1. 泛型（Generics）

我们先看一段代码。

```
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, but Runtime
error because "1" is a String
    }
}
```

这里的代码会报错，因为无法强行将 String 转换为 Integer，但这里是运行错误，而不是编译错误。但其实最好是出现编译错误，而不是运行错误！因为只有这样在代码部署前我们才能发现错误，不像运行错误可能会导致程序崩溃或行为异常，难以调试和修复。

我们可以使用泛型编程解决这个问题。

```
public class ShowCompilerError {
    public static void main(String[] args) {
        java.util.ArrayList<Integer> list = new java.util.ArrayList<Integer>();
        // list.add("1"); // Compiler error on this line
        list.add(new Integer(1)); // No error here
    }
}
```

这里创建了一个泛型化的 ArrayList，指定了其存储的元素类型为 Integer。这意味着列表中只能存储 Integer 类型的对象。
如果尝试向列表中添加一个字符串 "1"。由于泛型指定了列表只能存储 Integer 类型的对象，因此编译器会报错，指出类型不匹配。
如果尝试列表中添加了一个 Integer 对象，符合泛型的要求，因此不会报错。
但是这里最后一行会有红色的报错，但是我们可以正常运行，这里会提示你这个语句已经不推荐使用将在未来的版本中被移除。

1.1 泛型的定义

泛型允许程序员定义类和方法时使用类型参数。这些类型参数可以在编译时被具体的类型替换，从而提供类型安全和代码重用。
假设我们定义了一个泛型栈类 Stack<T>，其中 T 是一个类型参数。使用这个泛型类，我们可以创建不同类型元素的栈对象。
1.创建一个用于存储字符串的栈。
示例如下。

```
Stack<String> stringStack = new Stack<>();
stringStack.push("Hello");
stringStack.push("World");
```

2.创建一个用于存储数字的栈。
示例如下。

```
Stack<Integer> integerStack = new Stack<>();
integerStack.push(1);
integerStack.push(2);
```

在这两个例子中，String 和 Integer 是具体的类型，它们替换了泛型类 Stack<\T> 中的类型参数 T。这样，你就可以创建专门用于存储字符串或整数的栈对象，而不需要为每种类型编写不同的类。

1.2 泛型的优点

1. 能够在编译时检测错误，而不是在运行时。
2. 泛型类或方法指定类或方法可以处理的对象的允许类型。
3. 重用代码。例如，为一种特殊类型的数据结构编写单一实现，如泛型栈及其标准方法的单一实现。
4. 使用泛型可以避免在代码中进行强制类型转换，使得代码更加清晰和易于维护。后面将阐述这一点。

当然最重要的优点是如果尝试使用不兼容的对象使用类或方法，将发生编译错误。正如前面例子展示的一样。
我们再看一个泛型的优势。
在JDK1.5之前在Comparable接口的定义中，它有一个compareTo方法，该方法接受一个Object类型的

