Generics

WIA1002/ WIB1002 :
Data structure

Content

- Generics
- ArrayList
- Generic Classes and Interfaces
- Generic Methods
- Bounded Generics
- Raw Type
- Wildcards
- Erasure
- Restriction on Generics

What is Generics?

- ***Generics*** is the capability to parameterize types. With this capability, 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 class (e.g. “array”) that stores the elements of a generic type. From this generic class, you may create an “array” for holding strings and an “array” for holding numbers. Here, strings and numbers are concrete types that replace the generic type.
- The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Why Generics?

- Stronger type checks at compile time. The key benefit of generics is to enable errors to be detected at compile time rather than at runtime.

- A generic class or method permits you to **specify allowable types of objects** that the **class or method** may **work with**. If you attempt to use the class or method with an **incompatible object**, a **compile error occurs**.

Why Generics?

```
public class Box {
   private Comparable item;
   boolean full;

   public Box()   {
      full=false;
   }

   public void store(Comparable a){
      this.item = a;
      full=true;
   }

   public Comparable retrieve()   {
      return item;
   }

   public void remove() {
      item = null;
      full=false;
   }

   public String toString()   {
      if (full)
         return item.toString();
      else
         return "nothing";
   }
}
```

```
public class UseBox  {

   public static void main(String args[]) {
      Box box1 = new Box();
      Box box2 = new Box();

      box1.store("Hello World");
      box2.store(100);

      System.out.println("Box 1 has " + box1.toString() );
      System.out.println("Bos 2 has " + box2.toString() );

      int c = box1.retrieve().compareTo(box2.retrieve());
   }
}
```

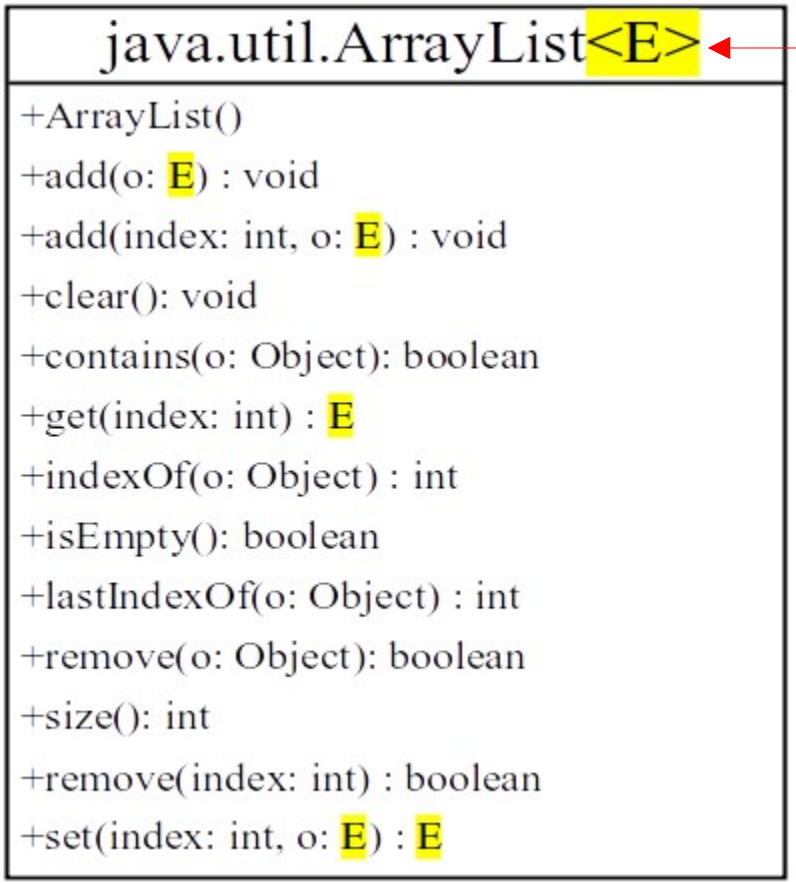
Compile ok, but
Runtime error

To further discuss the concept of generics, let’s introduce a data structure called ArrayList.

In the next 6 pages, we will learn generics along with ArrayList.

The ArrayList Class

You can create an array to store objects. But the array's size is fixed once the array is created. Java provides the ArrayList class that can be used to store an unlimited number of objects.



