Generics

Generics are a feature in Java that allows types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. Much like the more familiar formal parameters used in method declarations, type parameters allow you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Generics provide compile-time type safety, allowing programmers to catch invalid types simultaneously. Without generics, we can store any type of object in the collection, i.e., non-generic. Now generics force the Java programmer to store a specific type of object.

Java Generic methods and classes are distinguished by angle brackets `<>`. Inside these angle brackets, we specify the type. It's a way to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well.

The introduction of Generics has significantly broadened the expressiveness of the type system, and consequently, it has increased the ability of the Java compiler to detect errors at compile time rather than at runtime. This leads to safer, more robust code.

Non Generic PrintArray Class

If we want a class to print an array of several types. Then we can write,

```
public class noGenericPrintArray {

    public noGenericPrintArray() {
        System.out.println("noGenericPrintArray object created");
    }

    public void printArray(Integer [] intArray) {
        for (Integer element : intArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }

    public void printArray(Double [] doubleArray) {
        for (Double element : doubleArray) {
            System.out.printf("%s ", element);
```

```
        }
        System.out.println();
    }

    public void printArray(Character [] charArray) {
        for (Character element : charArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }
}
```

This Java code defines a class named `noGenericPrintArray` that has methods to print arrays of different types: `Integer`, `Double`, and `Character`.

When an object of this class is created, it prints “noGenericPrintArray object created” to the console.

The `printArray` methods take an array of a specific type as an argument and print each element of the array to the console. There are three versions of the `printArray` method, each for a different type of array: `Integer`, `Double`, and `Character`.

This is an example of method overloading, where multiple methods have the same name but different parameter lists. However, this approach does not use generics, so a separate method is needed for each type of array that you want to print. If you wanted to print an array of another type, you would need to write another method.

Generic Way to do this

```
public class GenericPrintArray {
    public GenericPrintArray() {
        System.out.println("GenericPrintArray object created");
    }

    public <T> void printArray(T [] array) {
        for (T element : array) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }
}
```

Class Declaration

```
public class GenericPrintArray { ... }
```

This line is declaring a public class named `GenericPrintArray`. The `public` keyword means that this class is accessible from any other class in the application.

Constructor

```
public GenericPrintArray() {
    System.out.println("GenericPrintArray object created");
}
```

This is a constructor for the `GenericPrintArray` class. A constructor is a special method that is called when an object of the class is created. This constructor takes no arguments and prints out “GenericPrintArray object created” when it is called.

Generic Method

```
public <T> void printArray(T [] array) {
    for (T element : array) {
        System.out.printf("%s ", element);
    }
    System.out.println();
}
```

This is a generic method named `printArray`.

- `public` means that this method can be accessed from any other class.
- `<T>` is the type parameter. This represents any type, and you can think of it as a placeholder for the actual type that will be used when the method is called.
- `void` means that this method does not return a value.
- `printArray(T [] array)` is the method signature. It means that this method takes one argument, which is an array of an arbitrary type `T`.
- The body of the method is a for-each loop that goes through each `element` in the `array` and prints it out. After all elements have been printed, it prints a new line.

This method demonstrates the power of generics. With generics, you can write a single method that can be used with many different types, rather than having to write a separate method for each type. In this case, you can use `printArray` to print an array of any type. The actual type to use in place of `T` will be determined at compile time-based on the type of the array you pass

in when you call the method. This provides type safety and can make your code more readable and reusable.

Supreemness of `Object` Class

In Java, every class implicitly inherits from the `Object` class if it does not extend any other class. This means that the `Object` class is at the top of the inheritance hierarchy for classes in Java.

How?

When you create a class in Java and don't specify a parent class using the `extends` keyword, the Java compiler automatically makes your class a child of the `Object` class. For example, if you write `public class MyClass { ... }`, the Java compiler treats it as `public class MyClass extends Object { ... }`.

Why?

There are several reasons why every class in Java inherits from the `Object` class:

1. **Common Methods**: The `Object` class provides some common methods like `equals()`, `hashCode()`, and `toString()` that can be used by all classes in Java. By inheriting from `Object`, every class gets these methods for free.
2. **Default Behavior**: The methods in the `Object` class provide default behavior that can be overridden by any class. For example, the `toString()` method returns a string representation of the object, but any class can override this method to provide a more meaningful string representation.
3. **Polymorphism**: Because all classes inherit from `Object`, any object can be treated as an `Object` type. This is useful for certain kinds of polymorphism, where you want to be able to write methods that work with objects of any type.

