# Chapter 19 Generics

# Objectives

- To know the benefits of generics (§19.1).
- To use generic classes and interfaces (§19.2).
- To declare generic classes and interfaces (§19.3).
- To understand why generic types can improve reliability and readability (§19.3).
- To declare and use generic methods and bounded generic types (§19.4).
- To use raw types for backward compatibility (§19.5).
- To know wildcard types and understand why they are necessary (§19.6).
- To convert legacy code using JDK 1.5 generics (§19.7).
- To understand that generic type information is erased by the compiler and all instances of a generic class share the same runtime class file (§19.8).
- To know certain restrictions on generic types caused by type erasure (§19.8).
- To design and implement generic matrix classes (§19.9).

## Why Do You Get a Warning?

```
public class ShowUncheckedWarning {
  public static void main(String[] args) {
    java.util.ArrayList list =
      new java.util.ArrayList();
    list.add("Java Programming");
  }
}
```

To understand the compile warning on this line, you need to learn JDK 1.6 generics.

# Fix the Warning

```
public class ShowUncheckedWarning {
  public static void main(String[] args) {
    java.util.ArrayList<String> list =
      new java.util.ArrayList<String>();
    list.add("Java Programming");
  }
}
```

No compile warning on this line.

## 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?

- One of 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.

## Generic Type

Here, <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.

```
package java.lang;

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

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<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

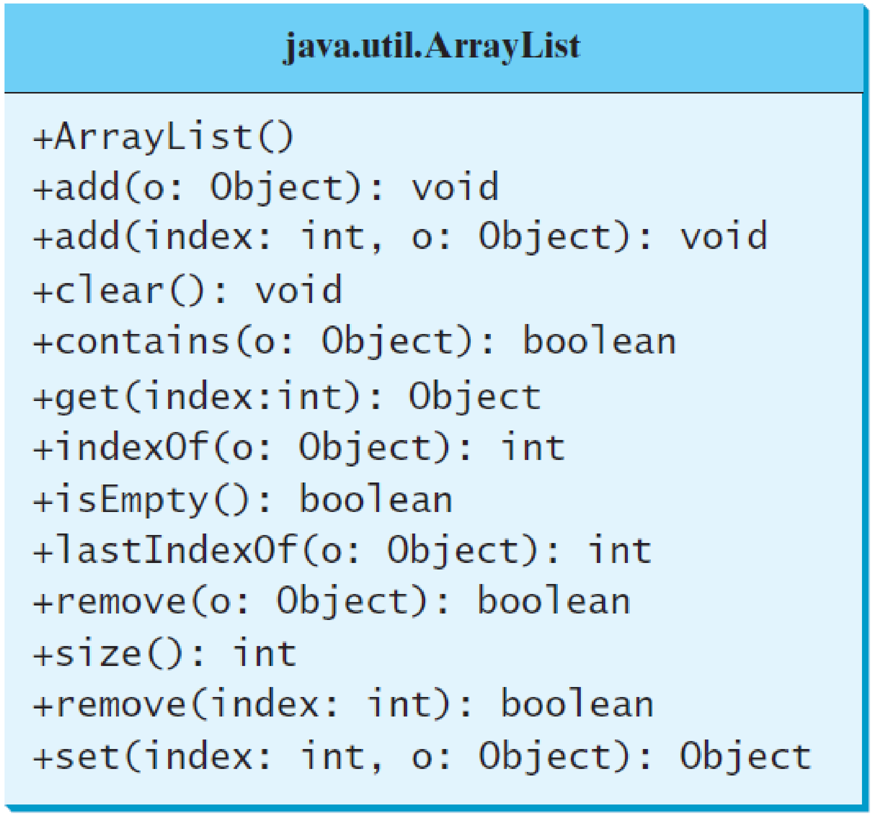
```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Compile error

