

You declare a generic interface in the same way you would declare a generic class or method, using angle brackets <> to specify type parameters. Here's a basic example of a generic interface:

interface Container<T> {  
 void add(T item);  
 T get();  
}

In this example, T is the generic type parameter for the Container interface. The add method accepts an argument of type T, and the get method returns a value of type T.

When you implement a generic interface, you need to specify the type for the generic parameter, or you can continue to make the implementation generic by using type parameters.

**Implementing with a specific type**

class StringContainer implements Container<String> {  
 private String item;  
  
 @Override  
 public void add(String item) {  
 this.item = item;  
 }  
  
 @Override  
 public String get() {  
 return item;  
 }  
}

In this example, StringContainer implements the Container interface with String as the specified type parameter.

**Implementing a generic interface generically**

class GenericContainer<T> implements Container<T> {  
 private T item;  
  
 @Override  
 public void add(T item) {  
 this.item = item;  
 }  
  
 @Override  
 public T get() {  
 return item;  
 }  
}

In this case, the class GenericContainer remains generic and can work with any type, just like the Container interface.

**Generic Interfaces with Multiple Type Parameters**

A generic interface can have multiple type parameters. This is useful when you need to work with more than one type in the same interface.

interface Pair<K, V> {  
 K getKey();  
 V getValue();  
}

Here, the Pair interface defines two generic type parameters K and V. Implementing this interface will require specifying both types.

class KeyValuePair<K, V> implements Pair<K, V> {  
 private K key;  
 private V value;  
  
 public KeyValuePair(K key, V value) {  
 this.key = key;  
 this.value = value;  
 }  
  
 @Override  
 public K getKey() {  
 return key;  
 }  
  
 @Override  
 public V getValue() {  
 return value;  
 }  
}

You can create an instance of KeyValuePair like this:

Pair<String, Integer> pair = new KeyValuePair<>("Age", 30);  
System.out.println(pair.getKey() + ": " + pair.getValue());

**Type Parameters and Bounded Types in Interfaces**

Just like generic classes, you can use bounded type parameters in generic interfaces to restrict the types that can be used as arguments.

interface NumberContainer<T extends Number> {  
 void add(T item);  
 T get();  
}

In this example, the type parameter T is restricted to subclasses of Number, so only numeric types like Integer, Double, etc., can be used.

class IntegerContainer implements NumberContainer<Integer> {  
 private Integer item;  
  
 @Override  
 public void add(Integer item) {  
 this.item = item;  
 }  
  
 @Override  
 public Integer get() {  
 return item;  
 }  
}

Attempting to implement the NumberContainer interface with a non-numeric type (like String) would result in a compile-time error.

**Wildcards with Generic Interfaces**

You can use **wildcards** when dealing with generic interfaces to allow for more flexibility with the types used at runtime. Wildcards allow an implementation to be more permissive about the types of parameters it accepts.

class WildcardExample {  
 public static void printContainer(Container<?> container) {  
 System.out.println(container.get());  
 }  
}

Here, the wildcard ? is used, meaning that printContainer can accept a Container of any type.

**Real-World Example of Generic Interfaces**

A common example of a generic interface in Java is the Comparable<T> interface, which is used to impose a natural ordering on objects.

class Employee implements Comparable<Employee> {  
 private String name;  
 private int age;  
  
 public Employee(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 @Override  
 public int compareTo(Employee other) {  
 return Integer.compare(this.age, other.age);  
 }  
}

In this example, the Comparable<Employee> interface ensures that the Employee objects can be compared based on their age.

**Summary**

* A generic interface allows you to write interfaces that can handle various types specified at the time of implementation.
* Type parameters are used to define the type of objects that the interface will work with.
* You can implement a generic interface by specifying the type or by keeping the implementation generic.
* Multiple type parameters can be used to work with more than one type at a time.
* Bounded type parameters restrict the types that can be passed to the generic interface.
* Wildcards can be used in generic interfaces for flexibility in accepting various types.

Generic interfaces in Java are powerful tools that increase the reusability and flexibility of your code by allowing you to abstract behaviour across multiple types, while maintaining type safety at compile time.

**Generics in enums**

Enums are inherently type-safe. You cannot assign a value to an enum that is not part of the defined constants. For example:

Day day = Day.MONDAY; // Type-safe  
// Day day = "MONDAY"; // Compile-time error

However, enums alone do not support generics. To add type parameters to an enum, you need to define generic methods or use enums with generic classes or interfaces.

**Generic Methods in Enums**

Enums can contain generic methods just like normal classes. This allows the enum to perform operations with type parameters. Here is an example of an enum with a generic method:

enum Operation {  
 ADD, SUBTRACT, MULTIPLY, DIVIDE;  
  
 public <T extends Number> double apply(T a, T b) {  
 switch (this) {  
 case ADD:  
 return a.doubleValue() + b.doubleValue();  
 case SUBTRACT:  
 return a.doubleValue() - b.doubleValue();  
 case MULTIPLY:  
 return a.doubleValue() \* b.doubleValue();  
 case DIVIDE:  
 return a.doubleValue() / b.doubleValue();  
 default:  
 throw new AssertionError("Unknown operation: " + this);  
 }  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 double result1 = Operation.ADD.apply(10, 20);  
 double result2 = Operation.MULTIPLY.apply(5.5, 4);  
 System.out.println(result1); // Output: 30.0  
 System.out.println(result2); // Output: 22.0  
 }  
}

In this example, the apply method accepts two parameters of type T (where T is bounded by Number). The method applies different operations based on the enum value. This ensures that any numeric types (such as Integer, Double, Float, etc.) can be used in the method.

**Generic Enums with a Type Parameter**

While you can’t define a generic type parameter at the enum declaration level (because enums can’t be generic themselves), you can achieve a similar effect by using an enum with a generic class or by adding a generic interface.

interface Calculator<T> {  
 T calculate(T a, T b);  
}  
  
enum ArithmeticOperation implements Calculator<Integer> {  
 ADD {  
 @Override  
 public Integer calculate(Integer a, Integer b) {  
 return a + b;  
 }  
 },  
 SUBTRACT {  
 @Override  
 public Integer calculate(Integer a, Integer b) {  
 return a - b;  
 }  
 };  
  
 // Additional operations can be added in the same way.  
}  
  
public class Main {  
 public static void main(String[] args) {  
 int result1 = ArithmeticOperation.ADD.calculate(10, 5);  
 int result2 = ArithmeticOperation.SUBTRACT.calculate(10, 5);  
 System.out.println(result1); // Output: 15  
 System.out.println(result2); // Output: 5  
 }  
}

Here, the ArithmeticOperation enum implements the Calculator interface, which is generic. This allows for an enum to perform operations based on the types specified in the interface.

