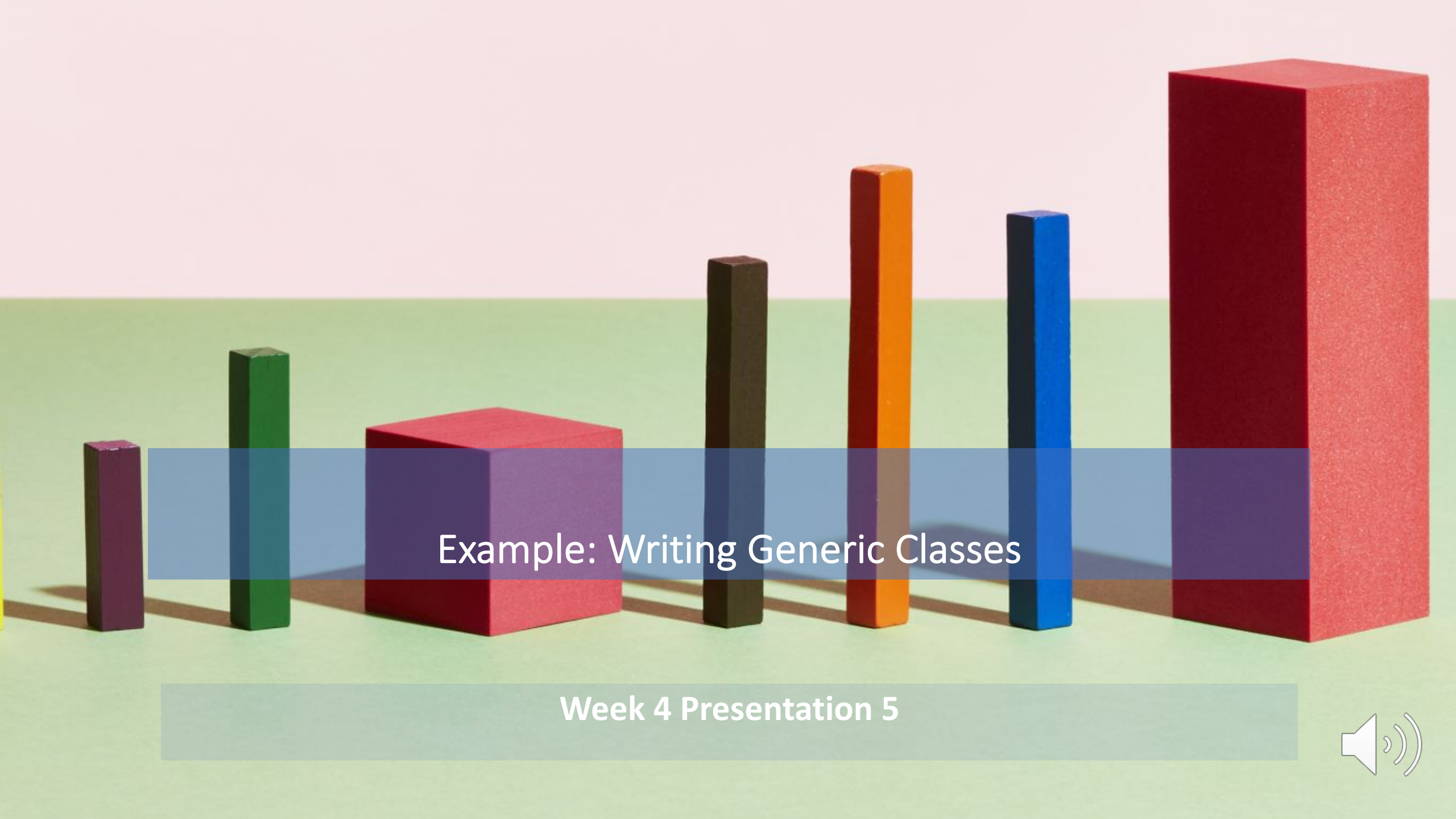


Example: Writing Generic Classes

Week 4 Presentation 5

To ensure that the only operations that are performed on the list are the queue operations `enqueue`, `dequeue`, and `isEmpty`, we can create a new class that contains the linked list as a private instance variable.

A Queue as a Linked List

```
class QueueOfStrings {
   private LinkedList<String> items = new LinkedList<>();
   public void enqueue(String item) {
      items.addLast(item);
   }
   public String dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

if we want queues of *Integers* or *Doubles* or *Colors* or any other type, we can write a **generic** *Queue* class that can be used to define queues of any type of object.

Generic Class: A Queue as a LinkedList

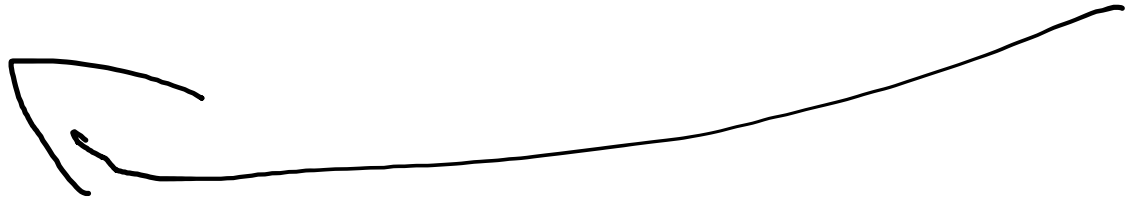
Given this class definition, we can use parameterized types such as *Queue<String>* and *Queue<Integer>* and *Queue<Color>*.

```
class Pair<T,S> {
   public T first;
   public S second;
   public Pair (T a, S b) {  // Constructor.
      first = a;
      second = b;
   }
}
```

can be used to declare variables and create objects such as:

```
Pair<String,Color> colorName = new Pair<>("Red", Color.RED);
Pair<Double,Double> coordinates = new Pair<>(17.3,42.8);
```

It's also easy to define generic classes and interfaces that have two or more type parameters, as is done with the standard interface *Map<K,V>*. A typical example is the definition of a "Pair" that contains two objects, possibly of different types. A simple version of such a class can be defined as the class Pair here



```
/**
 * Returns the number of times that itemToCount occurs in list. Items
 * in the list are tested for equality using itemToCount.equals(),
 * except in the special case where itemToCount is null.
 */
public static int countOccurrences(String[] list, String itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (String listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (String listItem : list)
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

For a generic method, the "<T>" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (T listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (T listItem : list)
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

For a generic method, the "<T>" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (T listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (T listItem : list)
         if (itemToCount.equals(list
            count++;
   }
   return count;
}
```

If `wordList` is of type `String[]` and `word` is of type *String*, then

```
int ct = countOccurrences( wordList, word );
```

will count the number of times that `word` occurs in `wordList`.

If `palette` is of type `Color[]` and `color` is of type *Color*, then

```
int ct = countOccurrences( palette, color );
```

will count the number of times that `color` occurs in `palette`.

If `numbers` is a variable of type `Integer[]`, then

```
int ct = countOccurrences( numbers, 17 );
```

will count the number of times that `17` occurs in `numbers`.

```
public static <T> int countOccurrences(Collection<T> collection, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T item : collection )
         if (item == null)
            count++;
   }
   else {
      for ( T item : collection )
         if (itemToCount.equals(item))
            count++;
   }
   return count;
}
```

The countOccurrences method operates on an array. We could also write a similar method to count occurrences of an object in any collection...