Generics

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

Objectives

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

Why Do We Get a Warning?

```
public class ShowUncheckedWarning {

  public static void main(String[] args) {
    java.util.ArrayList list =
          new java.util.ArrayList();
    list.add("1");

    Integer i = (Integer)(list.get(0));
    // No Compiler Errors
    // Runtime error because "1" is a String
  }
}
```

However, it is better to get compiling errors than runtime errors!

How can we Fix the Warning?

```
public class ShowCompilerError {

  public static void main(String[] args) {
    java.util.ArrayList<Integer> list =
      new java.util.ArrayList<Integer>();
    list.add("1");

    list.add(new Integer(1));
  }
}
```

Compiler error on this line

Is this better? Yes, because as programmers we can fix it before we deliver the code to a customer.

How can we Fix the Error?

```
public class FixCompilerError {

  public static void main(String[] args) {
    java.util.ArrayList<Integer> list =
      new java.util.ArrayList<Integer>();
    list.add(new Integer(1));// OR
    list.add(1);
  }
}
```

What are 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
  - 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
      - a stack object for holding numbers

    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
    - We still do **code reuse**, e.g., write a single implementation for a special kind of data structure, like a single implementation of a generic stack and its standard methods
  - Most important advantage: If you attempt to use the class or method with an incompatible object, a **compile error** occurs

Generic Types

```
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

<T> represents a formal generic type, which can be replaced later with an actual concrete type

This is called ***Generic Instantiation***

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

(a) Prior to JDK 1.5

Runtime error

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

(b) JDK 1.5

Compiler error

OK. We will fix it!

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) : Object
+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) : E
+set(index: int, o: E) : E

We can avoid casting.

(b) ArrayList in JDK 1.5

Advantages: Compiler Errors instead of Runtime Errors

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

// You can now add only Integers into the list
list.add(new Integer(1));

// If you attempt to add a non-Integer, then a
//  compiler error will occur.
list.add("1"); // Compiler Error
```

Advantages: No Casting Needed

```
// Casting is not needed:
ArrayList<Integer> list = new ArrayList<>();
list.add(new Integer(1));

// Prior to JDK 1.5, without using generics, you
//  would have had to cast the return value to
//  Integer as:
Integer i = (Integer)(list.get(0));

// Since JDK 1.5
Integer i = list.get(0);
```

No Primitive types

- **However, generic types must be reference types!**
  - **You cannot replace a generic type with a primitive type such as int, double, or char**
    - **The following statement is wrong (i.e., a compiler error):**

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

// But you can use the wrapper types:
ArrayList<Integer> intList = new ArrayList<>();

// You can still add an int value to intList by Boxing:
// Java automatically wraps 1 into: new Integer(1)
intList.add(1);
```

No Casting for Get and Unboxing

```
ArrayList<Double> list = new ArrayList<Double>();
list.add(5.5); // 5.5 is automatically boxed/
          // converted to new Double(5.5)
list.add(3.0); // 3.0 is automatically boxed/
          // converted to new Double(3.0)

Double doubleObject = list.get(0);
          // No casting is needed
```

- **Also automatic Unboxing:**

```
double d = list.get(1);
          // Unboxing: Automatic conversion to double
```

  - **This is a property of numeric wrapper classes.**

Declaring Your own Generic Classes and Interfaces

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): E	Adds a new element to the top of this stack.
+isEmpty(): boolean	Returns true if the stack is empty.

Declaring Generic Classes

GenericStack.java

```
public class GenericStack<E> {
    private java.util.ArrayList<E> list = new java.util.ArrayList<E>();
    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.remove(getSize() - 1);
        return o;
        // OR just: return list.remove(getSize() - 1);
    }
    public boolean isEmpty() {
        return list.isEmpty();
    }
    @Override    // Java annotation: also used at compile time to
    public String toString() {      // detect override errors
        return "stack: " + list.toString();
    }
}
```

Using Generic Classes

```
public static void main(String[] args){
      GenericStack<Integer> s1;
      s1 = new GenericStack<>();
      s1.push(1);
      s1.push(2);
      System.out.println(s1);

      GenericStack<String> s2 = new GenericStack<>();
      s2.push("Hello");
      s2.push("World");
      System.out.println(s2);
 }
```

Output:

```
stack: [1, 2]
stack: [Hello, World]
```

Generic Static Methods

- To declare a generic method, **you place the generic type immediately after the keyword static** in the method header:

```
public class GenericMethods1 {
        public static <E> void print(E[] list) {
           for (int i = 0; i < list.length; i++)
              System.out.print(list[i] + " ");
           System.out.println();
        }
        public static void main(String[] args) {
           String[] s3 = {"Hello", "again"};
           GenericMethods1.<String>print(s3);
           // OR simply:
           print(s3);
        }
}
```

Bounded Generic Types

- A generic type can be specified as a subtype of another type:
  - consider `Circle` and `Rectangle extend GeometricObject`

```
public class GenericMethods2 {
   public static <E extends GeometricObject> boolean
          equalArea(E object1, E object2) {
       return object1.getArea() == object2.getArea();
   }
   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));
   } // Note: see https://www3.cs.stonybrook.edu/~pfodor/courses/cse160.html
}  //  for GenericObject, Circle and Rectangle
```

Sorting an Array of Objects

- We can develop a generic method for sorting an array of **Comparable** objects:

```
public class GenericSelectionSort {
  public static <E extends Comparable<E>> void
      genericSelectionSort(E[] list) {
    E currentMin;
    int currentMinIndex;
    for (int i = 0; i < list.length - 1; i++) {
      // Find the minimum in the list[i...list.length-1]
      currentMin = list[i];
      currentMinIndex = i;
      for (int j = i + 1; j < list.length; j++) {
        if (currentMin.compareTo(list[j]) > 0) {
          currentMin = list[j];
          currentMinIndex = j;
        }
      }     // continue on next slide
```

Sorting an Array of Objects

```
      // Swap list[i] with list[currentMinIndex]
      //  if necessary;
      if (currentMinIndex != i) {
        list[currentMinIndex] = list[i];
        list[i] = currentMin;
      }
    }
  }
```

Sorting an Array of Objects

```
public static void main(String[] args) {
  // Create an Integer array
  Integer[] intArray = { new Integer(2), new Integer(4),
          new Integer(3) };
  // Sort the array
  GenericSelectionSort.<Integer>genericSelectionSort(intArray);
  // Display the sorted array
  System.out.print("Sorted Integer objects: ");
  printList(intArray); // Sorted Integer objects: 2 3 4
  Double[] doubleArray = { new Double(3.4), new Double(1.3) };
  GenericSelectionSort.<Double>genericSelectionSort(doubleArray);
  // same for Character, String, etc.
}
/** Print an array of objects */
public static void printList(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

Sorting an Array of Objects

```
public static <E extends Comparable<E>>
  void genericSelectionSort(E[] list) {
```

- The generic type **<E extends Comparable<E>>** has two meanings:
  - First, it specifies that **E** must be a subtype of **Comparable**
  - Second, it specifies that the elements to be compared are of the **E** type as well
- The sort method uses the **compareTo** method to determine the order of the objects in the array
  - **Integer**, **Double**, **Character**, and **String** implement **Comparable**, so the objects of these classes can be compared using the **compareTo** method

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
  - In JDK1.5 and higher, the ***Raw type:***

```
ArrayList list = new ArrayList();
```

is roughly equivalent to:

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

  - Therefore, all programs written in previous JDK versions are still executable.

- **Note:** ***Backward compatibility***: a technology is backward compatible if the input developed for an older technology can work with the newer technology:
  - Note: Python 3.X broke backward compatibility to Python 2.X
  - print 1

works in Python 2, but not Python 3

Raw Type is Unsafe

```
public class Unsafe {
  // Return the maximum between two objects
  public static Comparable max1(Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
  public static void main(String[] args) {
    System.out.println(max1("Welcome", 23));
  }
}
```

- **`Comparable o1`** and **`Comparable o2`** are raw type declarations
- Be careful: raw types are unsafe:

  `max1("Welcome", 23)`

  - No compilation error.
  - Runtime Error!

