



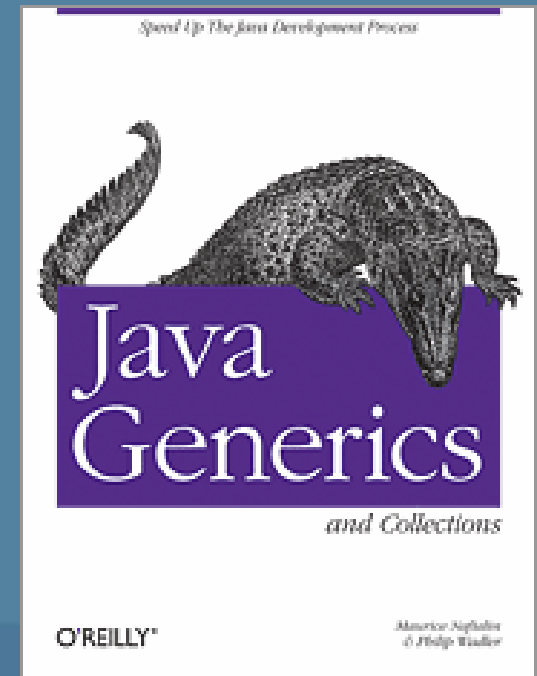
JavaOne

Java™ Generics and Collections: Tools for Productivity

Maurice Naftalin,
Morningside Light Ltd

Philip Wadler,
University of Edinburgh

TS-2890



The Right Tools for the Job

What you can – and can't! – do with the Generics and Collections features introduced in Java™ 2 Platform, Standard Edition 5.0 (J2SE™ 5.0 platform) and Java Platform, Standard Edition 6 (Java SE 6 platform)

Agenda

Generics

- Why have them?

- Implementation by erasure – benefits ...

- ... and problems

- What next?

Collections

- Trends in concurrency policy

- Trends in API design

- How to choose an implementation

Generics

Generics

Why have them?

Implementation by erasure – benefits ...

... and problems

What next?

Collections

Trends in concurrency policy

Trends in API design

How to choose an implementation

Cleaner code

Before:

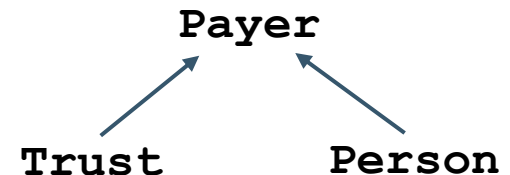
```
List ints = Arrays.asList(1,2,3);  
int s = 0;  
for (Iterator it = ints.iterator(); it.hasNext();) {  
    s += (Integer) it.next();  
}
```

After:

```
List<Integer> ints = Arrays.asList(1,2,3);  
int s = 0;  
for (int n : ints) { s += n; }
```

Detect more errors at compile-time

Strategy pattern for paying tax:



```
interface Strategy<P extends Payer>{ long computeTax(P p); }  
class DefaultStrategy<P extends Payer>  
    implements Strategy<P> { long computeTax(P p){...} }  
class TrustTaxStrategy extends DefaultStrategy<Trust> {  
    public long computeTax(Trust t) {  
        return trust.isNonProfit() ? 0 : super.computeTax(t);  
    }  
}
```

**new TrustTaxStrategy().computeTax(person)
fails at compile time with generics**

Detect more errors at compile-time

`ArrayStoreExceptions` become compile errors

- Arrays:

```
Integer[] ints = new Integer[]{1,2,3}
Number[] nums = ints;
nums[2] = 3.14;           // run-time error
```

`Integer[]` **is** a subtype of `Number[]`

- Collections:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<Number> nums = ints; // compile-time error
nums.put(2, 3.14);
```

`List<Integer>` **is not** a subtype of `List<Number>`

More Expressive Interfaces

From `javax.management.relation.Relation`

- Before

```
interface Relation {  
    public Map getReferencedMBeans()  
    ...  
}
```

- After

```
interface Relation {  
    public Map<ObjectName, List<String>>  
        getReferencedMBeans()  
    ...  
}
```

Explicit types in client code – much easier to maintain

Generics

Generics

Why have them?

Implementation by erasure – benefits ...

... and problems

What next?

Collections

Trends in concurrency policy

Trends in API design

How to choose an implementation

Migration Compatibility

Major design constraint for generics: *Binary for legacy client must link to generified library*

With erasure:

Generified
library binary = Legacy
library binary

Allows piecewise generification of libraries

Erasure Eases Evolution

From Legacy...

Library

