Generics in Java

This document explains the concept of **Generics** in Java, their **necessity**, **syntax**, and related features like **Bounded Types**.

---

1. Requirement of Generics

a. Improve Typedness in Collections

- Arrays in Java are **type-safe** — they only allow storing **homogeneous elements**.
- Attempting to store a different type causes a **compilation error**.

```
java
CopyEdit
Student[] stds = new Student[3];
stds[0] = new Student();
stds[1] = new Student();
stds[2] = new Customer(); // ---> Compilation Error
```

- Collections **before Generics** could store **heterogeneous elements**, reducing type safety.

```
java
CopyEdit
Collection c = new ArrayList();
c.add(new Student());   // ----> No Compilation Error
c.add(new Customer());  // ----> No Compilation Error
c.add(new Employee());  // ----> No Compilation Error
```

- **Generics were introduced** to ensure compile-time type safety for collections.

---

b. Avoid Typecasting

- In non-generic collections, `get()` returns `Object`. You need to **explicitly cast** it to the required type.

```
java
CopyEdit
Collection c = new ArrayList();
c.add(new Employee());
c.add(new Student());
c.add(new Customer());

Employee emp = (Employee) c.get(0);  // Correct
Student std = (Student) c.get(1);    // Correct
Customer cust = (Customer) c.get(2); // Correct
```

- **Generics remove the need for typecasting** during retrieval:

```
java
CopyEdit
Collection<Student> c = new ArrayList<Student>();
Student std1 = c.get(0);  // No casting required
```

---

2. What are Generics?

- A **generic class** is defined with a **type parameter**.
- Ensures that only specific types can be added to a collection.

```
java
CopyEdit
Collection<Student> c = new ArrayList<Student>();
c.add(new Student());    // Valid
// c.add(new Customer()); // Compilation Error
```

---

3. Conclusion on Basic Generics

✓ Improves **typedness** in collections
✓ **Avoids typecasting** while retrieving elements

---

4. Syntax for Generics

- **Syntax:**

  ```
  java
  CopyEdit
  CollectionClass<Type> refVar = new CollectionClass<Type>();
  ```

- `Type` must be:
  - A user-defined class (e.g., `Student`, `Employee`)
  - Or a **Wrapper class** for primitives (e.g., `Integer`, `Double`)
  - ✕ **Not** a primitive type like `int`, `char`, etc.

```
java
CopyEdit
// ArrayList<int> al = new ArrayList<int>(); // Invalid
ArrayList<Integer> al = new ArrayList<Integer>(); // Valid
```

Type Inference (Diamond Operator)

```
java
CopyEdit
ArrayList<Integer> al1 = new ArrayList<Integer>(); // Valid
ArrayList<Integer> al2 = new ArrayList<>();        // Valid (Type
Inference)
```

Compatibility Rule

- Types must **match exactly** on both sides.

```
java
CopyEdit
// ArrayList<Number> al = new ArrayList<Integer>(); // Invalid
// ArrayList<Integer> al = new ArrayList<Number>(); // Invalid
ArrayList<Integer> al = new ArrayList<Integer>();   // Valid
```

---

5. Comparison: Before and After Java 1.5 (Generics)

Feature	Before Java 1.5	After Java 1.5 (Generics)
Class Definition	`public class ArrayList {}`	`public class ArrayList<T> {}`
`add` Method	`public void add(Object obj)`	`public void add(T t)`
`get` Method	`public Object get(int index)`	`public T get(int index)`

Adding Elements	`ArrayList al = new ArrayList();`
	`al.add(10);`
	`al.add("ABC");`
		`ArrayList<Integer> al = new ArrayList<>();`
		`al.add(10);`
		`// al.add("abc"); // Error`

Retrieving Elements	`int val = (Integer) al.get(0);`
	`String str = (String) al.get(1);`
		`int val = al.get(0);`
		`String str = al.get(0);`

---

6. Generic Class

Declaration:

```
java
CopyEdit
public class ArrayList<T> {
    // ...
}

class Account<T> {
    T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }
    public void display() {
        System.out.println(t.getClass().getName());
    }
}
```

Usage:

```
java
CopyEdit
public class Main {
    public static void main(String[] args) {
        Account<String> account1 = new Account<>();
        account1.add("AAA");
        System.out.println(account1.get());
        account1.display();

        Account<Integer> account2 = new Account<>();
        account2.add(1);
        System.out.println(account2.get());
        account2.display();

        Account<Double> account3 = new Account<>();
        account3.add(1.0);
        System.out.println(account3.get());
        account3.display();
    }
}
```

Output:

```
vbnet
CopyEdit
AAA
java.lang.String
1
java.lang.Integer
1.0
java.lang.Double
```

---

7. Bounded Types

- Type parameters can be **restricted to a range** using the `extends` keyword.

Unbounded Type Example:

```
java
CopyEdit
class A<T> { }

A<String> a = new A<>();
A<Integer> b = new A<>();
A<Employee> c = new A<>();
```

Bounded Type: `extends` a Class

```
java
CopyEdit
class A<T extends Number> { }

A<Number> a = new A<>();   // Valid
A<Integer> b = new A<>();  // Valid
A<Float> c = new A<>();    // Valid
// A<String> d = new A<>(); // Invalid
```

Bounded Type: `extends` an Interface

```
java
CopyEdit
class A<T extends Serializable> { }

A<Number> a = new A<>();   // Valid
A<Integer> b = new A<>();  // Valid
A<Float> c = new A<>();    // Valid
A<String> d = new A<>();   // Valid
```

Invalid Bounded Types:

```
java
CopyEdit
// class A<T extends Integer, String> { }      // Invalid: Cannot extend
multiple classes
// class A<T implements Serializable> { }      // Invalid: Use `extends` for
interfaces
// class A<T super Integer> { }                // Invalid: Use `super` only
with wildcards
```

Multiple Bounds:

- You can use **one class** + **multiple interfaces**:

```
java
CopyEdit
class A<T extends Number & Runnable> { }
class A<T extends Serializable & Comparable> { }
```