Lab 2: Sets of Integers

originally by Gavin Lowe, updates by Joe Pitt-Francis

- This is a practical for *Imperative Programming Part 2*.
- Sign-off deadline is your practical session in Week 8.
- You must implement, comment and test non-optional operations; and answer specific questions.

The goal of this practical is to implement a class, each object of which acts like a mutable set of integers.

state: $S : \mathbb{P}\,\texttt{Int}$

We want to be able to create sets, add elements to them, remove elements from them, test for membership, test for equality between sets, form unions and intersections, and perform operations like map and filter.

Remember that sets have no order, so the set `{1, 2}` is the same as the set `{2, 1}`. Also, sets do not have repeated elements, so the set `{1, 1}` is the same as the set `{1}`.

Scala has a class `scala.collection.mutable.Set[T]`, each object of which represents a set containing elements of type `T`. The class you will implement will be similar to this, except we'll only consider integer elements (for simplicity), and we won't implement all of the operations (because there are so many).

On the course website you will find a file `IntSet.scala`, which will act as a template for your implementation. That file defines a class `IntSet`, but where the operations remain unimplemented; it also defines a companion object.

You will also find an example ScalaTest file from a Part One lecture which may help you get started with writing ScalaTest tests. Because it's meant to show you what ScalaTest can do, it doesn't test `IntSet` functionality and not all of its tests are meant to pass. Instructions for compiling and running are given in file itself:

```
fsc TestTest.scala
scala org.scalatest.run TestTest
```

This should work on the lab machines *without you needing to add any -cp flags or set environmental variables.* (See course website if you need ScalaTest on your own machine.)

Reporting

Your report should be in the form of a well-commented definition for the class `IntSet`, together with answers to the points raised below, and a ScalaTest test suite that tests the operations you write. (For guidance: my test suite contains 44 tests, which make a total of 59 individual checks.)

You can earn an S by satisfactory implementations of the operations excluding those marked "optional". You can earn an S+ by satisfactory implementations of all the operations, or particularly efficient implementations of all the non-optional operations and some of the optional ones.

The state of objects

Your implementation should use a linked list internally, using nodes of the following type

```
private class Node(var datum: Int, var next: Node)
```

You'll see this definition inside the companion object. Inside the `IntSet` class, you'll see the lines

```
private type Node = IntSet.Node
// Constructor
private def Node(datum: Int, next: Node) = new IntSet.Node(datum, next)
```

which define some aliases, so we can subsequently write "`Node`" for the type, rather than "`IntSet.Node`", and can construct new nodes using, e.g., `Node(x, null)` rather than `new IntSet.Node(x, null)`.

Internally, each object of the `IntSet` class will have a private variable

```
private var theSet : Node = ...
```

that stores the linked list containing the elements of the set.

However, there are a number of design decisions that you need to make:

- Do you use a dummy header node?
- Can your linked list store the same integer several times, or do you avoid repetitions?
- Do you store the elements of the set in increasing order?
- Do you include anything else in your state, for efficiency?

You should aim for an implementation that is (asymptotically) efficient, especially for the more common operations. *I strongly recommend that you read*

the rest of these practical instructions before deciding your answers to these questions, so you know what operations you will have to implement. There is no single set of correct answers to these questions, but some answers will give more efficient implementations than others.

Document your design decisions, by writing down an abstraction function and a datatype invariant. Also briefly describe the reasons for your decisions (about one sentence per question).

The operations

The remainder of the practical involves implementing the various operations, described below. For each operation you should state the complexity of the implementation (in $O(\_)$ notation), in terms of the size of the set and (where applicable) the size of any set provided as an argument. You should state (informally or formally) invariants for non-trivial loops that you write.

Initialisation Your class should initialise objects corresponding to the empty set.

init: $S = \{\}$

`toString` Write a definition for an operation

```
override def toString : String = ???
```

that converts a set into a string. (A default implementation of `toString` is provided in the class `Any`, the root of the class hierarchy; hence we have to use the keyword `override` to override the default definition.) The operation should produce a string such as `"{1, 2}"`; in particular, elements should not appear twice; however, the order in which elements are displayed does not matter.

`add` Write a definition for an operation

```
/** post: S = S0 ∪ {e} */
def add(e: Int) : Unit = ???
```

that adds the element `e` to the set.

