PROGRAMMING AND DATA STRUCTURES

GENERICS (TEMPLATES)

HOURIA OUDGHIRI

FALL 2023

OUTLINE

- What are generics?
- Using generic classes (`ArrayList`) and generic methods (`sort`)
- Creating generic classes
- Defining generic methods
- Restrictions on generic types

STUDENT LEARNING OUTCOMES

At the end of this chapter, you should be able to:

- Use generic classes, generic interfaces, and generic methods from the Java API
- Implement your own generic classes and generic methods

GENERICS

- Generics allow to specify a range of types allowable for a class, an interface, or a method

- Used to create classes that hold data of different types

- Used to create methods that accept parameters of different types

GENERICS

- interface **Comparable<E>** can be implemented for any reference type E

```
interface Comparable<E> {
  int compareTo(E obj);
}
```

GENERICS

- Generic Class - Class of type **`<E>`**
- **`E`** is the type parameter or generic type
- **`E`** can be replaced by any reference type (**`String`**, **`Integer`**, or **`Student`**)

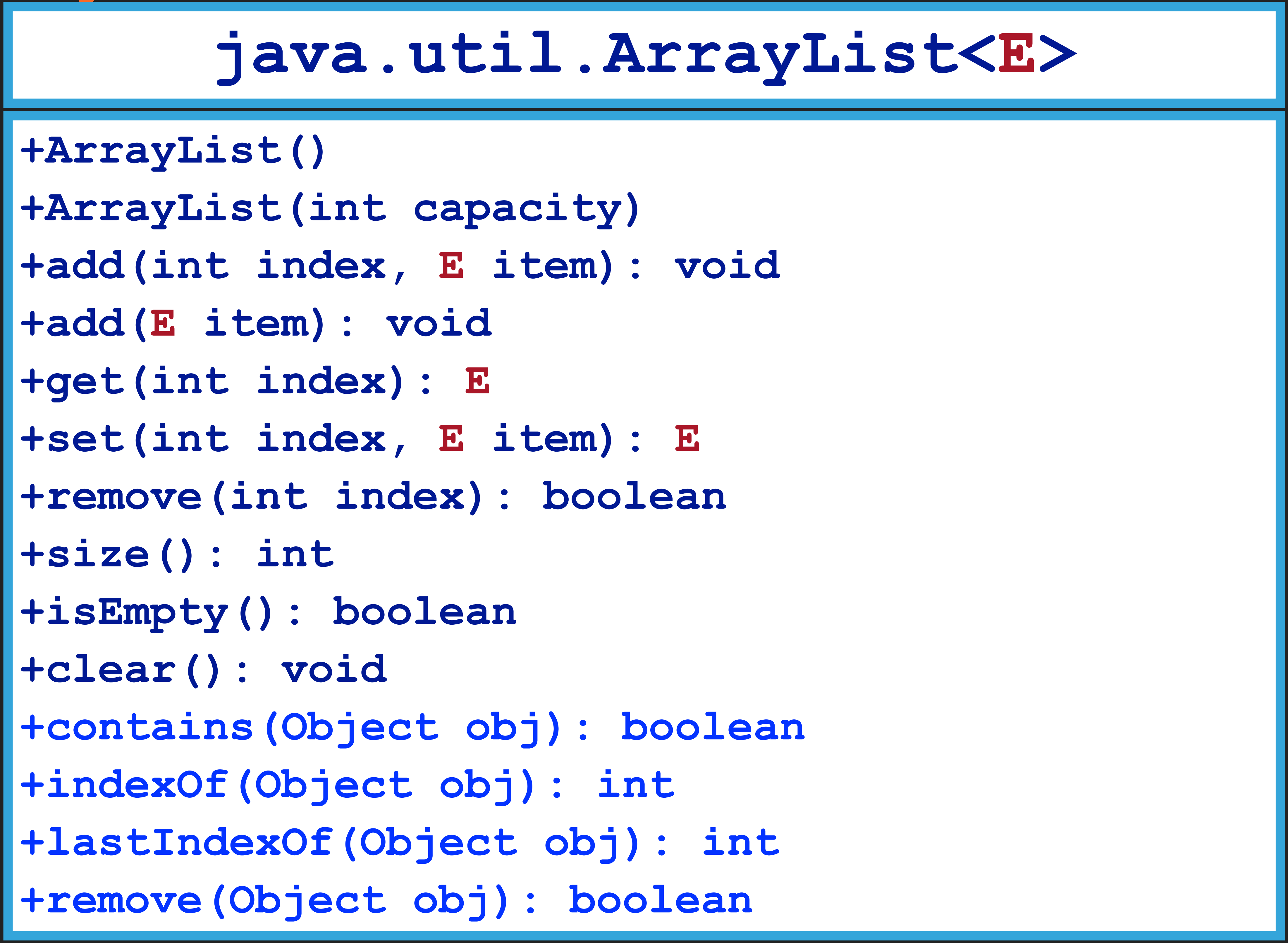
GENERICS

- Primitive types are not allowed as generic type parameters (`int`, `double`, `char`, ...) [use `Integer`, `Double`, `Character`, ...]
- Can use any name for the generic type (between `<>`) but the most commonly used is `<E>` or `<T>`

Generic Class - `java.util.ArrayList`

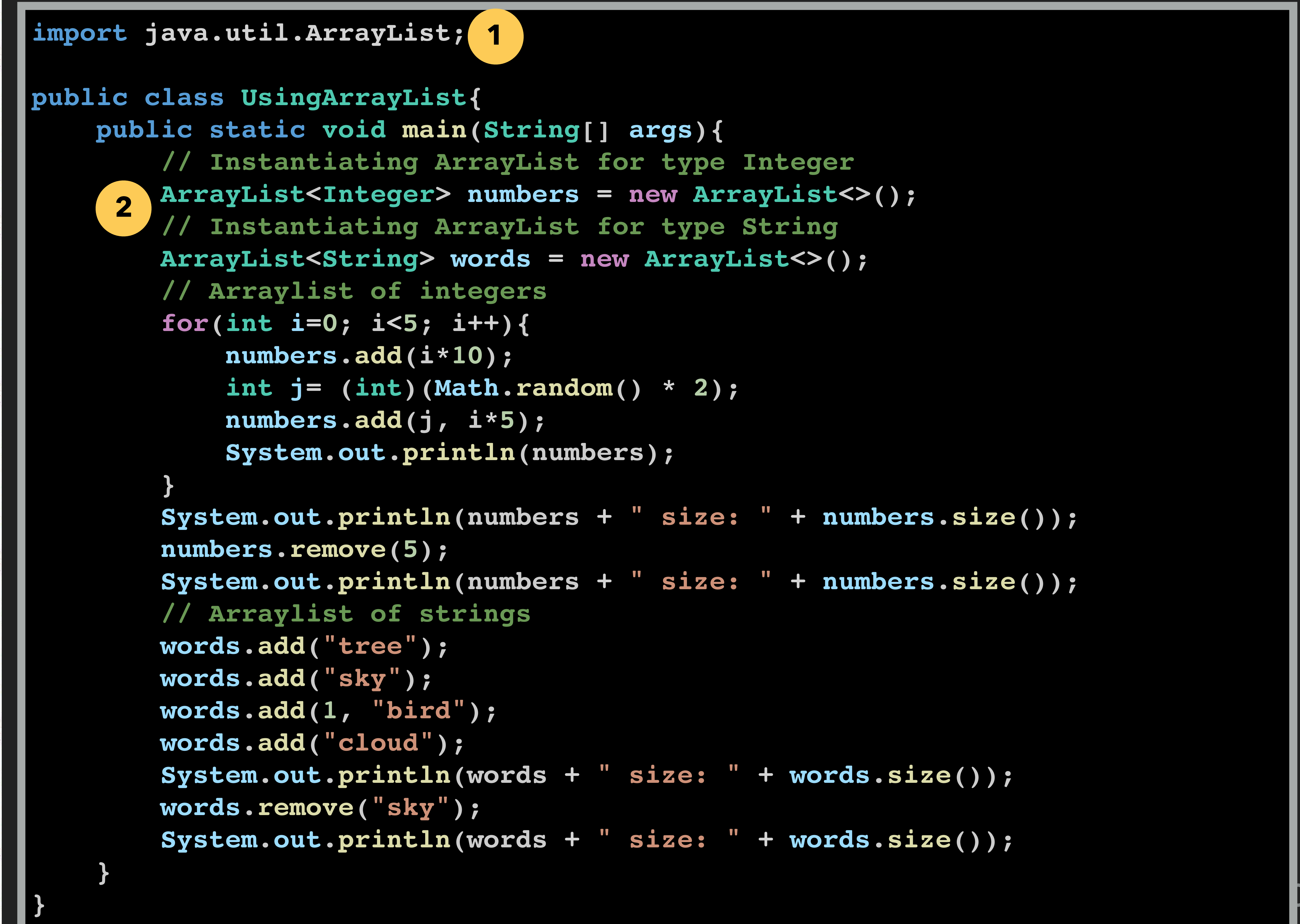
- Array of objects of any type
- Array of any size
- The size of the array may increase or decrease at runtime
- Like a Wrapper class for Arrays

