Hashing

Last updated: November 8th 2018, at 5.03pm

1 BlueJ

Please note: you should use *BlueJ* for the logbook exercise this week, rather than IntelliJ, as BlueJ's workbench makes inspection of objects easier.

You should download the `week08BJ.jar` file from Brightspace. Launch BlueJ. Select **Project→Open Project...** — see figure 1 — and open `week08BJ.jar`. The project only contains one class, `HashtableWrapper`, which, as the name suggests, is simply a wrapper for Java's own `Hashtable` class. You should now construct a `HashtableWrapper` object. To do this, right click on the `HashtableWrapper` class icon and select `new HashtableWrapper<S,T>(int size)` — see figure 2.

A pop-up window will appear, asking for the parameters for the `HashtableWrapper` constructor. The first field is for the name of the object. This is not important, and can be left as it is. The next two are the type parameters for the generic types. Type `S` is the key type, and we will be using `String`s, so you should enter `String`. The second type parameter is the type of value that we will be using. Enter `Integer` here. The final parameter determines the initial size of the hashtable. Please use `5`. See figure 3. Clicking **OK** will create a `HashtableWrapper` object, and place it on BlueJ's workbench.

You can inspect this object by right clicking on it, and selecting **Inspect**, as shown in figure 4, or by simply double clicking on the object. This brings up an inspector window, which allows you to inspect all fields of the `hashtableWrapper` object, whether they are **public** or **private**. You can see an example in figure 5. You can also inspect any of these fields in turn, if they have some further structure — for example the `HashtableWrapper`'s internal array, as shown in figure 6.

We are now ready to add a key/value pair to the hashtable. Right click on the `HashtableWrapper` object on the workbench, and select `Integer put(String key,Integer value)`, as shown in figure 7. Another pop-up window appears, asking for the key value pair. Enter these (figure 8). A "Method Result" window will pop up, as shown in figure 9. You can just close this.

If you now inspect the `HashtableWrapper` object again you should be able to see the changes that adding the key value pair have made to the object — see figure 10.

