Java Generics

Parametric Polymorphism

ERASURE AND RESTRICTION ON GENERICS (IMPLEMENTATION ISSUES)

DR. ERIC CHOU IEEE SENIOR MEMBER

Erasure and Restrictions on Generics

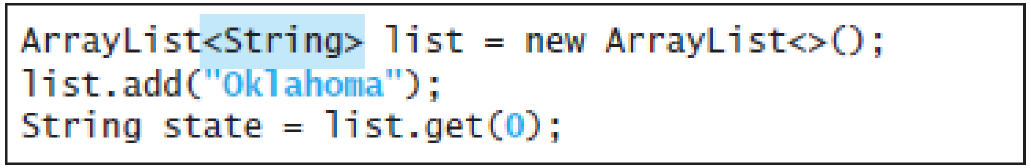
The information on generics is used by the compiler but is not available at runtime. This is called type erasure.

Generics are implemented using an approach called type erasure: The compiler uses the generic type information to compile the code, but erases it afterward. Thus, the generic information is not available at runtime. This approach enables the generic code to be backward compatible with the legacy code that uses raw types.

The generics are present at compile time. Once the compiler confirms that a generic type is used safely, it converts the generic type to a raw type.

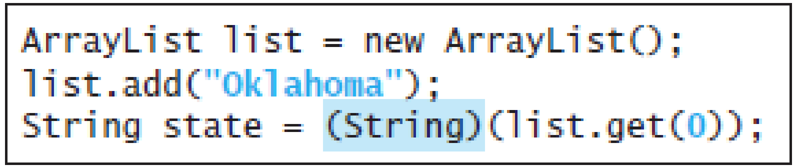
Erasure and Restrictions on Generics

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



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

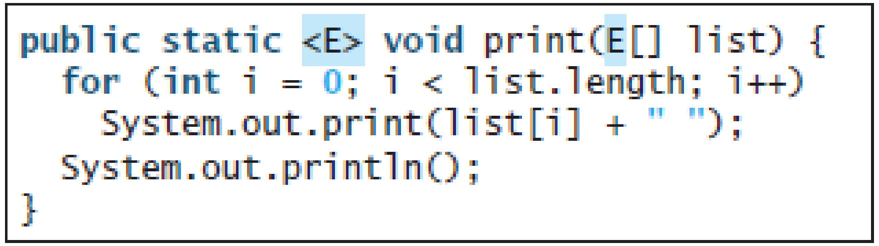
(a)



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

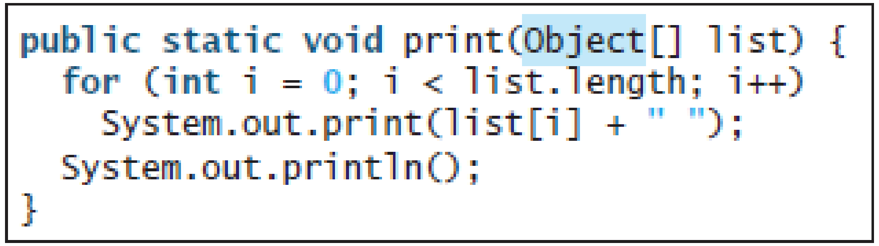
(b)

When generic classes, interfaces, and methods are compiled, the compiler replaces the generic type with the **Object** type. For example, the compiler would convert the following method in (a) into (b).



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

(a)

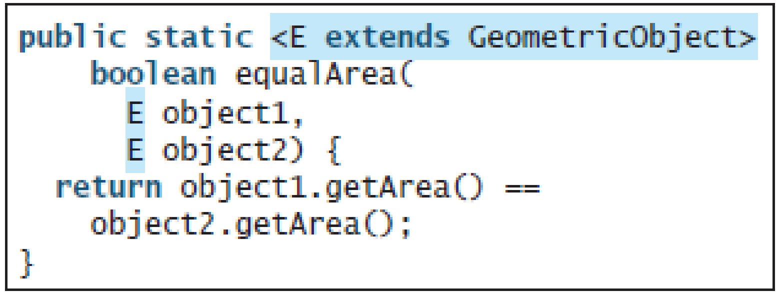


```
public static void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(b)

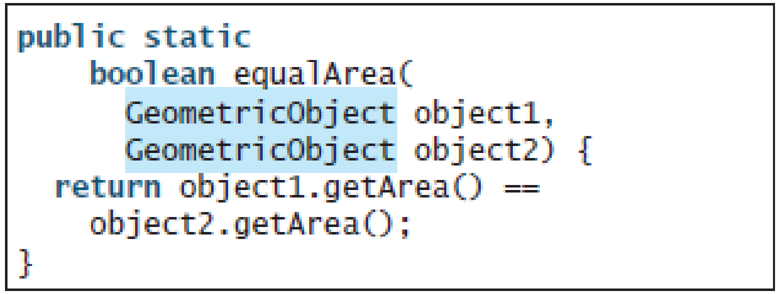
Erasure and Restrictions on Generics

If a generic type is bounded, the compiler replaces it with the bounded type. For example, the compiler would convert the following method in (a) into (b).



```
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)

It is important to note that a generic class is shared by all its instances regardless of its actual concrete type. Suppose list1 and list2 are created as follows:

```
ArrayList<String>   list1 = new ArrayList<>();
ArrayList<Integer> list2 = new ArrayList<>();
```

Erasure and Restrictions on Generics

Although **ArrayList<String>** and **ArrayList<Integer>** are two types at compile time, only one **ArrayList** class is loaded into the JVM at runtime. **list1** and **list2** are both instances of **ArrayList**, so the following statements display **true**:

```
System.out.println(list1 instanceof ArrayList);
System.out.println(list2 instanceof ArrayList);
```

However, the expression **list1 instanceof ArrayList<String>** is wrong. Since **ArrayList<String>** is not stored as a separate class in the JVM, using it at runtime makes no sense.

Erasure and Restrictions on Generics

Because generic types are erased at runtime, there are certain restrictions on how generic types can be used. Here are some of the restrictions:

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.

Erasure and 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];
```

However, casting to **(E[])** causes an unchecked compile warning. The warning occurs because the compiler is not certain that casting will succeed at runtime. For example, if **E** is **String** and **new Object[]** is an array of **Integer** objects, **(String[])(new Object[])** will cause a **ClassCastException**. This type of compile warning is a limitation of Java generics and is unavoidable.

Erasure and Restrictions on Generics

Restriction 2: (cont’d)

Generic array creation using a generic class is not allowed, either. For example, the following code is wrong:

```
ArrayList<String>[] list = new ArrayList<String>[10];
```

You can use the following code to circumvent this restriction:

```
ArrayList<String>[] list = (ArrayList<String>[]) new ArrayList[10];
```

However, you will still get a compile warning.

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

Erasure and 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 {
}
```

Why? If it were allowed, you would have a catch clause for MyException<T> as follows:

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

The JVM has to check the exception thrown from the try clause to see if it matches the type specified in a catch clause. This is impossible, because the type information is not present at runtime.