ArrayList



```
java.util.ArrayList<E>

+ArrayList()
+ArrayList(int capacity)
+add(int index, E item): void
+add(E item): void
+get(int index): E
+set(int index, E item): E
+remove(int index): boolean
+size(): int
+isEmpty(): boolean
+clear(): void
+contains(Object obj): boolean
+indexOf(Object obj): int
+lastIndexOf(Object obj): int
+remove(Object obj): boolean
```

GENERICS



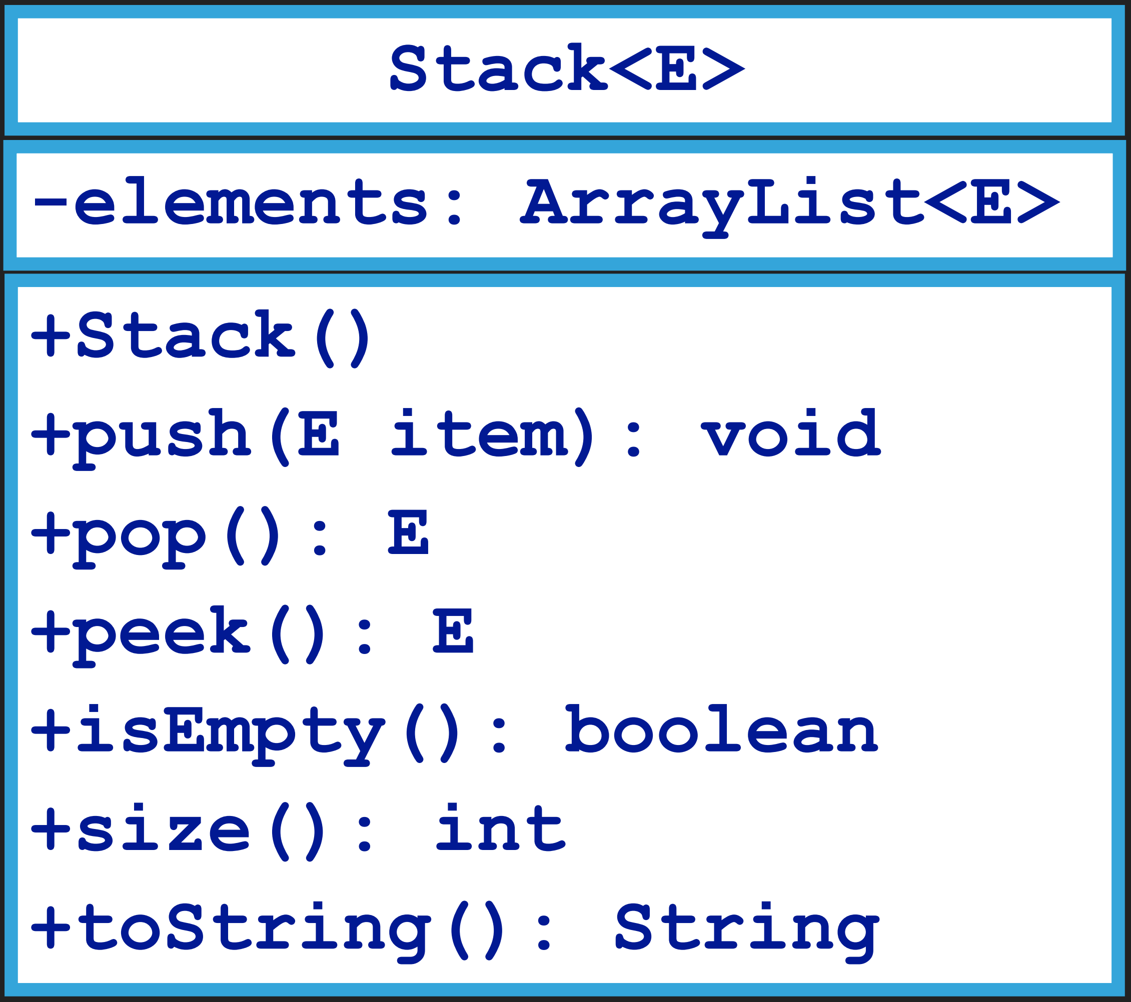
```java
import java.util.ArrayList;

public class UsingArrayList{
    public static void main(String[] args){
        // Instantiating ArrayList for type Integer
        ArrayList<Integer> numbers = new ArrayList<>();
        // Instantiating ArrayList for type String
        ArrayList<String> words = new ArrayList<>();
        // Arraylist of integers
        for(int i=0; i<5; i++){
            numbers.add(i*10);
            int j= (int)(Math.random() * 2);
            numbers.add(j, i*5);
            System.out.println(numbers);
        }
        System.out.println(numbers + " size: " + numbers.size());
        numbers.remove(5);
        System.out.println(numbers + " size: " + numbers.size());
        // Arraylist of strings
        words.add("tree");
        words.add("sky");
        words.add(1, "bird");
        words.add("cloud");
        System.out.println(words + " size: " + words.size());
        words.remove("sky");
        System.out.println(words + " size: " + words.size());
    }
}
```

GENERICS

- A generic type can be defined for a class or an interface
- A concrete type must be specified when using the generic class/interface
- Either to create objects or use the class/interface as a reference type

