8.1 Enhanced for loop

Enhanced for loop

The ***enhanced for loop*** is a for loop that iterates through each element in an array or Collection. An enhanced for loop is also known as a ***for each loop***. The enhanced for loop declares a new variable that will be assigned with each successive element of a container object, such as an array or ArrayList. The enhanced for loop only allows elements to be accessed forward from the first element to the last element.

PARTICIPATION ACTIVITY | 8.1.1: The enhanced for loop declares a new variable and assigns the variable with each successive element of the container object.

Animation content:

Static figure:
Begin Java code:

```
ArrayList<String> teamRoster = new ArrayList<String>();

// Adding player names
teamRoster.add("Mike");
teamRoster.add("Scottie");
teamRoster.add("Toni");

System.out.println("Current roster:");

for (String playerName : teamRoster) {
   System.out.println(playerName);
}
```

End Java code.
The line of code, System.out.println("Current roster:");, is highlighted.

There is a box with the line, Current roster:.

A box labeled "Memory" contains five memory addresses, labeled from 96 to 100. Memory address 96 contains the value, Mike, memory address 97 contains the value, Scottie, and memory address 98 contains the value, Toni. All three values are labeled teamRoster.

Step 1: Enhanced for loop iterates through each element of ArrayList. The line of code, ArrayList<String> teamRoster = new ArrayList<String>();, is highlighted and the term, teamRoster, appears next to the line of code and moves to the box labeled "Memory". The line of code,

teamRoster.add("Mike");, is highlighted and the term, Mike, appears next to the line of code and moves to memory address 96 of the box labeled "Memory". The line of code, teamRoster.add("Scottie");, is highlighted and the term, Mike, appears next to the line of code and moves to memory address 97 of the box labeled "Memory". The line of code, teamRoster.add("Toni");, is highlighted and the term, Mike, appears next to the line of code and moves to memory address 98 of the box labeled "Memory". All three values in the box labeled "Memory" are labeled teamRoster. The line of code, System.out.println("Current roster:");, is highlighted and the term, Current roster:, appears next to the line of code and moves into the empty box.

Step 2: Enhanced for loop declares a variable that will be assigned with each element of ArrayList. The line of code, for (String playerName : teamRoster) {, is highlighted and memory address 100 is labeled playerName. The term, "Mike", on memory address 96 duplicates and moves to memory address 100. The line of code, System.out.println(playerName);, is highlighted and the term, Mike, on memory address 100 is duplicated. The duplicated term, Mike, moves next to the line of code, System.out.println(playerName);, and then moves to the box under the term, Current roster:. The line of code, }, is highlighted. The line of code, for (String playerName : teamRoster) {, is highlighted and memory address 100 is labeled playerName. The term, "Scottie", on memory address 96 duplicates and moves to memory address 100. The line of code, System.out.println(playerName);, is highlighted and the term, Scottie, on memory address 100 is duplicated. The duplicated term, Scottie, moves next to the line of code, System.out.println(playerName);, and then moves to the box under the term, Mike. The line of code, }, is highlighted. The line of code, for (String playerName : teamRoster) {, is highlighted and memory address 100 is labeled playerName. The term, "Toni", on memory address 96 duplicates and moves to memory address 100. The line of code, System.out.println(playerName);, is highlighted and the term, Toni, on memory address 100 is duplicated. The duplicated term, Toni, moves next to the line of code, System.out.println(playerName);, and then moves to the box under the term, Scottie.

Animation captions:

1. Enhanced for loop iterates through each element of ArrayList.
2. Enhanced for loop declares a variable that will be assigned with each element of ArrayList.

PARTICIPATION ACTIVITY | 8.1.2: Enhanced for loop.

Refer to the animation above.

1) What variable is assigned with each ArrayList element?

- ○ String
- ○ playerName
- ○ teamRoster

2) How many times does the for loop iterate?

- 1
- 2
- 3

3) What is the value of playerName during the second iteration of the for loop?

- Mike
- Scottie
- Toni

Improved code readability

Compared to a regular for loop, an enhanced for loop decreases the amount of code needed to iterate through containers, thus enhancing code readability and clearly demonstrating the loop's purpose. An enhanced for loop also prevents a programmer from writing code that incorrectly accesses elements outside of the container's range.

Figure 8.1.1: Enhanced for loop decreases the amount of code needed to iterate through ArrayList.

Regular for loop:

```
ArrayList<String> teamRoster = new
ArrayList<String>();
String playerName;
int i;

// Adding player names
teamRoster.add("Mike");
teamRoster.add("Scottie");
teamRoster.add("Toni");

System.out.println("Current roster:
");

for (i = 0; i < teamRoster.size();
++i) {
   playerName = teamRoster.get(i);
   System.out.println(playerName);
}
```

Enhanced for loop:

```
ArrayList<String> teamRoster = new
ArrayList<String>();

// Adding player names
teamRoster.add("Mike");
teamRoster.add("Scottie");
teamRoster.add("Toni");

System.out.println("Current roster:
");

for (String playerName :
teamRoster) {
   System.out.println(playerName);
}
```

PARTICIPATION ACTIVITY | 8.1.3: Using enhanced for loops.

Using an enhanced for loop, complete the code to achieve the stated goal.

1) Calculate the sum of all values within the sensorReadings ArrayList.


```
Double sumVal = 0.0;
for (Double readingVal :
sensorReadings) {
   sumVal =
                          ;
}
```

Check | Show answer

2) Print each Double within the ArrayList interestRates.

```
for (                     :
interestRates) {
   System.out.println(theRate
+ "%");
}
```

Check | Show answer

3) Compute the product of all Integer elements within the ArrayList primeNumbers.

```
Integer primeProduct = 1;
for (Integer theNumber
                          ) {
   primeProduct = primeProduct
* theNumber;
}
```

Check | Show answer

4) Compute the sum of all int elements within an array dailySales declared as: `int[] dailySales = new int[7];`

```
int totalSales = 0;
for (int
) {
   totalSales = totalSales +
numSold;
}
```

Check Show answer

CHALLENGE ACTIVITY 8.1.1: Enhanced for loop.

437612.2739130.qx3zqy7

Start

Type the program's output

```
import java.util.ArrayList;

public class CarStorage {
   public static void main(String[] args) {
      ArrayList<String> myGarage = new ArrayList<String>();

      myGarage.add("Camry");
      myGarage.add("Lancer");
      myGarage.add("Huracan");

      System.out.println("Cars:");

      for (String carName : myGarage) {
         System.out.println(carName);
      }
   }
}
```

```
Cars:
Camry
Lancer
Huracan
```

1 2

Check Next

8.2 List: LinkedList

Linked list container

The ***List*** interface defined within the Java Collections Framework defines a Collection of ordered elements, i.e., a sequence. The List interface supports methods for adding, modifying, and removing elements.

A LinkedList is one of several types that implement the List interface. The LinkedList type is an ADT implemented as a generic class that supports different types of elements. A LinkedList can be declared and created as `LinkedList<T> linkedList = new LinkedList<T>();` where T represents the LinkedList's type, such as Integer or String. The statement `import java.util.LinkedList;` enables use of a LinkedList within a program.

A LinkedList supports insertion of elements either at the end of the list or at a specified index. If an index is not provided, as in `authorList.add("Martin");`, the add() method adds the element at the end of the list. If an index is specified, as in `authorList.add(0, "Butler");`, the element is inserted at the specified index, with the list element previously located at the specified index appearing after the inserted element.

PARTICIPATION ACTIVITY | 8.2.1: LinkedList: add() method adds elements to end of list or at specified index.

Animation content:

Static figure:
Begin Java code:
LinkedList<String> authorsList = new LinkedList<String>();

authorsList.add("Gamow");
authorsList.add("Penrose");
authorsList.add("Hawking");

authorsList.add(1, "Greene");
End Java code.
There is a linked list named authorsList with four nodes. The first node's value is Gamow, the second node's value is Greene, the third node's value is Penrose, and the fourth node's value is Hawking.

Step 1: The add() method adds an element, such as a String, to the end of the list. The line of code, LinkedList<String> authorsList = new LinkedList<String>();, is highlighted and the name, authorsList, appears next to the line of code and moves off the code. The line of code, authorsList.add("Gamow");, is highlighted and a node containing the value "Gamow" appears. The node, "Gamow", moves to the linked list, authorsList, at index 0. The line of code, authorsList.add("Penrose");, is highlighted and a node containing the value "Penrose" appears. The node, "Penrose", moves to the linked list, authorsList, at index 1, and an arrow pointing from node "Gamow" to node "Penrose" appears. The line of code, authorsList.add("Hawking");, is highlighted and a node containing the value "Hawking" appears. The node, "Hawking", moves to the linked list, authorsList, at index 2, and an arrow pointing from node "Penrose" to node "Hawking" appears.

