CS 91 USACO

Bronze Division

Unit 2: 1-D Data Structures

LECTURE 5: GENERIC PROGRAMMING

DR. ERIC CHOU

IEEE SENIOR MEMBER

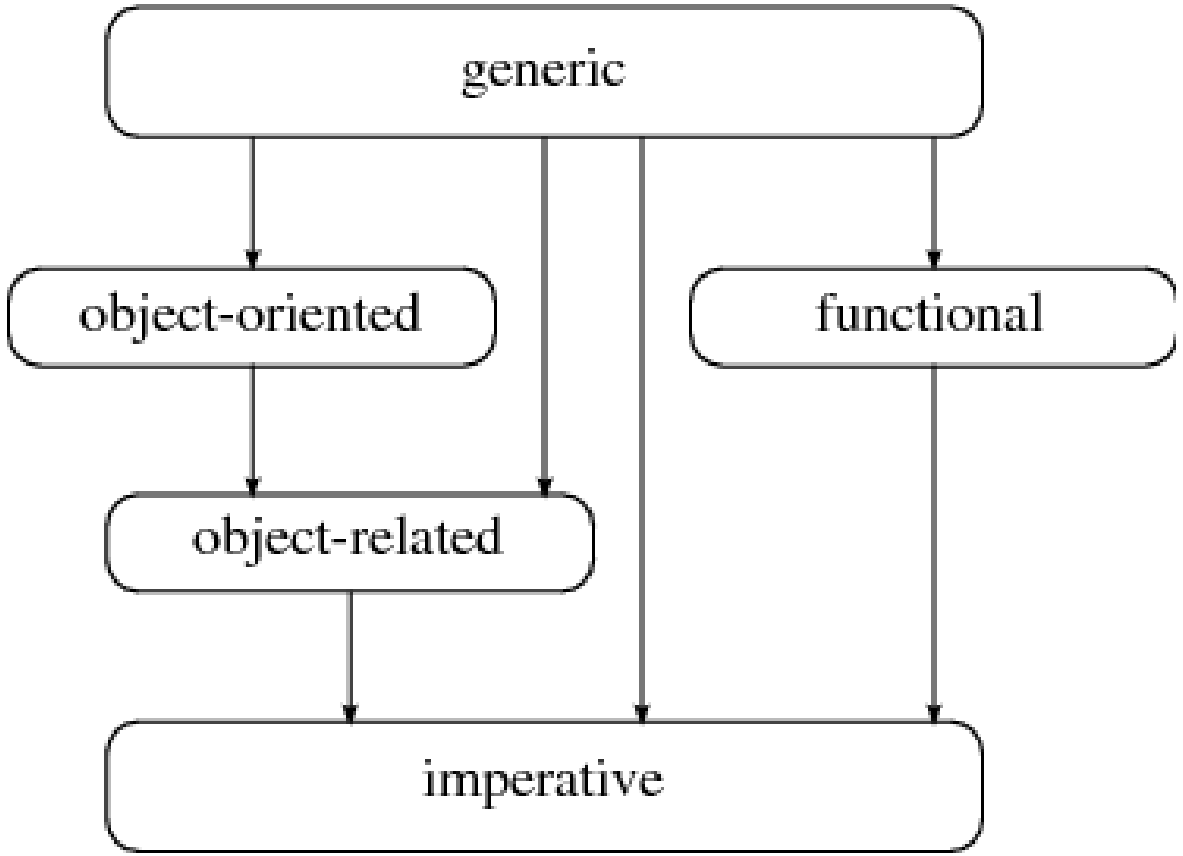
Objectives

- Generic programming is another way of overloading your functions (methods) for different data type. And, it can do more than overloading.
- Overloading can only be applied to methods.
- Generic Programming can be applied to both data field and methods. Generic Data Container, Generic Library Functions, Generic Polymorphic methods.
- Generic Programming further expands Object-Oriented Programming.