Formal generic type

<E> is meant to be an Element

The ArrayList Class

You can create an array to store objects. But the array's size is fixed once the array is created. Java provides the ArrayList class that can be used to store an unlimited number of objects.

java.util.ArrayList<E>	
`+ArrayList()`	Creates an empty list.
`+add(o: E) : void`	Appends a new element o at the end of this list.
`+add(index: int, o: E) : void`	Adds a new element o at the specified index in this list.
`+clear(): void`	Removes all the elements from this list.
`+contains(o: Object): boolean`	Returns true if this list contains the element o.
`+get(index: int) : E`	Returns the element from this list at the specified index.
`+indexOf(o: Object) : int`	Returns the index of the first matching element in this list.
`+isEmpty(): boolean`	Returns true if this list contains no elements.
`+lastIndexOf(o: Object) : int`	Returns the index of the last matching element in this list.
`+remove(o: Object): boolean`	Removes the element o from this list.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int) : boolean`	Removes the element at the specified index.
`+set(index: int, o: E) : E`	Sets the element at the specified index.

The ArrayList Class

ArrayList is known as a generic class with a generic type E. You can specify a concrete type to replace E when creating an ArrayList. For example, the following statement creates an ArrayList and assigns its reference to variable cities. This ArrayList object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();
```

Or, can be written as:

```
ArrayList<String> cities = new ArrayList<>();
```

Differences and Similarities between arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

```java
import java.util.ArrayList;

public class TestArrayList {
  public static void main(String[] args) {
    ArrayList<String> cityList = new ArrayList<>();

    cityList.add("London");
    cityList.add("Denver");
    cityList.add("Paris");
    cityList.add("Miami");
    cityList.add("Seoul");
    cityList.add("Tokyo");

    System.out.println("List size? " + cityList.size());
    System.out.println("Is Miami in the list? " + cityList.contains("Miami"));
    System.out.println("The location of Denver in the list? " + cityList.indexOf("Denver"));
    System.out.println("Is the list empty? " + cityList.isEmpty());

    cityList.add(2, "Xian");
    cityList.remove("Miami");
    cityList.remove(1);
    System.out.println(cityList.toString());

    for (int i = cityList.size() - 1; i >= 0; i--)
      System.out.print(cityList.get(i) + " ");
    System.out.println();

    ArrayList<Circle> list = new java.util.ArrayList<>();

    list.add(new Circle(2));
    list.add(new Circle(3));

    System.out.println("The area of the circle? " +
      list.get(0).getArea());
  }
}
```

```java
import java.util.ArrayList;

public class TestArrayList {
  public static void main(String[] args) {
    ArrayList<String> cityList = new ArrayList<>();

    cityList.add("London");
    cityList.add("Denver");
    cityList.add("Paris");
    cityList.add("Miami");
    cityList.add("Seoul");
    cityList.add("Tokyo");

    System.out.println("List size? " + cityList.size());
    System.out.println("Is Miami in the list? " + cityList.contains("Miami"));
    System.out.println("The location of Denver in the list? " + cityList.indexOf("Denver"));
    System.out.println("Is the list empty? " + cityList.isEmpty());

    cityList.add(2, "Xian");
    cityList.remove("Miami");
    cityList.remove(1);
    System.out.println(cityList.toString());

    for (int i = cityList.size() - 1; i >= 0; i--)
      System.out.print(cityList.get(i) + " ");
    System.out.println();

    ArrayList<Circle> list = new java.util.ArrayList<>();

    list.add(new Circle(2));
    list.add(new Circle(3));

    System.out.println("The area of the circle? " +
      list.get(0).getArea());
  }
}
```

```
List size? 6
Is Miami in the list? true
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
Tokyo Seoul Paris Xian London
The area of the circle? 12.566370614359172
```

Generics: Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about
- Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
- The most commonly used type parameter names are:
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

Generic types

```
ArrayList<String> list = new ArrayList<String>();
list.add("Red");     //ok
list.add(new Integer(1));
```

list is already defined as ArrayList of String. If you attempt to add a nonstring, a compile error will occur.

Generic types

```
ArrayList<Integer> x = new ArrayList<>();  //OK
ArrayList<Double> y = new ArrayList<>();  //OK
ArrayList<int> x = new ArrayList<>();
```

- E must be reference types (e.g. Integer, Double, MyPet), cannot be a primitive type (e.g. int, double, char etc.)

No Casting Needed

- The following code snippet without generics requires casting:

```
ArrayList list = new ArrayList();
list.add("hello");
String s = (String) list.get(0); //Casting needed prior to JDK1.5 or you will get compile error - incompatible types: java.lang.Object cannot be converted to java.lang.String
```

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

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

Check Point 1

1. What is the benefit of using generic types?
2. Are there any compile errors in (a) and (b)?

```
ArrayList dates = new ArrayList();
dates.add(new Date());
dates.add(new String());
```

(a) Prior to JDK 1.5

```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
dates.add(new String());
```

(b) Since JDK 1.5

3. What is wrong in (a)? Is code in (b) correct?

