# Generics in Java - Detailed Notes

## Introduction

Generics were introduced in Java 5 to ensure type safety and promote code reusability. They allow creating classes, interfaces, and methods that can work with different data types while providing compile-time type checking.

Key Benefits:  
- Type Safety: Prevents ClassCastException.  
- Compile-Time Checking: Errors are caught early.  
- Code Reusability: Same code works for different types.  
- No Explicit Type Casting: Type conversions are handled automatically.

## Example Without Generics (Before Java 5)

In the earlier versions of Java, collections allowed adding any type of object, which was risky.

Java Example:  
import java.util.\*;

public class NonGenericExample {  
 public static void main(String[] args) {  
 List list = new ArrayList();  
 list.add("Hello");  
 list.add(10);

for (Object obj : list) {  
 String str = (String) obj;  
 System.out.println(str.length());  
 }  
 }  
}

Drawbacks:  
- Allows adding different types in the same list.  
- Requires explicit type casting.  
- No compile-time error increases the risk of runtime errors like ClassCastException.

## Example With Generics (Java 5+)

Java Example:  
import java.util.\*;

public class GenericExample {  
 public static void main(String[] args) {  
 List<String> list = new ArrayList<>();  
 list.add("Hello");

for (String str : list) {  
 System.out.println(str.length());  
 }  
 }  
}

Advantages:  
- Ensures type safety.  
- No need for explicit type casting.  
- Errors are caught during compilation.

## Advantages of Using Generics

- Type Safety: Catches incorrect data type usage at compile-time.  
- Code Reusability: Generic classes and methods can handle multiple data types.  
- Eliminates Type Casting: Reduces the need for manual type conversions.  
- Cleaner and More Reliable Code.

## Generic Classes

A generic class works with different data types without rewriting the class.

Syntax:  
class Box<T> {  
 private T value;

public void set(T value) { this.value = value; }  
 public T get() { return value; }  
}

Example:  
public class Main {  
 public static void main(String[] args) {  
 Box<Integer> intBox = new Box<>();  
 intBox.set(100);  
 System.out.println(intBox.get());

Box<String> strBox = new Box<>();  
 strBox.set("Generics Example");  
 System.out.println(strBox.get());  
 }  
}

## Generic Methods

A generic method works with different data types within a single method.

Syntax:  
public <T> void display(T value) {  
 System.out.println(value);  
}

Example:  
public class Demo {  
 public static <T> void display(T value) {  
 System.out.println(value);  
 }

public static void main(String[] args) {  
 display(10);  
 display("Hello World");  
 }  
}

## Bounded Type Parameters

Bounded type parameters restrict which types can be used as generic arguments.

Syntax:  
class NumericBox<T extends Number> {  
 private T value;

public void set(T value) { this.value = value; }  
 public T get() { return value; }  
}

Example:  
public class Main {  
 public static void main(String[] args) {  
 NumericBox<Integer> intBox = new NumericBox<>();  
 intBox.set(10);

NumericBox<Double> doubleBox = new NumericBox<>();  
 doubleBox.set(15.5);

// NumericBox<String> strBox = new NumericBox<>(); // Compile-time error  
 }  
}

## Wildcard Types in Generics

Wildcard types are used when the exact type is unknown.

Types of Wildcards:  
1. Upper Bounded Wildcard (? extends T):  
- Allows reading but restricts adding elements.  
- Used when working with types that are T or subclasses of T.

Example:  
public static void printList(List<? extends Number> list) {  
 for (Number num : list) {  
 System.out.println(num);  
 }  
}

2. Lower Bounded Wildcard (? super T):  
- Allows adding elements but restricts reading.  
- Used when working with types that are T or superclasses of T.

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

## Raw Types (Not Recommended)

A raw type is using a generic class without specifying its type parameter.

Example:  
List list = new ArrayList();  
list.add("Hello");  
list.add(100);

for (Object obj : list) {  
 String str = (String) obj; // Risky casting  
 System.out.println(str.length());  
}

Problems:  
- Mixing data types is possible.  
- Can cause ClassCastException at runtime.  
- Compiler gives unchecked warnings.

## Type Erasure

Java uses type erasure to remove all generic type information at compile time. This allows backward compatibility with older Java versions.

Key Points:  
- JVM does not know generic types at runtime.  
- Generics exist only at compile time.  
- After compilation, type parameters are replaced with their upper bound or Object.

Example:  
// Before Compilation  
List<String> list = new ArrayList<>();

// After Compilation  
List list = new ArrayList<>();

Impact:  
- Cannot create arrays of generic types.  
- Cannot use instanceof with generic types.  
- Cannot access the generic type at runtime.

## Summary

- Generics improve type safety, readability, and reusability.  
- Wildcards add flexibility to method parameters.  
- Raw types should be avoided.  
- Type erasure limits some operations but provides backward compatibility.