**Generic Constructors**

A generic constructor can be defined in a generic class. However, the generic type parameter for the constructor may be different from the generic type parameter of the class:

class Test<T> {  
 private T value;  
  
 // Generic constructor  
 <U> Test(U input) {  
 System.out.println(input.getClass().getName());  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 Test<Integer> test = new Test<>(12.34); // Output: java.lang.Double  
 }  
}

In this case, the generic type parameter U is for the constructor only and is independent of the class type parameter T.

**Multiple Type Parameters in Constructors**

class Pair {  
 // Generic constructor with two type parameters  
 <A, B> Pair(A first, B second) {  
 System.out.println("First: " + first + ", Second: " + second);  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 new Pair(10, "Ten"); // Integer and String  
 new Pair(3.14, 42); // Double and Integer  
 }  
}

Here, The constructor accepts two parameters, each of a different type A and B. The constructor prints out both values, showing how it can handle multiple type parameters.

**Bounded Type Parameters in Generic Constructors**

You can also apply bounds to the type parameters in generic constructors, just like in generic classes or methods. Bounded type parameters restrict the types that can be used as arguments in the constructor.

class NumberPrinter {  
 // Bounded type parameter for generic constructor  
 <T extends Number> NumberPrinter(T number) {  
 System.out.println("Number: " + number);  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 new NumberPrinter(100); // Integer is a subclass of Number  
 new NumberPrinter(3.14); // Double is a subclass of Number  
   
 // The following would cause a compile-time error because String is not a subclass of Number  
 // new NumberPrinter("Hello");   
 }  
}

In above example, the constructor is generic and accepts only arguments of types that are subclasses of Number, such as Integer, Double, Float, etc.  
Trying to pass a type that does not extend Number (like String) results in a compilation error.

**Conclusion**

Generic constructors in Java provide flexibility and reusability, especially in situations where you need to deal with different types during object creation. They allow you to separate the type logic of the class from the constructor and ensure that the appropriate type is inferred or specified at the time of object creation.

* In non-generic classes, generic constructors allow creating instances using any type.
* In generic classes, the generic constructor’s type parameter can be independent of the class’s type parameter.
* You can have multiple type parameters, bounded type parameters, and type inference to handle complex situations with type-safe constructor definitions.

**Generic Static Members**

One restriction with generic classes is that **static members cannot use type parameters**. The reason for this is that static members belong to the class itself rather than to any instance, and the type parameter is tied to an instance

class MyClass<T> {  
 private T instanceVar; // Valid  
 static T staticVar; // Invalid - static members cannot use T  
}

You can still create a static method using its own type parameter:

class MyClass<T> {  
 public static <U> void staticMethod(U param) {  
 System.out.println(param);  
 }  
}

In this example, the static method staticMethod has its own type parameter U, which is different from the class’s type parameter T.

**Generic Methods**

Generic methods are methods that allow for type parameters, meaning they can be defined to accept different types of parameters in a type-safe way. By using generic methods, you can write code that works with any type and enforces compile-time type checking.

The syntax of a generic method is similar to that of a regular method, with the addition of type parameters declared in angle brackets (<>) before the return type. The type parameter can be any valid identifier but is often denoted by single capital letters like T, E, K, V, etc.

**Basic Syntax**

public <T> void methodName(T parameter) {  
 // method body  
}

Here:

* <T> is the type parameter.
* T can be any type such as Integer, String, Float, or any user-defined class.

**Example 1:**

public class GenericMethodExample {  
 // Generic method  
 public <T> void printArray(T[] array) {  
 for (T element : array) {  
 System.out.print(element + " ");  
 }  
 System.out.println();  
 }  
  
 public static void main(String[] args) {  
 GenericMethodExample example = new GenericMethodExample();  
   
 Integer[] intArray = {1, 2, 3, 4, 5};  
 String[] stringArray = {"A", "B", "C", "D"};  
   
 // Using the generic method  
 example.printArray(intArray); // Output: 1 2 3 4 5  
 example.printArray(stringArray); // Output: A B C D  
 }  
}

**Example 2:**

A generic method can also accept multiple type parameters. You can specify more than one type parameter by separating them with commas within the angle brackets.

public class GenericMethodExample {  
 public <T, U> void printTwoItems(T item1, U item2) {  
 System.out.println(item1 + " and " + item2);  
 }  
  
 public static void main(String[] args) {  
 GenericMethodExample example = new GenericMethodExample();  
   
 example.printTwoItems(10, "Apples"); // Output: 10 and Apples  
 example.printTwoItems("Hello", 3.14); // Output: Hello and 3.14  
 }  
}

**Example 3:**Generic methods can also be defined in static context. Since static methods belong to the class, not to instances, the generic type parameter for a static method is defined at the method level, not at the class level.

public class GenericMethodExample {  
 // Generic static method  
 public static <T> void printElement(T element) {  
 System.out.println("Element: " + element);  
 }  
  
 public static void main(String[] args) {  
 GenericMethodExample.printElement(42); // Output: Element: 42  
 GenericMethodExample.printElement("Generics in Java"); // Output: Element: Generics in Java  
 }  
}

**Example 4: Generic Methods and Method Overloading**

public class GenericMethodExample {  
 public <T> void display(T element) {  
 System.out.println("Generic display: " + element);  
 }  
  
 public void display(Integer element) {  
 System.out.println("Integer display: " + element);  
 }  
  
 public static void main(String[] args) {  
 GenericMethodExample example = new GenericMethodExample();  
   
 example.display(42); // Calls the Integer display method  
 example.display("Generics"); // Calls the generic display method  
 }  
}

In the above example, even though we have a generic method display, the non-generic display method that accepts an Integer gets called when we pass an integer value.

