



Generics (Chapter 19)

What is Generics?

Java Generics, sometimes used as a plural noun (generics) and sometimes as a singular noun (Generics), is a language feature of Java that allows for the definition and use of **generic types and methods**.

Generic types or methods differ from regular types and methods in that they have **type parameters**.

A class like **ArrayList<E>** is a generic type. It has a type parameter **E** that represents the type of the elements stored in the list. Instead of just using an **ArrayList**, not saying anything about the type of elements the list contains, we can use an **ArrayList<String>** or an **ArrayList<Integer>**.

Instantiations, such as **ArrayList<String>** or **ArrayList<Integer>**, are called **parameterized types**, and **String** and **Integer** are the respective **actual type arguments**.

What is Generics?

- Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Much like the more familiar formal parameters used in method declarations, **type parameters** provide a way for you to re-use the same code with different inputs.
- The difference is that the inputs to formal parameters are **values**, while the inputs to type parameters are **types**.

What is Generics?

- In other words, with generics you can define a class or a method with **generic types** that can be substituted using **concrete types** by the compiler.
- For example, you may define a generic stack class that stores the elements of a generic type. From this generic class, you may create a stack object for holding strings and a stack object for holding numbers.
- Here, strings and numbers are concrete types that replace the generic type.

Comparing and Contrasting Integer Stack

Build a Stack class according to the following UML diagram and add a class to test it:

StackOfIntegers

-elements: int[]

-size: int

+StackOfIntegers()

+StackOfIntegers(capacity: int)

+empty(): boolean

+peek(): int

+push(value: int): void

+pop(): int

+getSize(): int

An array to store integers in the stack.

The number of integers in the stack.

Constructs an empty stack with a default capacity of 16.

Constructs an empty stack with a specified capacity.

Returns true if the stack is empty.


Returns the integer at the top of the stack without removing it from the stack.

Stores an integer into the top of the stack.

Removes the integer at the top of the stack and returns it.

Returns the number of elements in the stack.

```
1 public class StackOfIntegers {
2     private int[] elements;
3     private int size;
4     public static final int DEFAULT_CAPACITY = 16;
5
6     /** Construct a stack with the default capacity 16 */
7     public StackOfIntegers() {
8         this(DEFAULT_CAPACITY);
9     }
10
11     /** Construct a stack with the specified maximum capacity */
12     public StackOfIntegers(int capacity) {
13         elements = new int[capacity];
14     }
15
16     /** Push a new integer into the top of the stack */
17     public void push(int value) {
18         if (size >= elements.length) {
19             int[] temp = new int[elements.length * 2];
20             System.arraycopy(elements, 0, temp, 0, elements.length);
21             elements = temp;
22         }
23
24         elements[size++] = value;
25     }
26
27     /** Return and remove the top element from the stack */
28     public int pop() {
29         return elements[--size];
30     }
31
32     /** Return the top element from the stack */
33     public int peek() {
34         return elements[size - 1];
35     }
36
37     /** Test whether the stack is empty */
38     public boolean empty() {
39         return size == 0;
40     }
41
42     /** Return the number of elements in the stack */
43     public int getSize() {
44         return size;
45     }
46 }
```



What would you do if you
also need a String stack?

Comparing and Contrasting Generic Stack

- A generic type can be defined for a class or interface.
- A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.

GenericStack<E>

-list: java.util.ArrayList<E>

+GenericStack()

+getSize(): int

+peek(): E

+pop(): E

+push(o: E): E

+isEmpty(): boolean

An array list to store elements.

Creates an empty stack.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns true if the stack is empty.