Overview

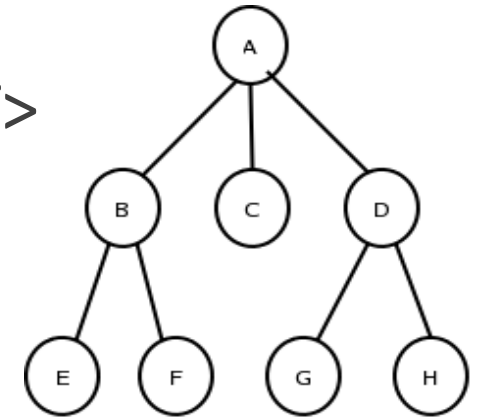
SECTION 1

Generic Programming

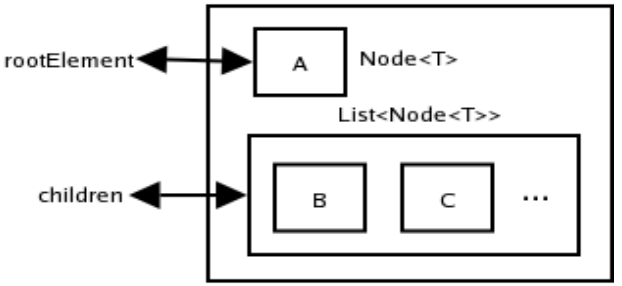


Generic Data Containers

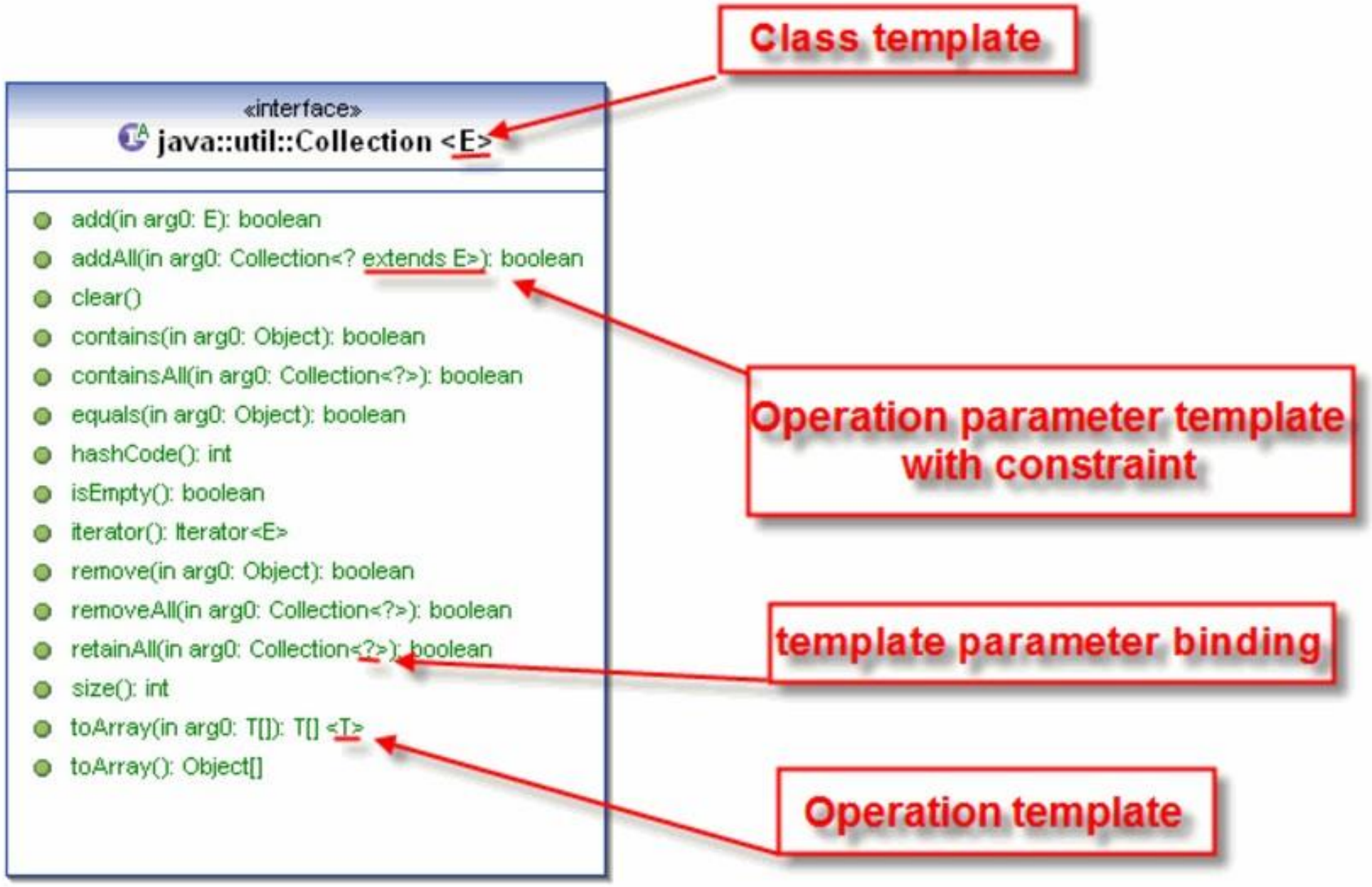
- Abstract Data Types (such as, Queue, Stack, Map, Set) tend to be Generic.
- Generic Programming can be realized by inheritance and polymorphism, or parametric polymorphism (Generic language structure)
- Object versus Type Variable<T>



