Java Collections Framework

Objectives

- Understand the structure and design of the JCF
- Know how to use the JCF API
- Know how to choose the appropriate collection classes
- Know how to adapt collection classes to your application

History

- Container classes in JDK 1.1.x
  - `Vector` class
  - `Hashtable` class
  - `Enumeration` Interface
- Java 2 Collections API starting with 1.2
  - Interfaces, classes built around fundamental computer science data structures
  - And utility classes, methods
- Generics added in Java 5
  - Added power…
  - But also some complexity

The Collections Framework

- A “Collection” (capital “C”) is just that: a collection of objects, stored in a nonspecific way
- The Java Collections Framework is based on a few interfaces and several implementations
- Most of the classes we will use are in the `java.util` package

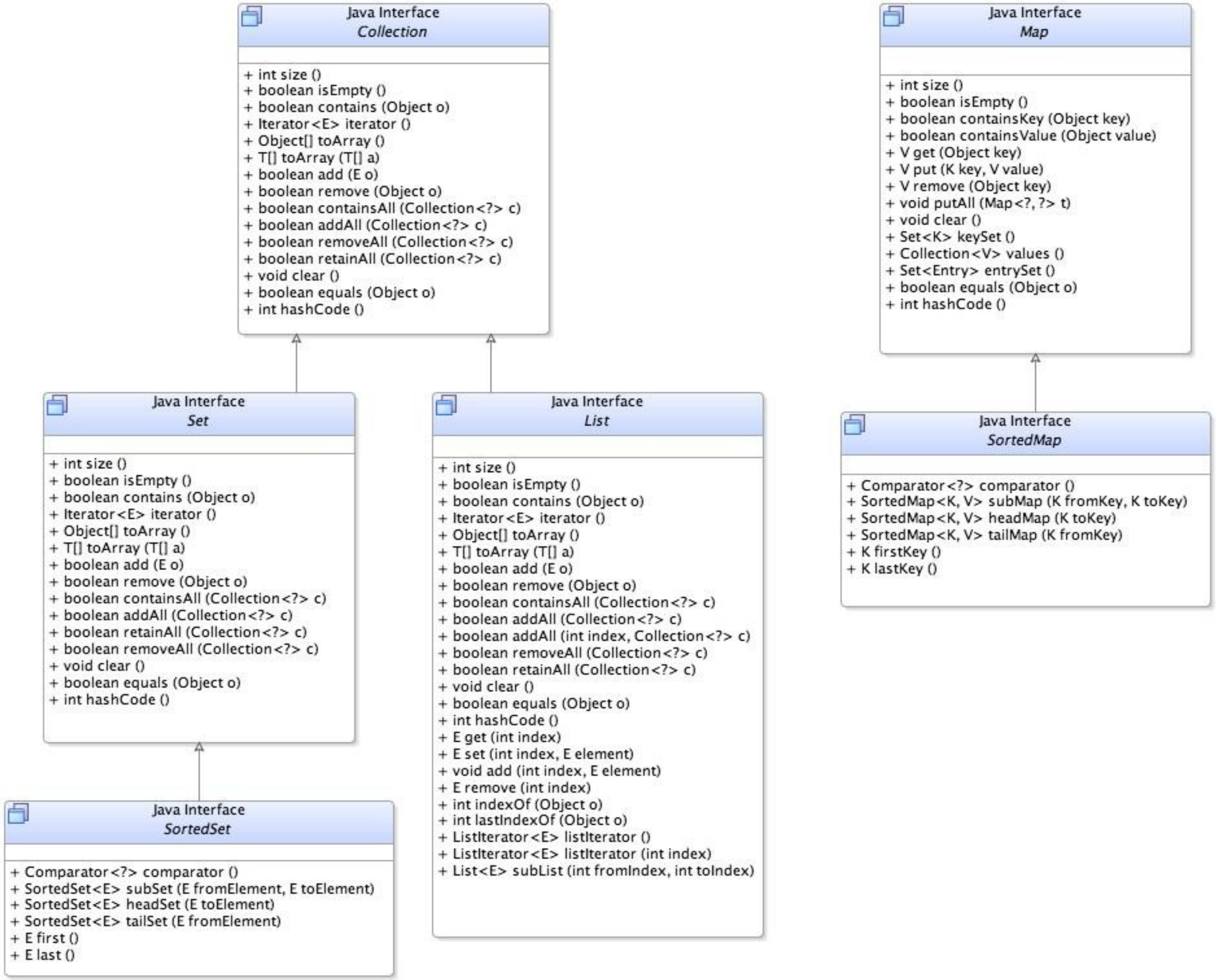
Java Collections API Design

- Design goals
  - 25 classes and interfaces
  - Small, manageable, and consistent
  - Version available for pre-Java 1.2 JDKs

Java 2 Collections API

- Using the collections API means coding to interfaces and choosing an implementation (a good practice in general)
- The interface indicates what you want to do
- The implementation indicates how

Collections



Java Interface
Collection

+ int size ()
+ boolean isEmpty ()
+ boolean contains (Object o)
+ Iterator<E> iterator ()
+ Object[] toArray ()
+ T[] toArray (T[] a)
+ boolean add (E o)
+ boolean remove (Object o)
+ boolean containsAll (Collection<?> c)
+ boolean addAll (Collection<?> c)
+ boolean removeAll (Collection<?> c)
+ boolean retainAll (Collection<?> c)
+ void clear ()
+ boolean equals (Object o)
+ int hashCode ()

Java Interface
Map

+ int size ()
+ boolean isEmpty ()
+ boolean containsKey (Object key)
+ boolean containsValue (Object value)
+ V get (Object key)
+ V put (K key, V value)
+ V remove (Object key)
+ void putAll (Map<?, ?> t)
+ void clear ()
+ Set<K> keySet ()
