Enhanced for loops vs. Iterator

- The enhanced for loop, or for-each loop, is provided to iterate over collections.
- The old way:

```
for( int j = 0; j < list.size(); j++)
  System.out.println(list.getPerson(j));
```

- Using Iterator :

```
Iterator itr<Person> = list.iterator();
while(itr.hasNext()) {
  Person p = itr.next();
  System.out.println(p);
}
```

- The new way:

```
for( Person p: list )
  System.out.println( p );
```

Java's Iterable<T> Interface

- Implementing the Iterable<T> interface (Aggregate) means that class:
  - Defines a method Iterator<T> iterator() which returns a ConcreteIterator object which iterates over a set of elements of type T (the type aggregated by Aggregator).
  - An object of the class can be the target of the foreach statement, i.e.,

```
for (Person p: Phonebook.iterator()) {
     System.out.println( p.name );
}
```

Java's Iterable<T> (contd.)

- The root interface of the collection hierarchy Collection<E> extends Iterable<E> which means that all of the following interfaces implement Iterable:
  - Deque, List, Queue, Set, SortedSet
- This means that all of the following concrete classes implement Iterable as well:
  - ArrayList, HashSet, LinkedList, PriorityQueue, Stack, TreeSet, Vector.

Lambda Expressions and Streams

- The for-each statement works with Iterable:

```
for( Person p: list )
   System.out.println( p );
```

- With lambda expressions and streams (Java 8+), it is even easier:

```
list.forEach(p -> System.out.println( p ));
```

- Here, Java figures out that p is of type Person (because list contains Persons), creates a stream of Persons, then performs the statement after the "->" on each Person using the iterator() from Iterable.

Polymorphic Algorithms

- Java has polymorphic algorithms to provide functionality for different types of collections
  - void sort(List<T> list)
  - void reverse(List<T> list)
  - void swap(List<T> list, int i, int j)
  - boolean replaceAll(List<T> list, T oldVal, T newVal)
  - void rotate(List<?> list, int distance)
- And since List extends interface Collection
  - int frequency(Collection<?> c, Object o)
  - public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)

Collection Implementation

- If we look at the methods in the Collection interface, we might see some that can be implemented purely in terms of other methods in the interface or in java.lang.Object
- For instance:

```
public boolean isEmpty() {
    return size() == 0;
}
```

- But some methods' implementations must be specific to the way the data is actually stored
  - add(), etc.

AbstractCollection

```
public abstract class AbstractCollection<E>
implements Collection<E> {
    boolean  add(E e)
    boolean  addAll(Collection<? extends E> c)
    void  clear()
    boolean  contains(Object o)
    boolean  containsAll(Collection<?> c)
    boolean  isEmpty()
    abstract Iterator<E>  iterator()
    boolean  remove(Object o)
    boolean  removeAll(Collection<?> c)
    boolean  retainAll(Collection<?> c)
    abstract int  size()
    Object[]  toArray()
    <T> T[]  toArray(T[] a)
    String  toString()
}
```

AbstractCollection Javadoc

- This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.
- To implement an unmodifiable collection, the programmer needs only to extend this class and provide implementations for the iterator and size methods. (The iterator returned by the iterator method must implement hasNext and next.)
- To implement a modifiable collection, the programmer must additionally override this class's add method (which otherwise throws an UnsupportedOperationException), and the iterator returned by the iterator method must additionally implement its remove method.

In-Class Activity

- Implement:

  ```
  public boolean contains(Object o)
  ```

- Using only methods from Object, Collection, and AbstractCollection

```
public abstract class
AbstractCollection<E> implements
Collection<E> {
    boolean  add(E e)
    boolean  addAll(Collection<? extends
    E> c)
    void  clear()
    boolean  contains(Object o)
    boolean  containsAll(Collection<?> c)
    boolean  isEmpty()
}
```

```
abstract Iterator<E>  iterator()
boolean  remove(Object o)
boolean  removeAll(Collection<?> c)
boolean  retainAll(Collection<?> c)
abstract int  size()
Object[]  toArray()
<T> T[]  toArray(T[] a)
String  toString()
```

Contains

```
public boolean contains(Object o) {
        Iterator<E> it = iterator();
        if (o==null) {
            while (it.hasNext())
                if (it.next()==null)
                    return true;
        } else {
            while (it.hasNext())
                if (o.equals(it.next()))
                    return true;
        }
        return false;
    }
```

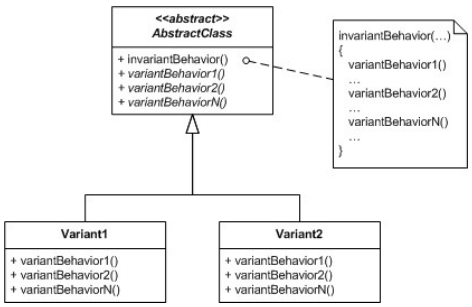
Template Method Pattern

- Recurring problem:
  - We want to share a general algorithm across several specific algorithms. Each specific algorithm has some of the code in the general algorithm.
  - How can code that uses these classes so that the shared code is written only once, but used in all of the specific algorithms, which each have their own code?
- Example: lists
  - There are several implementations of a list: linked list, array list, even a list stored as a tree.
  - Many specific operations, such as isEmpty(), contains(), and equals() can be implemented using just a few primitive operations: size() and get(index).