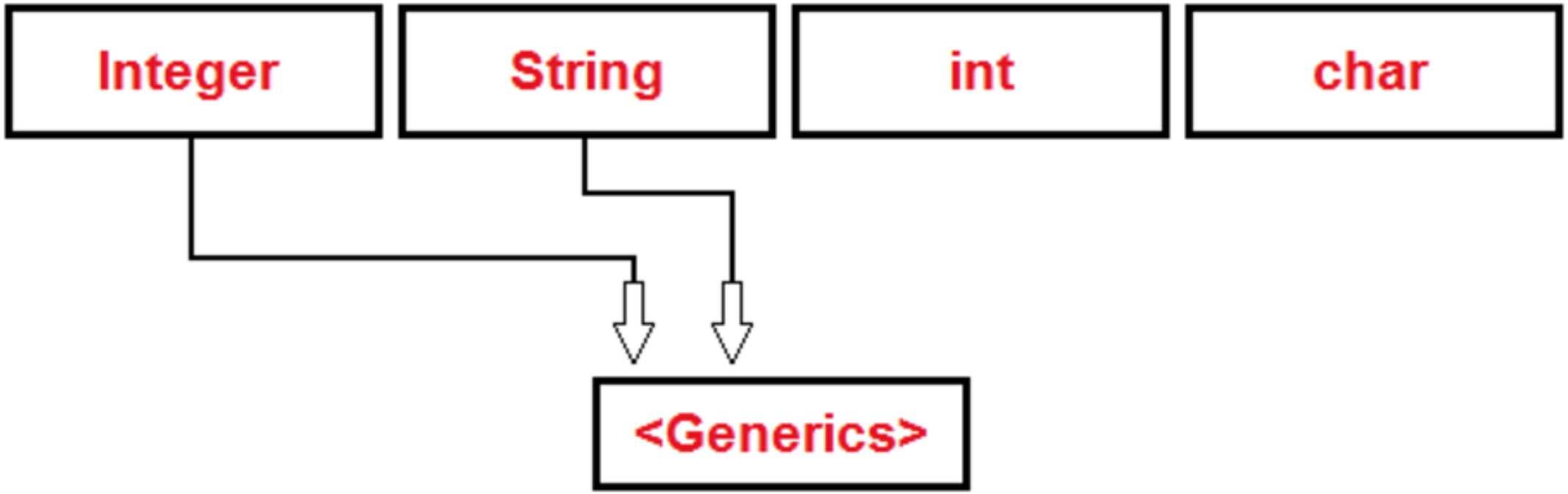
Logical View



Data Structure representation
'of an N-ary Tree


«interface»
java::util::Collection <E>
add(in arg0: E): boolean
addAll(in arg0: Collection<? extends E>): boolean
clear()
contains(in arg0: Object): boolean
containsAll(in arg0: Collection<?>): boolean
equals(in arg0: Object): boolean
hashCode(): int
isEmpty(): boolean
iterator(): Iterator<E>
remove(in arg0: Object): boolean
removeAll(in arg0: Collection<?>): boolean
retainAll(in arg0: Collection<?>): boolean
size(): int
toArray(in arg0: T[]): T[] <T>
toArray(): Object[]
Class template
Operation parameter template with constraint
template parameter binding
Operation template

Generic Methods



Array Sorting Algorithms

Generic Method for Sorting

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

What is Generics?

SECTION 2

Parametric Generics

- Generic Parameters on the parameter list of a method
- Generic return value for the output of a method
- Generic container such as **ArrayList**

We may define a generic sort method with input generic array and returned sorted generic array.

Java define a generic **ArrayList** class for storing the elements of a generic type.

Generic in Java

- Use a type variable declaration to notify compiler that the data type can be substituted in the implementation of method, data structure so that the method can take different parameter types and the data structure can contain element of different data types.

```
<T> T method(T arg);
};
```

```
Bar b = method(aBar); //OK
b = method(not_a_Bar);//Compile error
```

- The actual type parameter is not passed
  - The compiler infers the type argument
- Compiler ensures arg and return are same type
  - When the method is compiled
  - On method calls

```
class Foo <T> {
  void method(T arg);
};
```

```
Foo<Bar> fb = new Foo<bar>;
fb.method(aBar); //OK
fb.method(not_a_Bar); // compile error
```

- Foo is generic: Same behavior for any possible T (the type parameter)
- Single Source: only one foo.java
  - Low maintence
- Single binary: only one foo.class
  - No binary explosion (but imposes restrictions)

Advantage of Using Generic

- To detect errors at compile time rather than run-time.
- A generic class or method permits you to specify allowable types of objects that the class or method can work with. If you attempt to use an incompatible object, the compiler will detect that error.
- To improve the readability and reliability of software.

Generics Basics

SECTION 3

Type Variable

<T> in angle brackets

- **<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. (**E**ntity and **T**ype)

- To see the benefits of using generics, let us examine the code in Figure B. The statement in Figure B(a) declares that **c** is a reference variable whose type is **Comparable** and invokes the **compareTo** method to compare a **Date** object with a string. The code compiles fine, but it has a runtime error because a string cannot be compared with a date.

Type Variable

<T> in angle brackets

- The statement in Figure B(b) declares that **c** is a reference variable whose type is **Comparable<Date>** and invokes the **compareTo** method to compare a **Date** object with a string.

- This code generates a compile error, because the argument passed to the **compareTo** method must be of the **Date** type. Since the errors can be detected at compile time rather than at runtime, the generic type makes the program more reliable. (The compareTo() can be overrided.)

- The **ArrayList** Class. This class has been a generic class since JDK 1.5.

Type Variable

<T> in angle brackets

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

Figure C.

ArrayList as an Example for Generic Container

Declaration of the Pointer(Reference):

```
ArrayList<String> alist = new ArrayList<String>();
```

Addition of Element (body):

```
alist.add(new String(1));
```

Generic Container only for Reference Type:

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

The primitive type is not allowed here.

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.

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

Defining 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.

This example creates a stack to hold integers and adds three integers to the stack.