In summary, the `Object` class serves as a universal parent, providing common behavior and a form of polymorphism to all classes in Java. This design choice simplifies the language and makes it more flexible and powerful.

Try to print with `Object` class

```
public class ObjectPrint {
    public ObjectPrint() {
        System.out.println("ObjectPrint object created");
    }

    public void printArray(Object [] array) {
        for (Object element : array) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }
}
```

The `ObjectPrint` class in this Java code is designed to print arrays of any object type.

When an instance of `ObjectPrint` is created, it outputs “ObjectPrint object created”.

The `printArray` method accepts an array of Objects. It iterates over each element in the array and prints it. Since all classes in Java inherit from the `Object` class, this method can be used to print arrays of any type. However, unlike the generic method, this method does not provide compile-time type checking. If you try to perform an operation on an element that is not appropriate for its actual type, you will get a runtime error.

Problems of using `Object` Class Reference

Using the `Object` class as a type for collections or arrays, as in the `ObjectPrint` class, can lead to several issues:

1. **Lack of Type Safety**: When you use an `Object` array, you can put any type of object into the array. This can lead to problems if you later retrieve an element from the array and try to use it as a different type. For example, if you put a `String` into the array and later retrieve it and try to use it as an `Integer`, you will get a `ClassCastException` at runtime.
2. **Need for Type Casting**: When you retrieve an element from an `Object` array, you don’t know what its actual type is. You have to cast it to the type you expect. This can be error-prone, as it’s easy to make a mistake and cast it to the wrong type, leading to a `ClassCastException`.
3. **Loss of Compile-Time Checking**: One of the main advantages of generics is that they provide strong type checking at compile time. If you use an `Object` array, you lose this

benefit. Any type of error will only be caught at runtime, which can make debugging more difficult.

For example, consider the following code:

```
Object[] array = new Object[10];
array[0] = "Hello";
Integer num = (Integer) array[0]; // This will throw a ClassCastException at runtime
```

In this code, we create an `Object` array and store a `String` in it. Later, we try to retrieve the `String` and cast it to an `Integer`. This will compile fine, but it will throw a `ClassCastException` at runtime. If we had used a generic collection instead, this error would have been caught at compile time.

In conclusion, while using `Object` as a generic type can provide some flexibility, it also introduces potential errors and complexities that can be avoided by using Java's generics feature. Generics provide stronger type checks at compile time, eliminate the need for type casting, and can make your code easier to read and maintain.

Generic Compare Method

```
public static <T extends Comparable<T> > T max(T a, T b, T c){

    T max = a; // assume a is initially the largest

    if(b.compareTo(max) > 0){
        max = b;
    }
    if(c.compareTo(max) > 0){
        max = c;
    }
    return max;
}
```

This is a generic method in Java that finds the maximum of three objects. Here's a breakdown:

- `public static <T extends Comparable<T> > T max(T a, T b, T c)`: This is the method signature. It's a public and static method named `max`. The `<T extends Comparable<T>>` part is a type parameter that says this method will work with any type

`T` that implements the `Comparable` interface. The method takes three parameters of type `T` and returns a result of type `T`.
- `T max = a;`: This line initializes `max` as `a`, assuming `a` is the largest.
- `if(b.compareTo(max) > 0) { max = b; }`: This line compares `b` with the current `max`. If `b` is greater, it becomes the new `max`.
- `if(c.compareTo(max) > 0) { max = c; }`: This line does the same comparison for `c`.
- `return max;`: This line returns the `max` value.

The `compareTo` method is part of the `Comparable` interface and is used to compare the current object with another object. If the current object is less than, equal to, or greater than the other object, it returns a negative integer, zero, or a positive integer, respectively.

In summary, this method compares three objects of a type that implements the `Comparable` interface and returns the maximum. It's a good example of how generics can provide flexibility and type safety in code.

Generic Type Safety

In Java, **bounded type parameters** are used to restrict the types that can be used as arguments in a parameterized type. For example,

```
public static <T extends Comparable<T>> T max(T a, T b, T c)
```

The `<T extends Comparable<T>>` part is a bounded type parameter. Here's what it means:

- `T` is the type parameter. This represents any type.
- `extends` is a keyword used to specify the upper bound. Despite the keyword "extends", this can be used to specify that `T` must be a class that implements a particular interface, not just extend a particular class.
- `Comparable<T>` is the upper bound. This means that `T` can be any type that implements the `Comparable` interface.

