JavaFX
Using JavaFX Collections
Release 8
E50649-01

March 2014
Learn about the concept of collections as used in JavaFX.
Contents

1 Using JavaFX Collections
   Reviewing Java Collections Basics ................................................................. 1-1
   Learning JavaFX Collections ........................................................................... 1-6
Preface

This preface describes the document accessibility features and conventions used in this tutorial - *Using JavaFX Collections*.

**Audience**

This document is intended for JavaFX developers.

**Documentation Accessibility**


**Access to Oracle Support**


**Related Documents**

For more information, see the following documents in the JavaFX documentation set:

- *Getting Started with JavaFX*

**Conventions**

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><strong>italic</strong></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><strong>monospace</strong></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
Using JavaFX Collections

This tutorial describes the JavaFX Collections API — an extension of the Java Collections Framework — providing code samples that you can compile and run.

This tutorial begins with a short review of the relevant classes and interfaces from the Java Collections Framework, then explains how the JavaFX Collections API extends them to provide additional behavior. For an in-depth tutorial on Java Collections, see the Collections trail of the Java Tutorial.

Reviewing Java Collections Basics

This section summarizes the `java.util.List` and `java.util.Map` interfaces, and the `java.util.Collections` class. If you are already familiar with Java Collections, skip to the next section entitled Learning JavaFX Collections.

List

A List is an ordered collection of objects, represented by the `java.util.List` interface. The objects in a List are called its elements, and duplicate elements can exist in the same List. The List interface defines a number of useful methods, enabling you to add elements, access or change elements at a particular index, create sublists, search for an element within a list, clear a list, and more.

Example 1–1 demonstrates these methods with a List of String objects:

```
Example 1–1 Using a List

package collectionsdemo;

import java.util.List;
import java.util.ArrayList;

public class CollectionsDemo {

    public static void main(String[] args) {

        // Create a List.
        System.out.println("Creating the List...");
        List<String> list = new ArrayList<String>();
        list.add("String one");
        list.add("String two");
        list.add("String three");

        // Print out contents.
        printElements(list);
    }
}
```

// Set a new element at index 0.
System.out.println("Setting an element...");
list.set(0, "A new String");
printElements(list);

// Search for the newly added String.
System.out.println("Searching for content...");
System.out.println("Contains "A new String"? ");
System.out.println(list.contains("A new String"));
System.out.println("\n");

// Create a sublist.
System.out.println("Creating a sublist...");
list = list.subList(1,3);
printElements(list);

// Clear all elements.
System.out.println("Clearing all elements...");
list.clear();
printElements(list);
}

private static void printElements(List<String> list) {
    System.out.println("Size: "+list.size());
    for (Object o : list) {
        System.out.println(o.toString());
    }
    System.out.println("\n");
}

The output from Example 1-1 is as follows:

Creating the List...
Size: 3
String one
String two
String three

Setting an element...
Size: 3
A new String
String two
String three

Searching for content...
Contains "A new String"? true

Creating a sublist...
Size: 2
String two
String three

Clearing all elements...
Size: 0

This program first instantiates an ArrayList (a concrete implementation of the List interface) and assigns it to the list variable. Next, three String objects are added to the list by invoking its add method. (At various points throughout its execution, the program prints out the elements by invoking a custom private static method named printElements.) The line list.set(0,"A new String") replaces the original String object at the first index position with a new String object. The contains method reports whether or not the specified element is present in the List, and the sublist method returns a new List from the range specified by the given index values. Finally, the clear method removes all elements from the List.

Map
A Map is an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to only one value. You can put keys and values into a Map, then retrieve a value by passing in its key. For example, the key apple might return fruit, whereas carrot might return vegetable.

Example 1–2 demonstrates these methods with a Map of String objects:

Example 1–2 Using a Map
definitions/document/JavaFX_Collections/Chapter1-2/Listing1-2.java

public class CollectionsDemo {

    public static void main(String[] args) {

        // Create a Map.
        Map<String,String> map = new HashMap<String,String>();
        map.put("apple", "fruit");
        map.put("carrot", "vegetable");
        System.out.println("Size: "+map.size());
        System.out.println("Empty? "+map.isEmpty());

        // Pass in keys; print out values.
        System.out.println("Passing in keys and printing out values...");
        System.out.println("Key is apple, value is: "+map.get("apple");
        System.out.println("Key is carrot, value is: "+map.get("carrot");
        System.out.println("";

        // Check keys and values.
        System.out.println("Inspecting keys and values:");
        System.out.println("Contains key \"apple\"? "+map.containsKey("apple");
        System.out.println("Contains key \"carrot\"? "+map.containsKey("carrot");
        System.out.println("Contains key \"fruit\"? "+map.containsKey("fruit");
        System.out.println("Contains key \"vegetable\"? "+map.containsKey("vegetable");
        System.out.println("Contains value \"apple\"? "+map.containsValue("apple");
    }
}