Aside: factory methods At present, if we want to create an object representing the set `{1, 2, 3}`, we have to type something like

```
val s = new IntSet; s.add(1); s.add(2); s.add(3)
```

It would be nicer to be able to just type

```
val s = IntSet(1,2,3)
```

In the companion object there is some code that allows us to do this:

```
/** Factory method for sets.
   * This will allow us to write, for example, IntSet(3,5,1) to
   * create a new set containing 3, 5, 1 -- once we have defined
   * the main constructor and the add operation.
   * post: returns res s.t. res.S = {x_1, x_2, ..., x_n}
   *         where xs = [x_1, x_2, ..., x_n] */
  def apply(xs: Int*) : IntSet = {
   val s = new IntSet; for(x <- xs) s.add(x); s
 }
```

Recall that the notation `IntSet(1,2,3)` is short for `IntSet.apply(1,2,3)`, so this calls the `apply` function defined above. In the signature of `apply`, the notation `xs:Int*` represents that `apply` takes zero or more arguments, each of which should be an `Int`. The body of `apply` simply creates a new set `s`, and adds each element of `xs` to it.

The `apply` function is called a *factory method*, because it acts like a factory that builds new objects. Here the `apply` operation does not act on any particular `IntSet` object, so its definition belongs in the companion object.

`size` Write a definition for a function

```
/** post: S = S_0 ∧ returns #S */
def size : Int = ???
```

that returns the size of the set.

`contains` Write a definition for a function

```
/** post: S = S_0 ∧ returns (e ∈ S) */
def contains(e: Int) : Boolean = ???
```

that tests whether the set contains the element `e`.

If your code for `contains` has code that also appears in `add`, then you should factor it out into a separate (private) function.

`any` Write a definition for a function

```
/** pre: S ≠ {}
  * post: S = S_0 ∧ returns e s.t. e ∈ S */
def any : Int = ???
```

that returns an element of the set — but it doesn't matter which element is returned.

Think about what this function should do when called on the empty set.

`equals` Write a definition for a function

```
/** post: S = S_0 ∧ returns that.S = S */
override def equals(that: Any) : Boolean = ???
```

that tests whether this set is equal to the provided argument `that`, i.e. they contain the same elements. The `equals` operation is given a default implementation in the class `Any`, so we again need to use the `override` keyword. Note also that the type of `that` is `Any`: it might not be an `IntSet`. I therefore suggest that you use a definition of the following form:

```
override def equals(that:Any) : Boolean = that match {
  case s: IntSet => {???}
  case _ => false
}
```

The pattern `s: IntSet` matches just a value that is indeed an `IntSet`, and binds the name `s` to that value. Hence within the omitted code ("`{???}`") we can assume that `s` is indeed an `IntSet`. Note also that this definition will return `false` if `that` is not an `IntSet`, as required.

Once you have defined `equals`, you will be able to use the normal infix operators `==` and `!=` to test `IntSet`s for equality or inequality. (The `Any` class defines these two operators in terms of `equals`.)

`remove` Write a definition for a function

```
/** post: S = S_0 − {e} ∧ returns (e ∈ S_0) */
def remove(e: Int) : Boolean = ???
```

that removes the element `e` from the set (if it is there); the function should return `true` precisely if `e` was initially an element of the set.

If your code for `remove` has code that also appears in `add` and/or `contains`, then you should factor it out into a separate (private) function.

`subsetOf` Write a definition for a function

```
/** post: S = S_0 ∧ returns S ⊆ that.S */
def subsetOf(that: IntSet) : Boolean = ???
```

that tests whether this set is a subset of the provided argument `that`.

`union` **(optional)** Write a definition for a function

```
/** post: S = S_0 ∧ returns res s.t. res.S = S ∪ that.S */
def union(that: IntSet) : IntSet = ???
```

that returns a new set built as the union of this set and the provided argument `that`. Note that the new set should be independent from this set and `that`: they should not share any nodes.

`intersection` **(optional)** Write a definition for a function

```
/** post: returns res s.t. res.S = S ∩ that.S */
def intersect(that: IntSet) : IntSet = ???
```

that returns a new set built as the intersection of this set and the provided argument `that`. Again the new set should not share any nodes with the original sets.

`map` **(optional)** Write a definition for a function

```
/** post: S = S_0 ∧ returns res s.t. res.S = {f(x) | x ∈ S} */
def map(f: Int => Int) : IntSet = ???
```

that returns a new set built by applying `f` to all the elements of this set.

`filter` **(optional)** Write a definition for a function

```
/** post: S = S_0 ∧ returns res s.t. res.S = {x | x ∈ S ∧ p(x)} */
def filter(p : Int => Boolean) : IntSet = ???
```

that returns a new set that contains those elements of the current set that satisfy the predicate `p`.

More operations (very optional) Have a look at the API documentation for `scala.collection.mutable.Set`, pick some operations from there, and add corresponding operations to your implementation of `IntSet`.