Generics

What is Generics?

- ***Generics*** let you parameterize types. With this capability, you can define a class or a method with generic types that the compiler can replace with concrete types.
- For example, Java defines a generic ArrayList class for storing the elements of a generic type. From this generic class, you can create an ArrayList object for holding strings, and an ArrayList object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

```
ArrayList<T>

ArrayList<String> mylist = new ArrayList<String>();
ArrayList<Double> mylist = new ArrayList<Double>();
ArrayList<String> mylist = new ArrayList<String>();
```

Why Generics?

Programs that uses Generics has got many benefits over non-generic code.

- **Code Reuse**: We can write a method/class/interface once and use for any type we want.
- **Type Safety** : Generics make errors to appear compile time than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

Generic Type

```java
package java.lang;

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

(a) Prior to JDK 1.5

```java
package java.lang;

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

(b) JDK 1.5

Runtime Error

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

(a) Prior to JDK 1.5

Compile Error

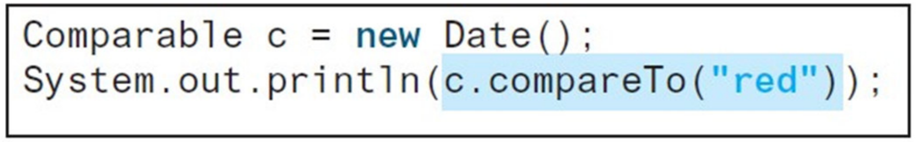
```java
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Generic Instantiation

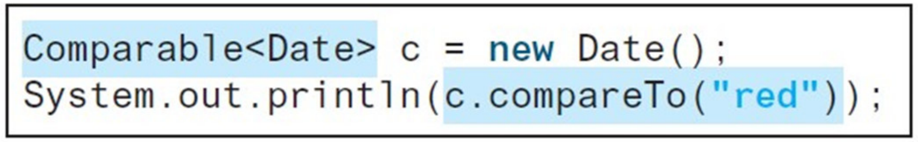
Generic Type

- 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.



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

(a) Prior to JDK 1.5

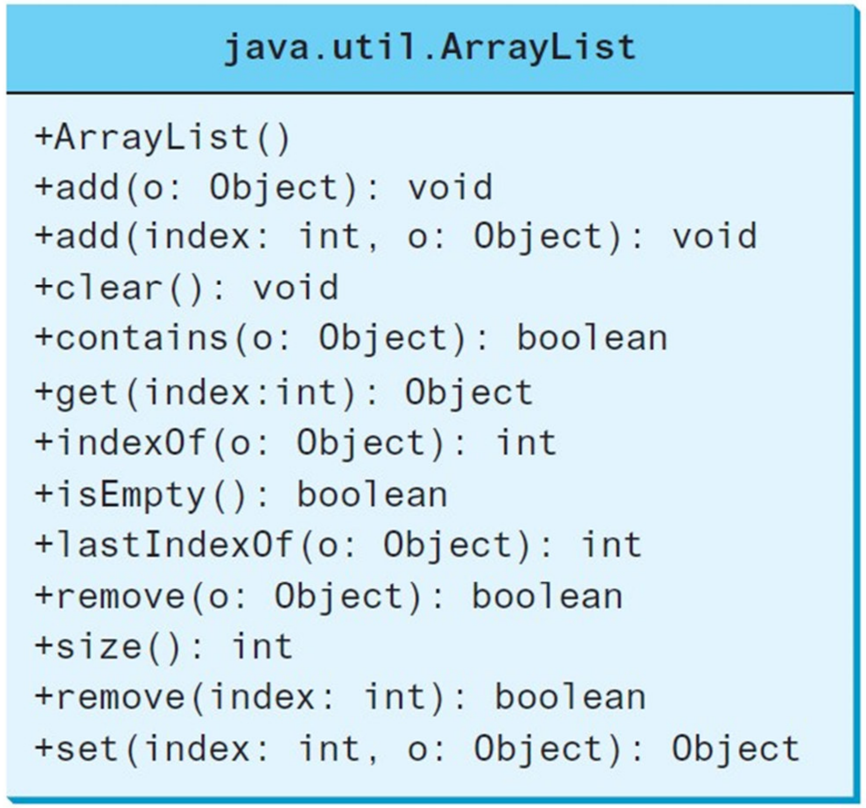