参数。
这种方式在编译时不会检查类型，因此可能会导致运行时错误（Runtime error）。

```
package java.lang;
public interface Comparable {
    public int compareTo(Object o);
}

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

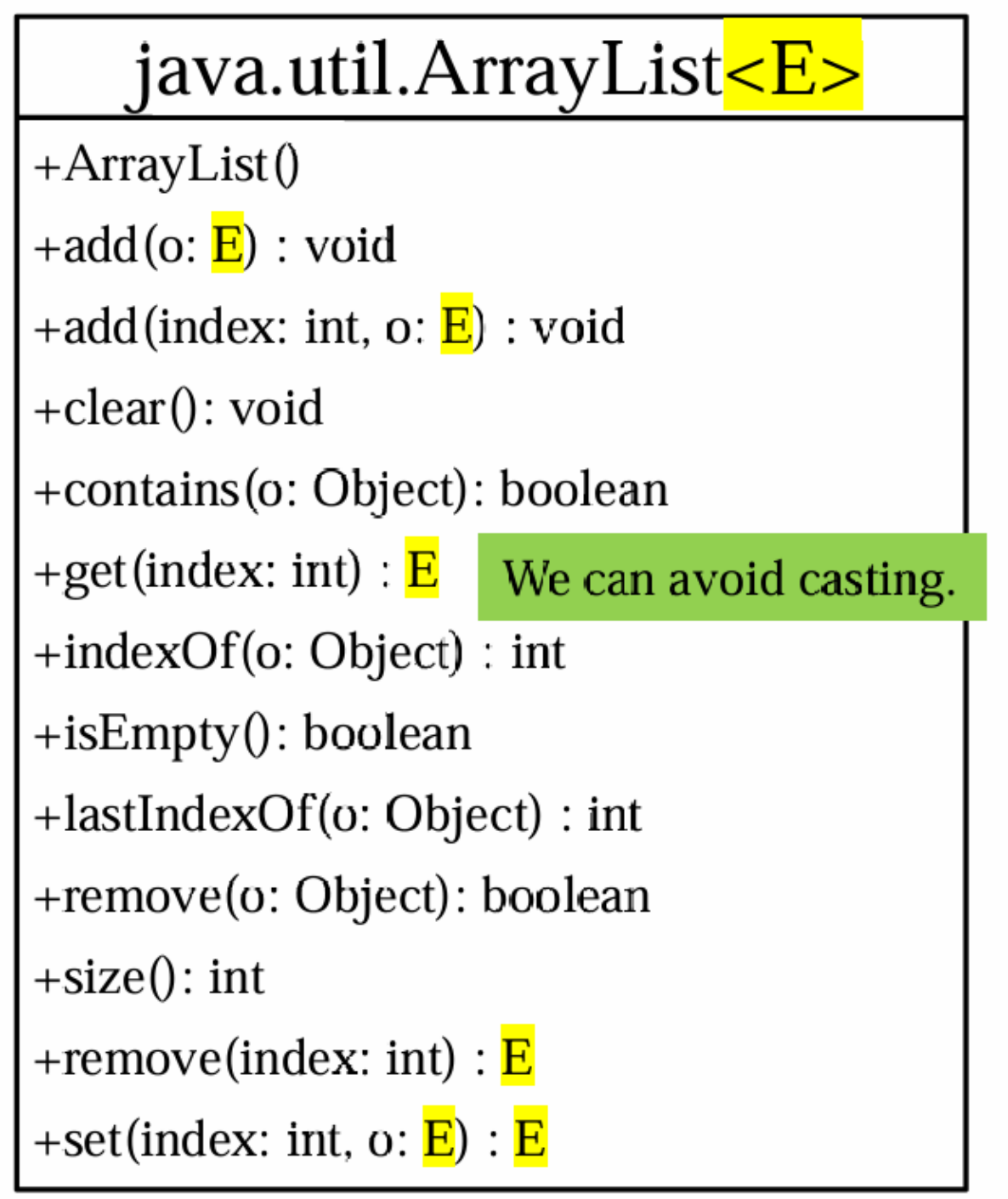
在JDK 1.5及以后版本中，Comparable接口被定义为泛型接口Comparable<T>。这种方式在编译时会检查类型，如果类型不匹配，编译器会报错（Compiler error），从而提高了代码的可靠性。

```
package java.lang;
public interface Comparable<T> {
    public int compareTo(T o);
}

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

所以上面的代码运行的时候我们才知道错误，下面的代码我们运行之前就可以发现错误。

下图展示了JDK1.5版本之后的 ArrayList 类。



由于之前的版本，没有泛型，所以是使用Object类型来存储所有元素的，所以在从集合中取出元素时，往往需要进行类型转换，以确保取出的元素是期望的类型。
如下所示。

```
ArrayList list = new ArrayList();
list.add("Hello");
list.add(123);

Object obj = list.get(0); // 获取到的是Object类型
String str = (String) obj; // 需要进行类型转换
```

而现在就可以避免这一点。

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
// list.add(123); // 这行代码会导致编译错误，因为123不是String类型

String str = list.get(0); // 直接获取String类型，无需类型转换
```

1.3 类型限制

泛型类型必须是引用类型（reference types），不能是基本数据类型（primitive types）。
泛型类型不能被替换为基本数据类型，如 int、double 或 char。
下列的代码会报错。

