Generics & Collections

CSGE601021 Dasar-Dasar Pemrograman 2

Fakultas Ilmu Komputer Universitas Indonesia

.:Generics:.

Generics at a Glance

- The capability to parameterize types.
- You can define a class or a method with generic types
- Later the types can be substituted using concrete types by the compiler.

Generics at a Glance (2)

- For example,
  - A generic stack class stores the elements of a generic type.
  - From this generic stack class
    - A stack object for holding strings
    - A stack object for holding numbers.
  - Strings and numbers are concrete types

Why?

- Enable errors to be detected at compile time rather than at runtime.
  - Why is this better? Discuss.

- A generic class or method permits you to specify allowable types of objects that the class or method may work with.

- If you attempt to use the class or method with an incompatible object, a compile error occurs.

Generic Type

```
package java.lang;

public interface Comparable {
  public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<T> {
  public int compareTo(T o)
}
```

(b) JDK 1.5

Generic Instantiation

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

(a) Prior to JDK 1.5

Runtime error

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

(b) JDK 1.5

Compile error

- Why?

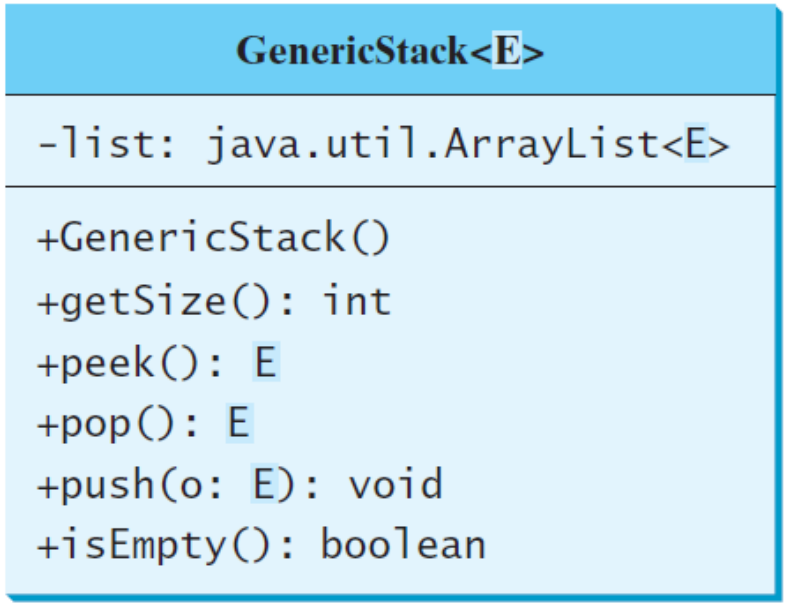
ArrayList is a Generic Class!

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E) : void	Appends a new element o at the end of this list.
+add(index: int, o: E) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : E	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : boolean	Removes the element at the specified index.
+set(index: int, o: E) : E	Sets the element at the specified index.

No Casting Needed!

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

list.add(5.5); // 5.5 is automatically converted to new Double(5.5)
list.add(3.0); // 3.0 is automatically converted to new Double(3.0)

Double doubleObject = list.get(0); // No casting is needed

double d = list.get(1); // Automatically converted to double
```

Declaring Generic Classes and Interfaces



An array list to store elements.

Creates an empty stack.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns true if the stack is empty.

http://www.cs.armstrong.edu/liang/intro10e/html/GenericStackWithLineNumber.html?

Generic Methods

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

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

Bounded Generic Type

```
public static void main(String[] args ) {
    Rectangle rectangle = new Rectangle(2, 2);
    Circle circle = new Circle (2);
    System.out.println("Same area? " + equalArea(rectangle, circle));
}
```

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

Raw Type and Backward Compatibility

- Raw type:

```
ArrayList list = new ArrayList();
```

- This is *roughly* equivalent to

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

Raw Type is Unsafe

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum between two objects */
  public static Comparable max(Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

- What will happen if we run `Max.max("Welcome", 23);`

Runtime Error

Make it Safe

```
// Max1.java: Find a maximum object
public class Max1 {
  /** Return the maximum between two objects */
  public static <E extends Comparable<E>> E max(E o1, E o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

