Generics

-> It's used to enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods.

-> This provides a way to ensure type safety and reduces the need for type casting

Type Safety: Generics allow for compile-time type checking, reducing runtime errors.

Elimination of Casts: Explicit casting is no longer needed, as the type is known at compile time.

Code Reusability: Generic methods and classes can be reused with different data types, leading to more flexible and reusable code.

Key Concepts of Generics:

1.Type Parameters:

-> Generics introduce the concept of type parameters.

-> It is a placeholder for a type that is specified when an instance of a generic type is created.

Example in Class:

```
public class Box<T> {
  private T t;
  public void set(T t) {
    this.t = t;
  }
  public T get() {
    return t;
  }
}
```

-> Here, T is a type parameter that can be replaced with any type (like Integer, String, etc.).

Example:

```
public class GenericClassType {
        static class Box<T> {
                // T stands for "Type"
                private T t;
                public void set(T t) {
                        this.t = t;
                }
                public T get() {
                        return t;
                }
        }
```

```
    static class Person {
        private String name;
        private int age;
        public Person(String name, int age) {
            this.name = name;
            this.age = age;
        }
        @Override
        public String toString() {
            return name + " (" + age + ")";
        }
    }

    public static void main(String[] args) {
        // Example with Integer
        Box<Integer> integerBox = new Box<>();
        integerBox.set(123);
        Integer intValue = integerBox.get();
        System.out.println("Integer Value: " + intValue);

        // Example with String
        Box<String> stringBox = new Box<>();
        stringBox.set("Abdulmajeeth");
        String strValue = stringBox.get();
        System.out.println("String Value: " + strValue);

        // Example with Custom Class Person
        Box<Person> personBox = new Box<>();
        Person person = new Person("Abdulmajeeth", 29);
        personBox.set(person);
        Person personValue = personBox.get();
        System.out.println("Person Value: " + personValue);
    }
}
```

Output:

```
Integer Value: 123
String Value: Abdulmajeeth
Person Value: Abdulmajeeth (29)
```

2.Generic Methods:

-> Methods can also be defined with type parameters, allowing for type-safe operations without knowing the specific type at compile time.

Syntax:

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

Example:

```
public class GenericMethodExample {
  // Generic method to find the maximum element in an array
  public static <T> T findMax(T[] array) {
    if (array == null || array.length == 0) {
      return null;
    }

    T max = array[0];
    for (int i = 1; i < array.length; i++) {
      if (array[i].compareTo(max) > 0) {
        max = array[i];
      }
    }
    return max;
  }

  public static void main(String[] args) {
    // Example with Integer array
    Integer[] intArray = {3, 7, 1, 9, 4};
    System.out.println("Maximum integer: " + findMax(intArray));
```

```
        // Example with Double array

        Double[] doubleArray = {3.5, 7.2, 1.8, 9.6, 4.3};

        System.out.println("Maximum double: " + findMax(doubleArray));


        // Example with String array

        String[] stringArray = {"apple", "orange", "banana", "pineapple"};

        System.out.println("Maximum string: " + findMax(stringArray));

    }

}
```

Output:

```
Max Integer: 5
Max String: pear
Max Person: Charlie (35)
```

3.Bounded Type Parameters:

-> You can restrict the types that can be used as type arguments by specifying bounds.

Syntax:

```
public <T extends Number> void printNumber(T number) {

    System.out.println(number);

}
```

-> T can be any type that is a subclass of Number.

Example:

```
public class GenericMethodExample {

    // Generic method to find the maximum element in an array of numeric elements

    public static <T extends Number> T findMax(T[] array) {

        if (array == null || array.length == 0) {

            return null;

        }

        T max = array[0];

        for (int i = 1; i < array.length; i++) {

            if (array[i].doubleValue() > max.doubleValue()) {

                max = array[i];

            }

        }
        return max;
    }

    public static void main(String[] args) {
        // Example with Integer array
        Integer[] intArray = {3, 7, 1, 9, 4};
        System.out.println("Maximum integer: " + findMax(intArray));

        // Example with Double array
        Double[] doubleArray = {3.5, 7.2, 1.8, 9.6, 4.3};
        System.out.println("Maximum double: " + findMax(doubleArray));

        // Example with Float array
        Float[] floatArray = {3.5f, 7.2f, 1.8f, 9.6f, 4.3f};
        System.out.println("Maximum float: " + findMax(floatArray));

        // Example with Long array
        Long[] longArray = {300L, 700L, 100L, 900L, 400L};
        System.out.println("Maximum long: " + findMax(longArray));
    }
}
```

Output:

```
Maximum integer: 9
Maximum double: 9.6
Maximum float: 9.6
Maximum long: 900
```

4.Multiple Bounds:

-> A type parameter can have multiple bounds.