```
ArrayList<int> intList = new ArrayList<>(); // 这是错误的，会导致编译错误
```

正确做法是使用基本数据类型的包装类（wrapper types），如 Integer、Double 或 Character。
如下所示。

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

得益于装箱（Boxing）机制，Java可以将基本数据类型自动转换为其对应的包装类。
如下所示。

```
intList.add(1); // Java自动将1装箱为：new Integer(1)
```

Java会自动将 1 装箱为 Integer 对象 new Integer(1)，然后添加到列表中。

因此当我们获取元素是，如果元素是包装类类型，不需要显式地进行类型转换。
如下所示。

```
ArrayList<Double> list = new ArrayList<Double>();
list.add(5.5); // 5.5自动装箱为new Double(5.5)
list.add(3.0); // 3.0自动装箱为new Double(3.0)
Double doubleObject = list.get(0); // 直接获取Double对象，无需类型转换
```

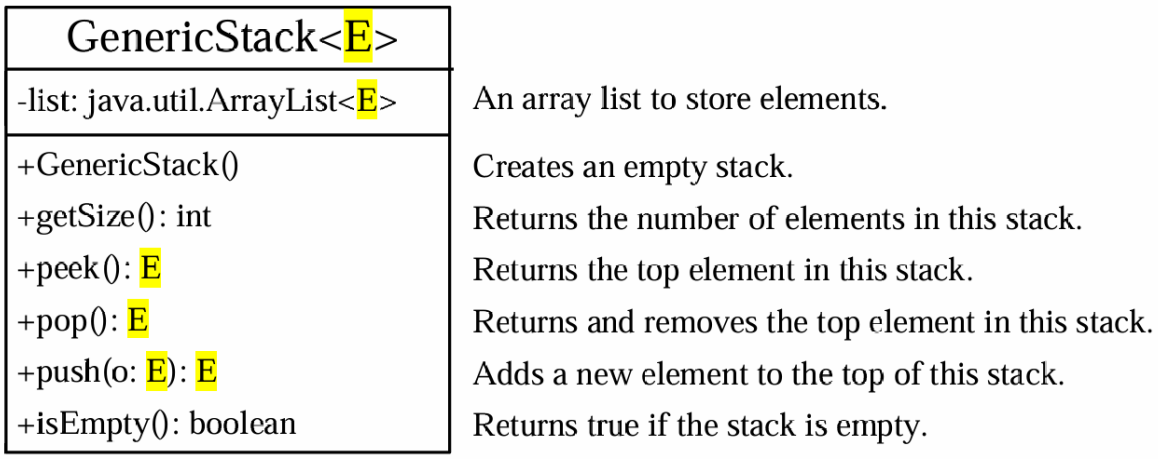
当然也支持拆箱（Unboxing）操作，从而将包装类自动转换为对应的基本数据类型。
如下所示。

```
double d = list.get(1); // 自动拆箱：自动转换为double
```

1.4 实践

1.4.1 GenericStack 类

我们可以设计一个支持泛型的 Stack 类。
下图展示了其的URL。



代码如下。

```
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 detect override errors
    public String toString() {
        return "stack: " + list.toString();
    }
}
```

测试代码如下。

```
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);
}
```

1.5 泛型静态方法的声明

在方法的声明中，泛型类型参数需要放在方法修饰符（如 public 和 static）之后，紧跟在方法名之前。

```
public class GenericMethods1 {
    // 声明一个泛型静态方法
    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) {
        // 创建一个字符串数组
        String[] s3 = {"Hello", "again"};

        // 调用泛型方法，显式指定类型参数
        GenericMethods1.<String>print(s3);

        // 或者简单地调用泛型方法，编译器会自动推断类型参数
        print(s3);
    }
}
```

1.6 有界泛型（Bounded Generic Types）

有界泛型是指在定义泛型时，可以指定泛型类型参数的上界，即泛型类型参数必须是指定类或其子类的实例。
如前面我们经常使用的GeometricObject下面有Circle和Rectangle类。对这个例子我们有如下代码。

```
public class GenericMethods2 {
    // 定义一个有界泛型方法，E 必须是 GeometricObject 或其子类的实例
    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));
    }
}
```

1.4.2 数组排序

基于上面一点我们可以有如下实践。

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

方法 genericSelectionSort 只能接受实现了 Comparable 接口的对象数组作为参数，从而完成排序。测试代码如下。