Step 2: Using the add() method with an index inserts an element at the specified index. The line of code, authorsList.add(1, "Greene");, is highlighted and the term, index: 0, appears next to the node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Penrose" and changes to "index: 1". The nodes, "Penrose" and "Hawking", move down and a new node "Greene" appears next to the line of code, authorsList.add(1, "Greene");, and moves next to the term, "index: 1". The arrow from node "Gamow" now points to node "Greene", and a new arrow pointing from node "Greene" to node "Penrose" appears.

Animation captions:

1. The add() method adds an element, such as a String, to the end of the list.
2. Using the add() method with an index inserts an element at the specified index.

PARTICIPATION ACTIVITY 8.2.2: LinkedList add() method.

Given the following code that creates and initializes a LinkedList:

```
LinkedList<String> wordsFromFile = new LinkedList<String>();
wordsFromFile.add("The");
wordsFromFile.add("fowl");
wordsFromFile.add("is");
wordsFromFile.add("the");
wordsFromFile.add("term");
```

1) At what index does the following statement insert the word "end"? Enter a number.

```
wordsFromFile.add("end");
```

Check Show answer

2) Given the original list initialization above, how many elements does wordsFromFile contain after the following statement? Enter a number.

```
wordsFromFile.add(4, "big");
```

Check Show answer

3) Write a statement to insert the word "not" between the elements "is" and "the".

```
wordsFromFile.add(
                  );
```

Check | Show answer

Common LinkedList methods

The get() method returns the element at the specified index. To access the specified list element, the get() method will start at the first list element and then traverse the list element by element until the specified index is reached. Ex: `playerName.get(3)` starts at the element located at index 0, moves to the element at index 1, then moves to index 2, and finally moves to index 3 returning that element.

The set() method replaces the element at the specified location with a new element.

PARTICIPATION ACTIVITY | 8.2.3: LinkedList: get() returns elements at specified location and set() replaces element at specified location.

Animation content:

Static figure:
Begin Java code:
System.out.println(authorsList.get(2));

authorsList.set(2, "Sagan");

System.out.println(authorsList.get(2));
End Java code.
There is a linked list named authorsList with three nodes. The first node's value is Gamow, the second node's value is Greene, and the third node's value is Sagan. There is a box with the lines: Penrose Sagan

Step 1: The get() method searches through the LinkedList to return the element at the specified index. The line of code, System.out.println(authorsList.get(2));, is highlighted and the term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Greene" and changes to "index: 1". The term, index: 1, moves down to node "Penrose" and changes to "index: 2". The value, Penrose, duplicates, moves to the line of code, System.out.println(authorsList.get(2));, and moves to the empty box.

Step 2: The set() method replaces the element at the specified position. The line of code,

authorsList.set(2, "Sagan");, is highlighted and the term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Greene" and changes to "index: 1". The term, index: 1, moves down to node "Penrose" and changes to "index: 2". The value, Sagan, appears at the line of code, authorsList.set(2, "Sagan");, and moves to replace the value in the node, Penrose. The line of code, System.out.println(authorsList.get(2));, and the term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Greene" and changes to "index: 1". The term, index: 1, moves down to node "Sagan" and changes to "index: 2". The value Sagan duplicates and moves to the box under the term "Penrose".

Animation captions:

1. The get() method searches through the LinkedList to return the element at the specified index.
2. The set() method replaces the element at the specified position.

The remove() method removes the element at the specified index.

PARTICIPATION ACTIVITY 8.2.4: LinkedList: remove() method removes element from specified position.

Animation content:

Static figure:
Begin Java code:
System.out.println(authorsList.get(1));

authorsList.remove(1);

System.out.println(authorsList.get(1));
End Java code.
There is a linked list named authorsList with two nodes. The first node's value is Gamow, the second node's value is Penrose. There is a box with the lines: Greene
Penrose

Step 1: The remove() method removes the element at the specified position. The line of code, System.out.println(authorsList.get(1));, is highlighted, and the term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Greene" and changes to "index: 1". The value, Greene, duplicates, moves to the line of code, System.out.println(authorsList.get(1));, and moves to the empty box. authorsList.remove(1);, is highlighted, and the term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Greene" and changes to "index: 1". The node, Greene, is deleted and an arrow from node "Gamow" to node "Penrose" appears. System.out.println(authorsList.get(1));, is highlighted, and the term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Penrose" and changes to "index: 1". The value, Greene,

duplicates, moves to the line of code, System.out.println(authorsList.get(1));, and moves to the box under the term Greene.

Animation captions:

1. The remove() method removes the element at the specified position.

The LinkedList class implements several methods, defined in the List interface, for getting information about a list and accessing and modifying a list's elements.

Table 8.2.1: Common LinkedList methods.

get()	`get(index)` Returns element at specified index.	`// Assume List is: 5, 10, 11` `exList.get(2);  // Returns 11`

set()	`set(index, newElement)`
	
	Replaces element at specified index with newElement. Returns element previously at specified index.
		`// Assume List is: 5, 8, 11`
		`exList.set(0, new Integer(3)); // Returns 5`
		`// List is now: 3, 8, 11`

add()	`add(newElement)`
	
	Adds newElement to the end of the List. List's size is increased by one.
	
	`add(index, newElement)`
	
	Adds newElement to the List at the specified index. Indices of the elements previously at that specified index and higher are increased by one. List's size is increased by one.
		`// Assume List is empty`
		`exList.add(new Integer(7));  // List is: 7`
		`exList.add(14);  // List is: 7, 14`
		
		`exList.add(0, new Integer(21));  // List is: 21, 7, 14`
		`exList.add(2, 9);  // List is: 21, 7, 9, 14`

size()	`size()`
	
	Returns the number of elements in the List.
		`// Assume List is empty`
		`exList.size(); // Returns 0`
		
		`exList.add(8);`
		`exList.add(22);`
		`// List is now: 8, 22`
		
		`exList.size(); // Returns 2`

remove()	`remove(index)`
	
	Removes element at specified index. Indices for elements from higher positions are decreased by one. List size is decreased by one. Returns reference to element removed from List.
	
	`remove(existingElement)`
	
	Removes the first occurrence of an element which is equal to existingElement. Indices for elements from higher positions are decreased by one. List size is decreased by one. Returns true if specified element was found and removed.
		`// Assume List is: 22, 8, 4, 4`
		`exList.remove(1); // Returns 8`
		`// List is now: 22, 4, 4`
		
		`exList.remove(5); // Throws exception`
		`// List is still: 22, 4, 4`
		
		`exList.remove(2); // Returns 4`
		`// List is now: 22, 4`

PARTICIPATION ACTIVITY 8.2.5: Using LinkedList's get(), set(), and remove() methods.

Given the following code that creates and initializes a LinkedList:

```
LinkedList<Double> accelerometerValues = new LinkedList<Double>();
accelerometerValues.add(9.8);
accelerometerValues.add(10.2);
accelerometerValues.add(15.4);
```


1) Complete the statement to print the first list element.

```
System.out.println(accelerometerValues.
                   );
```

Check Show answer

2) Complete the statement to assign currentValue with the element at index 2.

```
currentValue = accelerometerValues.___;
```


Check Show answer

3) Complete the statement to update the element at index 1 to 10.6.

```
accelerometerValues.___;
```

Check Show answer

4) Write a statement to remove the value 9.8 from the list.

```
accelerometerValues.___;
```

Check Show answer

Iterating through a list

LinkedList methods with index parameters, such as get() or set(), cause the list to be traversed from the first element to the specified element each time the method is called. Thus, using the LinkedLists' get() or set() methods within a loop that iterates through all list elements is inefficient.

PARTICIPATION ACTIVITY 8.2.6: Using get() within a loop that iterates through all LinkedList elements is inefficient.

Animation content:

Static figure:
Begin Java code:
LinkedList<String> authorsList = new LinkedList<String>();
String authorName;

```
int i;

authorsList.add("Gamow");
authorsList.add("Greene");
authorsList.add("Penrose");

for (i = 0; i < authorsList.size(); ++i) {
   authorName = authorsList.get(i);
   System.out.println(authorName);
}
```


End Java code.

There is a linked list in an undefined memory address authorsList with three nodes. The first node's value is "Gamow", the second node's value is "Greene", and the third node's value is "Penrose". Three dots lead from the memory address labeled authorList to two new undefined memory addresses. The first memory address is labeled authorName and contains the value, Penrose. The second memory address is labeled i and contains the value, 2. Three dots trail the third memory address. There is a box that contains the values, "Gamow", "Greene", and "Penrose".

