PROGRAMMING AND DATA STRUCTURES

GENERICS (TEMPLATES)

HOURIA OUDGHIRI

FALL 2021

OUTLINE

- What are generics?
- Using generic classes (`ArrayList`) and generic methods (`sort`)
- Creating new generic classes with one type parameter or more
- Creating generic methods
- Restrictions on generic type

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`, …)
- Can use any name for the generic type (between `<>`) but the convention 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
```

ArrayList

ArrayList<E>
+ArrayList() +ArrayList(int capacity)

Default constructor: creates an array of 10 elements

Second constructor: creates an array of **`capacity`** elements

ArrayList

ArrayList<E>
+add(int index, E item): void
+add(E item): void

adds **item** at location **index.** All elements from **index** to **size()-1** are pushed one position up

adds **item** at the first unused location

ArrayList

ArrayList<E>

+get(int index): E returns element at location **index**

+set(int index, E item): E replaces element at location **index** with **item** - returns the old value of the item at **index**

ArrayList

ArrayList<E>	
+size(): int	returns the actual size of the array (not the capacity)
+isEmpty(): boolean	returns true if the array is empty
+clear(): void	reset size to 0

ArrayList

ArrayList<E>

+contains(Object obj): boolean	Returns true is **obj** is found in the array
+indexOf(Object obj): int	Returns the index of obj if found, -1 otherwise
+lastIndexOf(Object obj): int	Returns the index of the last occurrence of obj if found, -1 otherwise

ArrayList

ArrayList<E>

```
+remove(int ind): boolean

+remove(Object obj): boolean
```

Returns true if **index** is valid and element at index removed, false otherwise

Returns true if **obj** is found and removed, and false otherwise

ArrayList

```
import java.util.ArrayList;

public static void main(String[] args){

  // Create an array words of 10 String elements
  ArrayList<String> words = new ArrayList<>();
}
```

1

2

Instantiating **ArrayList** for type **String**

ArrayList

```
import java.util.ArrayList;
public static void main(String[] args){

 // Create an array words of 10 String elements
 ArrayList<String> words = new ArrayList<>();

 // Adding string “Tree” to array words -
 // position 0
 words.add(“Tree”);
}
```

ArrayList

```
import java.util.ArrayList;
public static void main(String[] args){

  // Create an array words of 10 String elements
  ArrayList<String> words = new ArrayList<>();

  // Adding string “Tree” to words - position 0
  words.add(“Tree”);
  // Adding string “Sky” to words - position 1
  words.add(“Sky”);
  // Adding string “Bird” to words - position 2
  words.add(“Bird”);
}
```

```
import java.util.ArrayList;
public static void main(String[] args){

  // Create an array words of 10 String elements
  ArrayList<String> words = new ArrayList<>();

  // Adding string “Tree” to words - position 0
  words.add(“Tree”);
  // Adding string “Sky” to words - position 1
  words.add(“Sky”);
  // Adding string “Bird” to words - position 2
  words.add(“Bird”);
  words.add(1, “Squirrel”);
  // squirrel at position 1, Sky moves to 2
  // and Bird moves to 3
   System.out.println(words.size());  // 4
}
```

ArrayList

```
import java.util.ArrayList;
public static void main(String[] args){


//Create an arraylist accounts of 25 Integer elements

 ArrayList<Integer> accounts = new ArrayList<>(25);
}
```

ArrayList

```
import java.util.ArrayList;
public static void main(String[] args){

//Create accounts with 25 Integer elements
  ArrayList<Integer> accounts =
                              new ArrayList<>(25);

  // Adding number 112244 to accounts-position 0
  accounts.add(112244); // Auto-Boxing
  // Adding number 112244 to accounts-position 1
  accounts.add(221133);

}
```

ArrayList

```
import java.util.ArrayList;
public static void main(String[] args){

//Create an array accounts of 25 Integer elements
  ArrayList<Integer> accounts =
                              new ArrayList<>(25);

  // Adding number 112244 to accounts -position 0
  accounts.add(112244); // Auto-Boxing
  // Adding number 112244 to accounts -position 1
  accounts.add(221133);
  int account = accounts.get(1); // Auto-Unboxing
  System.out.println(accounts.size()); // 2
}
```

ArrayList