Template Method Pattern

- Solution:
  - Define an abstract class that implements most of the methods using just a few abstract variant methods, supplied by concrete classes using a specific representation (fields).
  - Differentiate variant methods (those that need to understand the representation) from invariant methods (those that can do their work by calling the variant methods).

Template Method Pattern



Participants

- AbstractClass
  - invariantBehavior() defines the skeleton of the algorithm. The invariantBehavior method is the template.
  - This method will invoke the various variantBehavior() methods, which themselves will be abstract, unless the AbstractClass wants to provide a default behavior.
- ConcreteClass
  - Implements all of the variantBehavior() methods, to carry out subclass-specific steps of the algorithm.

Explanation

- The skeleton of the code is in the invariantBehavior() method.
- This method includes calls to the variantBehavior() methods.
- Due to Java's dynamic binding, the actual methods called will be those in the specific subclass.

ArrayList

- **Class ArrayList<E>**
- Inheritance Hierarchy:
  - java.lang.Object
    - java.util.AbstractCollection<E>
      - java.util.AbstractList<E>
        - java.util.ArrayList<E>

Set<E> and SortedSet<E>

- interface Set<E>
  - add addAll(), but no get()
- Two classes that implement Set<E>
  - **TreeSet**: values stored in order, O(log n)
  - **HashSet**: values in a hash table, no order, O(1)
- SortedSet extends Set by adding methods
  - E first(), E last()
  - SortedSet<E> tailSet(E fromElement),
  - SortedSet<E> headSet(E fromElement)
  - SortedSet<E> subSet(E fromElement, E toElement)

TreeSet<E> elements are in order

```
Set<String> names = new TreeSet<>();

names.add("Dakota");
names.add("Devon");
names.add("Chris");
names.add("Chris"); // not added

for (String name : names)
  System.out.print(name + " "); // Chris Dakota Devon
```

- What if TreeSet<> were changed to HashSet<>?

interface Queue<E>

- boolean add(E e)
  - Inserts e into this queue
- E element()
  - Retrieves, but does not remove, the head of this queue
- boolean offer(E e)
  - Inserts e into this queue
- E peek()
  - Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty
- E poll()
  - Retrieves and removes the head of this queue, or returns null if this queue is empty
- E remove()
  - Retrieves and removes the head of this queue

A FIFO queue

```
ArrayBlockingQueue<Double> numberQ =  new
       ArrayBlockingQueue<>(40);

numberQ.add(3.3);
numberQ.add(2.2);
numberQ.add(5.5);
numberQ.add(4.4);
numberQ.add(7.7);

assertEquals(3.3, numberQ.peek(), 0.1);
assertEquals(3.3, numberQ.remove(), 0.1);
assertEquals(2.2, numberQ.remove(), 0.1);
assertEquals(5.5, numberQ.peek(), 0.1);
assertEquals(3, numberQ.size());
```

The Map Interface (ADT)

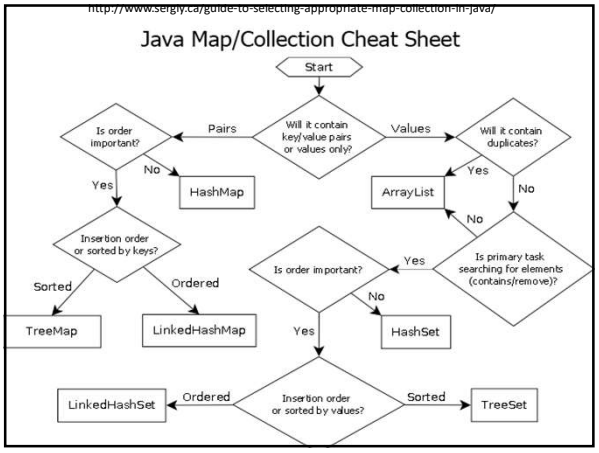
- Map describes a type that stores a collection of elements that consists of a key and a value
- A Map associates (maps) a key the it's value
- The keys must be unique
  - the values need not be unique
  - put destroys one with same key