```
interface Stack {  
    void push(Object elt);  
    Object pop();  
}  
  
class ArrayStack implements Stack {  
    private List li = new ArrayList();  
    public void push(Object elt) { li.add(elt); }  
    public Object pop(){ return li.remove(li.size()-1); }  
}
```

Client

```
Stack stack = new ArrayStack();  
stack.push("first");  
String top = (String)stack.pop();
```

...to Generic

Library

```
interface Stack<E> {  
    void push(E elt);  
    E pop();  
}  
  
class ArrayStack<E> implements Stack<E> {  
    private List<E> li = new ArrayList<E>();  
    public void push(E elt) { li.add(elt); }  
    public E pop() { return li.remove(li.size()-1); }  
}
```

Client

```
Stack<String> stack = new ArrayStack<String>();  
stack.push("first");  
String top = stack.pop();
```

Generic Library with Legacy Client

Library

```
interface Stack<E> {  
    void push(E elt);  
    E pop();  
}  
  
class ArrayStack<E> implements Stack<E> {  
    private List<E> li = new ArrayList<E>();  
    public void push(E elt) { li.add(elt); }  
    public E pop() { return li.remove(li.size()-1); }  
}
```

Client

```
Stack stack = new ArrayStack();  
stack.push("first");           // unchecked call  
String top = (String)stack.pop();
```

Legacy Library with Generic Client

Three options

- Minimal changes (surface generification)
- Stubs
- Wrappers - not recommended!

Minimal Changes

Library with “Surface Generification”

```
class ArrayStack<E> implements Stack<E> {  
    private List li = new ArrayList();  
    public void push(E elt){li.add(elt);} //unchecked call  
    public E pop(){  
        return (E)li.remove(li.size()-1); //unchecked cast  
    }  
}
```

Stubs

Stubs

```
class ArrayStack<E> implements Stack<E> {  
    public void push(E elt) { throw new StubException(); }  
    public E pop() { throw new StubException(); }  
    ...  
}
```

Compile with stubs, execute with legacy library

```
$ javac -classpath stubs Client.java  
$ java -ea -classpath legacy Client
```

Wrappers (not recommended!)

Generified wrapper class

```
interface GenericStack<E> {  
    void push(E elt);  
    E pop();  
    public Stack unwrap();  
}  
  
class StackWrapper<E> implements GenericStack<E> {  
    private Stack st = new ArrayStack();  
    public void push(E elt) { st.push(elt); }  
    public E pop(){ return (E)st.pop(); } //unchecked cast  
}
```

Generic client

```
GenericStack<String> stack = new StackWrapper<String>();  
stack.push("first");  
String top = stack.pop();
```

Problems With Wrappers

- Parallel class hierarchies
 - `Stack/GenericStack` etc
- Nested structures lead to multiple wrapper layers
 - E.g. a stack of stacks
- Library essentially in two versions
 - For generified and legacy clients

**Wrappers recreate the problems that
erasure solves**

Generics

Generics

Why have them?

Implementation by erasure – benefits ...

... and problems

What next?

Collections

Trends in concurrency policy

Trends in API design

How to choose an implementation

Problems of Erasure

- Parameter types are not *reified* – they are not represented at run-time
- Constructs requiring run-time type information don't work well (or don't work)
 - Casts and `instanceof`
 - Parametric exceptions
 - Problems with arrays
 - array run-time typing doesn't play well with erasure

No Arrays Of Generic Types

Converting a collection to an array:

```
class ConversionAttemptOne {  
    static <T> T[] toArray(Collection<T> c) {  
        T[] a = new T[c.size()];    // compile error  
        int i = 0;  
        for (T x : c) {  
            a[i++] = x;  
        }  
        return a;  
    }  
}
```

The Principle of Truth in Advertising

Converting a collection to an array:

```
class AttemptTwo {  
    static <T> T[] toArray(Collection<T> c) {  
        T[] a = (T[])new Object[c.size()]; // unchecked cast  
        int i = 0;  
        for (T x : c) {  
            a[i++] = x;  
        }  
        return a;  
    }  
}
```

Is the return type from `toArray` an honest description?

The Principle of Truth in Advertising

An innocent client tries to use `AttemptTwo` :