+ Collection<V> values ()
+ Set<Entry> entrySet ()
+ boolean equals (Object o)
+ int hashCode ()

Java Interface
Set

+ int size ()
+ boolean isEmpty ()
+ boolean contains (Object o)
+ Iterator<E> iterator ()
+ Object[] toArray ()
+ T[] toArray (T[] a)
+ boolean add (E o)
+ boolean remove (Object o)
+ boolean containsAll (Collection<?> c)
+ boolean addAll (Collection<?> c)
+ boolean retainAll (Collection<?> c)
+ boolean removeAll (Collection<?> c)
+ void clear ()
+ boolean equals (Object o)
+ int hashCode ()

Java Interface
List

+ int size ()
+ boolean isEmpty ()
+ boolean contains (Object o)
+ Iterator<E> iterator ()
+ Object[] toArray ()
+ T[] toArray (T[] a)
+ boolean add (E o)
+ boolean remove (Object o)
+ boolean containsAll (Collection<?> c)
+ boolean addAll (Collection<?> c)
+ boolean addAll (int index, Collection<?> c)
+ boolean removeAll (Collection<?> c)
+ boolean retainAll (Collection<?> c)
+ void clear ()
+ boolean equals (Object o)
+ int hashCode ()
+ E get (int index)
+ E set (int index, E element)
+ void add (int index, E element)
+ E remove (int index)
+ int indexOf (Object o)
+ int lastIndexOf (Object o)
+ ListIterator<E> listIterator ()
+ ListIterator<E> listIterator (int index)
+ List<E> subList (int fromIndex, int toIndex)

Java Interface
SortedMap

+ Comparator<?> comparator ()
+ SortedMap<K, V> subMap (K fromKey, K toKey)
+ SortedMap<K, V> headMap (K toKey)
+ SortedMap<K, V> tailMap (K fromKey)
+ K firstKey ()
+ K lastKey ()

Java Interface
SortedSet

+ Comparator<?> comparator ()
+ SortedSet<E> subSet (E fromElement, E toElement)
+ SortedSet<E> headSet (E toElement)
+ SortedSet<E> tailSet (E fromElement)
+ E first ()
+ E last ()

Java 2 Collections API--Interfaces

- Collection: Represents any group of objects
- Set: A collection that cannot contain duplicates
- List: An ordered collection or sequence
- Queue: A first in, first out queue
- Map: A collection of key-value pairs (does not implement Collection interface)