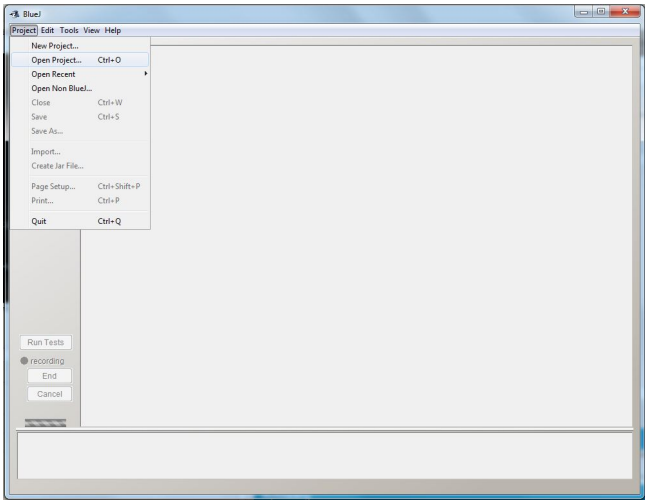

Figure 1: Opening a project in BlueJ

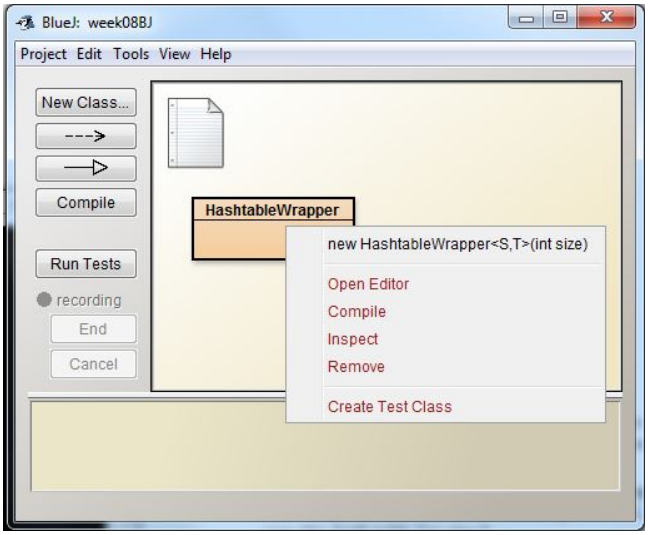

Figure 2: Creating an object in BlueJ

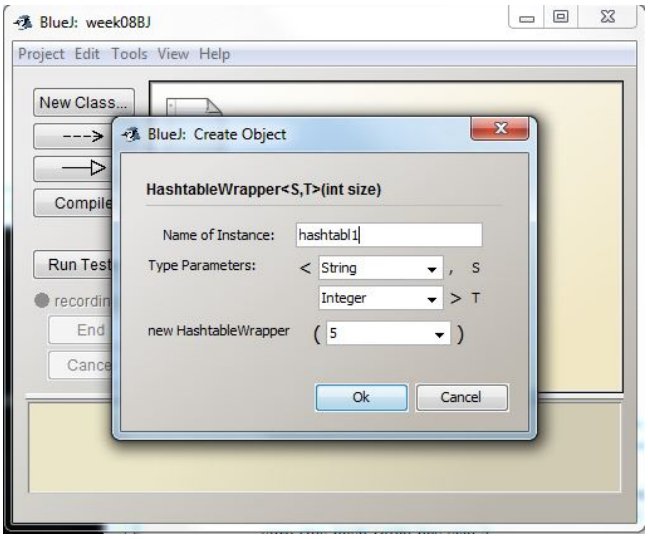

Figure 3: Setting a constructor's parameters in BlueJ

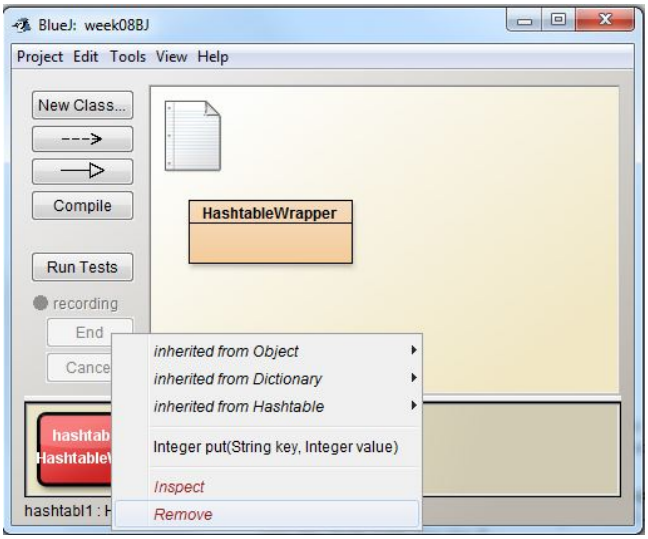

Figure 4: Opening an object inspector in BlueJ

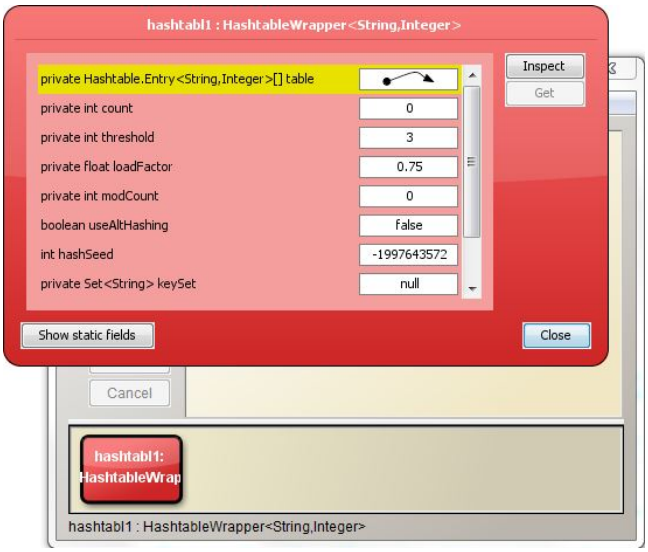

Figure 5: A Bluej object inspector

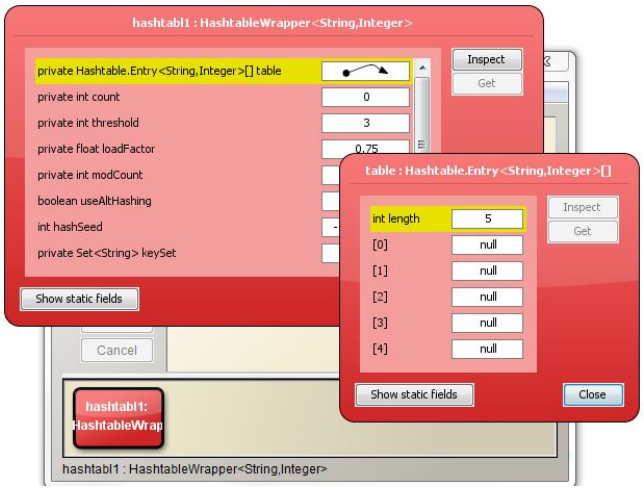

Figure 6: A BlueJ object inspector showing a secondary inspector

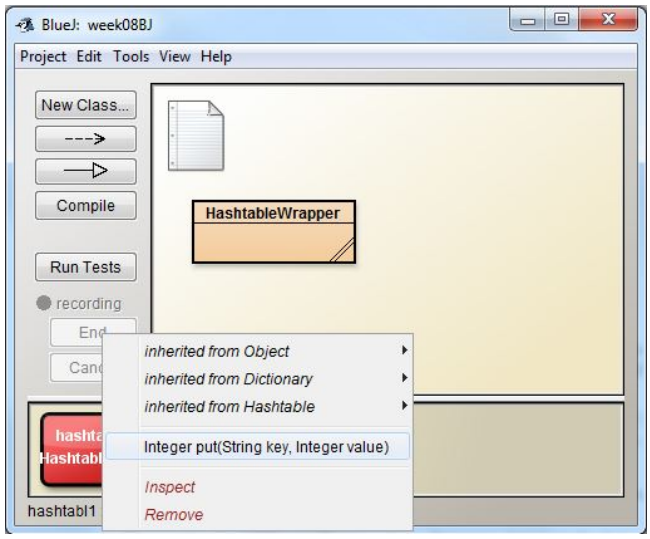