```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

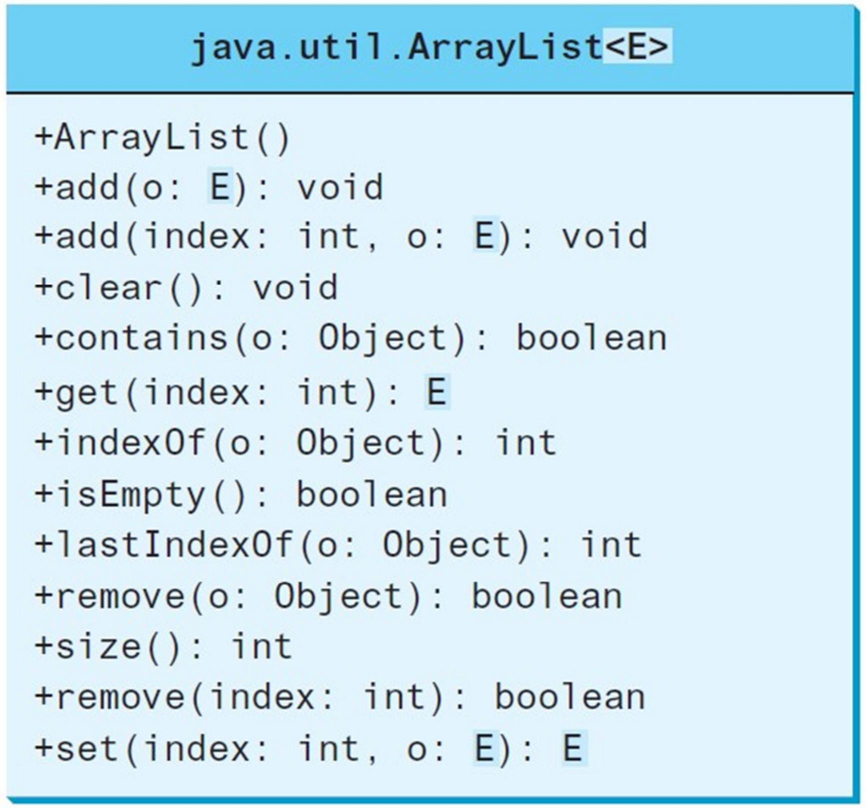
Generic Instantiation

Generic ArrayList in JDK1.5



```
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

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. For example, the following statements create a list that contains strings, add strings to the list, and retrieve strings from the list.

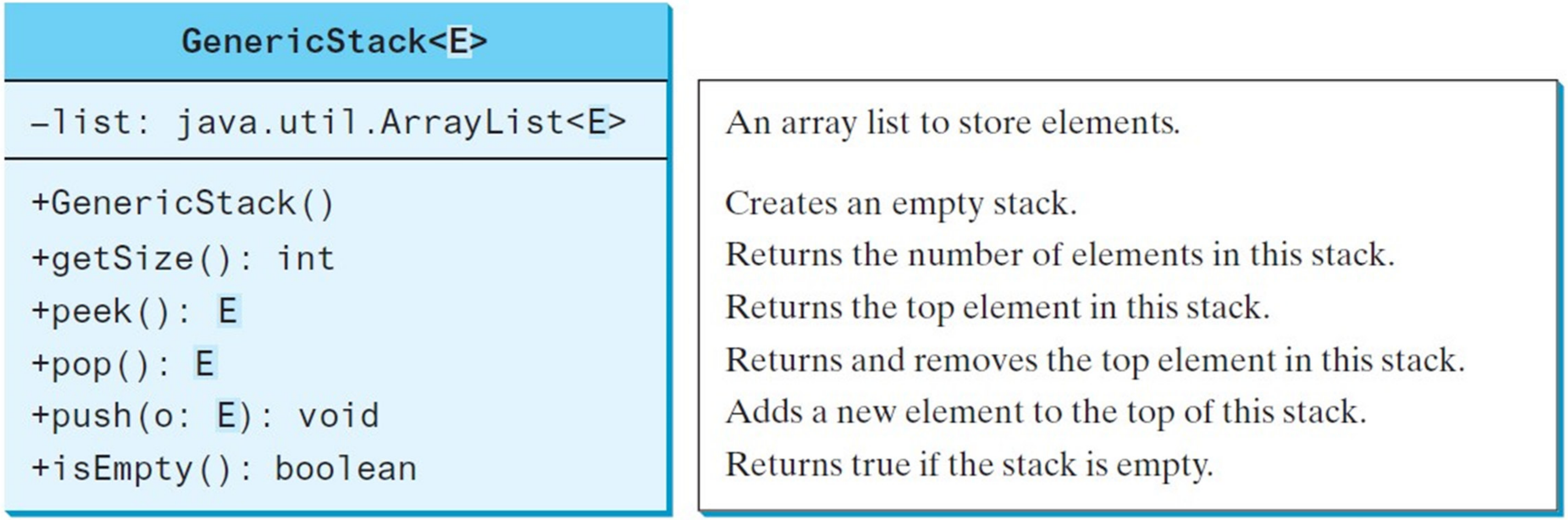
```java
ArrayList<String> list = new ArrayList<>();
list.add("Red");
list.add("White");
String s = list.get(0); // No casting is needed
```

Auto-boxing/unboxing

- 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 *auto unboxing*. For example, see the following code:

```java
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
```

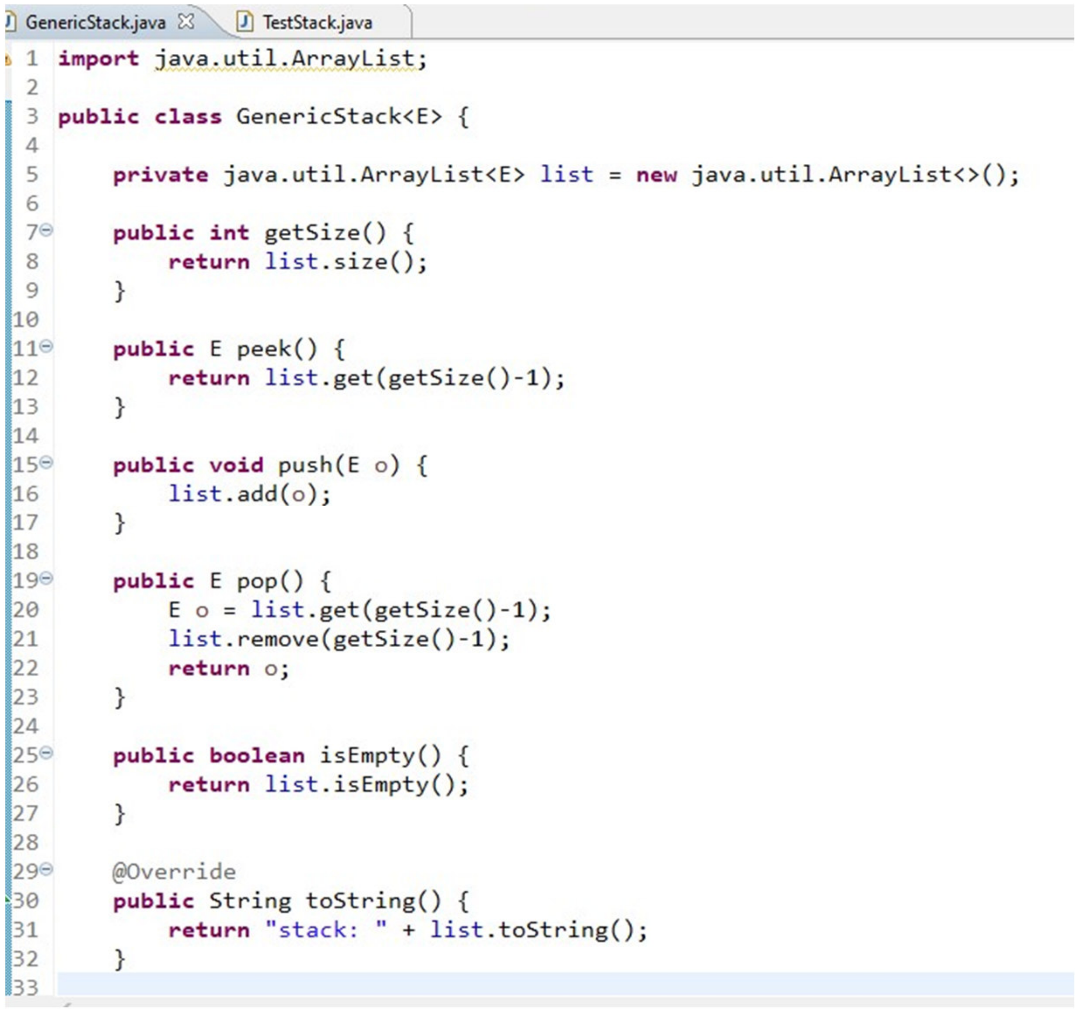
Defining Generic Classes



GenericStack<E>

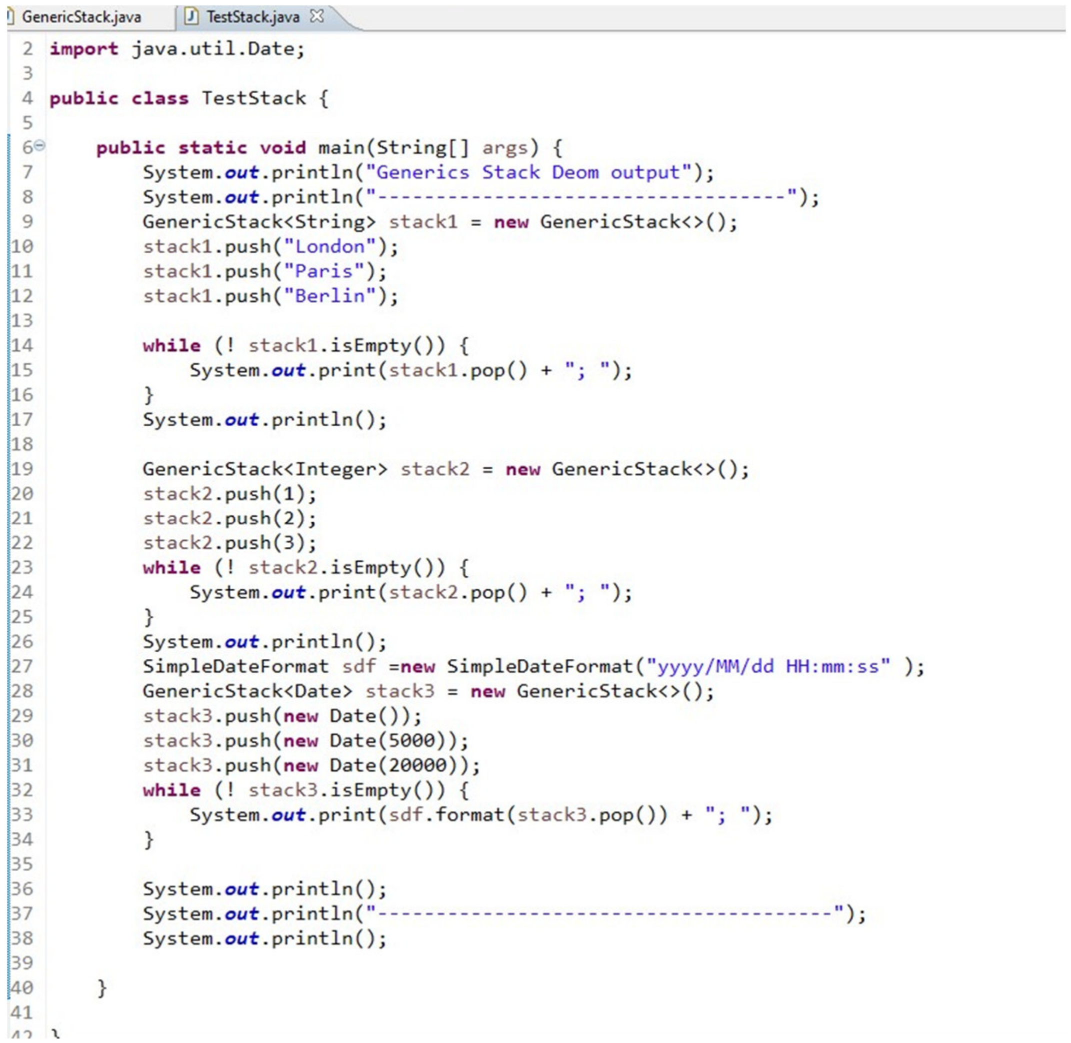
`-list: java.util.ArrayList<E>`	An array list to store elements.
`+GenericStack()`	Creates an empty stack.
`+getSize(): int`	Returns the number of elements in this stack.
`+peek(): E`	Returns the top element in this stack.
`+pop(): E`	Returns and removes the top element in this stack.
`+push(o: E): void`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

Defining Generic Classes