```
import java.util.ArrayList;
public static void main(String[] args){

  // Create numbers with 10 int elements
  // Error
  ArrayList<int> numbers = new ArrayList<>();
}
```

ArrayList

- **`ArrayList`** comes with an enhanced loop to access the array elements (For each loop)

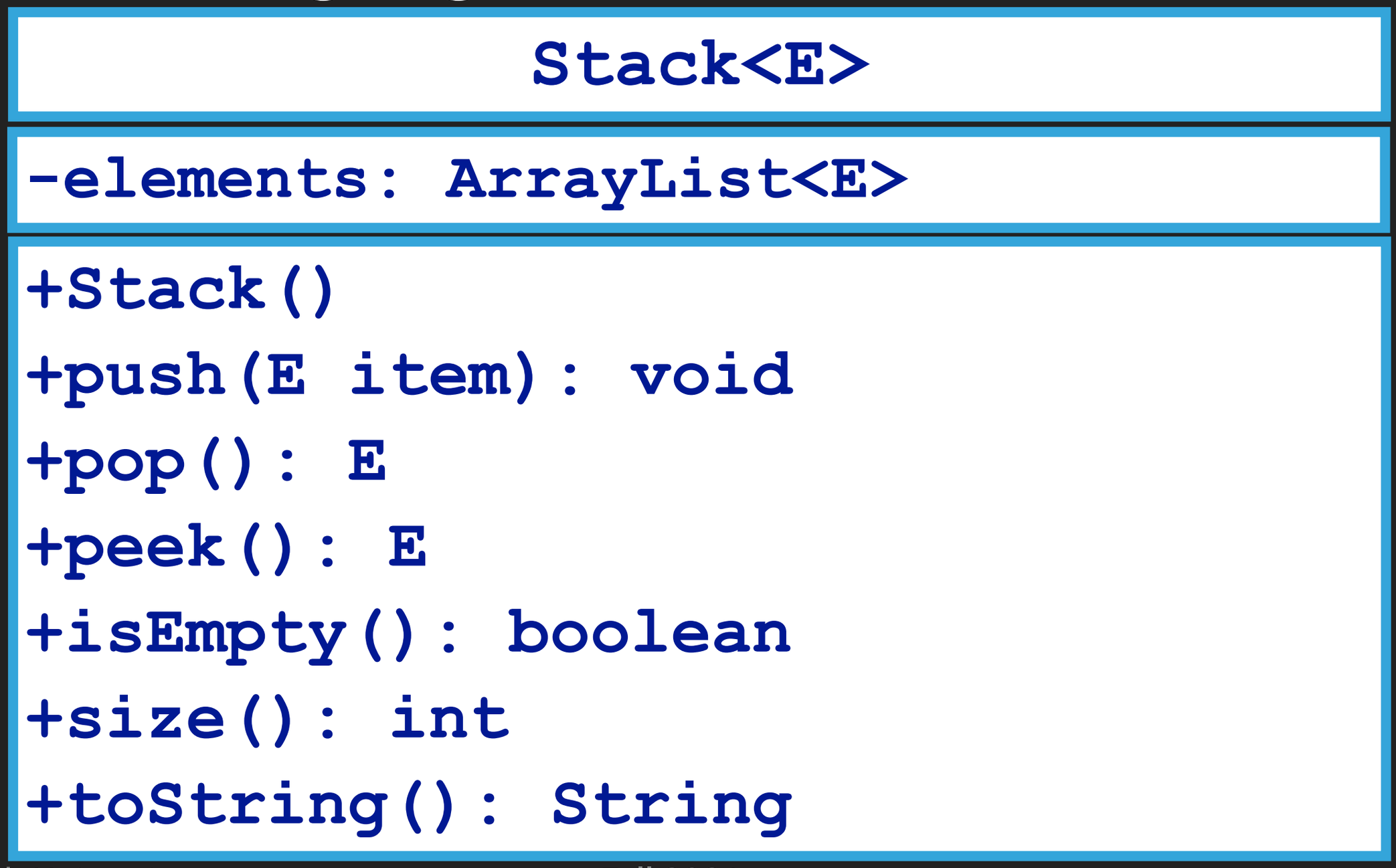
```
import java.util.ArrayList;
public static void main(String[] args){