Step 1: The get() method always traverses the list sequentially from the first element to the desired element. The line of code, LinkedList<String> authorsList = new LinkedList<String>();, is highlighted and the first of three memory addresses is labeled authorsList. The line of code, String authorName;, is highlighted, and the second memory address is labeled "authorName". The line of code, int i;, is highlighted, and the third memory address is labeled "i". The lines of code, authorsList.add("Gamow");
authorsList.add("Greene");
authorsList.add("Penrose");,
Are highlighted and three nodes, "Gamow", "Greene", and "Penrose" are added to the memory address labeled "authorList". There is an arrow pointing from node "Gamow" to node "Greene", and an arrow pointing from node "Greene" to node "Penrose". The line of code, for (i = 0; i < authorsList.size(); ++i) {, is highlighted, the value, 0, appears at the line of code, for (i = 0; i < authorsList.size(); ++i) {, and moves to the memory address labeled, i. The line of code, authorName = authorsList.get(i);, is highlighted, and the value, 0, at memory address labeled "i" is duplicated and moves to the line of code, authorName = authorsList.get(i);. The term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The value, "Gamow", duplicates, moves to the line of code, authorName = authorsList.get(i);, and moves to the memory address labeled "authorName". The line of code, System.out.println(authorName);, is highlighted, and the value, Gamow, at memory address labeled "authorName" is highlighted, moves to the line of code, System.out.println(authorName);, and moves to the empty box. The line of code, {, is highlighted. The line of code, for (i = 0; i < authorsList.size(); ++i) {, is highlighted, the value, 1, appears at the line of code, for (i = 0; i < authorsList.size(); ++i) {, and moves to the memory address labeled, i. The line of code, authorName = authorsList.get(i);, is highlighted, and the value, 1, at memory address labeled "i" is duplicated and moves to the line of code, authorName = authorsList.get(i);. The term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Greene" and changes to "index: 1". The value, "Greene", duplicates, moves to the line of code, authorName = authorsList.get(i);, and moves to the memory address labeled "authorName". The line of code,

System.out.println(authorName);, is highlighted, and the value, Greene, at memory address labeled "authorName" is highlighted, moves to the line of code, System.out.println(authorName);, and moves to the box under the value, Gamow. The line of code, {, is highlighted. The line of code, for (i = 0; i < authorsList.size(); ++i) {, is highlighted, the value, 2, appears at the line of code, for (i = 0; i < authorsList.size(); ++i) {, and moves to the memory address labeled, i. The line of code, authorName = authorsList.get(i);, is highlighted, and the value, 2, at memory address labeled "i" is duplicated and moves to the line of code, authorName = authorsList.get(i);. The term, index: 0, appears next to node "Gamow" in the linked list, authorsList. The term, index: 0, moves down to node "Greene" and changes to "index: 1". The term, index: 1, moves down to node "Penrose" and changes to "index: 2". The value, "Penrose", duplicates, moves to the line of code, authorName = authorsList.get(i);, and moves to the memory address labeled "authorName". The line of code, System.out.println(authorName);, is highlighted, and the value, Penrose, at memory address labeled "authorName" is highlighted, moves to the line of code, System.out.println(authorName);, and moves to the box under the value, Greene. The line of code, {, is highlighted.

Animation captions:

1. The get() method always traverses the list sequentially from the first element to the desired element.

Efficient iteration through a LinkedList necessitates keeping track of the current position in the loop without using an index. A ***ListIterator*** is an object that points to a location in a List and provides methods to access an element and advance the ListIterator to the next position in the list.

LinkedList's listIterator() method returns a ListIterator object for traversing a list. The statement `import java.util.ListIterator;` enables use of a ListIterator in a program.

PARTICIPATION ACTIVITY 8.2.7: ListIterator provides methods to access elements of a LinkedList.

Animation content:

Static figure:
Begin Java code:

```
LinkedList<String> authorsList = new LinkedList<String>();
String authorName;
ListIterator<String> listIterator;

authorsList.add("Gamow");
authorsList.add("Greene");
authorsList.add("Penrose");

listIterator = authorsList.listIterator();

while (listIterator.hasNext()) {
```

```
  authorName = listIterator.next();
  System.out.println(authorName);
}
```
End Java code.
There are undefined memory addresses sequentially labeled, "authorList", "authorName", and "listIterator". There is a linked list with three nodes in the memory address labeled, authorsList. The first node's value is "Gamow", the second node's value is "Greene", and the third node's value is "Penrose". The memory address labeled "AuthorName" contains the value, Penrose. The memory address labeled listIterator contains an arrow pointing from the memory address labeled "listIterator" to the end of the linked list at memory address labeled "authorList".

Step 1: listIterator() method returns a ListIterator object for iterating through list elements. The line of code, LinkedList<String> authorsList = new LinkedList<String>();, is highlighted and the first of three memory addresses is labeled authorsList. The line of code, String authorName;, is highlighted, and the second memory address is labeled "authorName". The line of code, ListIterator<String> listIterator;, is highlighted, and the third memory address is labeled "listIterator". The lines of code,
authorsList.add("Gamow");
authorsList.add("Greene");
authorsList.add("Penrose");,
are highlighted and three nodes, "Gamow", "Greene", and "Penrose" are added to the memory address labeled "authorList". There is an arrow pointing from node "Gamow" to node "Greene", and an arrow pointing from node "Greene" to node "Penrose". The line of code, listIterator = authorsList.listIterator();, is highlighted, and an arrow pointing from memory address labeled, "listIterator" to the start of memory address "authorList" appears.

Step 2: ListIterator's hasNext() method returns true if the list has more elements. Otherwise, returns false. The line of code, while (listIterator.hasNext()) {, is highlighted, the value, true, appears at the end of the arrow from memory address "listIterator" to memory address "authorList", and the value "true" moves to the line of code, while (listIterator.hasNext()) {.

Step 3: ListIterator's next() method returns the next element in the list and moves the iterator to the next location. The line of code, authorName = listIterator.next();, is highlighted and the value "Gamow" of the first node of the linked list duplicates, moves to the line of code, authorName = listIterator.next();, and moves to the memory address labeled authorName. The arrow from memory address "listIterator" to memory address "authorList" moves from pointing to the start of the memory address labeled "authorList", to between nodes "Gamow" and "Greene". The line of code, System.out.println(authorName);, is highlighted, and the value, Gamow, at memory address labeled "authorName" is duplicated, moves to the line of code, System.out.println(authorName);, and moves to the empty box. The line of code, {, is highlighted. The line of code, while (listIterator.hasNext()) {, is highlighted, the value, true, appears at the end of the arrow from memory address "listIterator" to memory address "authorList", and the value "true" moves to the line of code, while (listIterator.hasNext()) {. The line of code, authorName = listIterator.next();, is highlighted and the value "Greene" of the second node of the linked list duplicates, moves to the line of code, authorName = listIterator.next();, and moves to the memory address labeled authorName. The arrow from memory

address "listIterator" to memory address "authorList" moves from pointing to between nodes "Gamow" and "Greene", to between nodes "Greene" and "Penrose". The line of code, System.out.println(authorName);, is highlighted, and the value, Greene, at memory address labeled "authorName" is duplicated, moves to the line of code, System.out.println(authorName);, and moves to the box under the value, Gamow. The line of code, {, is highlighted. The line of code, while (listIterator.hasNext()) {, is highlighted, the value, true, appears at the end of the arrow from memory address "listIterator" to memory address "authorList", and the value "true" moves to the line of code, while (listIterator.hasNext()) {. The line of code, authorName = listIterator.next();, is highlighted and the value "Penrose" of the third node of the linked list duplicates, moves to the line of code, authorName = listIterator.next();, and moves to the memory address labeled authorName. The arrow from memory address "listIterator" to memory address "authorList" moves from pointing to between nodes "Greene" and "Gamow", to after node "Penrose". The line of code, System.out.println(authorName);, is highlighted, and the value, Penrose, at memory address labeled "authorName" is duplicated, moves to the line of code, System.out.println(authorName);, and moves to the box under the value, Greene. The line of code, {, is highlighted. The line of code, while (listIterator.hasNext()) {, is highlighted, the value, false, appears at the end of the arrow from memory address "listIterator" to memory address "authorList", and the value "false" moves to the line of code, while (listIterator.hasNext()) {. The line of code, {, is highlighted.

Animation captions:

1. listIterator() method returns a ListIterator object for iterating through list elements.
2. ListIterator's hasNext() method returns true if the list has more elements. Otherwise, returns false.
3. ListIterator's next() method returns the next element in the list and moves the iterator to the next location.

A ListIterator is located between two elements in a list. The next() method returns the next element in the list and moves the ListIterator to the next location. The hasNext() method returns true if there is a next element, and false otherwise. *A good practice is to always call hasNext() before calling next() to ensure a list element exists.*

The ListIterator's set() method replaces the last element accessed by the iterator, e.g., the element returned by the prior next() call.

PARTICIPATION ACTIVITY | 8.2.8: ListIterator's set() method replaces the prior element accessed by the iterator.

Animation content:

Static figure:
Begin Java code:
LinkedList<String> authorsList = new LinkedList<String>();
String authorName;

```
String upperCaseName;
ListIterator<String> listIterator;

authorsList.add("Gamow");
authorsList.add("Greene");
authorsList.add("Penrose");

listIterator = authorsList.listIterator();
while (listIterator.hasNext()) {
   authorName = listIterator.next();
   upperCaseName = authorName.toUpperCase();
   listIterator.set(upperCaseName);
}

listIterator = authorsList.listIterator();
while (listIterator.hasNext()) {
   authorName = listIterator.next();
   System.out.println(authorName);
}
```


End Java code.