Syntax:

```
public <T extends Number & Comparable<T>> void compareNumbers(T num1, T num2) {
  if (num1.compareTo(num2) > 0) {
    System.out.println(num1 + " is greater than " + num2);
  } else {
    System.out.println(num1 + " is less than or equal to " + num2);
  }
}
```

Example:

```
public class GenericMethodExample {
  // Generic method to find the maximum element in an array of numeric elements
  public static <T extends Number & Comparable<T>, U extends Number> T findMax(T[] array1, U[]
array2) {
    if (array1 == null || array1.length == 0 || array2 == null || array2.length == 0) {
      return null;
    }

    T max = array1[0];
    for (T element : array1) {
      if (element.compareTo(max) > 0) {
        max = element;
      }
    }

    for (U element : array2) {
      double doubleValue = element.doubleValue();
      if (doubleValue > max.doubleValue()) {
        max = (T) element;
      }
    }

    return max;
  }

  public static void main(String[] args) {
    // Example with Integer array and Double array
    Integer[] intArray = {3, 7, 1, 9, 4};
    Double[] doubleArray = {3.5, 7.2, 1.8, 9.6, 4.3};
    System.out.println("Maximum integer: " + findMax(intArray, doubleArray));

    // Example with Float array and Long array
```

```
        Float[] floatArray = {3.5f, 7.2f, 1.8f, 9.6f, 4.3f};

        Long[] longArray = {300L, 700L, 100L, 900L, 400L};

        System.out.println("Maximum float: " + findMax(floatArray, longArray));

    }

}
```

Output:

```
Maximum integer: 9.6
Maximum float: 900
```

5.Wildcards:

-> Wildcards are used to denote unknown types.

-> They are represented by the ? symbol and can be used with extends and super bounds.

-> Wildcards in Java generics provide a way to specify a range of allowable types. They are particularly useful when dealing with generic types and can increase the flexibility and reusability of your code.

-> Wildcards can be categorized into three main types: unbounded wildcards, bounded wildcards, and lower-bounded wildcards.

Syntax:

```
public void printList(List<?> list) {

    for (Object element : list) {

        System.out.println(element);

    }

}
```

Why Use Wildcards?

Wildcards are useful for increasing the flexibility of your API by allowing it to operate on a range of types. They are particularly useful in the following scenarios:

-> Reading Data (Covariance with <? extends Type>): When you need to read data from a structure, but you do not need to know the exact type of the data being read, upper bounded wildcards are appropriate.

-> Writing Data (Contravariance with <? super Type>): When you need to write data to a structure, but you do not need to know the exact type of the data being written, lower bounded wildcards are appropriate.

-> Unknown Type (<?>): When you are using a generic type but do not care about the specific type, unbounded wildcards are appropriate.

Unbounded Wildcards (<?>):

An unbounded wildcard represents an unknown type. It is useful when you want to work with generic types but do not care about the specific type parameter.

Upper-bounded wildcard (<? extends Type>):

-> An upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type.

-> It is useful when you want to read items from a generic collection and you know that the items are of a specific type or its subtype.

-> List<? extends Number> allows any list of a type that is a subclass of Number.

Lower Bounded Wildcards (<? super Type>):

-> A lower bounded wildcard restricts the unknown type to be a specific type or a supertype of that type.

-> It is useful when you want to write items to a generic collection and you know that the items are of a specific type or its supertype.

-> List<? super Integer> allows any list of a type that is a superclass of Integer.

Example:

```
import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;


public class WildcardExample {


  // Generic method using an unbounded wildcard

  public static void printElements(List<?> list) {

    for (Object element : list) {

      System.out.println(element);

    }

  }


  // Generic method using an upper bounded wildcard

  public static double calculateSum(List<? extends Number> list) {

    double sum = 0.0;

    for (Number number : list) {

      sum += number.doubleValue();

    }

    return sum;
    }

    // Generic method using a lower bounded wildcard
    public static void addIntegers(List<? super Integer> list) {
        for (int i = 1; i <= 10; i++) {
            list.add(i);
        }
    }

    public static void main(String[] args) {
        // Unbounded wildcard example
        List<String> stringList = Arrays.asList("one", "two", "three");
        printElements(stringList);

        // Upper bounded wildcard example
        List<Integer> intList = Arrays.asList(1, 2, 3);
        List<Double> doubleList = Arrays.asList(1.1, 2.2, 3.3);
        System.out.println("Sum of integers: " + calculateSum(intList));
        System.out.println("Sum of doubles: " + calculateSum(doubleList));

        // Lower bounded wildcard example
        List<Number> numberList = new ArrayList<>();
        addIntegers(numberList);
        System.out.println("Number list: " + numberList);
    }
}
```