```
ArrayList dates = new ArrayList();
dates.add(new Date());
Date date = dates.get(0);
```

(a) Prior to JDK 1.5

```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
Date date = dates.get(0);
```

(b) Since JDK 1.5

Generic types

- Generics can be defined for a :
  1. Class
  2. Interface
  3. Method

Declaring Generic Classes

- Syntax :

```
public class ClassName<E> { }
```

e.g.

public class GenericStack<E> {……}

Declaring Generic Interface

- Syntax :

```
public interface IntefaceName<E> { }
```

e.g.

public interface Comparable<E> {.....}

public interface Edible<E> {.....}

Example: GenericBox

```java
public class GenericBox<T>  {
    private T item;
    boolean full;

    public GenericBox() {
        full=false;
    }

    public void store(T a)  {
        this.item = a;
        full=true;
    }

    public void remove()    {
        item = null;
        full=false;
    }

    public String toString()    {
        if (full)
            return item.toString();
        else
            return "nothing";
    }

}
```

```java
public class UseGenericBox {

   public static void main(String args[]) {
      GenericBox<String> box1 = new GenericBox<>();
      GenericBox<Integer> box2 = new GenericBox<>();

      box1.store("Hello World");
      box2.store(100);

      System.out.println("Box 1 has " + box1.toString() );
      System.out.println("Bos 2 has " + box2.toString() );

      box1.remove();
      box2.remove();

      System.out.println("After removal, box 1 has " + box1.toString() );
      System.out.println("After removal, box 2 has " + box2.toString() );

      //box1.store(100);              these lines were removed because
      //box2.store("Hello World");  they caused compilation error
   }
}
```

```
Box 1 has Hello World
Bos 2 has 100
After removal, box 1 has nothing
After removal, box 2 has nothing
```

Info about Generics

1. Generics class constructor is defined as

   e.g. `public GenericStack()`
2. Generics class may have more than 1 parameter.

   e.g. `<E1, E2, E3>`
3. You can define a class/interface as a subtype of a generic class/interface. For example, the **java.lang.String** class is defined to implement the **Comparable** interface in the Java API as follows:

```
public class String implements Comparable<String>
```

Check Point 2

1. Declare generic class for GenericList?
2. Declare generic interface for List?
3. What are generic classes and interfaces?
4. Can a generic class have multiple generic parameters?
5. How to create an instance of ArrayList of strings?

Generic Methods

- Syntax :

```
public static <E> returnType methodName(E parameter)
```

- Example :

```
public static <E> void print(E[] list)
public <E> boolean isFilled(E filled)
```

- To declare generics – place <E> before returnType
- To invoke/call method – place <actualType>methodName
  or as usual method call

LISTING 19.2 GenericMethodDemo.java

```
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) {                    generic method
11     for (int i = 0; i < list.length; i++)
12       System.out.print(list[i] + " ");
13     System.out.println();
14   }
15 }
```

Bounded Generic Type/Bounded Type Parameters

- Bounded generic type is a generic type specified as a **subtype of another type**
- E.g. : **`<E extends GeometricObject>`**

  `E is a generic type of GeometricObject`
- An unbounded generic type <E> is the same as <E extends Object>.

Bounded Generic Type/Bounded Type Parameters

- 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/bounded generic types* 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.

```
public <U extends Number> void inspect(U u)
```

Consider the Following Code

```
public class BoundedGeneric2<T extends Number> {
   T data;
   public BoundedGeneric2(T t) {
      data = t;
   }

   void display(){
      System.out.println("Value is : " + data );
      System.out.println("   and type is " + data.getClass().getName() );
   }

   public static void main(String[] args) {
      BoundedGeneric2<Integer> b1 = new BoundedGeneric2<Integer>(3);
      b1.display();

      BoundedGeneric2<Double> b2 = new BoundedGeneric2<Double>(3.14);
      b2.display();

      //BoundedGeneric2<String> b3 = new BoundedGeneric2<String>("Hello World");
      // This line is commented because compilation error
      // error: type argument String is not within bounds of type-variable T

    }
}
```

```
Value is : 3
   and type is java.lang.Integer
Value is : 3.14
   and type is java.lang.Double
```

Check Point 3

1. Declare a static generic method for *myMethod* that does not return anything, but accepts one parameter called *a*.
2. What is bounded generic type?

Raw Type and Backward Compatibility

- **Raw type** : a generic class/interface used **without specifying a concrete type/without a type parameter**

// 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);
```

Note: **Comparable o1** and **Comparable o2** are raw type declarations. 23 is autoboxed into new Integer(23)