//Create an array accounts of 25 Integer elements
  ArrayList<Integer> accounts =
                              new ArrayList<>(25);

 for(Integer item: accounts){
   System.out.println(item);
 }
}
```

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 as a reference type

Generic Class - Stack<E>

- Creating a generic class



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();
  }
}
```

Generic Class - Stack<E>

```
public class Stack<E> {
    .
    .
    .
  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;
  }
}
```

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("Stack of Cities”);
    System.out.println(cityStack.toString());
    System.out.println("Top element: “ +
                       cityStack.peek());
  }
}
```

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("Stack of Cities”);
    System.out.println(cityStack.toString());
    System.out.println("Top element: “ +
                       cityStack.peek());
  }
}
```

```
Stack of Cities
Stack: [New York, London, Paris, Tokyo]
Top element: Tokyo
```

Generic Class - Stack<E>

```
public class TestStack {
  public static void main(String[] args) {
    Stack<Integer> numberStack = new Stack<>();
    numberStack.push(11);
    numberStack.push(22);
    numberStack.push(33);
    numberStack.push(44);
    numberStack.push(55);

    System.out.println("Stack of numbers”);
    System.out.println(numberStack.toString());
    System.out.println("Top element: " +
                       numberStack.peek());
  }
}
```

Generic Class - Stack<E>

```
public class TestStack {
  public static void main(String[] args) {
    Stack<Integer> numberStack = new Stack<>();
    numberStack.push(11);
    numberStack.push(22);
    numberStack.push(33);
    numberStack.push(44);
    numberStack.push(55);

    System.out.println("Stack of numbers”);
    System.out.println(numberStack.toString());
    System.out.println("Top element: " +
                       numberStack.peek());
  }
}
```

```
Stack of numbers
Stack: [11, 22, 33, 44, 55]
Top element: 55
```

Generic Class - Stack<E>

- After compile time, **`E`** is removed and replaced with the raw type (**`Object`**) - Erasure of the generic type

- Old way of implementing generics: use type **`Object`** instead of **`E`**

- Using array with elements of type **`Object`** would also work

Generic Class - Stack<E>

```
public class ObjectStack {
   private Object[] elements;
   int size;
   public ObjectStack() {
     elements = new Object[10]; size = 0;
   }
   public void push(Object item) { elements[size++] = item; }
   public Object peek() { return elements[size-1]; }
   public int size() { return size; }
   public Object pop() { return elements[—size]; }
   public boolean isEmpty() { return (size == 0); }
   public String toString(){
      String s = “Stack: [”;
      for(int i=0; i<size-1; i++){
         s+= elements[i].toString() + “ “);
      }
      s+= elements[i].toString() + “]”;
      return s;
   }
}
```

Generic Class - Stack<E>

```
public class TestObjectStack {

  public static void main(String[] args) {
    ObjectStack cityStack = new ObjectStack();
    cityStack.push("New York");
    cityStack.push("London");
    cityStack.push("Paris");
    cityStack.push("Tokyo");
    cityStack.push(22); // ok

    System.out.println("Stack of Cities\n" +
                                cityStack.toString());
    System.out.println("Top element: " + cityStack.peek());
  }

}
```

Generic Class - Stack<E>

```
public class TestObjectStack {

  public static void main(String[] args) {
    ObjectStack cityStack = new ObjectStack();
    cityStack.push("New York");
    cityStack.push("London");
    cityStack.push("Paris");
    cityStack.push("Tokyo");
    cityStack.push(22); // ok

    System.out.println("Stack of Cities\n" +
                       cityStack.toString());
    System.out.println("Top element: " + cityStack.peek());
  }

}
```

```
Stack of Cities
[New York, London, Paris, Tokyo, 22]
Top element: 22
```

