Chapter 5

Objectives

- Introduce the Java `generic`
- Introduce the Java Collections Framework (JCF)
- Introduce the Java `iterator`
- Implementation of a user-constructed Collection class
- Testing using `JUnit`

Generics

- Up till now we have hard-coded the `Record` data payload
- Why can't we have the type itself be a variable?
- This is the Java generic construct

Generics (2)

```
ArrayList<Record> myList;
```

```
void swap(ArrayList<T> list, int sub1, int sub2)
{
  T temp;
  temp = list.get(sub2);
  list.set(sub2) = this.get(sub1);
  list.set(sub1) = this.get(temp;
}
```

Old Style DLL Code

```
public class DLLNode
{
  private DLLNode next;
  private DLLNode prev;
  private Record nodeData;

  constructor code ...

  public Record getNodeData()
  {
    return this.nodeData;
  }
  public void setNodeData(Record newData)
  {
    this.nodeData = newData;
  }
  public DLLNode getNext()
  {
    return this.next;
  }
  public void setNext(DLLNode newNext)
  {
    this.next = newNext;
  }
  public DLLNode getPrev()
  {
    return this.prev;
  }
  public void setPrev(DLLNode newPrev)
  {
    this.prev = newPrev;
  }

}
```

Figure 4.10 Code fragment for a node

Old Style DLL Code (2)

```
public class DLL
{
  private int size;
  private DLLNode head;
  private DLLNode tail;

 constructor code ...

  public void add(Record dllData)
  {
    this.addAtHead(dllData);
  }

  private void addAtHead(Record dllData
  {
    DLLNode newNode = null;
    newNode = new DLLNode();
    newNode.setNodeData(dllData);
    this.linkAfter(this.getHead(), newNode);
  }
```

```
  private void linkAfter(DLLNode baseNode, DLLNode newNode)
  {
    newNode.setNext(baseNode.getNext());
    newNode.setPrev(baseNode);
    baseNode.getNext().setPrev(newNode);
    baseNode.setNext(newNode);
    this.incrementSize();
  }

  private void unlink (DLLNode node)
  {
    node.getNext().setPrev(node.getPrev());
    node.getPrev().setNext(node.getNext());
    node.setNext(null);
    node.setPrev(null);
    this.decrementSize();
  }
}
```

Figure 4.11 Code fragment for a doubly linked list

DLL Code With Generics

```
public class DLL<T extends Comparable<T>>
{
  private int size;
  private DLLNode<T> head;
  private DLLNode<T> tail;

  public DLL()
  {
    this.head = new DLLNode<T>();
    this.tail = new DLLNode<T>();
    head.setNext(this.tail);
    tail.setPrev(this.head);
    this.setSize(2);
  }
  private DLLNode<T> getHead()
  {
    return this.head;
  }
  private void setHead(DLLNode<T> value)
  {
    this.head = value;
  }
    code for the size variable here ...
  private DLLNode<T> getTail()
  {
    return this.tail;
  }
  private void setTail(DLLNode<T> value)
  {
    this.tail = value;
  }
    more code ...
}
```

Figure 5.1 Code fragment for a doubly linked list using generics, part 1

DLL Code with Generics (2)

```
private void linkAfter(DLLNode<T> baseNode, DLLNode<T> newNode)
{
  newNode.setNext(baseNode.getNext());
  newNode.setPrev(baseNode);
  baseNode.getNext().setPrev(newNode);
  baseNode.setNext(newNode);
  this.incSize();
}
private void unlink (DLLNode<T> node)
{
  node.getNext().setPrev(node.getPrev());
  node.getPrev().setNext(node.getNext());
  node.setNext(null);
  node.setPrev(null);
  this.decSize();
}
```

Figure 5.2 Code fragment for a doubly linked list using generics, part 2

DLL Code with Generics (3)

```
/*************************************************************************
 * Method to find if a list has a given data item.
 * @param dllData the <code>T</code> to match against.
 * @return the <code>boolean</code> answer to the question.
**/
  public boolean contains(T dllData)
  {
    boolean returnValue = false;
    DLLNode<T> foundNode = null;
    foundNode = this.containsNode(dllData);
    if(null != foundNode)
    {
      returnValue = true;
    }
    return returnValue;
  }
```

```
/*************************************************************************
 * Method to remove a node with a given record as data.
 * @param dllData the <code>T</code> to match against.
 * @return the <code>boolean</code> as to whether the record was
 *         found and removed or not.
**/
  public boolean remove(T dllData)
  {
    boolean returnValue = false;
    DLLNode<T> foundNode = null;
    foundNode = this.containsNode(dllData);
    if(null != foundNode)
    {
      this.unlink(foundNode);
      returnValue = true;
    }
    return returnValue;
  }
```

