# CO518 Assessment 1

In this assessment you are requested to produce a bespoke implementation for sets of integers (that is: integers of type `int`). A set is a collection of elements that has no particular order, and no concept of multitude of occurrences – two sets are the same if the contain the same elements.

First, here is the specification of how the implementation should be structured.

The overall type of integer sets should be called `IntSet`, and this should be an interface.

We then distinguish 3 kinds of concrete sets:

- The empty set. We really only need one such thing, but because it behaves differently from any other set it is a class too. So, this gives us the set {}.
- Singleton sets. These are sets containing precisely one number, e.g. {8} or {42}.
- Tree sets. These are sets containing at least two numbers. All tree sets are a tree of two branches which are both integer sets. However, all even numbers go into the left branch, all odd numbers go into the right branch. Insides those branches we forget their last bit, i.e. divide all numbers by 2.

Here is an example: How do we represent the set {2,3,6,8}? It has more than 1 element, so it is a tree set. We put all the even numbers to the left (but halved), so the left branch is the set {1,3,4}. All the odd numbers (well, the 3) go right, but again halved, so the right branch of the tree set is the set {1}. These two sets in turn are represented using the same principles: {1,3,4} has more than one element, so it is a tree set with {2} on the left and {0,1} on the right. {2} is a singleton (as is the {1} we had earlier), {0,1} is a tree set with {0} in both branches; and {0} is a singleton. Notice that it can happen in a tree set that one branch is the empty set: for {2,4,64,100} we get an empty right branch, and the left branch is {1,2,32,50}.

All our set objects are (should be) immutable, i.e. you cannot change the content of a set object after construction. So, how do you build up a set? Not entirely unlike you build an arraylist: you initially create an empty set object, and then the ‘add’ method adds elements – but instead of modifying an existing set each call returns the new set as a result. For example, {2,3,6,8} can be built as empty().add(2).add(3).add(6).add(8).

It is sometimes desirable to define classes with so-called *smart constructors*. This means: we hide the normal constructor(s) from ordinary folk, and supply them instead with a static method that does the constructing and which returns an object of the right type. However, this object will not always be a freshly constructed one, or even use the constructor of this subclass, so this gives us an opportunity to be smart. For certain smart constructors, and that applies here, some data needs to be stored in static variables, so that the smart constructor has access to it.

Criteria for this assessment: is the class design ok? Are the right values represented? Are the sets indeed immutable? Are the requirements for sharing sets implemented? Is the implementation run-time efficient? Generally, you can create more methods than specified if you want to – but no additional fields.

1. Define the 3 classes and the interface as indicated above.
    a. The class of empty sets should use a smart constructor which always returns the same object.
    b. The class of singleton sets should also use a smart constructor, singleton(n). Note that singletons with small numbers in them are relatively frequent in our representation. So, we require that if $0 \leq n \leq 7$ then two different calls of singleton(n) should return identical objects. So, these objects need to be memoized somehow.
    c. For the class of tree sets there are no special requirement of object sharing. Note though that the class design means that numbers change when you go down the tree. For example, when checking whether the number 35 is contained in a tree set you go the right branch (because 35 is odd) and look there for 17 (because this is 35/2).

    30 Marks: 5 for IntSet/TreeSet each, 10 for EmptySet/Singleton each.

2. The IntSet interface (and thus all its implementing classes) should provide the following methods:
    a. `IntSet add(int x)`, which adds the number x to the set, i.e. returning a set object that contains all numbers of **this** set, plus the number x, but no others. Beware that the number x changes when you go down the branches of a tree set. Hint: spend some thought on how to make this work for singleton sets, an auxiliary method that builds a set for two numbers may be useful.
    b. `boolean contains(int x)`. This should return true iff x is in this set. Again, as you traverse the branch(es) of a tree set the number x will change (unless it is 0 to begin with).
    c. Intset union(IntSet other). This should compute to union of this IntSet with the other IntSet, returning an IntSet that represents this union. For example, if x represents the set {4,7,9,11} and y the set {2,7,9,13} then x.union(y) should represent the set {2,4,7,9,11,13}, as $\{4,7,9,11\} \cup \{2,7,9,13\} = \{2,4,7,9,11,13\}$.
    d. `String toString()`. This is inherited from class object, but we should replace it with a bespoke method that puts the elements next to one another, separated by commas, and enclosed in curly brackets. The order of elements in the output is not important. Hint: notice that a straightforward recursive implementation will not work well for tree sets. One could require an auxiliary method (for all IntSet) that gives sufficient context information to produce the required output. Done well, toString() could be given a default implementation in IntSet only, leaving other classes to implement the auxiliary operation only.

    70 marks: 15 for 2a/b/c each, 25 for 2d.