There are four undefined memory addresses sequentially labeled, "authorList", "authorName", "upperCaseName", and "listIterator". There is a linked list with three nodes in the memory address labeled, authorsList. The first node's value is "GAMOW", the second node's value is "GREENE", and the third node's value is "PENROSE". The memory address labeled "AuthorName" contains the value, Penrose. The memory address labeled "upperCaseName" contains the value, Penrose. The memory address labeled listIterator contains an arrow pointing from the memory address labeled "listIterator" to the end of the linked list at memory address labeled "authorList".

Step 1: ListIterator's set() method replaces the last element accessed by the iterator, such as the prior next() call. The line of code, while (listIterator.hasNext()) {, is highlighted, the value, true, appears at the end of the arrow from memory address "listIterator" to memory address "authorList", and the value "true" moves to the line of code, while (listIterator.hasNext()) {. The line of code, authorName = listIterator.next();, is highlighted and the value "Gamow" of the first node of the linked list duplicates, moves to the line of code, authorName = listIterator.next();, and moves to the memory address labeled authorName. The arrow from memory address "listIterator" to memory address "authorList" moves from pointing to the start of the memory address labeled "authorList", to between nodes "Gamow" and "Greene". The line of code, upperCaseName = authorName.toUpperCase();, is highlighted, The value, Gamow, at memory address labeled "authorName" duplicates, moves to the line of code, upperCaseName = authorName.toUpperCase();, changes to all uppercase letters, and moves to the memory address labeled "upperCaseName". The line of code, listIterator.set(upperCaseName);, is highlighted. The value "GAMOW" at memory address labeled "upperCaseName" duplicates, moves to the line of code, listIterator.set(upperCaseName);, and moves to the node "Gamow". The linked list is now "GAMOW" in all capital letters, "Greene", and "Penrose". The line of code, {, is highlighted. The line of code, while (listIterator.hasNext()) {, is highlighted, the

value, true, appears at the end of the arrow from memory address "listIterator" to memory address "authorList", and the value "true" moves to the line of code, while (listIterator.hasNext()) {. The line of code, authorName = listIterator.next();, is highlighted and the value "Greene" of the second node of the linked list duplicates, moves to the line of code, authorName = listIterator.next();, and moves to the memory address labeled authorName. The arrow from memory address "listIterator" to memory address "authorList" moves from pointing to between nodes "Gamow" and "Greene", to between nodes "Greene" and "Penrose". The line of code, upperCaseName = authorName.toUpperCase();, is highlighted, the value, Greene, at memory address labeled "authorName" duplicates, moves to the line of code, upperCaseName = authorName.toUpperCase();, changes to all uppercase letters, and moves to the memory address labeled "upperCaseName". The line of code, listIterator.set(upperCaseName);, is highlighted. The value "GAMOW" at memory address labeled "upperCaseName" duplicates, moves to the line of code, listIterator.set(upperCaseName);, and moves to the node "Greene". The linked list is now "GAMOW" in all capital letters, "GREENE" in all capital letters, and "Penrose". The line of code, {, is highlighted. The line of code, while (listIterator.hasNext()) {, is highlighted, the value, true, appears at the end of the arrow from memory address "listIterator" to memory address "authorList", and the value "true" moves to the line of code, while (listIterator.hasNext()) {. The line of code, authorName = listIterator.next();, is highlighted and the value "Penrose" of the third node of the linked list duplicates, moves to the line of code, authorName = listIterator.next();, and moves to the memory address labeled authorName. The arrow from memory address "listIterator" to memory address "authorList" moves from pointing to between nodes "Greene" and "Penrose", to after node "Penrose". The line of code, upperCaseName = authorName.toUpperCase();, is highlighted, the value, Penrose, at memory address labeled "authorName" duplicates, moves to the line of code, upperCaseName = authorName.toUpperCase();, changes to all uppercase letters, and moves to the memory address labeled "upperCaseName". The line of code, listIterator.set(upperCaseName);, is highlighted. The value "PENROSE" at memory address labeled "upperCaseName" duplicates, moves to the line of code, listIterator.set(upperCaseName);, and moves to the node "Penrose". The linked list is now "GAMOW" in all capital letters, "GREENE" in all capital letters, and "PENROSE" in all capital letters. The line of code, {, is highlighted. The line of code, while (listIterator.hasNext()) {, is highlighted, the value, false, appears at the end of the arrow from memory address "listIterator" to memory address "authorList", and the value "false" moves to the line of code, while (listIterator.hasNext()) {. The line of code, {, is highlighted.

Step 2: All strings in the list have been replaced with uppercase versions. The line of code, listIterator = authorsList.listIterator();, is highlighted and the arrow at the memory address labeled "listIterator" now points the the start of the memory address labeled "authorList". The lines of code, while (listIterator.hasNext()) {

```
  authorName = listIterator.next();
  System.out.println(authorName);
},
```

Are highlighted, and the lines, GAMOW
GREENE
PENROSE,
appear in the empty box. The arrow from memory address "listIterator" to memory address

"authorList" moves from pointing to the start of memory address ;abe;ed "authorList" to after node "Penrose".

Animation captions:

1. ListIterator's set() method replaces the last element accessed by the iterator, such as the prior next() call.
2. All strings in the list have been replaced with uppercase versions.

The ListIterator class implements several methods for traversing a list and accessing and modifying a list's elements.

Table 8.2.2: Common ListIterator methods.

next()	`next()`
	
	Returns the next element in the List and moves the ListIterator after that element.
		`// Assume exList is:  3 4`
		`exIterator = exList.listIterator();`
		`// exList is now:  3 4`
		`// Iterator position: ^`
		
		`exIterator.next(); // Returns 3`
		`// exList is now:  3 4`
		`// Iterator position:  ^`

nextIndex()	`nextIndex()`
	
	Returns the index of the next element.
		`// Assume exList is:  3 4`
		`// Iterator position: ^`
		
		`exIterator.nextIndex(); // Returns 0`
		
		`exIterator.next();  // Returns 3`
		`// exList is now:  3 4`
		`// Iterator position:  ^`
		
		`exIterator.nextIndex(); // Returns 1`

previous()	`previous()`
	
	Returns the previous element in the List and moves the ListIterator before that element.
		`// Assume exList is:  3 4`
		`exIterator = exList.listIterator();`
		`// exList is now:  3 4`
		`// Iterator position: ^`
		
		`exIterator.next();  // Returns 3`
		`// exList is now:  3 4`
		`// Iterator position:  ^`
		
		`exIterator.previous(); // Returns 3`
		`// exList is now:  3 4`
		`// Iterator position: ^`

previousIndex()	`previousIndex()`
	
	Returns the index of the previous element.
		`// Assume exList is:  3 4 5`
		`// Iterator position:  ^`
		
		`exIterator.previousIndex(); // Returns 2`
		
		`exIterator.previous();  // Returns 5`
		`// exList is now:  3 4 5`
		`// Iterator position:  ^`
		
		`exIterator.previousIndex(); // Returns 1`

hasNext()	`hasNext()`
	
	Returns true if ListIterator has a next element. Otherwise, returns false.
		`// Assume exList is:  3 4`
		`// Iterator position:  ^`
		`exIterator.hasNext(); // Returns true`
		
		`exIterator.next(); // Returns 4`
		`// exList is now:  3 4`
		`// Iterator position:  ^`
		
		`exIterator.hasNext(); // Returns false`

hasPrevious()	`hasPrevious()`
	
	Returns true if ListIterator has a previous element. Otherwise, returns false.
		`// Assume exList is:  3 4`
		`// Iterator position:  ^`
		`exIterator.hasPrevious(); // Returns true`
		
		`exIterator.previous(); // Returns '3'`
		`// exList is now:  3 4`
		`// Iterator position: ^`
		
		`exIterator.hasPrevious(); // Returns false`

add()	`add(newElement)`
	
	Adds the newElement between the next and previous elements and moves the ListIterator after newElement.
		`// Assume exList is:  4 5`
		`// Iterator position: ^`
		`exIterator.add(3);`
		`// exList is now:  3 4 5`
		`// Iterator position:  ^`
		
		`exIterator.add(11);`
		`// exList is now:  3 11 4 5`
		`// Iterator position:  ^`

remove()	`remove()`
	
	Removes the element returned by the prior call to next() or previous(). Fails if used more than once per call to next() or previous(). Fails if add() has already been called since the last call to next() or previous().
		`// Assume exList is:  3 4 5`
		`// Iterator position:  ^`
		`exIterator.next();  // Returns 4`
		`// exList is now:  3 4 5`
		`// Iterator position:  ^`
		
		`exIterator.remove();  // Removes 4`
		`// exList is now:  3 5`
		`// Iterator position:  ^`
		
		`exIterator.previous();  // Returns 3`
		`// exList is now:  3 5`
		`// Iterator position:  ^`
		`exIterator.remove();  // Removes 3`
		`// exList is now:  5`
		`// Iterator position:  ^`

set()	`set(newElement)`
	
	Replaces the element returned by the prior call next() or previous() with newElement.
		`// Assume exList is:  3 4 5`
		`// Iterator position:  ^`
		`exIterator.next();  // Returns 4`
		`// exList is now:  3 4 5`
		`// Iterator position:  ^`
		
		`exIterator.set(44);`
		`// exList is now:  3 44 5`
		`// Iterator position:  ^`
		`exIterator.set(21);`
		`// exList is now:  3 21 5`
		`// Iterator position:  ^`
		
		`exIterator.previous();  // Returns 21`
		`// exList is now:  3 21 5`
		`// Iterator position:  ^`
		
		`exIterator.set(15);`
		`// exList is now:  3 15 5`
		`// Iterator position:  ^`

PARTICIPATION ACTIVITY | 8.2.9: Using ListIterator's next(), hasNext(), and set() methods to traverse and modify a List.