Using JavaFX Collections 1-3
map.containsKey("apple");
System.out.println("Contains key "apple"? "+
map.containsKey("apple");
System.out.println("Contains key "carrot"? "+
map.containsKey("carrot");
System.out.println("Contains key "fruit"? "+
map.containsKey("fruit");
System.out.println("Contains key "vegetable"? "+
map.containsKey("vegetable");
System.out.println(" contiene din: ");

// Remove objects from the map.
System.out.println("Removing apple from the map...");
map.remove("apple");
System.out.println("Size: "+map.size());
System.out.println("Contains key "apple"? "+
map.containsKey("apple");
System.out.println("Invoking map.clear()");
map.clear();
System.out.println("Size: "+map.size());
}
}

The output of Example 1–2 is as follows:
Size: 2
Empty? false
Passing in keys and printing out values...
Key is apple, value is: fruit
Key is carrot, value is: vegetable

Inspecting keys and values:
Contains key "apple"? true
Contains key "carrot"? true
Contains key "fruit"? false
Contains key "vegetable"? false
Contains value "apple"? false
Contains value "carrot"? false
Contains value "fruit"? true
Contains value "vegetable"? true

Removing apple from the map...
Size: 1
Contains key "apple"? false
Invoking map.clear()...
Size: 0

This program first instantiates a HashMap (a concrete implementation of the Map interface) and assigns it to the map variable. Key-value pairs are then added to map by invoking its put method. The program then obtains (and prints out) some information about the map by invoking size() and isEmpty(). The program also demonstrates
how to obtain the value for a given key (for example, map.get("apple") returns the value fruit). The containsKey and containsValue methods demonstrate how to test if a particular key or value is present, and the clear method removes all of the key-value mappings.

**Collections**

In addition to the methods found in List and Map, the Collections class exposes a number of static utility methods that operate on or return collections. Example 1–3 demonstrates a few such methods by creating a List, then using the Collections class to reverse, swap, and sort its elements.

**Example 1–3 Using the Collections Class**

```java
package collectionsdemo;

import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class CollectionsDemo {

    public static void main(String[] args) {
        System.out.println("Creating the list...\n");
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");
        list.add("c");
        list.add("d");
        printElements(list);
        System.out.println("Reversing the elements...\n");
        Collections.reverse(list);
        printElements(list);
        System.out.println("Swapping the elements around...\n");
        Collections.swap(list, 0, 3);
        Collections.swap(list, 2, 0);
        printElements(list);
        System.out.println("Alphabetically sorting the elements...\n");
        Collections.sort(list);
        printElements(list);
    }

    private static void printElements(List<String> list) {
        for (Object o : list) {
            System.out.println(o.toString());
        }
    }
}
```

The output of Example 1–3 is as follows:

Creating the list...

a
b
c
d
Learning JavaFX Collections

This program first adds the letters a b c and d to a List, again using ArrayList as the concrete implementation.) It then reverses the elements of the list by invoking Collections.reverse(list). To swap the elements around within the List, the program invokes the Collections.swap method (For example, Collections.swap(list,0,3) swaps the elements at index positions 0 and 3. Finally, the Collections.sort() method alphabetically sorts the elements.

Having reviewed the most relevant areas of the Java Collections Framework, you are now ready to learn how Collections are represented in JavaFX.

Learning JavaFX Collections

Collections in JavaFX are defined by the javafx.collections package, which consists of the following interfaces and classes:

**Interfaces**
- **ObservableList:** A list that enables listeners to track changes when they occur
- **ListChangeListener:** An interface that receives notifications of changes to an ObservableList
- **ObservableMap:** A map that enables observers to track changes when they occur
- **MapChangeListener:** An interface that receives notifications of changes to an ObservableMap

**Classes**
- **FXCollections:** A utility class that consists of static methods that are one-to-one copies of java.util.Collections methods
- **ListChangeListener.Change:** Represents a change made to an ObservableList
- **MapChangeListener.Change:** Represents a change made to an ObservableMap

The following section explains how to use these interfaces and classes.
Using ObservableList, ObservableMap, and FXCollections

The `javafx.collections.ObservableList` and `javafx.collections.ObservableMap` interfaces both extend `javafx.beans.Observable` (and `java.util.List` or `java.util.Map`, respectively) to provide a List or Map that supports observability. If you look at the API specification for either of these interfaces, you will find methods for adding or removing the appropriate listeners (`ListChangeListener` for `ObservableList`, and `MapChangeListener` for `ObservableMap`). Unlike the previous List and Map examples — which used `ArrayList` and `HashMap` as their concrete implementations — Example 1–4 uses the `javafx.collections.FXCollections` class to create and return the `ObservableList` and `ObservableMap` objects.