**Example 5: Generic Methods and Varargs**

public class GenericMethodExample {  
 public <T> void display(T element) {  
 System.out.println("Generic display: " + element);  
 }  
  
 public void display(Integer element) {  
 System.out.println("Integer display: " + element);  
 }  
  
 public static void main(String[] args) {  
 GenericMethodExample example = new GenericMethodExample();  
   
 example.display(42); // Calls the Integer display method  
 example.display("Generics"); // Calls the generic display method  
 }  
}

**How Generics Work (Type Erasure)**

Generics in Java are implemented through a process called **type erasure**. This means that the generic types are replaced with their bounds or Object during compilation, and the resulting bytecode contains only ordinary classes, methods, and fields.

Consider the following generic class Box<T> and a specific instantiation where T is replaced with String:

public class Box<T> {  
 private T item;  
  
 public void setItem(T item) {  
 this.item = item;  
 }  
  
 public T getItem() {  
 return item;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Box<String> stringBox = new Box<>();  
 stringBox.setItem("Hello, Generics!");  
 String item = stringBox.getItem();  
 System.out.println(item);  
 }  
}

When you instantiate Box<String>, the Java compiler checks the type at compile-time and ensures that only String objects are assigned to the Box. However, during compilation, this type information (T = String) is **erased**, and the generic type T is replaced with its bound, which defaults to Object if no explicit bound is specified.

After type erasure, the generic class Box<T> is transformed by the compiler into the following code:

public class Box {  
 private Object item; // `T` is replaced with `Object`  
  
 public void setItem(Object item) { // `T` replaced with `Object`  
 this.item = item;  
 }  
  
 public Object getItem() { // `T` replaced with `Object`  
 return item;  
 }  
}

At runtime, there is no knowledge of the type T being String. The class Box only knows that it stores an Object, and all operations on the generic type are done as if it were an Object. However, the code that interacts with Box<String> still works safely because the compiler has enforced the correct type usage at compile-time.

For example, when you write:

Box<String> stringBox = new Box<>();  
stringBox.setItem("Hello, Generics!");  
String item = stringBox.getItem();

The following happens at runtime:

1. **Type-Safe Operations:** At compile-time, the type safety of Box<String> is ensured. The compiler makes sure that only String objects are added to the Box.
2. **Type Erasure:** At runtime, the JVM treats the stringBox as a Box<Object>. When you call getItem(), the JVM returns an Object (not a String), but the compiler-generated code has inserted an implicit cast to String during compilation:

String item = (String) stringBox.getItem(); // Implicit cast inserted by the compiler

Even though type erasure replaces T with Object, the cast back to String ensures that the original type is respected. If the type doesn't match the expected one, the program will throw a ClassCastException at runtime.

**Bounded Type Parameters**

When you define a generic class or method, you can specify a bound for the type parameter. This means that the type argument passed to the generic type must either extend a specific class or implement a specific interface.

<T extends ClassOrInterface>

Here, T is the generic type, and ClassOrInterface is the upper bound. This means T must be either ClassOrInterface itself or a subclass of ClassOrInterface.

**Example 1: Bounded Type Parameters with Classes**

Let’s say we want a method that operates only on types that extend the class Number. We can restrict the generic type parameter by bounding it to Number.

class GenericClass<T extends Number> {  
 private T value;  
  
 public GenericClass(T value) {  
 this.value = value;  
 }  
  
 public void display() {  
 System.out.println("Value: " + value);  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 GenericClass<Integer> intObj = new GenericClass<>(123);  
 intObj.display(); // Output: Value: 123  
  
 GenericClass<Double> doubleObj = new GenericClass<>(45.67);  
 doubleObj.display(); // Output: Value: 45.67  
  
 // The following will cause a compile-time error because String does not extend Number  
 // GenericClass<String> strObj = new GenericClass<>("Hello");  
 }  
}

Here, T extends Number means that T can only be Integer, Double, Float, Long, or any other subclass of Number. A type like String is not allowed.

**Example 2: Multiple Bounds**

In Java Generics, **multiple bounds** allow you to specify that a type parameter must satisfy multiple constraints. In simpler terms, a type parameter can be constrained by more than one bound (interfaces and/or classes). This is done using the syntax of a single class type followed by multiple interfaces. The class type must be listed first, followed by any interfaces.

The general syntax for multiple bounds in Java Generics is as follows:

<T extends ClassType & Interface1 & Interface2>

Here:

* ClassType is a class that T must extend (inherit).
* Interface1 and Interface2 are interfaces that T must implement.

Key rules:

* A class type must always be the first bound, followed by interfaces.
* You can have multiple interface bounds but only one class bound.

interface Printable {  
 void print();  
}  
  
class MyNumber extends Number implements Printable {  
 private final int value;  
  
 public MyNumber(int value) {  
 this.value = value;  
 }  
  
 @Override  
 public void print() {  
 System.out.println("MyNumber: " + value);  
 }  
  
 @Override  
 public int intValue() {  
 return value;  
 }  
  
 @Override  
 public long longValue() {  
 return value;  
 }  
  
 @Override  
 public float floatValue() {  
 return value;  
 }  
  
 @Override  
 public double doubleValue() {  
 return value;  
 }  
}  
  
class Boxx<T extends Number & Printable>{  
 private T item;  
  
 public Boxx(T item) {  
 this.item = item;  
 }  
  
 public void display() {  
 item.print();  
 }  
  
 public T getItem() {  
 return item;  
 }  
}  
  
public class Test{  
 public static void main(String[] args) {  
 MyNumber myNumber = new MyNumber(12);  
 Boxx<MyNumber> box = new Boxx<>(myNumber);  
 box.display();  
 }  
}

This example demonstrates how to use multiple bounds in a simple and straightforward manner. The Box class can only work with objects that are both a subclass of Number and implement the Printable interface. This ensures that the objects it handles are numeric and can also be printed.

**Order of Bounds**

If you reverse the order of the bounds by placing an interface before the class, you’ll encounter a compile-time error:

// Compile-time error  
<T extends Runnable & Animal>

The correct order is to place the class first, followed by the interfaces:

<T extends Animal & Runnable>

**Type Bound Restrictions**

* You cannot have more than one class as a bound because Java does not support multiple inheritance for classes. However, you can have as many interface bounds as you need.
* If no class bound is specified, the type parameter defaults to extending Object implicitly.