```java
import java.util.ArrayList;

public class GenericStack<E> {

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

    public int getSize() {
        return list.size();
    }

    public E peek() {
        return list.get(getSize()-1);
    }

    public void push(E o) {
        list.add(o);
    }

    public E pop() {
        E o = list.get(getSize()-1);
        list.remove(getSize()-1);
        return o;
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    @Override
    public String toString() {
        return "stack: " + list.toString();
    }

```

Defining Generic Classes



GenericStack.java | TestStack.java

```
import java.util.Date;

public class TestStack {

    public static void main(String[] args) {
        System.out.println("Generics Stack Deom output");
        System.out.println("-----------------------------------");
        GenericStack<String> stack1 = new GenericStack<>();
        stack1.push("London");
        stack1.push("Paris");
        stack1.push("Berlin");

        while (! stack1.isEmpty()) {
            System.out.print(stack1.pop() + "; ");
        }
        System.out.println();

        GenericStack<Integer> stack2 = new GenericStack<>();
        stack2.push(1);
        stack2.push(2);
        stack2.push(3);
        while (! stack2.isEmpty()) {
            System.out.print(stack2.pop() + "; ");
        }
        System.out.println();
        SimpleDateFormat sdf =new SimpleDateFormat("yyyy/MM/dd HH:mm:ss" );
        GenericStack<Date> stack3 = new GenericStack<>();
        stack3.push(new Date());
        stack3.push(new Date(5000));
        stack3.push(new Date(20000));
        while (! stack3.isEmpty()) {
            System.out.print(sdf.format(stack3.pop()) + "; ");
        }

        System.out.println();
        System.out.println("---------------------------------------");
        System.out.println();

    }

}
```

RUN

Defining Generic Classes

- To create a stack of strings, you use `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()
```

- Occasionally, a generic class may have more than one parameter. In this case, place the parameters together inside the brackets, separated by commas—for example,

```
<E1, E2, E3>
```

Generic Methods

- You can define generic interfaces (e.g., the **Comparable** interface) and classes(e.g., the **GenericStack** class). You can also use generic types to define generic methods.

```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.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();
    }
}
```

RUN

Generic Methods

- To declare a generic method, you place the generic type `<E>` before the return type in the method header.

```
public static <E> void print(E[] list);
```

- To invoke a generic method, prefix the method name with the actual type in angle brackets. For example,

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

- or simply invoke it as follows:

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

- In the latter case, the actual type is not explicitly specified. The compiler automatically discovers the actual type.

Bounded Generic Type

- A generic type can be specified as a subtype of another type. Such a generic type is called ***bounded***.

```java
public class BoundedTypeDemo {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        Circle circle = new Circle();

        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 Types and Backward Compatibility

- You can use a generic class without specifying a concrete type such as the following:

```
GenericStack stack = new GenericStack(); // raw type
```

- This is roughly equivalent to

```
GenericStack<Object> stack = new GenericStack<Object>();
```

- A generic class such as `GenericStack` and `ArrayList` used without a type parameter is called a raw type. Using raw types allows for backward compatibility with earlier versions of Java.

Raw Type is Unsafe

```java
public class Max {
    /** Return the maximum of two objects */
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

```java
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

```
public class MaxUsingGenericType {
    /** Return the maximum of two objects */
    public static <E extends Comparable<E>> E max(E o1, E o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

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

a ***compile error*** will be displayed because the two arguments of the **max** method in **MaxUsingGenericType** must have the same type

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> stack = new GenericStack<String>();
GenericStack<Integer> stack = 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

- You cannot create an instance using a generic-type parameter. For example, the following statemen is wrong:

```
E object = new E();
```

- You cannot create an array using a generic type parameter. For example, the following statement is wrong:

```
E[] elements = new E[capacity];
```

- A Generic Type Parameter of a Class Is Not Allowed in a Static Context

```java
public class Test<E> {

    public static void m(E o1) { // Illegal
    }

    public static E o1; // Illegal
}
```