```
public static void main (String[] args) {  
    List<String> strings = Arrays.asList("one","two");  
    String[] sa =  
        AttemptTwo.toArray(strings); //ClassCastException!  
}
```

What happened?

The Principle of Truth in Advertising

This is AttemptTwo after erasure:

```
class AttemptTwo {  
    static Object[] toArray(Collection c) {  
        Object[] a = (Object[])new Object[c.size()];  
        ...  
        return a;  
    }  
}
```

And this is the innocent client:

```
String[] sa = (String[])AttemptTwo.toArray(strings);
```

The Principle of Truth in Advertising

Static type of the array

Compiler inserts
cast to static type

Reified (ie run-time)
type is `Object[]`

```
String[] sa = (String[]) AttemptTwo.toArray(strings);
```

**The reified type of an array must be a subtype
of the erasure of its static type**
(and here, it's not)

Converting A Collection To An Array

Get type information at run-time from array or class token

```
class SuccessfulConversion {
    static <T> T[] toArray(Collection<T> c, T[] a) {
        if (a.length < c.size())
            a = (T[])Array.newInstance( // unchecked cast
                a.getClass().getComponentType(), c.size());
        int i = 0; for (T x : c) a[i++] = x;
        if (i < a.length) a[i] = null;
        return a;
    }
    static <T> T[] toArray(Collection<T> c, Class<T> k) {
        T[] a = (T[])Array. // unchecked cast
            newInstance(k, c.size());
        int i = 0; for (T x : c) a[i++] = x;
        return a;
    }
}
```

Principle of Indecent Exposure

Don't ignore unchecked warnings!

```
class Cell<T> {
    private T value;
    Cell(T v) { value = v; }
    T getValue() { return value; }
}

class DeceptiveLibrary {
    static Cell<Integer>[] createIntCellArray(int size) {
        return (Cell<Integer>[])           // unchecked cast
            new Cell[size];
    }
}

class InnocentClient {
    Cell<Integer>[] intCellArray = createIntCellArray(3);
    Cell<? extends Number>[] numCellArray = intCellArray;
    numCellArray[0] = new Cell<Double>(1.0);
    int i = intCellArray[0].getValue(); //ClassCastException
}
```

Principle of Indecent Exposure

```
return (Cell<Integer>[])new Cell[size];
```

Don't publicly expose an array whose components do not have a reifiable type
(and here, we have done)

Generics

Generics

Why have them?

Implementation by erasure – benefits ...

... and problems

What next?

Collections

Trends in concurrency policy

Trends in API design

How to choose an implementation

What Next For Generics?

- Reification?
 - The debate rages on...
 - Technically feasible?
 - Compatibility problems
 - One possible approach: distinguish reified type parameters with new syntax
 - `interface NewCollection<class E> extends Collection<E> { ... }`
 - Discussion on Java SE 7 platform still in early stages

Collections

Generics

Why have them?

Implementation by erasure – benefits ...

... and problems

What next?

Collections

Trends in concurrency policy

Trends in API design

How to choose an implementation

Collections concurrency policy

How has it changed?

- JDK™ 1.0 Software
 - Synchronized collection methods
- JDK 1.2 Software
 - Java Collections Framework API – unsynchronized
 - Optional method synchronization with synchronized wrappers
- J2SE 5 platform
 - `java.util.concurrent` (Java Specification Request (JSR)166)
 - Thread-safe classes designed for efficient concurrent access

Many java.util Collections Aren't Thread-Safe (by design)

- From `java.util.ArrayList`

```
public boolean add(E e) {  
    ensureCapacity(size + 1);  
    elementData[size++] = e;  
    return true;  
}
```

- The value in `elementData` is set, then `size` is incremented
Two threads could execute `add` concurrently, with `size == 0` initially:
 1. Thread A sets `elementData[0]`
 2. Thread B sets `elementData[0]`
 3. Thread A increments `size`
 4. Thread B increments `size`
- Unsynchronized method access leaves the `ArrayList` in an inconsistent state

Some java.util Collections Are Thread-Safe (at a cost)

From `java.util.Vector` (JDK 1.0)