The Collection Interface

- Basic operations

```
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(Object element); // Optional
boolean remove(Object element); // Optional
Iterator iterator();
```

The Collection Interface (cont’d)

- Bulk operations

```
boolean containsAll(Collection c);
boolean addAll(Collection c); // Optional
boolean removeAll(Collection c); // Optional
boolean retainAll(Collection c); // Optional
void clear(); // Optional
```

The Collection Interface (cont’d)

- Array operations

```
Object[] toArray();
<T> T[] toArray(T[] a);
```

- The latter method allows specifying the type of array to return

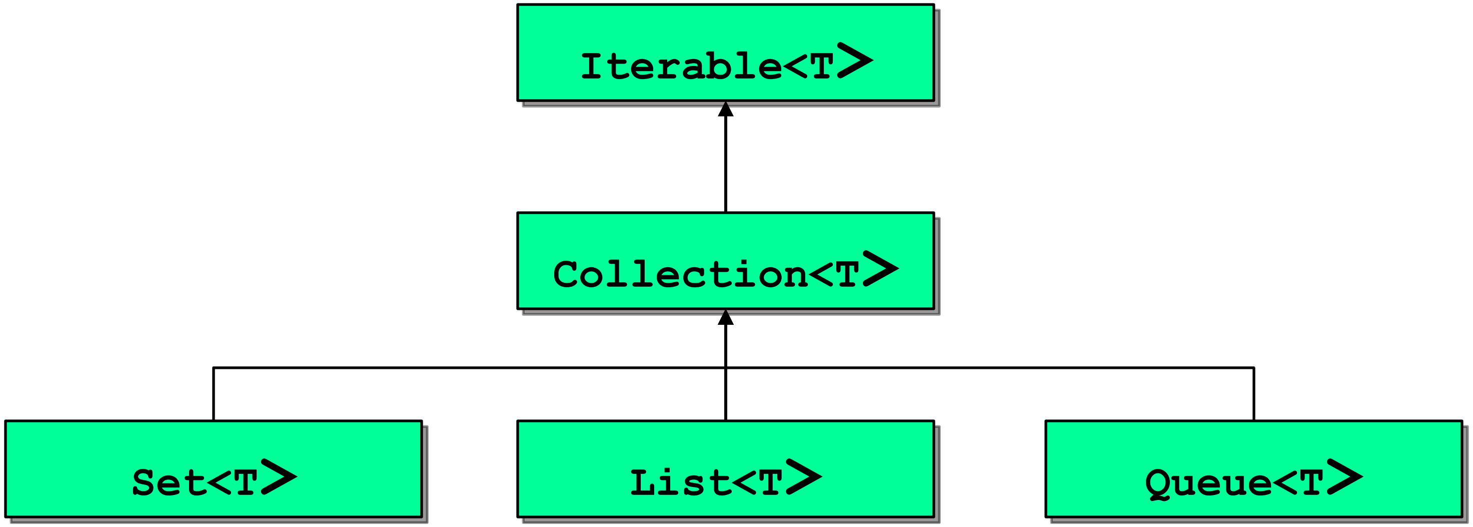
```
String[] strings =
    myCollection.toArray(new String[] {});
```

Utility Classes

Collection<E>

- The `Collection` interface represents collections in a general way
- Serves as a base interface from which more restrictive collections are extended.
- Provides a lowest common denominator that all implementing interfaces can extend