**Example 1–4 Using an ObservableList**

```java
package collectionsdemo;
import java.util.List;
import java.util.ArrayList;
import javafx.collections.ObservableList;
import javafx.collections.ListChangeListener;
import javafx.collections.FXCollections;
public class CollectionsDemo {
    public static void main(String[] args) {
        // Use Java Collections to create the List.
        List<String> list = new ArrayList<String>();
        // Now add observability by wrapping it with ObservableList.
        ObservableList<String> observableList = FXCollections.observableList(list);
        observableList.addListener(new ListChangeListener() {
            @Override
            public void onChanged(ListChangeListener.Change change) {
                System.out.println("Detected a change! ");
            }
        });
        // Changes to the observableList WILL be reported.
        // This line will print out "Detected a change!"
        observableList.add("item one");
        // Changes to the underlying list will NOT be reported
        // Nothing will be printed as a result of the next line.
        list.add("item two");
        System.out.println("Size: "+observableList.size());
    }
}
```

In Example 1–4, a standard List is first created. It is then wrapped with an `ObservableList`, which is obtained by passing the list to `FXCollections.observableList(list)`. A `ListChangeListener` is then registered, and will receive notification whenever a change is made on the `ObservableList`. You can listen for changes on an `ObservableMap` in a similar manner, as shown in Example 1–5.
Example 1–5 Using ObservableMap

```java
package collectionsdemo;

import java.util.Map;
import java.util.HashMap;
import javafx.collections.ObservableMap;
import javafx.collections.MapChangeListener;
import javafx.collections.FXCollections;

public class CollectionsDemo {

    public static void main(String[] args) {

        // Use Java Collections to create the List.
        Map<String,String> map = new HashMap<String,String>();

        // Now add observability by wrapping it with ObservableList.
        ObservableMap<String,String> observableMap = FXCollections.observableMap(map);
        observableMap.addListener(new MapChangeListener() {
            @Override
            public void onChanged(MapChangeListener.Change change) {
                System.out.println("Detected a change! ");
            }
        });

        // Changes to the observableMap WILL be reported.
        observableMap.put("key 1","value 1");
        System.out.println("Size: "+observableMap.size());

        // Changes to the underlying map will NOT be reported.
        map.put("key 2","value 2");
        System.out.println("Size: "+observableMap.size());
    }
}
```

And finally, you can use the static utility methods from either Collections or FXCollections (for example, to reverse the elements of a list). Keep in mind, however, that the FXCollections class will yield the smallest number of change notifications (usually one) when its methods are invoked. Invoking Collections methods, on the other hand, might result in multiple change notifications, as shown in Example 1–6.

Example 1–6 Collections Vs. FXCollections Change Notifications

```java
package collectionsdemo;

import java.util.List;
import java.util.ArrayList;
import javafx.collections.ObservableList;
import javafx.collections.ListChangeListener;
import javafx.collections.FXCollections;

public class CollectionsDemo {

    public static void main(String[] args) {

        // Use Java Collections to create the List
        List<String> list = new ArrayList<String>();
        list.add("d");
        list.add("b");
        list.add("a");
    }
}
```
list.add("c");

// Now add observability by wrapping it with ObservableList
ObservableList<String> observableList = FXCollections.observableList(list);
observableList.addListener(new ListChangeListener() {
    @Override
    public void onChanged(ListChangeListener.Change change) {
        System.out.println("Detected a change! ");
    }
});

// Sort using FXCollections
FXCollections.sort(observableList);

In Example 1–6, the line FXCollections.sort(observableList) alphabetically sorts the String objects in the list, and prints only one change notification to the screen; but if you use Collections.sort(observableList), the change notification is printed four times.

When using a ListChangeListener or MapChangeListener, the onChanged method always contains an object that encapsulates information about the change. This is an instance of ListChangeListener.Change (for ObservableList) or MapChangeListener.Change (for ObservableMap). When working with ListChangeListener.Change, always wrap any calls to the change object in a loop that invokes change.next(). Example 1–7 provides a demonstration.

Example 1–7 Querying a ListChangeListener.Change Object

... // This code will work with any of the previous ObservableList examples observableList.addListener(new ListChangeListener() {
    @Override
    public void onChanged(ListChangeListener.Change change) {
        System.out.println("Detected a change! ");
        while (change.next()) {
            System.out.println("Was added? "+ change.wasAdded());
            System.out.println("Was removed? "+ change.wasRemoved());
            System.out.println("Was replaced? "+ change.wasReplaced());
            System.out.println("Was permutated? "+ change.wasPermutated());
        }
    }
});
...

Example 1–7 invokes various methods on the ListChangeListener.Change object. The most important point to remember is that a ListChangeListener.Change object can contain multiple changes, and therefore must be iterated by calling its next() method in a while loop. Note, however, that MapChangeListener.Change objects will only contain a change that represents the put or remove operation that was performed.

For information on available methods, see the ListChangeListener.Change and MapChangeListener.Change API documentation.