How can we make it Safe?

```
public class Safe {
  // Return the maximum between two objects
  public static <E extends Comparable<E>> E max2(E o1, E o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
  public static void main(String[] args) {
    System.out.println(max2("Welcome", 23));
  }
}
```

Now: **`max2("Welcome", 23)`** becomes a Compiler Error because the two arguments of the max method **must have the same type** (e.g., two **`Strings`** or two **`Integer`** objects)

- Note: the type **`E`** must be a subtype of **`Comparable<E>`**
  - E.g., **`String`** is a subtype of **`Comparable<String>`**

Wildcards

- A problem to be solved:
  - **`Integer`** is a subclass of **`Number`**, but **`GenericStack<Integer>`** is not a subclass of **`GenericStack<Number>`**
    - You can use unbounded wildcards, bounded wildcards, or lower-bound wildcards to specify a **range** for a **generic type**:
      - **`?`** unbounded wildcard: any class
      - **`? extends T`** bounded wildcard: any **subclass** of T (including T)
      - **`? super T`** lower bound wildcard: any **superclass** of T (including T)
  - **`<? extends Number>`** is a wildcard type that represents **`Number`** or a subtype of **`Number`**
  - **`GenericStack<Integer>`** is a subclass of **`GenericStack<? extends Number>`**

```
public class WildCardDemo1 {
  /**           // It expects GenericStack<Number>
   * Find the maximum in a stack of numbers
  */
  public static double max(GenericStack<Number> stack) {
    double max = stack.pop().doubleValue(); // initialize max
    while (!stack.isEmpty()) {
      double value = stack.pop().doubleValue();
      if (value > max)
        max = value;
    }
    return max;
  }
  public static void main(String[] args) {
    GenericStack<Integer> intStack = new GenericStack<Integer>();
    intStack.push(1); // 1 is autoboxed into new Integer(1)
    intStack.push(2);
    intStack.push(-2);
    System.out.print("The max number is " + max(intStack));
      // Compile Error: max cannot be applied to GenericStack<Integer>
    }
}     // Integer is a subclass of Number, but GenericStack<Integer> is
      //  not a subclass of GenericStack<Number>
```

```
public class WildCardDemo1B {
  /**
   * Find the maximum in a stack of numbers
  */
  public static double max(GenericStack<? extends Number> stack) {
    double max = stack.pop().doubleValue(); // initialize max
    while (!stack.isEmpty()) {
      double value = stack.pop().doubleValue();
      if (value > max)
        max = value;
    }
    return max;
  }
  public static void main(String[] args) {
    GenericStack<Integer> intStack = new GenericStack<Integer>();
    intStack.push(1); // 1 is autoboxed into new Integer(1)
    intStack.push(2);
    intStack.push(-2);
    System.out.print("The max number is " + max(intStack));
  }
}
```

Output:

The max number is 2.0

```
public class WildCardDemo2 {
  /**
   * Print objects and empties the stack
  */
  public static void print(GenericStack<?> stack) {
    while (!stack.isEmpty())
      System.out.print(stack.pop() + " ");
  }
  public static void main(String[] args) {
    GenericStack<Integer> intStack = new GenericStack<Integer>();
    intStack.push(1); // 1 is autoboxed into new Integer(1)
    intStack.push(2);
    intStack.push(-2);
    print(intStack);
  }
}
```

Output:
-2 2 1

```
public class WildCardDemo3 {
    // Add stack1 TO stack2: the type ofelements in stack2 must be
    // a SUPERTYPE of the type of elements in stack1
    public static <T> void add(GenericStack<? extends T> stack1,
                               GenericStack<T> stack2) {
        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }
    public static void print(GenericStack<?> stack) {
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }
    public static void main(String[] args) {
        GenericStack<String> stack1 = new GenericStack<String>();
        GenericStack<Object> stack2 = new GenericStack<Object>();
        stack2.push("Java");
        stack2.push(2);
        stack1.push("Sun");
        add(stack1, stack2);
        print(stack2);
    }
}
```

Output:
Sun 2 Java

```
public class WildCardDemo4 {
    // Add stack1 TO stack2: the type ofelements in stack2 must be
    // a SUPERTYPE of the type of elements in stack1
    public static <T> void add(GenericStack<T> stack1,
                               GenericStack<? super T> stack2) {
        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }
    public static void print(GenericStack<?> stack) {
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }
    public static void main(String[] args) {
        GenericStack<String> stack1 = new GenericStack<String>();
        GenericStack<Object> stack2 = new GenericStack<Object>();
        stack2.push("Java");
        stack2.push(2);
        stack1.push("Sun");
        add(stack1, stack2);
        print(stack2);
    }
}
```