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

Instead of using a generic type, you could simply make the type element Object, which can accommodate any object type. However, using generic types can improve software reliability and readability, because certain errors can be detected at compile time rather than at runtime. For example, because stack1 is declared GenericStrck<String>, only strings can be added to the stack. It would be a compile error if you attempted to add an integer to stack1.

Note:

- Multiple type variables for a generic class definition. For example, **<E1, E2, E3>**
- To create a stack of strings, you can **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()

Demonstration Program

GENERICSTACK.JAVA +
TESTGENERICSTACK.JAVA

Generic Methods

SECTION 4

Generic Methods

A generic type can be defined for a static method

- You can define generic interfaces (e.g., the **Comparable** interface in Figure B(b) and classes (e.g., the **GenericStack** class in **GenericStack.java**). You can also use generic types to define generic methods. For example,
  - **GenericMethodDemo.java** defines a generic method to print an array of objects.
  - **GenericMethodDemo.<Integer>print(integers)** passes an array of integer objects to invoke the generic print method.
  - **GenericMethodDemo.<String>print(strings)** invokes print with an array of strings.

Declaration of a Generic Method

Generic Instance Method has different way or declaring type variable from Generic Static Method.

- To declare a generic method, you place the generic type **<E>** immediately after the keyword **static** in the method header. For example,

