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 parameters 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, a compile error occurs.

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

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 since JDK 1.5

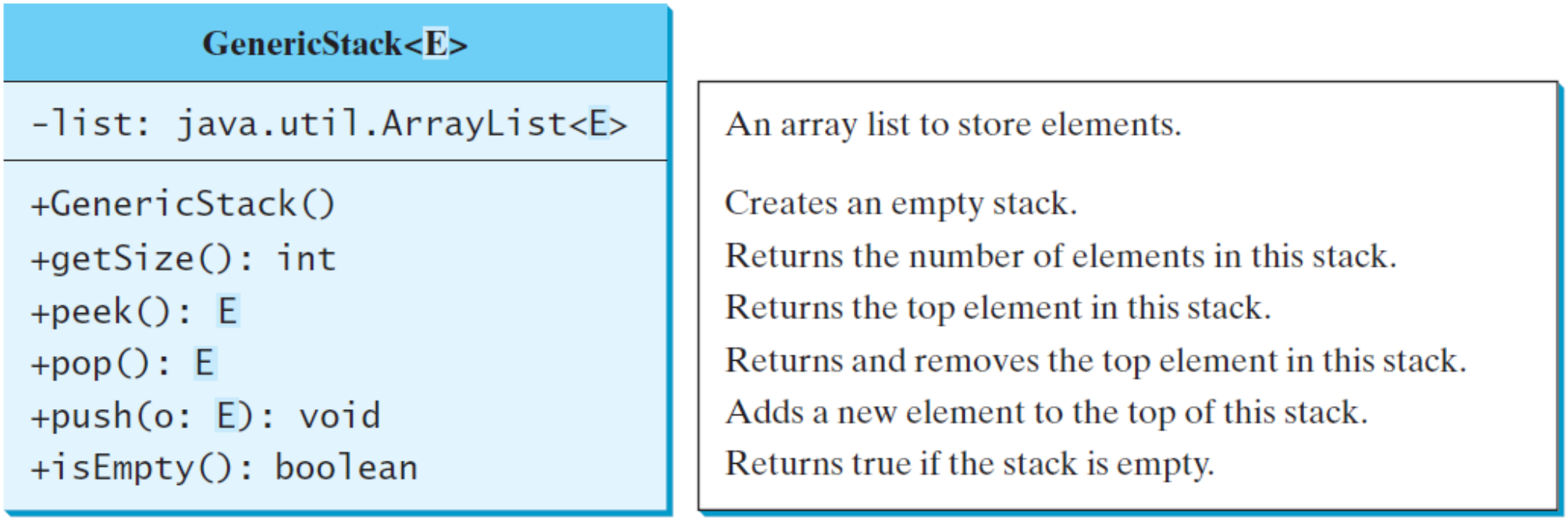
No Casting Needed

AutoBoxing

```
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
```

AutoUnBoxing

Declaring Generic Classes and Interfaces



GenericStack

```java
public class GenericStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();

  public GenericStack () {
  }

  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();
  }
}
```

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

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

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();
 }
```

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

```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();
  }
}
```

To declare a generic method, you place the generic type **<E>** immediately after the keyword **static** 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:

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

or simply invoke it as follows:

print(integers);
print(strings);

Bounded Generic Type

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

```
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

```
// raw type
ArrayList list = new ArrayList();
```

This is *roughly* equivalent to

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

A generic class or interface used without specifying a concrete type, called a raw type*, enables backward compatibility with earlier versions of Java.*

Raw Type is Unsafe

```
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;
  }
}
```

Max.max("Welcome", 23);

Runtime Error:

Make it Safe

```
public class MaxUsingGenericType {
  /** 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 Error :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

Wildcards

Why wildcards are necessary? See this example.

WildCardNeedDemo

?	unbounded wildcard	**? extends Object**
? extends T	bounded wildcard	**T** or an unknown subtype of **T**
? super T	lower bound wildcard	**T** or an unknown supertype of **T**

AnyWildCardDemo

SuperWildCardDemo

```java
public class WildCardNeedDemo {
  public static void main(String[] args ) {
     GenericStack<Integer> intStack = new GenericStack<>();
     intStack.push(1); // 1 is autoboxed into new Integer(1)
     intStack.push(2);
     intStack.push(-2);

