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

`Iterable` 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 concurrent accesses

`List` is a collection with position order and duplicates

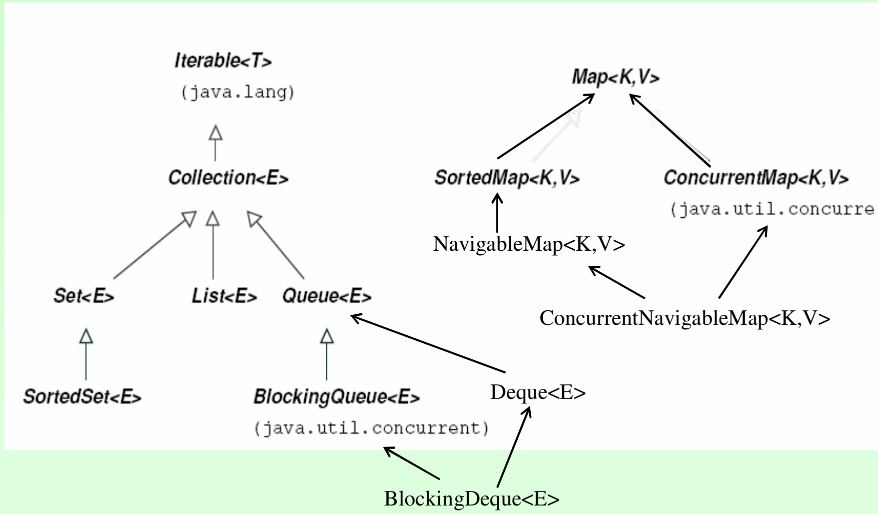
`Map` is a collection with `key,value` pair

`ConcurrentMap` to support concurrency

`SortedMap` returns in ascending key order

`NavigableMap` allows closest match return

Main Interfaces for Collections



Iterator/Iterable

To support sequential access to a collection.


```
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`

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

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

Solution

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


Collections

Core Functionalies :

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

Adding elements:

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

Removing elements:

```
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
```


Collections

Querying the contents of a collection:

```
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:

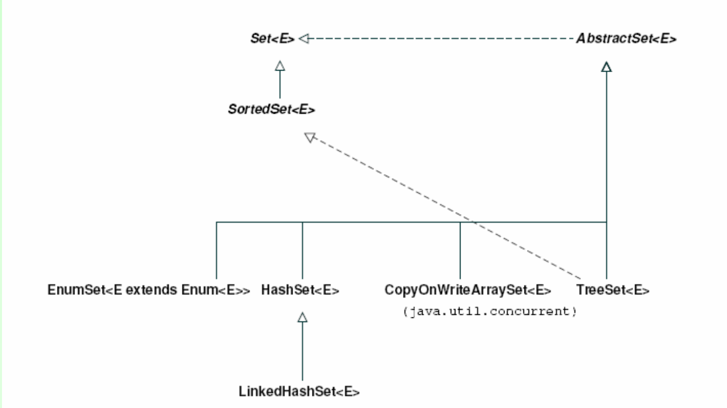
```
Iterator iterator()          // returns an Iterator over the elements
Object[] toArray()           // copies contents to an Object[]
<T> T[] toArray( T[] t )     // copies contents to a T[] (for any T)
```


Sets

Cannot contain duplicates :

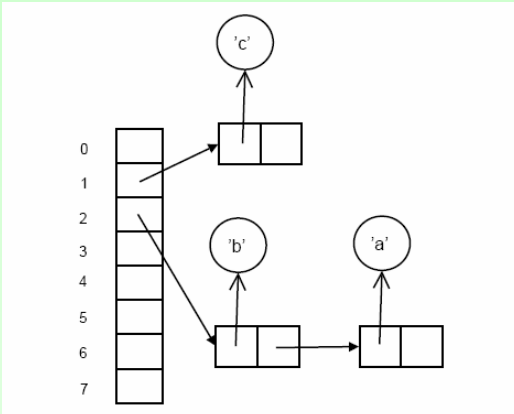
- Adding an element that already exists has no effect

