Java Collections

Containers like List, Stack, Queue have many applications and are commonly used.

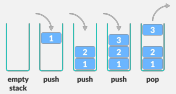
The Java Collections Framework provides implementations of these (and other collections ADTs).

Java Collections – Containers

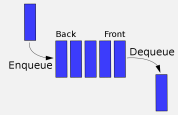
Containers are characterized by the idea of the elements arranged in a line (ordered, not sorted).
- elements accessed by position

- java.util.List<E>
  - supports rank-based operations (by index)
  - only sequential access operations are add (at the end) and iterator

- java.util.Stack<E>
  - add, remove, access elements only at the top



- java.util.Queue<E>
  - add elements at the back, access and remove elements at the front



Generics

The definition of List, Stack, or Queue doesn't depend on the specific type of element in the container.

- java.util.List<E>
  - supports rank-based operations (by index)
  - only sequential access operations are add (at the end) and iterator
- java.util.Stack<E>
  - add, remove, access elements only at the top
- java.util.Queue<E>
  - add elements at the back, access and remove elements at the front

<E> is a type parameter
- defines what kind of element is in the collection
- must be a class type – `Integer`, `Double`, `Boolean`, etc instead of `int`, `double`, `boolean`
- can exploit autoboxing to handle conversions between primitive types and their class versions

- Java supports *parameterized types* where a specific type can be provided when the type is used
  - <E> indicates a class with one type parameter named E
  - E is used in the definition in place of a specific type
  - there's nothing special about E specifically or even a single letter for the type parameter name – convention is that type parameter names are often single letters, and E is a convenient choice to represent an element type

You want to create a list of integers in Java. What should you write? Choose all that apply.

	Answer	Respondents	Percentage
✓	`ArrayList<Integer> list = new ArrayList<Integer> ();`	7	78%
×	`ArrayList<int> list = new ArrayList<int> ();`	0	0%
×	`ArrayList<int[]> list = new ArrayList<int[]>();`	0	0%
×	`ArrayList<Object> list = new ArrayList<Object> ();`	2	22%

legal but not ideal because it requires more casting and you trade compile-time typechecking for runtime typechecking – better to catch errors sooner

Assume the following variable definitions:

```
ArrayList<Integer> numlist = new ArrayList<Integer>();
int x = 5;
```

Which of the following statements are legal syntax? Choose all that apply.

	Answer	Respondents	Percentage
✓	numlist.add(5);	7	32%
×	numlist.add(5.5);	0	0%
×	numlist.add(5.0);	0	0%
×	numlist.add("hello");	1	5%
✓	numlist.add(x);	6	27%
✓	int y = numlist.get(0);	6	27%
×	String y = numlist.get(0);	0	0%
×	String y = (String)numlist.get(0);	2	9%

not legal because `ArrayList<Integer>` means that only integers can be added

not legal because can't cast from `Integer` to `String`

Java Collections – Containers

How can you find out what methods are supported?

google `java 17` *`classname`* to find the API

- `java.util.List<E>`
- `java.util.Stack<E>`
- `java.util.Queue<E>`

java.util.List<E>

Method Summary

boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.

java.util.List<E>

int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

java.util.Stack<E> and java.util.Queue<E>

Method Summary	
boolean	**empty**() Tests if this stack is empty.
E	**peek**() Looks at the object at the top of this stack without removing it from the stack.
E	**pop**() Removes the object at the top of this stack and returns that object as the value of this function.
E	**push**(E item) Pushes an item onto the top of this stack.
int	**search**(Object o) Returns the 1-based position where an object is on this stack.

Method Summary	
boolean	**add**(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
E	**element**() Retrieves, but does not remove, the head of this queue.
boolean	**offer**(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E	**peek**() Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or returns `null` if this queue is empty.
E	**remove**() Retrieves and removes the head of this queue.

Collections Inheritance Hierarchy

- `java.util.Collection<E>`
  - interface
- `java.util.List<E>`
  - abstract class
  - concrete classes are `ArrayList<E>`, `LinkedList<E>`
- `java.util.Stack<E>`
  - concrete class
- `java.util.Queue<E>`
  - interface
  - concrete classes are `ArrayDeque<E>`, `LinkedList<E>`

you can use any of these as types for variables, parameters, return values

`List<Integer> list` is OK

you can't create instances of abstract classes or interfaces, only concrete classes

`new List<Integer>()` is not

Usage

Declare variables, parameters, and return types using the most general type that is appropriate.

- `List<E>`, `Stack<E>`, `Queue<E>`
- code better reflects the actual concepts
- allows for greater flexibility and reusability

New objects can only be created using a concrete class. Choose the appropriate implementation based on the needed operations and their efficiency.

- `ArrayList<E>` vs `LinkedList<E>`, `Stack<E>`, `ArrayDeque<E>` vs `LinkedList<E>`
- linked lists avoid cost of growing/shrinking and shifting, but are inefficient for accessing at a particular index
- a rough guideline: use `ArrayList<E>` for lists and `LinkedList<E>` for queues

Example

- `Reverser`
- `ListDemo`