Java Programming

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Generics
- Collections

Collections

- Collection – A data structure that can hold references to other objects.
- The collection itself is an object.
- For example…

Collections

- In general, a collection is just a bunch of "things" that are stored together.

Java - ArrayList

- ArrayList stores its objects just like a normal array.
- Put values in and get values out of the collection using an index.
- **ArrayList resizes itself.**

Java - ArrayList

Generic Collections

- A generic collection can store any type of object inside of it.
- Old versions of Java collections only stored objects of type "Object".
- When retrieving objects from the old Java collections you had to "cast" them.
- With generics there is no need to cast.
- Generic collections store any reference type.

Generic Collections

Collections and Primitive Types

- Collections can only store objects.
- Is there a way to store a primitive type inside of a collection?

Collections and Primitive Types

Type-Wrapper Classes

- Is there a way to store a primitive type inside of a collection?
  YES
- You must use a **type-wrapper class** for the primitive-type variable. Do the following:

1. Create an instance of the appropriate type-wrapper class.
2. Store the primitive value in the type-wrapper class instance.
3. Store the type-wrapper class object in the collection.

Type-Wrapper Example

```
Integer[] ia;
Integer io;

// Create an array of type-wrapper
Integer[] ia = new Integer[ 5 ];

// Create type-wrapper and store primitve data
io = new Integer(10);

// Put the type-wrapper object in the array
ia[0] = io;

// Directly put an int in array
ia[0] = 10;
```

Use type-wrapper to store primitive data in a collection

"Box" the int 10 in an Integer instance.

Automatically "boxes" the int 10 in an Integer instance

Type-Wrapper Example

```
Integer[] ia;
Integer io;

// Create an array
Integer[] ia = new Integer[ 5 ];

// Create type-wrapper
io = new Integer(10);

// Put the type-wrapper object in the array
ia[0] = io;

// Retrieve primitive data from collection
int value = ia[0].intValue();
```

Retrieve primitive data from a collection

Type-Wrapper Classes

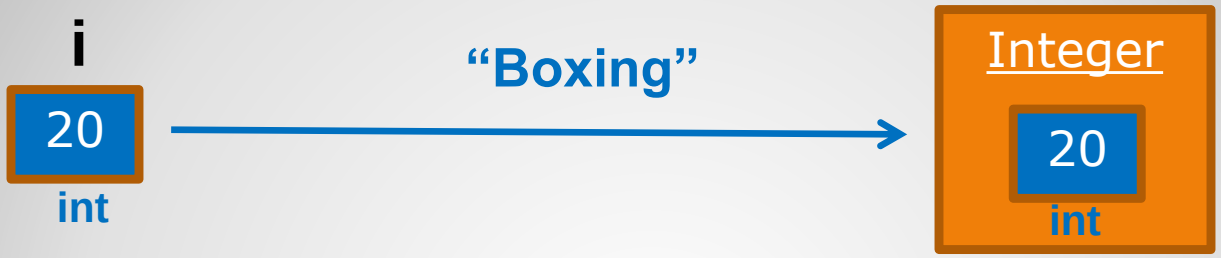
- Boolean
- Byte
- Character
- Double
- Float
- Integer
- Long
- Short

Note: The type-wrapper classes are declared as final.

- **Boxing** – Converts a value of a primitive type to an object of the corresponding type-wrapper class.
- **Unboxing** –Converts an object of a type-wrapper class to a value of the corresponding primitive type.

Boxing and Unboxing

Boxing – Convert from a primitive type to a reference type.



Unboxing – Convert back from reference type to a primitive type.

Boxing and Unboxing

Boxing and Unboxing

```
Integer[] ia;
Integer io;

// Create an array
Integer[] ia = new Integer[ 5 ];

// Create type-wrapper
io = new Integer(10);

// Put the type-wrapper object in the array
ia[0] = io;

// Retrieve primitive data from collection
int value = ia[0].intValue();
```

Boxing

Unboxing

Autoboxing and Autounboxing

- Later versions of Java can perform boxing and unboxing automatically.
- This means you do not have to explicitly create an instance of the type-wrapper class.

For example…

```
Integer[] ia;
Integer io;

// Create an array
Integer[] ia = new Integer[ 5 ];

// Create type-wrapper
io = 10;

// Put the type-wrapper object in the array
ia[0] = io;

// Retrieve primitive data from collection
int value = ia[0];
```

Autoboxing
Converts automatically

Autounboxing
Converts automatically

Autoboxing and Autounboxing

- Now on to collections…

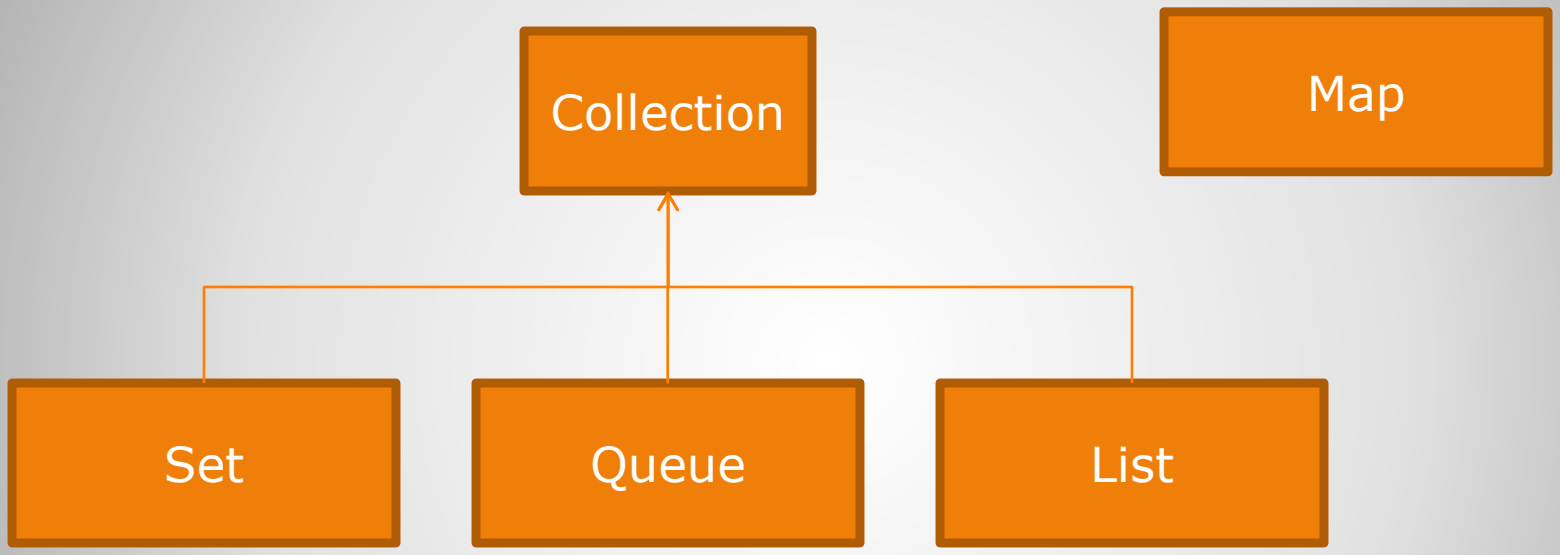
Collections

Collection Interfaces

- The Java framework defines interfaces for a number of different collection types.
- For example…

Partial Collection Interface Hierarchy

- Partial collection interface hierarchy:



Note: Map does not implement the Collection interface

Collection Interfaces

- **Collection** – Base interface. Contains common behaviors of some other interfaces.