Output:
Sun 2 Java

Erasure and Restrictions on Generics

- Generics are implemented in Java using an approach called ***type erasure***:
  - The compiler uses the generic type information to compile the code, but erases it afterwards
    - This approach enables the generic code to be backward-compatible with the legacy code that uses raw types
  - So **the generic information is not available at run time**
  - Once the compiler confirms that a generic type is used safely, it converts the generic type back to a raw type **`Object`**
    - 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 **`Object`**

```
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)

Erasure and Restrictions on Generics

- More examples of translations:

```
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)

```
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)

Erasure and Restrictions on Generics

- The 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
  - These are **true**:

```
System.out.println(stack1 instanceof GenericStack);
System.out.println(stack2 instanceof GenericStack);
```

  - These are **compile errors**:

```
System.out.println(stack1 instanceof GenericStack<String>);
System.out.println(stack2 instanceof GenericStack<Integer>);
```

since **GenericStack<String>** and **GenericStack<Integer>** are not stored as separate classes in the JVM, using them at runtime makes no sense

Restrictions on Generics

- Because generic types are erased at runtime, there are certain restrictions on how generic types can be used:
  - **Restriction 1:** Cannot Create an Instance of a Generic Type (i.e., `new E()`). 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
  - **Restriction 2:** Generic Array Creation is Not Allowed (i.e., `new E[100]`). For example, the following statement is wrong:

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

Restrictions on Generics

- **Restriction 3:** 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

```
public class DataStructure<E> {
    public static void m(E o1) { // Illegal
    }

    public static E o1; // Illegal

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

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 { … }
```

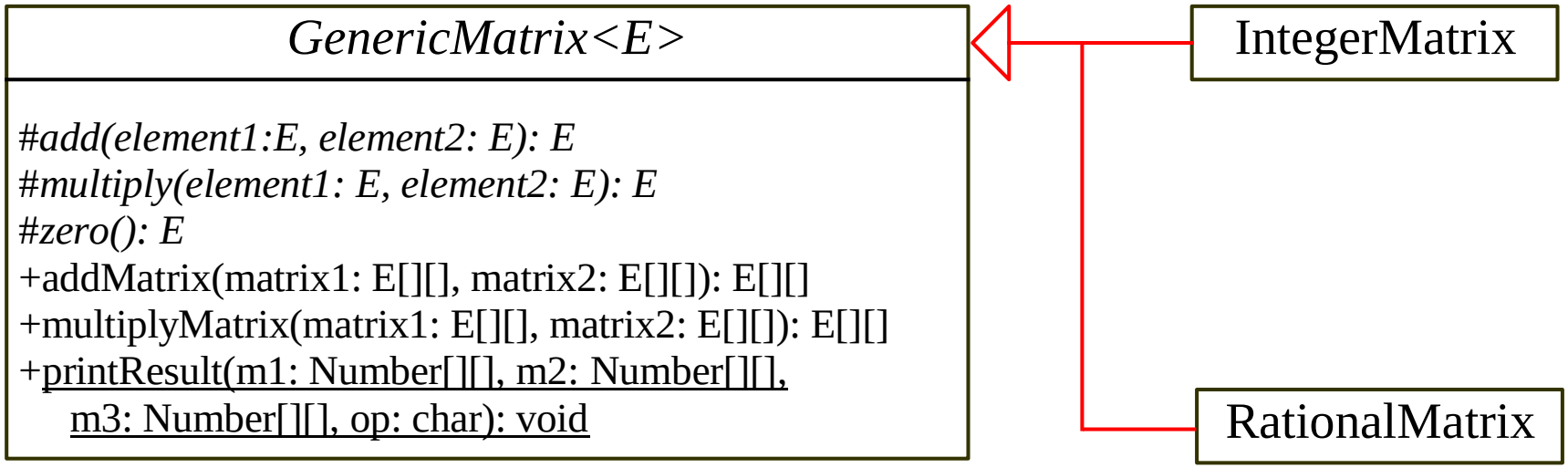
  - If it were allowed, you would have a catch clause for **`MyException<T>`** as follows:

```
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.**

Use case: Designing Generic Matrix Classes

- A generic class for matrix arithmetic.
  - The class implements matrix addition and multiplication common for all types of matrices.



```
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
     */
    protected abstract E multiply(E o1, E o2);

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

```
/**
 * 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;
}
```

```
/**
 * 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;
}
```

```
/** 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();
    }
}
```

```
public class IntegerMatrix extends GenericMatrix<Integer> {
    @Override
    /**
     * Add two integers
     */
    protected Integer add(Integer o1, Integer o2) {
        return o1 + o2;
    }