Generic Class - Stack<E>

- Using Generics improves software reliability and readability
- Errors are detected at compile time

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");
    cityStack.push(22);
    System.out.println("Top element: " +
                       cityStack.peek());
  }
}
```

The method push(String) in the type Stack<String> is not applicable for the arguments (int)

Restrictions on generic type

✦ Restriction 1

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

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

Restrictions on generic type

✦ Restriction 2

Cannot create an array of type E

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

Restrictions on generic type

✦ Restriction 3

Generic type is not allowed in a static context

```
public static E item;

public static void m(E object)
```

Restrictions on generic type

- 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){
   }
}
```

Generic Class - Multiple types

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

Generic Class - multiple 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

Generic Class - multiple types

```
public class Pair<E1, E2> {

  private E1 first;
  private E2 second;

}
```

Generic Class - multiple types

```
public class Pair<E1, E2> {

  private E1 first;
  private E2 second;

  public Pair(E1 first, E2 second) {
    this.first = first;
    this.second = second;
  }
}
```

Generic Class - multiple types

```
public class Pair<E1, E2> {
  private E1 first;
  private E2 second;
  public Pair(E1 first, E2 second) {
    this.first = first;
    this.second = second;
  }
  public void setFirst(E1 first) {
    this.first = first;
  }
  public void setSecond(E2 second) {
    this.second = second;
  }
}
```

Generic Class - multiple types

```
public class Pair<E1, E2> {
    private E1 first;
    private E2 second;
    public Pair(E1 first, E2 second) {
        this.first = first;
        this.second = second;
    }
    public void setFirst(E1 first) {
        this.first = first;
    }
    public void setSecond(E2 second) {
        this.second = second;
    }
    public E1 getFirst() {
        return first;
    }
    public E2 getSecond() {
        return second;
    }
}
```

Generic Class - multiple types

```
public class Pair<E1, E2> {

  . . .

  public String toString() {
    return "(" + first.toString() + ", " +
                 second.toString() + ")";
  }
  public boolean equals(Object obj) {
    Pair<E1, E2> p = (Pair<E1, E2>) obj;
    boolean eq1 =  p.getFirst().equals(first);
    boolean eq2 =  p.getSecond().equals(second);
    return eq1 && eq2;
  }
}
```

Generic Class - multiple types

```
import java.util.ArrayList;
public class TestPair {
   public static void main(String[] args) {

      ArrayList<Pair<Integer, String>> list = new ArrayList<>();
      Pair<Integer, String> p;
      p = new Pair<Integer, String>(12345, "Lisa Bello”);
      list.add(p);
      p = new Pair<Integer, String>(54321, "Karl Johnson”);
      list.add(p);
      p = new Pair<Integer, String>(12543, "Jack Green”);
      list.add(p);
      p = new Pair<Integer, String>(53241, "Emma Carlson”);
      list.add(p);
      System.out.println(list.toString());

   }
}
```

Generic Class - multiple types

```
import java.util.ArrayList;
public class TestPair {
   public static void main(String[] args) {

      ArrayList<Pair<Integer, String>> list = new ArrayList<>();
      Pair<Integer, String> p;
      p = new Pair<Integer, String>(12345, "Lisa Bello”);
      list.add(p);
      p = new Pair<Integer, String>(54321, "Karl Johnson”);
      list.add(p);
      p = new Pair<Integer, String>(12543, "Jack Green”);
      list.add(p);
      p = new Pair<Integer, String>(53241, "Emma Carlson”);
      list.add(p);
      System.out.println(list.toString());

   }
}
```

```
[(12345, Lisa Bello),
 (54321, Karl Johnson),
 (12543, Jack Green),
 (53241, Emma Carlson)]
```

Generic Class - multiple types

```
import java.util.ArrayList;
public class TestPair {
 public static void main(String[] args) {
   ArrayList<Pair<String, String>> list = new ArrayList<>();
  Pair<String, String> p;
  p = new Pair<String, String>(“New York, “New York City”);
  list.add(p);
  p = new Pair<String, String>(“Pennsylvania”, "Harrisburg”);
  list.add(p);
  p = new Pair<String, String>(“Ohio”, "Columbus”);
  list.add(p);
  p = new Pair<String, String>(“California”, "Sacramento”);
  list.add(p);
  System.out.println(list.toString());
   }
}
```