Answer the questions given the following code that creates and initializes a LinkedList and a ListIterator. For every question, assume that the ListIterator is located before the first list element (i.e., the starting position).

```
LinkedList<Integer> numbersList = new LinkedList<Integer>();
ListIterator<Integer> numberIterator;

numbersList.add(3);
numbersList.add(1);
numbersList.add(4);

numberIterator = numbersList.listIterator();
```


1) What does numberIterator.hasNext() return the first time?

Check Show answer

2) What value does numberIterator.hasNext() return *after* three calls to numberIterator.next().

Check Show answer

3) Given the original list initialization above, what is the value of numVal after executing the following statements?

```
numberIterator.next();
numVal =
numberIterator.next();
```

Check Show answer

4) Complete the code to replace the next list element with newDigit.

```
numberIterator.next(); // Necessary!
numberIterator.
                    ;
```


Check Show answer

CHALLENGE ACTIVITY | 8.2.1: List: LinkedList.

437612.2739130.qx3zqy7

Start

Type the program's output


```
import java.util.LinkedList;

public class Letters {
   public static void main(String[] args) {
      LinkedList<Character> letters = new LinkedList<Character>();
      Character letter;
      int i;

      letters.add('A');
      letters.add('B');
      letters.add('C');
      letters.add('D');

      letters.add(2, 'E');
      letters.set(3, 'F');
      letters.remove(1);

      for (i = 0; i < letters.size(); ++i) {
         letter = letters.get(i);
         System.out.print(letter);
      }
      System.out.println();
   }
}
```

AEFD

1 2

Check Next

LinkedList vs. ArrayList

LinkedList and ArrayList are ADTs implementing the List interface. Although both LinkedList and ArrayList implement a List, a programmer should select the implementation that is appropriate for the intended task. A LinkedList typically provides faster element insertion and removal at the list's ends (and middle if using ListIterator), whereas an ArrayList offers faster positional access with indices. In this material, we use the LinkedList class, but the examples can be modified to use ArrayList.

Exploring further:

- LinkedList from Oracle's Java documentation.
- ListIterator from Oracle's Java documentation.
- ArrayList from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

8.3 Map: HashMap

Map container

A programmer may wish to lookup values or elements based on another value, such as looking up an employee's record based on an employee ID. The ***Map*** interface within the Java Collections Framework defines a Collection that associates (or maps) keys to values. The statement `import java.util.HashMap;` enables use of a HashMap.

The HashMap type is an ADT implemented as a generic class (discussed elsewhere) that supports different types of keys and values. Generically, a HashMap can be declared and created as `HashMap<K, V> hashMap = new HashMap<K, V>();` where K represents the HashMap's key type and V represents the HashMap's value type.

The put() method associates a key with the specified value. If the key does not already exist, a new entry within the map is created. If the key already exists, the old value for the key is replaced with the new value. Thus, a map associates at most one value for a key.

The get() method returns the value associated with a key, such as `statePopulation.get("CA")`.

PARTICIPATION ACTIVITY 8.3.1: A HashMap allows a programmer to map keys to values.

Animation content:

Static figure:
Begin Java code:

```
HashMap<String, Integer> statePopulation = new HashMap<String, Integer>();

// 2013 population data from census.gov
statePopulation.put("CA", 38332521);
statePopulation.put("AZ", 6626624);
statePopulation.put("MA", 6692824);
```

System.out.print("Population of Arizona in 2013 is ");
System.out.print(statePopulation.get("AZ"));
System.out.println(".");
End Java code.
An output console and a memory box labeled statePopulation representing the HashMap are displayed.
Step 1: The put() method associates a key with the specified value. The line of code, HashMap<String, Integer> statePopulation = new HashMap<String, Integer>();, is highlighted and creates the HashMap. The line of code, statePopulation.put("CA", 38332521);, is highlighted and "CA", 38332521 is added to the HashMap. The line of code statePopulation.put("AZ", 6626624);, is highlighted and "AZ", 6626624 is added to the HashMap. The line of code, statePopulation.put("MA", 6692824);, is highlighted and "MA", 6692824 is added to the HashMap.
Step 2: The get() method returns the value associated with a key. The line of code, System.out.print("Population of Arizona in 2013 is ");, is highlighted and the text, Population of Arizona in 2013 is, is output to the output console. The line of code, System.out.print(statePopulation.get("AZ"));, is highlighted and the key "AZ" is found in the HashMap which returns the value that corresponds to the key and 6626624 is output to the output console. The line of code, System.out.println(".");, is highlighted and the output console contains the output: Population of Arizona in 2013 is 6626624.

Animation captions:

1. The put() method associates a key with the specified value.
2. The get() method returns the value associated with a key.

PARTICIPATION ACTIVITY 8.3.2: HashMap's put() method replaces the value associated with a key.

Animation content:

Static figure:
Begin Java code:

```
import java.util.HashMap;

public class StatePopulations {
  public static void main (String[] args) {
    HashMap<String, Integer> statePopulation = new HashMap();

    // 2013 population data from census.gov
    statePopulation.put("CA", 38332521);
    statePopulation.put("AZ", 6626624);
    statePopulation.put("MA", 6692824);

    System.out.print("Population of Arizona in 2013 is ");
```

```
    System.out.print(statePopulation.get("AZ"));
    System.out.println(".");

    // 2014 estimated population
    statePopulation.put("AZ", 6871809);

    System.out.print("Population of Arizona in 2014 is ");
    System.out.print(statePopulation.get("AZ"));
    System.out.println(".");
  }
}
```


End Java code.

An output console and a memory box labeled statePopulation representing the HashMap are displayed.

Step 1: The line of code, HashMap<String, Integer> statePopulation = new HashMap();, is highlighted and creates the HashMap. The lines of code, statePopulation.put("CA", 38332521);, statePopulation.put("AZ", 6626624);, statePopulation.put("MA", 6692824);, are highlighted and the HashMap now contains:

"CA", 38332521
"AZ", 6626624
"MA", 6692824

The lines of code, System.out.print("Population of Arizona in 2013 is ");, System.out.print(statePopulation.get("AZ"));, System.out.println(".");, are highlighted and the output console now contains the output:

Population of Arizona in 2013 is 6626624.

The line of code, statePopulation.put("AZ", 6871809);, is highlighted and the key "AZ" is found in the HashMap and the previous value, 6626624, is replaced with the new value, 6871809. The lines of code, System.out.print("Population of Arizona in 2014 is ");, System.out.print(statePopulation.get("AZ"));, System.out.println(".");, are highlighted and the text, Population of Arizona in 2014 is 6871809., is output to the output console.

Animation captions:

1. Using put() with an existing key replaces the value associated with that key.

PARTICIPATION ACTIVITY 8.3.3: Basic HashMap operations: put() and get().

Given the following code that creates and initializes a HashMap:

```
HashMap<String, Integer> playerScores = new HashMap<String, Integer>();
playerScores.put("Jake", 14);
playerScores.put("Pat", 26);
playerScores.put("Hachin", 60);
playerScores.put("Michiko", 21);
playerScores.put("Pat", 31);
```

1) Write a statement to add a mapping for a player named Kira with a score of 1.

Check Show answer

2) Write a statement to assign Hachin's score to a variable named highScore.

Check Show answer

3) What value will playerScores.get("Pat") return.

Check Show answer

4) Write a single statement to update Jake's score to the value 34.

Check Show answer

Determining if a key exists

If the map does not contain the specified key, the get() method returns ***null***. A programmer can use the containsKey() method to check if a map contains the specific key. In the program above, `statePopulation.containsKey("NY")` would return false.

zyDE 8.3.1: Use containsKey() to check if the HashMap contains the use specified key.

This program uses a HashMap to associate a race distance in kilometers with a runner's time. If a race distance does not exist, the program prints "null". Modify the program to us containsKey() method to check if the runner has run a race of the specified distance, and display a message of "No race of the specified distance exists."