```
    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();
    }
```

我们再看一下这个代码，其实这里的泛型类型参数 <E extends Comparable<E>>有两个含义：

1. E 必须是 Comparable 接口的实现者。
2. 确保了在比较操作中使用的元素都是同一类型 E。

由于 Integer、Double、Character 和 String 等类都实现了 Comparable 接口，所以这些类的对象可以通过 compareTo 方法进行比较，因此可以使用 genericSelectionSort 方法对这些类的对象数组进行排序。

1.7 原始类型（Raw Type）和向后兼容性（Backward Compatibility）

在Java中，如果使用泛型类或接口时没有指定具体的类型参数，这种类型被称为原始类型（Raw Type）。原始类型允许Java程序与Java泛型出现之前的版本兼容，这便是向后兼容性（Backward Compatibility）。也就是说，即使在引入泛型之后的版本中，不使用泛型的旧代码仍然可以正常编译和运行。
在JDK 1.5及更高版本中，声明一个原始类型的变量大致等同于声明一个指定了Object类型参数的泛型变量。
如下面的两串代码是几乎等价的。

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

因此，所有在旧版本的JDK中编写的程序在新版本中仍然是可执行的。
当较旧的技术或程序能够无需修改或只需很少修改就能在更新的技术或系统中使用时，就称新技术具有向后兼容性。在Java的例子中，新版本通过引入原始类型的概念保持了对旧代码的兼容性。
例如Python 3.X版本在设计时没有保持与Python 2.X版本的向后兼容性，导致一些在Python 2中可以正常工作的代码在Python 3中不再适用。
在Python 2中，print 1 是有效的代码，会输出数字1。
但在Python 3中，这种语法不再有效，因为Python 3要求使用函数语法 print(1) 来进行输出。

1.7.1 原始类型的不安全性

由于之前的代码没有使用泛型，所以它们很可能会导致不安全性。
如下面的代码所示。

```
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));
    }
}
```

在这个例子中，Comparable o1 和 Comparable o2 是原始类型声明，所以这些原始类型在编译时被视为 Object 类型，而 Object 类型可以接受任何类型的对象。
但是在运行时会报错，因为字符串 "Welcome" 和整数 23 不是同一类型，不能直接进行比较。

因此我们回到前面我们所说的泛型的优点，我们利用泛型就可以解决这里的不安全性。
下面的代码修复了上面例子中的问题。

```
public class Safe {
    // Return the maximum between two objects of the same type
    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) {
        // 下面的调用现在会导致编译错误，因为两个参数必须具有相同的类型
        // System.out.println(max2("Welcome", 23));

        // 正确的调用方式，使用相同类型的参数
        System.out.println(max2("Welcome", "World"));
        System.out.println(max2(1, 2));
    }
}
```

通过这种方式，编译器会在编译时检查传递给 max2 方法的两个参数是否为同一类型，并且该类型是否实现了 Comparable 接口。从而避免了不安全性，将运行错误变成了编译错误。

1.8 通配符（Wildcards）

通配符（Wildcards）用于解决泛型中的一些类型限制问题。其可以允许我们在不知道具体类型的情况下，对一组类型进行操作。
Integer 是 Number 的子类，但是 GenericStack<Integer> 并不是 GenericStack<Number> 的子类。
这意味着你不能将 GenericStack<Integer> 赋值给 GenericStack<Number> 类型的变量，即使 Integer 是 Number 的子类。
这里介绍通配符的三种类型：

1. 无界通配符（Unbounded Wildcard）：? 表示任何类。例如，List<?> 表示一个可以包含任何类型元素的 List。

2. 有界通配符（Bounded Wildcard）：? extends T 表示任何 T 的子类（包括 T 本身）。例如，List<? extends Number> 表示一个可以包含任何 Number 子类元素的 List。
3. 下界通配符（Lower Bound Wildcard）：? super T 表示任何 T 的超类（包括 T 本身）。例如，List<? super Number> 表示一个可以接受任何 Number 类型及其超类型（如 Object）作为元素的 List。

所以我们可以在通配符的帮助下，让 GenericStack<Integer> 成为 GenericStack<? extends Number> 的子类，当然能这么做也是因为 Integer 是 Number 的一个子类。
这种关系允许我们能够编写更灵活的代码。
下面给出一些例子。
例1：