- **List** – An ordered collection that can contain duplicate elements.
- **Set** – A collection that does not contain duplicates.
- **Map** – A collection that associates keys to values and cannot contain duplicate keys.
- **Queue** – First-in, First-out (FIFIO) collection that models a waiting line.

- Stored in the **java.util** package.

Collection Interfaces

interface Collection<E>

- Base interface for some other collection interfaces (not all).
- E stands for the data type of the items in the collection.

- Contains methods for the following categories of operations:
  - Adding
  - Clearing
  - Contains
  - Removing
  - Iterator
- Classes that implement this interface should have two constructors:
  - Default constructor
  - Constructor that takes a Collection as a parameter.

interface Collection<E>

interface List<E>

- Inherits from Collection interface.

- An ordered collection.
- Can contain duplicates.
- Can manipulate elements in the list according to their indices.
- Implemented by the following classes:
  ArrayList, LinkedList, Vector

Note: ArrayList and Vector are basically both resizable arrays. They differ with respect to thread synchronization and a few other things.

interface List<E>

Create a List

- Create an empty list using **ArrayList** (ArrayList implements the List interface):

```
List<String> myList = new ArrayList<>();
```

Instance type is inferred from the variable data type

or

```
List<String> myList = new ArrayList<String>();
```

- Create a list from data using Arrays.asList (you can pass as many parameters as you want):

```
List<String> myList = Arrays.asList("a", "b", "c");
```

- Create a list from an existing array:

```
String[] myArray = {"a", "b", "c"};
List<String> myList = Arrays.asList(myArray);
```

Note: Arrays is a prewritten class in the JDK that contains static helper methods for dealing with arrays.

Create a List

- List example. ArrayList implements the List interface.

Declare interface reference

```
List<String> langList;
langList = new ArrayList<String>();
```

Create instance of ArrayList

```
langList.add("Java");
langList.add("C++");
langList.add("Python");
```

Use List interface reference to add items to the collection

```
for (String s : langList) {
   System.out.println(s);
}
```

List Example

Cannot Use Primitive Types in a Collection

- Can only use reference types as the data type in a collection.

```
List<int> myList = new ArrayList<>();
```

CANNOT use a primitive type as the data type. You will see a compile error similar to the following: "Unexpected type, required reference, found int".

```
// Use the wrapper type instead of a primitive type
List<Integer> myList = new ArrayList<>();
myList.add(10);
```

Will auto box the int data

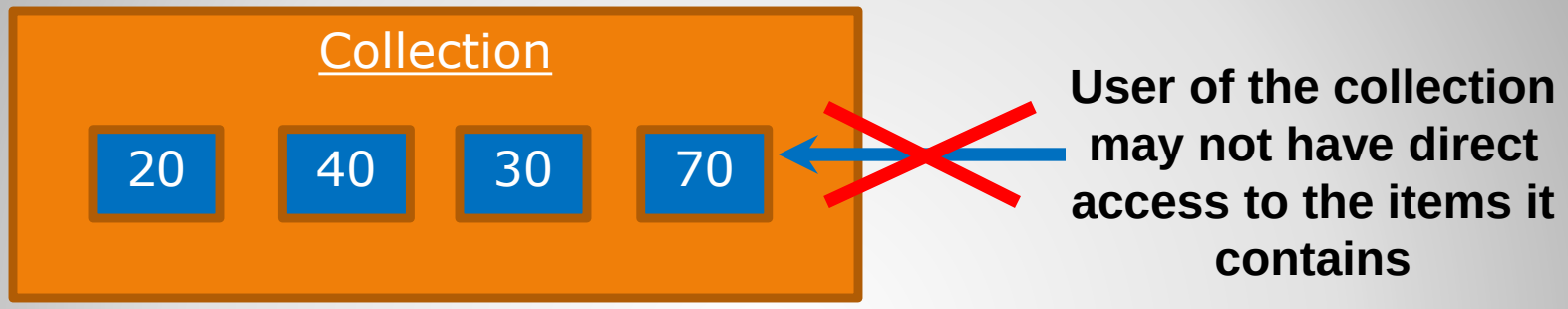
- You can write your own List collection.
- If you write a new class that contains a collection of some sort you could have your class implement the List interface.

List Interface and Custom Collections

General Description of Iterators

- **Iterators** – In general, objects which allow a program to traverse through a collection.
- Internals of a collection may be hidden (private) so there needs to be a way to access them all.
- Iterators are used to "visit" each element of a collection.

- Here is a collection with data (could be an array):

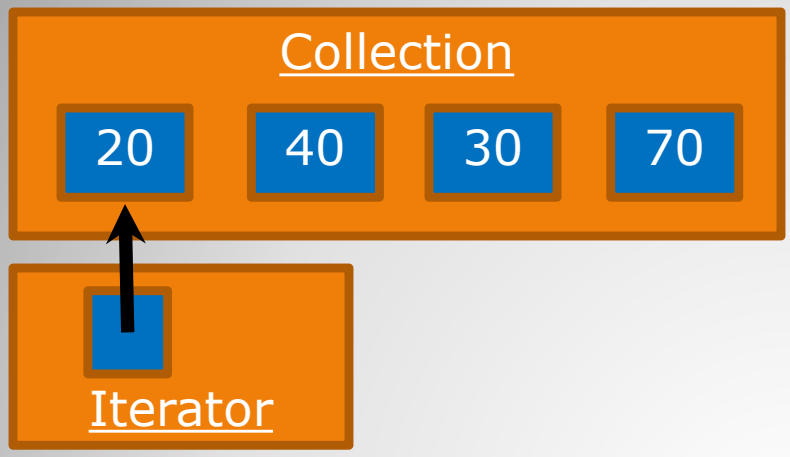


- If we want to print all items in this collection, we would not be able to in this case.

General Description of Iterators

General Description of Iterators

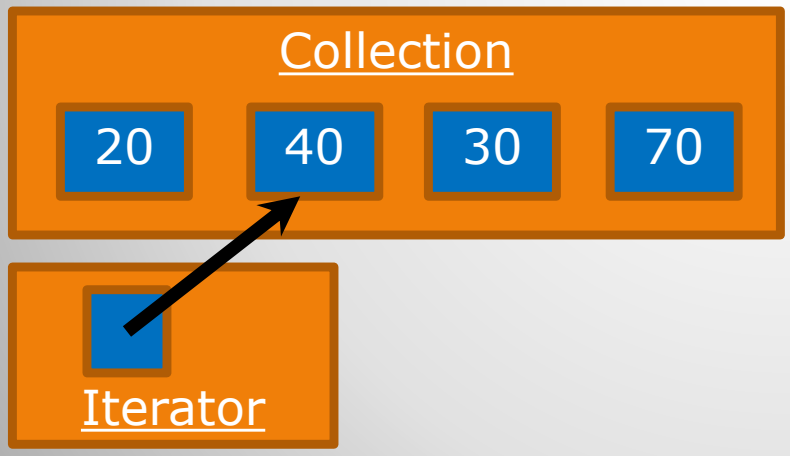
- Iterators are helper classes that have access to the items of the collection.
- An iterator "points at" one item of the class.
- In general, you can do the following with an iterator:
  - Get data from the current item.
  - Go to the next item in the collection.
  - Some iterators allow you to traverse the collection in reverse.
  - Some iterators allow you to remove items from the collection.
- For example…



This iterator points at the first item of the collection.

