The Java™ Collections API is far more than a replacement for arrays, though that’s not a bad place to start. Ted Neward dispenses five tips for doing more with Collections, including a primer on customizing and extending the Java Collections API.

The Java Collections API came to many Java developers as a much needed replacement for the standard Java array and all of its shortcomings. Associating Collections primarily with ArrayList isn’t a mistake, but there’s much more to the Collections for those who go looking.

About this series
So you think you know about Java programming? The fact is, most developers scratch the surface of the Java platform, learning just enough to get the job done. In this series, Ted Neward digs beneath the core functionality of the Java platform to uncover little known facts that could help you solve even the stickiest programming challenges.

Similarly, while the Map (and its oft-chosen implementation, HashMap) are great for doing name-value or key-value pairs, there’s no reason to limit yourself to these familiar tools. You can fix a lot of error prone code with the right API, or even the right Collection.

This second article in the 5 things series is the first of several devoted to Collections, because they’re so central to what we do in Java programming. I’ll start at the beginning with a look at the quickest (but possibly not the most common) ways to do everyday things, like swapping out Arrays for Lists. After that we’ll delve into lesser known stuff, like writing a custom Collections class and extending the Java Collections API.

1. Collections trump arrays
Developers new to Java technology may not know that arrays were originally included in the language to head-off performance criticism from C++ developers back in the early 1990s. Well, we’ve come a long way since then, and the array’s performance advantages generally come up short when weighed against those of the Java Collections libraries.
Dumping array contents into a string, for example, requires iterating through the array and concatenating the contents together into a `String`; whereas, the Collections implementations all have a viable `toString()` implementation.

Except for rare cases, it’s good practice to convert any array that comes your way to a collection as quickly as possible. Which then begs the question, what’s the easiest way to make the switch? As it turns out, the Java Collections API makes it easy, as shown in Listing 1:

Listing 1. ArrayToList

```java
class ArrayToList
{
    public static void main(String[] args)
    {
        // This gives us nothing good
        System.out.println(args);

        // Convert args to a List of String
        List<String> argList = Arrays.asList(args);

        // Print them out
        System.out.println(argList);
    }
}
```

Note that the returned `List` is unmodifiable, so attempts to add new elements to it will throw an `UnsupportedOperationException`.

And, because `Arrays.asList()` uses a `varargs` parameter for elements to add into the `List`, you can also use it to easily create `List`s out of `new`ed objects.

2. Iterating is inefficient

It's not uncommon to want to move the contents of one collection (particularly one that was manufactured out of an array) over into another collection or to remove a small collection of objects from a larger one.

You might be tempted to simply iterate through the collection and add or remove each element as it's found, but don't.

Iterating, in this case, has major disadvantages:

- It would be inefficient to resize the collection with each add or remove.
- There's a potential concurrency nightmare in acquiring a lock, doing the operation, and releasing the lock each time.
- There's the race condition caused by other threads banging on your collection while the add or remove is taking place.

You can avoid all of these problems by using `addAll` or `removeAll` to pass in the collection containing the elements you want to add or remove.
3. For loop through any Iterable

The enhanced for loop, one of the great conveniences added to the Java language in Java 5, removed the last barrier to working with Java Collections.

Before, developers had to manually obtain an `Iterator`, use `next()` to obtain the object pointed to from the `Iterator`, and check to see if more objects were available via `hasNext()`. Post Java 5, we're free to use a for-loop variant that handles all of the above silently.

Actually, this enhancement works with any object that implements the `Iterable` interface, not just `Collections`.

Listing 2 shows one approach to making a list of children from a `Person` object available as an `Iterator`. Rather than handing out a reference to the internal `List` (which would enable callers outside the `Person` to add kids to your family — something most parents would find uncool), the `Person` type implements `Iterable`. This approach also enables the enhanced for loop to walk through the children.

**Listing 2. Enhanced for loop: Show me your children**

```java
// Person.java
import java.util.*;

public class Person
    implements Iterable<Person>
{
  public Person(String fn, String ln, int a, Person... kids)
  {
    this.firstName = fn; this.lastName = ln; this.age = a;
    for (Person child : kids)
      children.add(child);
  }
  public String getFirstName() { return this.firstName; }
  public String getLastName() { return this.lastName; }
  public int getAge() { return this.age; }
  public Iterator<Person> iterator() { return children.iterator(); }
  public void setFirstName(String value) { this.firstName = value; }
  public void setLastName(String value) { this.lastName = value; }
  public void setAge(int value) { this.age = value; }
  public String toString()
  {
    return "[Person: " +
          "firstName=" + firstName + " " +
          "lastName=" + lastName + " " +
          "age=" + age + "]";
  }
  private String firstName;
  private String lastName;
  private int age;
  private List<Person> children = new ArrayList<Person>();
}

// App.java
public class App
{
  public static void main(String[] args)
  {
```
Person ted = new Person("Ted", "Neward", 39,
        new Person("Michael", "Neward", 16),
        new Person("Matthew", "Neward", 10));

    // Iterate over the kids
    for (Person kid : ted)
    {
        System.out.println(kid.getFirstName());
    }
}
}

Using `Iterable` has some obvious drawbacks when domain modeling, because only one such collection of objects can be so "implicitly" supported via the `iterator()` method. For cases where the child collection is obvious and apparent, however, `Iterable` makes programming against the domain type much easier and more obvious.