Comparing and Contrasting Generic Stack

```
1 public class GenericStack<E> {
2     private java.util.ArrayList<E> list = new java.util.ArrayList<E>();
3
4     public int getSize() {
5         return list.size();
6     }
7
8     public E peek() {
9         return list.get(getSize() - 1);
10    }
11
12    public void push(E o) {
13        list.add(o);
14    }
15
16    public E pop() {
17        E o = list.get(getSize() - 1);
18        list.remove(getSize() - 1);
19        return o;
20    }
21
22    public boolean isEmpty() {
23        return list.isEmpty();
24    }
25
26    @Override
27    public String toString() {
28        return "stack: " + list.toString();
29    }
30 }
```

Generic Methods

- Generic methods are methods that introduce their 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.
- Static and non-static generic methods are allowed, as well as generic class constructors.
- The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type.
- For static generic methods, the type parameter section must appear before the method's return type.

Generic Methods

```
1 public class GenericMethodDemo {
2     public static void main(String[] args ) {
3         Integer[] integers = {1, 2, 3, 4, 5};
4         String[] strings = {"London", "Paris", "New York", "Austin"};
5
6         GenericMethodDemo.<Integer>print(integers);
7         GenericMethodDemo.<String>print(strings);
8     }
9
10    public static <E> void print(E[] list) {
11        for (int i = 0; i < list.length; i++)
12            System.out.print(list[i] + " ");
13        System.out.println();
14    }
15 }
```

Bounded Generic Type

- There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of **Number or its subclasses**.
- This is what **bounded type parameters** are for.
- To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its upper bound, which in this example is `Number`.
- Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

Bounded Generic Type

```
1 public class BoundedTypeDemo {
2     public static void main(String[] args ) {
3         Rectangle rectangle = new Rectangle(2, 2);
4         Circle circle = new Circle(2);
5
6         System.out.println("Same area? " +
7             equalArea(rectangle, circle));
8     }
9
10    public static <E extends GeometricObject> boolean equalArea(
11        E object1, E object2) {
12        return object1.getArea() == object2.getArea();
13    }
14 }
```

Bounded Generic Type

Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

`<T extends B1 & B2 & B3>`

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
```

```
interface B { /* ... */ }
```

```
interface C { /* ... */ }
```

```
class D <T extends A & B & C> { /* ... */ }
```

If bound A is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ } // compile-time error
```

Wildcard Generic types

- A wildcard generic type has three forms: `?` and `? extends T`, as well as `? super T`, where `T` is a generic type.
- The first form, `?`, called an **unbounded wildcard**, is the same as `? extends Object`.
- The second form, `? extends T`, called a **bounded wildcard**, represents `T` or a subtype of `T`.
- The third form, `? super T`, called a **lower-bound wildcard**, denotes `T` or a supertype of `T`.

Wildcard Generic Types

Consider the following example

This is a compile error. Why?

```
1  public class WildCardDemo1 {
2      public static void main(String[] args ) {
3          GenericStack<Integer> intStack = new GenericStack<Integer>();
4          intStack.push(1); // 1 is autoboxed into new Integer(1)
5          intStack.push(2);
6          intStack.push(-2);
7
8          // Error:      System.out.print("The max number is " + max(intStack));
9      }
10
11     /** Find the maximum in a stack of numbers */
12     public static double max(GenericStack<Number> stack) {
13         double max = stack.pop().doubleValue(); // initialize max
14
15         while (!stack.isEmpty()) {
16             double value = stack.pop().doubleValue();
17             if (value > max)
18                 max = value;
19         }
20
21         return max;
22     }
23 }
```


Wildcard Generic Types

Fix for Previous Example

```
1  public class AnyWildcardDemo {
2      public static void main(String[] args ) {
3          GenericStack<Integer> intStack = new GenericStack<>();
4          intStack.push(1); // 1 is autoboxed into new Integer(1)
5          intStack.push(2);
6          intStack.push(-2);
7
8          print(intStack);
9      }
10
11     /** Prints objects and empties the stack */
12     public static void print(GenericStack<?> stack) {
13         while (!stack.isEmpty()) {
14             System.out.print(stack.pop() + " ");
15         }
16     }
17 }
```