You can get the data (20) at that item if you want but not any other item's data.

If we told the iterator to go to the next item then it would look like the following….



Iterator now points at the second item.

You can get the data in the second item (40) but not the other items.

General Description of Iterators

interface Iterator<E>

- The Iterator interface allows you to traverse a collection from beginning to the end (not in reverse).

- Some methods:
  - **next** – Returns the data at the current item and moves the iterator to the next item.
  - **hasNext** – Returns true if there is another item in the collection and false otherwise.

interface Iterator<E>

interface Iterator<E>

- The Iterator interface allows you to traverse a collection from beginning to the end (not in reverse).

```
// Code to create langList here…
Iterator<String> iter;
iter = langList.iterator();


while (iter.hasNext()) {
   String current = iter.next();
   System.out.println(current);
}
```

Declare iterator reference

Get an iterator from a collection (assumes langList was created and populated with data)

Keep going while there is another item

Returns the current item and moves the iterator to the next item

interface Iterator<E>

interface ListIterator<E>

- Iterator used specifically for a list.
- Derived from the Iterator<E> interface.
- Allows for:
  - Traversing the list in reverse.
  - Adding new items into the list at the iterator's current location.
  - Some other functionality as well.

interface ListIterator<E>

interface ListIterator<E>

- The ListIterator interface also allows you to traverse a collection in reverse.

```
// Code to create langList here...
ListIterator<String> listIter2;
listIter2 = langList.listIterator(langList.size());

while (listIter2.hasPrevious())
{
   String current = listIter2.previous();
   System.out.println(current);
}
```

Get iterator to the last element by giving the index (first call to previous will return the item at size-1)

Call listIterator (not iterator)

Keep going while there is a previous item

Previous returns the previous item in the list and moves the iterator backwards in the list

interface ListIterator<E>

Converting a List to an Array

- Use List's **toArray** method.

Pass in 0 length String array

- For example:

String[] myArray = **myList.toArray(new String[0]);**

- A new string array is passed in as a parameter.
- It will try and put the list data into this array and it will fail.
- Since the data will not fit (array size is 0) it will create another array of the appropriate size and return it with all the data inside.

Converting a List to an Array

Collections Class

- The Collections class is **different than the Collection interface discussed earlier**.
- Contains methods that can be used to manipulate and query a given collection.
- Only contains **static methods**.
- Here are a few of the methods on the Collections class…

Collections Class

Collections Class Methods

- **sort** – Sorts the elements of a list.
- **binarySearch** – Locates an object in a list.
- **reverse** – Reverses the elements of a list.
- **shuffle** – Randomly orders a List's elements.
- **min** – Returns the smallest element in a Collection.
- **max** – Returns the largest element in a Collection.

- For an exhaustive list of methods go to:

https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Collections.html

Collections Class Methods

Collections Class Method (example for sort)

- The sort method puts the items in sorted order.

```
System.out.println("Original order");
for (String s : langList) {
   System.out.println(s);
}

// Sort the list
Collections.sort(langList);

System.out.println("Sorted order");
for (String s : langList) {
   System.out.println(s);
}
```

Call the sort method on collections (it is a static method so we call directly on the class)

Note: To sort a collection of a user-defined class that class must implement the Comparable interface.

sort

Other Collections

- Now on to other collections.
- We will start with queues…

Queue

- An abstract data type in which elements are added to the rear and removed from the front; a "first in, first out" (FIFO) structure.
- With a queue you can only add items to the end of the collection and remove items from the front.
- You only have access to the first item in the queue.

Queue

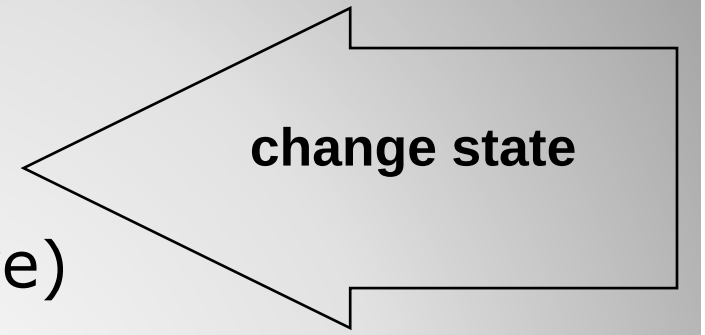
Queue

- What operations would be appropriate for a queue?

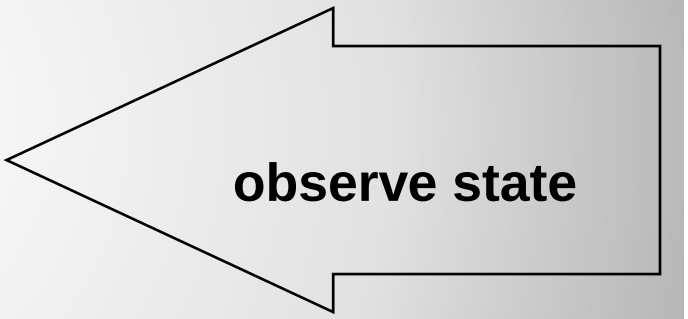
Queue – Logical View

Queue – Logical View

- **Transformers**
  - MakeEmpty
  - Enqueue (Insert or Add)
  - Dequeue (Delete or Remove)