Creating a generic class

- Class Stack<E>

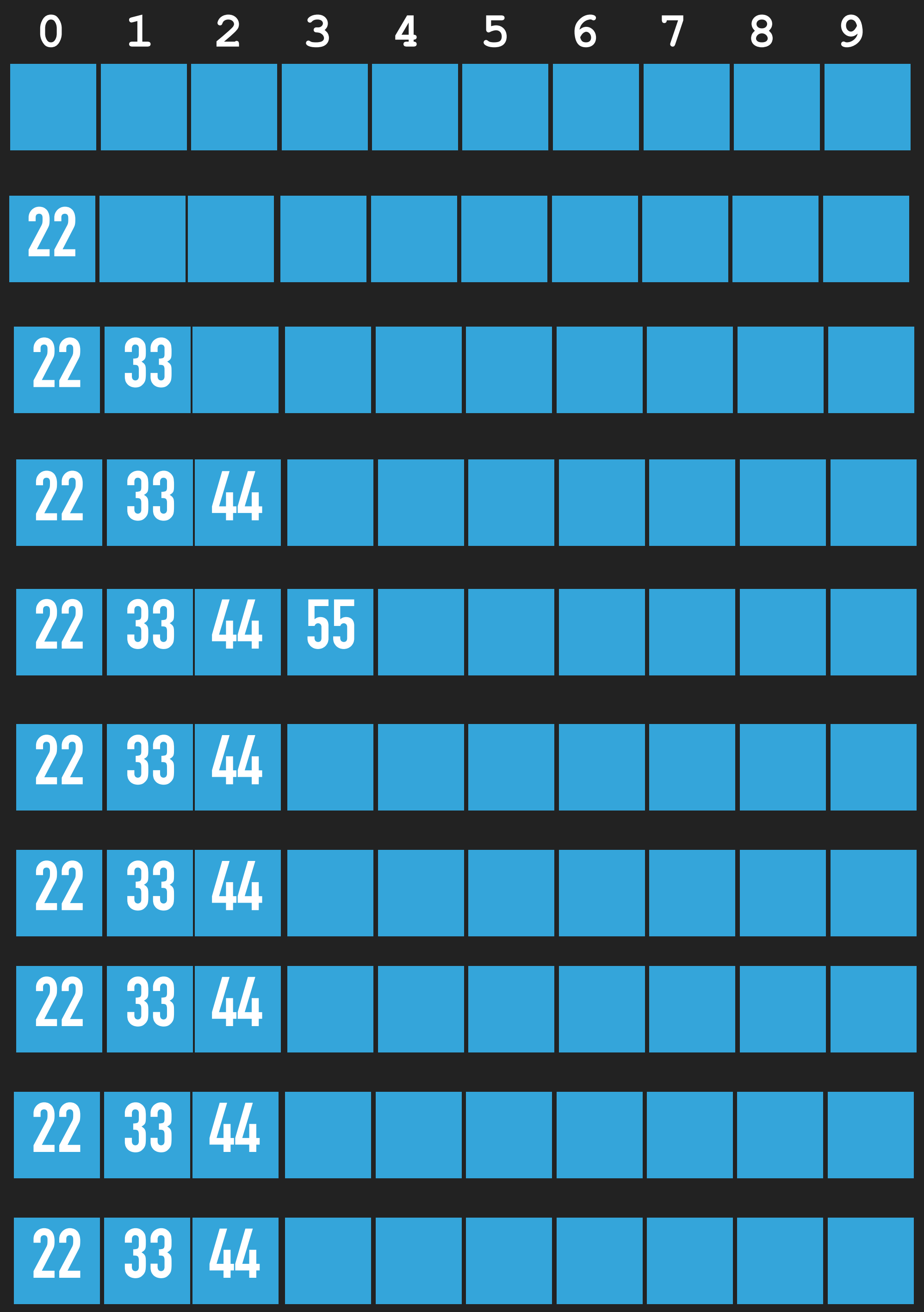


Stack<E>

-elements: ArrayList<E>	Storage of the stack data
+Stack()	Creates an empty stack
+push(E item): void	Adds item at the top of the stack
+pop(): E	Removes the item at the top of the stack
+peek(): E	Returns the value of the item at the top
+isEmpty(): boolean	Returns true if the stack is empty
+size(): int	Returns the number of items in the stack
+toString(): String	Returns the contents of the stack in a string

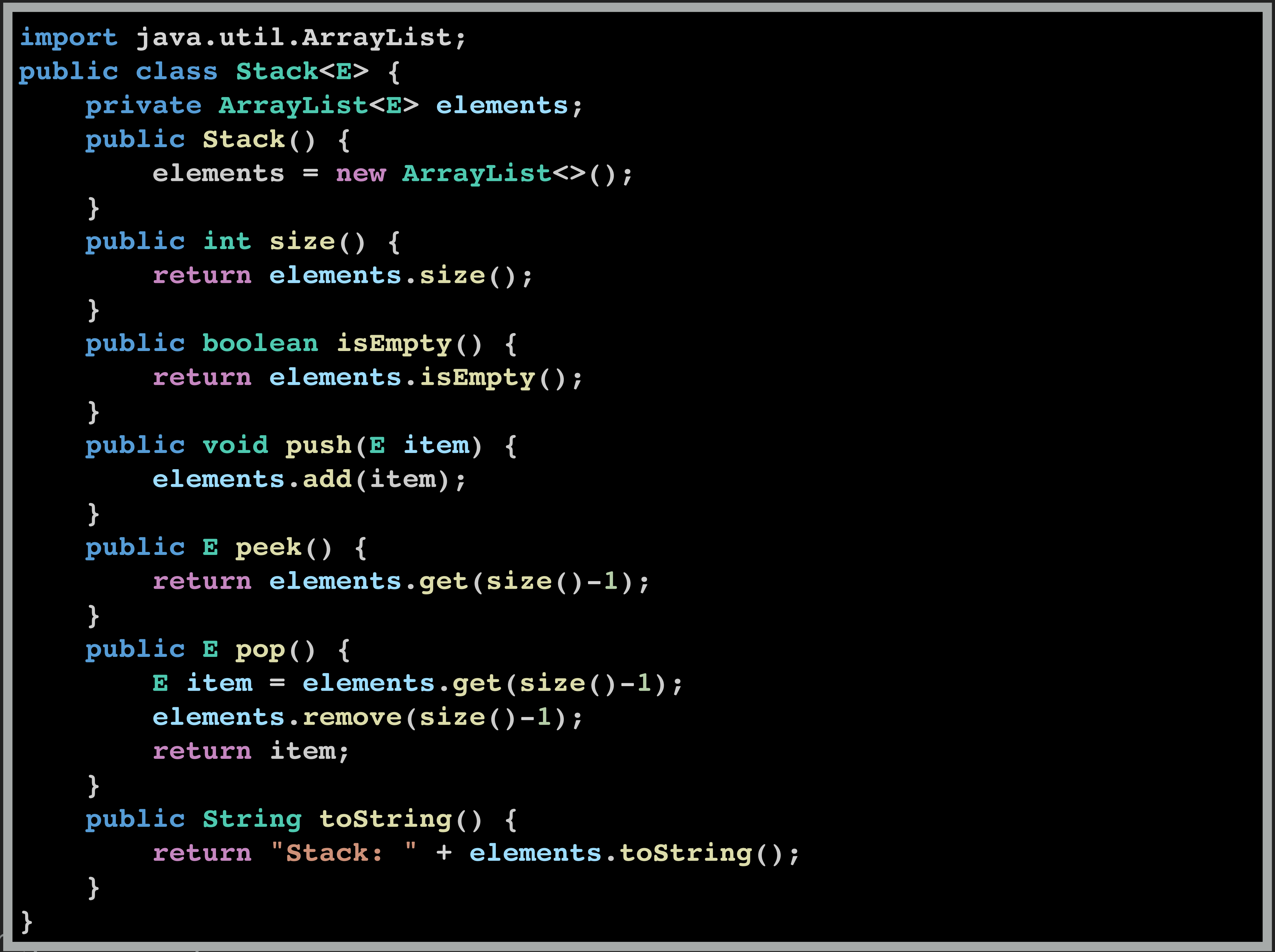
Creating a generic class

- Class Stack<E>



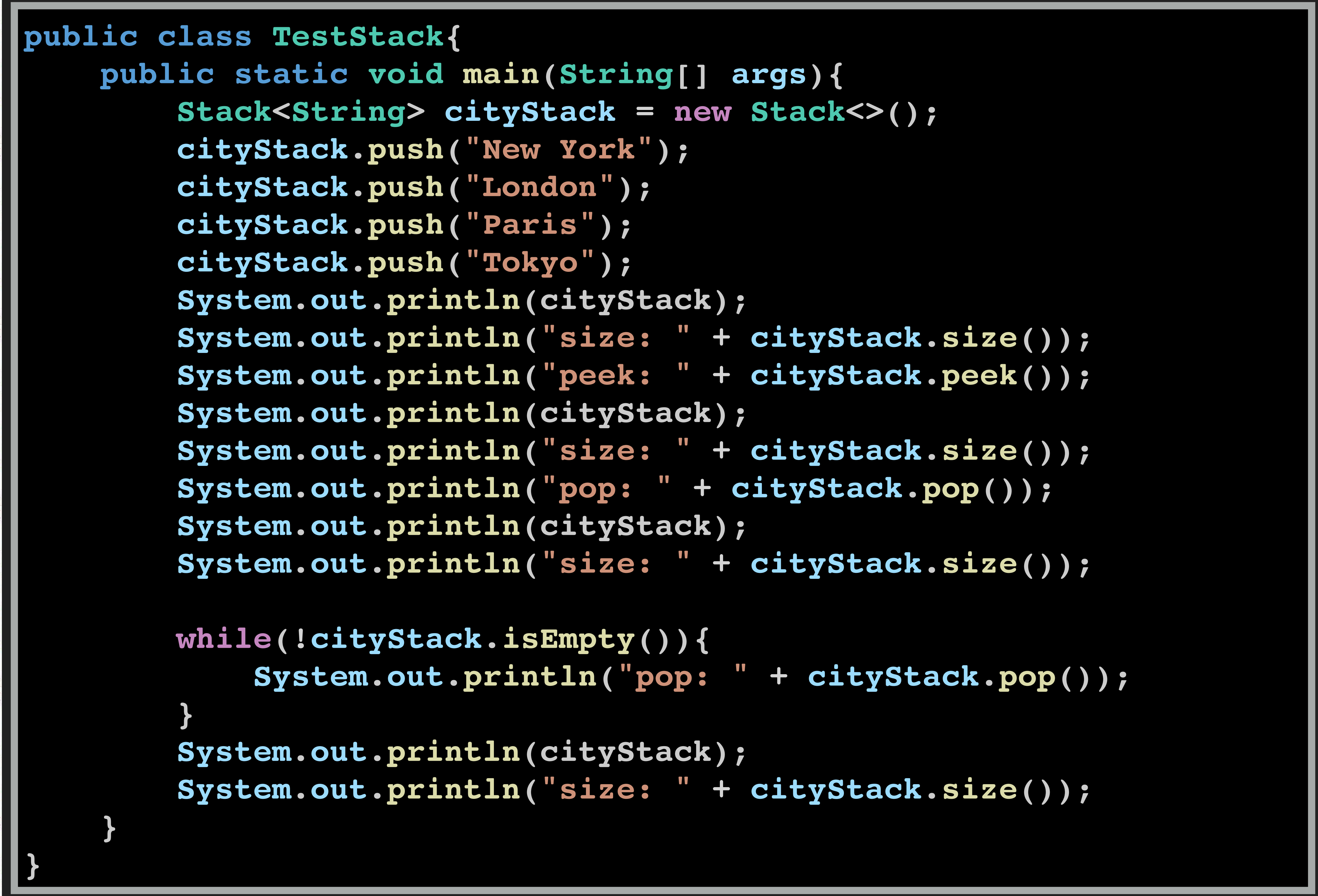
0	1	2	3	4	5	6	7	8	9	
										`Stack<Integer> stack = new Stack<>();`
22										`Stack.push(22);`
22	33									`Stack.push(33);`
22	33	44								`Stack.push(44);`
22	33	44	55							`Stack.push(55);`
22	33	44								`Stack.pop() returns 55;`
22	33	44								`Stack.peek() returns 44;`
22	33	44								`Stack.size() returns 3;`
22	33	44								`Stack.isEmpty() returns false;`
22	33	44								`Stack.toString() returns “[22,33,44]”;`

