Chapter 21 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 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.

Why Generics?

- 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, the compile error occurs.

Generic Type

```
package java.lang;

public interface Comaprable {
  public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comaprable<T> {
  public int compareTo(T o)
}
```

(b) JDK 1.5

Generic Instantiation

Runtime error

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Improves reliability

Compile error

Generic ArrayList in JDK 1.5

java.util.ArrayList
+ArrayList()
+add(o: Object) : void
+add(index: int, o: Object) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : Object
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: Object) : Object

(a) ArrayList before JDK 1.5

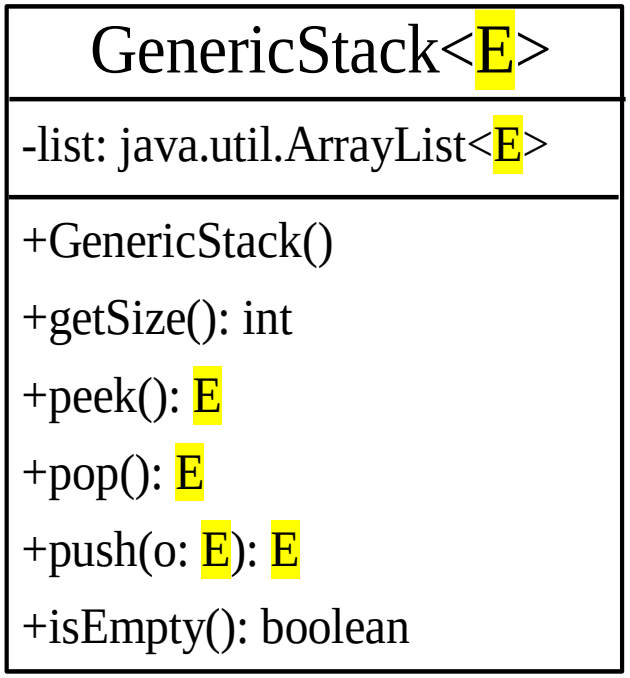
java.util.ArrayList<E>
+ArrayList()
+add(o: E) : void
+add(index: int, o: E) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : E
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: E) : E

(b) ArrayList in JDK 1.5

No Casting Needed

```
ArrayList<Double> list = new ArrayList<Double>();
list.add(5.5); // 5.5 is automatically converted to new Double(5.5)
list.add(3.0); // 3.0 is automatically converted to new Double(3.0)
Double doubleObject = list.get(0); // No casting is needed
double d = list.get(1); // Automatically converted to double
```

Declaring Generic Classes and Interfaces



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.

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

Use of Generic Classes

```
GenericStack<String> stack1 = new GenericStack<>();
stack1.push("London");
stack1.push("Paris");
stack1.push("Berlin");
```

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

Generic Methods

- Methods inside a generic class can make use of a class' type parameter and are, therefore, automatically generic relative to the type parameter.
- However, it is possible to declare a generic method that uses one or more type parameters of its own.
- Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

Generic Method Example

```
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 }
```

generic method

or simply invoke it as follows:

```
print(integers);
print(strings);
```

Generic Methods

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

Equivalent to

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

Generic Method Example

```
public class GenericMethodDemo {
   static <T extends Comparable<T>, V extends T> boolean isInList(T a, V[] b)    {
      for (T x:b) {
         if (a.equals(x)) return true;
      }
      return false;
   }
   public static void main(String[] args) {
      Integer[] list = {3,4,5,6,7};
      if (isInList(5, list)) System.out.println("Found");
      else System.out.println("Not Found");
   }
}
```

Bounded Generic Type

```
public static void main(String[] args ) {
    Rectangle rectangle = new Rectangle(2, 2);
    Circle9 circle = new Circle9(2);
    System.out.println("Same area? " + equalArea(rectangle, circle));
}

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

<E extends superclass>

This specifies that E can only be replaced by superclass, or subclasses of superclass. Thus, superclass defines an inclusive, upper limit.

Wildcard Argument and Bounded Wildcard Argument

Raw Type and Backward Compatibility

```
// 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);
```

Compile Time Checking

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. 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)

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.

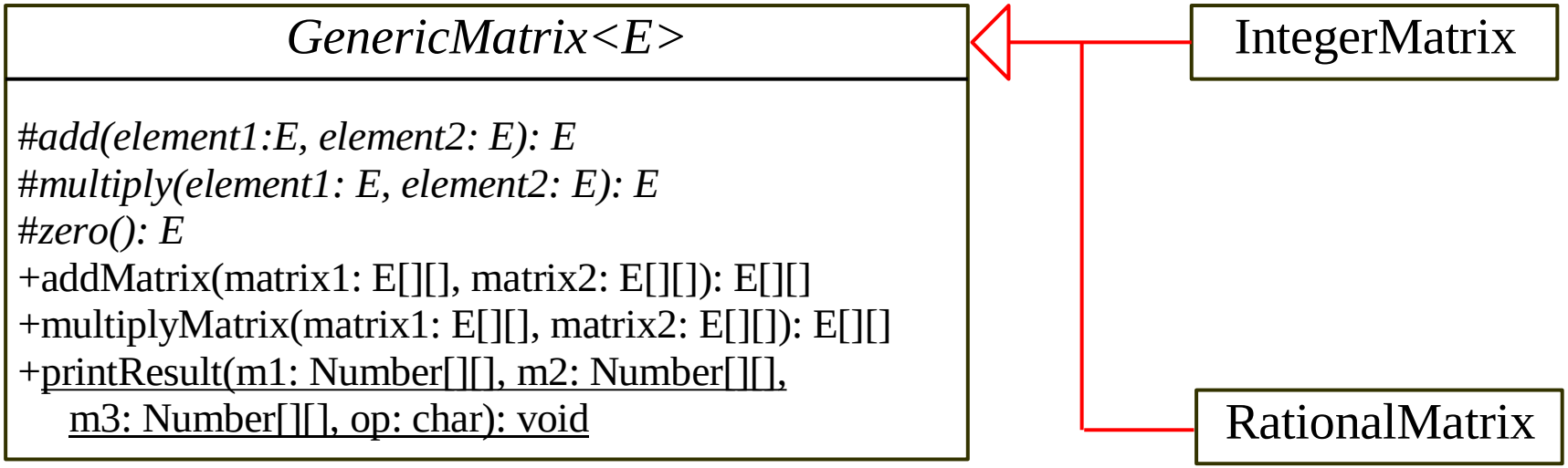
Restrictions on Generics

- Restriction 1: Cannot Create an Instance of a Generic Type. (i.e., new E()).
- Restriction 2: Generic Array Creation is Not Allowed. (i.e., new E[100]).
- 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.
- Restriction 4: Exception Classes Cannot be Generic. See book.

Static Methods with Own Type

- Although you can't declare static members that use a type parameter declared by the enclosing class, you can declare static generic methods, which define their own type parameters, as was shown earlier in these slides.

UML Diagram


GenericMatrix<E>
#add(element1:E, element2: E): E
#multiply(element1: E, element2: E): E
#zero(): E
+addMatrix(matrix1: E[][], matrix2: E[][]): E[][]
+multiplyMatrix(matrix1: E[][], matrix2: E[][]): E[][]
+printResult(m1: Number[][], m2: Number[][],
m3: Number[][], op: char): void
IntegerMatrix
RationalMatrix