Collection Interface Hierarchy



Utility Classes

List<E>

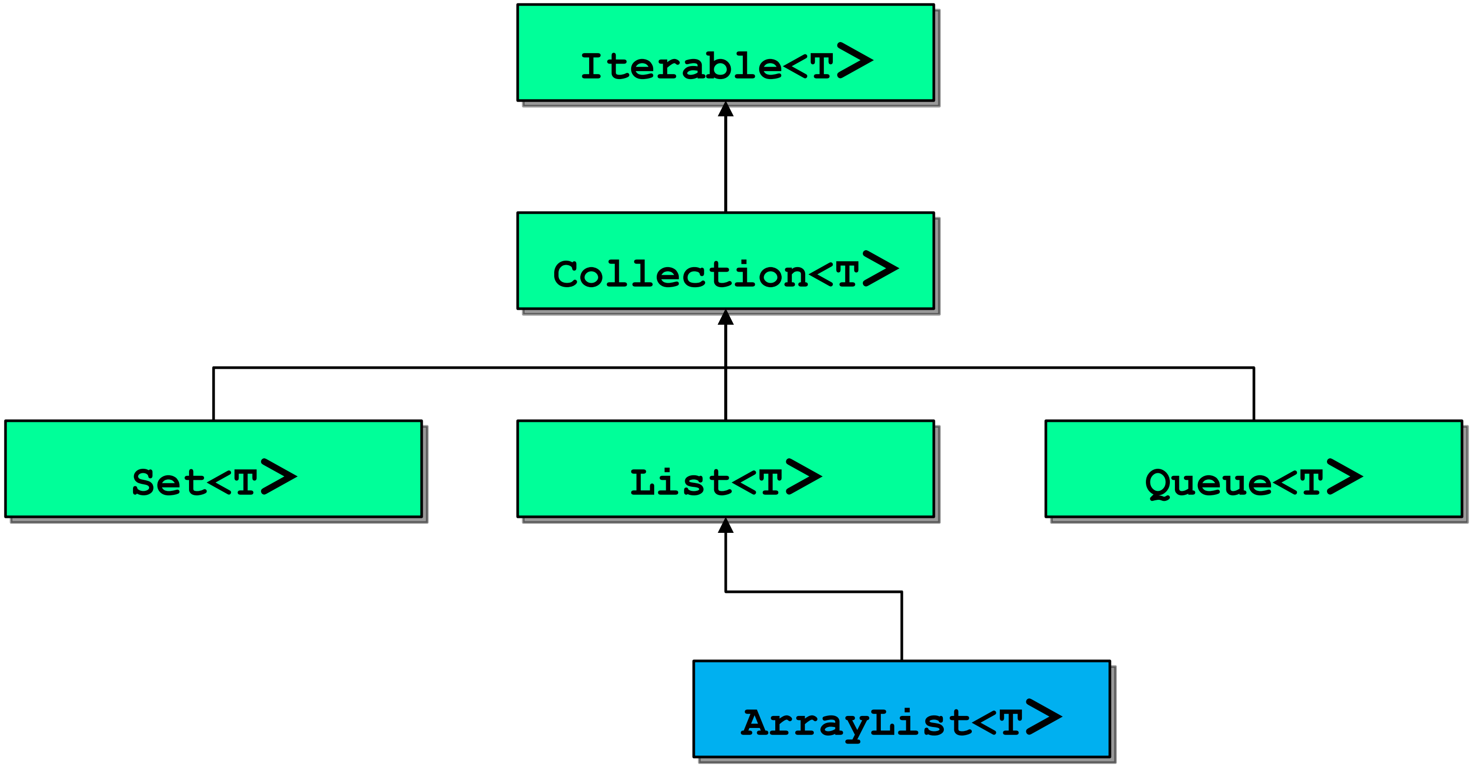
- An ordered collection.
- Provides precise control over where in the list each element is inserted.
- Elements may be accessed by their integer index.
- Provides for searching for elements in the list.
- Typically allow duplicate elements.

Utility Classes

List<E> Operations

```
void add(int index, E element)

boolean add(E o)

boolean addAll(Collection<? extends E> c)

E get(int index)

int indexOf(Object o)

int lastIndexOf(Object o)

E remove(int index)

Object set(int index, E element)

List<E> subList(int fromIndex, int toIndex)
```

List<E>



Utility Classes

ArrayList<E>

- Implementation of a growable array of objects.
- Like an array, contains components that can be accessed using an integer index

Utility Classes

ArrayList<E> Methods

- ArrayList()
- ArrayList(Collection<? extends E> c)
- ArrayList(int initialCapacity)
- void ensureCapacity(int minCapacity)
- void trimToSize()