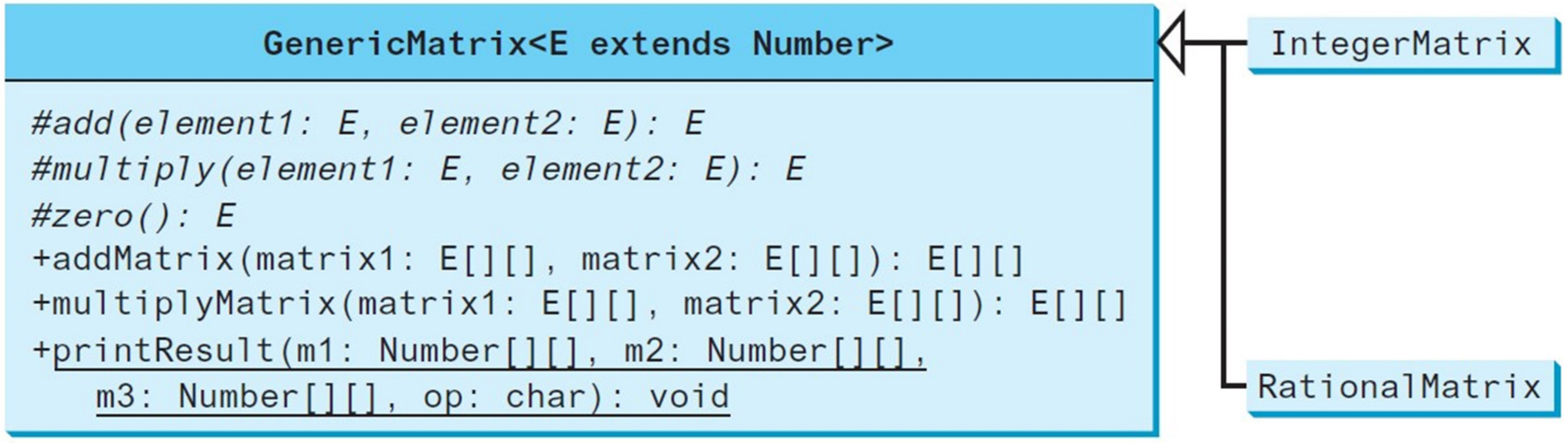
Restrictions on Generics

- 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 {
}
```

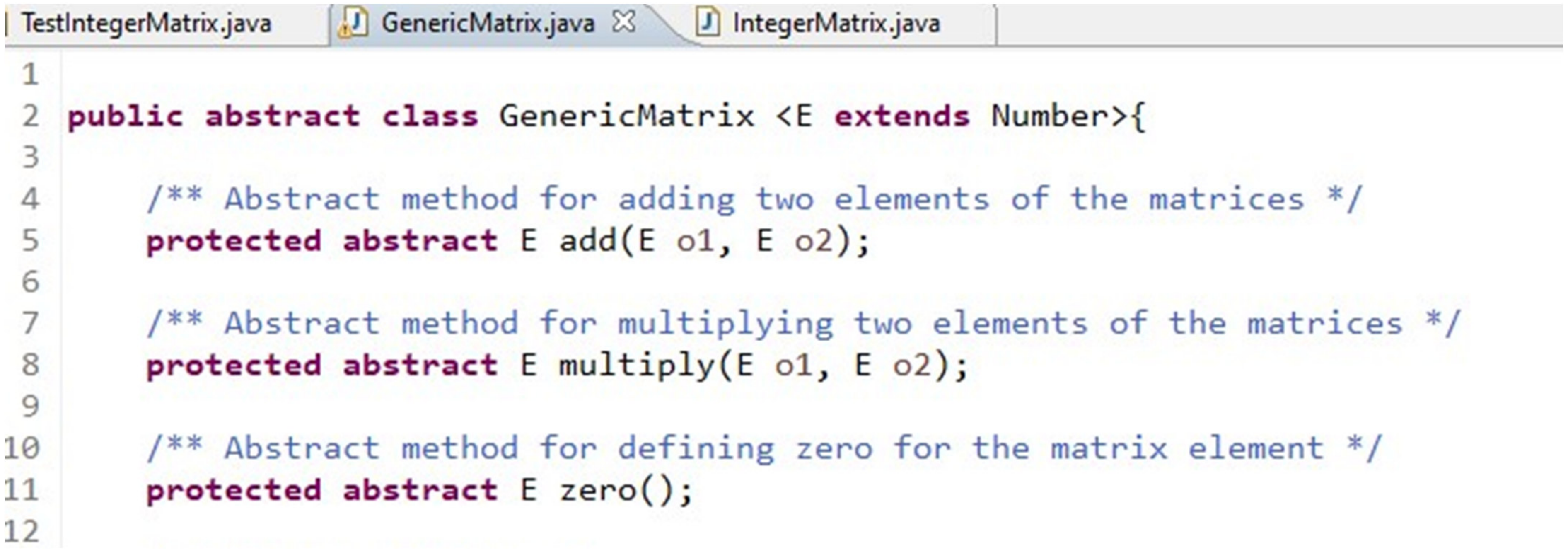
Designing Generic Matrix Classes

