Convert Java Objects to String With the Iterator Pattern

By Dele Taylor

The visitor pattern often comes to mind when you need to operate on a graph of objects (like JSON, XML, or Java beans). Unfortunately, the visitor pattern uses call backs which are difficult to control from the calling code. For example, it's not easy to conditionally skip a branch with all its child branches and leaves from a callback. This how-to will instead use the iterator pattern to traverse your Java object graph and create a human readable debug string. The iterator will be general enough for you to use in other ways, as I did in building a tool to search Java objects using XPath or to record exceptions in StackHunter.

The APIs

This blog creates two separate tools for you to use: `StringGenerator` and `ObjectIterator` .

String Generator

The `StringGenerator` utility class converts your object graph to a string that's easy for us humans to read. You can use it to implement `toString` in your classes or just to log an object's complete graph when debugging (regardless of how their `toString` methods are implemented).

```java
public class StringGeneratorExample {

    public static void main(String[] args) {
        Department department = new Department(5775, "Sales")
        .setEmployees(
                new Employee(111, "Bill", "Gates"),
                new Employee(222, "Howard", "Schultz"),
                new Manager(333, "Jeff", "Bezos", 75000));

        System.out.println(StringGenerator.generate(department));
        System.out.println(StringGenerator.generate(new int[] { 111, 222, 333 }));
        System.out.println(StringGenerator.generate(true));
    }

}
```

The above code uses `StringGenerator.generate()` to convert a `department` , an array, and a `boolean` to the following formatted output.

```
com.stackhunter.example.employee.Department@129719f4
  deptId = 5775
  employeeList = java.util.ArrayList@7037717a
    employeeList[0] = com.stackhunter.example.employee.Employee@17a323c0
      firstName = Bill
      id = 111
      lastName = Gates
    employeeList[1] = com.stackhunter.example.employee.Employee@57801e5f
      firstName = Howard
      id = 222
      lastName = Schultz
    employeeList[2] = com.stackhunter.example.employee.Manager@1c4a1bda
      budget = 75000.0
      firstName = Jeff
      id = 333
      lastName = Bezos
  name = Sales
[I@39df3255
  object[0] = 111
  object[1] = 222
  object[2] = 333
true
```

Object Iterator

The `ObjectIterator` class uses the iterator patten to traverse the properties in your object (and all its children) as key-value pairs. It treats everything the same whether they're Java beans, collections, arrays, or maps. `ObjectIterator` also takes care not to follow cycles in your object's graph (see `StringGeneratorTest.testCircularGraph()` in the download).

```java
public class ObjectIteratorExample {

    public static void main(String[] args) {
        Department department = new Department(5775, "Sales")
        .setEmployees(
                new Employee(111, "Bill", "Gates"),
                new Employee(222, "Howard", "Schultz"),
                new Manager(333, "Jeff", "Bezos", 75000));

        ObjectIterator iterator = new ObjectIterator("some department", department)

        while (iterator.next()) {
            System.out.println(iterator.getName() + "=" + iterator.getValueAsString
        }
    }

}
```

The above code walks an object graph to produce a flat set of key-value pairs. It uses the `getValueAsString()` method to bypass each object's `toString()` implementation to produce a standard format. For primitive, boxed types, strings, dates, and enums, it's uses their original `toString()` implementation. For others, it's their class name and hashcode.

You can use the `ObjectIterator.getDepth()` method to add indents for easier reading (as done in the `StringGenerator.generate()` method). You can also use its `nextParent()` method before calling `next()` to short circuit the current branch of the tree and skip to the next. `StringGenerator` uses this to limit the number of children it outputs to 64.

```
some department=com.stackhunter.example.employee.Department@780324ff
deptId=5775
employeeList=java.util.ArrayList@6bd15108
employeeList[0]=com.stackhunter.example.employee.Employee@22a79c31
firstName=Bill
...
```

