Collections & Generics in J2SE 1.5

Dean Wette
Principal Software Engineer, Instructor
Object Computing, Inc.
St. Louis, MO

wette_d@ociweb.com

Contents

- Quick Review of Collections
- Perceived Issues with Collections
- Introduction to Generics
- Inheritance & Wildcards
- Generic Methods
- Translation: Erasure & Bridging
- Enhanced for loops
- Autoboxing
- Miscellaneous New Features

Collection Interface

- `java.util.Collection` represents the most abstract container of objects

Methods in italics are *optional*. They throw `UnsupportedOperationException` if implementation doesn't support the operation.

Note that optional operations are mutators.

```
public interface Collection {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);
    boolean remove(Object element);
    Iterator iterator();

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

    Object[] toArray();
    Object[] toArray(Object[] a);
}
```

basic operations

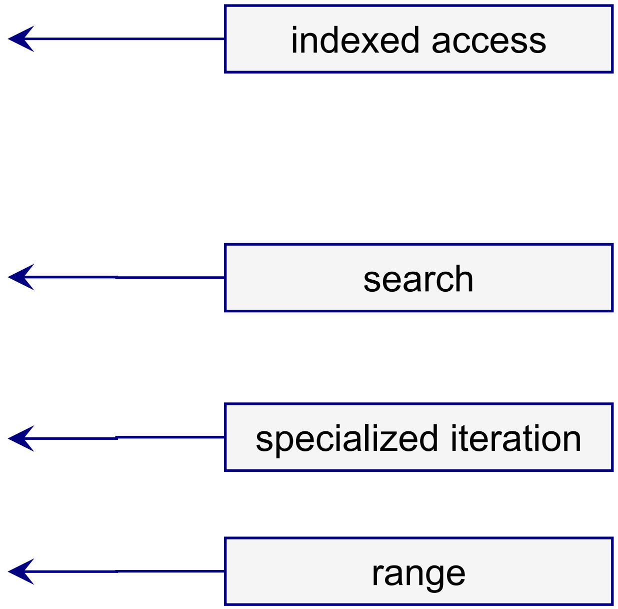
bulk operations

array operations

List Interface

- `java.util.List` represents a collection that may include duplicate elements

```
public interface List extends Collection {
    Object get(int index);
    Object set(int index, Object element);
    void add(int index, Object element);
    Object remove(int index);
    boolean addAll(int index, Collection c);

    int indexOf(Object o);
    int lastIndexOf(Object o);

    ListIterator listIterator();
    ListIterator listIterator(int index);

    List subList(int from, int to);
}
```



indexed access

search

specialized iteration

range

Set and SortedSet Interfaces

- `java.util.Set` is a restriction of `Collection` to define that elements are unique

```
public interface Set extends Collection {}
```

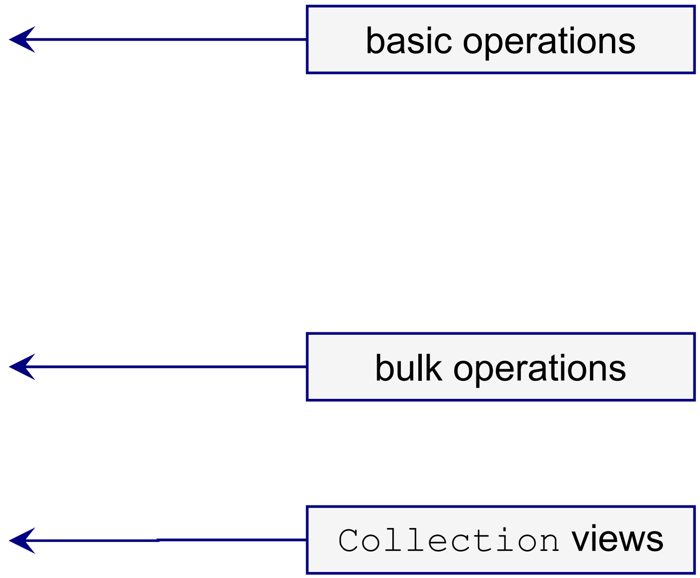
- `java.util.SortedSet` represents a `Set` that maintains elements in sorted order
  - elements implement the `Comparable` interface, or
  - the `SortedSet` is constructed with a `Comparator`

```
public interface SortedSet extends Set {
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet headSet(Object toElement);
    SortedSet tailSet(Object fromElement);
    Object first();
    Object last();
    Comparator comparator();
}
```