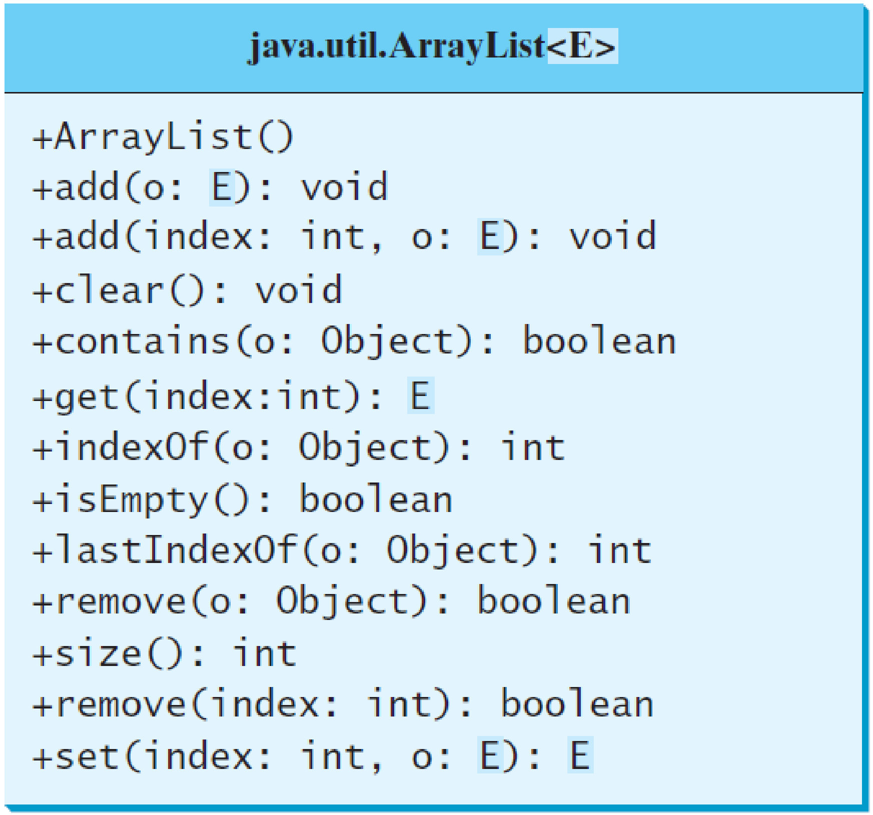
Improves reliability

# 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 since JDK 1.5

**ArrayList** was introduced in Section 11.11, The **ArrayList** Class. This class has been a generic class since JDK 1.5. Figure 19.3 shows the class diagram for **ArrayList** before and since JDK 1.5, respectively.

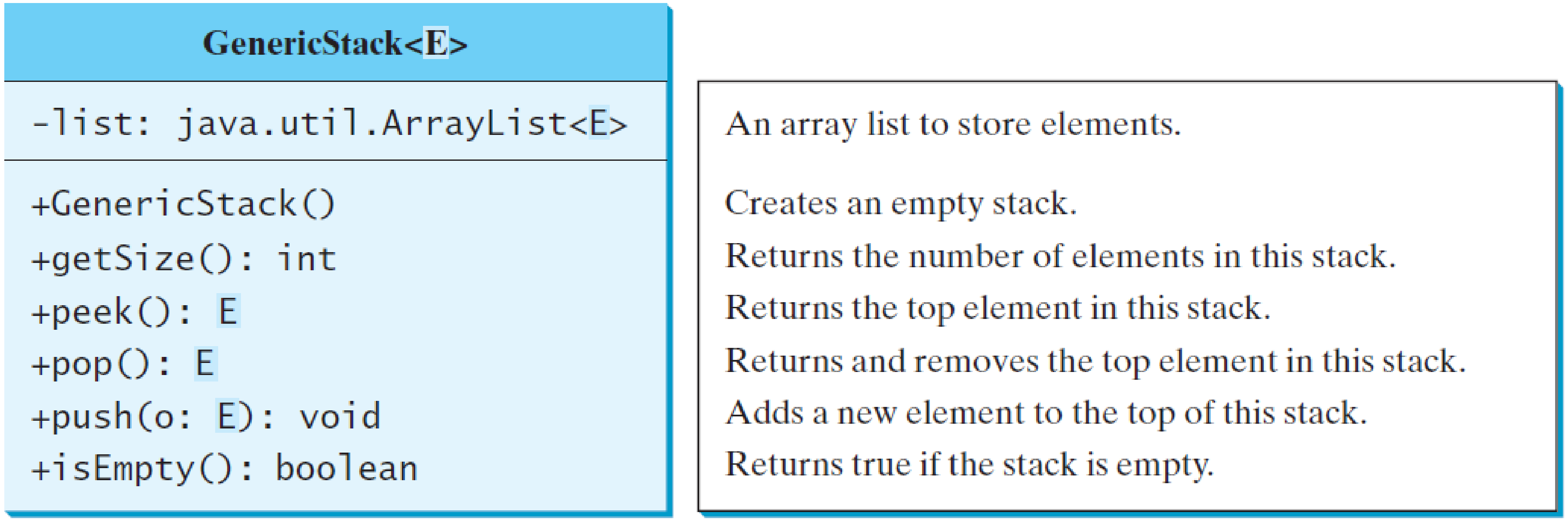
## No Casting Needed

- Casting is not needed to retrieve a value from a list with a specified element type because the compiler already knows the element type.
- If the elements are of wrapper types, such as **Integer**, **Double**, and **Character**, you can directly assign an element to a primitive-type variable. This is called *autounboxing*, as introduced in Section 10.8. For example, see the following code:

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

- *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.*



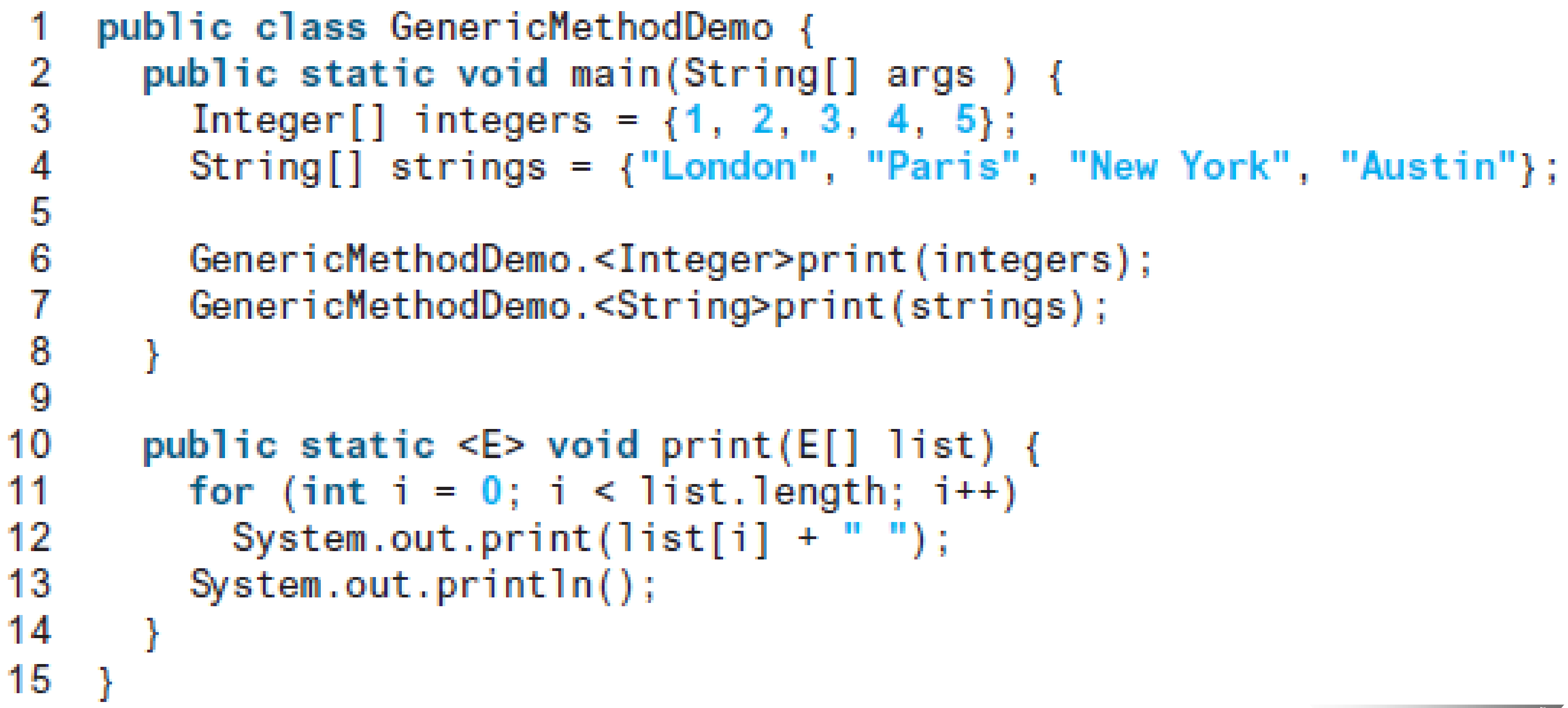
**GenericStack<E>**

| | |
|---|---|
| `-list: java.util.ArrayList<E>` | An array list to store elements. |
| `+GenericStack()` | Creates an empty stack. |
| `+getSize(): int` | Returns the number of elements in this stack. |
| `+peek(): E` | Returns the top element in this stack. |
| `+pop(): E` | Returns and removes the top element in this stack. |
| `+push(o: E): void` | Adds a new element to the top of this stack. |
| `+isEmpty(): boolean` | Returns true if the stack is empty. |