    @Override
    /**
     * Multiply two integers
     */
    protected Integer multiply(Integer o1, Integer o2) {
        return o1 * o2;
    }

    @Override
    /**
     * Specify zero for an integer
     */
    protected Integer zero() {
        return 0;
    }
```

```
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), '*');
}
```

Output:

```
m1 + m2 is
1 2 3    1 1 1    2 3 4
4 5 6 +  2 2 2 =  6 7 8
1 1 1    0 0 0    1 1 1

m1 * m2 is
1 2 3    1 1 1     5  5  5
4 5 6 *  2 2 2 =  14 14 14
1 1 1    0 0 0     3  3  3
```

```
public class Rational extends Number implements Comparable<Rational> {
    // Data fields for numerator and denominator

    private long numerator = 0;
    private long denominator = 1;

    /**
     * Construct a rational with specified numerator and denominator
     */
    public Rational(long numerator, long denominator) {
        long gcd = gcd(numerator, denominator);
        this.numerator = ((denominator > 0) ? 1 : -1) * numerator / gcd;
        this.denominator = Math.abs(denominator) / gcd;
    }
    private static long gcd(long n, long d) {
        long n1 = Math.abs(n);
        long n2 = Math.abs(d);
        int gcd = 1;

        for (int k = 1; k <= n1 && k <= n2; k++) {
            if (n1 % k == 0 && n2 % k == 0) {
                gcd = k;
            }
        }

        return gcd;
    }
```

```
/**
 * Add a rational number to this rational
 */
public Rational add(Rational secondRational) {
    long n = numerator * secondRational.getDenominator()
            + denominator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
}
/**
 * Multiply a rational number to this rational
 */
public Rational multiply(Rational secondRational) {
    long n = numerator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
}
@Override
public String toString() {
    if (denominator == 1) {
        return numerator + "";
    } else {
        return numerator + "/" + denominator;
    }
}
```

```
public class RationalMatrix extends GenericMatrix<Rational> {

    @Override
    /**
     * Add two rational numbers
     */
    protected Rational add(Rational r1, Rational r2) {
        return r1.add(r2);
    }

    @Override
    /**
     * Multiply two rational numbers
     */
    protected Rational multiply(Rational r1, Rational r2) {
        return r1.multiply(r2);
    }

    @Override
    /**
     * Specify zero for a Rational number
     */
    protected Rational zero() {
        return new Rational(0, 1);
    }
```

```
private long getDenominator() {
        return denominator;
}
private long getNumerator() {
        return numerator;
}
 // Skeleton methods
@Override
public int compareTo(Rational arg0) {
        return 0;
}
@Override
public double doubleValue() {
        return 0;
}
@Override
public float floatValue() {
        return 0;
}
@Override
public int intValue() {
        return 0;
}
@Override
public long longValue() {
        return 0;
}
```

```
public static void main(String[] args) {
    // Create two Rational arrays m1 and m2
    Rational[][] m1 = new Rational[3][3];
    Rational[][] m2 = new Rational[3][3];
    for (int i = 0; i < m1.length; i++) {
        for (int j = 0; j < m1[0].length; j++) {
            m1[i][j] = new Rational(i + 1, j + 5);
            m2[i][j] = new Rational(i + 1, j + 6);
        }
    }
    // Create an instance of RationalMatrix
    RationalMatrix rationalMatrix = new RationalMatrix();
    System.out.println("\nm1 + m2 is ");
    GenericMatrix.printResult(
            m1, m2, rationalMatrix.addMatrix(m1, m2), '+');
    System.out.println("\nm1 * m2 is ");
    GenericMatrix.printResult(
            m1, m2, rationalMatrix.multiplyMatrix(m1, m2), '*');
}
```

Output:

```
m1 + m2 is
 1/5 1/6 1/7     1/6 1/7 1/8     11/30 13/42 15/56
 2/5 1/3 2/7  +   1/3 2/7 1/4  =  11/15 13/21 15/28
 3/5 1/2 3/7     1/2 3/7 3/8     11/10 13/14 45/56
m1 * m2 is
 1/5 1/6 1/7     1/6 1/7 1/8     101/630 101/735 101/840
 2/5 1/3 2/7  *   1/3 2/7 1/4  =  101/315 202/735 101/420
 3/5 1/2 3/7     1/2 3/7 3/8     101/210 101/245 101/280
```