Generic Class - Stack<E>



```
import java.util.ArrayList;
public class Stack<E> {
    private ArrayList<E> elements;
    public Stack() {
        elements = new ArrayList<>();
    }
    public int size() {
        return elements.size();
    }
    public boolean isEmpty() {
        return elements.isEmpty();
    }
    public void push(E item) {
        elements.add(item);
    }
    public E peek() {
        return elements.get(size()-1);
    }
    public E pop() {
        E item = elements.get(size()-1);
        elements.remove(size()-1);
        return item;
    }
    public String toString() {
        return "Stack: " + elements.toString();
    }
}
```

Generic Class - Stack<E>

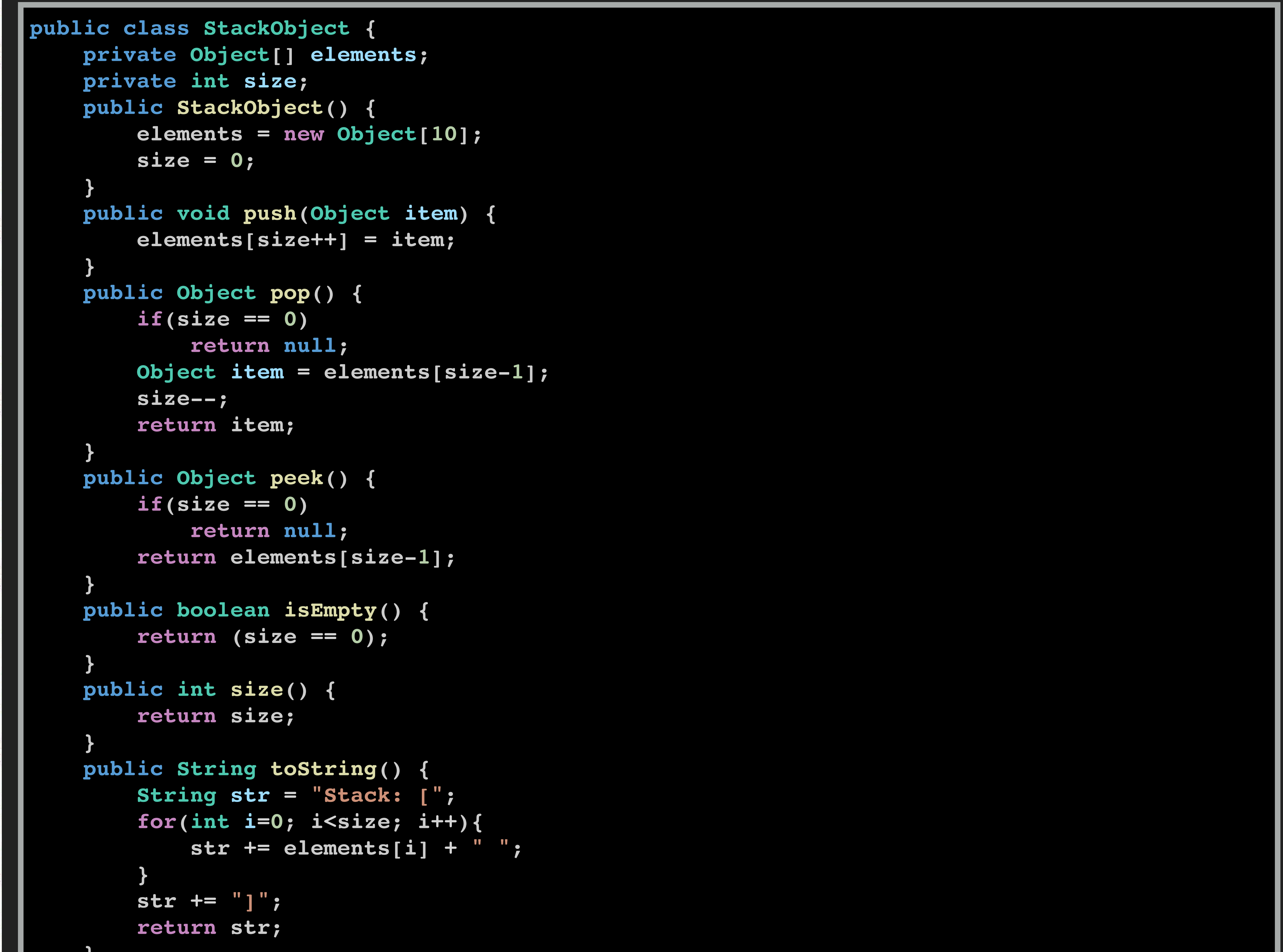


```
public class TestStack{
    public static void main(String[] args){
        Stack<String> cityStack = new Stack<>();
        cityStack.push("New York");
        cityStack.push("London");
        cityStack.push("Paris");
        cityStack.push("Tokyo");
        System.out.println(cityStack);
        System.out.println("size: " + cityStack.size());
        System.out.println("peek: " + cityStack.peek());
        System.out.println(cityStack);
        System.out.println("size: " + cityStack.size());
        System.out.println("pop: " + cityStack.pop());
        System.out.println(cityStack);
        System.out.println("size: " + cityStack.size());

        while(!cityStack.isEmpty()){
            System.out.println("pop: " + cityStack.pop());
        }
        System.out.println(cityStack);
        System.out.println("size: " + cityStack.size());
    }
}
```

How are generics implemented?