- What will happen if we run `Max.max("Welcome", 23);`

Wildcards

https://liveexample.pearsoncmg.com/html/WildCardNeedDemo.html

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

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

  /** 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;
  }
}
```

ERROR

intStack is not an instance of **GenericStack <Number>**

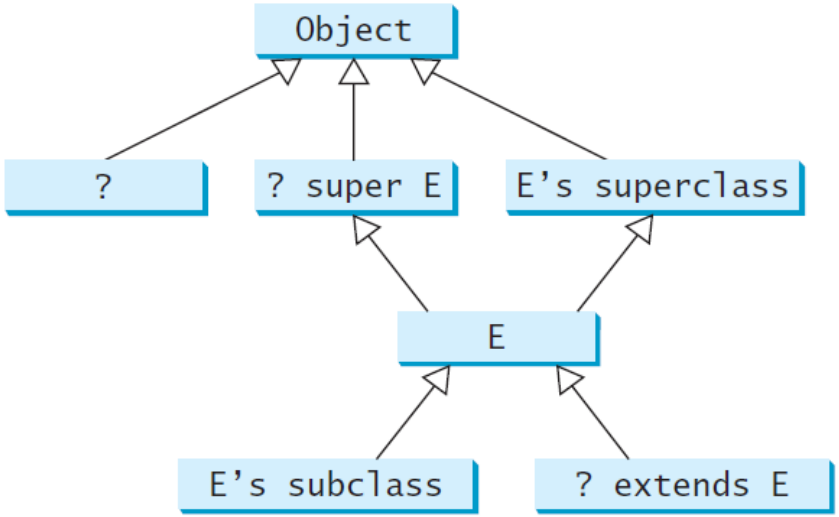
Wildcards

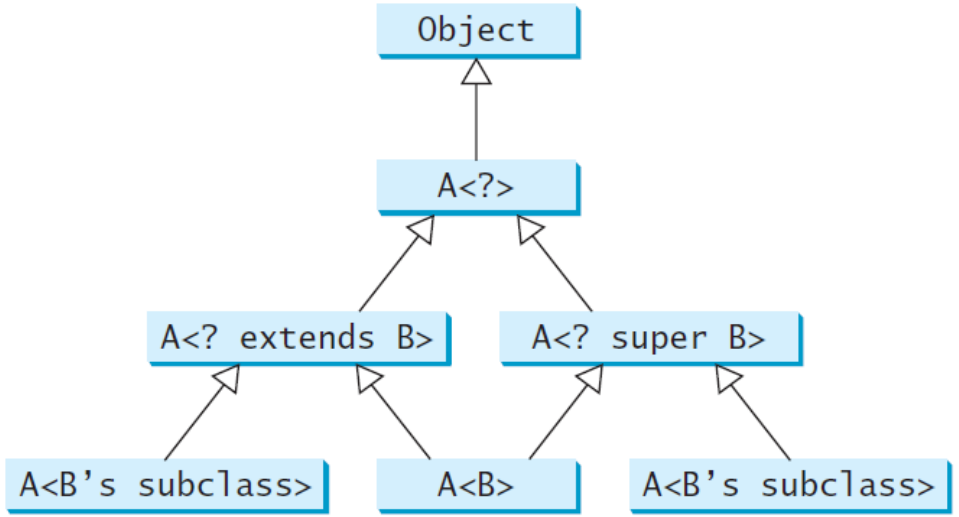
?	unbounded wildcard ➔ equivalent to: ? extends Object
? extends T	bounded wildcard
? super T	lower bound wildcard

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

    print(intStack);
  }

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

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

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

Generic Types and Wildcard Types





Erasure and Restrictions on Generics

- Generics are implemented using an approach called *type erasure*.
- The compiler uses the generic type to compile the code, but erases it after.
- The generic information is **not available at run time**.
- This approach enables the generic code to be **backward-compatible** with the legacy code that uses raw types.

Compile Time Checking

- The compiler checks whether generics is used correctly for the following code in (a) and translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

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

(a)

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

(b)

Important Facts