Utility Classes

Set<E>

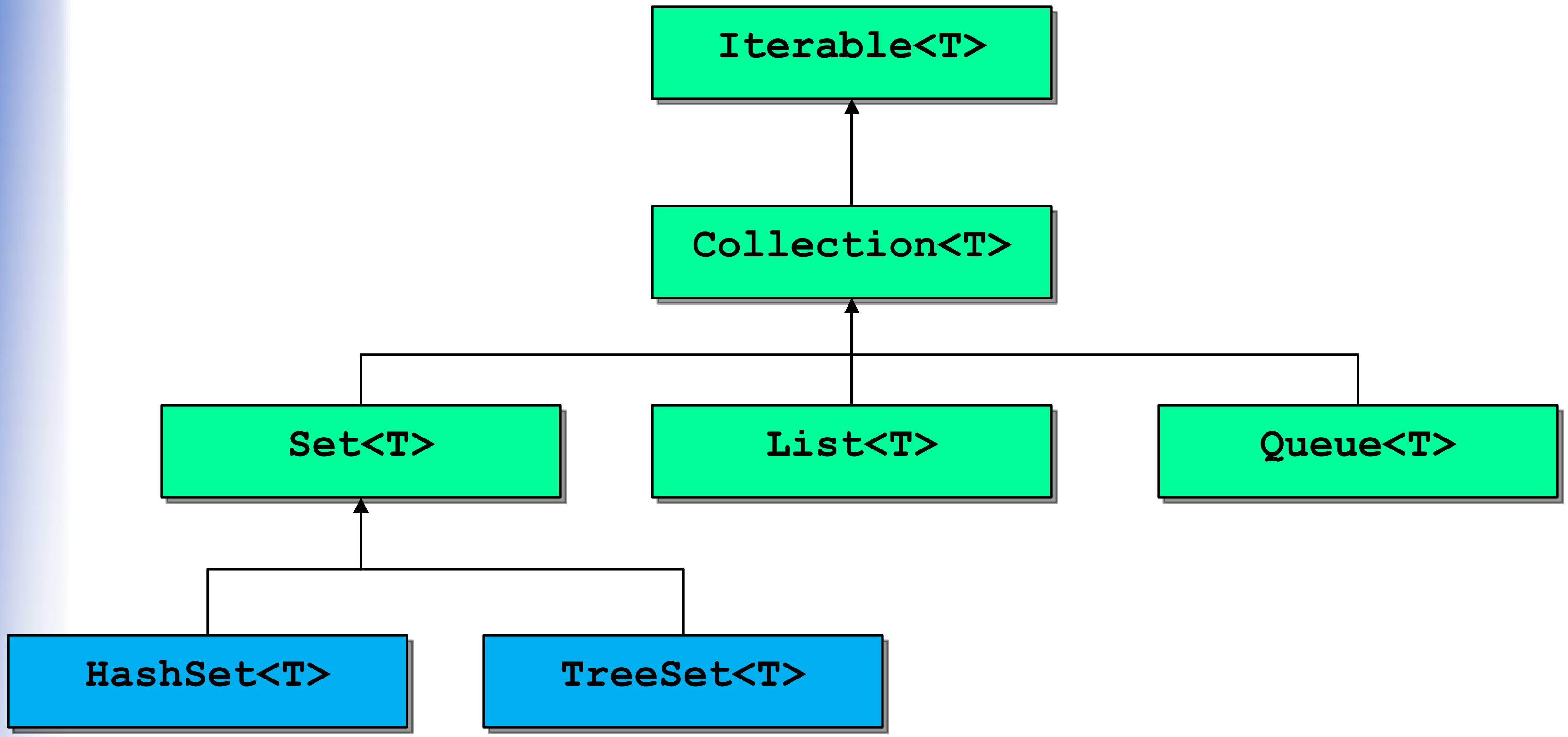
- A collection that contains no duplicate elements.
- Models the mathematical set abstraction.
- Specifies no operations beyond those of the Collection interface.