- After compile time, **`E`** is removed and replaced with the raw type (**`Object`**)
- Old way of implementing generics: use type **`Object`** instead of **`E`**
- Using an array of type **`Object`** would also work to implement a generic stack

GENERICS



```java
public class StackObject {
    private Object[] elements;
    private int size;
    public StackObject() {
        elements = new Object[10];
        size = 0;
    }
    public void push(Object item) {
        elements[size++] = item;
    }
    public Object pop() {
        if(size == 0)
            return null;
        Object item = elements[size-1];
        size--;
        return item;
    }
    public Object peek() {
        if(size == 0)
            return null;
        return elements[size-1];
    }
    public boolean isEmpty() {
        return (size == 0);
    }
    public int size() {
        return size;
    }
    public String toString() {
        String str = "Stack: [";
        for(int i=0; i<size; i++){
            str += elements[i] + " ";
        }
        str += "]";
        return str;
    }
```

GENERICS

```java
public class TestStackObject{
    public static void main(String[] args){
        StackObject cityStack = new StackObject();
        cityStack.push("New York");
        cityStack.push("London");
        cityStack.push("Paris");
        cityStack.push("Tokyo");
        System.out.println(cityStack);
        System.out.println("size: " + cityStack.size());
        System.out.println("peek: " + cityStack.peek());
        System.out.println(cityStack);
        System.out.println("size: " + cityStack.size());
        System.out.println("pop: " + cityStack.pop());
        System.out.println(cityStack);
        System.out.println("size: " + cityStack.size());

        while(!cityStack.isEmpty()){
            System.out.println("pop: " + cityStack.pop());
        }
        System.out.println(cityStack);
        System.out.println("size: " + cityStack.size());

        StackObject doubleStack = new StackObject();
        doubleStack.push(1.75);
        doubleStack.push(3.75);
        doubleStack.push(2.25);
        System.out.println(doubleStack);
        System.out.println("size: " + doubleStack.size());
    }
```

Erasure of the Generic Type

- Using Generics improves software reliability and readability
- Errors are detected at compile time
- A generic class or interface used without specifying a concrete type, is raw type (replaced with **`Object`**)

```
Stack stack = new Stack();
```

Equivalent to:

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

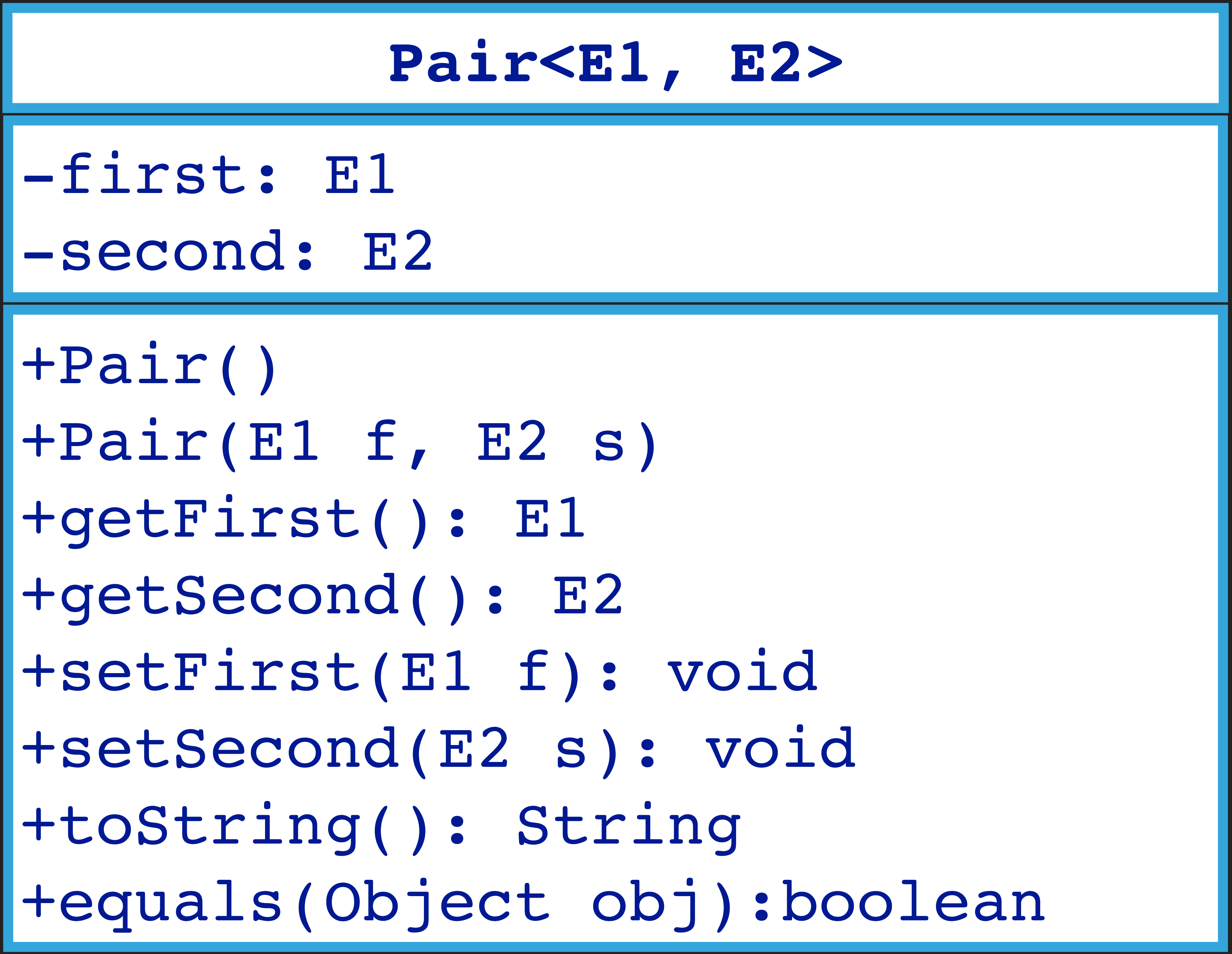
Erasure of the Generic Type

- Raw types are unsafe - may generate runtime errors
- Raw types should not be used unless backward compatibility is required

Multiple Generic Types

- A class/interface may have multiple type parameters (generic types)
- Class `Pair<E1, E2>` - two generic types
- Pair of string and number (name and id) for example

Multiple Generic Types



Pair<E1, E2>

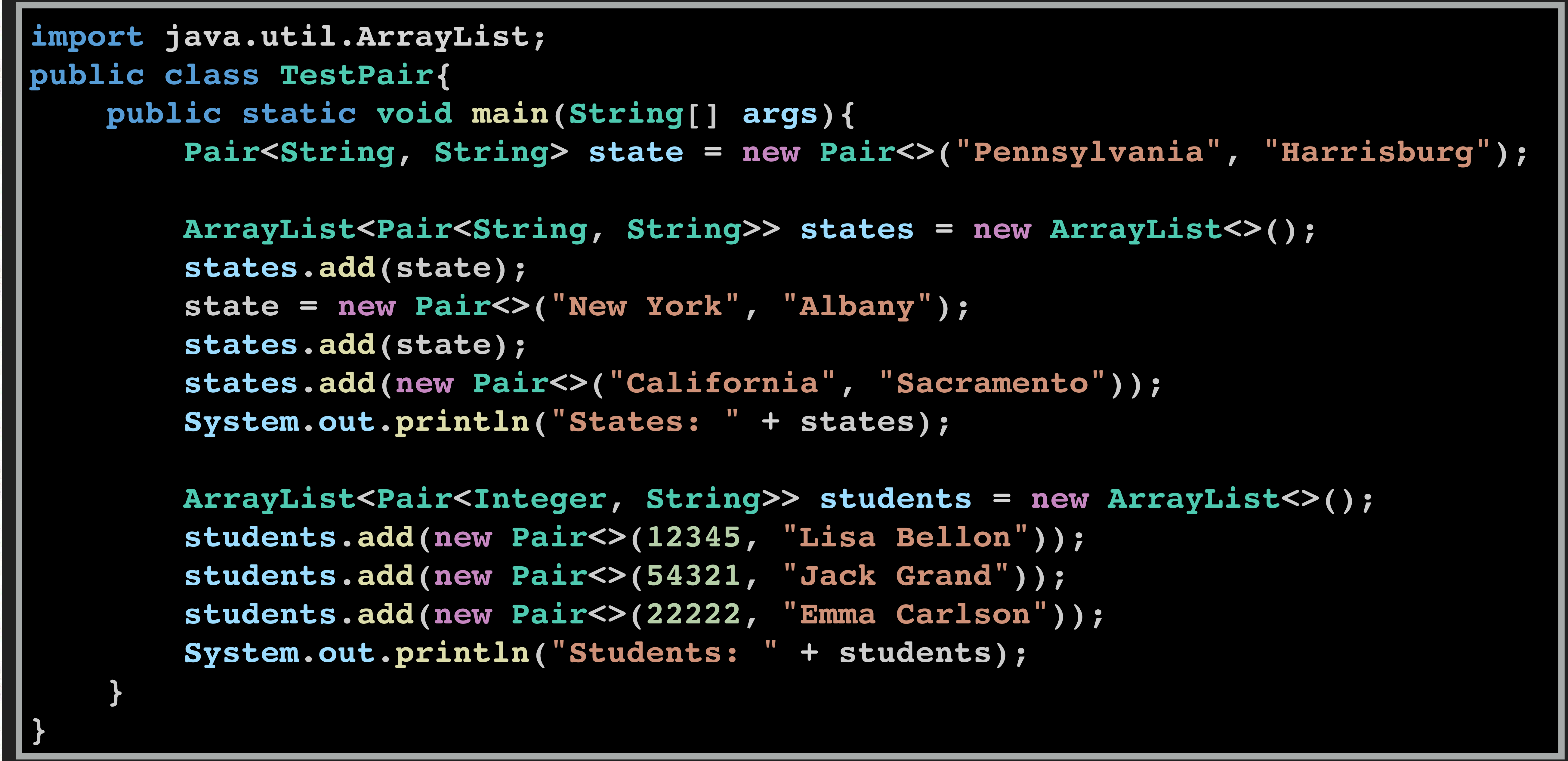
```
-first: E1
-second: E2
```

```
+Pair()
+Pair(E1 f, E2 s)
+getFirst(): E1
+getSecond(): E2
+setFirst(E1 f): void
+setSecond(E2 s): void
+toString(): String
+equals(Object obj):boolean
```

