As a concrete example, Code Fragment 9.3 defines a comparator that evaluates strings based on their length (rather than their natural lexicographic order).

```
public class StringLengthComparator implements Comparator<String> {
  /** Compares two strings according to their lengths. */
  public int compare(String a, String b) {
    if (a.length( ) < b.length( )) return −1;
    else if (a.length( ) == b.length( )) return 0;
    else return 1;
  }
}
```

Code Fragment 9.3: A comparator that evaluates strings based on their lengths.

Comparators and the Priority Queue ADT

For a general and reusable form of a priority queue, we allow a user to choose any key type and to send an appropriate comparator instance as a parameter to the priority queue constructor. The priority queue will use that comparator anytime it needs to compare two keys to each other.

For convenience, we also allow a default priority queue to instead rely on the natural ordering for the given keys (assuming those keys come from a comparable class). In that case, we build our own instance of a DefaultComparator class, shown in Code Fragment 9.4.

```
public class DefaultComparator<E> implements Comparator<E> {
  public int compare(E a, E b) throws ClassCastException {
    return ((Comparable<E>) a).compareTo(b);
  }
}
```

Code Fragment 9.4: A DefaultComparator class that implements a comparator based upon the natural ordering of its element type.

9.2.3 The AbstractPriorityQueue Base Class

To manage technical issues common to all our priority queue implementations, we define an abstract base class named AbstractPriorityQueue in Code Fragment 9.5. (See Section 2.3.3 for a discussion of abstract base classes.) This includes a nested PQEntry class that implements the public Entry interface.

Our abstract class also declares and initializes an instance variable, comp, that stores the comparator being used for the priority queue. We then provide a protected method, compare, that invokes the comparator on the keys of two given entries.