Figure 5.3 Code fragment for a doubly linked list using generics, part 3

DLL Code with Generics (4)

```
/*************************************************************************
 * Method to return the node with a given data item in it, else null.
 * This method eliminates duplicate code in <code>contains</code>
 * and <code>remove</code>.
 * @param dllData the <code>T</code> to match against.
 * @return the <code>DLLNode</code> answer, else null.
**/
  public DLLNode<T> containsNode(T dllData)
  {
    DLLNode<T> returnValue = null;
    DLLNode<T> currentNode = null;

    currentNode = this.getHead();
    currentNode = currentNode.getNext();
    while(currentNode != this.getTail())
    {
      if(0 == currentNode.getNodeData().compareTo(dllData))
      {
        returnValue = currentNode;
        break; // we violate the style rule against 'break'
      }
      currentNode = currentNode.getNext();
    }

    return returnValue;
  }
```

Figure 5.4 Code fragment for a doubly linked list using generics, part 4

DLL Code with Generics (5)

```
public class DLLNode<T>
{
  private DLLNode<T> next;
  private DLLNode<T> prev;
  private T nodeData;
  public DLLNode()
  {
    super();
    this.setNext(null);
    this.setPrev(null);
    this.setNodeData(null);
  }
  public DLLNode(T data)
  {
    super();
    this.setNext(null);
    this.setPrev(null);
    this.setNodeData(data);
  }
  public T getNodeData()
  {
    return this.nodeData;
  }
  public void setNodeData(T newData)
  {
    this.nodeData = newData;
  }
  public DLLNode<T> getNext()
  {
    return this.next;
  }
  public void setNext(DLLNode<T> newNext)
  {
    this.next = newNext;
  }
  public DLLNode<T> getPrev()
  {
    return this.prev;
  }
  public void setPrev(DLLNode<T> newPrev)
  {
    this.prev = newPrev;
  }
}
```

Figure 5.5 Code fragment for a node in a doubly linked list using generics

Generics

In order for the compiler to generate code, it must know that methods invoked are **always** going to exist.

We often override `compareTo` or `equals` exactly for this purpose.

compareTo

```
public class DLL<T extends Comparable>
```

```
if(0 == currentNode.getNodeData().compareTo(dllData))
```

```
public class Record extends Comparable<Record>
```

Comparable

java.lang

Interface Comparable<T>

Type Parameters:

`T` - the type of objects that this object may be compared to

All Known Subinterfaces:

Delayed, Name, RunnableScheduledFuture<V>, ScheduledFuture<V>

All Known Implementing Classes:

Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, Character, CharBuffer, Charset, ClientInfoStatus, CollationKey, Component.BaselineResizeBehavior, CompositeName, CompoundName, Date, Date, Desktop.Action, Diagnostic.Kind, Dialog.ModalExclusionType, Dialog.ModalityType, Double, DoubleBuffer, DropMode, ElementKind, ElementType, Enum, File, Float, FloatBuffer, Formatter.BigDecimalLayoutForm, FormSubmitEvent.MethodType, GregorianCalendar, GroupLayout.Alignment, IntBuffer, Integer, JavaFileObject.Kind, JTable.PrintMode, KeyRep.Type, LayoutStyle.ComponentPlacement, LdapName, Long, LongBuffer, MappedByteBuffer, MemoryType, MessageContext.Scope, Modifier, MultipleGradientPaint.ColorSpaceType, MultipleGradientPaint.CycleMethod, NestingKind, Normalizer.Form, ObjectName, ObjectStreamField, Proxy.Type, Rdn, Resource.AuthenticationType, RetentionPolicy, RoundingMode, RowFilter.ComparisonType, RowIdLifetime, RowSorterEvent.Type, Service.Mode, Short, ShortBuffer, SOAPBinding.ParameterStyle, SOAPBinding.Style, SOAPBinding.Use, SortOrder, SourceVersion, SSLEngineResult.HandshakeStatus, SSLEngineResult.Status, StandardLocation, String, SwingWorker.StateValue, Thread.State, Time, Timestamp, TimeUnit, TrayIcon.MessageType, TypeKind, URI, UUID, WebParam.Mode, XmlAccessOrder, XmlAccessType, XmlNsForm

---

```
public interface Comparable<T>
```

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`. Note that `null` is not an instance of any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.

For example, if one adds two keys `a` and `b` such that `(!a.equals(b) && a.compareTo(b) == 0)` to a sorted set that does not use an explicit comparator, the second `add` operation returns false (and the size of the sorted set does not increase) because `a` and `b` are equivalent from the sorted set's perspective.