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


GenericMatrix<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

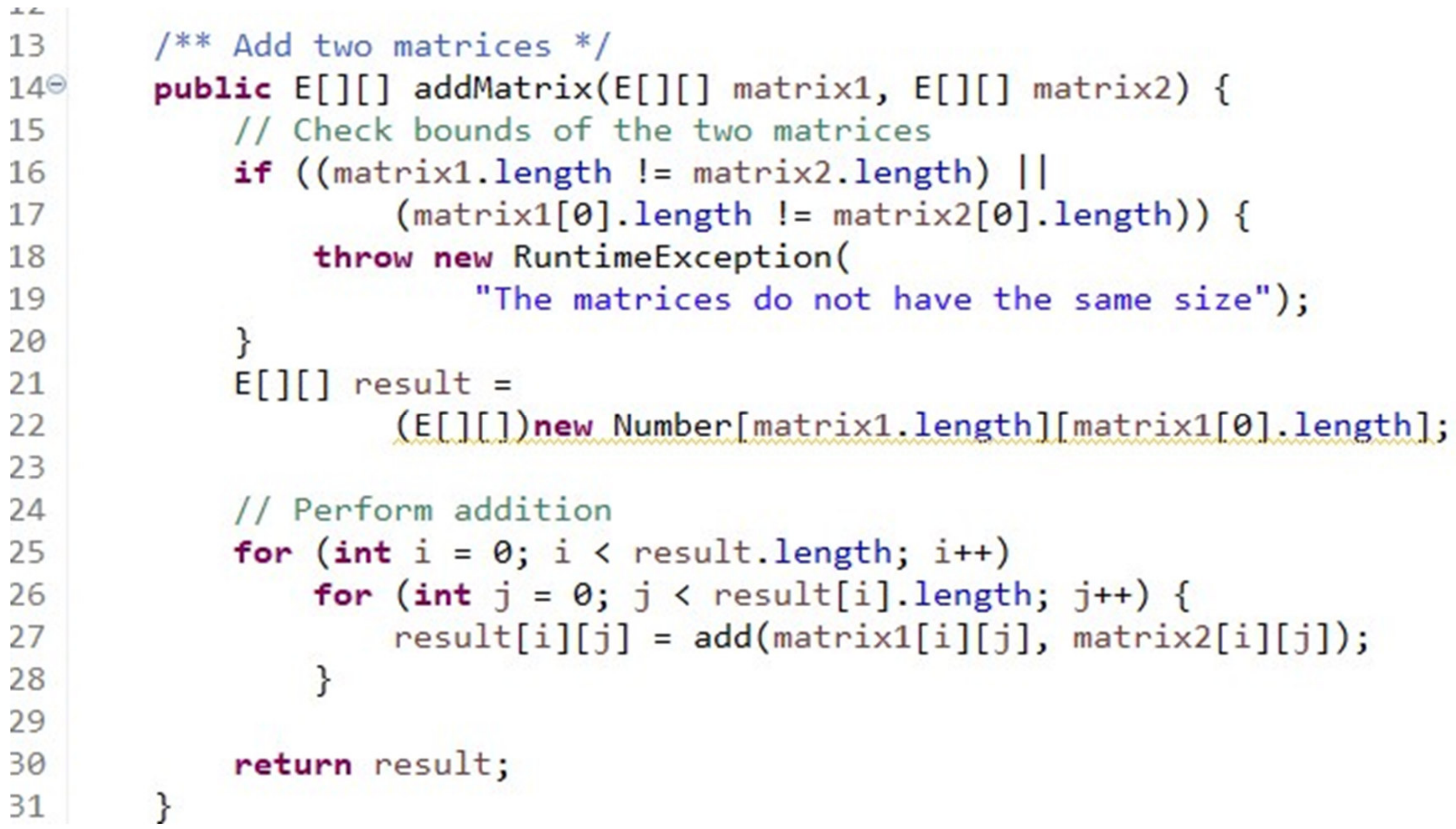
UML Diagram

Example: GenericMatrix



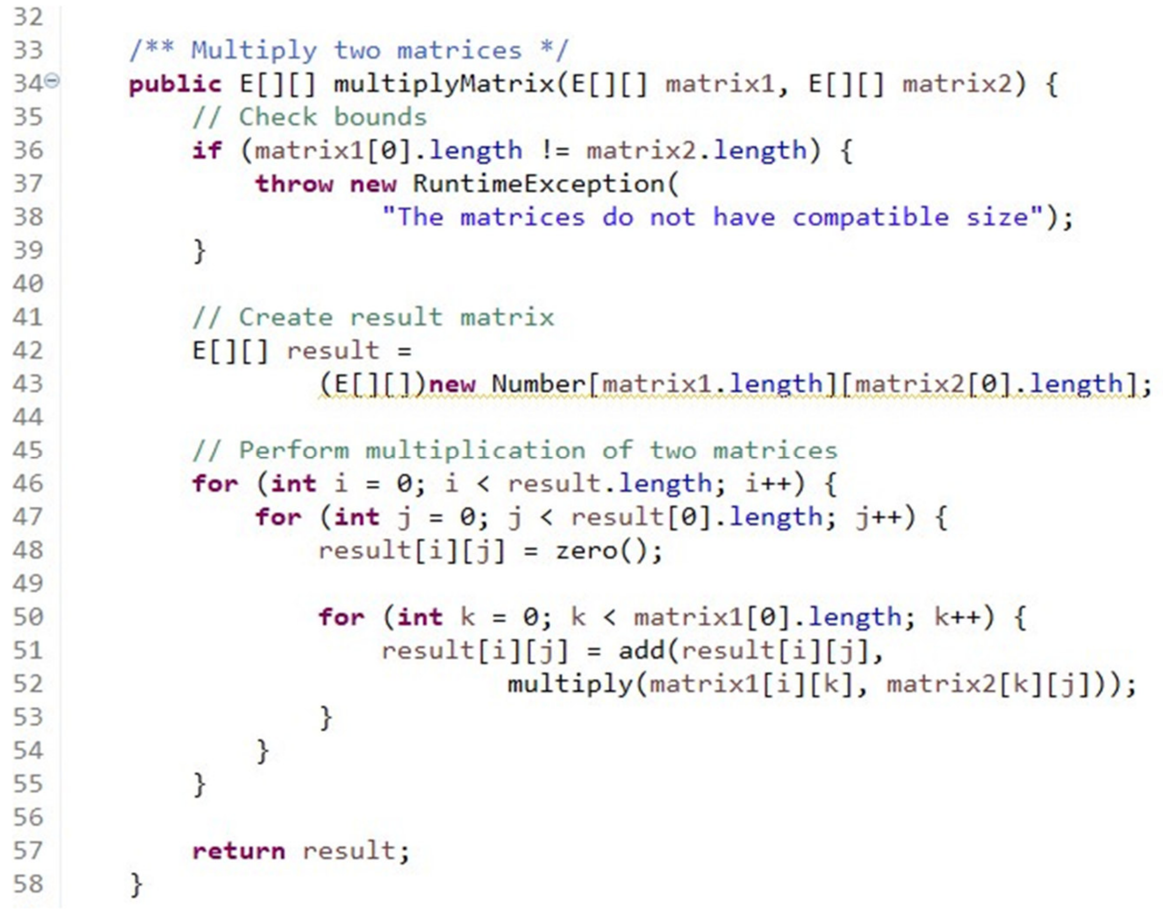
```java