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);
```

Demonstration Program

GENERICMETHODDEMO.JAVA

Bounded Generic Type

SECTION 5

Declaration of a Bounded Generic Type

Children Under the Parent Class

- A generic type can be specified as a subtype of another type. Such a generic type is called ***bounded***. For example, BouundedTypeDemo.java revises the **equalArea** method in BouundeTypeDemo.java,
- TestGeometricObject.java, to test whether two geometric objects have the same area. The bounded generic type **<E extends GeometricObject>** (line 7) specifies that **E** is a generic subtype of **GeometricObject**. You must invoke **equalArea** by passing two instances of **GeometricObject**.
- The unbounded generic type**<E>** is the same as **<E extends Object>**.
- To define a generic type for a class, place it after the class name, such as **GenericStack<E>**. To define a generic types for a method, place the generic type before the method return type, such as **<E> void max(E o1, E o2)**

Demonstration Program

MAX.JAVA+MAXUSINGGENERICTYPE.JAVA +MAXDEMO.JAVA

Demonstration Program

BOUNDEDTYPEDEMO.JAVA

Case Study: Generic Sorting program

SECTION 6

Sort Algorithm on Generic Data Collection

- Sort algorithm is a good example to demonstrate the flexibility of generic method.
- For a data type to be qualified for sorting, it must be comparable. That means there is certain order among the elements. In Java objects, the data class must implement **Comparable** Interface or the data type must have a method working like compareTo or those basic comparison operators (==, >=, >, <, <=, !=).
- To make a generic method for sorting, it will be easier for object types which support **Comparable** Interface.

Case Study: Sorting an Array of Objects

You can develop a generic method for sorting an array of Comparable objects.

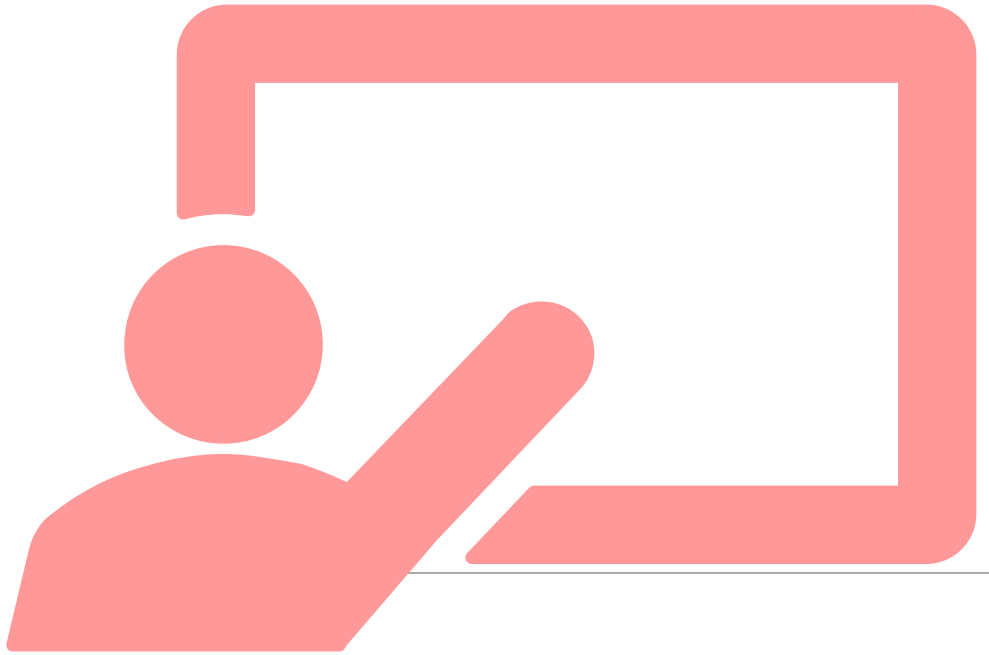
- The algorithm for the sort method is the same as in **SelectionSort.java**.
- The sort method in that program sorts an array of double values. The sort method in this example can sort an array of any object type, provided that the objects are also instances of the Comparable interface.
- The generic type is defined as **<E extends Comparable<E>>**. This has two meanings. First, it specifies that **E** is 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. The program creates arrays of Integer objects, Double objects, Character objects, and String objects and invoke the sort method to sort these arrays.

Demonstration Program

GENERICSORT.JAVA

Wildcard Generics

SECTION 7



Wildcard Generics

LECTURE 9

Wildcard Generic Type

You can use unbounded wildcards, bounded wildcards, or lower-bound wildcards to specify a range for a generic type.

- What are wildcard generic types and why are they needed? **WildCardNeedDemo.java** gives an example to demonstrate the needs. The example defines a generic max method for finding the maximum in a stack of numbers. 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 **WildCardNeedDemo.java** has a compile error in because intStack is not an instance of GenericStack<Number>. Thus, you cannot invoke max(intStack). The fact is that Integer is a subtype of Number, but **GenericStack<Integer>** is not a subtype of **GenericStack<Number>**.

- To circumvent this problem, use wildcard generic types. A wildcard generic type has three forms: **?** and **? extends T**, as well as **? super T**, where **T** is a generic type.

Demonstration Program

WILDCARDNEEDDEMO.JAVA

Unbounded Wildcard Generics

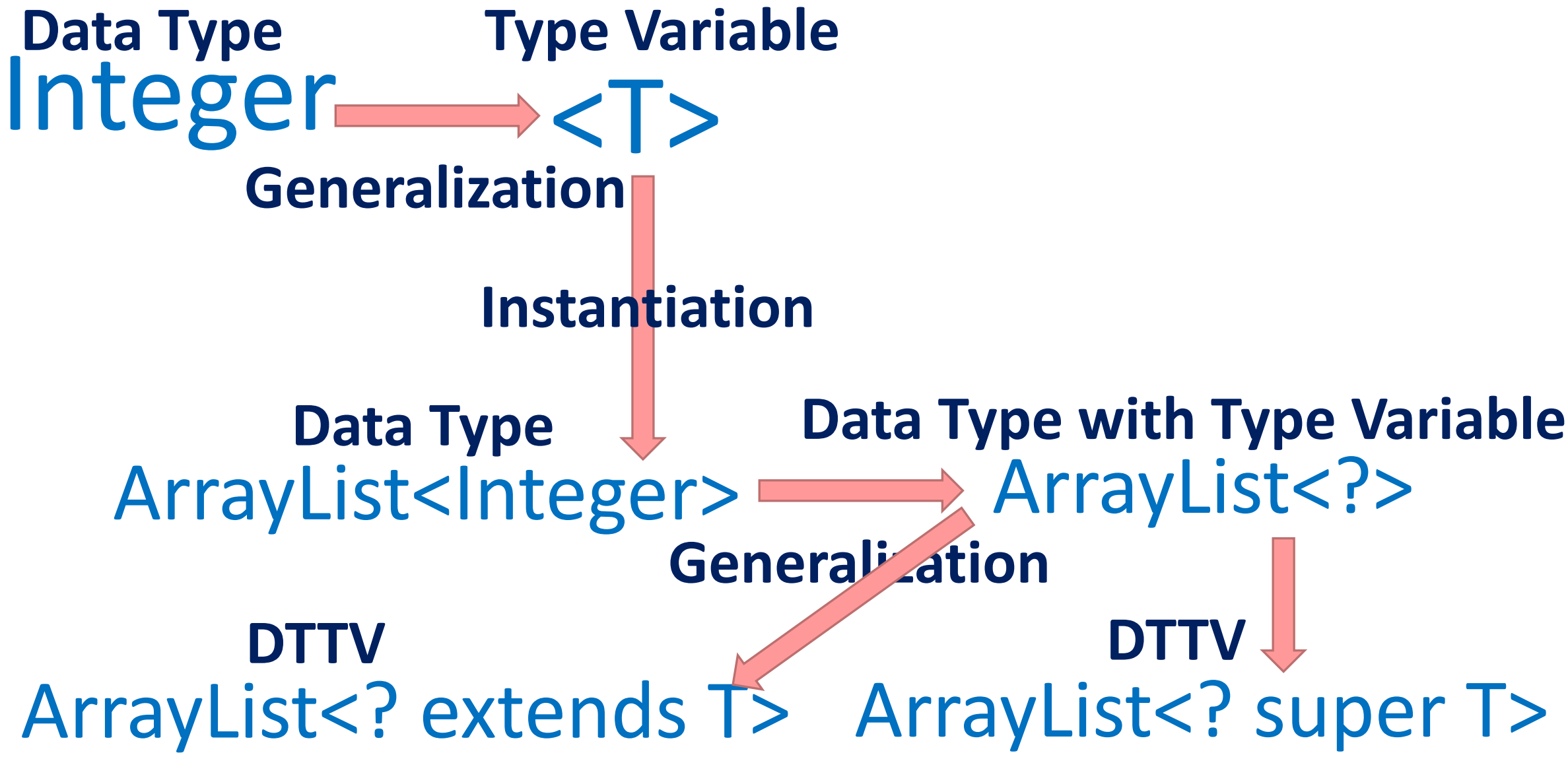
SECTION 8

Unbounded Wildcard

