Sets

A Set is an abstract data type representing an unordered collection of distinct items.

Sets appear in many problems: All the words used by Shakespeare. All correctly spelled words. All prime numbers. All the pixels of the same color that should be flooded in flood-fill.

We could represent a set as an array or a list, but that is not natural (and often not efficient): Lists are ordered sequences of not necessarily distinct elements.

```
scala> val s = Set(2, 3, 5, 7, 9)
s: scala.collection.immutable.Set[Int] =
  Set(9, 5, 2, 7, 3)
```

Empty set: `Set()`

Set operations

The standard set operations have operators:

- union `s1 union  s2`
- intersection `s1 intersect s2`
- difference `s1 diff s2`
- is `x` in `s`? `s contains x`
- is `s1` subset of `s2`? `s1 subsetOf s2`

Sets

Sets are unordered and elements are distinct:

```
scala> val s2 = Set(9, 9, 5, 7, 3, 5, 3, 2)
s2: Set[Int] = Set(9, 5, 2, 7, 3)
scala> s == s2
res3: Boolean = true
```

Adding and removing elements:

```
scala> s + 11
res0: Set[Int] = Set(11, 9, 5, 2, 7, 3)
scala> s - 6
res1: Set[Int] = Set(9, 5, 2, 7, 3)
scala> s - 5
res2: Set[Int] = Set(9, 2, 7, 3)
scala> s + 7
res3: Set[Int] = Set(9, 5, 2, 7, 3)
```



```
scala> val A = (1 to 10).toSet
A: Set[Int] = Set(8, 4, 9, 5, 10, 6, 1, 2, 7, 3)
scala> val B = (1 to 10 by 2).toSet
B: Set[Int] = Set(9, 5, 1, 7, 3)
scala> val C = (1 to 5).toSet
C: Set[Int] = Set(4, 5, 1, 2, 3)

scala> (B subsetOf A, C subsetOf B, C subsetOf A)
res5 = (true,false,true)
scala> A diff B
res6: Set[Int] = Set(8, 4, 10, 6, 2)
scala> B union C
res7: Set[Int] = Set(4, 9, 5, 1, 2, 7, 3)
scala> B intersect C
res8: Set[Int] = Set(5, 1, 3)
```

A simple spell checker

```
val F = scala.io.Source.fromFile("words.txt")
val words = F.getLines().toSet

while (true) {
  val w = readLine("Enter a word> ").trim
  if (w == "")
    sys.exit()
  if (words contains w)
    println(w + " is a word")
  else
    printf("Error: %s is not a word\n", w)
}
```


Applications

- A spell checker.
  (Use set of correctly spelled words.)
- Measuring similarity between texts.
  (Consider set of words of each text, look at the size of their intersection and union.)
- Computing prime numbers.
  (Sieve of Erathosthenes).
- Remembering visited positions in a maze.

Mutable Sets

Scala also provides a mutable Set type:

```
scala> val S =
         scala.collection.mutable.Set(1, 2, 3, 4)
S: scala.collection.mutable.Set[Int] =
   Set(2, 1, 4, 3)

scala> S += 9
res0: S.type = Set(9, 2, 1, 4, 3)

scala> S += 13
res1: S.type = Set(9, 2, 1, 4, 13, 3)

scala> S -= 2
res2: S.type = Set(9, 1, 4, 13, 3)
```


Maps

Let's add variables to our simple calculator.
A variable should store a number.

```
> A = 7
==> A = 7
> 3 * (A + 5)
==> 36
```

Data type "Map"

We need a data structure to store pairs of (variable name, variable value), that is (String, Double).

It should support the following operations:

- insert a new variable definition (given name and value),
- find a variable value, given its name

This abstract data type is called a map (or dictionary).
A map implements a mapping from some key type to some value type.

Scala maps

A Scala map implements the trait `Map[K,V]`.
We can think of a map as a container for `(K,V)` pairs.

```
scala> val m1 = Map(("A",3), ("B",7))
m1: scala.collection.immutable.Map[String,Int] =
  Map((A,3), (B,7))
```

However, Scala provides a nicer syntax to express the mapping:

```
scala> val m = Map("A" -> 7, "B" -> 13)
m: scala.collection.immutable.Map[String,Int] =
  Map((A,7), (B,13))
```


Querying maps

```
scala> val m = Map("A" -> 7, "B" -> 13)

scala> m("A")
res1: Int = 7
scala> m("C")
java.util.NoSuchElementException: key not found: C
scala> m contains "C"
res2: Boolean = false
scala> m contains "A"
res3: Boolean = true
scala> m.getOrElse("A", 99)
res4: Int = 7
scala> m.getOrElse("C", 99)
res5: Int = 99
```


