Outline

• Overview
• Subtyping and Wildcard
• Comparison and Bounds
• Declaration and Erasure
• Reification and Reflection

Collections
  • Iterator, Iterable, Collection
  • Set, Queues, List, Maps
• Design Patterns
• Other Issues

Main Interfaces for Collections

Iterables to support foreach
Set is a collection without duplicates
  navigableSet find closest element match
Queue is a collection with tail/head
  Deque supports double-ended processing
  BlockingQueue supports double-ended accesses
List is a collection with position order and duplicates

Map is a collection with key, value pair
  ConcurrentHashMap to support concurrency
  SortedMap returns in ascending key order
  NavigableMap allows closest match return

Collections

Main Interfaces for Collections

- Iterable<T> (java.lang)
- Collection<E>
  - Set<E>
  - List<E>
  - Queue<E>
  - SortedSet<E>
  - Deque<E>
  - BlockingQueue<E>
  - NavigableMap<K,V>
- SortedMap<K,V>
- Map<K,V>
- BlockingDeque<E>
- ConcurrentHashMap<K,V>

Iterator/Iterable

To support sequential access to a collection.

```java
class Counter implements Iterable<Integer> {
    private int count;
    public Counter(int count) { this.count = count; }
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            private int i = 0;
            public boolean hasNext() { return i < count; }
            public Integer next() { i++; return i; }
            public void remove() { throw new UnsupportedOperationException(); }
        };
    }
}
```
### Fail-Fast Iterators

Policy of Java 2 is to **fail fast** if structural modification made by sequencing through an iterator. It throws `ConcurrentModificationException`.

**Synchronized Wrapper for ArrayStack**

```java
public synchronized void push(int elt) { .. }
public synchronized int pop() { .. }

if (!stack.isEmpty()) {
    stack.pop();  // can throw exception
}
```

**Solution**

```java
synchronized(stack) { if (!stack.isEmpty()) {
    stack.pop();
}
```

---

### Collections

Core functionalities:
- Adding elements
- Removing elements
- Querying the contents
- Making contents for further processing.

**Adding elements:**

```java
boolean add( E o )  // add the element o
boolean addAll( Collection<? extends E> c )  // add the contents of c
```

**Removing elements:**

```java
boolean remove( Object o )  // remove the element o
void clear()  // remove all elements
boolean removeAll( Collection<? c )  // remove the elements in c
boolean retainAll( Collection<? c )  // remove the elements «not» in c
```

**Querying the contents of a collection:**

```java
boolean contains(Object o)  // true if o is present
boolean containsAll(Collection<? c)  // true if all elements of c are present
boolean isEmpty()  // true if any elements are present
int size()  // returns the element count (or Integer.MAX_VALUE if that is less)
```

**Making a collection’s contents available for further processing:**

- `toArray()`  // copies contents to an Object[]
- `<T> T[] toArray( T[] t )`  // copies contents to a T[] (for any T)

---

### Sets

Cannot contain duplicates:
- Adding an element that already exists has no effect

Different implementation of Set

---

---
**HashSet**

Most common implementation of set via a hash table

**CopyOnWriteSet**

The array it uses is immutable with each change producing a new copy.

Add an element has complexity O(n).

Iteration costs O(1) per element.

Provides thread safety through immutable array implementation without the need for locking.

**LinkedHashSet**

It inherits from HashSet and adds a guarantee that its iterators will return elements in the same order as was added.

**SortedSet**

Allows traversal of set in ascending order.

Implementation before Java 6 is known as TreeSet.

Retrieving the comparator:

```java
Comparator<? super E> comparator()
```

Getting the head and tail elements:

```java
E first()
E last()
```
**Sorted Set**

Finding subsequences:

```
SortedSet<E> subset( E fromElement, E toElement )
SortedSet<E> headSet( E toElement )
SortedSet<E> subset( E fromElement )
```

Inclusive of the fromElement but exclusive of the toElement.

```
Task highestMediumPriorityTask = new Task( "\0", Priority.MEDIUM );
SortedSet<Task> highPriorityTasks = priorityOrderedTasks.headSet( highestMediumPriorityTask );
```

Sets returned are view of the original set and not new sets.

**Navigable Set**

Navigating Set in Reverse Order

```
NavigableSet<E> descendingSet()
    // returns a reverse order view for this

Iterator<E> descendingIterator()
    // return a reverse-order iterator
```

**Navigable Set**

Introduced in Java 6 to supplement deficiencies

Getting the First and Last Elements

```
E pollFirst()    // retrieve and remove smallest element
E pollLast()     // retrieve and remove largest element
```

Getting Closest Matches

```
E ceiling(E e)   // returns least element greater or equal to e
E floor(E e)     // return greatest element less or equal to e
E higher(E e)    // returns least element greater than e
E lower(E e)     // returns greatest element lower than e
```

**Tree Set**

TreeSet can be implemented using balanced tree to support fast insert, remove and lookup.