Utility Classes

HashSet<E>

- A set supported by a HashMap instance.
  - HashSet()
  - HashSet(Collection<? extends E> c)
  - HashSet(int initialCapacity)
  - HashSet(int initialCapacity, float loadFactor)

Set<E>



Utility Classes

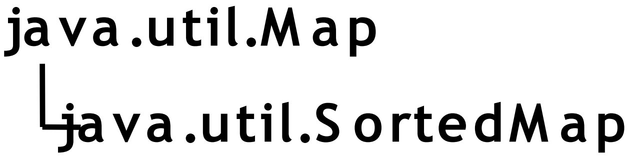
Map<K,V>

- An interface for mapping keys to values.
- Prohibits duplicate keys
- Each key can map to at most one value.
- Provides three collection views
    - a set of keys
    - collection of values
    - set of key-value mappings

Utility Classes

Map<K,V>

– Serves as the root of the map interface hierarchy.



java.util.Map

└java.util.SortedMap

Utility Classes

Map<K,V> Operations

- void clear()
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- Set<Map.Entry<K,V>> entrySet()
- V get(Object key)
- boolean isEmpty()
- Set<K> keySet()
- Object put(K key, V value)
- void putAll(Map<? extends K,? extends V> t)
- V remove(Object key)
- int size()
- Collection<V> values()

Utility Classes

HashMap<K,V>

- Hash table implementation
- Provides all of the optional map operations
- Permits null values and the null key

`Collections` are Type-safe

- Collections use generics to provide type safety (the pre-Java 1.5 way was to cast upon retrieval from collections)
- Typically holds a specific type

```
List<MyClass>
```