So, `<T extends Comparable<T>>` means that `T` can be any type that is `Comparable` with itself. This is necessary for the `max` method because it needs to be able to call the `compareTo` method on objects of type `T`.

Here's an example,

```
Integer a = 1;
Integer b = 2;
Integer c = 3;
Integer max = max(a, b, c);  // This will return 3
```

In this example, `T` is `Integer`, which is a class that implements `Comparable<Integer>`. So, it's valid to use `Integer` as the type argument for the `max` method.

Bounded type parameters are a powerful feature that increases the flexibility and type safety of your generic classes and methods. They allow you to write code that works with a range of types while still being able to use methods of a certain interface or class. This leads to code that is both more reusable and safer.

Generic OverLoading

In Java, method overloading occurs when two or more methods in the same class have the same name but different parameters. However, when it comes to generics, the concept of "generic overloading" can be a bit misleading.

Java's type erasure means that generic types are not available at runtime. This means that you cannot have two methods that only differ by generic type:

```
public class Example {
    public <T extends Number> void method(T t) { ... }
    public <T extends String> void method(T t) { ... }
    // This would cause a compile error
}
```

In the above example, the Java compiler would throw an error because after type erasure, both methods have the same parameter type - they both take an `Object`.

However, you can overload methods that have different number or types of parameters, even in a generic class:

```
public class Example<T> {
    public void method(T t) { ... }
    public void method(T t1, T t2) { ... } // This is valid overloading
}
```

In this case, `method(T t)` and `method(T t1, T t2)` are considered two different methods, so this is valid overloading.

So, while you can overload methods in a generic class, the overloading is based on the number and types of parameters, not the generic type itself. This is due to type erasure in Java's generics implementation.

Generic Stack Example

```
public class Stack <T> {
    private final int size;
    private int top;
    private T[] elements;

    public Stack(){
        this(10);
    }

    @SuppressWarnings("unchecked")
    public Stack(int s){
        size = s > 0 ? s : 10;
        top = -1;
        elements = (T[]) new Object[size];
    }

    public void push(T element){
        if(top == size - 1){
            throw new FullStackException(String.format("Stack is full, cannot push"
                                                       + "%s", element));
        }
        elements[++top] = element;
    }

    public T pop(){
        if(top == -1){
            throw new EmptyStackException("Stack is empty, cannot pop");
        }
        return elements[top--];
    }
}
```

```
public class FullStackException extends RuntimeException {
    public FullStackException() {
        this("Stack is full");
    }
```

```
    public FullStackException(String message) {
        super(message);
    }
}
```

```
public class EmptyStackException extends RuntimeException {
    public EmptyStackException() {
        this("Stack is empty");
    }

    public EmptyStackException(String message) {
        super(message);
    }
}
```

Generics in Java is a powerful feature that allows you to define a common set of APIs for various types of data. In the provided code, a generic `Stack` class is defined. The `<T>` notation in the class declaration is a type parameter that stands for the type of object the stack will hold.

The Generic Stack Class

The `Stack` class is a generic data structure that can hold elements of any type `T`. It has a fixed size, a `top` index to track the top of the stack, and an array `elements` to store the elements.

```
public class Stack <T> {
    private final int size;
    private int top;
    private T[] elements;
```

Constructors

The `Stack` class has two constructors: a no-argument constructor and a constructor that takes an integer argument for the size of the stack.

```
public Stack(){
    this(10);
}

@SuppressWarnings("unchecked")
```

```
public Stack(int s){
    size = s > 0 ? s : 10;
    top = -1;
    elements = (T[]) new Object[size];
}
```

The no-argument constructor calls the other constructor with a default size of 10. The second constructor initializes the `size`, `top`, and `elements` of the stack.