Multiple Generic Types

```java
public class Pair<E1, E2>{
    private E1 first;
    private E2 second;

    public Pair(E1 f, E2 s){
        first = f;
        second = s;
    }
    public E1 getFirst(){
        return first;
    }
    public E2 getSecond(){
        return second;
    }
    public void setFirst(E1 f){
        first = f;
    }
    public void setSecond(E2 s){
        second = s;
    }
    public String toString(){
        return "(" + first.toString() + ", " + second.toString() + ")";
    }
}
```

Multiple Generic Types



```java
import java.util.ArrayList;
public class TestPair{
    public static void main(String[] args){
        Pair<String, String> state = new Pair<>("Pennsylvania", "Harrisburg");

        ArrayList<Pair<String, String>> states = new ArrayList<>();
        states.add(state);
        state = new Pair<>("New York", "Albany");
        states.add(state);
        states.add(new Pair<>("California", "Sacramento"));
        System.out.println("States: " + states);

        ArrayList<Pair<Integer, String>> students = new ArrayList<>();
        students.add(new Pair<>(12345, "Lisa Bellon"));
        students.add(new Pair<>(54321, "Jack Grand"));
        students.add(new Pair<>(22222, "Emma Carlson"));
        System.out.println("Students: " + students);
    }
}
```

Generic Methods

- A generic method: parameters or return value are of type generic
- Printing arrays of different types `printArray()`
- Searching arrays of different types
- Sorting arrays of different types `java.util.Arrays.sort()`

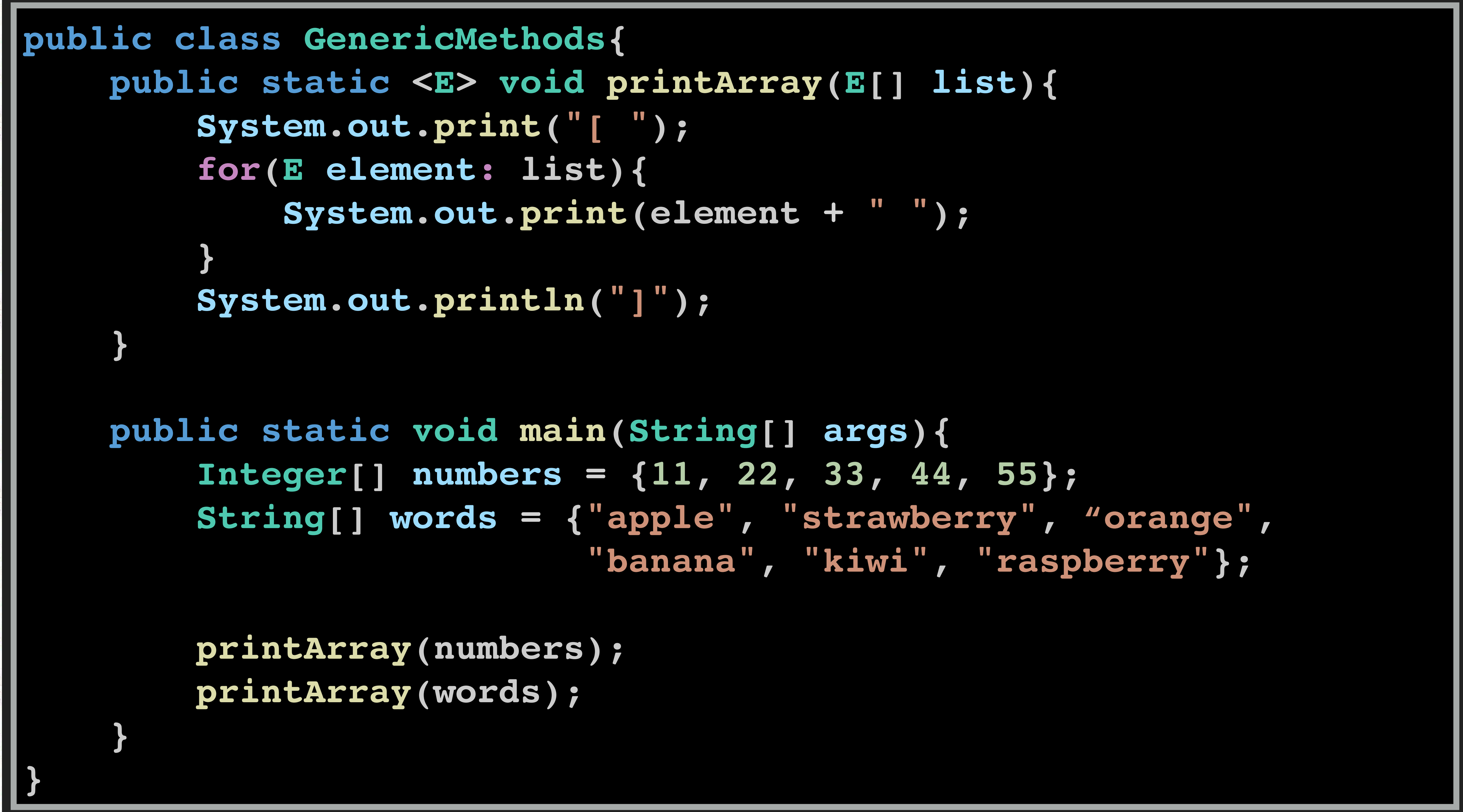
Generic Methods

- Generic method to print arrays of any type **E**

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

```
public static <E> void printAray(E[] list){
   System.out.print("[ ");
   for(E element: list){
      System.out.print(element.toString() + " ");
   }
   System.out.println("]");
}
```

Generic Methods



```
public class GenericMethods{
    public static <E> void printArray(E[] list){
        System.out.print("[ ");
        for(E element: list){
            System.out.print(element + " ");
        }
        System.out.println("]");
    }

    public static void main(String[] args){
        Integer[] numbers = {11, 22, 33, 44, 55};
        String[] words = {"apple", "strawberry", “orange",
                          "banana", "kiwi", "raspberry"};

        printArray(numbers);
        printArray(words);
    }
}
```

Generic Methods

- ✦ Sorting arrays of different types `java.util.Arrays.sort()` is a generic sort method
- ✦ `sort()` needs to compare the elements to order them
- ✦ The elements of the array need to be ordered - must be comparable

Generic Methods

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

Generic Type **`E`**

`E` must be a subtype of **`Comparable`**

Type **`E`** has a definition for the method **`compareTo()`**

WildCard Generic Type

- Generic type can be restricted to specific types or groups of types
- **`<E extends Comparable<E>>`** restricts the type **`E`** to be a subtype of **`Comparable`**
- Types of wildcards: `?`, `? extends T`, and `? Super T`

