## Data types and data structures

A *data type* (also called *abstract data type* or *ADT*) defines the operations and the behavior supported by an object. A data type is a *concept*, similar to mathematical concepts such as function, set, or sequence. The most important data types we will meet in this course are *stack*, *queue*, *set*, *priority queue*, and *map*.

A *data structure* is an implementation of a data type: An object that provides all the operations defined by the data type, with the correct behavior.

Quite often, there are different possible implementations for the same data type. For instance, stacks can be implemented with arrays or with linked lists, sets can be implemented with search trees or with hash tables.

The purpose of CS206 is to make students familiar with the important data types, and to discuss the different techniques for implementing them. In practice, understanding the data types is the most important part (since you will usually just use an implementation from a library anyway)!

### Stacks

A stack is a data type based on the principle of *Last In First Out* (LIFO). Just think about a stack of books or dishes. A stack supports the following operations:

- Push something on the top.
- Look at the top.
- Pop something off the top of the stack.
- Check if the stack is empty.

In Scala, we can represent a data type as a *trait* (the English word "trait" means "personality", in the sense that an object that has a stack trait has a "stack personality"—it behaves like a stack). (In other languages the same feature is called *interface* or *abstract class*.)

A trait prescribes the methods that the data type possesses, and the types of their arguments and the result type. (There is no way to prescribe the correct *behavior* of these methods—to do so, you would have to use an algebraic specification language.)

```
trait Stack[T] {
  def push(el: T): Stack[T]
  def top: T
  def pop(): T
  def isEmpty: Boolean
}
```

Note that the trait contains names and argument types for each method, but there is no implementation for the methods. This makes sense since the trait represents the *data type* stack, which is an abstract concept independent of a particular implementation.

Since a `Stack` is a trait (data type), not a class (implementation), it is impossible to create objects of type `Stack`:

```
scala> val s = new Stack[Int]
error: trait Stack is abstract; cannot be instantiated
```

However, it is possible to have variables of type `Stack`:

```
scala> var s: Stack[Int] = null
s: Stack[Int] = null
```

What this means is that the variable `s` will refer to an object that implements the `Stack` data type.

This allows us to write functions that will work correctly with any kind of stack, no matter how the stack is implemented:

```
  def reverse(stack: Stack[Char], s: String) {
    for (ch <- s)
      stack.push(ch)
    while (!stack.isEmpty)
    print(stack.pop())
    println()
  }
```

The type of the `stack` parameter is `Stack[Char]`, to indicate that the function `reverse` will work correctly if you pass it *any* object that provides the personality of a stack: that is, it provides `push`, `pop`, and `isEmpty` methods.

### Creating stack objects

To use a stack, we need to create an object that actually implements the stack. One possibility is to use a data structure provided by the Scala library:

```
  val stack: Stack[Int] =
    new scala.collection.mutable.Stack[Int] with Stack[Int]
```

The Scala stack implementation provides the necessary methods `push`, `pop`, `top`, and `isEmpty`, so we can use the `with` keyword to indicate that we want to use the Scala stack implementation to implement our `Stack` trait. We can now call our `reverse` function as follows:

```
reverse(new scala.collection.mutable.Stack[Char] with Stack[Char], "Hello CS206")
```

### Implementing a stack

(This is covered in a later class.)

To implement a stack ourselves, we create a class that provides an implementation for all methods declared in the trait. In the example code you find `FixedStack`, which implements a stack using a fixed size array, `GrowStack`, which uses an array that grows automatically with the size of the stack, and `LinkedStack`, which uses small objects linked together to store the contents of the stack.

Each of these classes provides the four methods `push`, `pop`, `top`, and `isEmpty` required by the `Stack` trait. We also indicate in the class definition that the class implements the `Stack` trait using the `extends` keyword:

```
class LinkedStack[T] extends Stack[T] {
  ...
}
```

Using the `extends` keyword has the advantage that the Scala compiler will give an error message if we forget to provide one of the methods of the `Stack` trait. A second advantage is that we can use `LinkedStack` objects without the `with` keyword: Scala already knows that every `LinkedStack` implements the `Stack` trait.

So we can use `LinkedStack` as follows:

```
scala> val stack: Stack[Int] = new LinkedStack[Int]
stack: Stack[Int] = LinkedStack()
scala> stack.push(5)
res0: Stack[Int] = LinkedStack(5, )
scala> stack push 3 push 7
res1: Stack[Int] = LinkedStack(7, 3, 5, )
scala> val k = stack.top
k: Int = 7
scala> val p = stack.pop()
p: Int = 7
```

We can also use a `LinkedStack` to call our `reverse` function from above (note that `with` is not necessary):

```
  reverse(new LinkedStack[Char], "Hello CS206")
```