RunDistTimeMap.java Load default templat

```
1 import java.util.HashMap;
```

```
2  import java.util.Scanner;
3
4  public class RunDistTimeMap {
5     public static void main (String[] args) {
6        HashMap<Integer, Double> raceTimes = new HashMap<Integer, Doub
7        Scanner scnr = new Scanner(System.in);
8        int userDistKm;
9
10       raceTimes.put(5, 23.14);
11       raceTimes.put(15, 78.5);
12       raceTimes.put(25, 120.75);
13
14       System.out.println("Enter race distance in km (0 to exit): ");
15       userDistKm = scnr.nextInt();
```


5 15 10 0

Run

PARTICIPATION ACTIVITY 8.3.4: Determining if map contains a key.

Given the code below, determine the result of each expression.

```
HashMap<Integer, Double> raceTimes = new HashMap<Integer, Double>();

raceTimes.put(5, 23.14);
raceTimes.put(15, 78.5);
raceTimes.put(25, 120.75);
```

1) raceTimes.containsKey(10)

- ○ true
- ○ false

2) raceTimes.containsKey(5)

- ○ true
- ○ false

3) raceTimes.get(7)

- ○ 0
- ○ null

Common HashMap methods

The HashMap class implements several methods, defined in the Map interface, for accessing and modifying entries within a map.

Table 8.3.1: Common HashMap methods.

put()	`put(key, value)`
	Associates key with specified value. If key already exists, replaces previous value with specified value.
		`// Map originally empty`
		`exMap.put("Tom", 14);`
		`// Map now: Tom->14,`
		`exMap.put("John", 86);`
		`// Map now: Tom->14, John->86`

putIfAbsent()	`putIfAbsent(key, value)`
	Associates key with specified value if the key does not already exist or is mapped to null.
		`// Assume Map is: Tom->14, John->86`
		`exMap.putIfAbsent("Tom", 20);`
		`// Key "Tom" already exists. Map is unchanged.`
		`exMap.putIfAbsent("Mary", 13);`
		`// Map is now: Tom->14, John->86, Mary->13`

get()	`get(key)`
	Returns the value associated with key. If key does not exist, return null.
		`// Assume Map is: Tom->14, John->86, Mary->13`
		`exMap.get("Tom")  // returns 14`
		`exMap.get("Bob")  // returns null`

containsKey()	`containsKey(key)`
	Returns true if key exists, otherwise returns false.
		`// Assume Map is: Tom->14, John->86, Mary->13`
		`exMap.containsKey("Tom")`
		`// returns true`
		`exMap.containsKey("Bob")`
		`// returns false`

containsValue()	`containsValue(value)`
	Returns true if at least one key is associated with the specified value, otherwise returns false.
		`// Assume Map is: Tom->14, John->86, Mary->13`
		`exMap.containsValue(86)  // returns true (key "John" associated with value 86)`
		`exMap.containsValue(17)  // returns false (no key associated with value 17)`

remove()	`remove(key)`
	Removes the map entry for the specified key if the key exists.
		`// Assume Map is: Tom->14, John->86, Mary->13`
		`exMap.remove("John");`
		`// Map is now: Tom->14, Mary->13`

clear()	`clear()`
	Removes all map entries.
		`// Assume Map is: Tom->14, John->86, Mary->13`
		`exMap.clear();`
		`// Map is now empty`

keySet()	`keySet()`
	Returns a Set containing all keys within the map.
		`// Assume Map is: Tom->14, John->86, Mary->13`
		`keys = exMap.keySet();`
		`// keys contains: "Tom", "John", "Mary"`

values()	`values()`
	Returns a Collection containing all values within the map.
		`// Assume Map is: Tom->14, John->86, Mary->13`
		`values = exMap.values();`
		`// values contains: 14, 86, 13`

PARTICIPATION ACTIVITY 8.3.5: Common HashMap methods.

Given the following code that creates and initializes a HashMap:

```
HashMap<String, Integer> exMap = new HashMap<String, Integer>();
exMap.put("Tom", 14);
exMap.put("John", 26);
exMap.put("Mary", 13);
exMap.put("Hans", 90);
exMap.put("Franz", 88);
```

Which of the following operations modify the HashMap?

1) exMap.putIfAbsent("Mary", 17);

- ○ True
- ○ False

2) exMap.putIfAbsent("Frederique", 36);

- ○ True
- ○ False

3) exMap.remove("Tom");

- True
- False

4) exMap.remove("john");

- True
- False

5) exMap.clear();

- True
- False

CHALLENGE ACTIVITY 8.3.1: Map: HashMap.

437612.2739130.qx3zqy7

Start

Type the program's output

```
import java.util.HashMap;

public class AirportCodes {
   public static void main (String[] args) {
      HashMap<String, String> airportCode = new HashMap<String, String>();

      airportCode.put("ALB", "Albany, USA");
      airportCode.put("LNZ", "Linz, Austria");
      airportCode.put("GRX", "Granada, Spain");

      System.out.print("GRX: ");
      System.out.println(airportCode.get("GRX"));

      airportCode.put("ALB", "Ambler, USA");

      System.out.print("ALB: ");
      System.out.println(airportCode.get("ALB"));
   }
}
```

```
GRX: Gra
ALB: Amb
```

1 2 3

Check Next

HashMap vs. TreeMap

HashMap and TreeMap are ADTs implementing the Map interface. Although both HashMap and TreeMap implement a Map, a programmer should select the implementation that is appropriate for the intended task. A HashMap typically provides faster access but does not guarantee any ordering of the keys, whereas a TreeMap maintains the ordering of keys but with slightly slower access. This material uses the HashMap class, but the examples above can be modified to use TreeMap.

Exploring further:

- HashMap from Oracle's Java documentation.
- TreeMap from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

8.4 Set: HashSet

Set container

The ***Set*** interface defined within the Java Collections Framework defines a Collection of unique elements. The Set interface supports methods for adding and removing elements, as well as querying if a set contains an element. For example, a programmer may use a set to store employee names and use that set to determine which customers are eligible for employee discounts.

The HashSet type is an ADT implemented as a generic class that supports different types of elements. A HashSet can be declared and created as `HashSet<T> hashSet = new HashSet<T>();` where T represents the HashSet's type, such as Integer or String. The statement `import java.util.HashSet;` enables use of a HashSet within a program.

PARTICIPATION ACTIVITY | 8.4.1: A HashSet's add(), remove(), and contains() methods add an item, remove an item, and check if an item exists within the set.

Animation content:

Static figure:

Begin Java code:
import java.util.HashSet;

```
public class BookCollection {
  public static void main(String[] args) {
    HashSet<String> ownedBooks = new HashSet<String>();

    ownedBooks.add("A Tale of Two Cities");
    ownedBooks.add("The Lord of the Rings");

    System.out.println("Contains \"A Tale of Two Cities\": " +
                       ownedBooks.contains("A Tale of Two Cities"));

    ownedBooks.remove("The Lord of the Rings");

    System.out.println("Contains \"The Lord of the Rings\": " +
                       ownedBooks.contains("The Lord of the Rings"));
  }
}
```


End Java code.

A memory region of 6 empty memory locations addressed 900 to 905.

An empty output box.

Step 1:
The add() method adds an item such as a String to a set.
The line of code, HashSet<String> ownedBooks = new HashSet<String>();, is highlighted.
Memory locations addressed 900 to 904 are assigned to variable ownedBooks.
The line of code, ownedBooks.add("A Tale of Two Cities");, is highlighted.
Memory address 901 is assigned the string "A Tale of Two Cities".
The line of code, ownedBooks.add("The Lord of the Rings");, is highlighted.
Memory address 903 is assigned the string "The Lord of the Rings".
Step 2:
The contains() method returns true if an item is in the set, and otherwise returns false.
The line of code,System.out.println("Contains \"A Tale of Two Cities\": " + ownedBooks.contains("A Tale of Two Cities"));, is highlighted.
The string, Contains "A Tale of Two Cities":, is outputted.
"A Tale of Two Cities" is found at memory address 903. true is outputted.

Step 3:
The remove() method removes an item from a set.
The line of code, ownedBooks.remove("The Lord of the Rings");, is highlighted.
Memory address 901 that contains "The Lord of the Rings" is now empty.
The line of code,System.out.println("Contains \"The Lord of the Rings\": " + ownedBooks.contains("The Lord of the Rings"));, is highlighted.
The string, Contains "The Lord of the Rings":, is outputted.

"The Lord of the Rings" is not found. false is outputted.

Animation captions:

1. The add() method adds an item such as a String to a set.
2. The contains() method returns true if an item is in the set, and otherwise returns false.
3. The remove() method removes an item from a set.

PARTICIPATION ACTIVITY 8.4.2: HashSet operations: add(), remove(), and contains().

Given the following code that creates and initializes a HashSet:

```
HashSet<Integer> employeeIDs = new HashSet<Integer>();
employeeIDs.add(1001);
employeeIDs.add(1002);
employeeIDs.add(1003);
```

1) Write a statement that adds an employee ID 1337.

Check Show answer

2) Write a statement that removes the employee ID 1002.

Check Show answer

3) What value will employeeIDs.contains(1001) return?

Check Show answer

add() and remove() methods

The add() method does not add duplicate elements to a set. If a programmer tries to add a duplicate element, the add() method fails and returns false. Otherwise, add() returns true.

