5 Content vs. Structure Equality

In this lab we explore the structural and behavioral equality of more complex data. Our examples will be based on binary search trees.

When are two binary search trees the same? First of all, they must be not only handling the same kind of data, but also use the same strategy for determining the ordering of two items. So, first we need to make sure that we can compare the ways the ordering is defined for two binary search trees. Next, we consider two ways of defining the equality — based on the structure of the tree, or based on the contents - whether they have the same number of items and for every item in one tree there is a matching item in the other one (including the same multiplicity, if applicable). Of course, we may require that the two trees have the same identical data (using the intensional equality for comparing the data in the nodes), or are completely identical (in which case the Java `equals` method does the work for us.

Start a new project *BSTs*. Download the file **BinarySearchTrees.zip**, unzip it, and import all files into your project. Add the *tester.jar* to the class path for your project and run the project.

The project deals with binary search trees of books, and with lists of books. We provided many examples and tests, so you can get started quickly. All but one of the provided tests succeed - we will talk about the failed one soon. Spend a bit of time to get familiar with the code, the examples of data, and the tests.

Function Objects

We start by looking at the interface `ICompareBooks` and the two classes `BookOrderByAuthor` and `BookOrderByTitleLength`, that implement this interface.

There is no natural ordering for books. We may want to order them by author's name, by title, by the year of publication, and we have also chosen the length of the book's title. When we insert an item into a binary search tree (or into an ordered list) we need to know how the ordering is determined. We need a *function/method* that allows us to compare two books and determine whether the first book comes before the second, or not. So the `insert` method for a binary search tree needs as one of its arguments a *function/method*. Alternately, and this is a better solution, the binary search tree should know what is the ordering *function/method*, i.e., one of its fields should record this information.

In some languages one can pass a *function/method* as an argument to other *function/method*. In Java this is not possible and the programmer is forced to use a complicated workaround:

- Design an interface that contains a method with the desired signature. In our case we have defined the interface `ICompareBooks`:

```
// interface to represent a method compare for books
public interface ICompareBooks{

  // does b1 come before b2 in this ordering?
  public boolean compare(Book b1, Book b2);
}
```

- Next we define a class that implements this interface. The class has no fields, and only defines one method:

```
// compare books by their author
public class BookOrderByAuthor implements ICompareBooks{

  // does b1 come before b2 when sorted by author?
  public boolean compare(Book b1, Book b2){
    return b1.author.compareTo(b2.author) <= 0;
  }
}
```

  We need one class to represent each ordering.

- The classes that represent the binary search tree can now contain a field of the type `ICompareBooks`, and when needed one can invoke its `compare` method to verify the ordering and handle the insertion or lookup of an item.

We forgot to test that the method`compare` in the classes `BookOrderByAuthor` and `BookOrderByTitleLength` works correctly. Add these tests to the test suite.

Note: If you have not worked with the TeachScheme! (How to Design Programs) before, think of how you would implement a simple `sort` method for the lists of books, then look at the actual definition we gave you. *Hint: It is very easy if you follow the Design Recipe carefully and forget everything that you ever knew about sorting.*

Understanding Function Objects

In Java, the name for the objects that only represent a method specified by some interface is *function objects*. How do we know that two instances of the type `ICompareBooks` are the same?

There are several things to consider. If the method produced an integer (< 0 if one is before two, 0 if one is the same as two, and > 0 if one is after two), we could have one class that compares the books by length, and another class that extends it, that tries to resolve the ties by comparing the year of publication. An instance of the second class is also and instance of the first class, and so the comparison that only checks if both *function objects* are instances of the same class could provide incorrect result.

A class that represents a function object is a *singleton*. We never need more than one instance of this class. If our program guarantees that there will never be more than one instance of this class, the Java `equals` method works correctly in checking the identity of two function objects.

In our course, at this point in time, we explain these issues, but do not go into the details how one implements the **Singleton Pattern**. You may want to look up the details in Joshua Bloch's Effective Java book.

In the `Examples` class, we define one instance of each *function object* and reuse it in all examples of data and all tests.

Note 1: One place where function objects are used frequently is in defining the action the program should when the user makes some selection in a GUI (clicks on a button, selects an option, clicks the mouse in some selected region, etc.).

Note2: The classes that implement some specific functional interface are sometimes defined as `inner classes`. For examples we may want to define within the `Book` class all classes that deal with ordering of books. We can further add a static field that defined the only instance of this class to the outside world, hiding the constructor from the outside user. *While these are important issues, we feel they are appropriate for a more seasoned programmer, and do not need to be covered in an introductory course.*

Constructor Integrity

We now want to make sure that as we build a binary search tree, the method for comparing two items remains the same.

We have added the field `ICompareBooks comp` to the abstract tree. The `insert` method uses the method `comp.compare` to determine where to insert the new item.

Of course, the method needs to construct a new tree and so it uses the constructor

```
NodeBook(Book data, ABSTBook left, ABSTBook right)
```

Notice that we do not provide a new instance of `comp`. The constructor verifies that the `comp` object in the left subtree is the same as the `comp` object in the right subtree.

If every new tree is constructed only by invoking the `insert` method on an existing tree, the constructor in this form would be sufficient. However, if this constructor is defined as `public` we need more safeguards to assure that the resulting tree will indeed be a binary search tree. The comment in the constructor suggests what needs to be done. We will work on it in a later part of this lab. For now, think how you would do it, but leave the programming for later.

Structural Equality

The classes that define the binary search tree already include the methods `sameTree`, `sameLeaf`, and `sameNode`, design in the manner similar to what we have learned about designing the equality comparison for unions.