- **Observers**
  - IsEmpty
  - IsFull

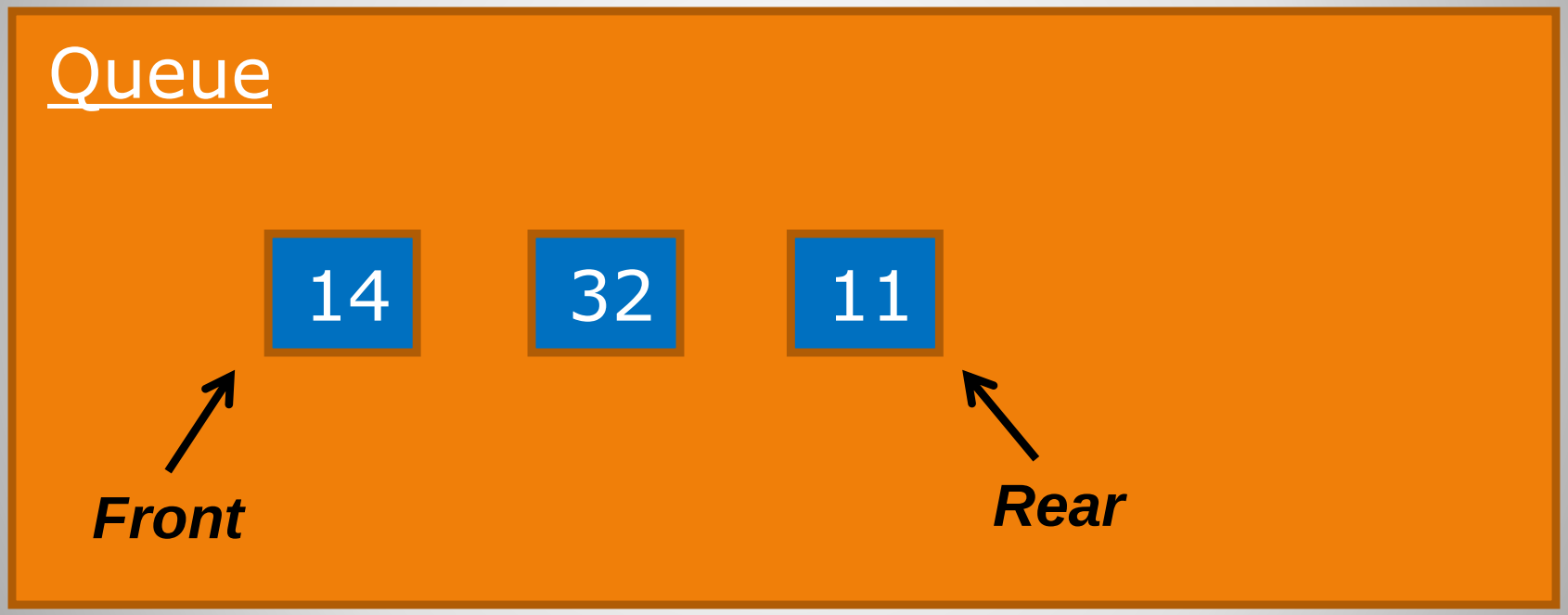


- What does a **queue** look like if we insert the following elements (in the given order):

  14, 32, 11

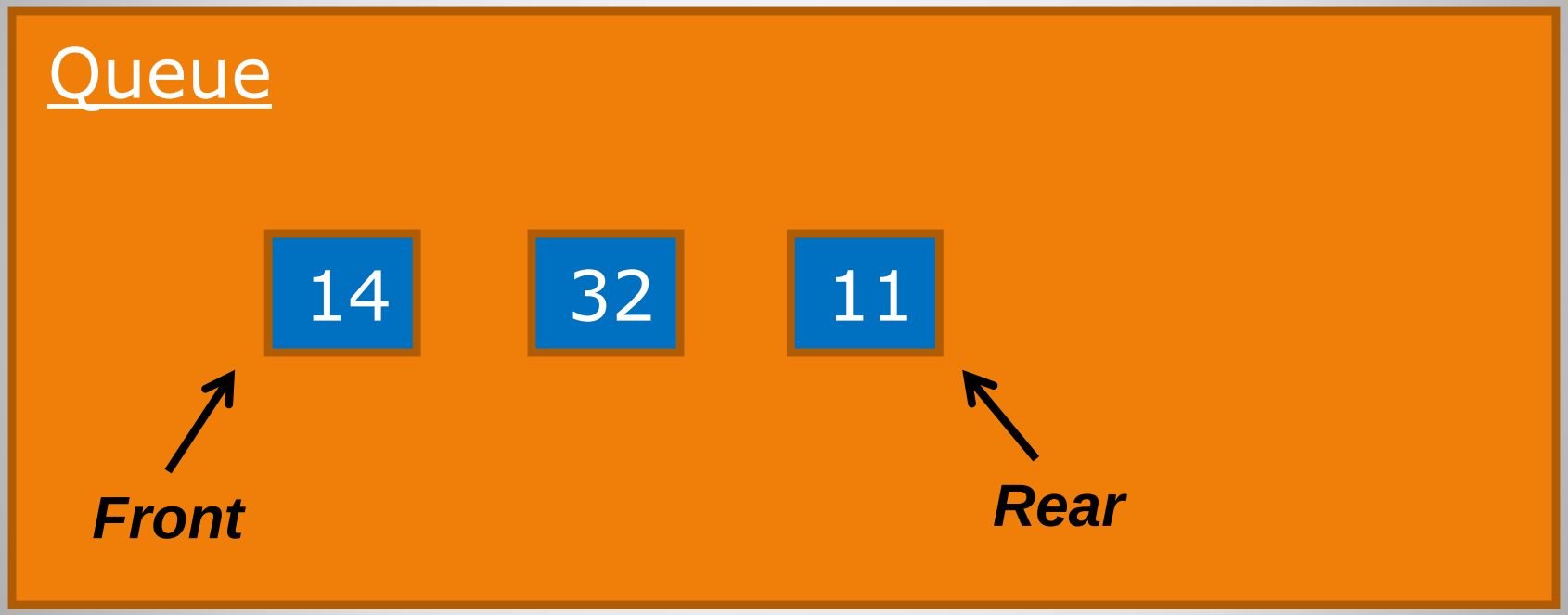
Queue – Logical View

- Insert: 14, 32, 11
- Here is the resulting queue…



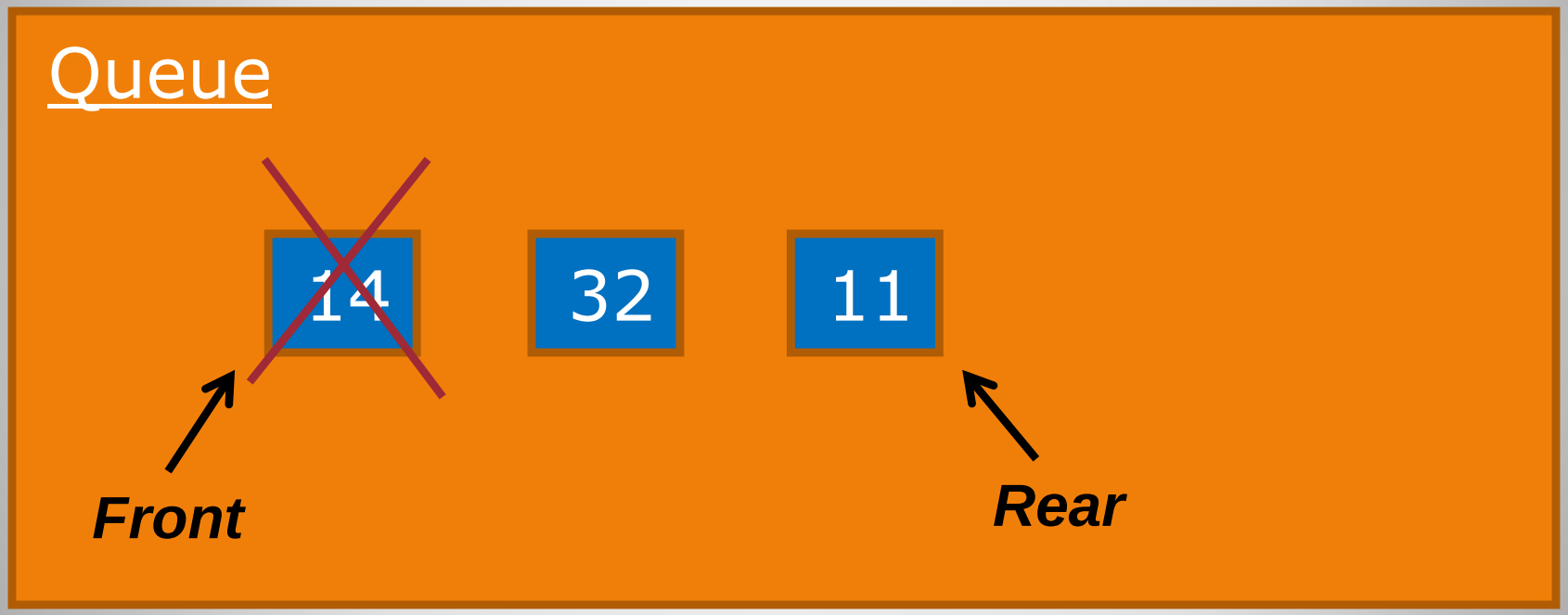
Queue – Logical View

- What if we remove an element?
- Where does it get removed from?
- Can we remove from in the middle?