The remove() method only removes elements that exist within the set. If the element exists, the remove() method removes the element and returns true. Otherwise, remove() returns false.

PARTICIPATION ACTIVITY | 8.4.3: A HashSet's add() and remove() methods return a boolean to indicate the operation's failure or success.

Animation content:

Static figure:
Begin Java code:

```
import java.util.HashSet;

public class HashSetAddRemoveResult {
  public static void main(String[] args) {
    HashSet<String> ownedBooks = new HashSet<String>();
    boolean addResult;
    boolean removeResult;

    ownedBooks.add("A Tale of Two Cities");
    ownedBooks.add("The Lord of the Rings");
    ownedBooks.add("Le Petit Prince");

    addResult = ownedBooks.add("Le Petit Prince");
    System.out.println("Added \"Le Petit Prince\" again: " + addResult);

    removeResult = ownedBooks.remove("The Hobbit");
    System.out.println("Removed \"The Hobbit\": " + removeResult);
  }
}
```

End Java code.

A memory region of 7 empty memory locations addressed 900 to 906.

An empty output box.

Step 1:
A set does not contain duplicate items. The add() method adds an item only if the item does not already exist. add() returns true if item added, else returns false.
Memory addresses 900 to 904 are assigned to ownedBooks.
Memory address 905 is labeled addResult.
Memory address 906 is labeled removeResult.
The line of code, ownedBooks.add("A Tale of Two Cities");, is highlighted.
Memory address 903 is assigned the string "A Tale of Two Cities".
The line of code, ownedBooks.add("The Lord of the Rings");, is highlighted.

Memory address 901 is assigned the string "The Lord of the Rings".
The line of code, ownedBooks.add("Le Petit Prince");, is highlighted.
Memory address 902 is assigned the string "Le Petit Prince".
The line of code, addResult = ownedBooks.add("Le Petit Prince");, is highlighted.
Memory address 905 labeled addResult is assigned false.
The line of code, System.out.println("Added \"Le Petit Prince\" again: " + addResult);, is highlighted.
The string, Added "Le Petit Prince" again: false, is outputted.

Step 2:
The remove() method returns true if item removal was successful, otherwise returns false.
The line of code,removeResult = ownedBooks.remove("The Hobbit");, is highlighted.
Memory address 906 labeled removeResult is assigned false.
The line of code, System.out.println("Removed \"The Hobbit\": " + removeResult);, is highlighted.
The string, Removed "The Hobbit": false, is outputted.

Animation captions:

1. A set does not contain duplicate items. The add() method adds an item only if the item does not already exist. add() returns true if item added, else returns false.
2. The remove() method returns true if item removal was successful, otherwise returns false.

PARTICIPATION ACTIVITY 8.4.4: Return value of HashSet's add() and remove() methods.

Given the following code that creates and initializes a HashSet:

```
HashSet<Character> guessedLetters = new HashSet<Character>();
guessedLetters.add('e');
guessedLetters.add('t');
guessedLetters.add('a');
```

1) What value will guessedLetters.remove('s') return?

Check Show answer

2) What value will guessedLetters.add('e') return?

Check Show answer

3) Write a single statement that adds to guessedLetters the variable favoriteLetter only if favoriteLetter does not already exist in the set.

Check Show answer

zyDE 8.4.1: Use only remove() to check if the user's guess is in the set.

The program below uses a HashSet to store the numbers a user has to guess to win. The program uses both contains() and remove() to remove correct guesses from the set, and game ends when user guesses all numbers (i.e., the set is empty). Modify the program to use remove(), not contains(). Hint: consider the return value of remove().

GuessTheNumbers.java Load default templat

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3
4  public class GuessTheNumbers {
5     public static void main(String[] args) {
6        Scanner scnr = new Scanner(System.in);
7        HashSet<Integer> numbersToGuess = new HashSet<Integer>();
8        int userGuess;
9
10       numbersToGuess.add(3);
11       numbersToGuess.add(5);
12       numbersToGuess.add(1);
13
14       System.out.print("Enter a number between 1 to 10 (0 to exit):
15       userGuess = scnr.nextInt();
```

1 2 5 3

Run

Common HashSet operations

The HashSet class implements several methods, defined in the Set interface, for modifying and querying the status of the set.

Table 8.4.1: Common HashSet methods.

add()	`add(element)`
	If element does not exist, adds element to the set and returns true. If element already exists, returns false.
		`// Set originally empty`
		`exSet.add("Kasparov"); // returns true (element "Kasparov" does not exist)`
		`// Set is now: Kasparov`
		`exSet.add("Kasparov"); // returns false (element "Kasparov" already exists)`
		`// Set is unchanged`

remove()	`remove(element)`
	If element exists, removes element from the set and returns true. If the element does not exist, returns false.
		`// Assume Set is: Kasparov, Fisher`
		`exSet.remove("Fisher"); // returns true (element "Fisher" exists)`
		`// Set is now: Kasparov`
		`exSet.remove("Carlsen"); // returns false (element "Carlsen" does not exist)`
		`// Set is unchanged`

contains()	`contains(element)`
	Returns true if element exists, otherwise returns false.
		`// Assume Set is: Kasparov, Fisher, Carlsen`
		`exSet.contains("Carlsen") // returns true`
		`exSet.contains("Anand") // returns false`

size()	`size()`
	Returns the number of elements in the set.
		`// Set originally empty`
		`exSet.size(); // returns 0`
		`exSet.add("Nakamura");`
		`exSet.add("Carlsen");`
		`// Set is now: Nakamura, Carlsen`
		`exSet.size(); // returns 2`

CHALLENGE ACTIVITY 8.4.1: Using a HashSet to define unique elements.

Start

Type the program's output

```
import java.util.Scanner;
import java.util.HashSet;

public class AstronautsSet {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      HashSet<String> astronauts = new HashSet<String>();
      String userInput;

      userInput = scnr.nextLine();
      while (userInput.compareTo("done") != 0) {
         if (astronauts.add(userInput)) {
            System.out.println("a");
         }
         else {
            System.out.println("n");
         }
         userInput = scnr.nextLine();
      }

      System.out.println("Size: " + astronauts.size());
   }
}
```

Input

```
Sally Ride
John Glenn
Sally Ride
Mae Jemison
Alan Shepard
Yang Liwei
done
```

Output

```
a
a
n
a
a
a
Size: 5
```

1 2

Check Next

HashSet vs. TreeSet

HashSet and TreeSet are ADTs implementing the Set interface. Although both HashSet and TreeSet implement a Set, a programmer should select the implementation that is appropriate for the intended task. A HashSet typically provides faster access but does not guarantee any ordering of the elements, whereas a TreeSet maintains the ordering of elements but with slightly slower access. In this material, we use the HashSet class, but the examples can be modified to TreeSet.

Exploring further:

- HashSet from Oracle's Java documentation.
- TreeSet from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

8.5 Queue interface

Queue interface

The ***Queue*** interface defined within the Java Collections Framework defines a Collection of ordered elements that supports element insertion at the tail and element retrieval from the head.

A LinkedList is one of several types that implements the Queue interface. A LinkedList implementation of a Queue can be declared and created as `Queue<T> queue = new LinkedList<T>();` where T represents the element's type, such as Integer or String. Java supports automatic conversion of an object, e.g., LinkedList, to a reference variable of an interface type, e.g., Queue, as long as the object implements the interface.

The statements `import java.util.LinkedList;` and `import java.util.Queue;` enable use of a LinkedList and Queue within a program.

A Queue's ***add()*** method adds an element to the tail of the queue and increases the queue's size by one. A Queue's ***remove()*** method removes and returns the element at the head of the queue. If the queue is empty, remove() throws an exception.

PARTICIPATION ACTIVITY | 8.5.1: Queue: add() method adds an element to the tail of the queue, and remove() method returns and removes the element at the head of the queue.

Animation captions:

1. The add() method adds an element, such as a String, to the tail of the queue.
2. The remove() method returns and removes the element at the head of the queue.

PARTICIPATION ACTIVITY | 8.5.2: Use Queue's add() and remove() methods to insert and retrieve elements.

Answer the questions given the following code that creates and initializes a Queue.

```
Queue<Integer> ordersQueue = new LinkedList<Integer>();

ordersQueue.add(351);
ordersQueue.add(352);
ordersQueue.add(353);
```

1) Complete the statement to add the value 354 to the tail of the queue.

```
ordersQueue.
                ;
```

Check Show answer

2) Complete the statement to print the element at the head of the queue.