Different implementation of Set



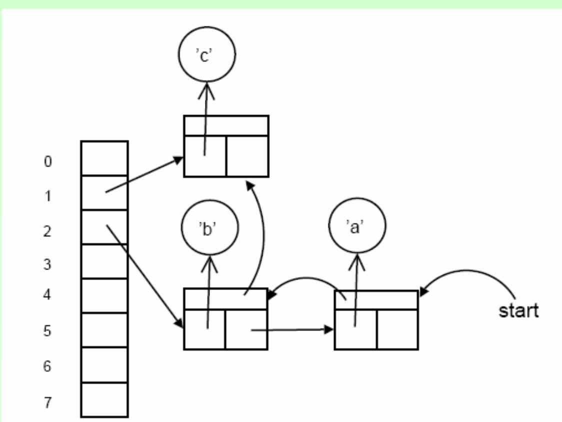
HashSet

Most common implementation of set via a hash table



LinkedHashSet

It inherits from HashSEt and add a guarantee that its iterators will return elements in same order as was added.



CopyOnWriteSet

The array it uses ia 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.

SortedSet

Allows traversal of set in ascending order.

Implementation before Java 6 is known as `TreeSet`.

Retrieving the comparator:

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

Getting the head and tail elements:

```
E first()
E last()
```

SortedSet

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

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
```


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
```


TreeSet

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.

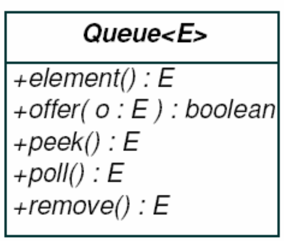
For insertion, each element must be inserted at level 0. Probability is used to decide if an insertion is to be added at the next level.

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.



Retrieving Elements from Queue

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

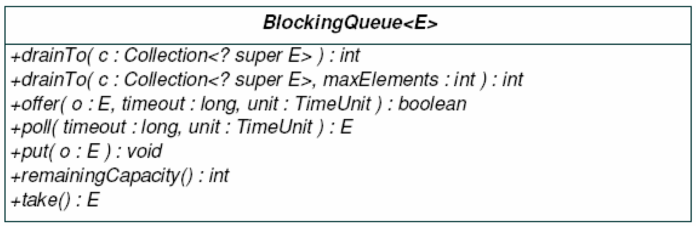
```
E element()  // retrieves but does not remove the head element
E remove()   // retrieves and removes the head element
```

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

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



Adding an element:

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

Blocking Queue

Removing an element.

```
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:

```
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
```


Blocking Queue

Five implementations :

LinkedBlokcingQueue
ArrayBlockingQueue
PriorityBlockingQueue
DelayBlockingQueue (based on delay time)
SynchronousQueue (no internal space)

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( o : Object ) : int
+lastIndexOf( o : Object ) : 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.

```
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)

```
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

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

Lists

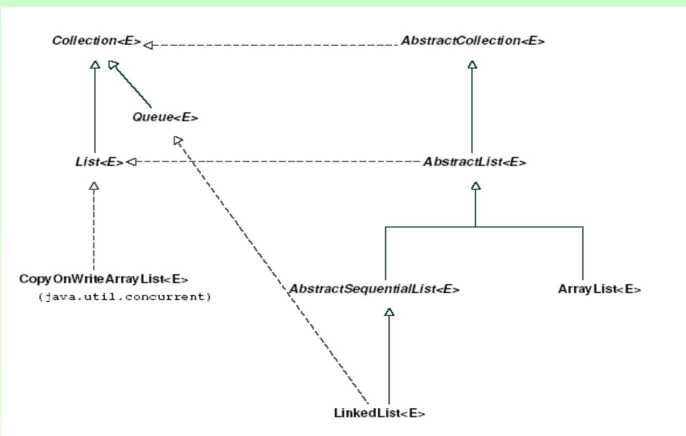
List Iteration.