Java's Built-In `Linked List`

```
import java.util.LinkedList

import java.util.ListIterator
```

LinkedList

ListIterator

Iterators

- One can write deep metaphysical commentaries on iterators
- But it really boils down to this: *track the data, not the object that contains the data*
- `Iterable`
- `Iterator`
- `ListIterator`

Iterators

Track the data, not the object that contains the data

```
// code fragment one
node = head;
while(node != tail)
{
  Record rec = node.getRecord();
  s = String.format("%s%n", rec.toString());
  node = node.next();
}
```

```
// code fragment two
for(Record rec: this.dll)
{
  s = String.format("%s%n", rec.toString());
}
```

Iterators

```
/**********************************************************************
 * Method to <code>toString</code> a complete Phonebook.
 * @return the <code>toString</code> rep'n of the entire DLL.
**/
  public String toString()
  {
    String s = "";
    Record rec;
    ListIterator<Record> iter = this.dll.listIterator();
    while(iter.hasNext())
    {
      rec = iter.next();
      s += String.format("%s%n",rec.toString());
    }
    return s;
  }
```

Figure 5.6 Code fragment for a `toString` method

Testing with `JUnit`

- `JUnit` allows for “unit testing” of Java programs
- Individual code modules can be tested and exercised to ensure that all lines of code have been executed before delivery
- Facilitates testing of obscure code and exception handling, among other things

Testing with `Junit` (2)

- Using `JUnit` requires the `jar` file
- In Eclipse:
  - Project, Properties, Build Path
  - Add the external `jar` file
- You will then see a `JUnit` option on the Run button

Testing with `Junit` (3)

- In addition to the “actual” Java class, you will need a tester class
- What you execute is the tester class, and in Eclipse you either get GREEN check marks for successful execution or RED for failure, with failure locations pointed out

Testing with `Junit` (4)

- The basic yoga is to write a tester class that executes code, assigns values to variables, etc., and then to `assert` that what ought to be true is in fact true.
- Yes, this requires writing the correct code for the "actual" code and then the correct code for the "testing" code.

Testing with `Junit` (5)

```
assertEquals(expectedValue, actualValue)}
assertEquals(messageString, expectedValue, actualValue)}
assertFalse(booleanCondition)}
assertFalse(messageString, booleanCondition)}
assertNotNull(object)}
assertNotNull(messageString, object)}
assertNotSame(expectedvalue, actualValue)}
assertNotSame(messageString, expectedValue, actualValue)}
assertNull(object)}
assertNull(messageString, object)}
assertSame(expectedValue, actualValue)}
assertSame(messageString, expectedValue, actualValue)}
assertTrue(booleanCondition)}
assertTrue(messageString, booleanCondition)}
failNotEquals(messageString, expectedValue, actualValue)}
failNotSame(messageString, expectedValue, actualValue)}
```

Figure 5.12 Assertion methods in JUnit

Junit Examples

- Look at actual code examples…

```
import junit.framework.*;
import java.util.Scanner;
/**************************************************************
 *
**/
public class RecordTester extends TestCase
{
  private Record rec1,rec2;
/**************************************************************
 *
**/
  public RecordTester(String name)
  {
    super(name);
  }
```

```
/**************************************************************
 *
**/
  protected void setUp()
  {
    rec1 = new Record();
    rec2 = new Record();
  }
/**************************************************************
 *
**/
  protected void tearDown()
  {
    rec1 = null;
    rec2 = null;
  }
/**************************************************************
 *
**/
  public void testConstructor()
  {
    System.out.println("Test the constructor");
    rec1 = new Record();
    assertEquals(Record.DUMMYSTRING, rec1.getName());
    assertEquals(Record.DUMMYSTRING, rec1.getPhone());
    assertEquals(Record.DUMMYSTRING, rec1.getOffice());
    assertEquals(Record.DUMMYINT, rec1.getTeaching());
  }
```

Figure 5.13 A JUnit testing class for Record, part 1

Junit Examples (2)

```
/************************************************************************
 *
**/
  public void testCompareTo()
  {
    System.out.println("Test compareTo");
    rec1 = new Record();
    rec2 = new Record();
    assertEquals(Record.DUMMYSTRING, rec1.getName());
    assertEquals(Record.DUMMYSTRING, rec2.getName());
    rec1.setName("duncan");
    assertEquals("Failure compareTo set", "duncan", rec1.getName());
    rec2.setName("duncan");
    assertEquals("duncan", rec2.getName());
    assertEquals("Failure compareTo equals", 0, rec1.compareTo(rec2));
    rec2.setName("aaaa");
    assertEquals("aaaa", rec2.getName());
    assertEquals("Failure compareTo greaterthan", 1, rec1.compareTo(rec2));
    rec2.setName("eeee");
    assertEquals("eeee", rec2.getName());
    assertEquals("Failure compareTo lessthan", -1, rec1.compareTo(rec2));
  }
TEST FOR compareName IS SIMILAR
TEST FOR compareName IS SIMILAR
/************************************************************************
 *
**/
  public void testDefaultInstances()
  {
    assertEquals(Record.DUMMYSTRING, rec1.getName());
  }
TESTS FOR getPhone, getOffice, getTeaching ARE SIMILAR
/************************************************************************
 *
**/
  public void testSetName()
  {
    assertEquals(Record.DUMMYSTRING, rec1.getName());
    rec1.setName("duncan");
    assertEquals("Name Failure","duncan", rec1.getName());
    assertFalse("messagefail name", rec1.getName().equals("Someone"));
  }
TESTS FOR setPhone, setOffice, setTeaching ARE SIMILAR
```

Figure 5.14 A JUnit testing class for Record, part 2

The End