Queue – Logical View

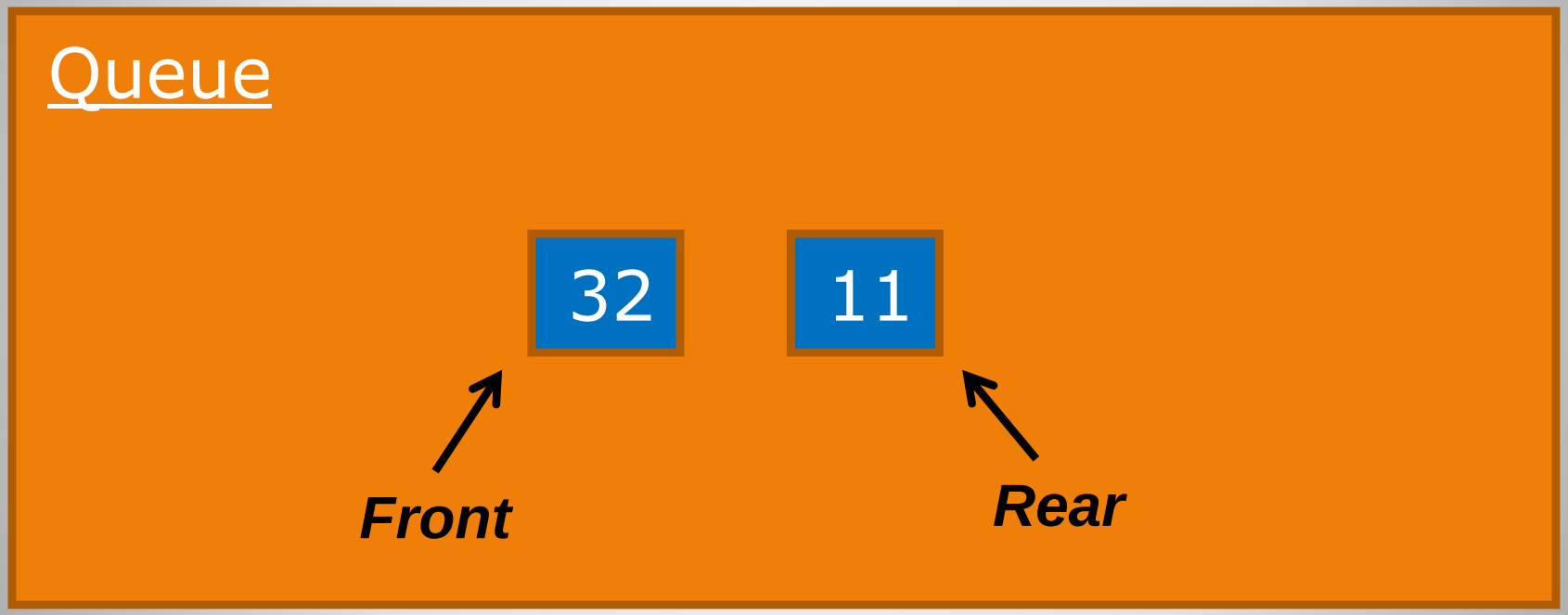
- What if we remove an element?
- Where does it get removed from? THE FRONT
- Can we remove from in the middle? NO



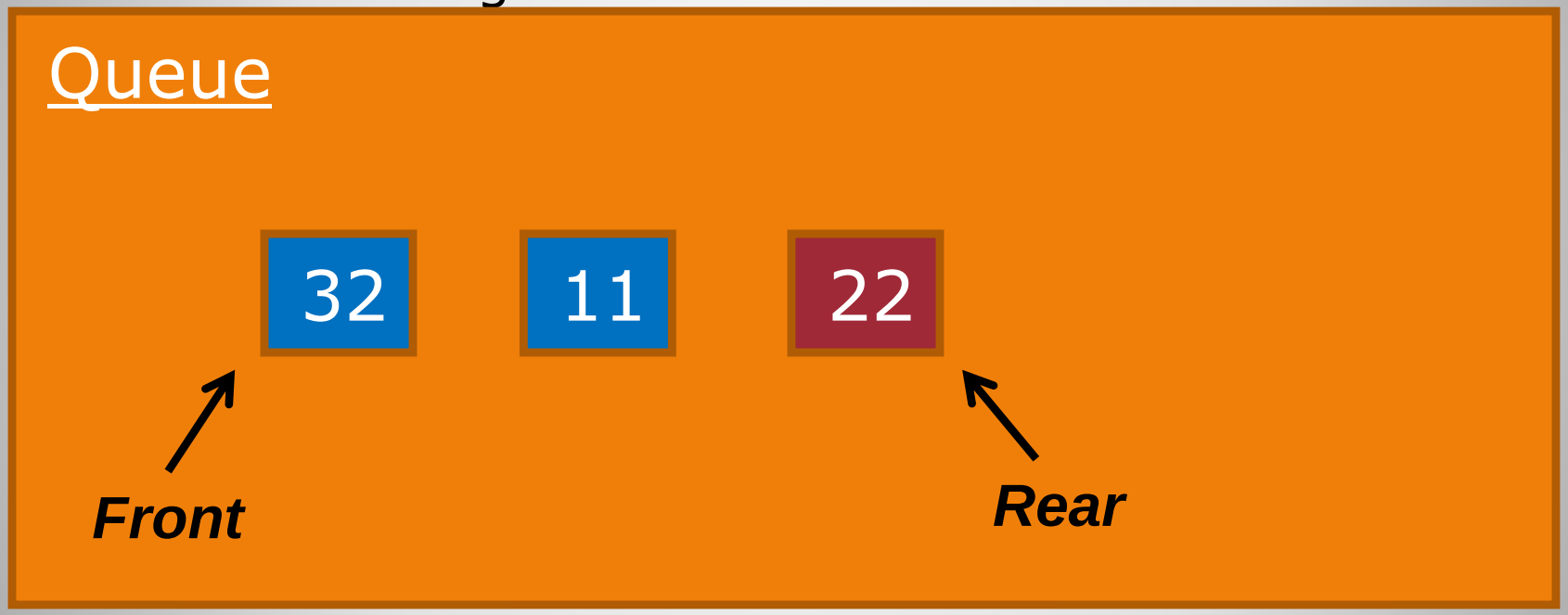
Queue – Logical View

Queue – Logical View

- Queue after removing one element.
- Can we add an element after we remove. For example, Enqueue(22)?
- Where does it get added?