public abstract class GenericMatrix <E extends Number>{

    /** Abstract method for adding two elements of the matrices */
    protected abstract E add(E o1, E o2);

    /** Abstract method for multiplying two elements of the matrices */
    protected abstract E multiply(E o1, E o2);

    /** Abstract method for defining zero for the matrix element */
    protected abstract E zero();

```

Example: GenericMatrix (addMatrix)



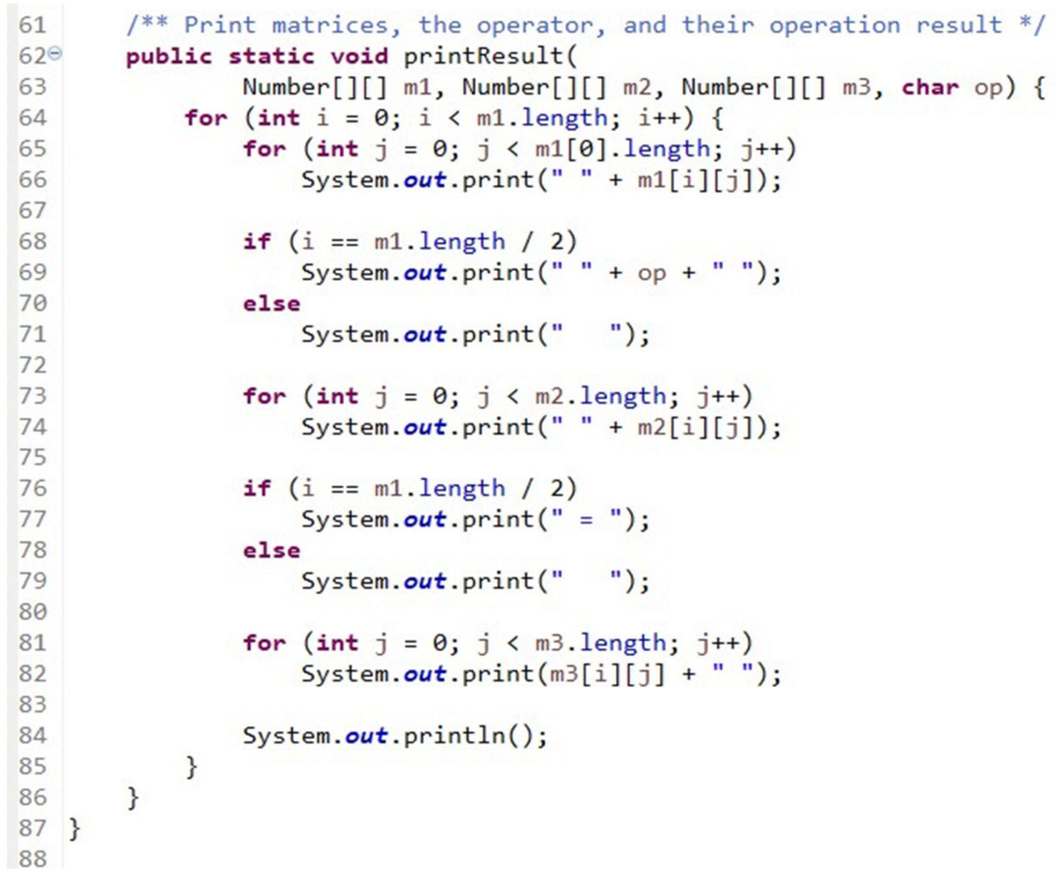
```
    /** Add two matrices */
    public E[][] addMatrix(E[][] matrix1, E[][] matrix2) {
        // Check bounds of the two matrices
        if ((matrix1.length != matrix2.length) ||
                (matrix1[0].length != matrix2[0].length)) {
            throw new RuntimeException(
                    "The matrices do not have the same size");
        }
        E[][] result =
                (E[][])new Number[matrix1.length][matrix1[0].length];

        // Perform addition
        for (int i = 0; i < result.length; i++)
            for (int j = 0; j < result[i].length; j++) {
                result[i][j] = add(matrix1[i][j], matrix2[i][j]);
            }