- It is important to note that a generic class is shared by all its instances regardless of its actual generic type.

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

- Although GenericStack<String> and GenericStack<Integer> are two types, but there is only one class GenericStack loaded into the JVM.

Restrictions on Generics

- Restriction 1: Cannot Create an Instance of a Generic Type. (i.e., new E()).
- Restriction 2: Generic Array Creation is Not Allowed. (i.e., new E[100]).
- Restriction 3: A Generic Type Parameter of a Class Is Not Allowed in a Static Context.
- Restriction 4: Exception Classes Cannot be Generic.

Example

https://liveexample.pearsoncmg.com/html/GenericMatrix.html

- Objective: This example gives a generic class for matrix arithmetic. This class implements matrix addition and multiplication common for all types of matrices.

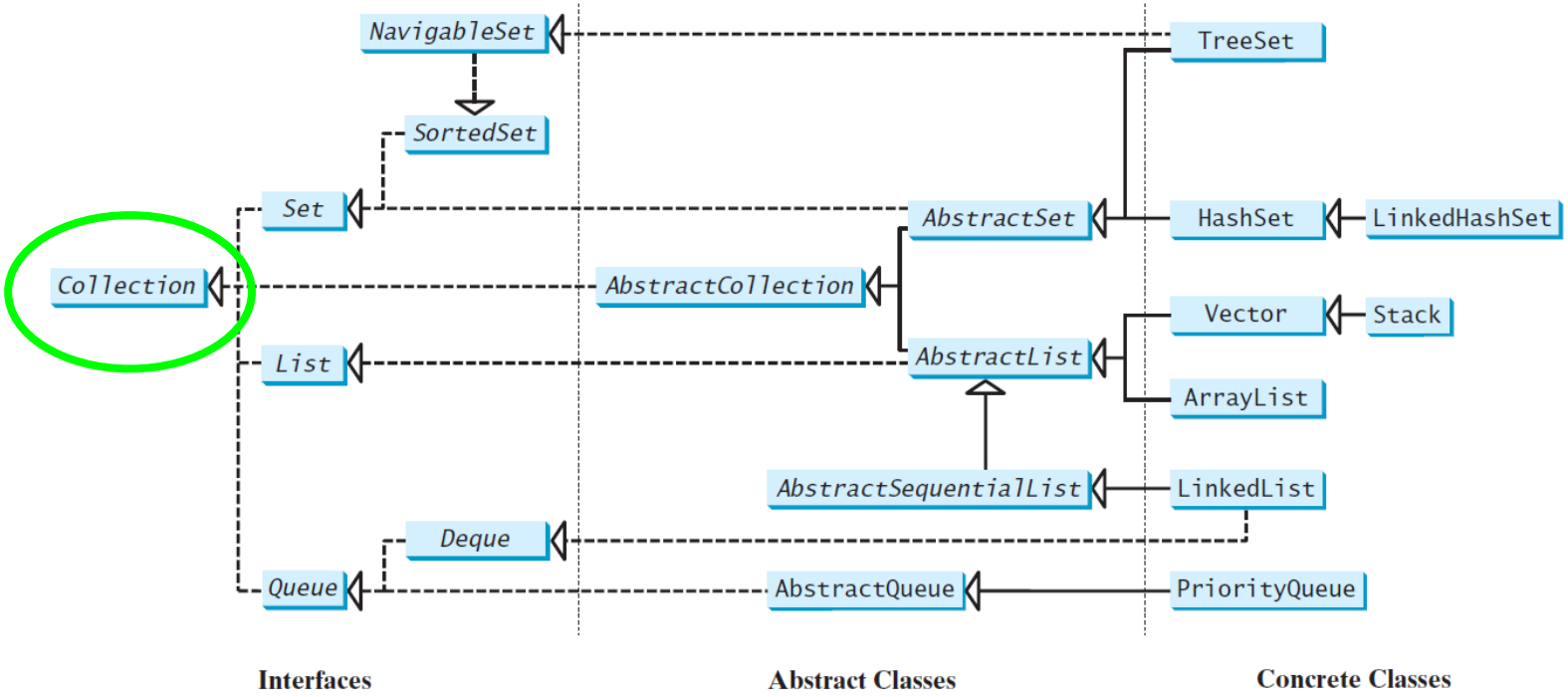
.:Collections:.

Java Collection Framework hierarchy

- A collection is a container object that holds a group of objects, often referred to as elements.
- The Java Collections Framework supports two types of containers
  - One for storing a collection of elements is simply called a *collection*.
  - The other, for storing key/value pairs, is called a *map*
- Maps are efficient data structures for quickly searching an element using a key
  - *We will not discuss map in this class*
- The Java Collections Framework supports three types of collections, named *lists*, *sets*, and *maps*.

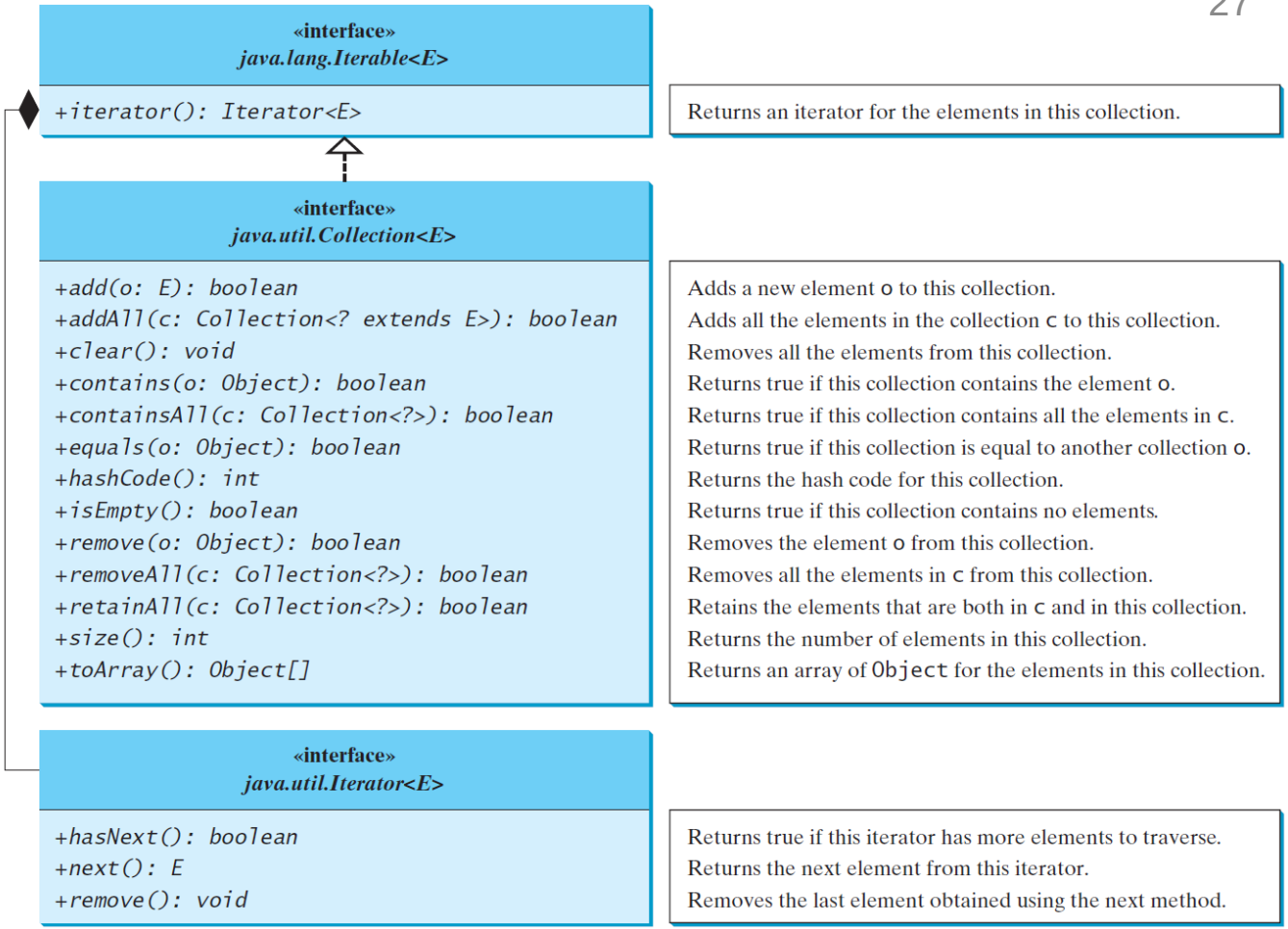
Java Collection Framework hierarchy