// Error:     System.out.print("The max number is " + max(intStack));
  }

  /** 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;
  }
}
```

The fact is that **Integer** is a subtype of **Number**, bu **GenericStack<Integer>** is not a subtype of **GenericStack<Number>**.

Fix 1:

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

Now can invoke:

max(new GenericStack<Integer>())
or
max(new GenericStack<Double>()).

When ? super T is needed

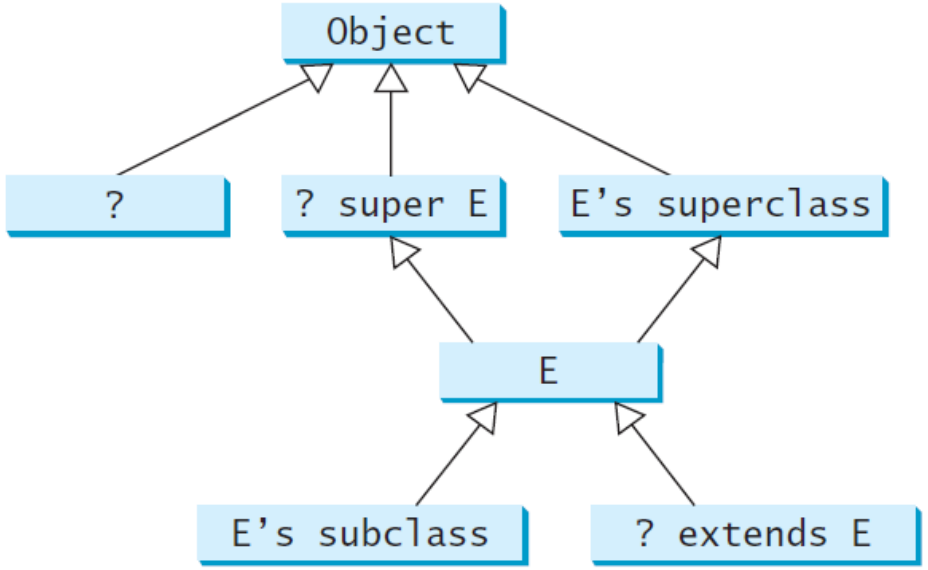
```
public class AnyWildCardDemo {
  public static void main(String[] args ) {
     GenericStack<Integer> intStack = new GenericStack<>();
     intStack.push(1); // 1 is autoboxed into new Integer(1)
     intStack.push(2);
     intStack.push(-2);

     print(intStack);
  }

  /** Print objects and empties the stack */
  public static void print(GenericStack<?> stack) {
    while (!stack.isEmpty()) {
      System.out.print(stack.pop() + " ");
    }
  }
}
```

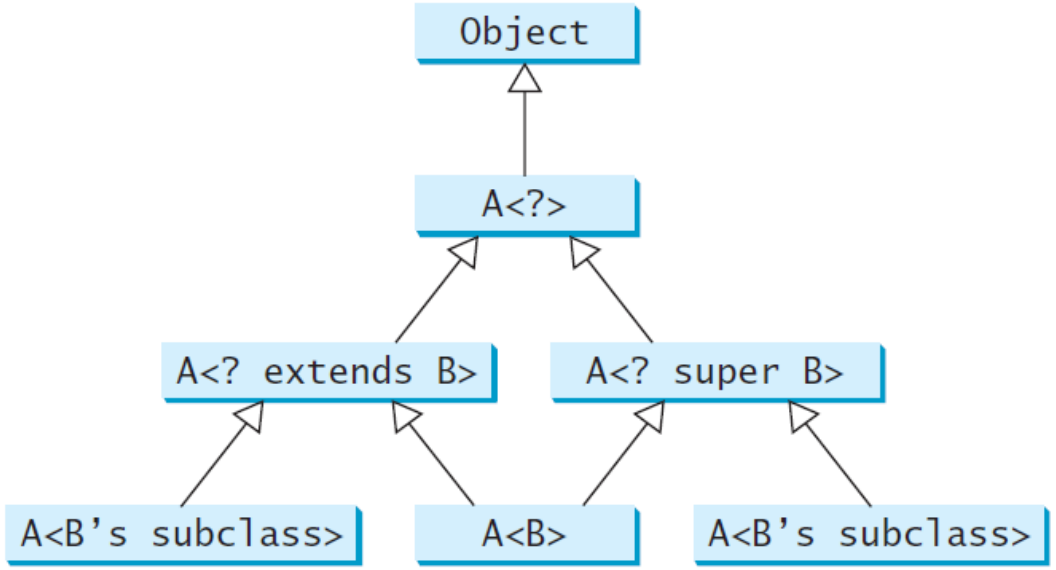
```
public class SuperWildCardDemo {
  public static void main(String[] args) {
    GenericStack<String> stack1 = new GenericStack<String>();
    GenericStack<Object> stack2 = new GenericStack<Object>();
    stack2.push("Java");
    stack2.push(2);
    stack1.push("Sun");
    add(stack1, stack2);
    AnyWildCardDemo.print(stack2);
  }