```
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>
    }
}
```

在这里由于 GenericStack<Integer> 不是 GenericStack<Number> 的子类型，这将导致编译错误。

```
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));
    }
}
```

在这个版本中，max 方法使用了有界通配符 <? extends Number>，这意味着它可以接受任何 Number 类型或其子类型的 GenericStack。因此，传递一个 GenericStack<Integer> 类型的参数给 max 方法是允许的，因为 Integer 是 Number 的子类。这解决了之前版本中的编译错误问题。
可以成功输出结果。

```
The max number is 2.0
```

例2：

```
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);
    }
}
```

结果如下。

```
-2 2 1
```

例3：

```
public class WildCardDemo3 {
    // Add stack1 TO stack2: the type of elements 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);
    }
}
```

这里 add 方法将 stack1 中的所有元素添加到 stack2 中，要求 stack1 中元素的类型是 stack2 中元素类型的子类型或和 stack2 元素类型一致。
结果如下。

```
Sun 2 Java
```

例4:

```
public class WildCardDemo4 {
    // Add stack1 TO stack2: the type of elements 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);
    }
}
```

这里 add 方法将 stack1 中的所有元素添加到 stack2 中，要求 stack2 中元素的类型是 stack1 中元素类型的超类型或和 stack1 元素类型一致。
结果如下。

```
Sun 2 Java
```

1.9 类型擦除（Type Erasure）

类型擦除是Java实现泛型的一种机制。在这种机制下，泛型类型信息在编译时被用来检查类型安全，但在编译完成后，这些类型信息会被“擦除”，即泛型类型参数会被替换为它们的上界（对于无界泛型，上界是 Object）。
这种机制使得泛型代码能够与Java早期版本中使用的原始类型（如 ArrayList 而不是 ArrayList<String>）保持向后兼容性。
所以下面的代码是泛型代码。

```
ArrayList<String> list = new ArrayList<String>();
list.add("Oklahoma");
String state = list.get(0);
```

在编译完成后，这串代码等效于下面的代码。

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)list.get(0);
```

下面出示更多的例子。
例1：

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

编译后的类型擦除后运行的版本如下。

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

例2：

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

编译后的类型擦除后运行的版本如下。

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

1.9.1 泛型类

泛型类在JVM中是以原始类型（非泛型）的形式存在的。这意味着，尽管你可以创建泛型类的不同实例，如 GenericStack<String> 和 GenericStack<Integer>，但它们在JVM中都被视为同一个类 GenericStack。

```
GenericStack<String> stack1 = new GenericStack<String>();
GenericStack<Integer> stack2 = new GenericStack<Integer>();
```

因此当创建 GenericStack<String> 和 GenericStack<Integer> 的实例时，它们共享同一个类 GenericStack，而不是创建两个不同的类。

所以正确的类型检查代码如下。

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

下面的代码会出现编译错误。

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

GenericStack<String> 和 GenericStack<Integer> 在JVM中并不是作为独立的类存在，它们只是 GenericStack 类的泛型实例。

1.9.2 泛型的限制

由于Java泛型在运行时进行类型擦除（Erasure），导致在使用泛型时存在的一些限制。

1. 不能创建泛型类型的实例
   问题：不能使用 new E() 这样的语法来创建泛型类型 E 的实例。例如，E object = new E(); 是错误的。
   原因：new E() 这种实例化操作是在运行时执行的，但在运行时，泛型类型 E 的信息已经被擦除，不再可用。因此，编译器无法在运行时确定 E 的具体类型，这会导致编译错误。
2. 不能创建泛型数组，如 new E[100]。例如，E[] elements = new E[capacity]; 是错误的。
   原因：与创建泛型类型的实例类似，泛型数组的创建也是在运行时执行的。由于泛型类型信息在运行时不可用，编译器无法确定数组的具体类型，这同样会导致编译错误。
3. 在静态方法、静态字段或初始化器中引用泛型类型参数是非法的。
   原因：泛型类的所有实例在运行时共享相同的类，因此泛型类中的静态变量和方法也是共享的，而泛型类型参数在运行时是擦除的，因此无法确定具体的类型，这同样会导致编译错误。
   如下面的代码都是报错的。

```
public class DataStructure<E> {
    // 错误的静态方法声明
    public static void m(E o1) { // Illegal
    }
    public static E o1; // Illegal
    static {
        E o2; // Illegal
    }
}
```