**No Class Bound (Only Interfaces)**

When there is no class bound, and only interfaces are used, multiple bounds are still valid, but Object is implicitly used as the class bound.

interface Walkable {  
 void walk();  
}  
  
interface Talkable {  
 void talk();  
}  
  
class Person implements Walkable, Talkable {  
 public void walk() {  
 System.out.println("Person is walking.");  
 }  
  
 public void talk() {  
 System.out.println("Person is talking.");  
 }  
}  
  
class Actions {  
 public static <T extends Walkable & Talkable> void perform(T creature) {  
 creature.walk();  
 creature.talk();  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Person person = new Person();  
 Actions.perform(person);  
 }  
}

**Benefits of Multiple Bounds**

1. **Code Flexibility:** Multiple bounds allow the generic class or method to work with types that meet several criteria, improving flexibility.
2. **Compile-Time Safety:** By enforcing multiple bounds, Java ensures that only types that fulfill all conditions are allowed, avoiding runtime errors.
3. **Enhanced Reusability:** Multiple bounds let you reuse generic logic across different types, as long as they satisfy the constraints.

**Key Takeaways**

* Multiple bounds allow a type parameter to extend one class and implement multiple interfaces.
* The class type must always be listed before any interfaces.
* Multiple bounds can be useful when you need to enforce multiple behavioral constraints on a type.
* There is a limit of one class bound, but no limit on the number of interface bounds.

**Example 3: Generic Methods with Bounded Type Parameters**

public class Util {  
 public static <T extends Number> void printDoubleValue(T value) {  
 System.out.println(value.doubleValue());  
 }  
  
 public static void main(String[] args) {  
 printDoubleValue(10); // Output: 10.0  
 printDoubleValue(3.14); // Output: 3.14  
  
 // The following will cause a compile-time error because String does not extend Number  
 // printDoubleValue("Hello");  
 }  
}

In this example, the generic method printDoubleValue has a type parameter T that is bounded by Number. This ensures that only subclasses of Number can be passed as arguments.

Bounded type parameters are a powerful feature of Java Generics that allow you to enforce constraints on generic types, ensuring that they meet certain requirements, which enhances both type safety and flexibility.

**Wildcards in Generics**

In Java Generics, wildcards (?) are a special kind of type argument that can be used in method arguments or class definitions to represent an unknown type. They allow for more flexible and dynamic code by letting the type be specified later or be more loosely defined.

Wildcards are useful when you don’t know the exact type at the time of defining a class or method, or when you want to allow a range of types rather than a single specific one.

**Basic syntax**

List<?> list;

Here, ? represents an unknown type. It means the list can contain elements of any type, but you cannot perform type-specific operations on the elements.

**Why Use Wildcards?**

Wildcards provide flexibility, especially when you want to write code that can work with different types, but you don’t want to be too specific about the types. They are often used when you want a method to accept arguments of generic types without tying it down to a particular class.

For instance, if you need to write a method that can handle a list of any object type (e.g., List of Integer, List of String, etc.), wildcards allow you to do that.

**Wildcards in Method Parameters**

Wildcards are typically used as method parameters where the exact type isn’t important. They allow a method to accept different types of collections or generic classes.

public void printList(List<?> list) {  
 for (Object element : list) {  
 System.out.println(element);  
 }  
}

In this example, List<?> can be a list of any object type, like List<String>, List<Integer>, etc. The method can process them all without needing to know the exact type.

List<String> stringList = Arrays.asList("Apple", "Banana", "Orange");  
List<Integer> integerList = Arrays.asList(1, 2, 3);  
  
printList(stringList); // Works  
printList(integerList); // Works

Here, the method is flexible enough to print elements from both List<String> and List<Integer>.

**When Wildcards are Useful**

Wildcards come in handy when you have methods that need to work on generic types, but the method doesn’t need to know the exact type. For example, you might want a method that can accept collections of different types (e.g., List<String>, List<Integer>) without being tied to a specific type.

Consider the following scenario:

public void processElements(List<?> elements) {  
 // Cannot add elements because we don't know the type.  
 for (Object element : elements) {  
 System.out.println(element);  
 }  
}

This method will work with any List, but since the type is unknown, we cannot safely add new elements to the list. We can read from it, but not write to it.

**Limitations of Wildcards**

One key limitation when using wildcards is that they are generally used in a read-only context. This means that while you can read elements from the collection, you cannot modify it in a type-safe manner.

**For example:**

List<?> wildcardList = new ArrayList<String>();  
wildcardList.add("Hello"); // This will cause a compilation error

This is because the compiler cannot guarantee type safety when adding elements.

**Summary of Wildcards**

* ? represents an unknown type.
* It is often used when the exact type isn’t relevant or needs to remain flexible.
* Wildcards are useful in read-only scenarios.
* You can read elements, but adding new elements is generally not allowed because the exact type is unknown.

**Upper-Bounded Wildcards**

In Java Generics, an upper-bounded wildcard restricts the types that can be passed as arguments to a parameterized type. We use the wildcard ? with an upper bound to indicate that the type can be any class that is a subclass of a specified class (including the class itself).

The syntax for upper-bounded wildcards is:

<? extends SomeClass>

This means that the type parameter can be any type that is SomeClass or a subclass of SomeClass.

**Why Use Upper-Bounded Wildcards?**

Upper-bounded wildcards are useful when you want to read data from a generic structure but you don’t need to modify it (except in certain restricted ways). This is sometimes referred to as “covariance.”

For example, if you have a method that processes a list of numbers, but it doesn’t need to add any new elements to the list, you can use an upper-bounded wildcard to allow the method to accept lists of Integer, Double, or any other subtype of Number.

**Basic Example**

Consider a method that calculates the sum of numbers in a list. You want this method to work for any type of numbers like Integer, Double, etc. You can use an upper-bounded wildcard to achieve this flexibility:

import java.util.List;  
  
public class UpperBoundedWildcardDemo {  
 public static double sum(List<? extends Number> numbers) {  
 double total = 0.0;  
 for (Number number : numbers) {  
 total += number.doubleValue();  
 }  
 return total;  
 }  
  
 public static void main(String[] args) {  
 List<Integer> intList = List.of(1, 2, 3);  
 List<Double> doubleList = List.of(1.1, 2.2, 3.3);  
  
 System.out.println("Sum of integers: " + sum(intList));  
 System.out.println("Sum of doubles: " + sum(doubleList));  
 }  
}

**Explanation**

* List<? extends Number> allows the sum method to accept a List of any type that extends Number, such as Integer, Double, Float, etc.
* The method can read elements from the list, but it cannot add elements to it (because it only knows that the elements are some subtype of Number, but it doesn't know the exact type).
* This flexibility is achieved using the upper-bounded wildcard ? extends Number.