The Push Method

The `push` method adds an element to the top of the stack. If the stack is full, it throws a `FullStackException`.

```
public void push(T element){
    if(top == size - 1){
        throw new FullStackException(String.format("Stack is full, cannot push %s", element));
    }
    elements[++top] = element;
}
```

The Pop Method

The `pop` method removes and returns the top element of the stack. If the stack is empty, it throws an `EmptyStackException`.

```
public T pop(){
    if(top == -1){
        throw new EmptyStackException("Stack is empty, cannot pop");
    }
    return elements[top--];
}
```

Custom Exceptions

The `EmptyStackException` and `FullStackException` classes are custom exceptions used by the `Stack` class. They extend `RuntimeException`, meaning they are unchecked exceptions.

```
public class EmptyStackException extends RuntimeException {
    public EmptyStackException() {
        this("Stack is empty");
    }

    public EmptyStackException(String message) {
        super(message);
    }
}

public class FullStackException extends RuntimeException {
    public FullStackException() {
        this("Stack is full");
    }

    public FullStackException(String message) {
        super(message);
    }
}
```

Each exception class has two constructors: a no-argument constructor and a constructor that takes a string message. The no-argument constructor calls the other constructor with a default message.

In summary, the `Stack` class is a great example of how generics can be used in Java to create flexible and reusable data structures. By using a type parameter `T`, the `Stack` class can be used with any object type, providing a type-safe way to work with stacks of different kinds of data.

Raw Type

In the context of Java generics, a raw type is the name of a generic class or interface without any type arguments. For example, if you have a generic class like `List<E>`, `List` would be the raw type.

Using raw types in your code can lead to warnings at compile time because they bypass the generics type checks that ensure type safety. Here's an example:

```
List rawList = new ArrayList(); // raw type
List<String> list = new ArrayList<String>(); // parameterized type
```

In the above code, `rawList` is a raw type, while `list` is a parameterized type. You can add any type of object to `rawList` without getting a compile-time error. However, if you try to add anything other than a `String` to the `list`, you will get a compile-time error.

While raw types can sometimes be useful for backward compatibility with older Java code that does not use generics, their use is generally discouraged because they negate the benefits of generics, such as type safety and clarity. It's usually better to use parameterized types and take advantage of the robustness and flexibility provided by generics.

Here is how we can use RawType in our previously built `Stack` class 😀

```
package Generics.RawType;
import Generics.stack.*;
public class Main {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        // Create a raw type stack
        Stack rawTypeStack = new Stack(5);

        // Create a generic type stack
        Stack<Integer> integerStack = new Stack<>(5);

        // Push elements to the raw type stack
        rawTypeStack.push("String");
        rawTypeStack.push(13.8);
        rawTypeStack.push(5);
        rawTypeStack.push('q');

        // Push elements to the generic type stack
        integerStack.push(1);
        integerStack.push(2);
        integerStack.push(3);
        integerStack.push(4);
        integerStack.push(5);

        // Pop elements from the raw type stack
        System.out.println("Pop from raw type stack: ");
        while (!rawTypeStack.isEmpty()){
            System.out.printf("%s ", rawTypeStack.pop());
        }
        System.out.println();

        // Pop elements from the generic type stack
        System.out.println("Pop from generic type stack: ");
        while (!integerStack.isEmpty()){
```

```
            System.out.printf("%d ", integerStack.pop());
        }
        System.out.println();
    }
}
```

This Java code demonstrates the use of raw types and generics in the context of a `Stack` class.

Two `Stack` objects are created: `rawTypeStack` and `integerStack`. `rawTypeStack` is a raw type, meaning it can hold elements of any type. `integerStack`, on the other hand, is a generic type specifically for `Integer` elements.

Elements of various types (`String`, `Double`, `Integer`, and `Character`) are pushed onto `rawTypeStack`, demonstrating the flexibility of raw types. However, this comes with the risk of type errors, as there's no compile-time check to ensure that all elements are of the same type.

Only `Integer` elements are pushed onto `integerStack`. This is type-safe, meaning you can't accidentally push an element of the wrong type onto the stack. If you tried to push a non-integer onto `integerStack`, you'd get a compile-time error.

The code then pops all elements from both stacks. For `rawTypeStack`, the type of the popped elements is `Object`, so they could be any type. For `integerStack`, the popped elements are guaranteed to be `Integer`.

This code illustrates the trade-off between flexibility and type safety in the use of raw types versus generics in Java. While raw types allow for greater flexibility, generics provide compile-time type checking, reducing the likelihood of runtime type errors.

WildCards

In Java, the wildcard character (`?`) is used in generics to represent an unknown type. This can be useful when you're writing a method and you want it to be flexible with regard to the types it can accept, but you still want to maintain type safety.