Generic Methods

- A method can be generic - 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)
```

Generic Methods

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

  System.out.print("[ “);

  for (int i=0; i<list.length; i++)
    System.out.print(list[i] + " “);

  System.out.println("]");
}
```

Generic Methods

```
public class GenericPrint{

  public static void main(String[] args) {
    Integer[] numbers = {11, 22, 33, 44, 55};

    printArray(numbers);

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

}
```

Generic Methods

```
public class GenericPrint{

  public static void main(String[] args) {
    Integer[] numbers = {11, 22, 33, 44, 55};
    printArray(numbers);

    String[] words = {"Apple", "Orange", “Banana",
                      "Kiwi", "Strawberry", "Raspberry"};
    printArray(words);
  }
  public static <E> void printArray(E[] list) {
    System.out.print("[ ");
    for (int i=0; i<list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println("]");
  }

}
```

Generic Methods

```
public class GenericPrint{

  public static void main(String[] args) {
    Integer[] numbers = {11, 22, 33, 44, 55};
    String[] words = {"Apple", "Orange", “Banana",
                      "Kiwi", "Strawberry", "Raspberry"};
    printArray(numbers);
    printArray(words);
  }
  public static <E> void printArray(E[] list) {
    System.out.print("[ ");
    for (int i=0; i<list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println("]");
  }

}
```

```
[ 11 22 33 44 55 ]
[ Apple Orange Banana Kiwi Strawberry Raspberry ]
```

Generic Methods

- Sorting arrays of different types **`java.util.Arrays.sort()`**
- **`sort()`** needs to compare the elements (order them)
- Elements of the array need to be compared - 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()**

Generic Methods

```
public static <E extends Comparable<E>> void sort(E[] list)
{     // Selection Sort
      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;
            }
         }
         if (currentMinIndex != i) {
            list[currentMinIndex] = list[i];
            list[i] = currentMin;
         }
      }
}
```

Generic Methods

```
public class GenericSort {
   public static void main(String[] args) {
      Integer[] numbers = {22, 11, 55, 44, 33};
      String[] words = {"Apple", "Orange", "Banana", “Kiwi",
                        “Strawberry", "Raspberry"};
      printArray(numbers);
      printArray(words);
      sort(numbers);
      sort(words);
      printArray(numbers);
      printArray(words);
   }
}
```

Generic Methods

```
public class GenericSort {
   public static void main(String[] args) {
      Integer[] numbers = {22, 11, 55, 44, 33};
      String[] words = {"Apple", "Orange", "Banana", “Kiwi",
                        “Strawberry", "Raspberry"};
      printArray(numbers);
      printArray(words);
      sort(numbers);
      sort(words);
      printArray(numbers);
      printArray(words);
   }
}
```

```
Before sort:
[ 22 11 55 44 33 ]
[ Apple Orange Banana Kiwi Strawberry Raspberry ]
After sort:
[ 11 22 33 44 55 ]
[ Apple Banana Kiwi Orange Raspberry Strawberry ]
```

Generic Methods

```
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

```
public class comparatorByColor
                implements Comparator<Shape> {
    int compare(Shape s1, Shape s2){
      return s1.getColor().compareTo(s2.getColor());
    }
}
```

```
public class comparatorByArea
                implements Comparator<Shape> {
    int compare(Shape s1, Shape s2){
      Double area1 = s1.getArea();
      Double area2 = s2.getArea();
      return area1.compareTo(area2);
    }
}
```

Generic Methods

```
public static void main(String[] args) {
    Shape[] s = {new Circle(),
                 new Circle("Red", 5.0),
                 new Circle("Blue", 2.5),
                 new Rectangle(),
                 new Rectangle("Green", 10.5, 5.5),
                 new Rectangle("Yellow", 4.0, 2.5)};
    printArray(s);
    System.out.println("\n");
    java.util.Arrays.sort(s, new ComparatorByArea());
    printArray(s);
    System.out.println("\n");
    java.util.Arrays.sort(s, new ComparatorByColor());
    printArray(s);
  }
```

Generic Methods