However, it is unsynchronized and not thread-safe.

TreeSet uses a red/black tree as its underlying implementation.

Its iterators are fail-fast.
**ConcurrentSkipListSet**

ConcurrentSkipListSet was introduced as in Java 6 as the first concurrent set implementation.

It is backed by *skip list* as an alternative to binary tree.

Each level in the skiplist contains about half of the elements below.

For insertion, each element must be inserted at level 0. Probability is used to decide if an insertion is to be added at the next level.

This has *lock-free* insert/delete concurrent algorithms.

---

**Queue**

Organises elements on a FIFO basis.

The method add is inherited from Collection and may throw an exception if bounded queue is full. A false is only returned if the same element was already present. **offer** method does not throw exception but return false instead.

**Blocking Queue**

Used in a producer/consumer setting, e.g. print spooling with clients and servers.

Adding an element:

```
boolean offer(E e, long timeout, TimeUnit unit) // inserts e, waiting up to the timeout
void put(E e) // adds e, waiting indefinitely if necessary
```

---

**Retrieving Elements from Queue**

The two methods inspect/retrieve the head element but may throw an exception for empty queue:

```
E element() // retrieves but does not remove the head element
E remove() // retrieves and removes the head element
```

Two methods inspect/retrieve the head element but returns null for empty queue:

```
E peek() // retrieves but does not remove the head element
E poll() // retrieves and removes the head element
```
### Blocking Queue

Removing an element.

```java
E poll(long timeout, TimeUnit unit)
    // retrieves and removes the head, waiting up to the timeout
E take()
    // retrieves and removes the head of this queue, waiting
    // indefinitely if necessary.
```

Retrieving or querying contents:

```java
int drainTo(Collection<? super E> c)
    // clears the queue into c
int drainTo(Collection<? super E> c, int maxElements)
    // clears at most specified number of elements into c
int remainingCapacity()
    // returns the number of elements that would be accepted
    // without blocking
```

### Lists

A collection that can contain duplicates and with elements fully visible.

#### List<E>

- `add(index : int, element : E) : boolean`
- `addAll(index : int c : Collection<? extends E>) : boolean`
- `indexOf(Object o) : int`
- `lastIndexOf(Object o) : int`
- `listIterator(index : int) : ListIterator<E>`
- `listIterator() : ListIterator<E>`
- `set(index : int, element : E) : E`
- `remove(index : int) : E`
- `get(index : int) : E`
- `subList(fromIndex : int, toIndex : int) : List<E>`

### Lists

Positional accesses.

```java
void add(int index, E element) // add element o at given index
boolean addAll(int index, Collection<? extends E> c) // add contents of c at given index
E get(int index) // return element with given index
E remove(int index) // remove element with given index
E set(int index, E element) // replace element with given index
```

Search for specified element (-1 if not found)

```java
int indexOf(Object o) // return index of first occurrence of o
int lastIndexOf(Object o) // return index of last occurrence of o
```

Returns a view of a subrange of the list

```java
List<E> subList(int fromIndex, int toIndex) // return a view of a portion of the list
```
Lists

List Iteration.

```java
class ListIterator<E> {
    // returns a ListIterator for this list
    public ListIterator() {
    }
    // returns a ListIterator for this list,
    // starting at the given index
    public ListIterator(int index) {
    }
}
```

List iterator has with extra methods

```java
public interface ListIterator<E> extends Iterator<E> {
    void add(E e); // Inserts the specified element into the list
    void remove();  // Removes the previous element from the list.
    boolean hasNext(); // Returns true if this list iterator has elements following it.
    int nextIndex(); // Returns the index of the element that would be returned by the next call.
    E previous(); // Returns the previous element in the list.
    int previousIndex(); // Returns the index of the element that would be returned by the previous call.
    void set(E e);  // Replaces the previous element returned by next or previous with the specified element.
}
```

Maps

This interface does not inherit from `Collection`.

```java
interface Map<K, V> {
    +clear();
    +containsKey( key : Object ) : boolean
    +containsValue( value : Object ) : boolean
    +entrySet() : Set<Map.Entry<K, V>>
    +get( key : Object ) : boolean
    +isEmpty() : boolean
    +keySet() : Set<K>
    +put( key : K, value : V ) : V
    +putAll( t : Map<? extends K, ? extends V> )
    +remove( key : Object ) : V
    +size() : int
    +values() : Collection<V>
}
```

List Implementation

`CopyOnWriteArrayList` is a kind of functional array.