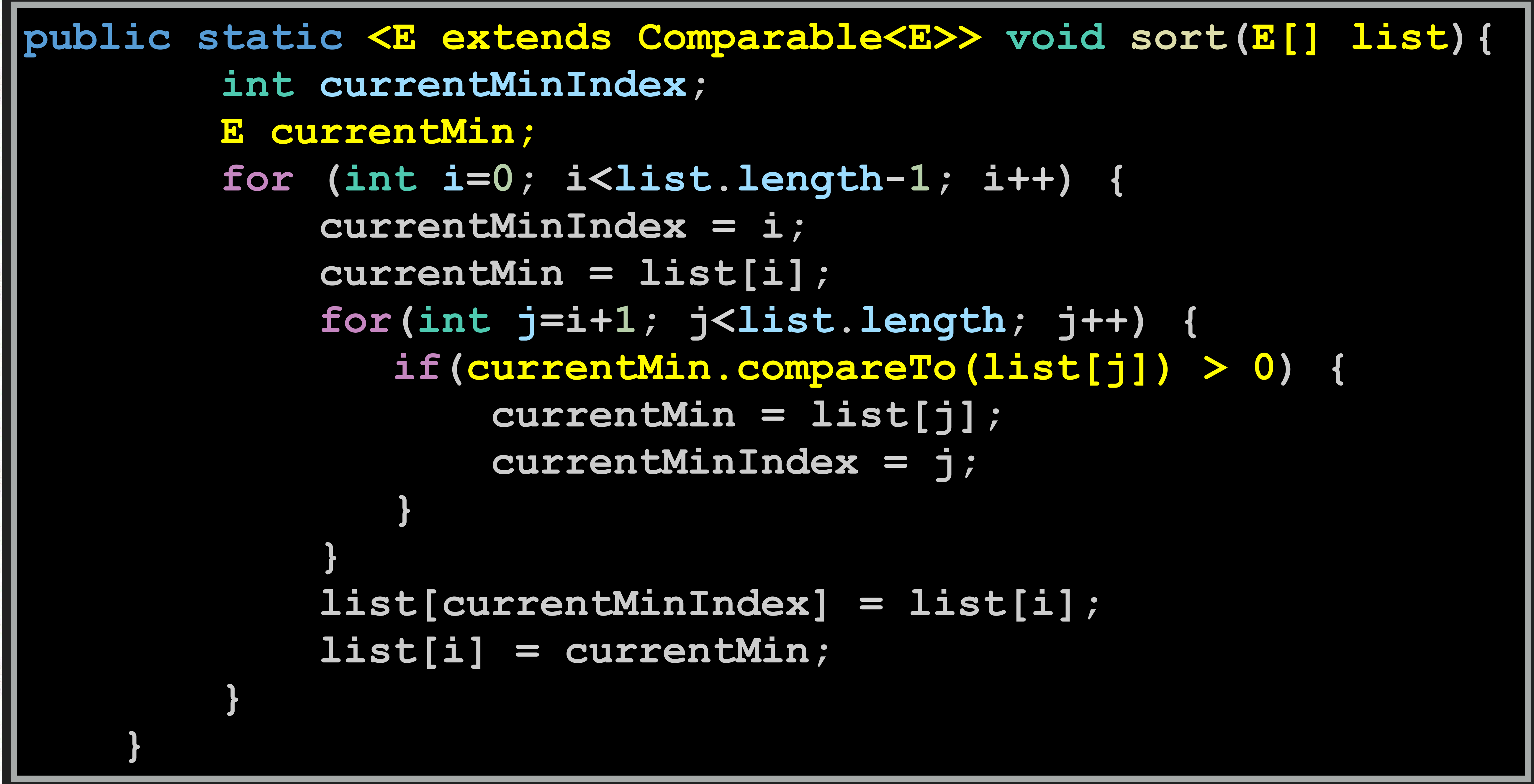
WildCard Generic Type

- Unbounded wildcard `?`
  - Equivalent to `? extends Object`
- Bounded wildcard `? extends T`
  - Generic Type must be `T` or a subtype of `T`
- Lower bound wildcard `? Super T`
  - Generic type must be `T` or super type of `T`

Generic Methods

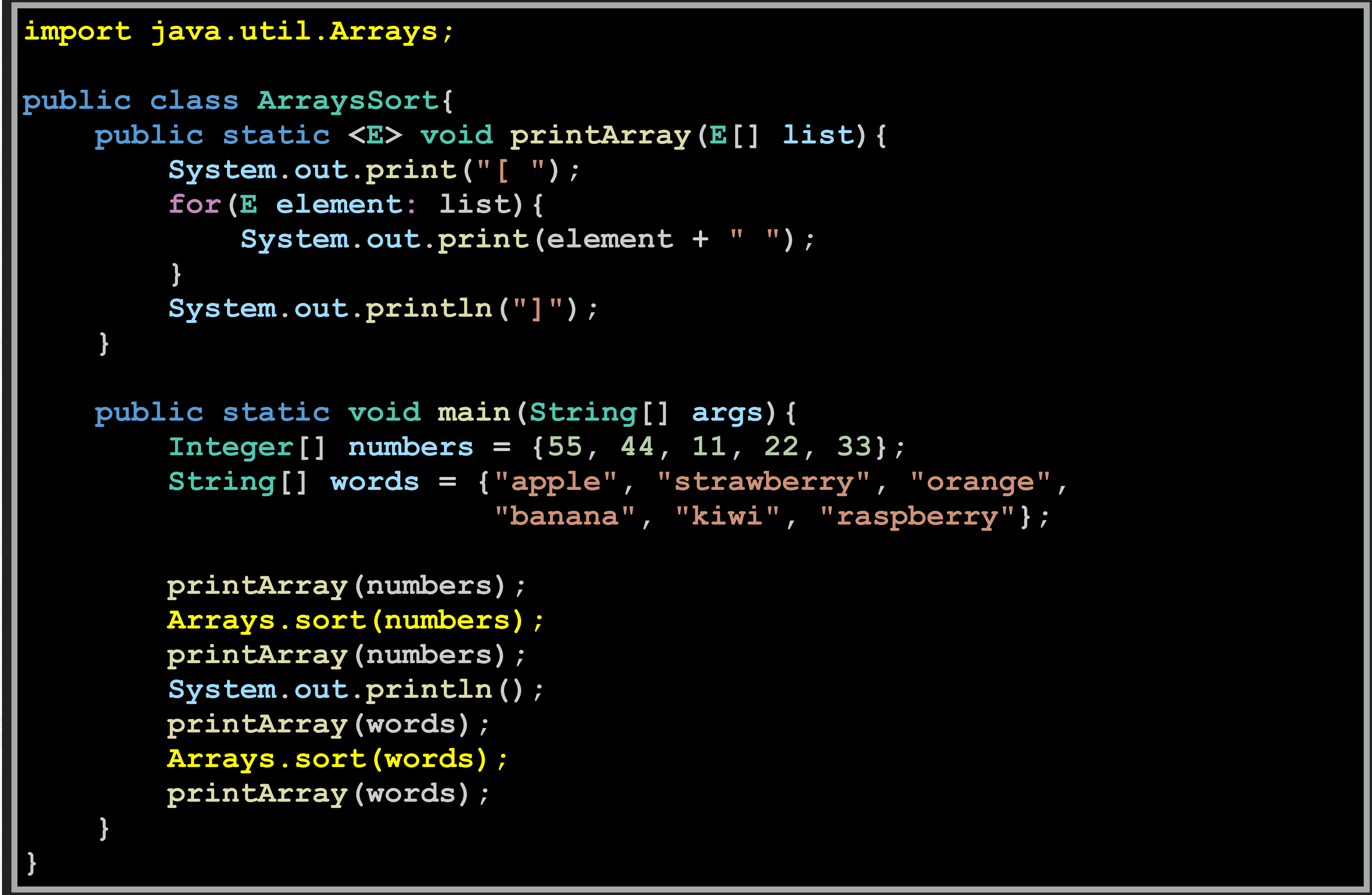
```
public static void sort(Integer[] list){
        int currentMinIndex;
        int currentMin;
        for (int i=0; i<list.length-1; i++) {
            currentMinIndex = i;
            currentMin = list[i];
            for(int j=i+1; j<list.length; j++) {
               if(currentMin > list[j]) {
                   currentMin = list[j];
                   currentMinIndex = j;
               }
            }
            list[currentMinIndex] = list[i];
            list[i] = currentMin;
        }
    }
```