The first form, **?**, called an unbounded wildcard, is the same as **? extends Object**. 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**. You can fix the error by replacing line 12 in **WildCardNeedDemo.java** as follows:

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

<? extends Number> is a wildcard type that represents Number or a subtype of Number, so it is legal to invoke max(new GenericStack<Integer>()) or max(new GenericStack<Double>()).


Data Type
Type Variable
Integer
<T>
Generalization
Instantiation
Data Type
Data Type with Type Variable
ArrayList<Integer>
ArrayList<?>
Generalization
DTTV
DTTV
ArrayList<? extends T>
ArrayList<? super T>

Unbounded Wildcard

- **AnyWildCardDemo.java** shows an example of using the **?** wildcard in the print method that prints objects in a stack and empties the stack. **<?>** is a wildcard that represents any object type.
- It is equivalent to **<? extends Object>.** What happens if you replace **GenericStack<?>** with **GenericStack<Object>**? It would be wrong to **invoke print(intStack)**, because **intStack** is not an instance of **GenericStack<Object>**.
- Please note that **GenericStack<Integer>** is not a subtype of **GenericStack<Object>**, even though Integer is a subtype of Object.

Demonstration Program

ANYWILDCARDDEMO.JAVA

Super-wildcard Generics

SECTION 9

Super Wildcard

- When is the wildcard **<? super T>** needed? Consider the example in **SuperWildCardDemo.java**. The example creates a stack of strings in **stack1** (line 3) and a stack of objects in **stack2** (line 4), and invokes **add(stack1, stack2)** (line 8) to add the strings in **stack1** into **stack2**.
- **GenericStack<? super T>** is used to declare **stack2** in line 13. If **<? super T>** is replaced by **<T>**, a compile error will occur on **add(stack1, stack2)** in line 8, because **stack1**'s type is **GenericStack<String>** and **stack2**'s type is **GenericStack<Object>**. **<? super T>** represents type **T** or a supertype of **T**. Object is a supertype of **String**.

Wildcard Hierarchy

This program will also work if the method header in lines 12–13 is modified as follows:

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

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

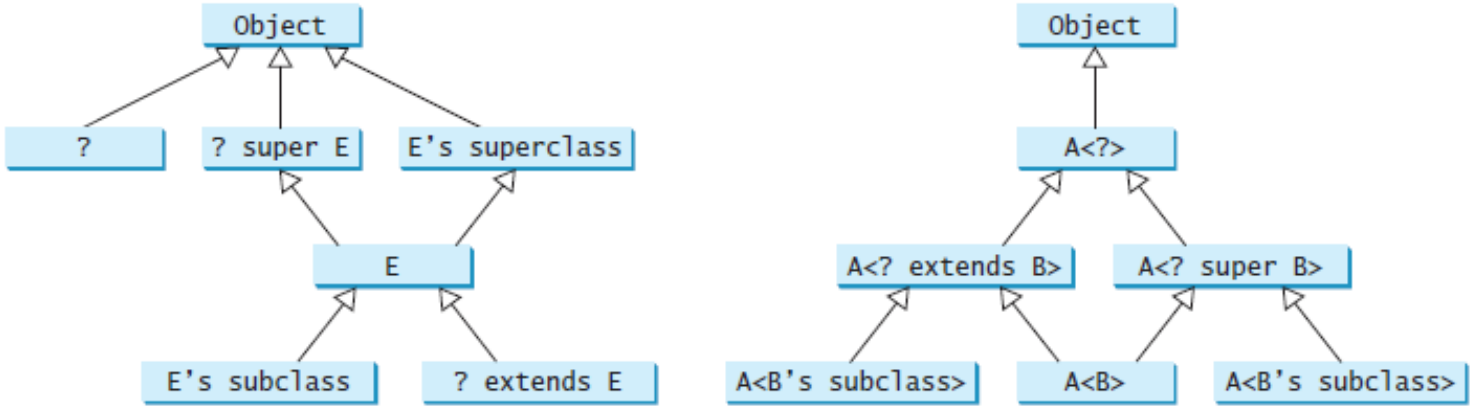


Figure D.

Demonstration Program

SUPERWILDCARDDEMO.JAVA

Case Study: Generic Matrix

SECTION 10

Case Study: Generic Matrix Class I

This lecture presents a case study on designing classes for matrix operations using generic types.

- The addition and multiplication operations for all matrices are similar except that their element types differ. Therefore, you can design a superclass that describes the common operations shared by matrices of all types regardless of their element types, and you can define subclasses tailored to specific types of matrices.

- This case study gives implementations for two types: **int** and **Rational**. For the **int** type, the wrapper class **Integer** should be used to wrap an **int** value into an object, so that the object is passed in the methods for operations.

Case Study: Generic Matrix Class II

This lecture presents a case study on designing classes for matrix operations using generic types.

- The class diagram is shown in Figure E. The methods **addMatrix** and **multiplyMatrix** add and multiply two matrices of a generic type **E[][]**. The static method **printResult** displays the matrices, the operator, and their result. The methods **add**, **multiply**, and **zero** are abstract, because their implementations depend on the specific type of the array elements. For example, the **zero()** method returns **0** for the **Integer** type and **0/1** for the **Rational** type. These methods will be implemented in the subclasses in which the matrix element type is specified.