Figure 7: Calling an object's function in BlueJ

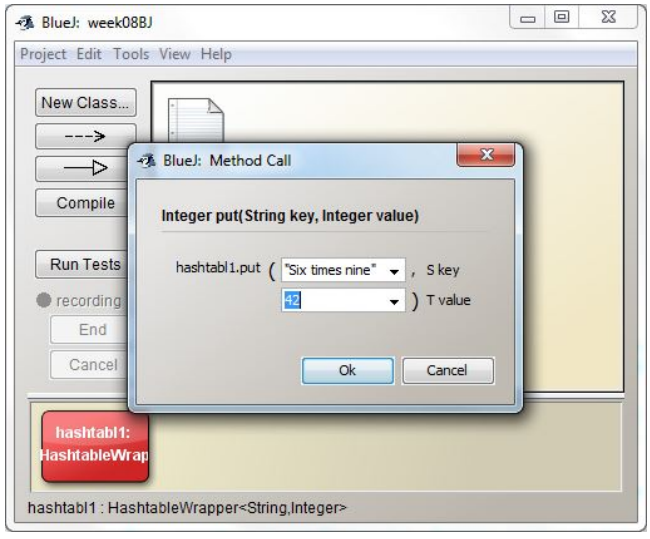

Figure 8: Entering a method's parameters in BlueJ

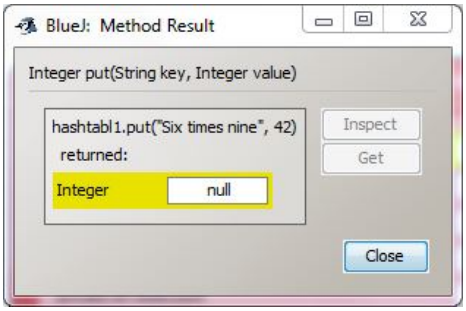

Figure 9: A BlueJ result window

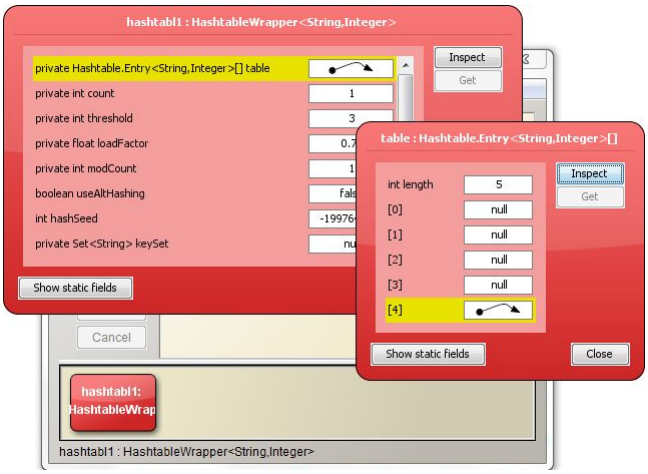

Figure 10: The `HashtableWrapper` object after a key/value pair is added

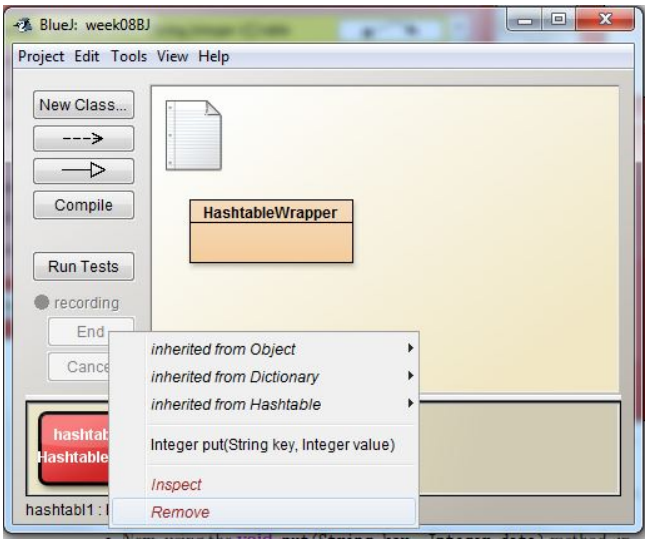

Figure 11: Removing an object from BlueJ's workbencg

Now remove the `Hashtable` wrapper object, by right clicking on the object and selecting remove, as shown in figure 11. You are now ready to start the logbook exercise.