Generic Methods



```
public static <E extends Comparable<E>> void sort(E[] list){
        int currentMinIndex;
        E currentMin;
        for (int i=0; i<list.length-1; i++) {
            currentMinIndex = i;
            currentMin = list[i];
            for(int j=i+1; j<list.length; j++) {
               if(currentMin.compareTo(list[j]) > 0) {
                   currentMin = list[j];
                   currentMinIndex = j;
               }
            }
            list[currentMinIndex] = list[i];
            list[i] = currentMin;
        }
    }
```

Generic Methods



```
import java.util.Arrays;

public class ArraysSort{
    public static <E> void printArray(E[] list){
        System.out.print("[ ");
        for(E element: list){
            System.out.print(element + " ");
        }
        System.out.println("]");
    }

    public static void main(String[] args){
        Integer[] numbers = {55, 44, 11, 22, 33};
        String[] words = {"apple", "strawberry", "orange",
                          "banana", "kiwi", "raspberry"};

        printArray(numbers);
        Arrays.sort(numbers);
        printArray(numbers);
        System.out.println();
        printArray(words);
        Arrays.sort(words);
        printArray(words);
    }
}
```

Generic Methods

- `java.util.Arrays.sort()` is overloaded (One version uses **`Comparable<E>`** or natural ordering)

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

public static <E> void sort(E[] list, int fromInd, int toInd)
```

Generic Methods

- **java.util.Arrays.sort()** is overloaded (Second version uses **Comparator**<E>)

```
public static <E> void sort(E[] list,
                            Comparator<? Super E> c)
```

```
java.util.Comparator

public interface Comparator<T>{
   int compare(T obj1, T obj2);
}
```

Generic Methods

- Example: sort objects of type `Student` using three criteria
  - Natural ordering (by id)
  - Ordering by name
  - Ordering by GPA

Natural ordering

```java
public class Student implements Comparable<Student>{
    private int id;
    private String name;
    private double gpa;

    public Student(int id, String name, double gpa){
        this.id = id;
        this.name = name;
        this.gpa = gpa;
    }
    public int getID(){ return id;}
    public String getName(){ return name;}
    public double getGPA(){ return gpa;}

    public void setID(int id){ this.id = id;}
    public void setName(String name){ this.name = name;}
    public void setGPA(double gpa){ this.gpa = gpa;}

    public int compareTo(Student s){
        return id - s.id;
    }
    public String toString(){
        return "(" + id + ", " + name + ", " + gpa + ")";
    }
}
```

Define the ordering by name

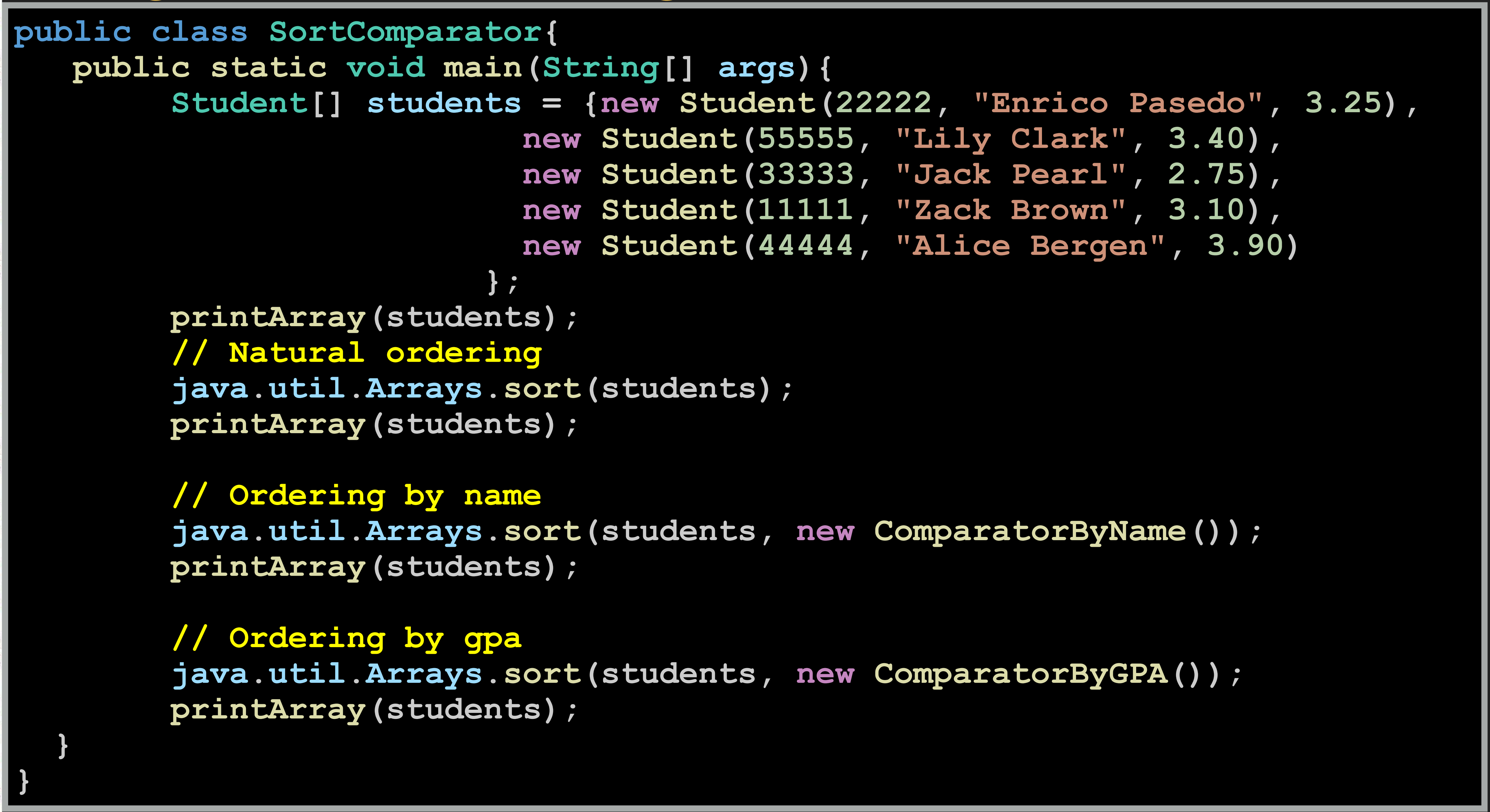
```
import java.util.Comparator;
public class ComparatorByName implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        return s1.getName().compareTo(s2.getName());
    }
}
```

Define the ordering by GPA

```
import java.util.Comparator;
public class ComparatorByGPA implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        Double gpa1 = s1.getGPA();
        Double gpa2 = s2.getGPA();
        return gpa1.compareTo(gpa2);
    }
}
```

Generic Methods

Using the three sorting criteria



```
public class SortComparator{
   public static void main(String[] args){
        Student[] students = {new Student(22222, "Enrico Pasedo", 3.25),
                          new Student(55555, "Lily Clark", 3.40),
                          new Student(33333, "Jack Pearl", 2.75),
                          new Student(11111, "Zack Brown", 3.10),
                          new Student(44444, "Alice Bergen", 3.90)
                        };
        printArray(students);
        // Natural ordering
        java.util.Arrays.sort(students);
        printArray(students);

        // Ordering by name
        java.util.Arrays.sort(students, new ComparatorByName());
        printArray(students);

        // Ordering by gpa
        java.util.Arrays.sort(students, new ComparatorByGPA());
        printArray(students);
  }
}
```

Restrictions on Generic types

✦ Restriction 1

Cannot create instances using the generic type **<E>**

```
E item = new E();
```

✦ Restriction 2

Cannot create an array of type E

```
E[] list = new E[20];
```

Restrictions on Generic types

✦ Restriction 3

Generic type is not allowed in a static context

```
public static E item;
public static void m(E object)
```

✦ Restriction 4

Exception classes cannot be generic

```
public class MyException<T> extends Exception{ }
public static void main(String[] args){
  try{
     Cannot check the thrown exception
  }
  catch(MyException<T> ex){
  }
}
```

Summary

- Generic classes and interfaces
- Multiple generic types
- Generic methods
- Raw types (unsafe)
- Restrictions on generic types