Another fundamental collection type is the *map*.

A map of type Map[Key, Value] is a data structure that associates keys of type Key with values of type Value.

Examples:

```scala
type romanNumerals = Map("I" -> 1, "V" -> 5, "X" -> 10)
type capitalOfCountry = Map("US" -> "Washington", "Switzerland" -> "Bern")
```
Maps are Iterables

Class Map[Key, Value] extends the collection type
Iterable[(Key, Value)].

Therefore, maps support the same collection operations as other
iterables do. Example:

```scala
val countryOfCapital = capitalOfCountry map {
  case (x, y) => (y, x)
}  // Map("Washington" -> "US", "Bern" -> "Switzerland")
```

Note that maps extend iterables of key/value pairs.

In fact, the syntax key -> value is just an alternative way to write
the pair (key, value).
Maps are Functions

Class Map[Key, Value] also extends the function type Key => Value, so maps can be used everywhere functions can.

In particular, maps can be applied to key arguments:

capitalOfCountry("US") // "Washington"
Querying Map

Applying a map to a non-existing key gives an error:

```java
capitalOfCountry("Andorra")
    // java.util.NoSuchElementException: key not found: Andorra
```

To query a map without knowing beforehand whether it contains a given key, you can use the `get` operation:

```java
capitalOfCountry get "US"      // Some("Washington")
capitalOfCountry get "Andorra" // None
```

The result of a `get` operation is an Option value.
The Option Type

The Option type is defined as:

```scala
trait Option[+A]
case class Some[+A](value: A) extends Option[+A]
object None extends Option[Nothing]
```

The expression `map get key` returns:

- `None` if map does not contain the given key,
- `Some(x)` if map associates the given key with the value `x`.
Decomposing Option

Since options are defined as case classes, they can be decomposed using pattern matching:

```scala
def showCapital(country: String) = capitalOfCountry.get(country) match {
  case Some(capital) => capital
  case None => "missing data"
}

showCapital("US")       // "Washington"
showCapital("Andorra")   // "missing data"
```

Options also support quite a few operations of the other collections.

I invite you to try them out!
Sorted and GroupBy

Two useful operation of SQL queries in addition to for-expressions are groupBy and orderBy.

orderBy on a collection can be expressed by sortWith and sorted.

```scala
val fruit = List("apple", "pear", "orange", "pineapple")
fruit sortWith (_.length < _.length) // List("pear", "apple", "orange", "pineapple")
fruit.sorted // List("apple", "orange", "pear", "pineapple")
```

groupBy is available on Scala collections. It partitions a collection into a map of collections according to a *discriminator function f*.

**Example:**

```scala
fruit groupBy (_.head) //> Map(p -> List(pear, pineapple),
//| a -> List(apple),
//| o -> List(orange))
```
Map Example

A polynomial can be seen as a map from exponents to coefficients. For instance, \( x^3 - 2x + 5 \) can be represented with the map.

\[
\text{Map}(0 \rightarrow 5, \ 1 \rightarrow -2, \ 3 \rightarrow 1)
\]

Based on this observation, let's design a class `Polynom` that represents polynomials as maps.
Default Values

So far, maps were *partial functions*: Applying a map to a key value in `map(key)` could lead to an exception, if the key was not stored in the map.

There is an operation `withDefaultValue` that turns a map into a total function:

```scala
val cap1 = capitalOfCountry withDefaultValue "<unknown>"
cap1("Andorra")     // "<unknown>"
```
Variable Length Argument Lists

It's quite inconvenient to have to write

Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))

Can one do without the Map(...)?

Problem: The number of key -> value pairs passed to Map can vary.
Variable Length Argument Lists

It's quite inconvenient to have to write

```java
Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))
```

Can one do without the `Map(...)`?

Problem: The number of `key -> value` pairs passed to `Map` can vary.

We can accommodate this pattern using a repeated parameter:

```java
def Polynom(bindings: (Int, Double)*) =
    new Polynom(bindings.toMap withDefaultValue 0)
```

```java
Polynom(1 -> 2.0, 3 -> 4.0, 5 -> 6.2)
```

Inside the `Polynom` function, `bindings` is seen as a `Seq[(Int, Double)]`. 
class Poly(terms0: Map[Int, Double]) {
    def this(bindings: (Int, Double)*) = this(bindings.toMap)
    val terms = terms0 withDefaultValue 0.0
    def + (other: Poly) = new Poly(terms ++ (other.terms map adjust))
    def adjust(term: (Int, Double)): (Int, Double) = {
        val (exp, coeff) = term
        exp -> (coeff + terms(exp))
    }

    override def toString =
    (for ((exp, coeff) <- terms.toList.sorted.reverse)
        yield coeff+"x"+exp) mkString " + "
}
Exercise

The + operation on Poly used map concatenation with ++. Design another version of + in terms of foldLeft:

```scala
def + (other: Poly) =
  new Poly((other.terms foldLeft ???)(addTerm)

def addTerm/terms: Map[Int, Double], term: (Int, Double)) =
  ???
```

Which of the two versions do you believe is more efficient?

0 The version using ++
0 The version using foldLeft
Exercise

The `+` operation on `Poly` used map concatenation with `++`. Design another version of `+` in terms of `foldLeft`:

```scala
def + (other: Poly) =
  new Poly((other.terms foldLeft ???)(addTerm)

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  ???
```

Which of the two versions do you believe is more efficient?

0. The version using `++`
0. The version using `foldLeft`