```
public synchronized void addElement(E obj){
    ensureCapacityHelper(elementCount + 1)
    elementData[elementCount++] = obj;
}
```

From `java.util.Collections` (JDK 1.2)

```
static class SynchronizedList<E> implements List<E> {
    final List<E> list; final Object mutex;
    SynchronizedList(List<E> list) {this.list = list;}
    public void add(int index, E element) {
        synchronized(mutex) {list.add(index, element);}
    }
    ...
}
```

Thread-Safe != Concurrent

Even thread-safe `java.util` collections have *fail-fast iterators*

```
List<String> sl = new ArrayList<String>();  
sl.addAll(Collections.nCopies(1000000, "x"));
```

- Thread A:

```
for( Iterator<String> itr = sl.iterator();  
    itr.hasNext(); ) {  
    System.out.println(itr.next());  
}
```

- Thread B:

```
for( int i = 999999; i > 0; i-- ) {  
    sl.remove(i);  
}
```

Thread A throws **ConcurrentModificationException** immediately after thread B *first* modifies the List

Using java.util Collections Concurrently

Additional safeguards needed for concurrent access

- Use *client-side locking*
- Subclass or wrap the collection:

```
public class WrappedList<T> implements List<T> {  
    private final List<T> list;  
    public WrappedList(List<T> list){ this.list = list; }  
    public synchronized void addIfAbsent(T x) {  
        if (!list.contains(x))  
            list.add(x);  
    }  
}  
// delegate other methods
```

For concurrent use, `java.util` collections must often be locked for all operations, ***including iteration!***