There are three types of wildcards in Java:

1. Unbounded Wildcards (`?`)

These can represent any type. They are useful when you are writing a method that works with lists of any type. For example, a method that prints out a list could use an unbounded wildcard:

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

2. Upper Bounded Wildcards (`? extends T`)

These can represent any type that is a subtype of `T` (or `T` itself). They are useful when you need a method to accept types that are a certain class or any of its subclasses. For example, a method that works with lists of numbers or their subclasses (like `Integer`, `Double`, etc.) could use an upper bounded wildcard:

```
public void processNumbers(List<? extends Number> list) {
    // ...
}
```

```
public static double sum(ArrayList< ? extends Number > list){
    double sum = 0;
    for (Number number : list) {
        sum += number.doubleValue();
    }
    return sum;
}
```

This Java method demonstrates the use of an upper-bounded wildcard in generics. Here's what it does:

- `public static double sum(ArrayList< ? extends Number > list)`: This is the method signature. It's a public and static method named `sum`. The `ArrayList< ? extends Number >` part is an upper bounded wildcard. It means this method accepts an `ArrayList` of any type that is a subclass of `Number` (or `Number` itself).
- `double sum = 0;`: This line initializes a `double` variable named `sum` to 0. This variable will be used to calculate the sum of all the elements in the `ArrayList`.
- `for (Number number : list) { ... }`: This is a for-each loop that iterates over each element in the `ArrayList`. Each element is treated as a `Number`, which is possible because the `ArrayList` is guaranteed to contain only `Number` or its subclasses due to the upper bounded wildcard.

- `sum += number.doubleValue();`: This line adds the `double` value of the current element to `sum`. The `doubleValue` method is a method of `Number` that returns the `double` value of the object.
- `return sum;`: This line returns the sum of all the elements in the `ArrayList`.

this method demonstrates how an upper bounded wildcard can be used to accept a collection of a certain type or its subclasses, while still being able to use methods of the upper bound class (`Number` in this case). This provides flexibility and type safety, as you can pass an `ArrayList` of any numeric type (like `Integer`, `Double`, etc.) to this method, and it will correctly calculate the sum.

3. Lower Bounded Wildcards (`? super T`)

These can represent any type that is a supertype of `T` (or `T` itself). They are useful when you need a method to accept types that are a certain class or any of its superclasses. For example, a method that works with lists of numbers or any of their superclasses could use a lower bounded wildcard:

```
public void addNumbers(List<? super Integer> list) {
    // ...
}
```

```
public static double sum(ArrayList< ? super Integer > list){
    double sum = 0;
    for (Object number : list) {
        sum += ((Integer)number).doubleValue();
    }
    return sum;
}
```

This Java method demonstrates the use of a lower bounded wildcard in generics. Here's what it does:

- `public static double sum(ArrayList< ? super Integer > list)`: This is the method signature. It's a public and static method named `sum`. The `ArrayList< ? super Integer >` part is a lower bounded wildcard. It means this method accepts an `ArrayList` of any type that is a superclass of `Integer` (or `Integer` itself).
- `double sum = 0;`: This line initializes a `double` variable named `sum` to 0. This variable will be used to calculate the sum of all the elements in the `ArrayList`.

- `for (Object number : list) { ... }`: This is a for-each loop that iterates over each element in the `ArrayList`. Each element is treated as an `Object`, which is the superclass of all classes in Java.
- `sum += ((Integer)number).doubleValue();`: This line adds the `double` value of the current element to `sum`. Since the elements are treated as `Object`, they need to be cast to `Integer` before the `doubleValue` method can be called.
- `return sum;`: This line returns the sum of all the elements in the `ArrayList`.

In terms of what types can be passed to the `sum` method, because `Integer` class extends `Number` and `Object` classes in Java, only an `ArrayList` of `Integer`, `Number`, or `Object` can be passed to this method. Any other type would cause a compile-time error.

this method demonstrates how a lower bounded wildcard can be used to accept a collection of a certain type or its superclasses. However, because the exact type is not known, elements are treated as `Object` and must be cast to the appropriate type before methods specific to that type can be called. This provides flexibility but can also lead to `ClassCastException` if the wrong type is provided.

wildcards in Java generics provide additional flexibility for parameterized types, allowing you to create methods that are type-safe yet flexible in terms of the types they can accept.