**Covariance in Generics**

Covariance allows a generic type to be assigned to another generic type with a broader bound. For example, a List<Integer> can be assigned to a List<? extends Number> but not the other way around.

List<? extends Number> numbers;  
numbers = List.of(1, 2, 3); // List of Integer is valid  
numbers = List.of(1.1, 2.2, 3.3); // List of Double is valid

You can use the wildcard type <? extends Number> to access elements from the list but cannot add elements to the list, except null:

List<? extends Number> numbers = List.of(1, 2, 3);  
// numbers.add(4); // Error: You can't add elements, because it doesn't know the exact type  
numbers.add(null); // Valid: You can add null

**Restricting Method Parameters**

Another use case of upper-bounded wildcards is when you want to restrict the types of arguments that a method can accept. You can ensure that the method can accept only arguments that are subtypes of a particular type.

public static void printNumbers(List<? extends Number> list) {  
 for (Number number : list) {  
 System.out.println(number);  
 }  
}  
  
List<Integer> intList = List.of(1, 2, 3);  
List<Double> doubleList = List.of(1.1, 2.2, 3.3);  
  
printNumbers(intList); // Valid  
printNumbers(doubleList); // Valid

This method accepts List<Integer>, List<Double>, or any other list that contains a subtype of Number.

**Upper-Bounded Wildcards and Inheritance**

Upper-bounded wildcards work well with inheritance in Java. For instance, you may want to create a method that works with a list of a specific subclass and all its subclasses.

Let’s take an example with an inheritance hierarchy:

class Animal {  
 public void sound() {  
 System.out.println("Some sound");  
 }  
}  
  
class Dog extends Animal {  
 @Override  
 public void sound() {  
 System.out.println("Bark");  
 }  
}  
  
class Cat extends Animal {  
 @Override  
 public void sound() {  
 System.out.println("Meow");  
 }  
}  
  
public class AnimalDemo {  
 public static void makeSound(List<? extends Animal> animals) {  
 for (Animal animal : animals) {  
 animal.sound();  
 }  
 }  
  
 public static void main(String[] args) {  
 List<Dog> dogs = List.of(new Dog(), new Dog());  
 List<Cat> cats = List.of(new Cat(), new Cat());  
  
 makeSound(dogs); // Valid  
 makeSound(cats); // Valid  
 }  
}

In this example:

* The makeSound method works with lists of any type that is a subtype of Animal, such as Dog or Cat.
* The upper-bounded wildcard <? extends Animal> ensures that we can pass in lists of Dog, Cat, or any other subclass of Animal.

**Common Mistakes with Upper-Bounded Wildcards**

1. Trying to Add Elements: The most common mistake is trying to add elements to a collection with an upper-bounded wildcard. As mentioned earlier, you can only safely read from such collections but not add to them (except null).
2. Confusion with Invariance: Many developers assume that List<Dog> can be assigned to List<Animal>, but this is not allowed. Wildcards help to solve this issue, but you have to use them correctly.

**Conclusion**

Upper-bounded wildcards (<? extends T>) in Java are a powerful way to make your generic methods more flexible. They allow you to handle a variety of types in a single method or class, especially when you are only interested in reading from a collection, not modifying it.

Mastering this concept helps you leverage polymorphism and the flexibility of generics while maintaining type safety.

**Lower-Bounded Wildcards in Java Generics**

Lower-bounded wildcards are a feature of Java generics that allow us to specify that a type must be a supertype of a given class. They are written using the ? super syntax. This helps when you want to define a method or collection that can operate on objects of a given type or any of its supertypes. Let’s break down lower-bounded wildcards, starting from the basics and moving towards more advanced topics.

**Basics of Lower-Bounded Wildcards**

Lower-bounded wildcards restrict the type of the argument to a class and its supertypes (ancestors). The syntax ? super T means "some unknown type that is a superclass of T."

For example, List<? super Integer> can accept Integer, Number, or Object (since Integer is a subclass of Number, which is a subclass of Object).

public static void addNumbers(List<? super Integer> list) {  
 list.add(10);  
 list.add(20);  
}

In the code above, addNumbers() accepts a List that can hold any type which is a supertype of Integer. This means you can pass a List<Integer>, List<Number>, or List<Object> to this method.

**Use Cases of Lower-Bounded Wildcards**

1. **Adding to a Collection**: Lower-bounded wildcards are useful when you want to add objects to a collection. You can add objects of the specified type or any of its subclasses.
2. **Consumer-Producer Pattern**: Lower-bounded wildcards are also used in the consumer-producer pattern, where a method consumes values but does not produce them. The method can accept objects that are supertypes of a certain type, so that it can safely add elements to the collection.

public static void addToCollection(List<? super Number> list) {  
 list.add(1); // Integer (subclass of Number)  
 list.add(1.5); // Double (subclass of Number)  
}

In this case, list can be a List<Number> or a List<Object>. You can add both Integer and Double because both are subclasses of Number.

public static void printNumbers(List<? super Integer> list) {  
 for (Object obj : list) {  
 System.out.println(obj);  
 }  
}

Here, you can pass a List<Integer>, List<Number>, or List<Object>, and it will print all the elements in the list, regardless of the specific type.

**Restrictions with Lower-Bounded Wildcards**

Lower-bounded wildcards have some limitations that you need to be aware of:

1. **Cannot Retrieve Specific Elements**: When using lower-bounded wildcards, you lose the ability to retrieve elements of a specific type because you don’t know the exact type at runtime.
2. **Cannot Add Subtypes Outside of Bounds**: You cannot add elements to the list if they are not compatible with the lower bound.

public static void processList(List<? super Number> list) {  
 // We cannot retrieve a specific type from the list.  
 Object obj = list.get(0); // This is the only safe type to retrieve.  
}

Since you only know that the list can hold objects of type Number or its supertypes, retrieving an element from the list will return an Object.

public static void invalidAddition(List<? super Number> list) {  
 // list.add("String"); // Compilation error  
}

In the code above, trying to add a String will result in a compilation error because String is not a subtype of Number.