UML GenericMatrix class

The GenericMatrix class is an abstract superclass for IntegerMatrix and RationalMatrix

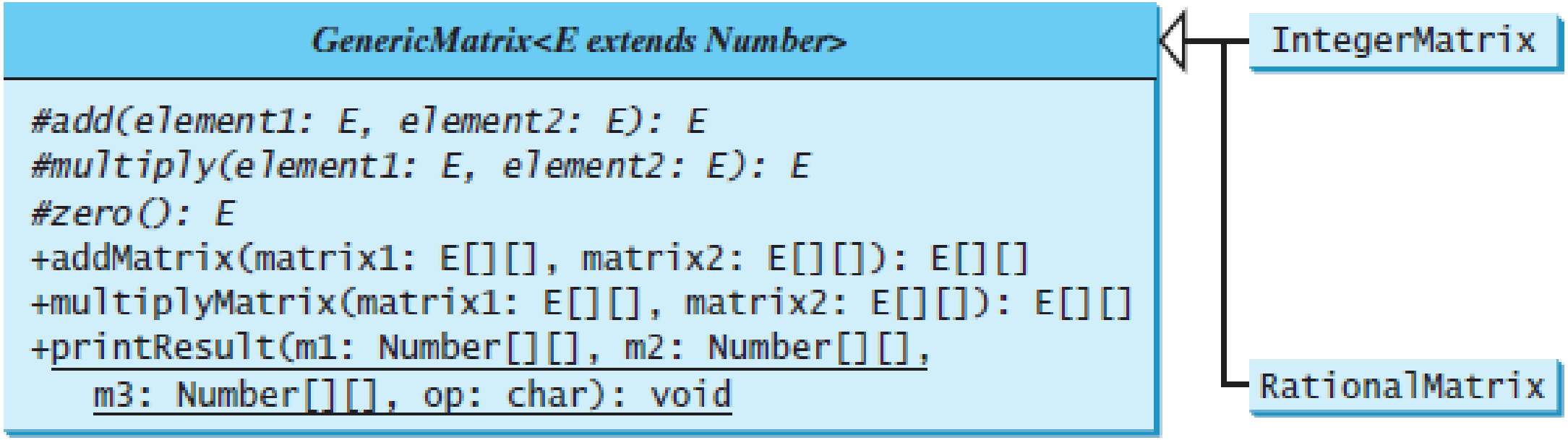


Figure E.

Case Study: Generic Matrix Class I

- **IntegerMatrix** and **RationalMatrix** are concrete subclasses of **GenericMatrix**. These two classes implement the **add**, **multiply**, and **zero** methods defined in the **GenericMatrix** class.
- **GenericMatrix.java** implements the **GenericMatrix** class. **<E extends Number>** in line 1 specifies that the generic type is a subtype of **Number**. Three abstract methods—**add**, **multiply**, and **zero**—are defined in lines 3, 6, and 9. These methods are abstract because we cannot implement them without knowing the exact type of the elements. The **addMaxtrix** (lines 12–30) and **multiplyMatrix** (lines 33–57) methods implement the methods for adding and multiplying two matrices. All these methods must be nonstatic, because they use generic type **E** for the class. The **printResult** method (lines 60–84) is static because it is not tied to specific instances.

Case Study: Generic Matrix Class II

- The matrix element type is a generic subtype of **Number**. This enables you to use an object of any subclass of **Number** as long as you can implement the abstract **add**, **multiply**, and **zero** methods in subclasses.
- The **addMatrix** and **multiplyMatrix** methods (lines 12–57) are concrete methods. They are ready to use as long as the **add**, **multiply**, and **zero** methods are implemented in the subclasses.
- The **addMatrix** and **multiplyMatrix** methods check the bounds of the matrices before performing operations. If the two matrices have incompatible bounds, the program throws an exception (lines 16, 36).

Case Study: Generic Matrix Class III

- IntegerMatrix.java implements the **IntegerMatrix** class. The class extends **GenericMatrix<Integer>** in line 1. After the generic instantiation, the **add** method in **GenericMatrix<Integer>** is now **Integer add(Integer o1, Integer o2)**. The **add**, **multiply**, and **zero** methods are implemented for **Integer** objects. These methods are still protected, because they are invoked only by the **addMatrix** and **multiplyMatrix** methods.

Case Study: Generic Matrix Class IV

- RationalMatrix.java implements the **RationalMatrix** class. The **Rational** class was introduced in Rational.java. **Rational** is a subtype of **Number**. The **RationalMatrix** class extends **GenericMatrix<Rational>** in line 1. After the generic instantiation, the **add** method in **GenericMatrix<Rational>** is now **Rational add(Rational r1, Rational r2)**. The **add**, **multiply**, and **zero** methods are implemented for **Rational** objects. These methods are still protected, because they are invoked only by the **addMatrix** and **multiplyMatrix** methods.

Case Study: Generic Matrix Class V

TestIntegerMatrix.java gives a program that creates two **Integer** matrices (lines 4–5) and an **IntegerMatrix** object (line 8), and adds and multiplies two matrices in lines 12 and 16.

TestRationalMatrix gives a program that creates two **Rational** matrices (lines 4–10) and a **RationalMatrix** object (line 13) and adds and multiplies two matrices in lines 17 and 19.

Demonstration Program

GENERICMATRIX.JAVA

Generic Iterator

SECTION 11

Generic Iterators

An Iterator is an object that will let you step through the elements of a list one at a time