- Queue after removing one element.
- Can we add an element after we remove. For example, Enqueue(22)?
- Where does it get added? REAR



Queue – Logical View

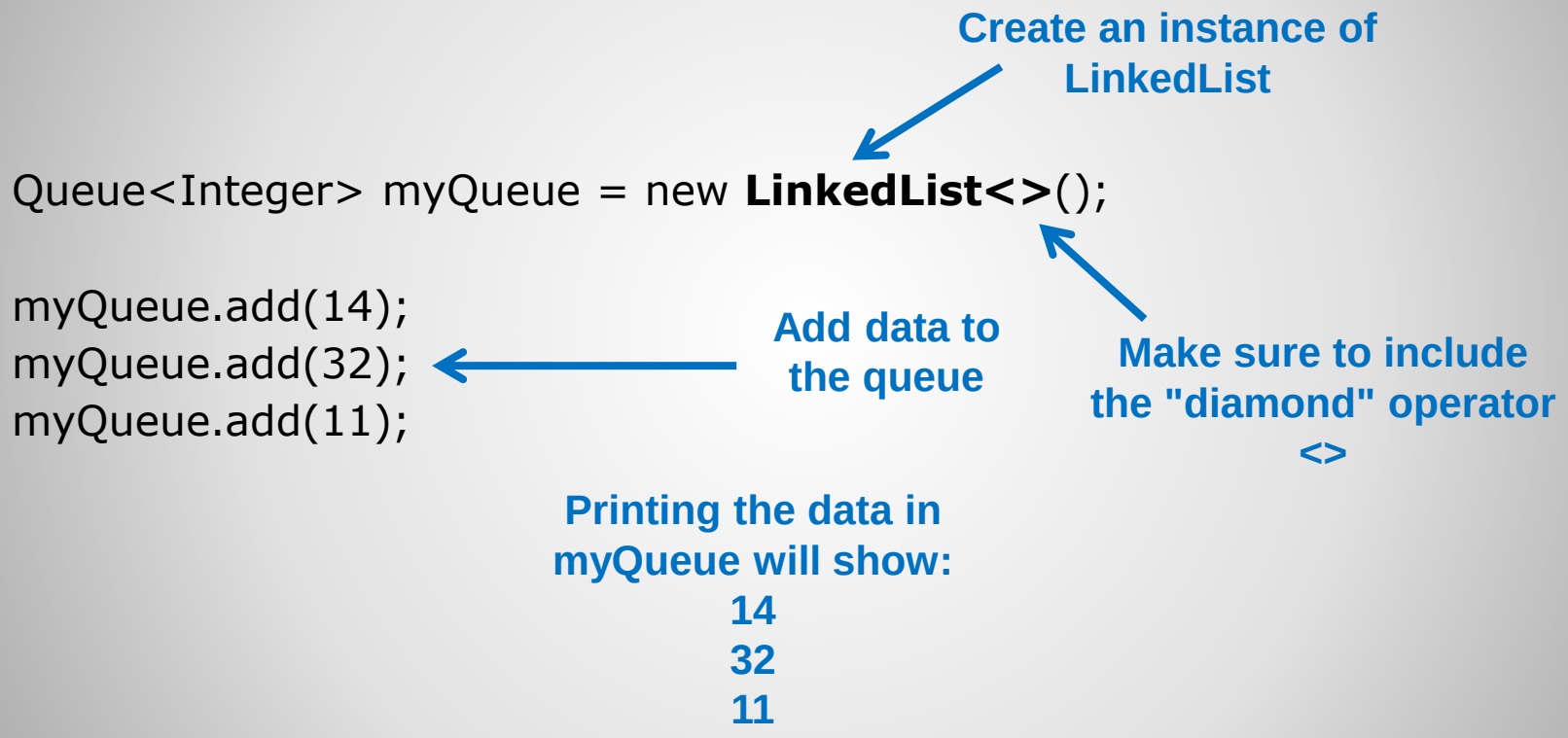
interface Queue<E>

- Inherits from Collection interface.
- add - Add to the end of the queue (enqueue).
- element – Gets first element of queue but does not remove it. Should throw an exception if queue is empty (dequeue).
- peek – Same as element method except it returns null for an empty queue instead of throwing an exception.
- remove – Gets and removes the first element. Should throw an exception if queue is empty (dequeue).
- poll - Same as remove method except it returns null for an empty instead of throwing an exception.

interface Queue<E>

Create a Queue

- Use LinkedList to create a normal queue with first in first out behavior (FIFO).



Create a Queue

Priority Queue

- Similar to a queue except items are removed from the queue in priority order.
- The highest priority item is always first in the queue.

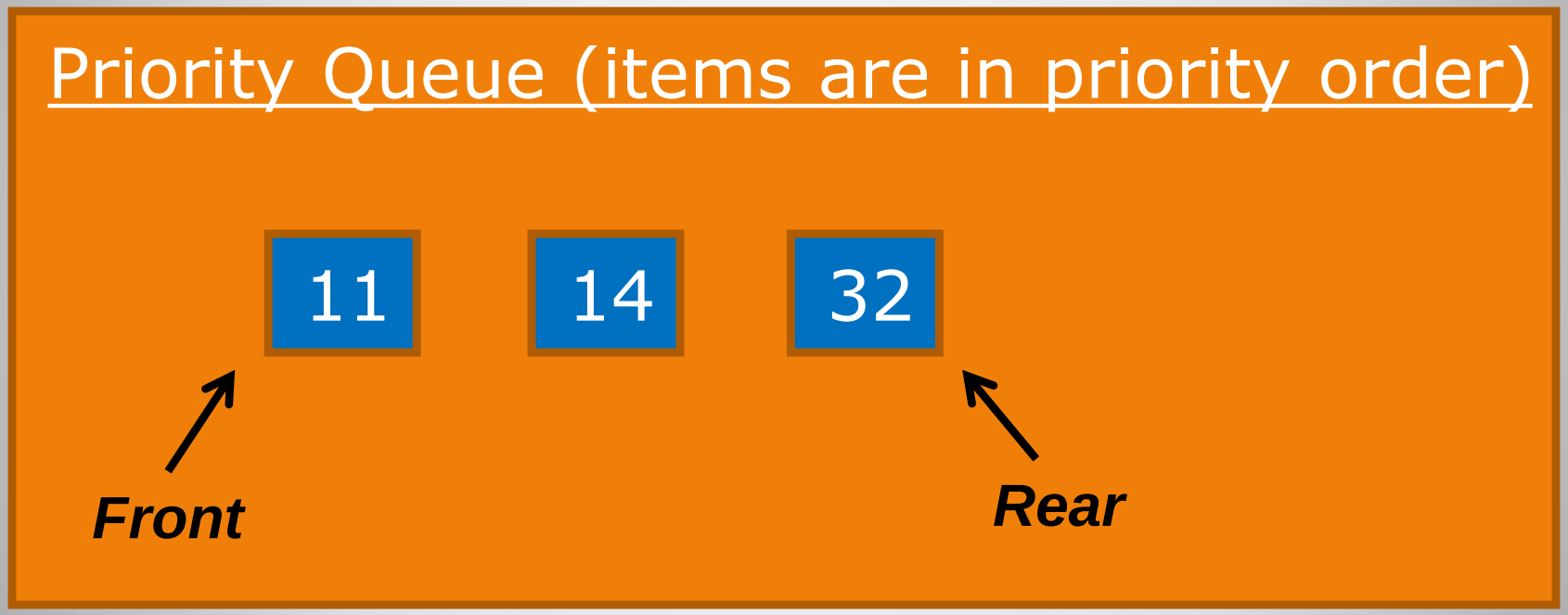
- For a priority queue of class type items, we must state how to compare items with each other.
- For example, we may have a priority queue of Employee objects.
- We could define that an employee with a higher salary has a higher priority.
- Create a class that implements the Comparator interface. This class will define how items are compared with each other.