**Working with Mixed Types**

Lower-bounded wildcards are particularly useful when working with mixed types. Suppose you have a method that takes a list of Number or any of its supertypes and processes them generically.

public static void processMixedTypes(List<? super Integer> list) {  
 list.add(100); // Adding an integer works  
 list.add(200); // Adding another integer works  
 // Cannot safely retrieve anything other than Object  
}  
  
public static void main(String[] args) {  
 List<Object> objects = new ArrayList<>();  
 processMixedTypes(objects); // Works with List<Object>  
  
 List<Number> numbers = new ArrayList<>();  
 processMixedTypes(numbers); // Works with List<Number>  
}

In this case, you can pass either List<Object> or List<Number>, and the method will safely add integers to the list.

**Generics with Inheritance and Polymorphism**

Lower-bounded wildcards are also useful in situations involving inheritance and polymorphism.

class Animal {}  
class Dog extends Animal {}  
class Cat extends Animal {}  
  
public static void addAnimal(List<? super Dog> list) {  
 list.add(new Dog());  
 // list.add(new Cat()); // Compilation error: Cat is not a subclass of Dog  
}

In this example, addAnimal() accepts a list that can hold objects of type Dog or any of its supertypes (e.g., Animal or Object). You can add a Dog to the list, but not a Cat, since Cat is not a subclass of Dog.

**Covariance and Contravariance**

Lower-bounded wildcards in Java are an example of **contravariance** in generics. Contravariant generics allow you to assign a collection of objects of a more specific type to a reference that expects a more general type.

List<? super Integer> list = new ArrayList<Number>();  
list.add(123); // This works

Here, you can assign a List<Number> to List<? super Integer>, allowing the list to accept Integer values but still be considered a List<Number>.

**Best Practices with Lower-Bounded Wildcards**

* Use lower-bounded wildcards when you want to add elements to a collection. They are ideal when you are writing to a collection but not reading from it.
* Avoid using lower-bounded wildcards when you need to retrieve specific elements from a collection, as retrieving elements can only guarantee an Object type.
* Keep method signatures clear. If your method is intended to add objects of a specific type or its supertypes, using ? super is appropriate. Otherwise, consider using more restrictive generics or specific types.

**Conclusion**

Lower-bounded wildcards (? super T) are a powerful tool in Java Generics, allowing for flexibility when writing to a collection or dealing with a consumer-only pattern. They let you specify a range of acceptable types, starting from a specific type and including all its supertypes. Understanding when and how to use lower-bounded wildcards effectively can help make your code more flexible, especially when dealing with generic methods and collections. However, they should be used carefully, particularly when working with collections where retrieval of specific types is needed.

**Raw Types in Generics**

Raw Types in Genericsrefer to using a generic class or interface without specifying a type parameter. Prior to Java 5, collections were raw types, meaning they didn’t enforce type safety. Using raw types disables generics and causes potential type safety issues.

List list = new ArrayList(); // Raw type  
list.add("Hello");  
list.add(123); // No compile-time error

Using raw types can lead to ClassCastException at runtime, as the generic type system isn't enforced.

Type Safety with Generics:

List<String> list = new ArrayList<>();  
list.add("Hello");  
// list.add(123); // Compile-time error

Java still supports raw types for backward compatibility but discourages their usage.

Best Practice is to always specify the generic type to maintain type safety and avoid runtime errors. Avoid mixing generic and raw types in the same code, as it leads to unchecked warnings.

**Generic Exceptions in Java**

In Java, exceptions are objects that represent an error or unexpected condition during the execution of a program. All exceptions in Java are descendants of Throwable, which has two main subclasses:

* Exception: Represents checked exceptions (those that need to be declared or handled).
* RuntimeException: Represents unchecked exceptions (those that don’t need to be declared or handled).

Here’s an example of a basic custom exception:

class MyException extends Exception {  
 public MyException(String message) {  
 super(message);  
 }  
}

**Understanding Generic Exceptions**

Generics allow us to parameterize types (as discussed in earlier points). The concept of generics can be applied to exceptions as well. However, generic exceptions have certain restrictions due to how Java handles exceptions.

**Key Restriction:**

* You cannot create instances of generic types as exceptions: The Java runtime needs to be able to construct exceptions based on the exact type. But due to type erasure (discussed later), the JVM loses the specific type information at runtime.

Thus, the direct usage of generic types in exceptions is limited. For example, the following is not allowed:

class MyGenericException<T> extends Exception { // Compilation error  
 private T data;  
  
 public MyGenericException(T data) {  
 this.data = data;  
 }  
}

This will cause a compilation error because the type T is erased at runtime, and Java can't determine what specific exception to instantiate.

**Advanced Generic Exception Patterns**

Even though you can’t directly instantiate generic exceptions, there are still advanced patterns and techniques to incorporate generic behaviour in exceptions. Here’s how:

**Using Generics with Exception Messages or Fields**

You can include generic types in the fields or messages of exceptions instead of trying to make the exception class itself generic.

Example:

class DetailedException<T> extends Exception {  
 private T details;  
  
 public DetailedException(String message, T details) {  
 super(message);  
 this.details = details;  
 }  
  
 public T getDetails() {  
 return details;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 try {  
 throw new DetailedException<Integer>("An error occurred", 404);  
 } catch (DetailedException<Integer> e) {  
 System.out.println(e.getMessage()); // Output: An error occurred  
 System.out.println(e.getDetails()); // Output: 404  
 }  
 }  
}

In this case, the DetailedException class contains a generic field details that can hold extra information about the exception. This information could be anything, such as the problematic value or other diagnostic data.