- Set and List are subinterfaces of Collection.

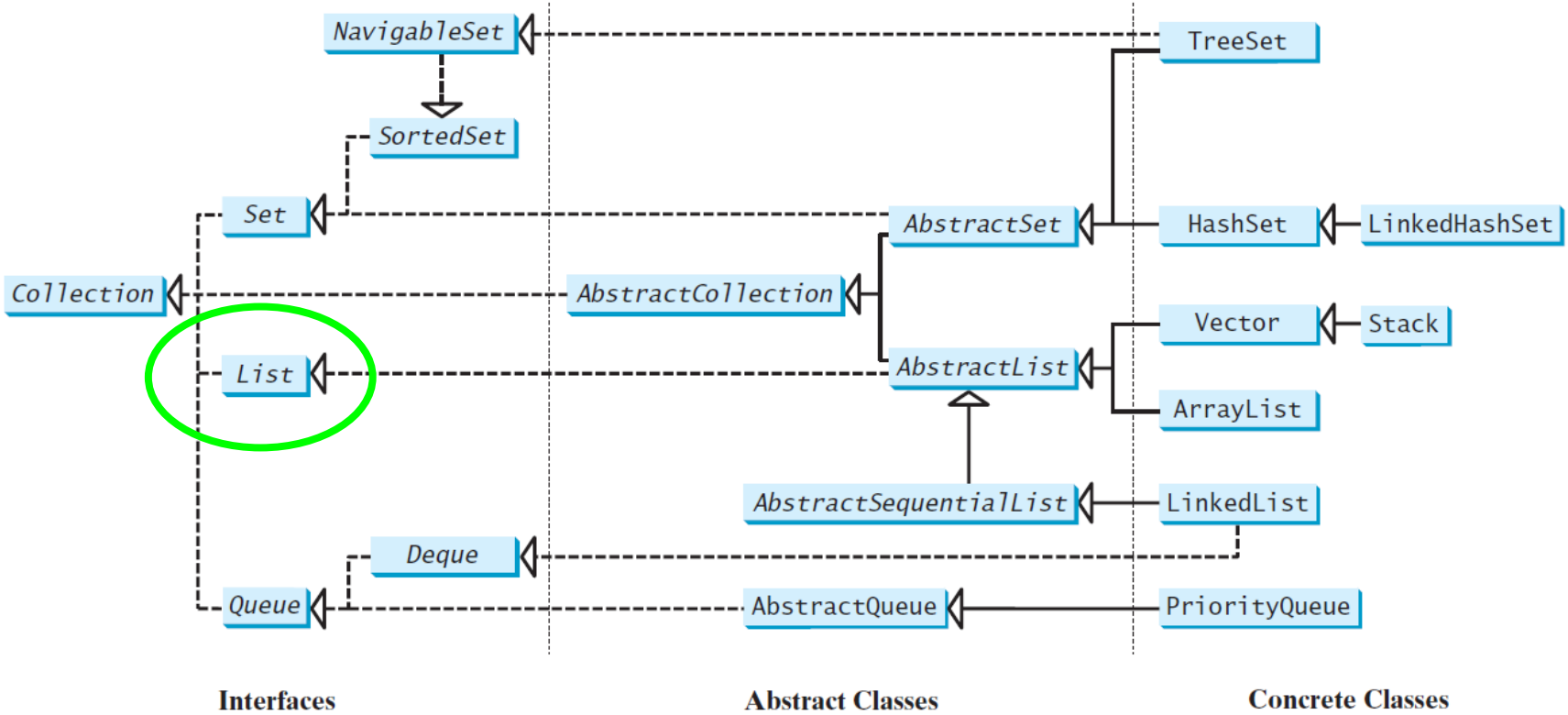


The `Collection` Interface

- The Collection interface is the root interface for manipulating a collection of objects.

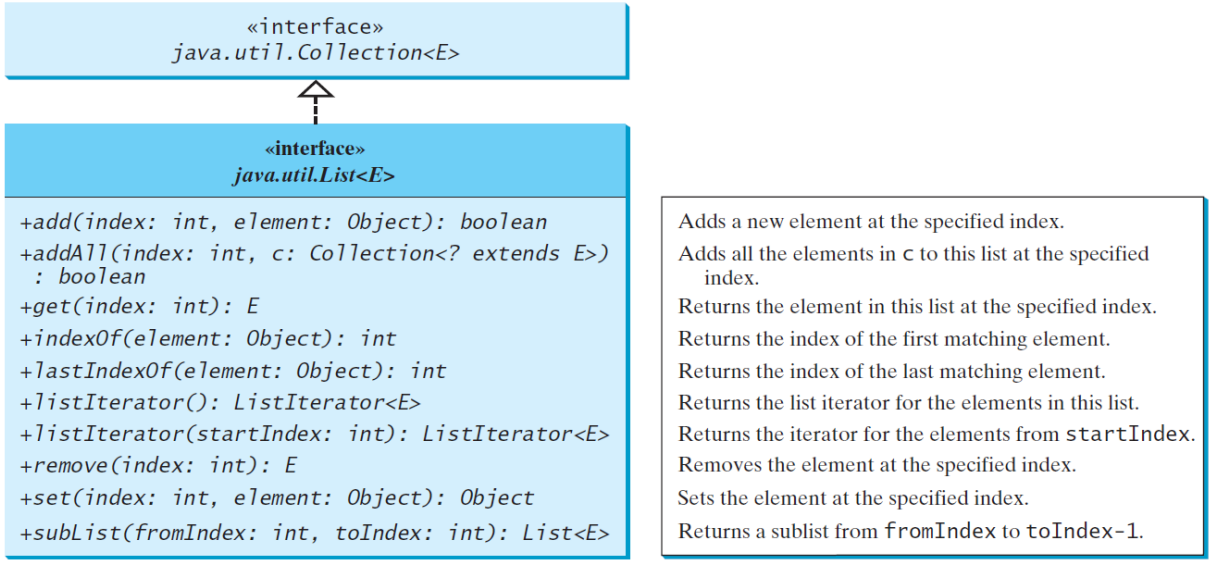


The `List` Interface

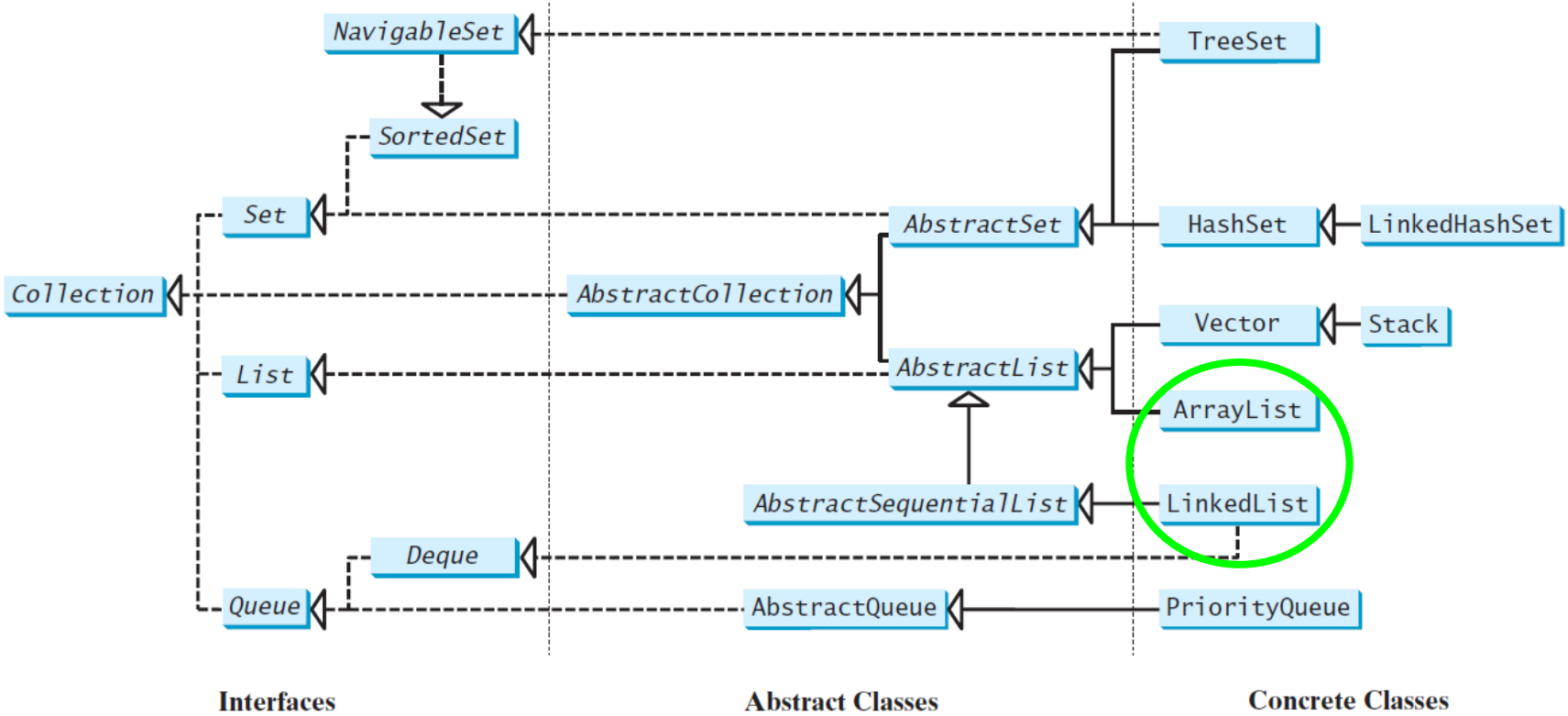


The `List` Interface

- A list stores elements in a sequential order and allows the user to specify where the element is stored. The user can access the elements by index.

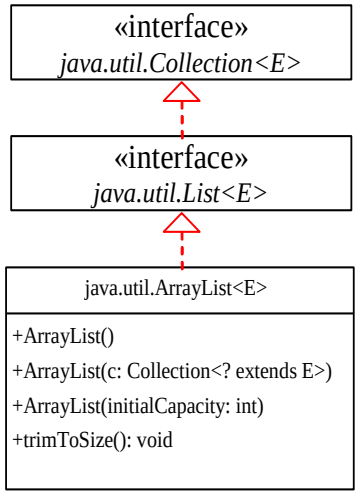


The `ArrayList` and `LinkedList` Classes



The `ArrayList` and `LinkedList` Classes

`java.util.ArrayList`



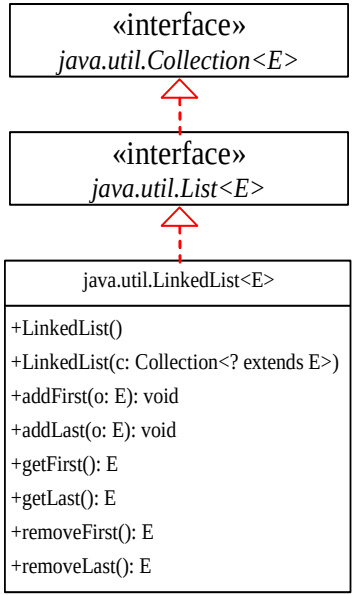
Creates an empty list with the default initial capacity.

Creates an array list from an existing collection.

Creates an empty list with the specified initial capacity.

Trims the capacity of this ArrayList instance to be the list's current size.

`java.util.LinkedList`



Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Returns and removes the last element from this list.

When to use `ArrayList` and `LinkedList`

- The `ArrayList` and `LinkedList` class are concrete implementations of the `List` interface.
  - *) Note: A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.
- Choose which you need based on your task
  - If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection.
  - If your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList.

Example

https://liveexample.pearsoncmg.com/html/TestArrayAndLinkedList.html

