ENGINEER SERIES

IN JAVA LANGUAGE

7 CH

GENERICS

ENG : Qusay Khudair

Creativity and Accuracy in Work

AUG PLUS .Edu

Chapter 15

Generics

ENG : Qusay khudair

Introduction to Generics

- **Definition: Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods.**
- **Using Generics, it is possible to create classes that work with different data types.**
- **Why Generics ?**
    - **Type Safety: Ensures that only a specific type of objects can be stored in a collection.**
    - **Code Reusability: The same code can work with different types of data.**
    - **Elimination of Type Casting: No need to cast objects when retrieving from a collection.**

Type Parameters in Java Generics

- **The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:**
    - **T – Type**
    - **E – Element**
    - **K – Key**
    - **N – Number**
    - **V – Value**

Generic Classes

```
public class name<Type> {
or
public class name<Type1, Type2, ..., TypeN> {
```

Syntax Example:

```
class Box<T> {
   private T item;
   public void setItem(T item) {
      this.item = item;}
   public T getItem() {
      return item;}}
```

- **Explanation: Here, T is a type parameter that will be replaced with a specific type when an instance of Box is created.**

Cast Exceptions at Runtime

- **Explanation: With generics, most type issues are caught at compile-time. However, unchecked casts can lead to `Class Cast Exception` at runtime if not used carefully.**

Example:

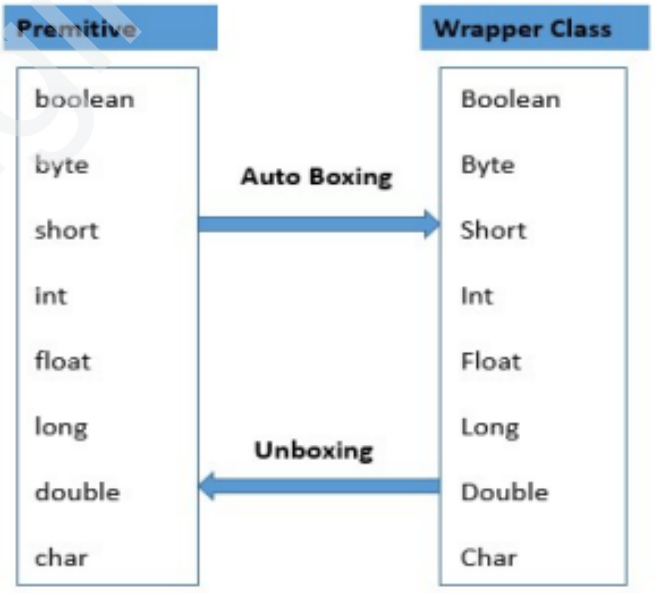
```
List list = new ArrayList<String>();

list.add(123);  // No compile-time error, but
potential runtime issue
```

Autoboxing and AutoUnboxing

- **Autoboxing: The automatic conversion of primitive types to their corresponding object wrapper classes (e.g., `int` to `Integer`).**
- **AutoUnboxing: The automatic conversion of wrapper class objects to their corresponding primitive types.**

Example:

```
Integer num = 5; // Autoboxing
int n = num;     // AutoUnboxin
```



Generic Methods

Syntax Example:

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

- **Explanation: The method printArray can accept arrays of any type, making it highly reusable.**

Generic Static Algorithms

- **Definition: Static methods can also be generic, allowing them to operate on various types while maintaining type safety.**

Example:

```
public class GenericAlgorithms {

    // Generic static method to swap elements in an array
    public static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
    // Generic static method to print an array
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Integer array
        Integer[] intArray = {1, 2, 3, 4, 5};
        System.out.println("Original Integer array:");
        printArray(intArray);

        // Swap elements at index 1 and 3
        swap(intArray, 1, 3);
        System.out.println("Integer array after swap:");
        printArray(intArray);

        // String array
        String[] strArray = {"Apple", "Banana", "Cherry",
"Date"};
        System.out.println("Original String array:");
        printArray(strArray);
        // Swap elements at index 0 and 2
        swap(strArray, 0, 2);
        System.out.println("String array after swap:");
        printArray(strArray); }}
```

```
Output

Original Integer array:

 1 2 3 4 5

 Integer array after swap:

 1 4 3 2 5

Original String array:

Apple Banana Cherry Date

 String array after swap:

 Cherry Banana Apple Date
```

Generic Interfaces

Example:

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

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;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }}
```

- **Explanation: This allows the `Pair` interface to be used with any types for keys and values.**

Java Comparable Interface

- **Definition: The `Comparable` interface is generic, allowing objects of the implementing class to be compared to one another.**
- **Java Comparable interface is used to order the objects. This interface is defined in the java.lang package and contains only one method named compareTo(Obj).**
- **It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only.**
- **public int compareTo(T obj) : It is used to compare the current object with the specified object. It returns ::**

▪positive integer : if the current object is greater than the specified object. ▪negative integer : if the current object is less than the specified object.

▪zero : if the current object is equal to the specified object.

Full Example:

```
class Person implements Comparable<Person> {
          private String name;
          private int age;
          public Person(String name, int age) {
              this.name = name;
              this.age = age; }
          public String getName() {
             return name;  }
          public int getAge() {
              return age;}
          @Override
          public int compareTo(Person other) {
              // Compare based on name first
              int nameComparison = this.name.compareTo(other.name);
              if (nameComparison != 0) {
                  return nameComparison;
              } else {
                  // If names are the same, compare based on age
                  return Integer.compare(this.age, other.age); }}
          @Override
          public String toString() {
              return name + " (" + age + ")";}}
```

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

public class ComparableExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("John", 30));
        people.add(new Person("Alice", 25));
        people.add(new Person("Bob", 30));
        people.add(new Person("Alice", 22));
        // Before sorting
        System.out.println("Before sorting:");
        for (Person person : people) {
            System.out.println(person);
        }
        // Sorting the list using Comparable
        Collections.sort(people);
        // After sorting
        System.out.println("\nAfter sorting:");
        for (Person person : people) {
            System.out.println(person);} }}
```

```
Output :

Before sorting:

John (30)

Alice (25)

Bob (30)

Alice (22)


After sorting:

Alice (22)

Alice (25)

Bob (30)

John (30)
```

Bounded Type Parameters

- **Definition: Restricts the types that can be used as arguments for a type parameter.**

Syntax Example:

```
public <T extends Number> void add(T num1, T num2) {
   System.out.println(num1.doubleValue() + num2.doubleValue());
}
```

- **Explanation: Here, T can only be a type that extends Number, ensuring that num1 and num2 have numeric methods like doubleValue().**

Full Example :

```
class Student implements Comparable<Student> {
int rollno;
String name;
int age;

Student(int rollno, String name, int age) {
this.rollno = rollno;
this.name = name;
this.age = age;
}

public int compareTo(Student st) {
if (age == st.age)
return 0;
else if (age > st.age)
return 1;
else
return -1;
}

public static void main(String args[]) {

Student s1 = new Student(10,"Ahmed",19);
Student s2 = new Student(11,"Rami",20);
Student s3 = new Student(12,"Tamer",19);
Student result4 = GenericMaxMethodTest .max(s1,s2,s3);
System.out.println(result4);}}
```

```
public class GenericMaxMethodTest {


public static <T extends Comparable<T>> T max(T x,T y, T z) {
T largest = x;
if(y.compareTo(largest) > 0){
largest =y;
}

if(z.compareTo(largest) >0) {
largest = z;
}

return largest;
}
```

Wildcards in Generics

- **Types of Wildcards:**
  - **? : Represents an unknown type (Unbounded Wildcards).**

- **Use unbounded wildcards when any type parameter works.**
- **is used to specify unbounded wildcards.**
- **The following are legal statements**

```
Box b1 = new Box<Integer>(31);

Box b2 = new Box<String>("Hi");

b1 = b2;
```

- **Wildcard capture:**

The compiler can figure out exactly what type b1 is above from the right hand side of the assignments.

This "capturing" of type information means:

1. The type on the left hand doesn't need to be specified.

2. The compiler can do additional type checks because it knows the type of b1.

- **? extends T: A wildcard with an upper bound**

  (e.g., List<? extends Number>).

```
public class Box<E> {
  public void copyFrom(Box<E> b) {
    this.data = b.getData();
  }
}
//We have seen this earlier
//We can rewrite copyFrom() so that it can take a box
//that contains data that is a subclass of E and
//store it to a Box<E> object

public class Box<E> {
  public void copyFrom(Box<? extends E> b) {
    this.data = b.getData(); //b.getData() is a subclass of this.data
  }
}
```

<? extends E> is called "*upper bounded wildcard*" because it defines a type that is bounded by the superclass E.

- **? super T: A wildcard with a lower bound**

  (e.g., List<? super Integer>).

```
public void copyTo(Box<E> b) {
  b.data = this.getData();
}
```

Above code is fine as long as b and the host are boxes of exactly same type. But b could be a box of an object that is a superclass of E.

This can be expressed as:

```
public void copyTo(Box<? super E> b) {
  b.data = this.getData();
  //b.data() is a superclass of this.data()
}
```

<? super E> is called a "*lower bounded wildcard*" because it defines a type that is bounded by the subclass E.

- Note : Java allows multiple inheritance in the form of implementing multiple interfaces. So multiple bounds may be necessary to specify a type parameter.

The following syntax is used then:

```
<T extends A & B & C & ...>
```

For Example:

```
interface A {
...
}
interface B {
...
}
class MultiBounds<T extends A & B> {
...
}
```

Erasure of Generics

- **Explanation: Java implements generics using a technique called "type erasure," which means that generic types are replaced with their raw types at runtime.**

Generics Work Only with Reference Types

- When we declare an instance of a generic type, the type argument passed to the type parameter must be a reference type.
- We cannot use primitive data types like int, char.

```
Test<int> obj = new Test<int>(20);
```

- The above line results in a compile-time error that can be resolved using type wrappers to encapsulate a primitive type.
- But primitive type arrays can be passed to the type parameter because arrays are reference types.

```
ArrayList<int[]> a = new ArrayList<>();
```
