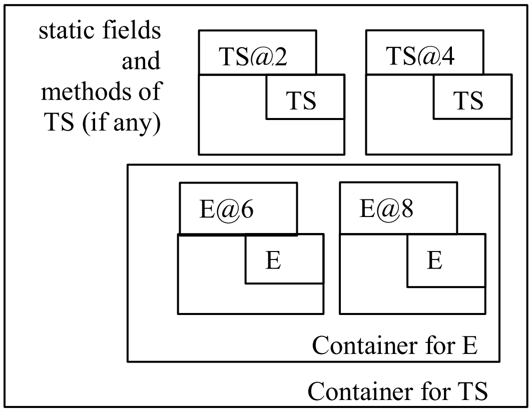
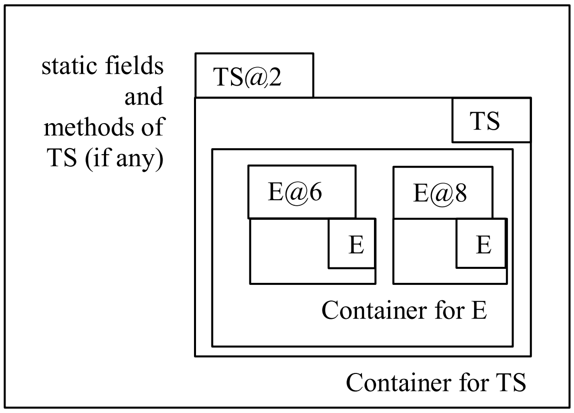
An inner class

Introduction

In a note on static nested classes, we said that class Entry, defined in class TimeSet, could be static because it did not refer to any component (e.g. field or instance method) of outer class TimeSet. But if some method in class Entry has to refer to field or method in TimeSet, then Entry cannot be static. We show a realistic example of this. But first, let us present a conceptual view of a non-static nested class, which is called an *inner class*.

To the left below, we show a conceptual view of class Entry as a static nested class of class TS (TS stands for TimeSet and E for Entry). However, if E is going to be non-static, so that it can refer to fields and methods of an object of class TS, then *its container must reside within each object of class TS*! This is shown to the right. We show only one object of class TS; it contains a container for all objects of class E that were created from this TS object. By the inside-out rule, each method in each E object can reference the fields and methods of the outer TS object.





Using generics

In the note on static nested classes, class TimeSet, outlined to the left below, maintains a set of integers. ArrayList s contains one object of static nested class Entry for each integer in the set; that object contains the integer and the time at which it was added to the set.

In order to be able to maintain not just a set of integers but a set of any class-type, we use generics. To the right below, you see that the class is declared as TimeSetE<E>. Field s now has type ArrayList<E>, and this means that field i of nested class Entry has to have type E.

This means that nested class Entry cannot be static. For example, here are two possible TimeSet objects:

TimeSetE<Integer> tsi= new TimeSetE<Integer>;   TimeSetE<String> tss= new TimeSetE<String>;

Object tsi needs objects of class Entry with field i having type Integer. Object tss needs objects of class Entry with field i having type String. So, class Entry has to be an *inner class*, residing in each object of class TimeSetE so that it has, by the inside-out rule, access to E.

```
/** An instance maintains a set of integers,
  * recording the time each was added .... */
public class TimeSet {
    private ArrayList s= …;
    …
    private static class Entry {
        private int i;    // the integer
        private long t; // the time …
        …
    }
}
```

```
/** An instance maintains a set of type E,
  * recording the time each was added .... */
public class TimeSetE<E> {
    private ArrayList<Entry> s= …;
    …
    private class Entry {
        private E i;      // the integer
        private long t; // the time …
        …
    }
}
```

©David Gries, 2018

An inner class

Below, we show the complete class TimeSetE, with class Entry being an inner class. The power of the object-oriented approach, with not only classes and subclasses but also inner classes, together with generics and the set of classes that comes with Java, like ArrayList, makes this data structure so simple and short.

```
import java.util.ArrayList;

/** An instance maintains a set of elements of type E,
  * recording also the time each element was added to the set. */
public class TimeSetE<E> {
   private ArrayList<Entry> s= new ArrayList<Entry>(); // Elements are of type Entry
                                                        // and contain elements in the set
   /** Constructor: an empty set. */
   public TimeSetE() {}

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

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

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

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

   /** An instance maintains an element of type E and the time the instance was created. */
   private class Entry {
      private E i;      // the element of type E
      private long t; // the time at which entry was created.

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

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