  public static <T> void add(GenericStack<T> stack1,
      GenericStack<? super T> stack2) {
    while (!stack1.isEmpty())
      stack2.push(stack1.pop());
  }
}
```

will also work:

public static <T> **void** add(GenericStack<? **extends** T> stack1, GenericStack<T> stack2)

Generic Types and Wildcard Types





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.

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)

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

(a)

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

(b)

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

(a)

```java
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

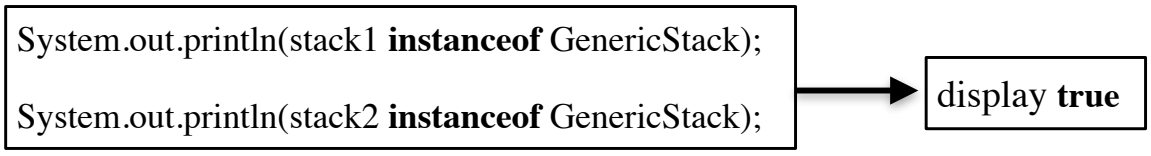
(b)

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];).
  Use:
  E[] elements = (E[])**new** Object[capacity];

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

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

- Restriction 4: Exception Classes Cannot be Generic.

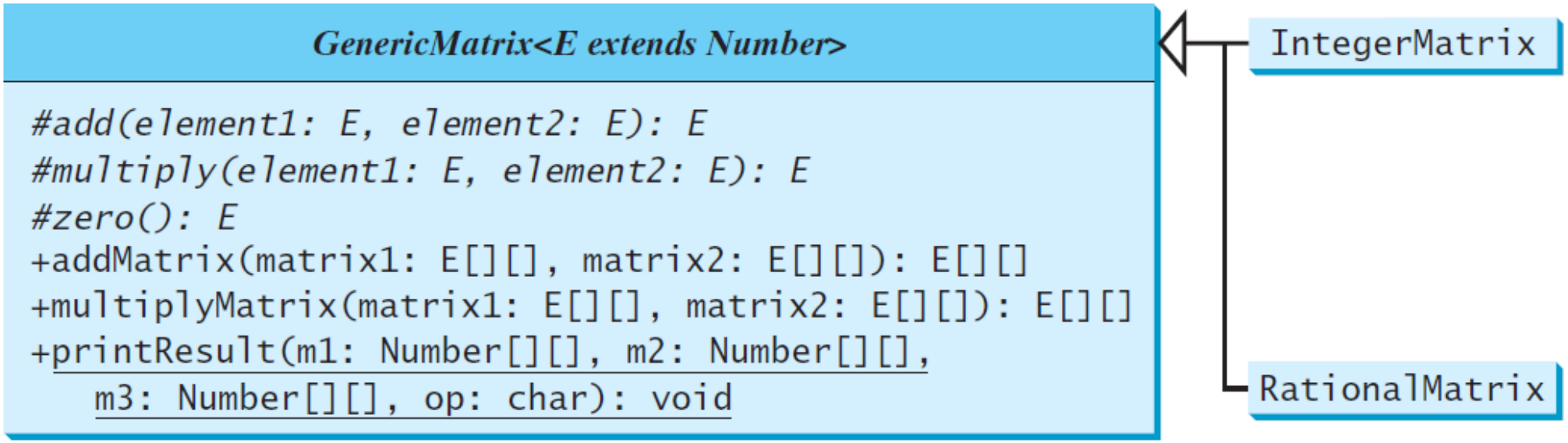
```
public class MyException<T> extends Exception { }
```

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

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

J>?

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

IntegerMatrix

TestIntegerMatrix

RationalMatrix

TestRationalMatrix