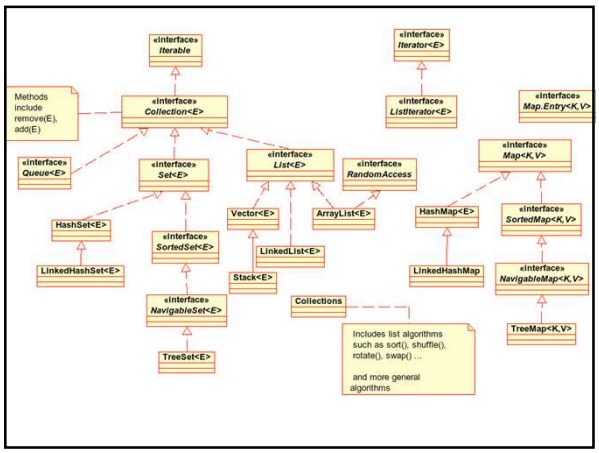
interface Map<K, V>

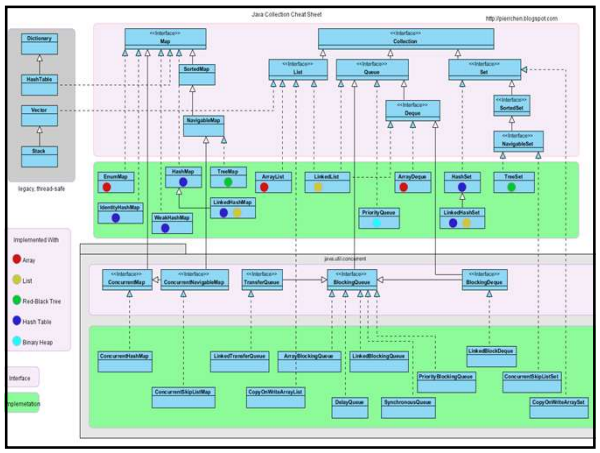
- public V put(K key, V value)
  - associates key to value and stores mapping
- public V get(Object key)
  - associates the value to which key is mapped or null
- public boolean containsKey(Object key)
  - returns true if the Map already uses the key
- public V remove(Object key)
  - returns previous value associated with specified key, or null if there was no mapping for key.
- Collection<V> values()
  - get a collection you can iterate over

keySet and Values

```
Map<String, Integer> rankings = new HashMap<>();
rankings.put("M", 1);
rankings.put("A", 2);
rankings.put("P", 3);
Set<String> keys = rankings.keySet();
System.out.println(keys.getClass());
System.out.println(keys);      // [P, A, M]
Collection<Integer> values = rankings.values();
System.out.println(values.getClass());
System.out.println(values);    // [3, 2, 1]
```




«interface»
Iterable
«interface»
Iterator<E>
Methods include remove(E), add(E)
«interface»
Collection<E>
«interface»
ListIterator<E>
«interface»
Map.Entry<K,V>
«interface»
Queue<E>
«interface»
Set<E>
«interface»
List<E>
«interface»
RandomAccess
«interface»
Map<K,V>
HashSet<E>
«interface»
SortedSet<E>
Vector<E>
ArrayList<E>
HashMap<E>
«interface»
SortedMap<K,V>
LinkedHashSet<E>
LinkedList<E>
Stack<E>
LinkedHashMap
«interface»
NavigableMap<K,V>
«interface»
NavigableSet<E>
Collections
TreeMap<K,V>
TreeSet<E>
Includes list algorithms such as sort(), shuffle(), rotate(), swap() ...
and more general algorithms


Java Collection Cheat Sheet
http://piercehan.blogspot.com
Dictionary
HashTable
Vector
Stack
Map
Collection
SortedMap
List
Queue
Set
Deque
SortedSet
NavigableMap
NavigableSet
EnumMap
HashMap
TreeMap
ArrayList
LinkedList
ArrayDeque
HashSet
TreeSet
IdentityHashMap
WeakHashMap
LinkedHashMap
PriorityQueue
LinkedHashSet
legacy, thread-safe
Implemented With
Array
List
Red-Black Tree
Hash Table
Binary Heap
Interface
Implementation
java.util.concurrent
ConcurrentMap
ConcurrentNavigableMap
TransferQueue
BlockingQueue
BlockingDeque
ConcurrentHashMap
LinkedTransferQueue
ArrayBlockingQueue
LinkedBlockingQueue
LinkedBlockingDeque
PriorityBlockingQueue
ConcurrentSkipListSet
ConcurrentSkipListMap
CopyOnWriteArrayList
DelayQueue
SynchronousQueue
CopyOnWriteArraySet