**Throwing and Catching Generic Exceptions**

While you can’t create a fully generic exception class, you can create exceptions that hold generic data and use them in a flexible way by catching specific exceptions.

class CustomException<T> extends Exception {  
 private final T errorCode;  
  
 public CustomException(String message, T errorCode) {  
 super(message);  
 this.errorCode = errorCode;  
 }  
  
 public T getErrorCode() {  
 return errorCode;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 try {  
 throwException();  
 } catch (CustomException<Integer> e) {  
 System.out.println("Caught an exception with error code: " + e.getErrorCode());  
 }  
 }  
  
 public static void throwException() throws CustomException<Integer> {  
 throw new CustomException<>("Something went wrong", 1001);  
 }  
}

**Generic Methods with Exceptions**

You can also create generic methods that throw exceptions. This is useful when you want to define a method that may throw different types of exceptions, but you want to maintain type safety.

public class Main {  
 public static <T extends Exception> void throwGenericException(Class<T> exceptionClass) throws T {  
 try {  
 throw exceptionClass.getDeclaredConstructor(String.class).newInstance("Generic Exception");  
 } catch (Exception e) {  
 throw (T) e; // Casting to generic type  
 }  
 }  
  
 public static void main(String[] args) {  
 try {  
 throwGenericException(IllegalArgumentException.class);  
 } catch (IllegalArgumentException e) {  
 System.out.println(e.getMessage()); // Output: Generic Exception  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

Here, the throwGenericException method uses reflection to create an instance of the provided exception class. This allows you to create and throw different types of exceptions based on the class passed into the method.

**Challenges with Generics in Exceptions**

1. Type Erasure: As mentioned earlier, Java’s type erasure makes it difficult to retain generic type information at runtime, limiting the usefulness of generics in exception handling.
2. Checked vs. Unchecked Exceptions: Java has strict rules around checked exceptions. You can’t make a generic method throw arbitrary checked exceptions unless you declare those exceptions explicitly.
3. Runtime Type Information: Since exceptions are usually thrown and caught at runtime, the type of the generic parameter will be lost, which might make generic exceptions less practical in certain scenarios.

**Best Practices with Generic Exceptions**

1. Use generics in exception fields: Instead of making the exception class itself generic, use generic fields to store error-related information.
2. Avoid overly complex exception hierarchies: Keep the use of generics in exceptions simple to maintain readability and prevent confusion.
3. Type-safe exception handling: Always be mindful of how generic exceptions are used and caught to ensure type safety and code clarity.

**Conclusion**

Although Java imposes several limitations on using generics with exceptions (primarily due to type erasure and runtime constraints), you can still leverage generics in a limited capacity to enhance the functionality and flexibility of exceptions. These include using generic fields, methods that throw generic exceptions, and handling various types of exceptions while maintaining type safety.

Understanding these techniques allows you to make better use of Java’s generic features in advanced exception handling scenarios.

**Intersection Types in Java Generics**

**Intersection Types** in Java allow you to define a type that must satisfy multiple constraints simultaneously. Specifically, it allows an object to be of more than one type at the same time. You use an intersection type when you want a variable or method to work with multiple types, combining those types’ capabilities.

**Syntax of Intersection Types**

In Java, intersection types are represented using the & operator between types. This operator is used in the context of generic type bounds and can combine multiple interfaces (or a class and interfaces).

Here is the syntax:

<T extends InterfaceA & InterfaceB>

In this case, T must implement both InterfaceA and InterfaceB.

**Examples of Intersection Types**

**1. Simple Example: Combining Two Interfaces**

Let’s look at a basic example where we want to combine two interfaces using intersection types.

interface Printer {  
 void print();  
}  
  
interface Scanner {  
 void scan();  
}  
  
class AllInOneMachine implements Printer, Scanner {  
 public void print() {  
 System.out.println("Printing...");  
 }  
  
 public void scan() {  
 System.out.println("Scanning...");  
 }  
}  
  
public class IntersectionTypeExample {  
 public static <T extends Printer & Scanner> void useDevice(T device) {  
 device.print();  
 device.scan();  
 }  
  
 public static void main(String[] args) {  
 AllInOneMachine machine = new AllInOneMachine();  
 useDevice(machine);  
 }  
}

In this example:

* We have an AllInOneMachine class that implements both Printer and Scanner interfaces.
* The method useDevice takes a generic parameter T which must be both Printer and Scanner (i.e., the type T is an intersection of Printer and Scanner).
* The useDevice method calls both print() and scan() on the device.

**2. Multiple Interface Bounds**

You can extend the intersection to combine more than two interfaces as well.

interface Copier {  
 void copy();  
}  
  
public static <T extends Printer & Scanner & Copier> void useAdvancedDevice(T device) {  
 device.print();  
 device.scan();  
 device.copy();  
}

Here, the type T must implement all three interfaces: Printer, Scanner, and Copier. This ensures that the passed object will have all the functionalities provided by these interfaces.

**3. Intersection Type with Class and Interfaces**

You can also use intersection types to combine a class with one or more interfaces. However, there are limitations:

* You can only have one class in an intersection, and it must appear first.
* Multiple interfaces can follow after the class.

Example:

class Device {  
 void start() {  
 System.out.println("Device starting...");  
 }  
}  
  
public static <T extends Device & Printer & Scanner> void useSmartDevice(T device) {  
 device.start();  
 device.print();  
 device.scan();  
}

In this example:

* The type T must extend the class Device and implement both Printer and Scanner interfaces.
* This allows you to use methods from both the class and the interfaces in useSmartDevice.

**4. Intersection Types with Anonymous Classes and Lambdas**

Intersection types can be used in anonymous classes or lambdas where you want the object to satisfy multiple interfaces at once.

Example with anonymous classes:

public static void main(String[] args) {  
 useDevice(new Printer() {  
 @Override  
 public void print() {  
 System.out.println("Anonymous printing...");  
 }  
 });  
}

**5. Generic Type Bounds Using Intersection Types**

In more complex generic methods or classes, intersection types can be useful for creating flexible APIs.

class GenericDevice<T extends Device & Printer & Scanner> {  
 T device;  
  
 GenericDevice(T device) {  
 this.device = device;  
 }  
  
 public void performOperations() {  
 device.start();  
 device.print();  
 device.scan();  
 }  
}

Here:

* GenericDevice is a generic class that takes a type T that must extend Device and implement Printer and Scanner.
* Inside the class, we can use methods from all the types, ensuring that T satisfies the intersection.

**Best Practices for Using Intersection Types**

1. Use Intersection Types for Flexibility: Intersection types are useful when you want to work with multiple types without creating complex hierarchies. Use them to combine interfaces when designing flexible and reusable APIs.
2. Avoid Excessive Intersection Types: Don’t overuse intersection types in cases where simpler inheritance would suffice. Overuse can make the code harder to read and maintain.
3. Order Matters: Always place the class type first when using an intersection with classes and interfaces, as Java will not allow multiple classes in the intersection.
4. Be Mindful of Ambiguities: If two interfaces define methods with the same name, intersection types may introduce ambiguities that need to be resolved carefully.