Make it Safe (by using generic type)

```
// Max.java: Find a maximum object
public class Max {
  /** 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);   // a compile error
```

Avoiding Unsafe Raw Types

Use

```
new ArrayList<ConcreteType>();
```

Instead of

```
new ArrayList();
```

Wildcard Generic Types

Why wildcards are necessary? See this example.

```
import java.util.ArrayList;

public class WildCardDemo1
{
   public static void main(String[] args) {
      ArrayList<Integer> list1 = new ArrayList<>();
      list1.add(3);
      list1.add(6);
      list1.add(9);
      display(list1);   // call method
      System.out.println();
   }

   public static void display(ArrayList<Number> list)   {
      for (int i=0; i<=2; i++)
         if ( list.get(i).equals(6.0) )
            System.out.println("yes");
         else
            System.out.println("no");
   }
}
```

* Compile error:

Integer is a subtype of Number.

However, list1, which is an instance of ArrayList<Integer>, is not an instance of ArrayList<Number>

Wildcard Generic Types

Why wildcards are necessary? See this example.

```
import java.util.ArrayList;

public class WildCardDemo2
{
   public static void main(String[] args) {
      ArrayList<Integer> list1 = new ArrayList<>();
      list1.add(3);
      list1.add(6);
      list1.add(9);
      display(list1);   // call method
      System.out.println();

      ArrayList<Double> list2 = new ArrayList<>();
      list2.add(3.0);
      list2.add(6.0);
      list2.add(9.0);
      display(list2);   // call method
      System.out.println();
   }
   public static void display(ArrayList<?> list)   {
      for (int i=0; i<=2; i++)
         if ( list.get(i).equals(6.0) )
            System.out.println("yes");
         else
            System.out.println("no");
   }
}
```

Wildcards (?) represents an “unknown type”.

```
no
no
no

no
yes
no
```

Wildcard Generic Types

3 forms of wildcard generic type:

? **unbounded wildcard** (represents any object type)

? extends T **bounded wildcard** (unknown subtype of T)

? super T **lower bound wildcard** (unknown supertype of T)

- <?> equals <? extends Object>

Erasure and Restrictions on Generics

Generics are implemented using an approach called *type erasure*. The **compiler uses the generic type information to compile the code, but erases it afterwards**. So the generic information is **not available at run time**. This approach enables the generic code to be backward-compatible with the legacy code that uses raw types.

Compile Time Checking

For example, the **compiler checks whether generics is used correctly** for the following code in (a) and translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

```
ArrayList<String> list = new ArrayList<String>();
list.add("Oklahoma");
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

(b)

Compile time checking

When generic classes, interfaces and methods are compiled, the compiler replaces the generic type with the **Object** type.

```
public static <E> void print(E [] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(a)

```
public static void print(Object [] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(b)

Compile time checking

If a generic type is bounded, the compiler replaces it with the bounded type.

```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(a)

```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(b)

Important Facts

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

```
ArrayList<String> list1 = new ArrayList<String>();
ArrayList<Integer> list2 = new ArrayList<Integer>();
System.out.println(list1  instanceof  ArrayList);    //true
System.out.println(list2  instanceof   ArrayList);   //true
```

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

Restrictions on Generics

1. Restriction 1: Cannot Create an Instance of a Generic Type. (i.e., new E()).

2. Restriction 2: Generic Array Creation is Not Allowed. (i.e., new E[100]).

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

4. Restriction 4: Exception Classes Cannot be Generic.

Restriction 1 : Cannot use new E()

E object = new E();

- new E() is executed at runtime, but E not available

Restriction 2 : Cannot use new E[]

E[] elements = new E[capacity]

- Avoid error by having :

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

- But causes unchecked compile warning because compiler not certain that casting will succeed at runtime. For e.g :
  If E is String, new Object[] is an array of Integer objects,
  then ((String[]) new Object[]) will cause a compile error (ClassCastException)

Restriction 3 : Generic type not allowed in Static Context

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

    static {
        E o2;  //Illegal
    }
}
```

Restriction 4: Exception classes cannot be Generic

- Generic class may not extend java.lang.Throwable. So, the following is illegal

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

- Because the type information is not present at runtime

```
try {
   ...
}
catch(MyException<T> ex) {
   ...
}
```

Check Point 5

1. If a program uses ArrayList<String> and ArrayList<Date>, does the JVM load both of them?
2. What is the problem with this code :

   E object = new E();

3. Can you define a generic exception class? Why?

References

1. Chapter 19 Generics, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015
2. https://docs.oracle.com/javase/tutorial/java/generics/types.html