Concurrent Collections

No safeguards needed for `java.util.concurrent` classes

Collections in `java.util.concurrent` don't require external locking:

- Atomic operators provided where necessary
 - **ConcurrentMap** operations
 - atomic test-then-act: `putIfAbsent`, `remove`, `replace`
 - **Blocking{Queue|Deque}** operations
 - blocking operations: `take`, `put`
 - operations from `Queue` or `Deque` now required to be atomic
- Iterators are *snapshot* or *weakly consistent*
 - Never throw **ConcurrentModificationException**

Concurrent Collections

Two kinds of iterator behavior

- Copy-on-write collections
 - `CopyOnWriteArraySet`, `CopyOnWriteArrayList`
 - *snapshot* iterators
 - underlying array is effectively immutable
 - iterators **do not** reflect changes in underlying collection
 - never fail with `ConcurrentModificationException`
- Other concurrent collections
 - *weakly consistent* (wc) iterators
 - Iterators **may** reflect changes in underlying collection
 - never fail with `ConcurrentModificationException`

Collections

Generics

Why have them?

Implementation by erasure – benefits ...

... and problems

What next?

Collections

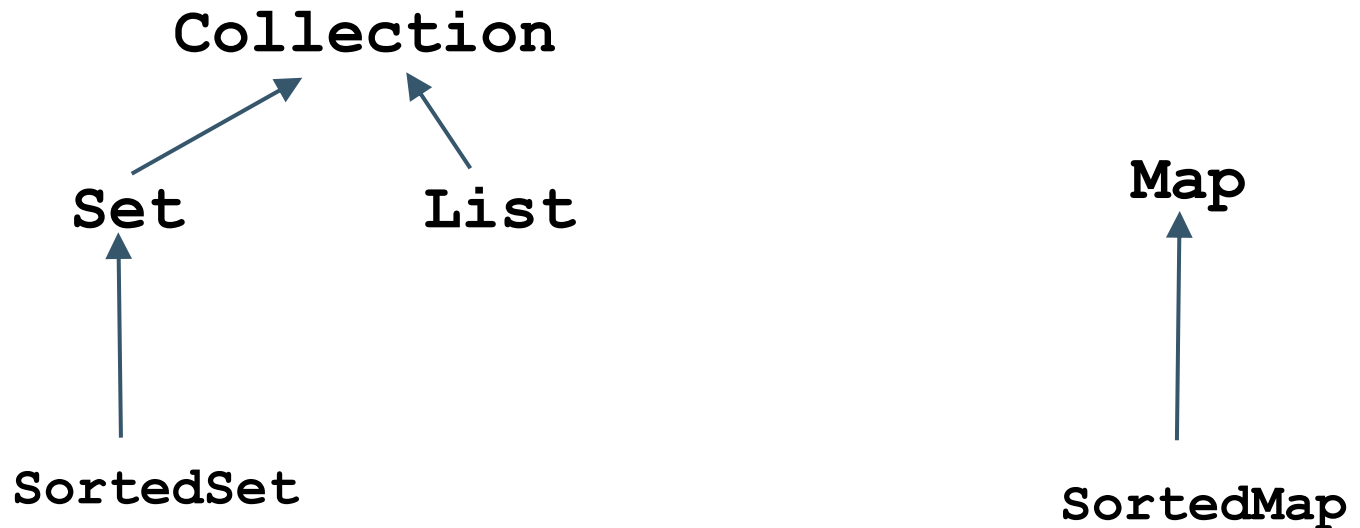
Trends in concurrency policy

Trends in API design

How to choose an implementation

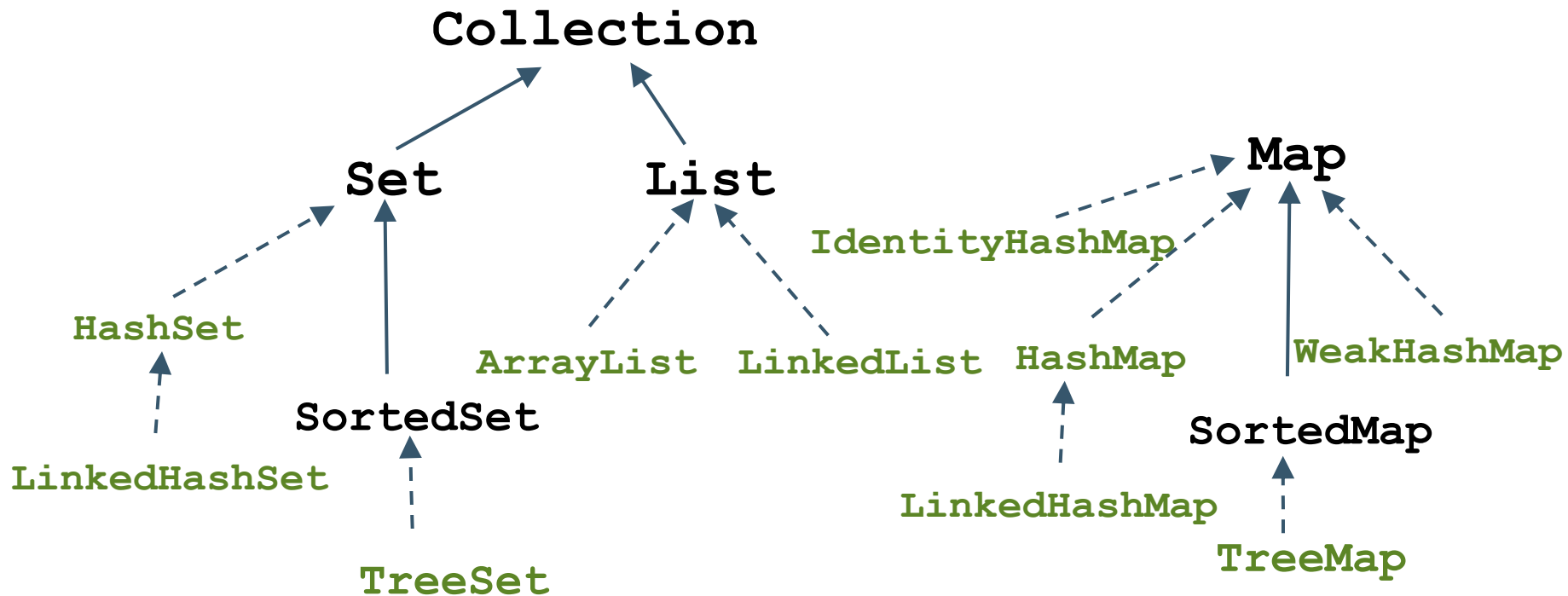
Java Collections Framework at Java 2

Interface-based API:



Implementations: JDK 1.2 software – JDK 1.4 software

Increasing choice of implementations:



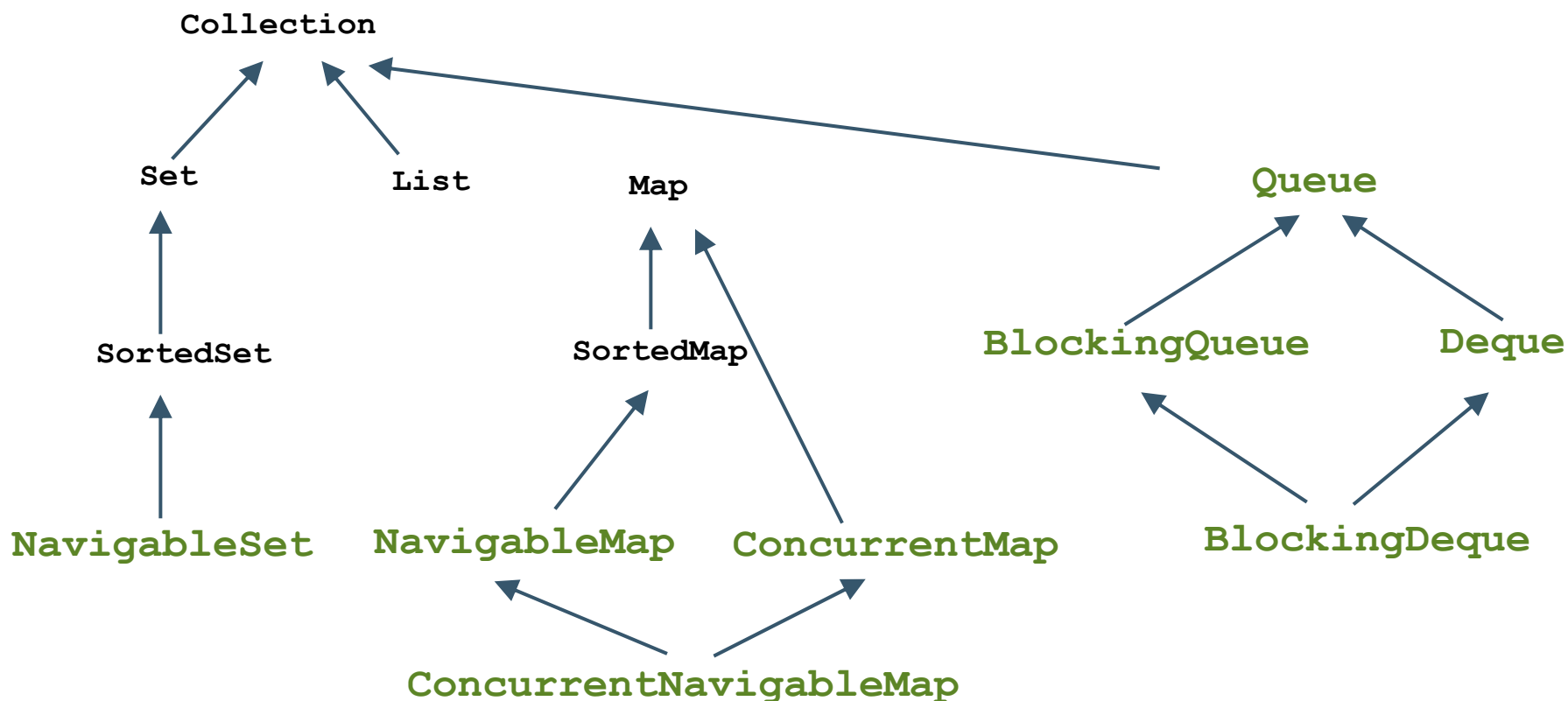