**Limitations and Pitfalls**

1. No Multiple Class Bounds: Java doesn’t allow you to specify multiple classes in an intersection type. You can only have one class, followed by interfaces.
2. Type Erasure Issues: With intersection types in generics, the type erasure mechanism in Java may lead to the loss of type information at runtime, so be cautious when using reflection or working with legacy code.
3. Complexity in Compilation: Intersection types, especially when involving generic methods and classes, may result in complex and harder-to-debug compiler errors.

**Conclusion**

Intersection types in Java Generics provide a powerful way to express constraints where a type must satisfy multiple requirements, combining the features of different interfaces (and a class). While they offer flexibility, they should be used with care to avoid unnecessary complexity and maintain readability in your code.

**Best Practices in Using Generics**

**1. Use Generics for Type Safety and Reusability**

Generics help enforce type safety at compile time, preventing ClassCastException at runtime. This is one of the core benefits of using generics.

Example (Without Generics):

List list = new ArrayList();  
list.add("Hello");  
String s = (String) list.get(0); // Cast needed, can lead to runtime error

Example (With Generics):

List<String> list = new ArrayList<>();  
list.add("Hello");  
String s = list.get(0); // No cast needed, type safety enforced

By using generics, we avoid the need for casting and improve the robustness of our code.

**2. Prefer Generic Methods Over Raw Methods**

When writing methods that can operate on various types, prefer to define them as generic methods rather than using raw types or non-generic methods.

public <T> void printArray(T[] array) {  
 for (T element : array) {  
 System.out.println(element);  
 }  
}

This method can handle arrays of any type, ensuring type safety without restricting the input type unnecessarily.

**3. Avoid Raw Types**

Raw types (using generics without specifying the type parameter) should be avoided because they bypass the type-checking mechanism that generics provide.

**Bad Practice (Raw Types)**:

List list = new ArrayList();  
list.add(1); // Allowed, but type safety is lost  
list.add("Hello"); // Both are allowed; dangerous

**Good Practice**:

List<String> list = new ArrayList<>();  
list.add("Hello"); // Now, only Strings are allowed

Using raw types defeats the purpose of generics and can introduce errors that are hard to detect.

**4. Use Wildcards to Increase Flexibility**

Wildcards (?, ? extends, ? super) allow you to write more flexible code when you don't need to specify a strict type but still want to leverage type safety.

**Example (Upper-Bounded Wildcard)**:

public void printNumbers(List<? extends Number> list) {  
 for (Number n : list) {  
 System.out.println(n);  
 }  
}

Here, List<? extends Number> means the method can accept lists of Integer, Double, or any other subclass of Number, making the method more flexible.

**5. Use Bounded Type Parameters When Necessary**

Using bounds (extends or super) on generic types ensures that your methods or classes work with a range of types, while still enforcing constraints.

**Example (Bounded Type Parameter)**:

public <T extends Number> void add(T number1, T number2) {  
 System.out.println(number1.doubleValue() + number2.doubleValue());  
}

This method only accepts subclasses of Number (e.g., Integer, Double), so you don't have to worry about invalid types being passed in.

**6. Use Generics for Better API Design**

Generics enable you to design APIs that are easier to use and less error-prone by making the types explicit.

**Bad API Design (Non-Generic)**:

public List getItems() {  
 // Returns a raw list, no type information  
}

**Good API Design (Generic)**:

public List<String> getItems() {  
 // Returns a list of strings, ensuring type safety  
}

Explicit generic return types make your API more predictable and safer for other developers to use.

**7. Prefer**<? extends T>**for Producer and**<? super T>**for Consumer**

This is a common best practice summarized by the PECS mnemonic: Producer Extends, Consumer Super.

**Producer (<? extends T>)**: Use this when your method provides data from the collection.

public void copy(List<? extends Number> source, List<? super Number> destination) {  
 for (Number n : source) {  
 destination.add(n);  
 }  
}

The source list can be of any subtype of Number, while the destination list can accept any supertype of Number.

**Consumer (<? super T>)**: Use this when your method accepts data into the collection.

public void addNumbers(List<? super Integer> list) {  
 list.add(42); // Accepts Integer or any superclass (e.g., Number, Object)  
}

This strategy improves flexibility while maintaining type safety.

**8. Use Type Parameters to Simplify Complex Type Relationships**

When dealing with complex class hierarchies or APIs, using multiple type parameters can make your code more generic and reusable.

public class Pair<K, V> {  
 private K key;  
 private V value;  
   
 public Pair(K key, V value) {  
 this.key = key;  
 this.value = value;  
 }  
   
 public K getKey() {  
 return key;  
 }  
   
 public V getValue() {  
 return value;  
 }  
}

By defining a class with two type parameters (K, V), you can create pairs of any two types without losing type safety.

**9. Avoid Overcomplicating with Too Many Wildcards and Bounds**

While generics and wildcards offer flexibility, overusing them can make your code hard to read and understand.

**Bad Practice (Too Complex)**:

public void process(List<? extends Comparable<? super T>> list) { ... }

**Good Practice**:

Simplify where possible. Use wildcards and bounds only when necessary, and document your code to clarify the relationships between types.

**10. Understand the Limitations of Generics**

Generics have limitations, such as type erasure and restrictions on certain types (like primitives). Be mindful of these when designing your code.

**No Generics with Primitives**: Generics work only with reference types, so you cannot use primitives like int, char, or double directly with generics.

List<int> list = new ArrayList<>(); // Not allowed

**Solution**: Use the corresponding wrapper types (Integer, Character, Double, etc.).

List<Integer> list = new ArrayList<>();

Understanding these limitations will prevent unexpected issues and help you work around them effectively.

**11. Use @SuppressWarnings with Caution**

Sometimes you may need to suppress warnings for unchecked operations when dealing with legacy code or unavoidable raw types. Use the @SuppressWarnings("unchecked") annotation cautiously and document why it's necessary.

@SuppressWarnings("unchecked")  
public void processRawTypes(List list) {  
 // Some unavoidable raw type operations  
}

Apply this annotation only in small, isolated parts of your code, and avoid using it globally.

By following these best practices, you can use Java Generics more effectively, ensuring your code is both type-safe and flexible without being overly complex. Generics, when used properly, make your codebase more robust and easier to maintain over time.