2 Exercises

1. **Logbook exercise.** Create an object instance of the `HashtableWrapper⟨String,Integer⟩` class as described above, and as shown in figure 2. Ensure this hash table uses `String`s as keys, `Integer`s as values, and has an initial size of 5, as shown in figure 3.

   - Inspect the object you have just created, paying particular attention to the object's internal array.
   - Now, using the `void put(String key, Integer data)` method, as shown in figure 7, add the key/value pair (`"fred"`,37) to the hashtable (`"fred"` is the key, 37 is the value). Inspect the object again.
   - Now add the following key/value pairs, again inspecting the hashtable object after each new pair is entered:
     - (`"is"`,69)
     - (`"dead"`,0)
     - (`"but"`,999)
     - (`"not"`,-42)
     - (`"me!"`,-1)

- Describe, and *explain* what happens.

Note: This is not a programming exercise. Your logbook should contain an explanation for the observed behaviour of the hash table when these values are added. You may also like to have a look at the Java `Hashtable` API, and consider how this behaviour might be changed by using a different constructor call than that used in this example. You could also think about what the advantages and disadvantages of such differences might be.

The code bundle for the remaining coding exercises is in `week8.zip`, and was developed using IntelliJ, though you may, of course, if you wish use some other development environment.

2. Extend the abstract class `FillingHashtable` to a full implementation (do not edit `FillingHashtable`, but create a new subclass).

   The method

   ```
   fill(int noElements,
      RandomGenerator<S> keyGenerator,
      RandomGenerator<T> valueGenerator)
   ```

   populates the given hash table with `noElements` elements. The two `RandomGenerators` are used to generate the keys and values.

3. Use your question 2 code to answer the following questions:

   (a) Create a hash table with 10 elements in it and then add in a random integer value (in the range -42 to 42) against a random word. Print out your hash table.

   (b) Create a hash table with 10 elements in it and then add in a random integer value (in the range -42 to 42) against a random word, 5 times. Print out your hash table.

   (c) Create a hash table with 10 elements in it and then add in a random integer value (in the range -42 to 42) against a random word, 11 times. Print out your hash table.

   (d) What happens if you add in different values against the same key?

4. Which of the following hash functions best avoids address collisions in a hash table with 100 elements (ensure you give reasons for your answer):

   ```
   public int hash1(String key) {
      return (int)(Math.round(Math.random() * 100));
   }

   public int hash2(String key) {
   ```

```
    return key.length() % 100;
}

public int hash3(String key) {
    int midpnt = key.length() / 2;
    return (key.charAt(0) + key.charAt(midpnt) + key.charAt(key.length()));
}

public int hash4(String key) {
    int midpnt = key.length() / 2;
    return ((key.charAt(0) + key.charAt(midpnt) + key.charAt(key.length())) % 100);
}
```

5. The class `HuddersfieldHashtable` defines a generic interface for hash tables. The primary choice in impelementing this interface is whether to use open addressing or chaining. The two abstract classes `OpenAddressingHashtable` and `ChainingHashtable` define the relevant hash table datastructures for each of these choices.

   In answering the following question note that all Java objects inherit the method `int hashCode()` from their parent class `Object`.

   (a) **Model exercise.** Extend the class `OpenAddressingHashtable` so that it is an almost complete implementation of the `HuddersfieldHashtable` interface.

   Do not fully implement the interface — in particular the abstract method `probe(S key)` will be implemented twice, in two separate classes, implementing linear and quadratic probing.

   (b) Extend the class created in question 5a (in a new class) so that address collisions are resolved using linear probing.

   (c) Extend the class created in question 5a (in a new class) so that address collisions are resolved using quadratic probing.

   (d) Provide a new class that extends the `ChainingHashtable` class to implement the `HashtableInterface` interface. Your implementation should use the techniques of chaining.

   `void insert(S key, T data)` should store the given `data` in the array at the position given by the key's hash code (taken modulo the table's size).

   `T retrieve(S key)` should search your hash table looking for an entry with the given key. Should such an entry not be found, then a `HuddersfieldHashtable.Error` exception hould be thrown.

End of hash tables tutorial