Collections in J2SE 5 platform and Java SE 6 platform

Additions to the Collections Framework

- Top-level Interface
 - Queue
- Subinterfaces
 - Deque, NavigableMap, NavigableSet
- Concurrent interfaces in `java.util.concurrent`
 - BlockingQueue, BlockingDeque,
ConcurrentMap, ConcurrentNavigableMap
- 18 implementation classes

Collections in J2SE 5 platform and Java SE 6 platform

Eight new interfaces



Queue and Deque

- Queues hold elements prior to processing
 - yield them in order for processing
 - typically in producer-consumer problems
- **`java.util.Queue`**
 - `offer/add`, `poll/remove`, `peek/element`
 - implementations provide FIFO, delay, or priority ordering
- **`java.util.Deque`**
 - `offerLast/addLast`, `pollFirst/removeFirst`, `peekFirst/elementFirst`
 - FIFO or LIFO ordering

Navigable Collections

- **`Navigable{Set|Map}`** improve on **`Sorted{Set|Map}`**
 - **`NavigableXXX`** extends and replaces **`SortedXXX`**
 - **`TreeSet`** and **`TreeMap`** retrofitted to implement new interfaces
 - Concurrent implementations: **`ConcurrentSkipListSet`**, **`ConcurrentSkipListMap`**
- Operations on **`NavigableSet`**
 - **`ceiling/floor`**, **`higher/lower`**, **`pollFirst/pollLast`**
 - **`headSet`**, **`tailSet`**, **`subSet`** overloaded to allow choice of inclusive or exclusive limits (unlike **`SortedSet`** operations)

Example Use of NavigableSet

- A set of dates suitable for use in an events calendar
- A date is in the set if there is an event on that date
- We use `org.joda.time.LocalDate` to represent dates