GenericStack

# Generic Methods

- You can also use generic types to define generic methods.
- To declare a generic method, you place the generic type **<E>** immediately after the keyword **static** in the method header.



```java
public class GenericMethodDemo {
  public static void main(String[] args ) {
    Integer[] integers = {1, 2, 3, 4, 5};
    String[] strings = {"London", "Paris", "New York", "Austin"};

    GenericMethodDemo.<Integer>print(integers);
    GenericMethodDemo.<String>print(strings);
  }

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

# Bounded Generic Type

- A generic type can be specified as a subtype of another type. Such a generic type is called *bounded*.
- Thebounded generic type **<E extends GeometricObject>** specifies that **E** is a generic subtype of **GeometricObject**.

```
public static void main(String[] args ) {
    Rectangle rectangle = new Rectangle(2, 2);
    Circle circle = new Circle (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();
}
```

# 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.*

```
// For example, the following statement creates a list for strings.
// You can now add only strings into the list.

ArrayList<String> list = new ArrayList<>();

// Raw type : You can use a generic class without specifying a concrete 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;
  }
}
```

**Comparable o1** and **Comparable o2** are raw type declarations. Be careful: *raw types are unsafe*. For example, you might invoke the **max** method using

```
Max.max("Welcome", 23); // 23 is autoboxed into new Integer(23)
```

This would cause a runtime error because you cannot compare a string with an integer object.

## 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;
  }
```

- If you invoke the **max** method using

    MaxUsingGenericType.max(**"Welcome"**, **23**); // 23 is autoboxed into new Integer(23)

- A compile error will be displayed because the two arguments of the **max** method in **MaxUsingGenericType** must have the same type (e.g., two strings or two integer objects).
- Furthermore, the type **E** must be a subtype of **Comparable<E>**.

## Avoiding Unsafe Raw Types

Use

new ArrayList<ConcreteType>()

Instead of

new ArrayList();

TestArrayListNew Run

## Java Generics Wildcards

**Methods with wildcards:**

- public static void print(GenericStack<?> stack)
- public static <T> void add(GenericStack<? extends T> stack1, GenericStack<T> stack2)
- public static <T> void add(GenericStack<T> stack1, GenericStack<? super T> stack2)

- Question mark (?) is the wildcard in generics and represent an unknown type. The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type.
- The first form, ?, called an unbounded wildcard, is the same as ? extends Object. We can provide GenericStack <String> or GenericStack <Integer> or any other type of Object list argument to the print method
- 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.

# Wildcards

Why wildcards are necessary? See this example.

WildCardNeedDemo

public static double max(GenericStack<Number> stack)

- The main method creates a stack of integer objects, adds three integers to the stack, and invokes the max method to find the maximum number in the stack.
- The program in Listing 19.7 has a compile error in line 8 because intStack is not an instance of GenericStack<Number>. Thus, you cannot invoke max(intStack). The fact is Integer is a subtype of Number, but GenericStack<Integer> is not a subtype of GenericStack<Number>.
- To circumvent this problem, use wildcard generic types.
- You can fix the error by replacing line 12 as of one of follows:

public static void print(GenericStack<?> stack)

public static double max(GenericStack<? extends Number> stack)

AnyWildCardDemo

# Wildcards

| | |
|---|---|
| ? | unbounded wildcard |
| ? extends T | bounded wildcard |
| ? super T | lower bound wildcard |

