Maps

Map

- A map of type ***Map[Key, Value]*** is a data structure that associates keys of type Key with values of type Value
- Examples:

  val romanNumerals = Map("I" -> 1, "V" -> 5, "X" -> 10)

  val capitalOfCountry = Map("US" -> "Washington", "Switzerland" -> "Bern")
- Maps are a special subtype of iterables: Class ***Map[Key, Value]*** extends the collection type ***Iterable[(Key, Value)]***
- Therefore, maps support the same collection operations as other iterables do
- 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)
- -> is implemented as an extension method in Predef:

  extension [K, V] (k: K)

  def -> (v: V) = (k, v)

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

  capitalOfCountry("Andorra")

  // java.util.NoSuchElementException: key not found: Andorra
- You should use the application syntax for maps only if you're sure that the key is in fact in the map

Querying Map

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

  capitalOfCountry.get("US") *// Some("Washington")*

  capitalOfCountry.get("Andorra") *//None*
- The get operation doesn't throw an exception if the key is not in the map
- The result of a get operation is an ***Option*** value

The Option None

- The Option type is defined as:

  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:

```
def showCapital(country: String) = captialOfCountry.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, even though they are not a collection type because common to all collections is that I can add arbitrary data to any collection
- Since an Option has only zero or one element, it's not a collection in that sense
- In other languages, the map would return ***null*** if the key was missing, but it turns out that null is actually really dangerous because if any value can be null, then you'll never know beforehand whether certain operations on that value are defined or not; if the value is null you would get a ***NullPointException***
- In Scala, null is actually available, but it's generally considered bad style to use it
- An Option is safer because the types force you to handle both cases (you either have something or nothing)
- If you forget the None case, then the compiler would complain and say that the pattern match is not exhaustive; it will essentially nudge you to really handle both of the cases; that makes an Option so much safer than null

Updating Maps

- Functional updates of a map are done with the + and ++ operations:
  - ***m + (k -> v)*** – the maps that takes key 'k' to value 'v' and is otherwise equal to m; 'k' might already be defined in m, in which case it will be overridden
  - ***m ++ kvs*** – here 'kvs' is a collection of pairs and the map 'm' updated via '+' with all key/value pairs in 'kvs'
- These operations are purely functional
- Example:

```
val m1 = Map("red" -> 1, "blue" -> 2) // m1 = Map("red" -> 1, "blue" -> 2)
val m2 = m1 + ("blue" -> 3) // m2 = Map("red" -> 1, "blue" -> 3)
m1 // Map("red" -> 1, "blue" -> 2)
```

- The old map stays in place while we update the new one
- For small sizes, maps are essentially single objects and we do copy the whole objects that holds for sizes up to 4
- For larger sizes, what we do is we essentially use a scheme similar to the vector scheme; We have essentially arrays of arrays of a shallow depth up to 5, and in each array we have key value pairs and we use a hash function to essentially select either the right sub-array or the right element in that array
- Similar to vectors, if we update, we get a log(n) update of these sub-arrays, where n is the depth of the tree. Basically, we copy between one and five of these sub-arrays, which is still reasonably bounded cost for updating a map

Sorted and GroupBy

- Two useful operations known from SQL queries are **groupBy** and **orderBy**
- **orderBy** on a collection can be expressed using ***sortWith*** and ***sorted***

```
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:

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

Polynomial Mapping 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:

```
Map(0 -> 5, 1 -> -2, 3 -> 1)
```

- Based on this information, let's design a class Polynom that represents polynomials as maps.

Default Values

- So far, maps were *partial functions*. Applaying 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:

```
val cap1 = capitalOfCountry.withDefaultValue("<unknown>")
cap1("Andorra")       //<unknown>
```

Variable Length Arguments Lists

- It's quite inconvenient to have to write *Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))*
- The number of key -> value pairs passed to Map can vary
- So in order to pass pairs without using the map would be by using a *repeated parameter*:

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

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

- Var-arg parameter is given by s parameter type followed by an asterisk
- Inside the Polynom function, ***bindings*** is seen as a ***Seq[(Int, Double)]***