- ```
  List<String> listOfStrings = new
  ArrayList<String>();
  ...
  for (Iterator i = listOfStrings.iterator();
  i.hasNext(); ) {
      String s = (String) i.next();
      System.out.println(s);
  }
  ```
  ```

Generic Iterators

- Iterators have also been genericized:
  - ```
    List<String> listOfStrings = new ArrayList<String>();
    ...
    for (Iterator<String> i = listOfStrings.iterator();
    i.hasNext(); ) {
        String s = i.next();
        System.out.println(s);
    }
    ```
- If a class implements `Iterable`, you can use the new for loop to iterate through all its objects
    ```

New `for` statement

The syntax of the new statement is

```
    for(type var : array) {...}
or  for(type var : collection) {...}
```

Example:

```
  for(float x : myRealArray) {
        myRealSum += x;
    }
```

New `for` statement

- For a collection class that implements `Iterable`, instead of

```
   for (Iterator iter = c.iterator();
iter.hasNext(); )
     ((TimerTask) iter.next()).cancel();
```

- you can now say

```
   for (TimerTask task : c)
      task.cancel();
```

New `for` statement with arrays

- The new `for` statement can also be used with arrays
- Instead of

```
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

- you can say (assuming `array` is an `int` array):

```
for (int value : array) {
    System.out.println(value);
}
```

Disadvantage: You don't know the index of any of your values

New `for` statement with arrays

- The new `for` statement can also be used with arrays of Integer
- Instead of

```
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

- you can say (assuming `array` is an `Integer` array):

```
for (Integer iObj : array) {
    System.out.println(iObj);
}
```

Disadvantage: If you update the value of the Integer iObj object, the iObj object is not part of the array. Instead, it is a separate copy.

Demonstration Program

ITERATORDEMO.JAVA

Generic Dynamic Array

SECTION 12

Generic Dynamic Array

https://www.techiedelight.com/creating-generic-array-java/

- Arrays in Java contains information about their component type for allocating memory during runtime. Now, if the component type is not know at the runtime, the array cannot be instantiated.

  Consider

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

- This uses Generics. We know that Generics are not present in the byte code generated by the compiler because of type erasure in Java. That means the Type information is erased at the runtime and new E[capacity] won't know that type needs to be instantiated. To avoid this behavior, we should use List provided by Java Collections Framework wherever we need generics.

1. Using Object Array

```
class Main{
    // Program to create a generic array in Java
    public static void main(String[] args){
        final int length = 5;
        // create an Integer array of given length
        Array<Integer> intArray = new Array(length);

        for (int i = 0; i < length; i++)
            intArray.set(i, i + 1);

        System.out.println(intArray);
        // create a String array of given length
        Array<String> strArray = new Array(length);
        for (int i = 0; i < length; i++)
            strArray.set(i, String.valueOf((char)(i + 65)));

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

```
import java.util.Arrays;
class Array<E> {
    private final Object[] arr;
    public final int length;
    // constructor
    public Array(int length){
        // Creates a new Object array of specified length
        arr = new Object[length];
        this.length = length;
    }
    // Function to get Object present at index i in the array
    E get(int i) {
        @SuppressWarnings("unchecked")
        final E e = (E)arr[i];
        return e;
    }
    // Function to set a value e at index i in the array
    void set(int i, E e) {
        arr[i] = e;
    }
    @Override
    public String toString() {
        return Arrays.toString(arr);
    }
}
```

2. Using Reflection

We can use the Reflection Array class to create an array of a generic type known only at runtime. Please note that unlike previous approach, here we're explicitly passing the Type information to the class constructor, which further being passed to the **Array.newInstance().**

2. Using Reflection

```
import java.util.Arrays;
class Array<E> {
	private final E[] arr;
	public final int length;
	// constructor
	public Array(Class<E> type, int length) {
		// Creates a new array with the specified type and length at runtime
		this.arr = (E[]) java.lang.reflect.Array.newInstance(type, length);
		this.length = length;
	}
	// Function to get element present at index i in the array
	E get(int i) {
		return arr[i];
	}
	// Function to set a value e at index i in the array
	void set(int i, E e) {
		arr[i] = e;
	}
	@Override
	public String toString() {
		return Arrays.toString(arr);
	}
}
```

2. Using Reflection

```
class Main{
	// Program to create a generic array in Java
	public static void main(String[] args){
		final int length = 5;
		// create an Integer array of given length
		Array<Integer> intArray = new Array(Integer.class, length);
		for (int i = 0; i < length; i++)
			intArray.set(i, i + 1);
		System.out.println(intArray);
		// create a String array of given length
		Array<String> strArray = new Array(String.class, length);

		for (int i = 0; i < length; i++)
			strArray.set(i, String.valueOf((char)(i + 65)));
		System.out.println(strArray);
	}
}
```