Output:

```
one
two
three
Sum of integers: 6.0
Sum of doubles: 6.6
Number list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Explanation:

Unbounded wildcard example:

Prints each element of the stringList ("one", "two", "three").

Upper bounded wildcard example:

Calculates the sum of integers in the intList (1 + 2 + 3 = 6.0).

Calculates the sum of doubles in the doubleList (1.1 + 2.2 + 3.3 = 6.6).

Lower bounded wildcard example:

Adds integers from 1 to 10 into the numberList.

Prints the contents of the numberList, which contains integers from 1 to 10.

6.Generic Class:

-> A generic class is defined with one or more type parameters.

```
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;
  }
}
```

7. Type Inference:

-> The Java compiler can infer the type arguments from the context in which they are used.

Pair<Integer, String> pair = new **Pair<>**(1, "apple");

Example:

```
import java.util.ArrayList;
import java.util.List;

public class GenericsExample {
  // Generic method
  public static <T> void addToList(T element, List<T> list) {
    list.add(element);
  }

  // Bounded generic method
  public static <T extends Number> void printSum(List<T> list) {
    double sum = 0.0;
    for (T number : list) {
      sum += number.doubleValue();
    }
    System.out.println("Sum: " + sum);
  }

  public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();
    addToList(10, intList);
    addToList(20, intList);
    printSum(intList);

    List<Double> doubleList = new ArrayList<>();
    addToList(10.5, doubleList);
    addToList(20.5, doubleList);
```

```
        printSum(doubleList);


        // Using a generic class
        Pair<String, Integer> pair = new Pair<>("age", 30);
        System.out.println("Key: " + pair.getKey() + ", Value: " + pair.getValue());
    }
}

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;
    }
}
```

Output:

```
Sum: 30.0
Sum: 31.0
Key: age, Value: 30
```

8) Type Erasure:

-> It allows the compiler to translate generic types, which are used at compile time for type checking, into non-generic types, which are used at runtime.

-> This translation process ensures that generics do not introduce any overhead or compatibility issues with existing code that does not use generics.

How it Works:

-> Type Checking: During compilation, the compiler performs type checking on generic code to ensure type safety. This includes verifying that the correct types are used in method invocations, assignments, and other operations involving generic types.

-> Type Erasure: After type checking is complete, the compiler removes all generic type information from the code. It replaces generic type parameters with their erasure, which is typically the upper bound of the type parameter. For example, if you have a generic class List<T>, its erasure is List<Object>.

-> Raw Types: In addition to replacing generic type parameters with their erasure, the compiler generates raw types for generic classes and interfaces. Raw types are non-generic versions of generic types, where all generic type parameters are replaced with raw types. For example, if you have a generic class List<T>, its raw type is List.

-> Bridge Methods: To maintain binary compatibility with pre-existing code that does not use generics, the compiler may generate bridge methods during type erasure. Bridge methods are synthetic methods that ensure that generic code can be used with non-generic code without breaking existing binary compatibility.

Before:

```
public class Node<T> {
   private T data;
   private Node<T> next;
   public Node(T data, Node<T> next) {
      this.data = data;
      this.next = next;
   }
   public T getData() { return data; }
   // ...
}
```

After:

```
public class Node {
  private Object data;
  private Node next;
  public Node(Object data, Node next) {
    this.data = data;
    this.next = next;
  }
  public Object getData() { return data; }
  // ...
}
```

Before:

```
public class Node<T extends Comparable<T>> {
  private T data;
  private Node<T> next;
  public Node(T data, Node<T> next) {
    this.data = data;
    this.next = next;
  }
  public T getData() { return data; }
  // ...
}
```

After:

```
public class Node {
  private Comparable data;
  private Node next;
  public Node(Comparable data, Node next) {
    this.data = data;
    this.next = next;
  }
  public Comparable getData() { return data; }
  // ...
}
```