Adv Java 01 - Generics

Agenda

- Intro to Generics
  - Generic Classes
  - Generic Methods
  - Wildcards in Generics
  - Bounded Generics
  - Generic Interfaces
- Additional Concepts
  - Type Erasure

Introduction

Generics in Java provide a way to create classes, interfaces, and methods with a type parameter. This allows you to write code that can work with different types while providing compile-time type safety. In this beginner-friendly tutorial, we'll explore the basics of Java generics.

Generics offer several benefits:

1. Type Safety: Generics provide compile-time type checking, reducing the chances of runtime errors.

2. Code Reusability: You can write code that works with different types without duplicating it.

3. Elimination of Type Casting: Generics eliminate the need for explicit type casting, making the code cleaner.

Generic Classes

A generic class is a class that has one or more type parameters. Here's a simple example of a generic class.

```java
public class Box<T> {
    private T content;

    public void addContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

In this example, T is a type parameter. You can create instances of Box for different types:

```java
Box<Integer> intBox = new Box<>();
intBox.addContent(42);
System.out.println("Box Content: " + intBox.getContent());  // Output: 42

Box<String> stringBox = new Box<>();
stringBox.addContent("Hello, Generics!");
System.out.println("Box Content: " + stringBox.getContent());  // Output: Hello,
Generics!
```

Generic Methods

You can also create generic methods within non-generic classes. Here's an example:

```java
public class Util {
    public <E> void printArray(E[] array) {
        for (E element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

You can use this method with different types:

```java
Integer[] intArray = {1, 2, 3, 4, 5};
String[] stringArray = {"apple", "banana", "orange"};

Util util = new Util();
util.printArray(intArray);    // Output: 1 2 3 4 5
util.printArray(stringArray); // Output: apple banana orange
```

Wildcard in Generics

The wildcard (?) is used to represent an unknown type. Let's see an example.

```java
public class Printer {
    public static void printList(List<?> list) {
        for (Object item : list) {
            System.out.print(item + " ");
        }
        System.out.println();
    }
}
```

You can use this method with lists of different types:

```java
List<Integer> intList = Arrays.asList(1, 2, 3);
List<String> stringList = Arrays.asList("apple", "banana", "orange");

Printer.printList(intList);    // Output: 1 2 3
Printer.printList(stringList); // Output: apple banana orange
```

Bounded Generics

Remember that type parameters can be bounded. Bounded means “restricted,” and we can restrict the types that a method accepts.

For example, we can specify that a method accepts a type and all its subclasses (upper bound) or a type and all its superclasses (lower bound).

Type bounds restrict the types that can be used as arguments in a generic class or method. You can use extends or super to set upper or lower bounds.

```
public class NumberBox<T extends Number> {
    private T content;

    public void addContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

In this example, T must be a subclass of Number.

To declare an upper-bounded type, we use the keyword extends after the type, followed by the upper bound that we want to use:

```
public <T extends Number> List<T> fromArrayToList(T[] a) {
    ...
}
```

We use the keyword extends here to mean that the type T extends the upper bound in case of a class or implements an upper bound in case of an interface.

There are two types of wildcards: `? extends T` and `? super T`. The former is for upper-bounded wildcards, and the latter is for lower-bounded wildcards.

Consider this example:

```
public static void paintAllBuildings(List<Building> buildings) {
    buildings.forEach(Building::paint);
}
```

If we imagine a subtype of Building, such as a House, we can’t use this method with a list of House, even though House is a subtype of Building.

If we need to use this method with type Building and all its subtypes, the bounded wildcard can do the magic:

```
public static void paintAllBuildings(List<? extends Building> buildings) {
    ...
}
```

Now this method will work with type Building and all its subtypes. This is called an upper-bounded wildcard, where type Building is the upper bound.

We can also specify wildcards with a lower bound, where the unknown type has to be a supertype of the specified type. Lower bounds can be specified using the super keyword followed by the specific type. For example, <? super T> means unknown type that is a superclass of T (= T and all its parents).

Generic Interfaces

Interfaces can also be generic. For example:

```
public interface Pair<K, V> {
    K getKey();
    V getValue();
}
```

You can implement this interface with different types:

```
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public K getKey() {
        return key;
    }

    @Override
    public V getValue() {
        return value;
    }
}
```

Additional Concepts

Type Erasure

Type erasure is a feature in Java generics where the type parameters used in generic code are removed (or erased) during compilation. This means that the generic type information is not available at runtime, and the generic types are replaced with their upper bounds or Object type.

How Type Erasure Works:

1. Compilation Phase:

During the compilation phase, Java generics are type-checked to ensure type safety. The compiler replaces all generic types with their upper bounds or with Object if no bound is specified.

2. Type Erasure:

The compiler removes all generic type information and replaces it with casting or Object. This process is known as type erasure, and it allows Java to maintain backward compatibility with non-generic code. Example: Consider the following generic class:

```java
public class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

After compilation, the generic type T is replaced with Object:

```java
public class Box {
    private Object content;

    public void setContent(Object content) {
        this.content = content;
    }

    public Object getContent() {
        return content;
    }
}
```

Implications of Type Erasure:

1. Loss of Type Information at Runtime:

Type information about generic types is not available at runtime due to type erasure. For example, you can't determine the actual type parameter used for a generic class or method at runtime.

2. Bridge Methods:

When dealing with generic methods in classes or interfaces, the compiler generates bridge methods to maintain compatibility with pre-generics code. 3. Arrays and Generics:

Due to type erasure, arrays of generic types are not allowed. You can't create an array of a generic type like T[] array = new T[5];.

4. Casting and Unchecked Warnings:

Type casts may be necessary when working with generic types, and this can lead to unchecked warnings. For example, when casting to a generic type, the compiler issues a warning because it can't verify the type at runtime.

```java
Box<Integer> integerBox = new Box<>();
integerBox.setContent(42);

// Warning: Unchecked cast
int value = (Integer) integerBox.getContent();
```

Summary

Type erasure is a mechanism in Java generics that removes generic type information during compilation to maintain compatibility with non-generic code. While this approach allows for seamless integration with existing code, it also means that certain generic type information is not available at runtime. Developers need to be aware of the implications of type erasure, such as potential unchecked warnings and limitations on working with arrays of generic types. -- End --