```
ListIterator<E> listIterator()          // returns a ListIterator for this list
ListIterator<E> listIterator( int index) // returns a ListIterator for this list,
                                         //   starting at the given index
```

List iterator has with extra methods

```
public interface ListIterator<E> extends Iterator<E> {
  void add(E o);          // Inserts the specified element into the list
  boolean hasPrevious();  // Returns true if this list iterator has further
                          //   elements in the reverse direction.
  int nextIndex();        // Returns the index of the element that would be
                          // returned by a subsequent call to next.
  E previous();           // Returns the previous element in the list.
  int previousIndex();    // Returns the index of the element that would be
                          // returned by a subsequent call to next
  void set(E o);          // Replaces the last element returned by next or
                          //   previous with the specified element
}
```


List Implementation



`CopyOnWriteArrayList` is a kind of functional array.

Maps

This interface does not inherit from `Collection`.

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>

Maps

Adding associations.

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

Removing associations.

```
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.

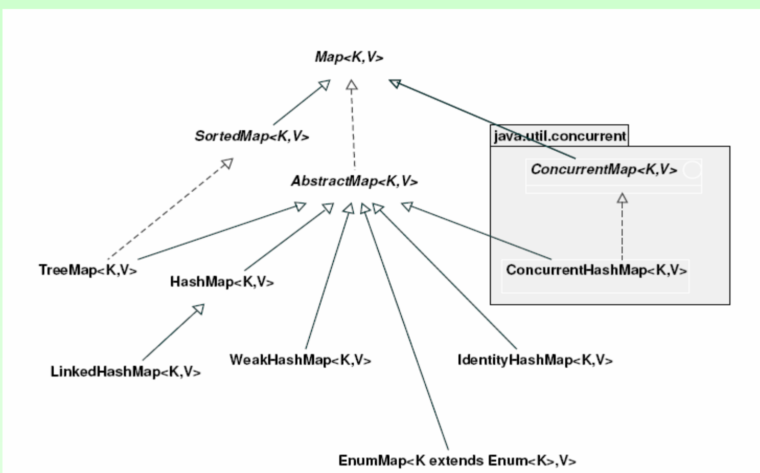
```
boolean containsKey(Object k)    // returns true if k is present as a key. k may be null
                                 // for maps that allow null keys.
boolean containsValue(Object v)  // returns true if v is present as a value. v may be null.
V get(Object k)                  // returns the value 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.



Implementing Map

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

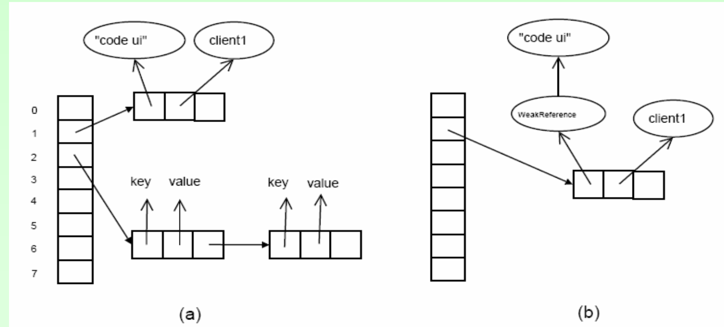


Figure 13.3: HashMap and WeakHashMap

Implementing Map

IdentityHashMap : two keys equal only if same object

EnumMap : key is from an Enum class
three public constructors :

```
EnumMao(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
```

SortedMap

Retrieving the comparator :

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

Getting first and last elements :

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

Finding subsequences :

