A static nested class

To illustrate the use of a static nested class, we use the task of writing a class that maintains a set of integers, also recording the time each integer was added to the set.

Class Entry

For each integer in the set we have to maintain two values: the integer and the time at which it was added to the set. Therefore, we write a class Entry (say), each instance of which will contain the two values. We don't want to the user to know about class Entry, so we make it a nested class. It can be static, because it does not refer to the components of the outer class. Here is class Entry:

```
/** An instance maintains an integer and the time the entry was created. */
private static class Entry {
    private int i;    // the integer
    private long t; // the time at which this Entry was created.

    /** Constructor: an instance for integer k. */
    private Entry(int k) {
        t= System.currentTimeMillis();
        i= k;
    }

    /** Return true iff ob is an Entry with the same integer as this one. */
    public @Override boolean equals(Object ob) {
        return ob instanceof Entry   &&   i == ((Entry)ob).i;
    }
}
```

We make some points about class Entry:

1. The constructor is private! We will see later that this doesn't matter.
2. Function System.currentTimeMillis() returns the time in milliseconds since midnight, 1 January 1970.
3. Function equals depends only on integer i and not on time t. Remember, the set we will maintain is a set of integers, and the times at which they were added to the set is just added information. This function will be used to search for an integer in the set.

Class TimeSet

Having written class Entry, we can now write class TimeSet (at the bottom of the next page). This is not an efficient implementation. Rather, it is done as simply as possible to show the use of a static nested class. Here are some points about the implementation.

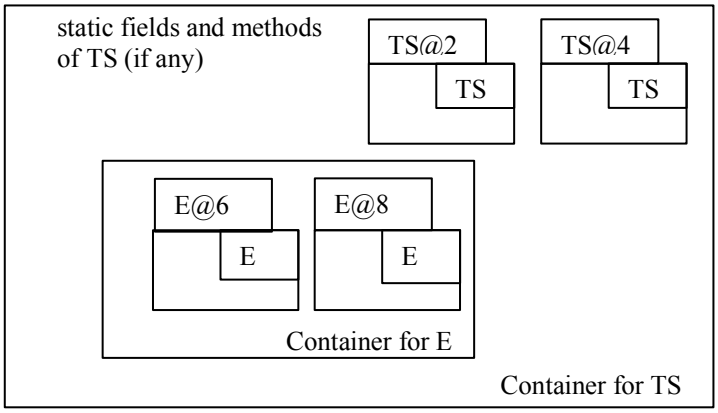
1. The set is maintained in an ArrayList.
2. Functions contains(i) and timeOf(i) have to create an object of class Entry in order to test whether i is in the set. To get around this, one could write a foreach loop to check each element of ArrayList s. Again, we aim for simplicity of explanation of a static nested class rather than efficiency.
3. Even though class Entry, its two fields, and its constructor are private, outer class TimeSet can call the constructor and reference the fields of any object of class Entry. The syntax rules for nested classes allows this. Making the class and its components private bars classes from outside the class from using them.

How to think about a static nested class

There is only one copy of a static variable. In the same way, there is only one copy of a static nested class. Conceptually, we can think of class Entry implemented as follows. On the next page is a "container" for class TimeSet (which we abbreviate as TS). You see two TS objects in it. You also see the static fields and methods of TimeSet (if it had any). Finally, since Entry is static, there is one container for class Entry, containing all of its objects. With this setup, you can see by the inside-out rule that methods in Entry can reference static fields and methods of class TS, but it cannot reference fields and methods in objects of class TimeSet directly (without having a pointer to a TimeSet object).

A static nested class

This is a conceptual view helps us understand how static nested classes work. Of course, the actual implementation may be different, but it has to have the same properties of access.




```
import java.util.ArrayList;

/** An instance maintains a set of integers, recording also the time it was added to the set. */
public class TimeSet {
   private ArrayList s= new ArrayList(); // Elements are of type Entry and contain integers in the set

   /** Constructor: an empty set. */
   public TimeSet() {}

   /** Return the size of the set. */
   public int size() { return s.size(); }

   /** Return true iff the set contains i. */
   public boolean contains(int i) { return s.contains(new Entry(i)); }

   /** Return the time in milliseconds at which i was added to the set.
     * (Return -1 if it is not in the set.) */
   public long timeOf(int i) {
      int k= s.indexOf(new Entry(i));
      return k == -1 ? -1 : ((Entry)(s.get(k))).t;
   }

   /** Add integer i to the set if it is not already in. */
   public void add(int i) {
      if (s.contains(new Entry(i))) return;
      s.add(new Entry(i));
   }

   /** An instance maintains an integer and the time the instance was created. */
   private static class Entry {
      private int i;    // the integer
      private long t; // the time at which entry was created.

      /** Constructor: an instance for integer k */
      private Entry(int k) {
         t= System.currentTimeMillis();
         i= k;
      }

      /** Return true iff ob is an Entry with the same integer as this one. */
      public @Override boolean equals(Object ob) {
         return ob instanceof Entry  &&  i == ((Entry)ob).i;
      }
   }
}
```