There is nothing new here - we just wanted to make sure ou see another example of the use of equality of unions.

Read the code, look at the tests.

Behavioral Equality

Next we would like to design a method that determines whether two binary search trees represent the same ordered data set. The task is really not that hard: we need to verify that the first item in the first tree matches the first item in the other tree, and that the rest of the first tree matches the rest of the other tree as well.

This leads us to defining the following methods:

```
// produce the smallest item in this tree
  abstract public Book getFirst();

  // produce the biggest item in this tree
  abstract public Book getLast();

  // produce a binary search tree with the first node removed
  abstract public ABSTBook getRest();
  // is this an empty tree? (a leaf?)
  abstract public boolean isLeaf();
```

We added the method `getLast` as it will be helpful in completing the design of the constructor for the `NodeBook` class.

1. Design the four methods specified above. We have already included tests for them in the `ExamplesBooks` class.

2. Complete the design of the constructor for the `NodeBook` class, so that it throws an exception if the resulting tree would not be a binary search tree. You should make sure that the test at the end of the `ExamplesBooks` class succeeds.

3. Design the method `sameData` that verifies that `this` binary search tree contains the same data as the given binary search tree.

4. With all these tools in place, it is now easy to design a method `buildList` that produces a list (an instance of `ILoB`) from this binary search tree.

5. The classes that represent a list of books include a method that inserts all elements of this list into the given binary search tree. They also include a *commented-out* method `bstsort`. Look at what it does — it produces a sorted list by first inserting all items in this list into a binary search tree, then removes them one by one and builds a sorted list.

   Un-comment this method and run it.

Traversal, Iterable, Equivalence

The *tester* library provides a special way with dealing with structural equality, and supporting more than one definition of equality for the given class of data.

The ISame interface

If the programmer want make sure that all tests for the given class are handled in a special way, he may implement the `ISame` interface:

```
interface ISame<T>{
  // is this object the same as the given one?
  boolean same(T that);
}
```

The *tester* library will then use this method for comparing any two objects in this class, even when they represent fields in some other containing class.

The Traversal interface

The *tester* library defines the `Traversal` interface that represents a *functional iterator* over a collection of data. The interface is defined as follows:

```
interface Traversal<T>{
  // Produce true if this Traversal
  // represents an empty dataset
  boolean isEmpty();

  // Produce the first element in the dataset
  // represented by this Traversal
  T getFirst();

  // Produce a Traversal
  // for the rest of the dataset
  Traversal<T>getRest();
}
```

This is very similar to what we just did for the binary search trees. If the programmer's class implements the `Traversal` interface, then the objects in this class can be compared in the usual manner using the `checkExpect` or the `checkInexact` method, or they can be compared in the sequence generated by the `Traversal` interface, using the `checkIterable` or the `checkInexactIterable` methods.

The Iterable interface

The Java Collections Framework library defines the `Iterable` interface that generates a destructive mutating *Iterator* over a collection of data. The interfaces `Iterable` and `Iterator` are defined as follows:

```
interface Iterable<T>{
  // Produce an Iterator for this dataset
 Iterator iterator();
}
```

```
interface Iterator<T>{
  // Produce true if this Iterator
  // represents an empty dataset
  boolean hasNext();

  // Produce the first element in this dataset
  // Effect: Advance this iterator
  // to refer to the next element of this dataset
  T next();
}
```

The *tester* library handles classes that implement the `Iterable` interface in two ways. Because all classes in the Java Collections Framework that implement the `java.lang.Iterable` interface contain no additional relevant data, the `checkExpect` and the `checkInexact` methods use the provided `Iterator` to traverse over the two collections that are being compared, matching them in the order generated by the iterator.

However, if the user implements the `java.lang.Iterable` interface for her own class, the `checkExpect` (or the `checkInexact`) method still examines the data within the class by comparing the values of the corresponding fields. To invoke the comparison of the data generated by the iterator, the user needs to use the `checkIterable` (or the `checkInexactIterable`) method.

The Equivalence interface

The *tester* library allows the user to check whether the two given objects are equivalent according to the user-defined function object that implements the `Equivalence` interface.

The `Equivalence` interface is defined as:

```
public interface Equivalence<T>{
  // is this object equivalent to the given one?
  public boolean equivalent(T t1, T t2);
}
```

It is assumed, but not required, that the implementation of the `Equivalence` interface represents a true equivalence relation, i.e. is reflexive, symmetric and transitive.

The following example illustrates the use of the `checkEquivalent` tests:

For the class that represents a `Book` with an `Author` we define two books to be equivalent if their author has the same name:

```
public class EquivBooks implements Equivalence<Book>{

  // two books are equivalent if the author's names match
  public boolean equivalent(Book b1, Book b2){
    return b1.author.name.equals(b2.author.name);
 }
}
```

We now run the following tests:

```
// sample books
public Author author1=new Author("author1",40);
public Author author2=new Author("author2",55);
public Author author3=new Author("author1",66);
public Author author4=new Author("author4",26);

// sample authors
public Book book1=new Book("title1",author1,4000);
public Book book2=new Book("title2",author2,4000);
public Book book3=new Book("title3",author3,3001);
public Book book4=new Book("title1",author4,4000);

// instance of the Equivalence
Equivalence equbooks = new EquivBooks();

void testEquivalence(Tester t){
  t.checkEquivalent(this.book1, this.book2, equbooks,
    "fails: different books, authors");
  t.checkEquivalent(this.book1, this.book3, equbooks,
    "should succeed - same author names");
  t.checkEquivalent(this.book1, this.book4, equbooks,
    "fails: different authors.");
  t.checkEquivalent(this.book1, this.book3, equbooks); // no testname
}
```