```
import java.util.*;

public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<>();
    arrayList.add(1); // 1 is autoboxed to an Integer object
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(1);
    arrayList.add(4);
    arrayList.add(0, 10);
    arrayList.add(3, 30);

    System.out.println("A list of integers in the array list:");
    System.out.println(arrayList);
    LinkedList<Object> linkedList = new LinkedList<>(arrayList);
    linkedList.add(1, "red");
    linkedList.removeLast();
    linkedList.addFirst("green");

    System.out.println("Display the linked list forward:");
    ListIterator<Object> listIterator = linkedList.listIterator();
    while (listIterator.hasNext()) {
      System.out.print(listIterator.next() + " ");
    }
    System.out.println();

    System.out.println("Display the linked list backward:");
    listIterator = linkedList.listIterator(linkedList.size());
    while (listIterator.hasPrevious()) {
      System.out.print(listIterator.previous() + " ");
    }
  }
}
```

Declare as `List` Interface

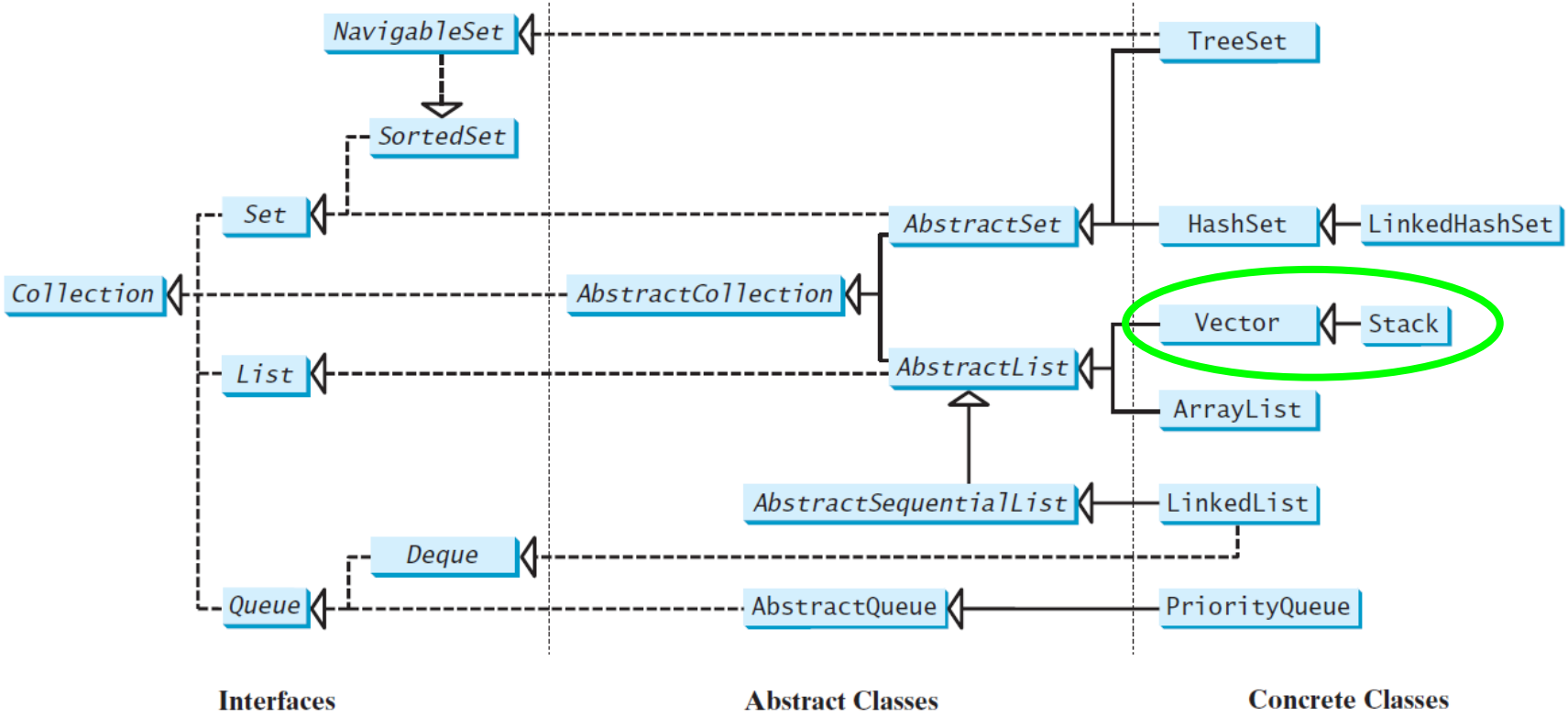
- If you only need the basic functions of add(), get(), remove(), dan set().

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
LinkedList<Integer> linkedList = new LinkedList<Integer>();
```

```
List<Integer> arrayList = new ArrayList<Integer>();
List<Integer> linkedList = new LinkedList<Integer>();
```