Priority Queue – Logical View

- What does a **priority queue** look like if we insert the following elements (in the given order):
  14, 32, 11

Priority Queue – Logical View

- Insert: 14, 32, 11
- Here is the resulting queue from a logical perspective…

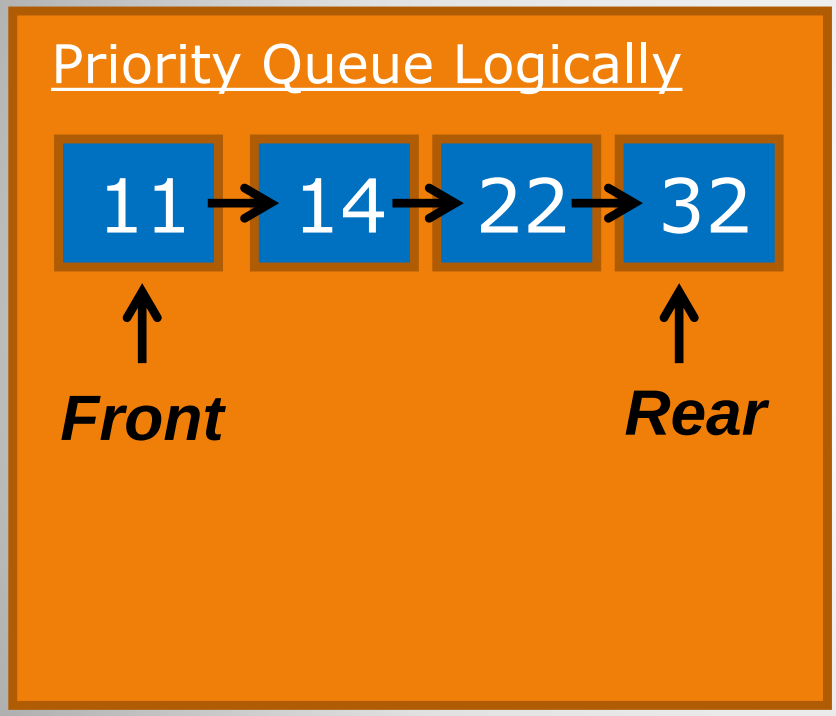


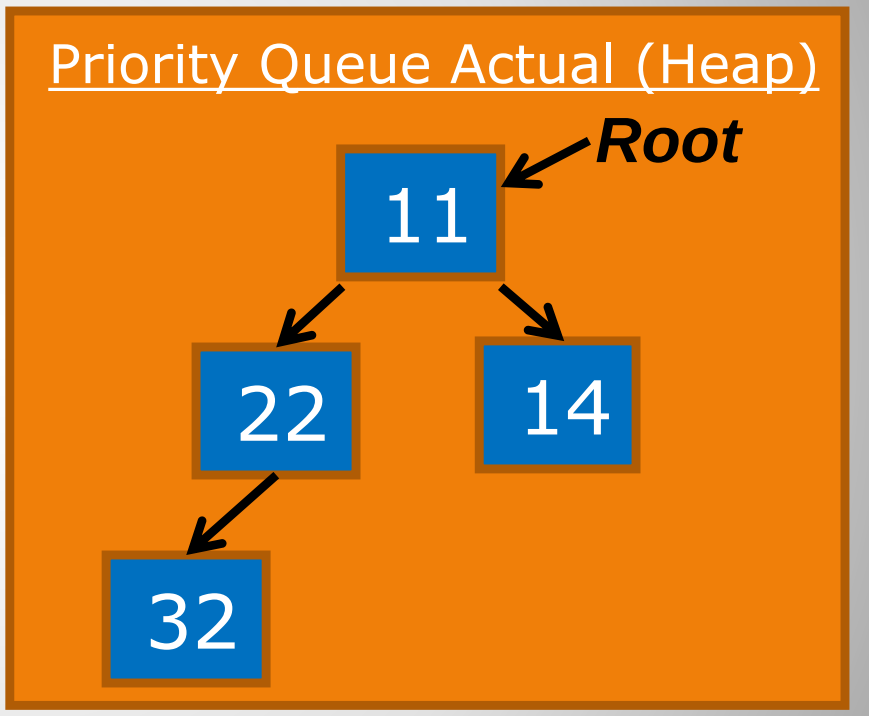
Priority Queue – Logical View

- Now Enqueue(22)?
- 22 will get added in the place according to its priority.

Priority Queue – Logical View

- A priority queue feels like it has its items one after another in order but that is not actually the case.
- Java's priority queue stores data in a heap (ordered by level in heap).





Priority Queue – Logical vs Actual Storage View

class PriorityQueue<E>

- Implements the Queue<E> interface.

- A collection class that allows retrieving data in priority order.
- It stores data as a heap internally (highest priority item is always in the root).

Note: If you use an iterator to traverse a PriorityQueue you are not guaranteed to see the data in priority order. Heaps are only partially ordered. To see all the data in order you will have to continually remove items and print them as you remove them.

The following is taken directly from the JDK documention:

"The Iterator provided in method iterator() and the Spliterator provided in method spliterator() are not guaranteed to traverse the elements of the priority queue in any particular order."

Link: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/PriorityQueue.html

Class PriorityQueue<E>

Create a Priority Queue

- Create an empty queue using **PriorityQueue**:

Create an instance of PriorityQueue

```
Queue<Integer> myQueue = new PriorityQueue<>();

myQueue.add(14);
myQueue.add(32);
myQueue.add(11);
```

Add data to the queue

Make sure to include the "diamond" operator <>

Removing and printing the data in myQueue will show data in priority order:

11
14
32

Create a Priority Queue

PriorityQueue of Class Type – Create Comparator Class

- Need to let the priority queue know how to compare items.
- Define a class that implements the Comparator interface.

```
class Employee {
    private String name;
    private int salary;
    // Get/Set methods here
}
```

Implement the Comparator interface

```
public class EmployeeComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        if (e1.getSalary() < e2.getSalary())
            return 1;
        else if (e1.getSalary() > e2.getSalary())
            return -1;
        return 0;
    }
}
```

Compares employees based on salary

Compare returns 0, 1, or -1
Returns 0 if equal
Returns 1 if e1 salary is less
Return -1 if e1 salary is greater

Priority Queue of Class Type

Create a Priority Queue with a Comparator

- Create an empty queue that uses a comparator.
- An instance of comparator should be passed into the PriorityQueue constructor.
- The PriorityQueue will now use the comparator to arrange its items (highest priority item is always first).

Create an instance of the comparator

```
EmployeeComparator empComp = new EmployeeComparator();
Queue<Employee> myQueue = new PriorityQueue<>(empComp);
```

This queue contains Employee objects

Pass in the comparator

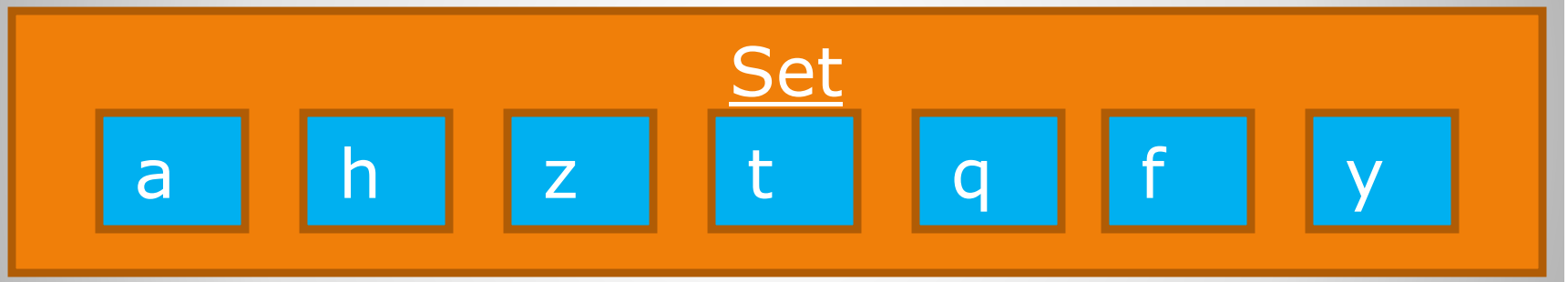
Create a Priority Queue with a Comparator

- Now on to sets…

Other Collections

Set

- Sets are generally unordered and CANNOT contain duplicates.



interface Set<E>

- Inherits from Collection interface.

