Java Generics

Parametric Polymorphism

GENERICS BASICS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Type Variable

<T> in angle brackets

- **<T>** represents a *formal generic type*, which can be replaced later with an *actual concrete type*. Replacing a generic type is called a *generic instantiation*.

- By convention, a single capital letter such as **E** or **T** is used to denote a formal generic type. (**E**ntity and **T**ype)

- To see the benefits of using generics, let us examine the code in Figure B. The statement in Figure B(a) declares that **c** is a reference variable whose type is **Comparable** and invokes the **compareTo** method to compare a **Date** object with a string. The code compiles fine, but it has a runtime error because a string cannot be compared with a date.

Type Variable

<T> in angle brackets

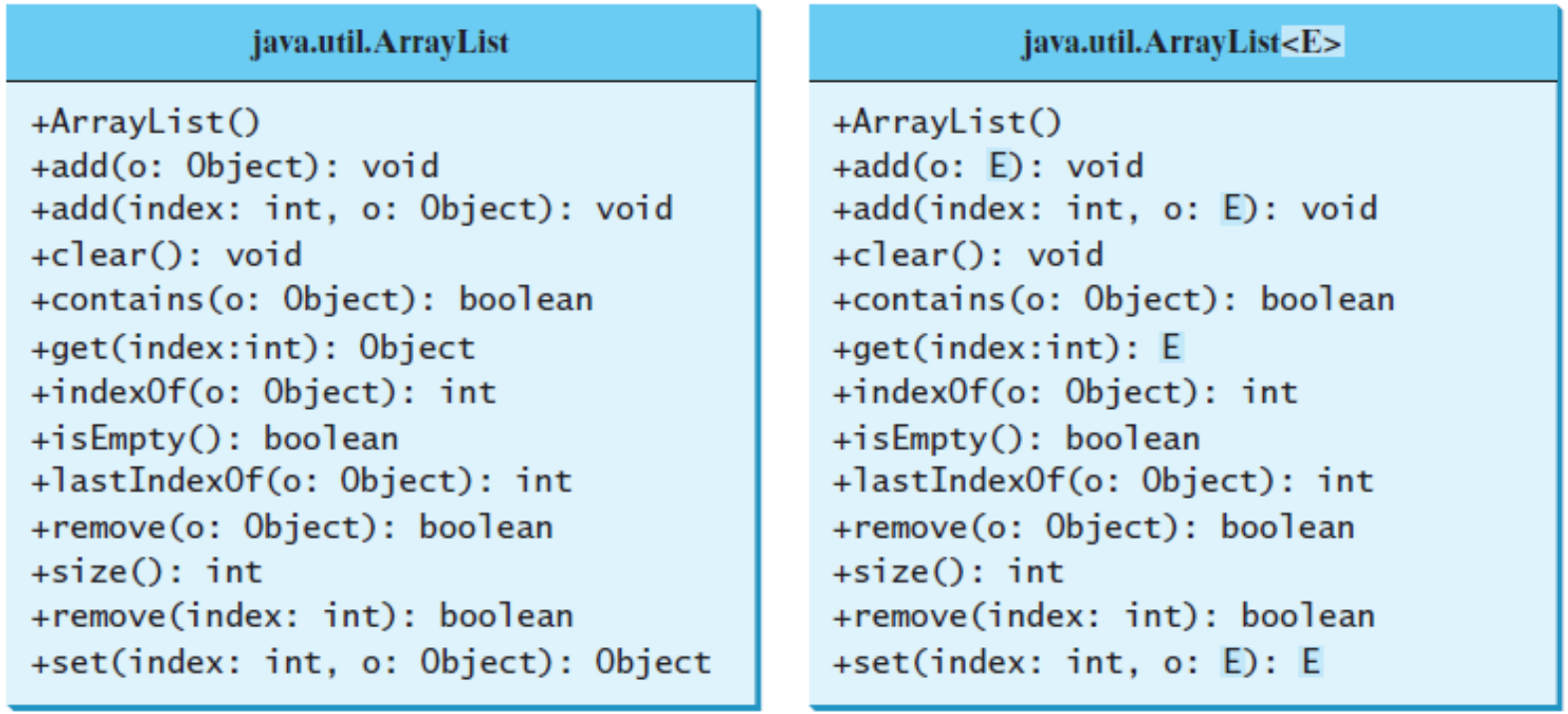
- The statement in Figure B(b) declares that **c** is a reference variable whose type is **Comparable<Date>** and invokes the **compareTo** method to compare a **Date** object with a string.

- This code generates a compile error, because the argument passed to the **compareTo** method must be of the **Date** type. Since the errors can be detected at compile time rather than at runtime, the generic type makes the program more reliable. (The compareTo() can be overrided.)

- The **ArrayList** Class. This class has been a generic class since JDK 1.5.

Type Variable

<T> in angle brackets



(a) ArrayList before JDK 1.5

(b) ArrayList since JDK 1.5

Figure C.

ArrayList as an Example for Generic Container

Declaration of the Pointer(Reference):

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

Addition of Element (body):

```
alist.add(new String(1));
```

Generic Container only for Reference Type:

```
ArrayList<int> alist = new ArrayList<int>();
```

The primitive type is not allowed here.

Casting is not needed to retrieve a value from a list with a specified element type, because the compiler already knows the element type. For example, the following statements create a list that contains strings, add strings to the list, and retrieve strings from the list.

```
ArrayList<String> alist = new ArrayList<>();
alist.add("Red");
alist.add("White");
String s = list.get(o); // No casting needed.
```

Defining Generic Classes and Interfaces

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.

This example creates a stack to hold integers and adds three integers to the stack.

```
GenericStack<Integer> stack2 = new GenericStack<>();
stack2.push(1);  // autoboxing
stack2.push(2);
stack2.push(3);
```

Instead of using a generic type, you could simply make the type element Object, which can accommodate any object type. However, using generic types can improve software reliability and readability, because certain errors can be detected at compile time rather than at runtime. For example, because stack1 is declared GenericStrck<String>, only strings can be added to the stack. It would be a compile error if you attempted to add an integer to stack1.

Note:

- Multiple type variables for a generic class definition. For example, **<E1, E2, E3>**
- To create a stack of strings, you can **new GenericStack<String>()** or **new GenericStack()**. This could mislead you into thinking that the constructor of GenericStack should be defined as

  public GenericStack<E>()

  This is wrong. It should be defined as

  public GenericStack()

Generic Stack

Demo Program: GenericStack.java + TestGenericStack.java

Go BlueJ!