- May hold a general type

```
List<Object>
```

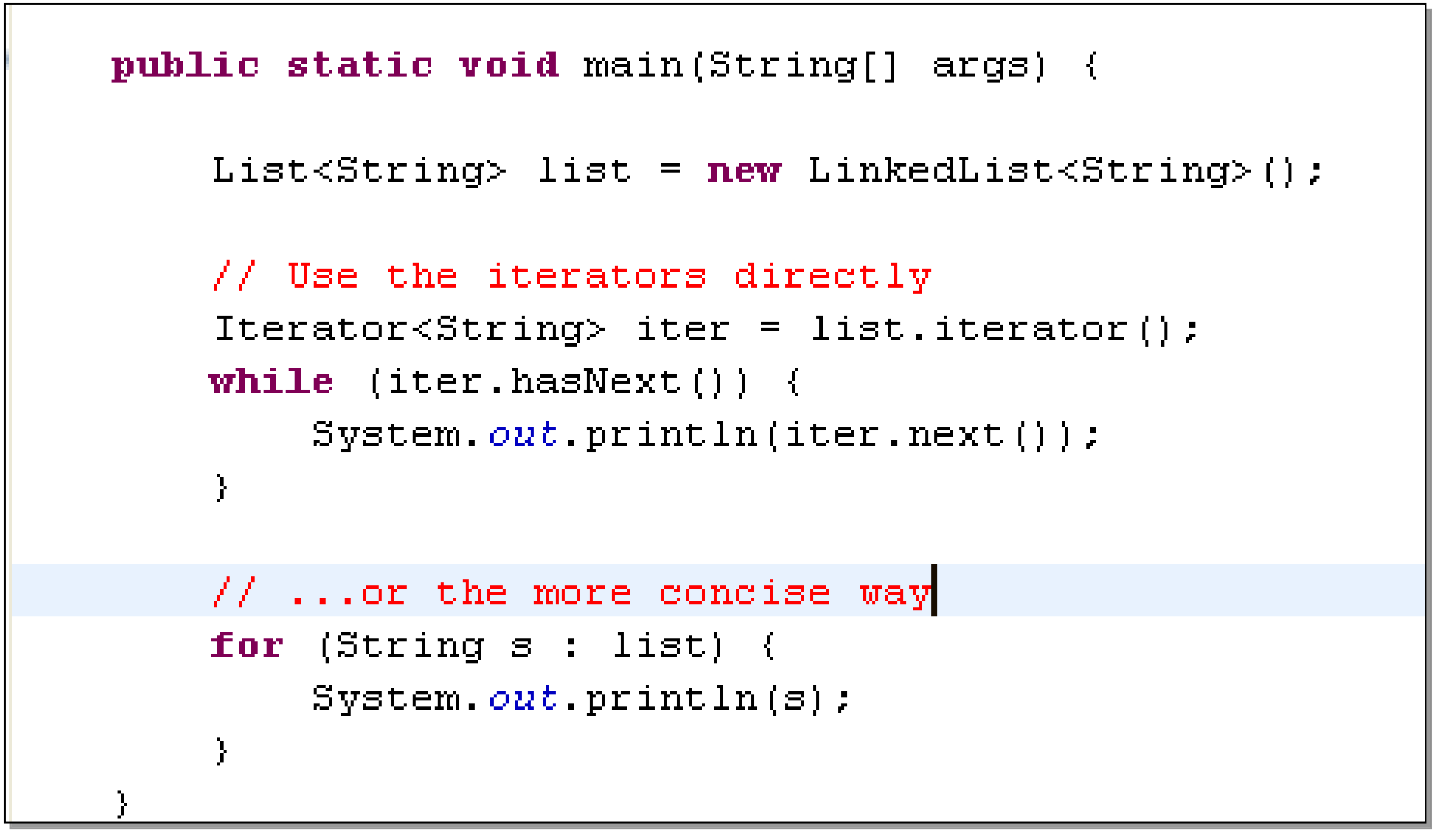
The `Iterator<E>` Interface

- Iterators are objects that know how to step through a collection
- The correct Iterator is obtained by calling the collection's *`iterator()`* method
- `Iterator` methods:

```
boolean hasNext();
<E> next();
void remove(); // Optional
```

- Iterators are NOT reusable, i.e. you can't "rewind" one back to the beginning

Iterator Example



```
public static void main(String[] args) {

    List<String> list = new LinkedList<String>();

    // Use the iterators directly
    Iterator<String> iter = list.iterator();
    while (iter.hasNext()) {
        System.out.println(iter.next());
    }

    // ...or the more concise way
    for (String s : list) {
        System.out.println(s);
    }
}
```

Collections API Implementations

- `List` is implemented as `ArrayList` and `LinkedList`
  - Use `ArrayList` unless you need to insert items at the front of the list, or delete items from the interior
- `Set` and `Map` are implemented as either a hash table or balanced tree (red-black tree)
- `HashSet`, `HashMap`, `TreeSet`, `TreeMap` classes
  - Use `Hash*` implementations unless you need to retrieve data in-order

HashCode and HashSet, HashMap

- HashSet, HashMap use Object.hashCode() to figure out where your object should be stored
- If you don't define your own `hashCode()` method, the inherited version is used
- Not best because objects with identical contents may have different hash codes

`Hashcode` and `Hash*` Classes

- An object's hash code is an integer value that uniquely identifies an object
  - Objects where equals() is true must have identical hash codes
  - Objects where equals() is not true **should** have distinct hash codes
- If you use objects you've created as keys in a `Hash*` collection, you should define a `hashCode()` method for the class
- Common to scramble values of key/unique data fields (Bloch's method in *Effective Java* is the standard)

Programming Tips and Design

- Program in terms of interface types rather than implementation types:
  - `List list = new ArrayList();`
- preferable to
  - `ArrayList list = new ArrayList();`

Programming Tips and Design(cont'd)

- Passing collection types as parameters to and return types from methods
  - Use the least specific type to promote generality
  - Examples:
    - `Collection` instead of `List`
    - `List` instead of `ArrayList`

The `Collections` and `Arrays` Classes

- `Collections` utility class contains methods to modify `Collection` class objects
  - sort elements
  - search for elements
  - reverse elements
  - provide synchronized access and read-only collections
  - randomize (shuffle) elements
  - etc. (study the javadoc)
- `Arrays` utility class has many of the same methods for arrays