- Unordered collection.
- Classes that implement this interface should not allow duplicates in their collection.

interface Set<E>

Java Set Implementation Classes

- **HashSet** – Uses a hash table internally.
- **LinkedHashSet** – Uses a hash table internally but also maintains a doubly linked list to keep the insertion order.
- **TreeSet** – Uses a tree internally. Stores data in sorted order.

- Differences between them are the "speed" of specific operations.
- For example, when using an average case data set HashSet has a O(1) search time while TreeSet has a O(log n) search time.

Set

Create a Set (HashSet)

- Create an empty set using **HashSet**:

Create an instance of HashSet

```
Set<Integer> mySet = new HashSet<>();

mySet.add(14);
mySet.add(32);
mySet.add(32);
mySet.add(11);
mySet.add(11);
mySet.add(11);

for (Integer i : mySet) {
   System.out.println(i);
}
```

Add data to the set

Make sure to include the "diamond" operator <>

Printing mySet shows no duplicates and insertion order is NOT preserved:

32
11
14

Create a Queue (HashSet)

Create a Set (LinkedHashSet)

- Create an empty set using **LinkedHashSet**:


```
Set<Integer> mySet = new LinkedHashSet<>();

mySet.add(14);
mySet.add(32);
mySet.add(32);
mySet.add(11);
mySet.add(11);
mySet.add(11);

for (Integer i : mySet) {
   System.out.println(i);
}
```

Printing mySet shows no duplicates and insertion order IS preserved:

14
32
11

Create a Set (LinkedHashSet)

Create a Set (TreeSet)

- Create an empty set using **TreeSet**:

Create an instance of TreeSet

```
Set<Integer> mySet = new TreeSet<>();

mySet.add(14);
mySet.add(32);
mySet.add(32);
mySet.add(11);
mySet.add(11);
mySet.add(11);

for (Integer i : mySet) {
   System.out.println(i);
}
```

Printing mySet shows no duplicates and data in SORTED order:

11
14
32

Create a Set (TreeSet)

Set

Hashset, Treeset, LinkedHashSet all behave like a traditional set.

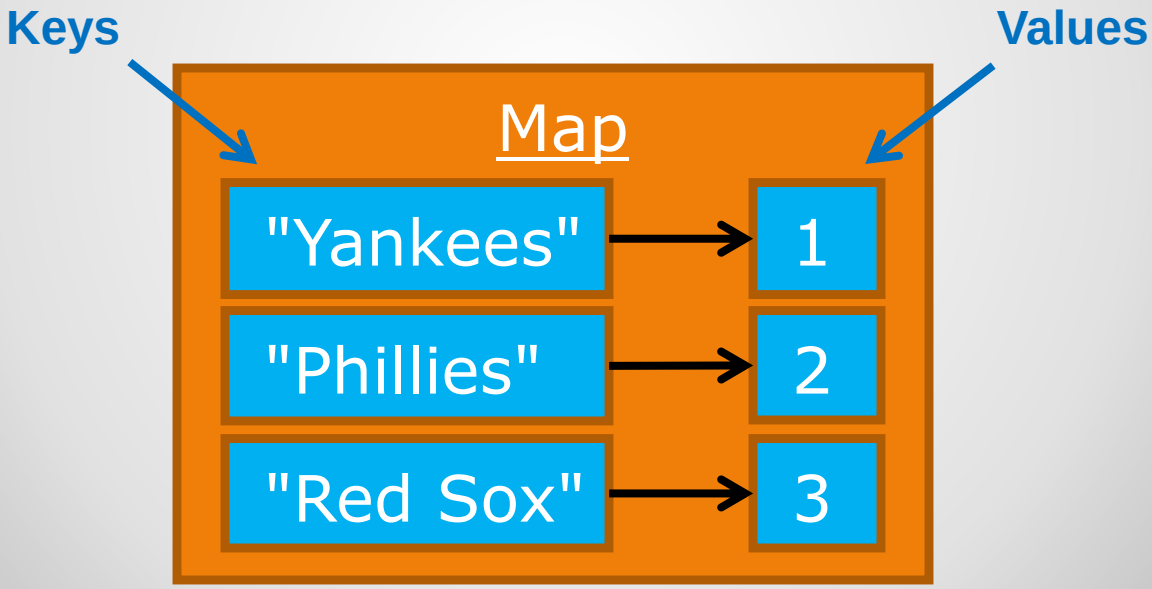
Again, the differences between them are the "speed" of specific operations.

- Now on to maps…

Other Collections

Map

- Maps associate one value with another value.
- Data are stored as key-value pairs.
- This map associates String objects to Integer objects.
- The team is the key and the number is the value.



Map

interface Map<K, V>

- Does **NOT** inherit from Collection interface.

- Associates keys with values.
- K is the key data type.
- V is the value data type.

interface Map<K, V>

Java Map Implementation Classes

- **HashMap** – Uses a hash table internally. Faster lookup time than TreeMap.
- **TreeMap** – Uses a tree internally.

- Main differences between them are the "speed" of specific operations.

Map Classes

Create and Use Map (HashMap)

- Create an empty map using **HashMap**:

Key Type

Value Type

Create an instance of HashMap

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

Make sure to include the "diamond" operator <>

Key

Value

```
myMap.put("Yankees", 1);
myMap.put("Phillies", 2);
myMap.put("Red Sox", 3);
```

Add data to the map

Get the value associated with the key "Yankees"

```
Integer value = myMap.get("Yankees");
System.out.println(value);
```

get will return null if the key is not found

Create a Map (HashMap)

Show Map Data

- Print all key-value pairs:

```
for (Map.Entry<String,Integer> entry : myMap.entrySet()) {
   System.out.println(entry.getKey() + ", " + entry.getValue());
}
```

- Print all keys:

```
for (String key : myMap.keySet()) {
   System.out.println(key);
}
```

- Print all values:

```
for (Integer v : myMap.values()) {
   System.out.println(v);
}
```

Show Map Data

Create and Use Map (TreeMap)

- Create an empty map using **TreeMap**:

```
Map<String, Integer> myMap = new TreeMap<>();

myMap.put("Yankees", 1);
myMap.put("Phillies", 2);
myMap.put("Red Sox", 3);

Integer value = myMap.get("Yankees");
System.out.println(value);
```

Create an instance of TreeMap

Make sure to include the "diamond" operator <>

Add data to the map

Get the value associated with the key "Yankees"

Create a Map (TreeMap)

- **End of Slides**

End of Slides