Updating maps

```
scala> val m = Map("A" -> 7, "B" -> 9)
m: Map[String,Int] = Map((A,7), (B,9))
scala> m + ("C" -> 13)
res0: Map[String,Int] = Map((A,7), (B,9), (C,13))
scala> m - "A"
res1: Map[String,Int] = Map((B,9))
scala> m - "C"
res2: Map[String,Int] = Map((A,7), (B,9))
scala> m + ("A" -> 99)
res3: Map[String,Int] = Map((A,99), (B,9))
```

Mutable maps

We can also use mutable maps:

```
scala> import scala.collection.mutable.Map
scala> val m = Map("A" -> 7, "B" -> 9)
m: Map[String,Int] = Map(B -> 9, A -> 7)
scala> m += ("C" -> 13)
res0: m.type = Map(C -> 13, B -> 9, A -> 7)
scala> m -= "A"
res1: m.type = Map(C -> 13, B -> 9)
scala> m("A") = 19
scala> m("B") = 99
scala> println(m)
Map(C -> 13, A -> 19, B -> 99)
```


Variables in our calculator

```
object Calculator {

  var variables = Map[String, Double]()
  // ...
```

In parseItem:

```
if (variables contains t.text)
  variables(t.text)
else
  throw new SyntaxError(startPos,
            "Undefined variable: " + t.text)
```


Concordance

A concordance lists all the words in a text with the line numbers where it appears.

```
1: Friends, Romans, countrymen, lend me your ears;
2: I come to bury Caesar, not to praise him.
3: The evil that men do lives after them;
4: The good is oft interred with their bones;
5: So let it be with Caesar. The noble Brutus
6: Hath told you Caesar was ambitious:
7: If it were so, it was a grievous fault,
8: And grievously hath Caesar answer'd it.
9: Here, under leave of Brutus and the rest–
10: For Brutus is an honourable man;
11: So are they all, all honourable men–
12: Come I to speak in Caesar's funeral.
13: He was my friend, faithful and just to me:
14: But Brutus says he was ambitious;
15: And Brutus is an honourable man.
16: He hath brought many captives home to Rome
17: Whose ransoms did the general coffers fill:
18: Did this in Caesar seem ambitious?
```

```
A         : 7,24
AFTER     : 3
ALL       : 11,11,23,30
AM        : 29
AMBITION      : 20,25
AMBITIOUS     : 6,14,18,21,26
AN        : 10,15,22,27
AND       : 8,9,13,15,22,27
ANSWER'D      : 8
ARE       : 11
....
WHOSE     : 17
WITH      : 4,5,33,34
WITHHOLDS     : 31
WITHOUT   : 30
YET       : 21,26
YOU       : 6,23,30,31
YOUR      : 1
```


Building a concordance

1. Create an empty map.
2. Scan the text word by word. For each word, look it up in the map.
   (a) If it does not yet appear, add it with the current line number.
   (b) If it already appears, add the current line number to its value.
3. Print out the map.

Concordance

```
var concordance = Map[String, String]()
var lineNum = 0
for (line <- F.getLines()) {
  lineNum += 1
  println(lineNum + ":\t" + line);
  val words = line.split("[ ,:;.?!-]+") map
    (_.toUpperCase)
  for (word <- words) {
    if (concordance contains word) {
      val lns = concordance(word)
      concordance += (word -> (lns +","+ lineNum))
    } else {
      concordance += (word -> ("" + lineNum))
    }
  }
}
```


Duplicated line numbers

```
var concordance = scala.collection.immutable.
  TreeMap[String, List[Int]]()
var lineNumber = 0
for (line <- F.getLines()) {
  val words = line.split("[ ,:;.?!-]+")
        map (_.toUpperCase)
  for (word <- words) {
    val lns = concordance.getOrElse(word, Nil)
    if (lns == Nil || lns.head != lineNumber)
      concordance += (word -> (lineNumber :: lns))
  }
}
for ((word, lns) <- concordance)
  println(word +": "+ lns.reverse.mkString(","))
```


Printing the map

```
for ((word, lns) <- concordance)
  printf("%20s: %s\n", word, lns)
```

But keys appear in some "random" order.

Scala provides several `Map` implementations: `HashMap`, `TreeMap`, `ListMap`.

All implement the `Map` trait, but their behavior and the running times are not the same.

The power of abstract data types: We can easily switch between different implementations.