Maps

Adding associations.

```java
V put( K key, V value ); // adds or replaces a key-value association.
                  // key was present, otherwise returns null
void putAll(Map<? extends K, ? extends V> t); // puts each of the key-value associations in the supplied map into the receiver
```

Removing associations.

```java
void clear(); // removes all associations from this map
V remove(Object key); // removes the association, if any, with the given key; returns the value with which it was associated, or null
```
Maps

Querying Contents of Map.

boolean containsKey(Object k) // returns true if k is present as a key. k may be null // for maps that allow null keys.
boolean containsValue(Object v) // returns true if v is present as a value. v may be null.
V get(Object k) // returns the value corresponding to k, or null if k is // not present as a key.

Providing Collection Views.

Set<Map.Entry<K, V>> entrySet() // returns a set view of the key-value associations.
Set<K> keySet() // returns a set view of the keys
Collection<V> values() // returns a set view of the values

Implementing Map

Eight different implementations.

IdentityHashMap : two keys equal only if same object

EnumMap : key is from an Enum class

three public constructors:

EnumMap(Class<K> keyType)
   // create an empty enum map
EnumMap(EnumMap<K<? Extends V> m)
   // create by taking from m
EnumMap(Map<K,? Extends V> m)
   // create by taking from m

HashMap : loadFactor to determine rehashing
LinkedHashMap – guarantees same order as insertion for iterator
WeakHashMap – uses weak references so that some objects can be garbage collected earlier

Implementing Map

Figure 13.3: HashMap and WeakHashMap
Retrieving the comparator:

```java
Comparator<? super K> comparator()
```

Getting first and last elements:

```java
K firstKey()
K lastKey()
```

Finding subsequences:

```java
SortedMap<K,V> subMap( K fromKey, K toKey )
SortedMap<K,V> headMap( K toKey )
SortedMap<K,V> tailMap( K fromKey )
```

Other Kinds of Maps

**NavigableMap**
- getting closest matches
- allows navigation of the Map via `NavigableSet`

**TreeMap**
- `SortedMap` is implemented by `TreeMap`

**ConcurrentMap**
- for high performance server application
- four atomic operations

Outline

- Overview
- Subtyping and Wildcard
- Comparison and Bounds
- Declaration and Erasure
- Reification and Reflection
- Collections
  - Iterator, Iterable, Collection
  - Set, Queues, List, Maps
- Design Patterns
- Other Issues

Design Patterns

Several well-known design patterns:

- Visitor
- Interpreter
- Function
- Strategy
- Subject-Observer

These patterns can take advantage of generics.
**Visitor Pattern**

Abstract class of tree is `Tree<E>` where `E` is the element type.

Tree traversal can be done in the abstract class, using specialised methods, such as `toString` and `sum`.

However, we may like to provide flexibility to client codes to add more of such traversal operations.

Thus, we provide an abstract interface `Visitor<E,R>` which has two methods:

- `leaf` — accepts a value of type `E` to return a value of `R`
- `branch` — accepts two values of type `R` to return a value of `R`

**Client Code for Tree Visitor**

class TreeClient {
    public static <T> String toString(Tree<T> t) {
        return t.visit(new Tree.Visitor<T,String>() {
            public String leaf(T e) {
                return e.toString();
            }
            public String branch(String l, String r) {
                return "(" + l + "\^" + r + ")";
            }
        })
    }
    public static <N extends Number> Double sum(Tree<N> t) {
        return t.visit(new Tree.Visitor<N, Double>() {
            public Double leaf(N e) {
                return e.doubleValue();
            }
            public Double branch(Double l, Double r) {
                return l + r;
            }
        })
    }
}

**Interpreter Pattern**

Possible to use trees to represent expressions.

For example:

- `Exp<Integer>` is an expression that returns integer
- `Exp<Pair<Integer,Integer>>` is an expression that returns a pair of integers.

Generic interpreter pattern shows that generics in Java can be more powerful than parameterised type of other languages as its capability is similar to GADT.
**Pair Class**

class Pair<A,B> {
    private final A left;
    private final B right;
    public Pair(A I, B r) {left=l; right=r; }
    public A left() { return left; }
    public B right() { return right; }
}

**GADT (Diversion)**

```
same as Term :: * -> *
```

data Term a where
    Lit :: Int -> Term Int
    Inc :: Term Int -> Term Int
    IsZ :: Term Int -> Term Bool
    If :: Term Bool -> Term a -> Term a -> Term a
    Pair :: Term a -> Term b -> Term (a,b)
    Fst :: Term (a,b) -> Term a
    Snd :: Term (a,b) -> Term b

Note that a has different types depending on the data constructor.

