

Unit 8: Generics

Generics

Generics means parameterized types.

The idea is to allow type (Integer, String, ... etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types."

Advantages of using Generics

1. **Code Reuse:** Write a method, class, or interface once and use it for any type you want.
2. **Type Safety:** Errors are caught at compile-time, rather than runtime, preventing issues like `ClassCastException`.
3. **Less Code:** Reduce the amount of code written, making it more concise and efficient.
4. **Avoid Subclassing:** Generics allow you to avoid subclassing, which can lead to brittle class hierarchies and difficult-to-understand classes.
5. **Stronger Typing:** Specify more precisely what you intend to do, making the code more robust and efficient.
6. **More Efficient:** Avoid unnecessary type conversions and reduce boilerplate code.
7. **Easier Maintenance:** Write code that is less error-prone and easier to maintain.
8. **Improved Code Clarity:** Write code that is clearer and more concise.

Generic Classes

A generic class in Java is a class that can operate on a specific type specified by the programmer at compile time. To achieve this, the class uses type parameters that act as variables that represent types (such as `int` or `String`).

Example of Generic Class with Single Type Parameter:



```
public class Employee<T> {  
    private T employeeId;  
  
    public Employee(T employeeId) {  
        this.employeeId = employeeId;  
    }  
  
    public T getEmployeeId() {  
        return employeeId;  
    }  
  
    public void setEmployeeId(T employeeId) {  
        this.employeeId = employeeId;  
    }  
  
    @Override  
    public String toString() {  
        return "Employee{" + "employeeId=" + employeeId + '}';  
    }  
}
```

Example of Generic Class with Multiple Type Parameters:

```
public class Student<U, V> {  
    private U studentId;  
    private V studentName;  
  
    public Student(U studentId, V studentName) {  
        this.studentId = studentId;  
        this.studentName = studentName;  
    }  
  
    public U getStudentId() {  
        return studentId;  
    }  
  
    public V getStudentName() {  
        return studentName;  
    }  
  
    public void setStudentId(U studentId) {  
        this.studentId = studentId;  
    }  
  
    public void setStudentName(V studentName) {  
        this.studentName = studentName;  
    }  
  
    @Override  
    public String toString() {  
        return "Student{" + "studentId=" + studentId + ", studentName=" + studentName + '}';  
    }  
}
```

Example of Using Generic Class:



```
public class Main {  
    public static void main(String[] args) {  
        Employee<String> employee = new Employee<>("E123");  
        System.out.println(employee.getEmployeeId()); // prints "E123"  
  
        Student<Integer, String> student = new Student<>(1, "John Doe");  
        System.out.println(student.getStudentId()); // prints 1  
        System.out.println(student.getStudentName()); // prints "John Doe"  
    }  
}
```

Generic Method

A generic method is a method that introduces its own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared.

Generic methods can be used with any type, making them more flexible and reusable. The compiler ensures the correctness of the type used, making the code more type-safe.

Example of Generic Method



```
public <T> List<T> genericMethod(List<T> list) {  
    return list.stream().collect(Collectors.toList());  
}
```

Explanation

In this example, the `genericMethod` takes a `List<T>` as an argument and returns a new `List<T>`. The type parameter `T` is specified before the method's return type. This method can be used with any type, such as `String`, `Integer`, or `Building`

Example of Using Generic Method:



```
List<String> stringList = Arrays.asList("Hello", "World");  
List<String> newList = genericMethod(stringList);  
System.out.println(newList); // prints [Hello, World]
```

```
List<Integer> intList = Arrays.asList(1, 2, 3);  
List<Integer> newIntList = genericMethod(intList);  
System.out.println(newIntList); // prints [1, 2, 3]
```

Explanation

In this example, we use the `genericMethod` with a `List<String>` and a `List<Integer>`. The method returns a new list with the same type as the input list.

Generic Constructor

A generic constructor is a constructor that introduces its own type parameters. It is similar to declaring a generic type, but the type parameter's scope is limited to the constructor where it is declared. Generic constructors can be used with any type, making them more flexible and reusable. The compiler ensures the correctness of the type used, making the code more type-safe.