Strategy Patterns

Strategy: *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.*

Strategy Pattern
Applicability

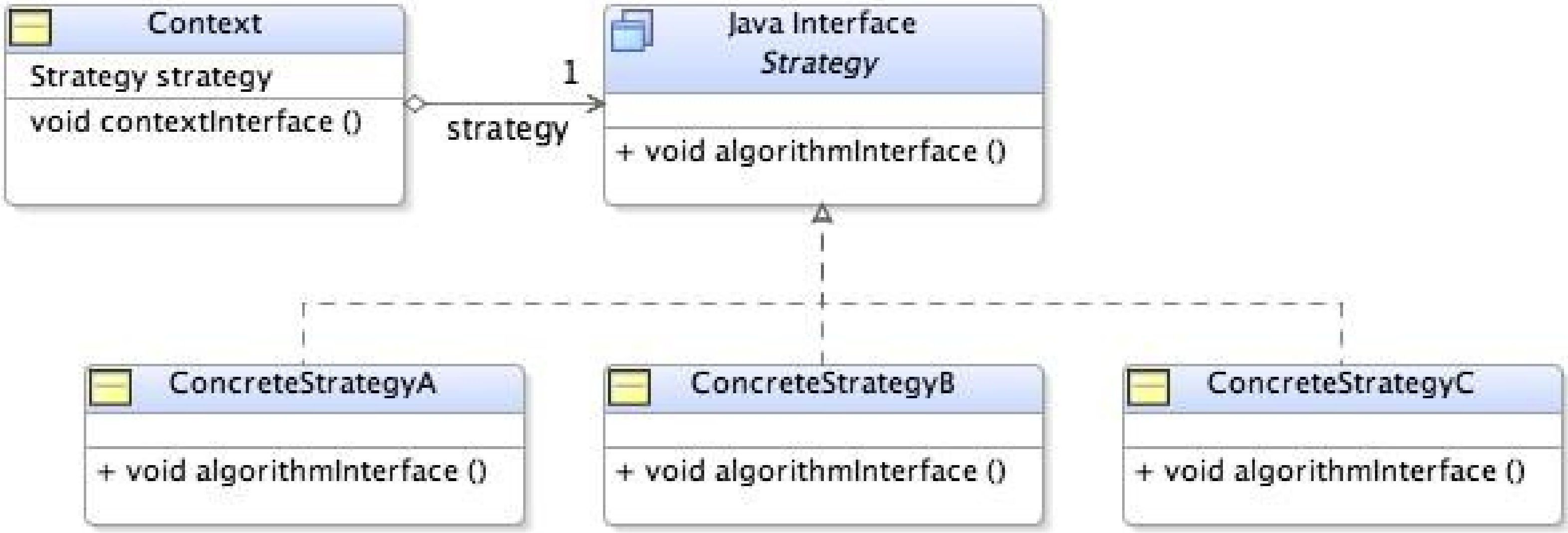
- Use the strategy pattern when:
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - Different variants of an algorithm are needed. For example, algorithms might be defined reflecting different space/time trade-offs.
  - An algorithm uses data that the client shouldn’t know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

Strategy Pattern

Applicability

- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many, conditionals move related conditional branches into their own Strategy class.

Strategy Pattern



Strategy Pattern
Responsibilities

- Strategy
  - Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy subclasses
  - Implements the algorithm using the Strategy interface.

Strategy Patterns

Responsibilities

- Context
  - Is configured with a ConcreteStrategy object
  - Maintains a reference to a Strategy object.
  - May define an interface that lets Strategy access its data.

Comparing Objects

- `java.lang.Comparable` interface
  - Designed to sort objects into “natural ordering”
  - Implemented by String and wrapper classes
  - Requires a single method:

    ```
    int compareTo(<T> other);
    ```
  - Returns negative int, 0, or positive int based on whether this object is less than, equal to , or greater than the other object

Comparing Objects (cont'd)

- Sometimes "natural ordering" isn't enough; there may be several useful ways to sort
- In these cases, implement the `java.util.Comparator` interface in a class separate from the class you want to compare
- Two methods:
  - `int compare(<T> first, <T> second)`
    - Just like `compareTo()`, returns negative int, 0, or positive int to indicate order
  - `boolean equals()` // optional
    - `Object.equals()` is used if you don't implement this method

Example

```
import java.util.Comparator;

public class IntegerComparator implements Comparator<Integer> {
    public IntegerComparator() {
    }



    {
        int diff = 0;
        if (arg0.intValue() > arg1.intValue()) {






    }
}
```

```
        return diff;
        }
```