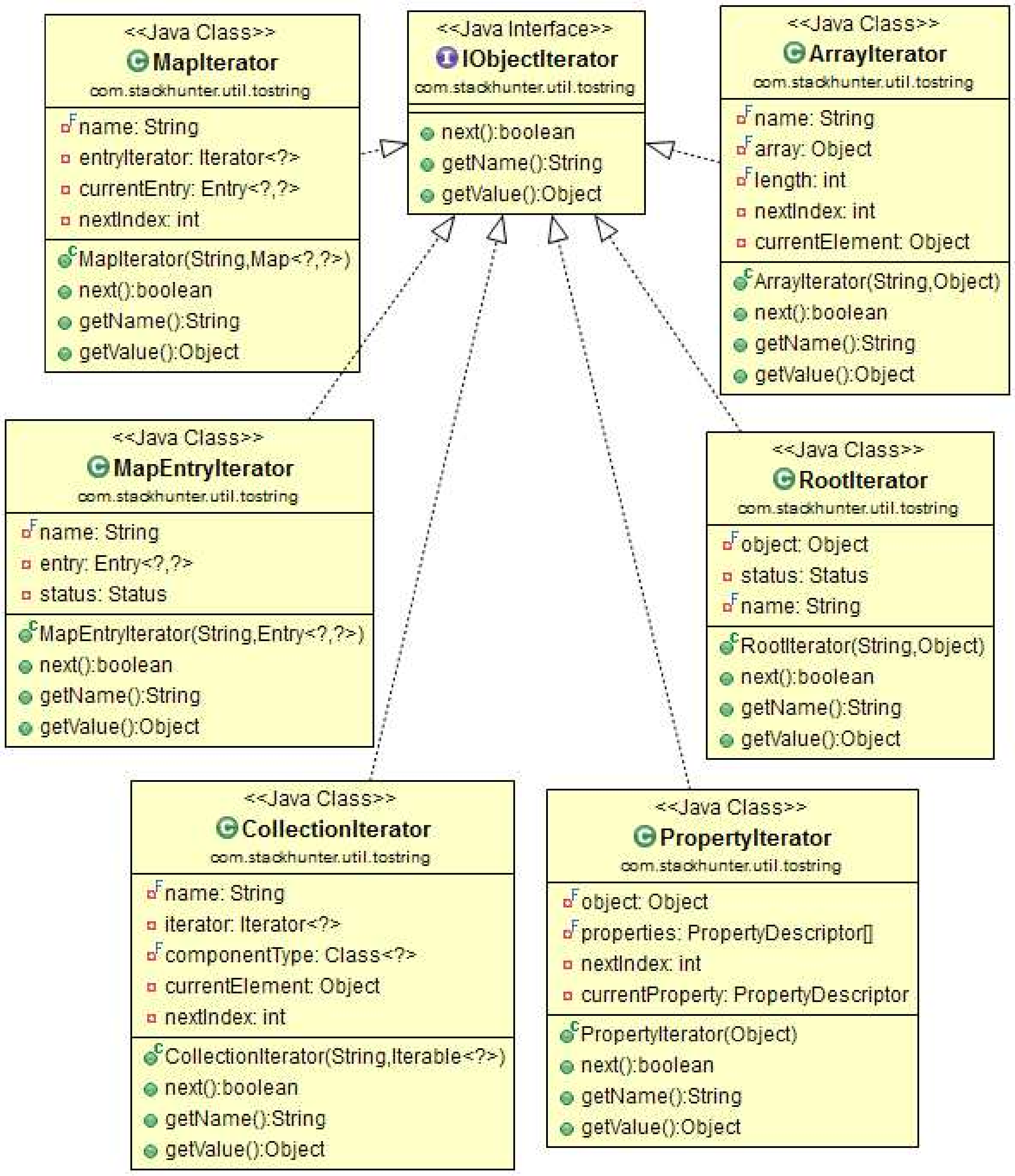
Implementing the Java Object Iterator

The first step when implementing the iterator pattern is to create a common, iterator interface: `IObjectIterator`. This interface will be used regardless of the actual type (Java bean, array, map, etc.) being traversed. (Sorry if the 'I' prefix offends you, blame the Eclipse platform folks.)

```
public interface IObjectIterator {
    boolean next();
    String getName();
    Object getValue();
}
```

The interface allows you to move in one direction — forward — and retrieve the current property's name and value along the way.

Each implementation of `IObjectIterator` is responsible for handling traversal of one type of object. Most take in a name prefix to use when answering their `getName()` call. In the case of `ArrayIterator`, it tacks on the element's index to its name: `return name + "[" + nextIndex + "]";`.



Property Iterator

`PropertyIterator` is probably the most important iterator class. It uses Java bean introspection to read the properties of an object to turn them into a sequence of key-value pairs.

```java
    public PropertyIterator(Object object) {
        this.object = object;
        try {
            BeanInfo beanInfo = Introspector.getBeanInfo(object.getClass());
            properties = beanInfo.getPropertyDescriptors();
        } catch (RuntimeException e) {
            throw e;
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage(), e);
        }
    }

    @Override
    public boolean next() {
        if (nextIndex + 1 >= properties.length) {
            return false;
        }

        nextIndex++;
        currentProperty = properties[nextIndex];
        if (currentProperty.getReadMethod() == null || "class".equals(currentProper
            return next();
        }
        return true;
    }
```