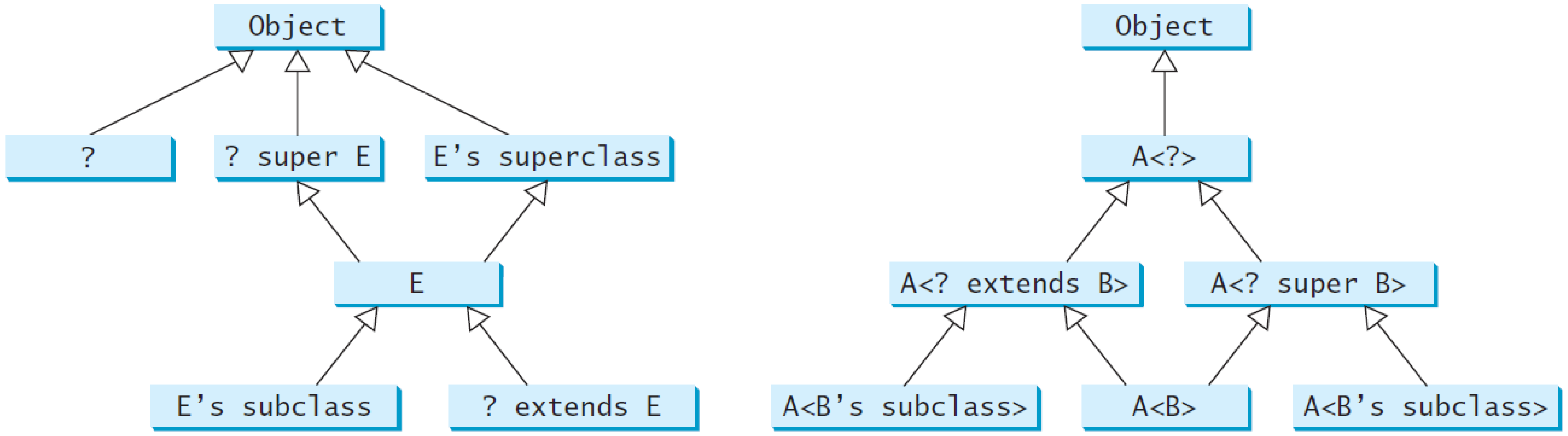
Lower bound wildcard example:

```
public static <T> void add(GenericStack<T> stack1, GenericStack<? super T> stack2)
```

SuperWildCardDemo

# Generic Types and Wildcard Types

- The inheritance relationship involving generic types and wildcard types is summarized in Figure 19.6.
- In this figure, **A** and **B** represent classes or interfaces, and **E** is a generic-type parameter.



## 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.
- 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.

## 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<>();
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<>();
GenericStack<Integer> stack2 = new GenericStack<>();
```

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., E object = new E()).

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

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

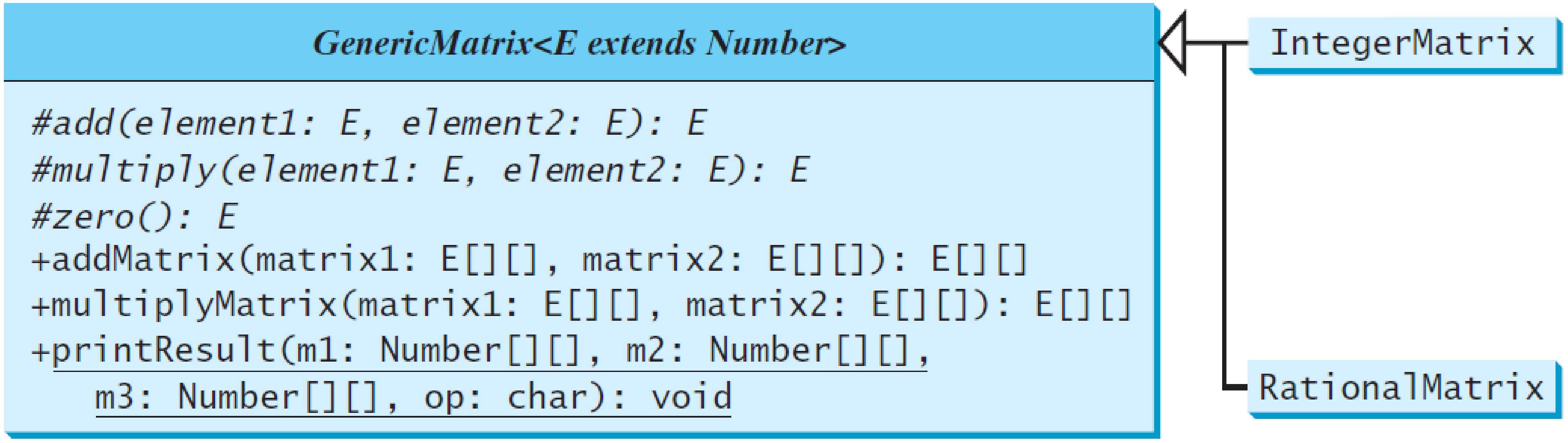
- Restriction 4: Exception Classes Cannot be Generic.

## Designing Generic Matrix Classes

Objective: This example gives a generic class for matrix arithmetic. This class implements matrix addition and multiplication common for all types of matrices.

GenericMatrix

# UML Diagram


*GenericMatrix<E extends Number>*
*#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

# Source Code

Objective: This example gives two programs that utilize the GenericMatrix class for integer matrix arithmetic and rational matrix arithmetic.

IntegerMatrix TestIntegerMatrix Run

RationalMatrix TestRationalMatrix Run