4. 不能声明一个泛型异常类，例如，public class MyException extends Exception { ... }

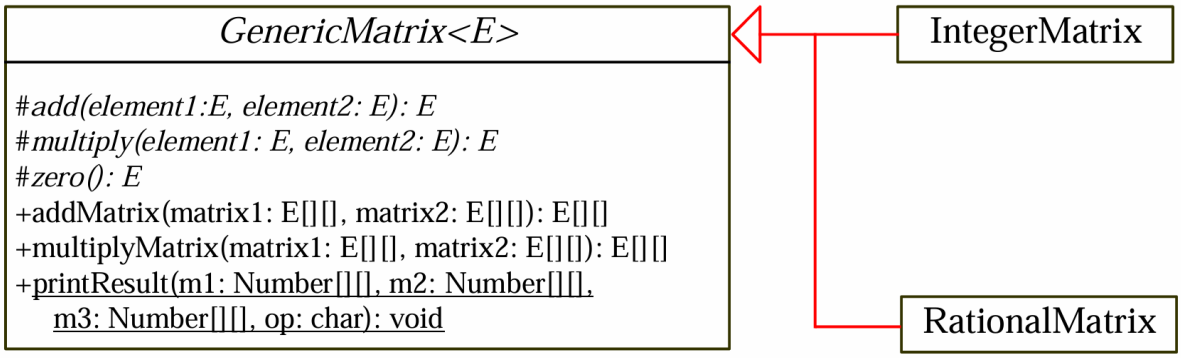
如果允许泛型异常类，你可能会尝试捕获一个泛型异常，例如：

```
try {
    ...
} catch (MyException<T> ex) {
    ...
}
```

这种捕获语句在编译时是合法的，但在运行时会遇到问题。
在运行时，Java虚拟机（JVM）需要检查从 try 块中抛出的异常是否与 catch 块中指定的异常类型匹配。由于泛型类型信息在运行时被擦除，JVM无法在运行时确定泛型异常的具体类型。这意味着JVM无法正确地匹配泛型异常类型，从而导致运行时错误。

1.4.3 GenericMatrix 类

我们尝试设计一个泛型类GenericMatrix<E> ，用于矩阵运算。
其UML图如下。



具体实现的代码如下。

```
// 定义 GenericMatrix 抽象类
public abstract class GenericMatrix<E extends Number> {
    protected abstract E add(E o1, E o2);
    protected abstract E multiply(E o1, E o2);
    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) {
        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(E[][] m1, E[][] m2, E[][] 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), '*');
   }
}
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), '*');
}
```

2. 练习

2.1 洗牌算法（泛型版）

修改上次的洗牌算法，使其能够处理泛型对象的 ArrayList。
示例代码如下。

```
import java.util.ArrayList;
import java.util.Random;

public class ShuffleArrayList {
    public static void main(String[] args) {
        // 创建一个包含数字的ArrayList
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);
        list.add(7);
        list.add(8);
        list.add(9);
        list.add(10);

        // 打印原始列表
        System.out.println("Original list: " + list);

        // 调用shuffle方法打乱列表
        shuffle(list);

        // 打印打乱后的列表
        System.out.println("Shuffled list: " + list);
    }
```

```
    public static <E> void shuffle(ArrayList<E> list) {
        Random random = new Random(); // 创建一个随机数生成器

        // Fisher-Yates洗牌算法
        for (int i = list.size() - 1; i > 0; i--) {
            // 生成一个从0到i（包含i）的随机索引
            int j = random.nextInt(i + 1);

            // 交换索引i和j处的元素
            E temp = list.get(i);
            list.set(i, list.get(j));
            list.set(j, temp);
        }
    }
}
```