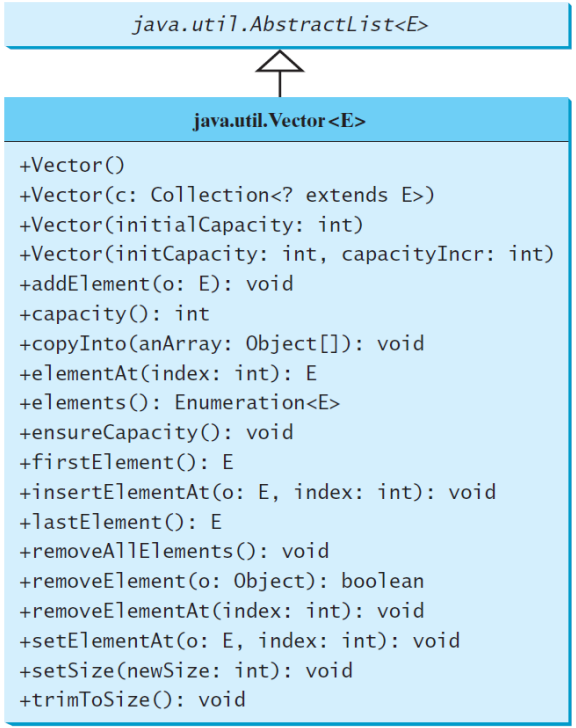
Using Interface References to Collections: Java Best Practices

The `Vector` and `Stack` Classes



The `Vector` Class

- In Java 2, `Vector` is the same as `ArrayList`, except that `Vector` contains the synchronized methods for accessing and modifying the vector.
- For the many applications that do not require synchronization, using ArrayList is more efficient than using Vector.



A Stack of Cookies

- You can add more on top
- You can take one from the top – and eat it!
- Don’t take from the middle or bottom or you would topple the stack!

The `Stack` Class

- A stack is a collection of **last-in-first-out (LIFO)** stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element **only from the top of the stack.**

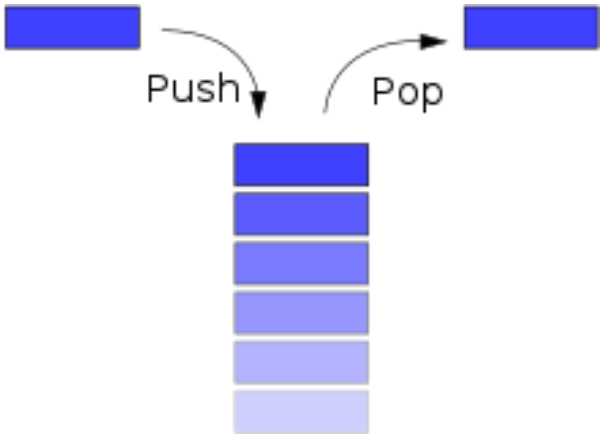
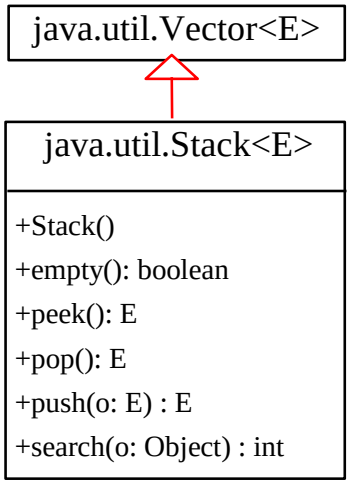


Image source: Wikipedia



Creates an empty stack.
Returns true if this stack is empty.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns the position of the specified element in this stack.

The `Stack` Class: Example

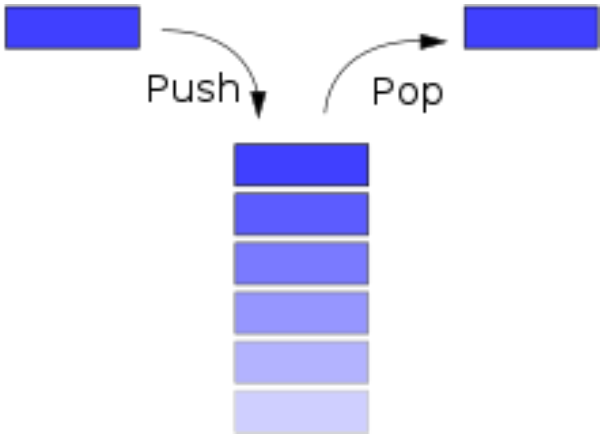
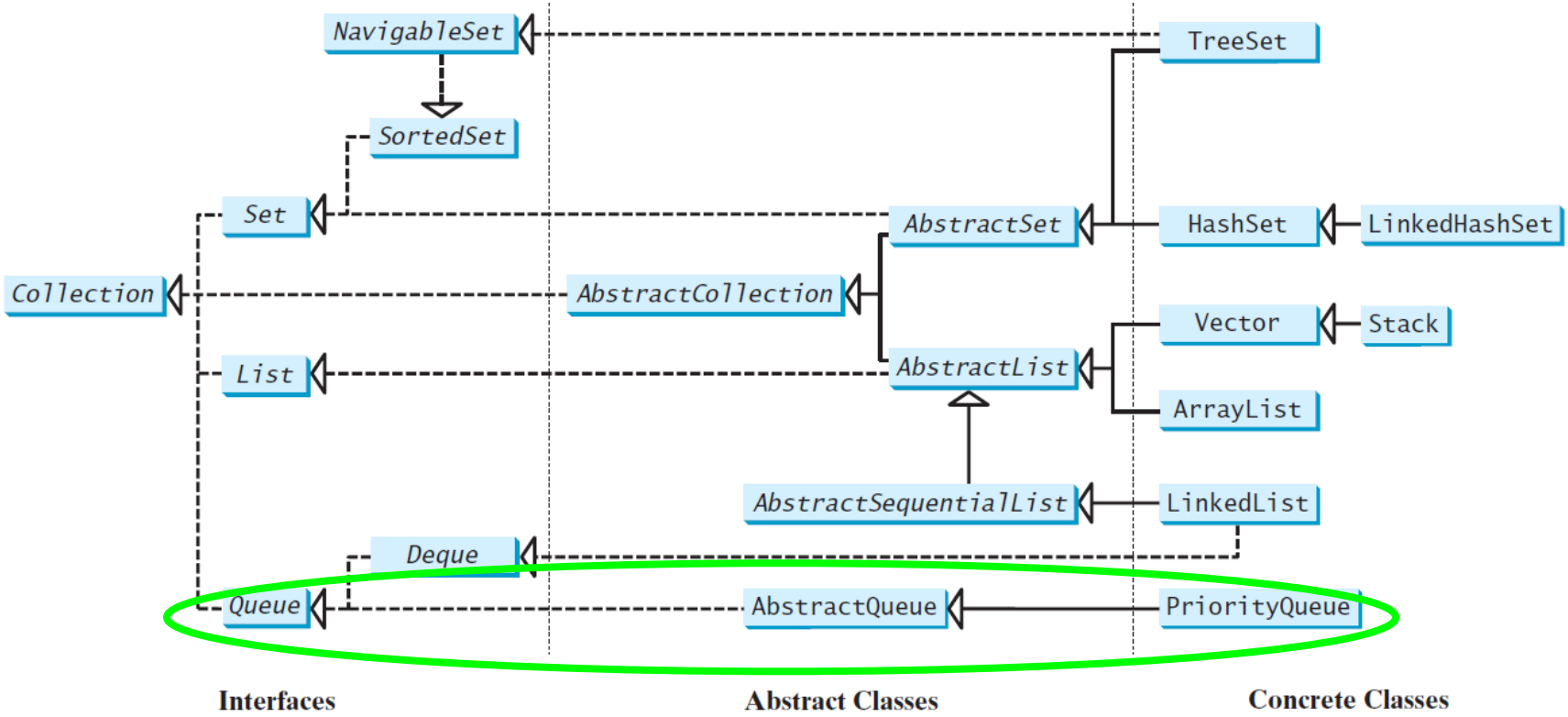