```
System.out.println(ordersQueue.
                );
```

Check Show answer

3) Given the original queue initialization above, what is the size of the queue *after* three calls to ordersQueue.remove()?

Check Show answer

Common Queue methods

The Queue interface has several methods for inserting and removing elements and examining the contents of a queue.

Table 8.5.1: Common Queue methods.

add()	`add(newElement)`
	
	Adds newElement element to the tail of the queue. The queue's size increases by one.
		`// Assume exQueue is: "down" "right" "A"`
		`exQueue.add("B");`
		`// exQueue is now: "down" "right" "A" "B"`

remove()	`remove()`
	
	Removes and returns the element at the head of the queue. Throws an exception if the queue is empty.
		`// Assume exQueue is: "down" "right" "A"`
		`exQueue.remove(); // Returns "down"`
		`// exQueue is now: "right" "A"`

poll()	`poll()`
	
	Removes and returns the element at the head of the queue if the queue is not empty. Otherwise, returns null.
		`// Assume exQueue is: "down"`
		`exQueue.poll(); // Returns "down"`
		`// exQueue is now empty`
		`exQueue.poll(); // Returns null`

element()	`element()`
	
	Returns, but does not remove, the element at the head of the queue. Throws an exception if the queue is empty.
		`// Assume exQueue is: "down" "right" "A"`
		`exQueue.element(); // Returns "down"`
		`// exQueue is still: "down" "right" "A"`

peek()	`peek()`
	
	Returns, but does not remove, the element at the head of the queue if the queue is not empty. Otherwise, returns null.
		`// Assume exQueue is: "down"`
		`exQueue.peek(); // Returns "down"`
		`// exQueue is still: "down"`

The programmer should select the Queue implementation that is appropriate for the intended task. In this material, we use the LinkedList class, but the examples can be modified to use other Queue implementations, including PriorityQueue, LinkedBlockingQueue, and ArrayBlockingQueue, among others.

CHALLENGE ACTIVITY 8.5.1: Enter the output for queue interface.

437612.2739130.qx3zqy7

Start

Type the program's output

```
import java.util.Queue;
import java.util.LinkedList;

public class ItemsList {
   public static void main(String[] args) {
      Queue<String> flowers = new LinkedList<String>();

      flowers.add("poppy");
      flowers.add("lilac");
      flowers.add("iris");

      System.out.println(flowers.remove());
   }
}
```


poppy

1 2

Check Next

Exploring further:

- Queue from Oracle's Java documentation.
- LinkedList from Oracle's Java documentation.
- PriorityQueue from Oracle's Java documentation.
- LinkedBlockingQueue from Oracle's Java documentation.
- ArrayBlockingQueue from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

8.6 Deque interface

Deque interface

The ***Deque*** (pronounced "deck") interface defined within the Java Collections Framework defines a Collection of ordered elements that supports element insertion and removal at both ends (i.e., at the head and tail of the deque).

A LinkedList is one of several types that implements the Deque interface. A LinkedList implementation of a Deque can be declared and created as `Deque<T> deque = new LinkedList<T>();` where T represents the element's type, such as Integer or String. Java supports automatic conversion of an object, e.g., LinkedList, to a reference variable of an interface type, e.g., Deque, as long as the object implements the interface.

The statements `import java.util.LinkedList;` and `import java.util.Deque;` enable use of a LinkedList and Deque within a program.

Deque's addFirst() and removeFirst() methods allow a Deque to be used as a stack. A ***stack*** is an ADT in which elements are only added or removed from the top of a stack. Deque's addFirst() method adds an element at the head of the deque and increases the deque's size by one. The addFirst() method shifts elements in the deque to make room for the new element. The removeFirst() method removes and returns the element at the head of the deque. If the deque is empty, removeFirst() throws an exception.

PARTICIPATION ACTIVITY | 8.6.1: Deque: addFirst() method adds an element at the head, and removeFirst() method returns and removes the element at the head.

Animation captions:

1. The addFirst() method adds an element, such as a String, at the head of the deque.
2. The removeFirst() method returns and removes the element at the head of the deque.

PARTICIPATION ACTIVITY | 8.6.2: Use Deque's addFirst() and removeFirst() methods to insert and retrieve elements.

Given the following code that creates and initializes a Deque.

```
Deque<String> jobsDeque = new LinkedList<String>();

jobsDeque.addFirst("Filter");
jobsDeque.addFirst("Download");
jobsDeque.addFirst("Process");
```

1) What is the value of the element at the head of the deque?

Check | Show answer

2) What is the value of the element at the tail of the deque?

Check Show answer

3) Complete the statement to add the value "Draw" at the head of the deque.

```
jobsDeque.
                ;
```


Check Show answer

4) Complete the statement to remove and print the element at the head of the deque.

```
System.out.println(jobsDeque.
                );
```

Check Show answer

Common Deque methods

The Deque interface has methods for inserting and removing elements at both ends of the deque and examining the contents of the deque.

Table 8.6.1: Common Deque methods.

addFirst()	`addFirst(newElement)`
	
	Adds newElement element at the head of the deque. The deque's size increases by one.
		`// Assume exDeque is: 3 5 6`
		`exDeque.addFirst(1);`
		`// exDeque is now: 1 3 5 6`

addLast()	`addLast(newElement)`
	
	Adds newElement element at the tail of the deque. The deque's size increases by one.
		`// Assume exDeque is: 3 5 6`
		`exDeque.addLast(7);`
		`// exDeque is now: 3 5 6 7`

removeFirst()	`removeFirst()`
	
	Removes and returns the element at the head of the deque. Throws an exception if the deque is empty.
		`// Assume exDeque is: 3 5 6`
		`exDeque.removeFirst(); // Returns 3`
		`// exDeque is now: 5 6`

removeLast()	`removeLast()`
	
	Removes and returns the element at the tail of the deque. Throws an exception if the deque is empty.
		`// Assume exDeque is: 3 5 6`
		`exDeque.removeLast(); // Returns 6`
		`// exDeque is now: 3 5`

pollFirst()	`pollFirst()`
	
	Removes and returns the element at the head of the deque if the deque is not empty. Otherwise, returns null.
		`// Assume exDeque is: 3 5 6`
		`exDeque.pollFirst(); // Returns 3`
		`// exDeque is now: 5 6`
		
		`exDeque.pollFirst(); // Returns 5`
		`exDeque.pollFirst(); // Returns 6`
		`// exDeque is now empty`
		
		`exDeque.pollFirst(); // Returns null`

pollLast()	`pollLast()`
	
	Removes and returns the element at the tail of the deque if the deque is not empty. Otherwise, returns null.
		`// Assume exDeque is: 3 5 6`
		`exDeque.pollLast(); // Returns 6`
		`// exDeque is now: 3 5`
		
		`exDeque.pollLast(); // Returns 5`
		`exDeque.pollLast(); // Returns 3`
		`// exDeque is now empty`
		
		`exDeque.pollLast(); // Returns null`

getFirst()	`getFirst()`
	
	Returns, but does not remove, the element at the head of the deque. Throws an exception if the deque is empty.
		`// Assume exDeque is: 3 5 6`
		`exDeque.getFirst(); // Returns 3`
		`// exDeque is still: 3 5 6`

getLast()	`getLast()`
	
	Returns, but does not remove, the element at the tail of the
		`// Assume exDeque is: 3 5 6`
		`exDeque.getLast(); // Returns 6`
		`// exDeque is still: 3 5 6`

	deque. Throws an exception if the deque is empty.	

peekFirst()	`peekFirst()`
	
	Returns, but does not remove, the element at the head of the deque if the deque is not empty. Otherwise, returns null.
		`// Assume exDeque is: 3 5 6`
		`exDeque.peekFirst(); // Returns 3`
		`// exDeque is still: 3 5 6`

peekLast()	`peekLast()`
	
	Returns, but does not remove, the element at the tail of the deque if the deque is not empty. Otherwise, returns null.
		`// Assume exDeque is: 3 5 6`
		`exDeque.peekLast(); // Returns 6`
		`// exDeque is still: 3 5 6`

The programmer should select the Deque implementation that is appropriate for the intended task. In this material, we use the LinkedList class, but the examples can be modified to use other Deque implementations, including ArrayDeque, LinkedBlockingDeque, and ConcurrentLinkedDeque.

CHALLENGE ACTIVITY 8.6.1: Deque interface.

437612.2739130.qx3zqy7

Start

Type the program's output

```
import java.util.LinkedList;
import java.util.Deque;

public class ToInvite {
   public static void main (String[] args) {
      Deque<String> invites = new LinkedList<String>();

      invites.addFirst("Keith");
      invites.removeFirst();
      invites.addFirst("Bill");
      invites.addFirst("Sue");

      System.out.println("1. " + invites.removeFirst());
   }
}
```

```
1. Sue
```


1 2 3

Check Next

Exploring further:

- Deque from Oracle's Java documentation.
- LinkedList from Oracle's Java documentation.
- ArrayDeque from Oracle's Java documentation.
- LinkedBlockingDeque from Oracle's Java documentation.
- ConcurrentLinkedDeque from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

8.7 LAB: Grocery shopping list (LinkedList)

This section's content is not available for print.

8.8 LAB: Student grades (HashMap)

This section's content is not available for print.

8.9 LAB: Ticketing service (Queue)

This section's content is not available for print.

8.10 LAB: Palindrome (Deque)

This section's content is not available for print.

8.11 LAB: Unique random integers (HashSet)

This section's content is not available for print.