Updated: 2021-02-18

3.9 Containers, Collections, Wrappers, etc.

Northeastern University

Real-life programming

- Unlike the Newton’s method, for example, just about every other program involves grouping things together under one identifier.

- These groups are known as containers, collections and wrappers.

What is the difference between a container, a collection and a wrapper?

- A *collection* is a data structure that collects together some data objects (duh!).
- A *container* is a data structure that contains some data (duh!)
- So, how does a collection differ from a container?
- These terms overlap quite a bit, but usually:
  - a collection is made up of 0 thru many elements *of the same type.* Such elements are usually accessed by key or index.
  - a container is made up of *disparate* elements (a container is typically a tuple with one or more fields). Such elements are usually accessed by name.
- a *wrapper* is a specialized container that, conventionally, wraps either a single value or nothing at all.

What is a container?

- Collections and containers (including wrappers) are classes and are constructed by a constructor (!)
- If, for example, we say:
    - `val y = Bag(x)`
- what we mean is *val y = Bag.apply(x)* and that means construct a *Bag* with *x* as its contents. Typically (if *Bag* is a case class) that will be the same as *val y = new Bag(x).*
- A case class is basically a tuple with named fields, and a bunch of compiler-provided methods.
- A wrapper may contain an element, or some indication of a non-element.

What about a collection?

- A collection is usually linear in its “shape”:
  - Elements can be accessed by position, by index, or by key (or all of these);
  - There may be zero through infinity elements;
  - All the elements conform to a particular “underlying” type.

Methods on these aggregate types

- *val aw: Wrapper[A]*
- How do we get the value out of *aw* when *aw* is a wrapper?
  - *val a: A = aw()* ?
    - No, we use *aw()* when *aw* is a function that take no parameters
  - *val a: A = aw get* ?
    - Yes, we use *aw get* when *aw* is a wrapper.
    - But! if *aw* is empty (or there was an error creating it), then the *get* method will typically throw an exception. We will learn therefore never to actually use *get*.

More methods on these aggregate types

- *val as: Seq[A]*
- How do we get a value out of *as* when *as* is a collection?
  - *val a: A = as(x)* ?
    - Yes, if *x* is an *Int* (for the index) or perhaps a key, then we can use *as(x)*.
    - This is short for *as.apply(x)*
    - But note that an exception might be thrown if x is not a valid index*.*
  - *val a: A = as* method ?
    - Yes, we use *as head* when we want the first element (the head).
    - But! if *as* is empty, then the *head* method will typically throw an exception. We will learn therefore to never actually use *head,* unless it’s part of a pattern match.

Even more methods on these aggregate types

- *val as: Seq[A]*
- What about other types of result (different from *A*)?
  - *val x: Int = as* method *?*
    - Yes, we can use *as size* to get the length of *as*.
  - *val b: Boolean = as* method *?*
    - Yes, we use *as isEmpty* to find out whether *as* is empty.

Yet more methods on these aggregate types

- *val as: Seq[A]*
- What about other types of result (different from A)?
  - *val xs: Seq[A] = as* method *?*
    - We can use *as tail* to get the tail of *as*.
  - *val ao: Option[A] = as get x* ?
    - An alternative to *as(x)* is *get* (not available in all collections).
    - But note that what we actually get back is the value wrapped in *Option.*
  - *val xs: Seq[A] = as* method *predicate ?*
    - This method is *filter* which takes a predicate (a function which yields a *Boolean*) and returns a *Seq[A]* which may be shorter (but not longer) than *this*.

Still more methods on these aggregate types

- *val as: Seq[A]*
- What about other types of result (different from *A*)?
  - *val bs: Seq[B] = as* method*[B] f* where *f: A=>B*?
    - This method is *map*, one of the most important methods.
    - We don’t normally need to explicitly state the [B] after the method name*.*
  - *val bs: Seq[B] = as* method*[B] f* where *f: A=>Seq[B]*?
    - This method is *flatMap*, one of the most important methods.
    - We don’t normally need to explicitly state the [B] after the method name*.*
  - *val xs: Option[A] = as* method *predicate ?*
    - This method is *find* which takes a predicate (a function which yields a *Boolean*) and returns an *Option[A]* which is *Some(x)* if found, else *None*.