Wildcard Generic Types

Super Example

```
1  public class SuperWildcardDemo {
2      public static void main(String[] args) {
3          GenericStack<String> stack1 = new GenericStack<>();
4          GenericStack<Object> stack2 = new GenericStack<>();
5          stack2.push("Java");
6          stack2.push(2);
7          stack1.push("Sun");
8          add(stack1, stack2);
9          AnyWildcardDemo.print(stack2);
10     }
11
12     public static <T> void add(GenericStack<T> stack1,
13         GenericStack<? super T> stack2) {
14         while (!stack1.isEmpty())
15             stack2.push(stack1.pop());
16     }
17 }
```

Raw Type and Backward Compatibility

- A generic class or interface used without specifying a concrete type, called a **raw type**, enables backward compatibility with earlier versions of Java.

```
// raw type
```

```
ArrayList list = new ArrayList();
```

This is *roughly* equivalent to

```
ArrayList<Object> list = new ArrayList<Object>();
```

Raw Type is Unsafe

```
// Max.java: Find a maximum object
public class Max {
    /** Return the maximum between two objects */
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

Runtime Error:

```
Max.max("Welcome", 23);
```

Make it Safe

```
// Max1.java: Find a maximum object
```

```
public class Max1 {
```

```
    /** Return the maximum between two objects */
```

```
    public static <E extends Comparable<E>> E max(E o1, E o2) {
```

```
        if (o1.compareTo(o2) > 0)
```

```
            return o1;
```

```
        else
```

```
            return o2;
```

```
    }
```

```
}
```

```
Max.max("Welcome", 23);
```

Important Facts

It is important to note that a generic class is shared by all its instances regardless of its actual generic type.

```
GenericStack<String> stack1 = new GenericStack<String>();  
GenericStack<Integer> stack2 = new GenericStack<Integer>();
```

Although `GenericStack<String>` and `GenericStack<Integer>` are two types, but there is only one class `GenericStack` loaded into the JVM.

Restrictions on Generics

Restriction 1: Cannot use `new E()`.

You cannot create an instance using a generic type parameter. For example, the following statement is wrong:

```
E object = new E();
```

The reason is that `new E()` is executed at runtime, but the generic type `E` is not available at runtime.

Restrictions on Generics

Restriction 2: Cannot use new E[]).

You cannot create an array using a generic type parameter. For example, the following statement is wrong:

```
E[] elements = new E[capacity];
```

You can circumvent this limitation by creating an array of the Object type and then casting it to E[], as follows:

```
E[] elements = (E[])new Object[capacity];
```


Restrictions on Generics

Restriction 3: A Generic Type Parameter of a Class Is Not Allowed in a Static Context.

Since all instances of a generic class have the same runtime class, the static variables and methods of a generic class are shared by all its instances. Therefore, it is illegal to refer to a generic type parameter for a class in a static method, field, or initializer. For example, the following code is illegal:

```
public class Test<E> {  
    public static void m(E o1) { // Illegal  
    }  
  
    public static E o1; // Illegal  
}
```

Restrictions on Generics

Restriction 4: Exception Classes Cannot be Generic.

A generic class may not extend `java.lang.Throwable`, so the following class declaration would be illegal:

```
public class MyException<T> extends Exception {  
}
```

Benefits of Using Generics?

1. Stronger type checks at compile time.
 - A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.
 - Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

Benefits of Using Generics?

2. Elimination of casts.

- The following code snippet without generics requires casting:

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

- When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0); // no cast
```

Benefits of Using Generics?

3. Enabling programmers to implement generic algorithms.

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

More About Generics Here

<https://docs.oracle.com/javase/tutorial/java/generics/index.html>

Programming Challenge

Implement the following method using binary search.

```
public static <E extends Comparable<E>> int binarySearch(E[] list, E key)
```

Programming Challenge

Write the following method that shuffles an ArrayList:

```
public static <E> void shuffle(ArrayList<E> list)
```


Programming Challenge

- Implement the following method using selection sort.

```
public static <E extends Comparable<E>> void selectionSort(E[]  
list)
```

Programming Challenge

- Write the following method that returns a new ArrayList.
- The new list contains the non-duplicate elements from the original list.

```
public static <E> ArrayList<E> removeDuplicates(ArrayList<E> list)
```