# Homework #3 (v1.1)
# due Monday, February 11, 10:00 PM

In this homework, you will use a collection class with an interface based on Java's library collection system in a simple HexTile file GUI editor using Java AWT/Swing.

## 1 Concepts

In this week, you will work with several concepts that don't have their own chapter in the textbook.

### 1.1 Interfaces

A Java "interface" is a special kind of "abstract class." A Java *interface* specifies a set of (public) operations ("methods" in Java) that an ADT might be expected to implement. An ADT implementation signals its adherence to the interface by adding "`implements` *I*" to the class header, where *I* is the name of the interface.

A Java variable (field, parameter or local) may have an interface as a type. Of course the variable is *not* an instance of this type, since interfaces are abstract—they don't provide any implementation. Instead they are instances of some class that implements the interface. In Java, it is considered better to use interfaces for the type of variables or method returns instead of concrete classes since it makes the program more general.

### 1.2 Generics

Many of the library classes are *generic* so that they work with different reference types (e.g., classes). We've already seen that with `List`. In this homework, we will work with several generic interfaces and classes. If you fail to give the actual generic type parameter (inside angle brackets, e.g., `List<String>`), Eclipse will warn you that you are using "raw types." Raw types are not allowed for CS 351 homeworks, so make sure to take heed of the warnings.

### 1.3 Collections

The Java standard collection framework includes an interface `Collection` with a number of methods. See the textbook Figure 5.7, page 301 (3rd. ed. Fig. 5.6, p. 290), or look online. We have an implementation of this interface in class `edu.uwm.cs351.util.ArrayCollection<E>`. This class is a stripped down version of the Java library class `java.util.ArrayList`. For this Homework, **do not** use the standard Java library `ArrayList`.

### 1.4 Iterators

The collections classes and others provide iterators, which are more powerful than the "cursors" we implemented in the previous homework assignment. The big advantages are (1) that one can have multiple iterators (whereas there's just one cursor for each Sequence) and (2) then changing the state of an iterator (e.g., advancing it) does not affect the container. But since iterators are separate objects from the collections, if the collection changes, the iterator can go "stale." Implementations try to prevent the usage of stale iterators, throwing an instance of `ConcurrentModificationException` but the checks are not foolproof.

Iterators have three methods:

**hasNext()** Return true if there still remain elements to be returned.

**next()** Advances the iterator and returns the next (now current) element in the container. If there is no such element (in which case `hasNext()` should have returned false), then throw an instance of `NoSuchElementException`.

**remove()** Remove the element *previously* returned by `next()`. If `next()` has not yet been called, or if the element has already been removed, then this method throws an `IllegalStateException`.

Each iterator moves separately through the container.

The `Iterator` interface is generic with the type of the elements in the collection. To access all the elements of a collection, and decide whether to delete them, one can write:

```
for (Iterator<T> it = c.iterator(); it.hasNext();) {
   T x = it.next();
   if (we don't want element x any more) it.remove();
}
```

Java has a special syntax of "for" loops to make it easy to use iterators. The shortcut:

```
for (T x : c) {
   ...
}
```

is short for

```
for (Iterator<T> _secret = c.iterator(); _secret.hasNext();) {
   final T x = _secret.next();
   ...
}
```

### 1.5 Fail-Fast Iterators

When using standard Java collections, iterators become "stale" if the collection changes, except by using the iterator's own `remove` method. It is not legal to use a stale iterator for anything, even calling `hasNext`. In other words, if you request an iterator and later add an element to the collection, then you are not allowed to use the iterator again. If you want an iterator, you must request a new one.

Implementors of Java's standard collections are encouraged to provide *fail fast* implementations of iterators which "usually" throw an exception (an instance of ConcurrentModificationException) when a stale iterator is used. A "fail fast" implementation is not an absolute requirement.

The `ArrayCollection` CS 351 library class has this feature.

### 1.6 Event Handlers

With Java's graphics model, events are reacted to, rather than detected. In other words, rather than repeatedly asking "Did someone click my window?" a Java application program will say "when someone clicks my window, let me know." The application program will call `add`$X$`Listener` on the window or applet from which it wants to receive events, where $X$ is the kind of event. Then the general event system will call a method when the window is clicked. The object to be notified must implement the required interface.

### 1.7 Anonymous classes

This week, we will work with the "action listener" and "mouse listener" interfaces. The former is simpler:

```
public interface ActionListener {
  public void actionPerformed(ActionEvent e);
}
```

One could define a class that implemented this interface and then create an instance of it:

```
public class MyActionListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // do something
  }
}
...
        new MyActionListener()
```

But frequently, we don't really care what the name of the class is, we only want to make an instance of it. In that case we can create an instance of an *anonymous class* using the following syntax:

```
new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    // do something
  }
}
```

This *looks* as if we are creating an instance of the interface, but actually we are creating an instance of a class that is never given a (visible) name. The anonymous class implements the interface and defines the method `actionPerformed`.

This syntax can be used to create instances of anonymous subclasses of normal classes too. For instance, the class `MouseAdapter` implements the `MouseListener` interface with methods that do nothing. One can make an instance that does something different for one or more of the methods by using the anonymous class syntax:

```
this.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) { ... }
});
```

**NB**: To distinguish between a double-click and a single-click, use `getClickCount` on the mouse event object. A double click is always preceded by a single click in the same location.
**NB:** The solution uses anonymous classes for the mouse listener and for the action listeners on the terrain buttons.

In Java 8, one can use "lambda" expression syntax for these anonymous classes if there is only a single method to implement. For this Homework, you may use named classes, anonymous classes or lambda expressions as you see fit.

## 3 Conversions from String

In our first week, we implemented `toString()` methods for getting a string version of the ADTs. The reverse methods enables ADT values to be written to text files and then read back in. For this homework, you need to implement `fromString` methods for `HexCoordinate` and `HexTile`. These methods should throw an instance of `FormatException` if an error in the string is detected. In particular, a `NumberFormatException` must be caught and wrapped into a `FormatException` (there is a constructor for this purpose) which is then thrown. We provide test cases for these conversions.

## 4 Concerning the `HexBoardEditor` program

You will need to complete a `HexBoardEditor` class that creates windows with buttons for creating hex grids. The parts you need to write are (not in program order!):

1. At initialization, add each tile read from the file into the collection. Test this using `test/sample.hex` as a "Program Argument."
2. The window should display all the tiles in the board. The display should be updated when a new location is selected, the size is changed (done for you) or a tile is added or removed.
3. There should be an indication of the current terrain type. This should have background color set to the terrain color.
4. The window should have one button for each kind of terrain; this button should be colored with the terrain color and labeled with the terrain name.
5. If such a button is pressed, this should set the terrain to be used when adding new hex tiles to the grid.
6. If someone clicks in the hex grid area, the hexagon location being clicked should be outlined in magenta.
7. If someone double-clicks on the hex grid area, then if there is no tile there already, a new tile of the currently selected terrain should be created there. If the selected terrain is `INACCESSIBLE` then delete any existing tile, otherwise, replace any existing tile with a new tile of the terrain.
8. Print out all tiles when the application is closed.

The user interface could be better, but we don't want to make the homework too long, and this isn't a user interface or advanced programming class.

## 5 Files

In the starting repository `homework3.git`, we provide the skeleton file for `HexBoardEditor.java` and also the `HexCoordinate` and `HexTile` classes from Homework #1, which need to be changed (to add `fromString` methods). Don't change anything else in these classes!