```
Type        Color     Dimensions                Area      Perimeter
Circle      Black     1.00                      3.14      6.28
Circle      Red       5.00                      78.54     31.42
Circle      Blue      2.50                      19.63     15.71
Rectangle   Black     1.00      1.00            1.00      4.00
Rectangle   Green     10.00     5.50            55.00     31.00
Rectangle   Yellow    4.00      2.50            10.00     13.00

List of shapes sorted by area:
Rectangle   Black       1.00        1.00          1.00          4.00
Circle      Black       1.00                      3.14          6.28
Rectangle   Yellow      4.00        2.50          10.00         13.00
Circle      Blue        2.50                      19.63         15.71
Rectangle   Green       10.00       5.50          55.00         31.00
Circle      Red         5.00                      78.54         31.42

List of shapes sorted by color:
Rectangle   Black       1.00        1.00          1.00          4.00
Circle      Black       1.00                      3.14          6.28
Circle      Blue        2.50                      19.63         15.71
Rectangle   Green       10.00       5.50          55.00         31.00
Circle      Red         5.00                      78.54         31.42
Rectangle   Yellow      4.00        2.50          10.00         13.00
```

GENERICS

- A generic class or interface used without specifying a concrete type, is raw type and will be replaced with **`Object`**

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

Equivalent to:

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

- Raw types are used for backward compatibility only

GENERICS

- Old Java version of the interface **`Comparable`** is not generic

```
int compareTo(Object obj)
```

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

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`

WildCard Generic Type

Parameter is a Stack with any type that extends **`Object`**

```
public static void print(Stack<?> stack)
{
   System.out.print("From top: [");
   while(!stack.isEmpty()) {
      System.out.print(stack.top() + ", “);
      stack.pop();
   }
   System.out.println("]");
}
```

WildCard Generic Type

Parameter **`stack2`** must be of type **`T`** or a super type of **`T`**

```
public static <T> void add(Stack<T> stack1,

                           Stack<? super T> stack2)
{
  while(!stack1.isEmpty()) {
    stack2.push(stack1.pop());
  }
}
```

WildCard Generic Type

```
public static void main(String[] args) {
   Stack<String> cities = new Stack<>();
   cities.push("New York");
   cities.push("London");
   cities.push("Paris");
   cities.push("Tokyo");

   Stack<Object> mix = new Stack<>();
   mix.push("Bangkok");
   mix.push(333);
   mix.push(75.25);

   add(cities, mix); // OK

   print(mix);

}
```

WildCard Generic Type

```
public static void main(String[] args) {
   Stack<String> cities = new Stack<>();
   cities.push("New York");
   cities.push("London");
   cities.push("Paris");
   cities.push("Tokyo");

   Stack<Object> mix = new Stack<>();
   mix.push("Bangkok");
   mix.push(333);
   mix.push(75.25);

   add(cities, mix); // OK

   print(mix);
```

```
From top: [New York, London, Paris, Tokyo, 75.25, 333, Bangkok]
```

WildCard Generic Type

```
public static void main(String[] args) {
   Stack<String> cities = new Stack<>();
   cities.push("New York");
   cities.push("London");
   cities.push("Paris");
   cities.push("Tokyo");

   Stack<Integer> mix = new Stack<>();
   mix.push(1267);
   mix.push(333);
   mix.push(755);

   add(cities, mix); // Error

   print(mix);

}
```

WildCard Generic Type

```
public static void main(String[] args) {
   Stack<String> cities = new Stack<>();
   cities.push("New York");
   cities.push("London");
   cities.push("Paris");
   cities.push("Tokyo");

   Stack<Integer> mix = new Stack<>();
   mix.push(1267);
   mix.push(333);
   mix.push(755);

   add(cities, mix); // Error

   print(mix);
}
```

```
The method add(Stack<T>, Stack<? super T>) in the type
TestStack is not applicable for the arguments
(Stack<String>, Stack<Integer>)
```

SUMMARY

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