```
NavigableSet<LocalDate> calendar = new TreeSet<LocalDate>();  
LocalDate today = new LocalDate();  
calendar.ceiling(today); // the next date, starting with  
                        // today, that is in the calendar  
calendar.higher(today); // the first date in the future  
                        // that is in the calendar  
calendar.pollFirst();    // the first date in the calendar  
calendar.tailSet(today, false);  
                        // all future dates in the calendar
```

Collections

Generics

Why have them?

Implementation by erasure – benefits ...

... and problems

What next?

Collections

Trends in concurrency policy

Trends in API design

How to choose an implementation

Choosing a Collection Implementation

- Choose on the basis of
 - Functional behavior
 - Performance characteristics
 - Concurrency policies
- Not all combinations available
 - Like buying a car – if you want VXR trim, you have to have the 2.8i engine
 - Some customization
 - Synchronized wrappers

Choosing a Set Implementation

- Special-purpose implementations:
 - **EnumSet** – for sets of enum – not thread-safe; wc iterators
 - **CopyOnWriteArraySet** – thread-safe, snapshot iterators, used when there are more reads than writes and set is small
- General-purpose implementations:
 - **HashSet, LinkedHashSet** – not thread-safe; fail-fast iterators
 - **LinkedHashSet** faster for iteration, provides access ordering

	add	contains	next
HashSet	O(1)	O(1)	O(n/h)
LinkedHashSet	O(1)	O(1)	O(1)

- **TreeSet, ConcurrentSkipListSet** – provide ordering
 - **ConcurrentSkipListSet** thread-safe, slower for large sets

Choosing a List Implementation

- Special-purpose implementation:
 - **CopyOnWriteArrayList** – thread-safe, snapshot iterators, used when there are more reads than writes and list is small
- General-purpose implementations:
 - **LinkedList** – not thread-safe; fail-fast iterators
 - May be faster for insertion and removal using iterators
 - **ArrayList** – not thread-safe; fail-fast iterators
 - Still the best general-purpose implementation (until Java SE 7 platform?)

				iterator. remove
	get	add(e)	add(i,e)	
ArrayList	O(1)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(1)	O(1)

Choosing a Queue Implementation

- Don't need thread safety?
 - FIFO ordering – use `ArrayDeque` (not `LinkedList`!)
 - Priority ordering – `PriorityQueue`
- Thread-safe queues:
 - Specialised orderings:
 - `PriorityBlockingQueue`, `DelayQueue`
 - Best general purpose non-blocking thread-safe queue:
 - `ConcurrentLinkedQueue`
 - Blocking queue without buffering
 - `SynchronousQueue`
 - Bounded blocking queues, FIFO ordering:
 - `LinkedBlocking{Queue|Deque}`, `ArrayBlockingQueue`
 - `LinkedBlockingQueue` typically performs better with many threads

Choosing a Map Implementation

- Special-purpose implementations:
 - **EnumMap** – mapping from enums – non-thread-safe, wc iterators
 - **IdentityHashMap** – keys on identity instead of equality
 - **WeakHashMap** – allows garbage collection of “abandoned” entries
- General-purpose implementations:
 - **HashMap, LinkedHashMap** – non-thread-safe, fail-fast iterators
 - **LinkedHashMap** faster for iteration, provides access ordering, useful for cache implementations
 - **TreeMap, ConcurrentSkipListMap** – provide ordering
 - **ConcurrentSkipListMap** thread-safe, slower for large maps
 - **ConcurrentMap** – thread-safe, uses *lock striping*
 - Map divided into separately locked segments (not locked for reads)

Summary

- Generics and new Collections major step in Java Platform evolution
- Generics are a quick win in client code
 - Primary use-case: collections
 - Understand the corner cases for API design
- Collections Framework evolution
 - Fixing many deficiencies
 - **`java.util.concurrent`** – great new toolset for the Java platform programmer

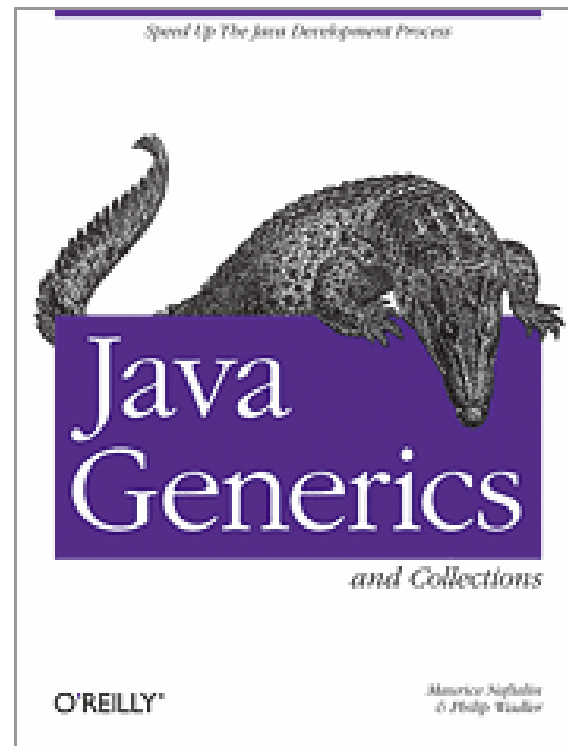
For More Information

- Angelika Langer's Generics FAQ
 - <http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>
- Java Concurrency in Practice (Goetz, et al)
Addison-Wesley, 2006
- JavaDoc for `java.util`, `java.util.concurrent`
- Concurrency-interest mailing list
 - <http://gee.cs.oswego.edu/dl/concurrency-interest/index.html>

For Much More Information

Java Generics and Collections (Naftalin and Wadler) O'Reilly, 2006

- Everything discussed today, plus
 - Subtyping and Wildcards
 - Reflection
 - Effective Generics
 - Design Patterns
 - Collection Implementations
 - The Collections class
- And lots more!





Q&A

Maurice Naftalin

<code>

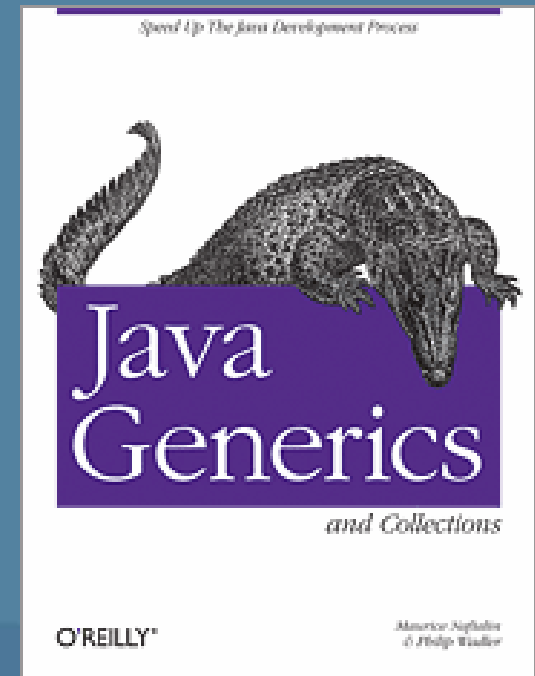


JavaOne

Java™ Generics and Collections: Tools for Productivity

Maurice Naftalin, Morningside Light
<http://www.morninglight.co.uk/>

Philip Wadler, University of Edinburgh
<http://homepages.inf.ed.ac.uk/wadler/>



TS-2890