Image source: Wikipedia

```
import java.util.Stack;
public class Test {
   public static void main(String[] args) {
      Stack<Integer> stack = new Stack<Integer>();
      stack.push(10);
      stack.push(20);
      stack.push(30);
      stack.push(40);
      while(!stack.isEmpty()) {
         System.out.println( stack.pop() );
      }
   }
}
```

The `Queue` Interface and the `PriorityQueue` Class



A Queue of People

- More people can line up at the back
- The first person in line can go to checkout – leave the queue – first (FIFO)
- No cutting the line!

The **Queue** Interface

- A queue is a collection of is a **first-in-first-out (FIFO)** data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue.

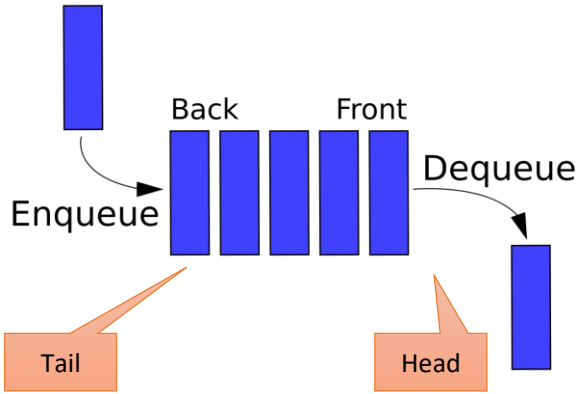
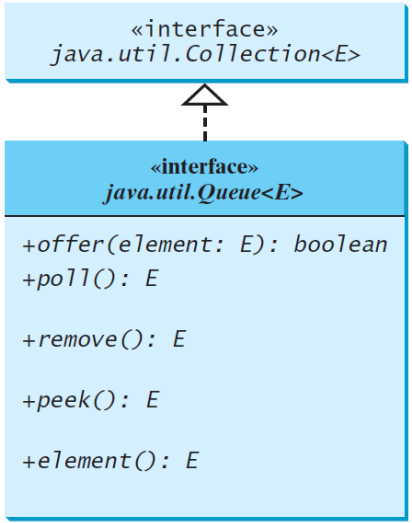


Image source: Wikipedia



Inserts an element into the queue.

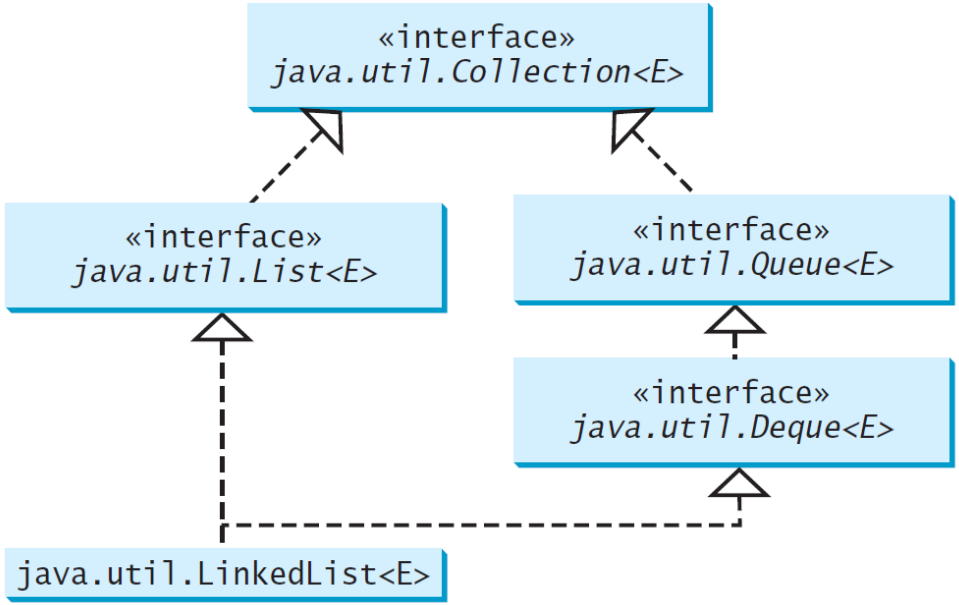
Retrieves and removes the head of this queue, or `null` if this queue is empty.

Retrieves and removes the head of this queue and throws an exception if this queue is empty.

Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.

Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

The `Queue` interface with `LinkedList`



Example

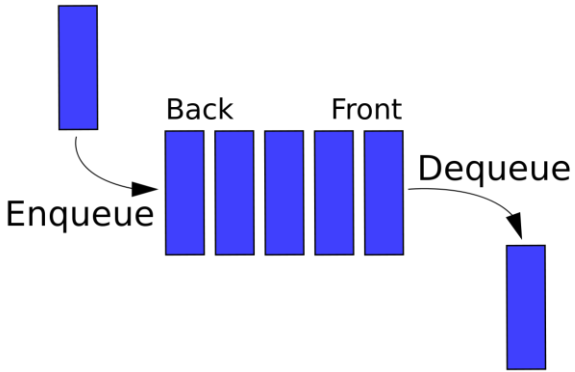
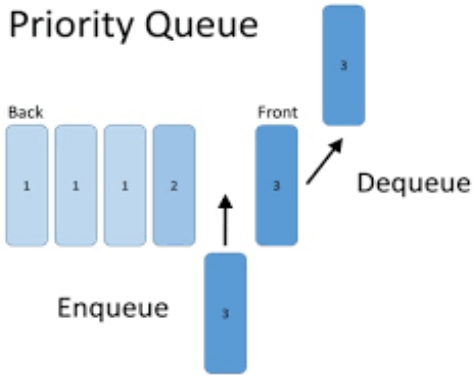


Image source: Wikipedia

```
import java.util.Queue;
import java.util.LinkedList;
public class Test {
   public static void main(String[] args) {
      Queue<Integer> q = new LinkedList<Integer>();
      q.offer(10);
      q.offer(20);
      q.offer(30);
      q.offer(40);
      while(!q.isEmpty()) {
         System.out.println( q.remove() );
      }
   }
}
```

The `PriorityQueue` Class

- In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.



