# 8 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 *lab8.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.

   (a) 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.

   (b) Define an interface `MakeString<T>` that represents the `makeString` method for the objects of the type `<T>`. The method produces a `String` representation of the entire object, or of some part of the object.

(c) For each of the classes `Book`, `Song`, `Image` design a class that implements the `MakeString<T>` interface.The method `makeString` should produce a `String` representing all data in `this` instance of the class, or some part of it. For example, you may define a `String` that contains the book title and the author's name; the image title and its size, etc.

(d) Design the method `makeStrings` for the classes that represent a list of items of the type `<T>` that produces a list of `String`s, applying the`makeString` method in the given instance of the class that implements the `MakeString<T>` interface to every item in the list.

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 generalize the method `filter` we have seen earlier so that it works for an arbitrary lists of items. The method produces a list of all items that satisfy some predicate. We modify the `ISelect` interface so it can be applied to any type of data:

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

Design the method `filter` that produces a list of all items in the list (parametrized by the type `T` that satisfy the given predicate (an instance of a class that implements the `ISelect<T>` interface. 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. The `makeStrings` method consumed `this` list of items of the type `T` and produced a list of items of the type `String`.

Think of the *Scheme* function `map`. It consumes a list of the type `X`, a function of the type `X -> Y`, and produces a list of items of the type `Y`, applying the given function to every item in the list.

So, our `makeStrings` method is a `map` from lists of the type `T` (we used `Song`s, `Book`s, and `Image`s) to a list of items of the type `String`.

(a) Design the interface `ITransform<T, S>` that represents a method `transform` that converts the given item of the type `T` to an item of the type `S`. The interface will be parametrized over two (possibly different) data types, `T` and `S`.

(b) Design three classes that implement this interface as follows:

— from the type `Book` to the type `String` e.g. the book title

– from the type `Image` to the type `Integer`, e.g. the image size, or width, or height

– from the type `Song` to the type `Boolean`, e.g. by the given artist, or short song...)

Notice that we use the types `Integer` and `Boolean` for the primitive types. These are so called *wrapper classes* that allow us to define a primitive data type as if it were a regularly defined class. Java automatically converts the instances of these classes to their primitive values, and primitive values or data may be used anywhere the *wrapper class* type is required.

(c) Design the method `map` for the classes that represent a list of items of the type `T`. The method header will be:

```
// produce a list of type S from this list
// of items of the type T by applying
// the given function to every item in this list
ILo<S> map(ITransform<T, S> transform);
```

**Note:** Finish this lab and include your work in your portfolio.