Array Iterator

The `ArrayIterator` uses reflection to determine the length of the array and to retrieve each of its elements in turn. `ArrayIterator` doesn’t need to worry about the details of the values returned from its `getValue()` method. It’s very likely they will either be be passed to a `PropertyIterator` somewhere down the line.

```
public class ArrayIterator implements IObjectIterator {

    private final String name;
    private final Object array;
    private final int length;
    private int nextIndex = -1;
    private Object currentElement;

    public ArrayIterator(String name, Object array) {
        this.name = name;
        this.array = array;
        this.length = Array.getLength(array);
    }

    @Override
    public boolean next() {
        if (nextIndex + 1 >= length) {
            return false;
        }

        nextIndex++;
        currentElement = Array.get(array, nextIndex);
        return true;
    }

    @Override
    public String getName() {
        return name + "[" + nextIndex + "]";
    }

    @Override
    public Object getValue() {
        return currentElement;
    }

}
```

Collection Iterator

The `CollectionIterator` is very similar to the `ArrayIterator` . It takes an `java.lang.Iterable` and calls its `Iterable.iterator()` method to initialize its internal iterator.

Map Iterator

The `MapIterator` traverses the entries in a `java.util.Map`. It does not actually delve into each entry's key-value pairs, that's the responsibility of the `MapEntryIterator` class.

```java
public class MapIterator implements IObjectIterator {

    private final String name;
    private Iterator<?> entryIterator;
    private Map.Entry<?, ?> currentEntry;
    private int nextIndex = -1;

    public MapIterator(String name, Map<?, ?> map) {
        this.name = name;
        this.entryIterator = map.entrySet().iterator();
    }

    @Override
    public boolean next() {
        if (entryIterator.hasNext()) {
            nextIndex++;
            currentEntry = (Entry<?, ?>) entryIterator.next();
            return true;
        }
        return false;
    }

    ...

}
```

Map Entry Iterator

The `MapEntryIterator` handle a single entry from a `java.util.Map`. It only ever returns two things: the entry's key, then its value. Like the `ArrayIterator` and others, its results may eventually be passed to a `PropertyIterator` and treated as Java beans if they are complex types.

Root Iterator

The `RootIterator` returns a single element — the initial node. Think of it as the root (or most outer) node in an XML document. Its purpose is to start things off.

Putting It All Together

The `ObjectIterator` class (used earlier) acts as a facade, wrapping all the traversal logic together. It determines which `IObjectIterator` subclass to instantiate based on the current type returned from the last `getValue()` call (see its `iteratorFor()` factory method). It preserves the current iterator's state on a stack when a new child iterator is created internally. It also exposes methods like `getChild()` and `getDepth()` to provide the caller with a picture of its progress.

```java
    private IObjectIterator iteratorFor(Object object) {
        try {
            if (object == null) {
                return null;
            }

            if (object.getClass().isArray()) {
                return new ArrayIterator(name, object);
            }

            if (object instanceof Iterable) {
                return new CollectionIterator(name, (Iterable<?>) object);
            }

            if (object instanceof Map) {
                return new MapIterator(name, (Map<?, ?>) object);
            }

            if (object instanceof Map.Entry) {
                return new MapEntryIterator(name, (Map.Entry<?, ?>) object);
            }

            if (isSingleValued(object)) {
                return null;
            }

            return new PropertyIterator(object);
        } catch (RuntimeException e) {
            throw e;
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage(), e);
        }

    }
```

Implementing the String Generator

You've already seen how to iterate over all the properties in your object. All that's left is to pretty it up and add some constraints (so that you're not creating a gigabyte-sized string).

```
 1 public static String generate(Object object) {
 2     String s = "";
 3
 4     ObjectIterator iterator = new ObjectIterator("object", object);
 5
 6     ...
 7
 8     while (iterator.next()) {
 9         if (s.length() >= MAX_STRING_LENGTH) {
10             return s;
11         }
12
13         if (iterator.getChild() >= MAX_CHILDREN) {
14             iterator.nextParent();
15             continue;
16         }
17
18         String valueAsString = iterator.getValueAsString();
19
20         s += System.lineSeparator();
21         s += indent(iterator.getDepth()) + truncateString(iterator.getName());
22         if (valueAsString == null) {
23             s += " = null";
24         } else {
25             s += " = " + truncateString(valueAsString);
26         }
27     }
28
29     return s;
30 }
```

The formatting is done on line 21. It's nothing fancy, just an indent appropriate to the property's distance (or depth) from the root node.

The constraints can be seen on each of the highlighted lines:

- line 9 – limits the generated string to about 16k characters.
- line 13 – limits the number of children to 64 for any parent.

- lines 21 & 25 – limits the keys and values to 64 characters each.

Conclussion

You’ve now seen how to use the iterator pattern to traverse a heterogeneous graph of objects. The key is to delegate iteration of each type to its own class. You also now have two tools you can modify or use as-is in your software.

About Dele Taylor

Dele Taylor is the founder of StackHunter.com -- a tool to track Java exceptions. You can follow him on Twitter, G+, and LinkedIn.