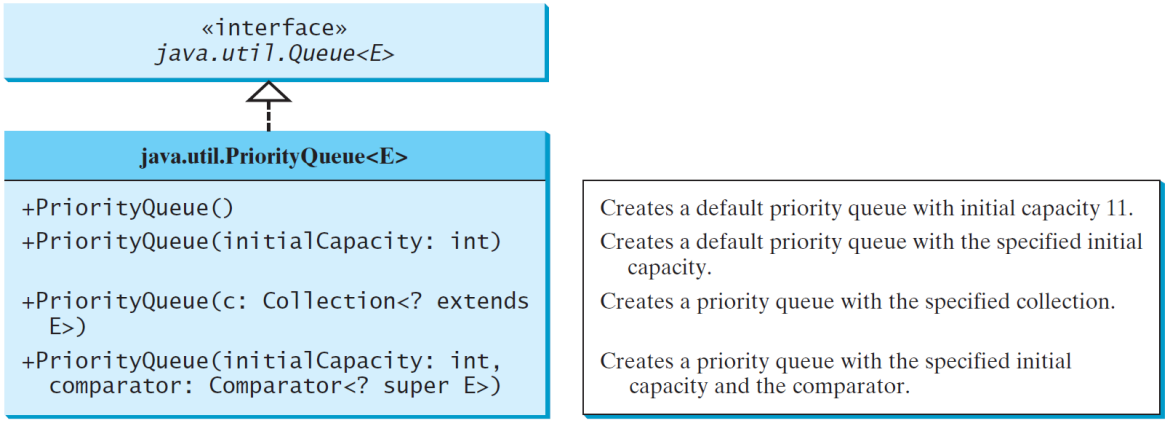


Image source: Wikipedia

Example

```
import java.util.*;

public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<String> queue1 = new
PriorityQueue<>();
    queue1.offer("Oklahoma");
    queue1.offer("Indiana");
    queue1.offer("Georgia");
    queue1.offer("Texas");

    System.out.println("Priority queue
using Comparable:");
    while (queue1.size() > 0) {
      System.out.print(queue1.remove() + "
");
    }
```

```
PriorityQueue<String> queue2 = new PriorityQueue<>(
      4, Collections.reverseOrder());
    queue2.offer("Oklahoma");
    queue2.offer("Indiana");
    queue2.offer("Georgia");
    queue2.offer("Texas");

    System.out.println("\nPriority queue using
Comparator:");
    while (queue2.size() > 0) {
      System.out.print(queue2.remove() + " ");
    }
  }
}
```

Stacks and Queues: What for?

- Queue
  - Event queue of all events, kept by the Java GUI system
  - Queue of print jobs
- Stack
  - Arithmetic Expression Evaluation
  - Run-time stack that a processor or virtual machine keeps to organize the variables of nested methods (Stack of Activation Records)
  - Memory stack for calling recursive methods

Data Structures

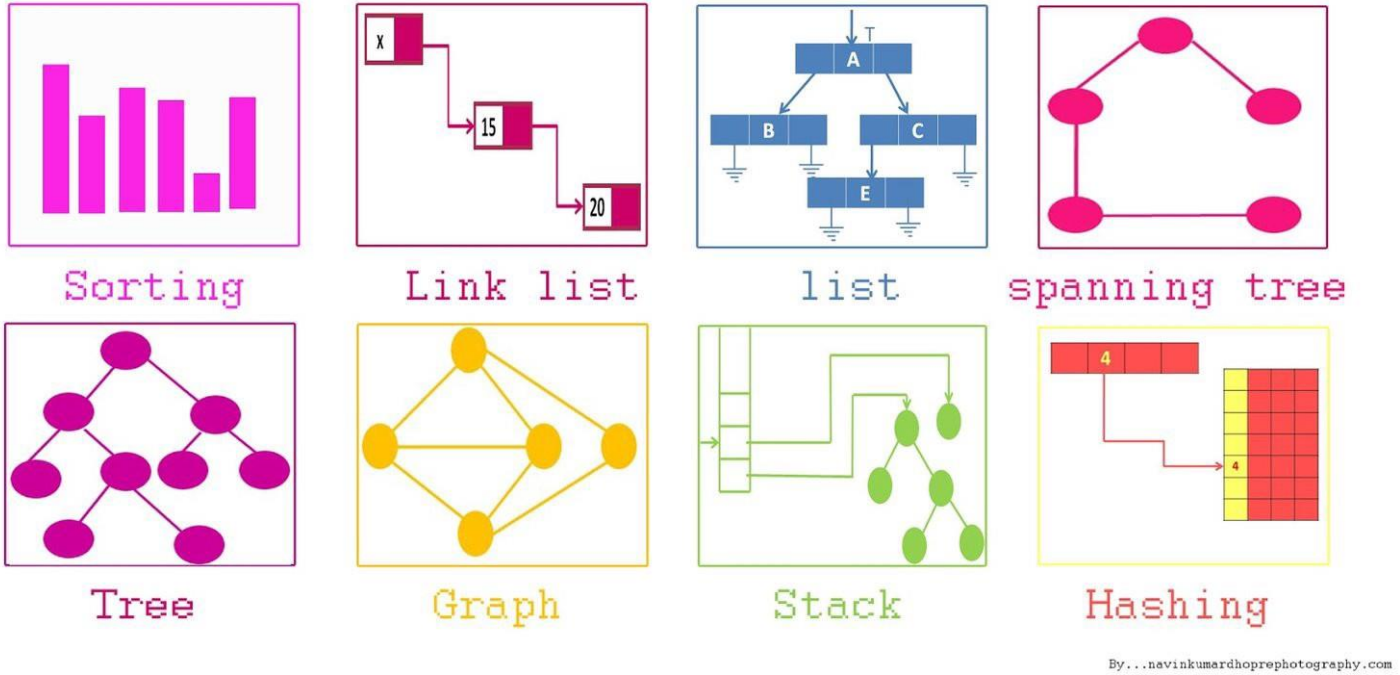


Image credit: https://medium.com/@Arafat.