```
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

```
V putIfAbsent(K key, V value)
        // associate key with value only if key is not currently present
boolean remove(Object key, Object value)
        // remove key only if it is currently mapped to value.
V replace(K key, V value)
        // replace entry for key only if it is currently present
boolean replace(K key, V oldValue, V newValue)
        // Replace entry for key only if it is currently mapped to oldValue
```


Outline

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`

Visitor Pattern

```
abstract class Tree<E> {

public interface Visitor<E,R> {
 public R leaf(E elt);
 public R branch(R left, R right);
}

public abstract <R> R visit(Visitor<E,R> v);

public static <T> Tree<T> leaf(final T e) {
   return new Tree<T>() { … }

public static <T> Tree<T> branch(final Tree<T> l,
  final Tree<T>r) {
   return new Tree<T>() { … }
```


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; }

}
```


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() {…}}

 static <A,B> Exp<Pair<A,B>> pair(final Exp<A> e1,
    final Exp<B> e2)
      { return new Exp<Pair<A,B>>() { …. }

  …
}
```


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.

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.

```
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 Pattern

Allows an arbitrary method to be converted to an object.

Class Function<A,B,X> has an abstract method

```
interface Function<A,B,X extends Throwable> {
  public B apply(A x) throws X;
 }
```

This provides for a generic reflection mechanism.

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.

Function

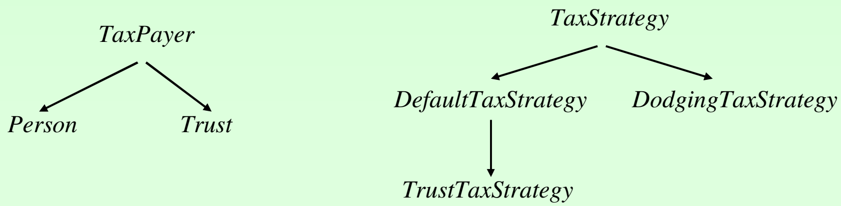
```
class Functions {
 public <A,B, X extends Throwable> List<B>
   applyAll(List<A> list) throws X {
     List<B> result = new ArrayList<B<(list.size());
     for (A x : list) result.add(apply(x));
     return result;
 }
 public static void main(String[] args) {
   Function<String,Integer,Error> length =
     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.



Strategy

```
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;}
}
```


Strategy

```
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
    trust.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
 private TaxStrategy<P> strategy;
 public TaxPayer(long income,
   TaxStrategy<P> strategy)
   {this.income=income; this.strategy=strategy; }
 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 hierachies that are mutually dependent.

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

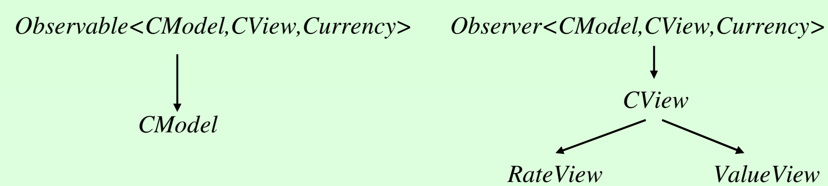

Subject-Observer

```
public class Observable<… >
{
  public void addObserver(O o) { … }
  protected void clearChanged()  { … }
  public int countObserver() { … }
  public void deleteObserver(O o) { … }
  public boolean hasChanged() { … }
  public void notifyObserver() { … }
  public void notifyObserver(A a) { … }
  protected void setChanged()  { … }
}

public interface Observer<…>
{
  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 { …
```



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

Changing the order of list elements.

```
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 rnd)
        // 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 extreme elements

```
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

Finding specific values

```
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

```
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<T> list, T oldVal, T newVal)
         // replaces all occurrences of oldVal in list with newVal
```


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.

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


Unchecked Warnings

Casting to type parameters and complex types not supported:

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

Creating an array of non-reifiable type.

```
// 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>());
  }
}
```

```
VarArgsProblem.java:5: warning: [unchecked] unchecked generic array
creation of type java.util.List<java.lang.String>[] for varargs parameter
```