Example



```
public class GenericEntry<T> {  
    private T data;  
  
    public <E> GenericEntry(E element) {  
        this.data = (T) element;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

Explanation

In this example, the `GenericEntry` class has a generic constructor that takes an element of type `E`, which is different from the class's generic type `T`. This allows us to create instances of `GenericEntry` with different types.

Example of Using Generic Constructor



```
GenericEntry<String> stringEntry = new GenericEntry<>("Hello");  
String data = stringEntry.getData(); // returns "Hello"
```

```
GenericEntry<Integer> intEntry = new GenericEntry<>(42);  
Integer data = intEntry.getData(); // returns 42
```


Polymorphism in Generic


Polymorphism in generics allows you to define a method, class, or interface that can operate on objects of various types, while providing a level of abstraction. In Java, generics enable polymorphism by allowing you to create classes, methods, and interfaces that can work with different types while ensuring type safety.

How Polymorphism is Used in Generics

1. **Type Parameterization:** Generics allow you to parameterize types. This means you can define a class, interface, or method with a placeholder for a type, which can be specified when the class is instantiated or the method is called.
2. **Bounded Type Parameters:** You can restrict the types that can be used as type arguments by specifying bounds. This is done using the `extends` keyword to limit the types to a particular class or interface and its subclasses.
3. **Type Safety:** Generics provide compile-time type checking, which reduces runtime errors by catching them early during the compilation.


Example: Polymorphism with Generics in Java

Creating a Generic Interface:



```
public interface Animal {  
    void makeSound();  
}  
  
public class Dog implements Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Woof");  
    }  
}  
  
public class Cat implements Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Meow");  
    }  
}
```

Using a Generic Class with Bounded Type Parameters:



```
public class AnimalShelter<T extends Animal> {  
    private List<T> animals = new ArrayList<>();  
  
    public void addAnimal(T animal) {  
        animals.add(animal);  
    }  
  
    public void makeAllSounds() {  
        for (T animal : animals) {  
            animal.makeSound();  
        }  
    }  
}
```

Using the Generic Class with Different Types:



```
public class Main {  
    public static void main(String[] args) {  
        AnimalShelter<Dog> dogShelter = new AnimalShelter<>();  
        dogShelter.addAnimal(new Dog());  
        dogShelter.addAnimal(new Dog());  
  
        AnimalShelter<Cat> catShelter = new AnimalShelter<>();  
        catShelter.addAnimal(new Cat());  
        catShelter.addAnimal(new Cat());  
  
        System.out.println("Dog Shelter:");  
        dogShelter.makeAllSounds();  
  
        System.out.println("Cat Shelter:");  
        catShelter.makeAllSounds();  
    }  
}
```

Explanation

1. **Type Parameterization:** The AnimalShelter class is defined with a type parameter T, which allows it to work with any type that extends Animal. When creating an instance of AnimalShelter, the type parameter is specified (e.g., AnimalShelter<Dog>).
2. **Bounded Type Parameters:** The type parameter T is bounded by Animal (T extends Animal), meaning T can only be Animal or a subclass of Animal.
3. **Type Safety:** The generic type ensures that only Animal or its subclasses can be added to the shelter, preventing type errors at runtime. The method makeAllSounds can call makeSound on each animal in the list without casting, thanks to the type constraint.

Real World Use Case of Generics

https://drive.google.com/file/d/1Q_RgVpIU_IAtANI_0fE0eT2_GAkNivGk/view?usp=sharing

Practical

1. Write a generic class `Box` that can hold any type of object. Demonstrate how generics can help prevent runtime type errors by trying to add incompatible types without using generics.
2. Create a generic class `Pair` that holds two values of potentially different types. Add methods to get the values and set new values. Demonstrate its usage with different type pairs.
3. Write a generic class `Container` that holds an array of objects. Provide methods to add, get, and remove elements from the container. Ensure that the container dynamically resizes as needed.
4. Write a generic method `swap` that swaps the positions of two elements in an array. Demonstrate this method with arrays of different types (e.g., `Integer`, `String`).
5. Write a generic method `printArray` that prints all elements of an array. Test this method with arrays of different types.

Resources

<https://www.youtube.com/watch?v=K1iu1kXkVoA>

<https://www.javatpoint.com/generics-in-java>

<https://www.geeksforgeeks.org/generics-in-java/>

<https://www.javacodegeeks.com/2015/03/polymorphism-in-java-generics.html>