        return result;
    }
```

Example: GenericMatrix (multiplyMatrix)



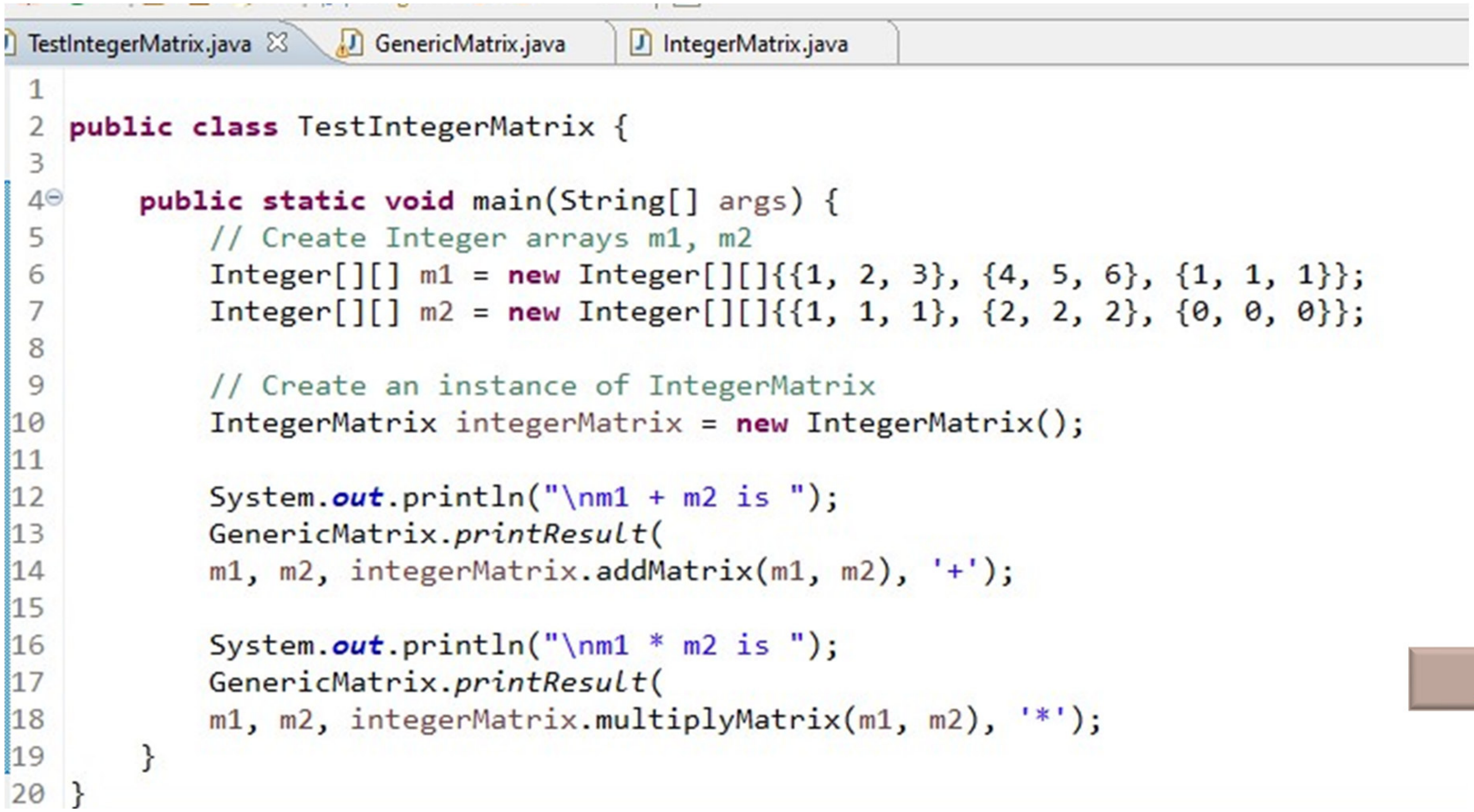
```java

    /** Multiply two matrices */
    public E[][] multiplyMatrix(E[][] matrix1, E[][] matrix2) {
        // Check bounds
        if (matrix1[0].length != matrix2.length) {
            throw new RuntimeException(
                    "The matrices do not have compatible size");
        }

        // Create result matrix
        E[][] result =
                (E[][])new Number[matrix1.length][matrix2[0].length];

        // Perform multiplication of two matrices
        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result[0].length; j++) {
                result[i][j] = zero();

                for (int k = 0; k < matrix1[0].length; k++) {
                    result[i][j] = add(result[i][j],
                            multiply(matrix1[i][k], matrix2[k][j]));
                }
            }
        }

        return result;
    }
```

Example: GenericMatrix (printResult)



```
    /** Print matrices, the operator, and their operation result */
    public static void printResult(
            Number[][] m1, Number[][] m2, Number[][] m3, char op) {
        for (int i = 0; i < m1.length; i++) {
            for (int j = 0; j < m1[0].length; j++)
                System.out.print(" " + m1[i][j]);

            if (i == m1.length / 2)
                System.out.print(" " + op + " ");
            else
                System.out.print("   ");

            for (int j = 0; j < m2.length; j++)
                System.out.print(" " + m2[i][j]);

            if (i == m1.length / 2)
                System.out.print(" = ");
            else
                System.out.print("   ");

            for (int j = 0; j < m3.length; j++)
                System.out.print(m3[i][j] + " ");

            System.out.println();
        }
    }
}
```

Test IntegerMatrix



```java

public class TestIntegerMatrix {

    public static void main(String[] args) {
        // Create Integer arrays m1, m2
        Integer[][] m1 = new Integer[][]{{1, 2, 3}, {4, 5, 6}, {1, 1, 1}};
        Integer[][] m2 = new Integer[][]{{1, 1, 1}, {2, 2, 2}, {0, 0, 0}};

        // Create an instance of IntegerMatrix
        IntegerMatrix integerMatrix = new IntegerMatrix();

        System.out.println("\nm1 + m2 is ");
        GenericMatrix.printResult(
        m1, m2, integerMatrix.addMatrix(m1, m2), '+');

        System.out.println("\nm1 * m2 is ");
        GenericMatrix.printResult(
        m1, m2, integerMatrix.multiplyMatrix(m1, m2), '*');
    }
}
```

RUN