2.2 ArrayList 中的最大元素

编写一个泛型方法，该方法能够找出 ArrayList 中的最大元素。
方法签名如下。

```
public static <E extends Comparable<E>> E max(ArrayList<E> list)
```

示例代码如下。

```
import java.util.ArrayList;

public class MaxElementFinder {
    public static void main(String[] args) {
        // 创建一个包含Integer对象的ArrayList
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(20);
        list.add(5);
        list.add(30);
        list.add(15);

        // 调用max方法找出最大元素
        Integer maxElement = max(list);

        // 打印最大元素
        System.out.println("The maximum element is: " + maxElement);
    }

    public static <E extends Comparable<E>> E max(ArrayList<E> list) {
        if (list == null || list.isEmpty()) {
            throw new IllegalArgumentException("List must not be null or
empty");
        }

        // 假设第一个元素是最大的
        E maxElement = list.get(0);
```

```
        // 遍历列表中的所有元素
        for (E element : list) {
            // 如果找到更大的元素，更新maxElement
            if (element.compareTo(maxElement) > 0) {
                maxElement = element;
            }
        }

        // 返回最大元素
        return maxElement;
    }
}
```

2.3 Pair 类

实现一个 Pair 类，它包含两个泛型元素 first 和 second，这两个元素必须是相同的类型 E。同时，你需要开发一个 print() 方法，用于在一行中输出这两个元素。
示例代码如下。

```
public class Pair<E> {
    public E first; // 第一个元素
    public E second; // 第二个元素

    // 构造函数：初始化一个包含两个元素的Pair
    public Pair(E e, E f) {
        first = e;
        second = f;
    }

    public static void main(String[] args) {
        Pair<Integer> p1 = new Pair<>(1, 85);
        Pair<Integer> p2 = new Pair<>(2, 63);
        print(p1);
        print(p2);
    }

    public static void print(Pair p) {
        System.out.println(p.first + " " + p.second);
    }
}
```

现在要修改这里的代码，使其能够存储不同类型的对象。例如，一个 Pair 可以同时存储一个 Integer 和一个 Double，或者一个 Integer 和一个 String。`

```
public class Pair<E, F> {
    public E first; // 第一个元素
    public F second; // 第二个元素

    // 构造函数：初始化一个包含两个不同类型元素的Pair
    public Pair(E e, F f) {
        first = e;
        second = f;
    }
```

```
    public static void main(String[] args) {
        Pair<Integer, Double> p1 = new Pair<>(1, 85.5);
        Pair<Integer, String> p2 = new Pair<>(2, "good");
        print(p1);
        print(p2);
    }

    public static void print(Pair p) {
        System.out.println(p.first + " " + p.second);
    }
}
```

测试代码如下。

```
public static void main(String[] args) {
    Pair<Integer, Double> p1 = new Pair<>(1, 85.5);
    Pair<Integer, String> p2 = new Pair<>(2, "good");
    print(p1);
    print(p2);
}
```

2.4 类型通配符

下面的代码出示了一个 print 方法，设计用来打印 ArrayList 中的所有元素。这里的代码有问题，需要解释代码不能编译的原因，以及修正方案。

```
import java.util.*;

public class week4test1{
    public static void main(String[] args) {
        ArrayList<Integer> c = new ArrayList<>();
        c.add(3);
        c.add(4);
        c.add(12);
        print(c);
    }
    public static void print(ArrayList<Object> o){
        for (Object e : o)
            System.out.println(e);
    }
}
```

尽管 Integer 是 Object 的子类，但是ArrayList<Integer> 不是 ArrayList<Object> 的子类型。因此，直接将 ArrayList<Integer> 传递给期望 ArrayList<Object> 的方法会导致编译错误。
为了解决这个问题，可以使用通配符？来表示 print 方法可以接受任何类型的 ArrayList。
修正方案如下。

```
import java.util.*;

public class week4test1{
    public static void main(String[] args) {
        ArrayList<Integer> c = new ArrayList<>();
        c.add(3);
```

```
        c.add(4);
        c.add(12);
        print(c);
    }
    public static void print(ArrayList<? extends Object> o){
        for (Object e : o)
            System.out.println(e);
    }
}
```