### 4. Classic and custom algorithms

Have you ever wanted to walk a `Collection`, but in reverse? That's where a classic Java Collections algorithm comes in handy.

The children of `Person` in Listing 2 above, are listed in the order that they were passed in; but, now you want to list them in the reverse order. While you could write another for loop to insert each object into a new `ArrayList` in the opposite order, the coding would grow tedious after the third or fourth time.

That's where the underused algorithm in Listing 3 comes in:

**Listing 3. ReverseIterator**

```java
public class ReverseIterator
{
    public static void main(String[] args)
    {
        Person ted = new Person("Ted", "Neward", 39,
            new Person("Michael", "Neward", 16),
            new Person("Matthew", "Neward", 10));

        // Make a copy of the List
        List<Person> kids = new ArrayList<Person>(ted.getChildren());
        // Reverse it
        Collections.reverse(kids);
        // Display it
        System.out.println(kids);
    }
}
```

The `Collections` class has a number of these "algorithms," static methods that are implemented to take `Collections` as parameters and provide implementation-independent behavior on the collection as a whole.

What's more, the algorithms present on the `Collections` class certainly aren't the final word in great API design — I prefer methods that don't modify the contents (of the Collection passed in) directly, for example. So it's a good thing you can write custom algorithms of your own, like the one shown in Listing 4:
Listing 4. Reverseliterator made simpler

```java
class MyCollections {
    public static <T> List<T> reverse(List<T> src) {
        List<T> results = new ArrayList<T>(src);
        Collections.reverse(results);
        return results;
    }
}
```

5. Extend the Collections API

The customized algorithm above illustrates a final point about the Java Collections API: that it was always intended to be extended and morphed to suit developers' specific purposes.

So, for example, say you needed the list of children in the `Person` class to always be sorted by age. While you could write code to sort the children over and over again (using the `Collections.sort` method, perhaps), it would be far better to have a `Collection` class that sorted it for you.

In fact, you might not even care about preserving the order in which the objects were inserted into the `Collection` (which is the principal rationale for a `List`). You might just want to keep them in a sorted order.

No `Collection` class within `java.util` fulfills these requirements, but it's trivial to write one. All you need to do is create an interface that describes the abstract behavior the `Collection` should provide. In the case of a `SortedCollection`, the intent is entirely behavioral.

Listing 5. SortedCollection

```java
public interface SortedCollection<E> extends Collection<E> {
    public Comparator<E> getComparator();
    public void setComparator(Comparator<E> comp);
}
```

It's almost anticlimactic to write an implementation of this new interface:

Listing 6. ArraySortedCollection

```java
import java.util.*;

public class ArraySortedCollection<E> implements SortedCollection<E>, Iterable<E> {
    private Comparator<E> comparator;
    private ArrayList<E> list;
    public ArraySortedCollection(Comparator<E> c) {
        this.list = new ArrayList<E>();
        this.comparator = c;
    }
    public ArraySortedCollection(Collection<? extends E> src, Comparator<E> c) {
        this.list = new ArrayList<E>(src);
        this.comparator = c;
    }
    public void sortThis() {
        Collections.sort(this.list, comparator);
    }
}
```
This quick-and-dirty implementation, written with no optimizations in mind, could obviously stand some refactoring. But the point is, the Java Collections API was never intended to be the final word in all things collection-related. It both needs and encourages extensions.

Certainly, some extensions will be of the "heavy-duty" variety, such as those introduced in java.util.concurrent. But others will be as simple as writing a custom algorithm or a simple extension to an existing Collection class.
Extending the Java Collections API might seem overwhelming, but once you start doing it, you'll find it's nowhere near as hard as you thought.

**Conclusion**

Like Java Serialization, the Java Collections API is full of unexplored nooks and crannies — which is why we're not done with this subject. The next article in the 5 things series will give you five more ways to do even more with the Java 6 Collections API.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code for this article</td>
<td>j-5things2-src.zip</td>
<td>10KB</td>
</tr>
</tbody>
</table>
Related topics

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- "Java theory and practice: Concurrent collections classes" (Brian Goetz, developerWorks, July 2003): Learn how Doug Lea's util.concurrent package revitalizes standard collection types List and Map.
- "Spice up collections with generics and concurrency" (John Zukowski, developerWorks, April 2008): Introduces changes to the Java Collections Framework in Java 6.