7 Abstracting over the Data Type

The goal of this lab is to understand how we can design a more general programs by defining the common behavior for structured data, such as lists, using parametrized data types.

Begin by downloading *lab7.zip* and building a project that contains all the files as well as the latest version of the *tester.jar*.

Your project should have the following files:

- *Book.java*
- *Song.java*
- *Image.java*
- *ILo.java*
- *Examples.java*

Run the project and make sure all tests passed.

A. The file *Examples.java* contains tests for the method `totalValue` in the classes that represent a list of items of the type `<T>`.

   If you un-comment the test method, the program breaks. Modify the classes `Book`, `Song`, `Image` so that the method `totalValue` works correctly for the classes that represent a list of items of the types `Book`, `Song`, `Image` and the tests pass.

B. We now want to design the method `makeString` for the classes that represent a list of items of the type `<T>` that produces a readable `String` representation of the data in the list.

   - Design a method `makeString` for each of the classes `Book`, `Song`, `Image` that produces a `String` representing all data in `this` instance of the class.
   - Define a common interface `MakeString<T>` that represents the `makeString` method for the objects of the type `<T>`.
   - Design the method `makeString` for the classes that represent a list of items of the type `<T>`.

     Test your methods on the lists of books, songs, and images, in the manner similar to that shown in the previous examples.

C. We would like to design the method `filter` for the lists of items. The method produces a list of all items that satisfy some predicate. We could use the following interface:

```
// a method to decide whether this item
// has the desired property
interface ISelectable<T>{
  // does this data item have the desired property?
  boolean pick();
}
```

Design the method `filter` that produces a list of all items in the list that satisfy this predicate. Test it by selecting all books that cost less than $25, all songs that play for more than 180 minutes, and all images with the jpeg file type.

D. This is getting very tedious and is not flexible enough. There is no easy way to change the way we select the desired items.

The `filter` function in Scheme had the following definition:

```
;; filter: (X -> Boolean) (Listof X) -> (Listof X)
;; to construct a list from all items in alox
;; for which p holds
(define (filter p alox) ...)
```

The Scheme `filter` function consumes a predicate `p`, a function that determines for every item in the list whether it should be included in the resulting list.

So, at different times we can supply a different predicate.

We try to do the same in Java. Start by defining the interface

```
// a method to decide whether the given item
// has the desired property
interface ISelector<T>{
  // does the given item have the desired property?
  boolean pick(T t);
}
```

This represents a method that does not depend on the class where the method is defined. It consumes an item and determines whether it satisfies the predicate.

Design the method `filter2` in the classes that represent the list of items of the type `<T>` that consumes an instance of the predicate of the type `ISelector<T>` and produces a list of items that satisfy the predicate.

The problem is that we do not know how to test this method. We need an instance of the class that implements the `ISelector` interface.

Define the following class:

```
// a method to decide whether the given book
// costs less than $20
class CheapBook implements ISelector<Book>{
  // does the given book cost less than $20?
  boolean pick(TBook b){
    return b.price < 20;
  }
}
```

It is indeed a strange class — it contains a method, but no data. Its only purpose is to define an object that can invoke the desired method. We call it *function object*.

In the `Examples` class make an instance of the class and use it as an argument for the tests for the method `filter2`.

We will extend this example further during the next couple of days.