**Exp Class**

abstract class Exp<T> {
    abstract public T eval();

    static Exp<Integer> lit(final int i) {
        return new Exp<Integer>() {
            public Integer eval() { return I; }
        };
    }

    static Exp<Integer> plus(final Exp<Integer> e1,
        final Exp<Integer> e2) {
        return new Exp<Integer>() {
            public Integer eval() {
                return I;
            }
        };
    }

    static <A,B> Exp<Pair<A,B>> pair(final Exp<A> e1,
        final Exp<B> e2) {
        return new Exp<Pair<A,B>>() {
            public Pair<A,B> eval() {
                return (a,b);
            }
        };
    }

    ...
}

**GADT (Haskell Diversion)**

```
same as Term :: * -> *
```

data Term a where
    Lit :: Int -> Term Int
    Inc :: Term Int -> Term Int
    IsZ :: Term Int -> Term Bool
    If :: Term Bool -> Term a -> Term a -> Term a
    Pair :: Term a -> Term b -> Term (a,b)
    Fst :: Term (a,b) -> Term a
    Snd :: Term (a,b) -> Term b

Note that a has different types depending on the data constructor.
**GADT (Haskell Diversion)**

An interpreter in Haskell with precise type information.

```haskell
eval :: Term a -> a
eval (Lit i) = i
eval (Inc t) = eval t + 1
eval (IsZ t) = eval t == 0
eval (If b t e) = if eval b then eval t else eval e
eval (Pair a b) = (eval a, eval b)
eval (Fst t) = fst (eval t)
eval (Snd t) = snd (eval t)
```

Type refinement in pattern-matching. No runtime type passing

---

**Function**

Some interesting methods:

- `applyAll` accepts a `List<A>` to return a `List<B>` as result; it may throw an exception of type `X`
- `main` method defines three objects:
  - `length` of a given `String`
  - `forName` which returns a `Class` for a given `name`
  - `getRunMethod` to return a method named `run` for a class of a given `name`.

```java
class Functions {
    public <A,B,X extends Throwable> List<B> applyAll(List<A> list) throws X {
        List<B> result = new ArrayList<B>(list.size());
        for (A x : list) result.add(apply(x));
        return result;
    }
    public static void main(String[] args) {
        Function<String,Integer,Error> length =
            new Function<String,Integer,Error>() {
                public Integer apply(String s) { return s.length(); }
            };
        Function<String,Class<?>,ClassNotFoundException> forName = ...
        Function<String,Method,Exception> getRunMethod = ...
    }
}
```
**Strategy Pattern**

Used to decouple a method from an object, allowing many possible instances of the method.

Supports parallel class hierarchy, e.g.
- hierarchy of tax payers
- hierarchy of tax strategies

This provides for a generic reflection mechanism.

```
abstract class TaxPayer {
    public long income; // in cents
    public TaxPayer(long income) {this.income = income;}
    public long getIncome() {return income;}
}
```

class Person extends TaxPayer {
    public Person(long income) { super(income); }
}

class Trust extends TaxPayer {
    private boolean nonProfit;
    public Trust(long inc, boolean np) {
        super(inc); this.nonProfit = np;
    }
    public boolean isNonProfit() { return nonProfit; }
}

```
interface TaxStrategy<P extends TaxPayer> {
    public long computeTax(P p);
}
```

class DefaultTaxStrategy<P extends TaxPayer> implements TaxStrategy<P> {
    private static final double RATE = 0.40;
    public long computeTax(P payer) {
        return Math.round(payer.getIncome() * RATE);
    }
}

class DodgingTaxStrategy<P extends TaxPayer> implements TaxStrategy<P> {
    public long computeTax(P payer) { return 0; }
}

class TrustTaxStrategy<P extends TaxPayer> implements DefaultTaxStrategy<P> {
    public long computeTax(Trust t) {
        return t.isNonProfit() ? 0 : super.computeTax(t);
    }
}

**More Advanced Strategy**

Object may also contain strategy to be applied.

Requires recursive bounds and also a special `getThis`.

```
abstract class TaxPayer<P extends TaxPayer<P>> {
    public long income; // in cents
    public TaxPayer(long income) { super(income); }
}
```

class Person extends TaxPayer {
    public Person(long income) { super(income); }
}

class Trust extends TaxPayer {
    private boolean nonProfit;
    public Trust(long inc, boolean np) {
        super(inc); this.nonProfit = np;
    }
    public boolean isNonProfit() { return nonProfit; }
    protected abstract P getThis();
    public long getIncome() { return income; }
    public long computeTax() {
        return strategy.computeTax(getThis());
    }
}
Why getThis needed?

this has type TaxPayer<P>

computeTax requires type P

class DefaultTaxStrategy<P extends TaxPayer<P>>
    implements TaxStrategy<P>
    {
        private static final double RATE = 0.40;
        public long computeTax(P payer)
            {
                return Math.round(payer.getIncome() * RATE);
            }
    }

Above seems like a hack for acceptable typing! Better solution?

Subject-Observer Pattern

Parallel class hierarchies that are mutually dependent.

public class Observable<S extends Observable<S,O,A>,
    O extends Observer<S,O,A>,A>

public interface Observer<S,O,A>
    {
        public void update(S o, A a);
    }

Foreign Currency Views

enum Currency {DOLLAR, EURO, POUND}
class CModel extends
    Observable<CModel, CView,Currency>
interface CView extends
    Observer<CModel, CView,Currency>
class RateView extends JTextField
    implements CView
class ValueView extends JTextField
    implements CView

Observable<CModel,CView,Currency> Observer<CModel,CView,Currency>

CModel

RateView

ValueView
Outline

- Overview
- Subtyping and Wildcard
- Comparison and Bounds
- Declaration and Erasure
- Reification and Reflection
- Collections
  - Iterator, Iterable, Collection
  - Set, Queues, List, Maps
- Design Patterns
- Other Issues

Generic Algorithms

Finding extreme elements

```java
static <T extends Object & Comparable<? super T>> T
    max(Collection<? extends T> coll)
        // returns the maximum element using natural ordering
static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
        // returns the maximum element using the supplied comparator
static <T extends Object & Comparable<? super T>> T
    min(Collection<? extends T> coll)
        // returns the minimum element using natural ordering
static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
        // returns the maximum element using the supplied comparator
```

Generic Algorithms

Changing the order of list elements.

```java
static void reverse(List<?> list)
    // reverses the order of the elements
static void rotate(List<?> list, int distance)
    // rotates the elements of the list; the element at index
    // i is moved to index (distance + i) % list.size()
static void shuffle(List<?> list)
    // randomly permutes the list elements
static void shuffle(List<?> list, Random rand)
    // randomly permutes the list using the randomness source rnd
static <T extends Comparable<? super T>> void sort(List<T> list)
    // sorts the supplied list using natural ordering
static <T> void sort(List<T> list, Comparator<? super T> c)
    // sorts the supplied list T using the supplied ordering
static void swap(List<?> list, int i, int j)
    // swaps the elements at the specified positions
```

Generic Algorithms

Finding specific values

```java
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
    // searches for key using binary search
static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
    // searches for key using binary search
static int indexOfSubList(List<?> source, List<?> target)
    // finds the first sublist of source which matches target
static int lastIndexOfSubList(List<?> source, List<?> target)
    // finds the last sublist of source which matches target
```
**Generic Algorithms**

Changing contents

```java
static <T> void copy(List<? super T> dest, List<? extends T> src) {
    // copies all of the elements from one list into another
}
static <T> void fill(List<? super T> list, T obj) {
    // replaces every element of list with obj
}
static <T> boolean replaceAll(List<? super T> list, T oldVal, T newVal) {
    // replaces all occurrences of oldVal in list with newVal
}
```

---

**Unchecked Warnings**

Casting to type parameters and complex types not supported:

```java
// unsafe unchecked cast - not recommended
List<Track> trackList = (List<Track>) query.list();
Collections.sort(trackList);
```

Creating an array of non-reifiable type.

```java
// creating array of non-reifiable type - not recommended
class VarArgsProblem {
    public <T> List<T>[] createArray ( List<T>... lsa ) { return lsa; }
    public static void main(String[] args) {
        List<String>[] lsa =
            new VarArgsProblem().createArray(new ArrayList<String>());
    }
}
```

---

**Unchecked Warnings**

Cast eliminated are mostly safe except for unchecked warnings.

Note: MyClass.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

Option `-Xlint:unchecked` will allow report details of type insecurity.

An example is missing type parameter causing raw type.

```java
// omitting type parameter from instance creation - not recommended
List l = new ArrayList();
// ...
1.add("abc"); // unchecked call
```