Map Interface

- `java.util.Map` represents a set of key/value pairs
  - a `Map` it is not a `Collection`

```
public interface Map {
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    void putAll(Map map);
    void clear();

    public Set keySet();
    public Collection values();
    public Set entrySet();
    // more...
}
```



basic operations

bulk operations

`Collection` views

Map.Entry Interface

- The `Map` interface has an inner interface representing a key-value pair
  - the `Set` returned by `entrySet()` contains elements of this type

```
public interface Map {
    // Map methods
    ...
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

SortedMap Interface

- java.util.SortedMap represents a Map that maintains keys in sorted order
  - keys implement the Comparable interface, or
  - the SortedMap is constructed with a Comparator

```
public interface SortedMap extends Map {
    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);
    Object firstKey();
    Object lastKey();
    Comparator comparator();
}
```

Iterators

- Collections provide iterators for traversal of elements

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

- A typical pattern for use

```
Iterator iter = c.iterator();
while (iter.hasNext()) {
    Object o = iter.next();
}
```

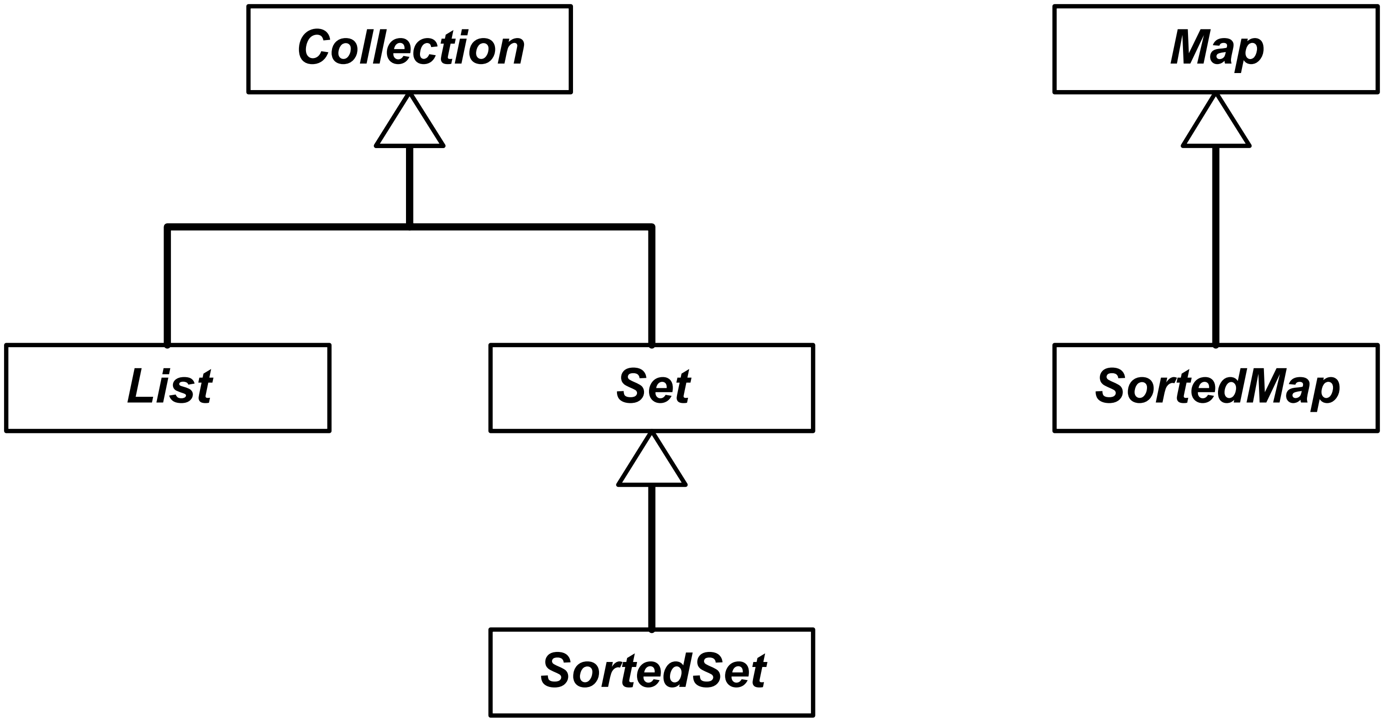
or…

```
for(Iterator i = c.iterator();i.hasNext();){
    Object o = i.next();
}
```

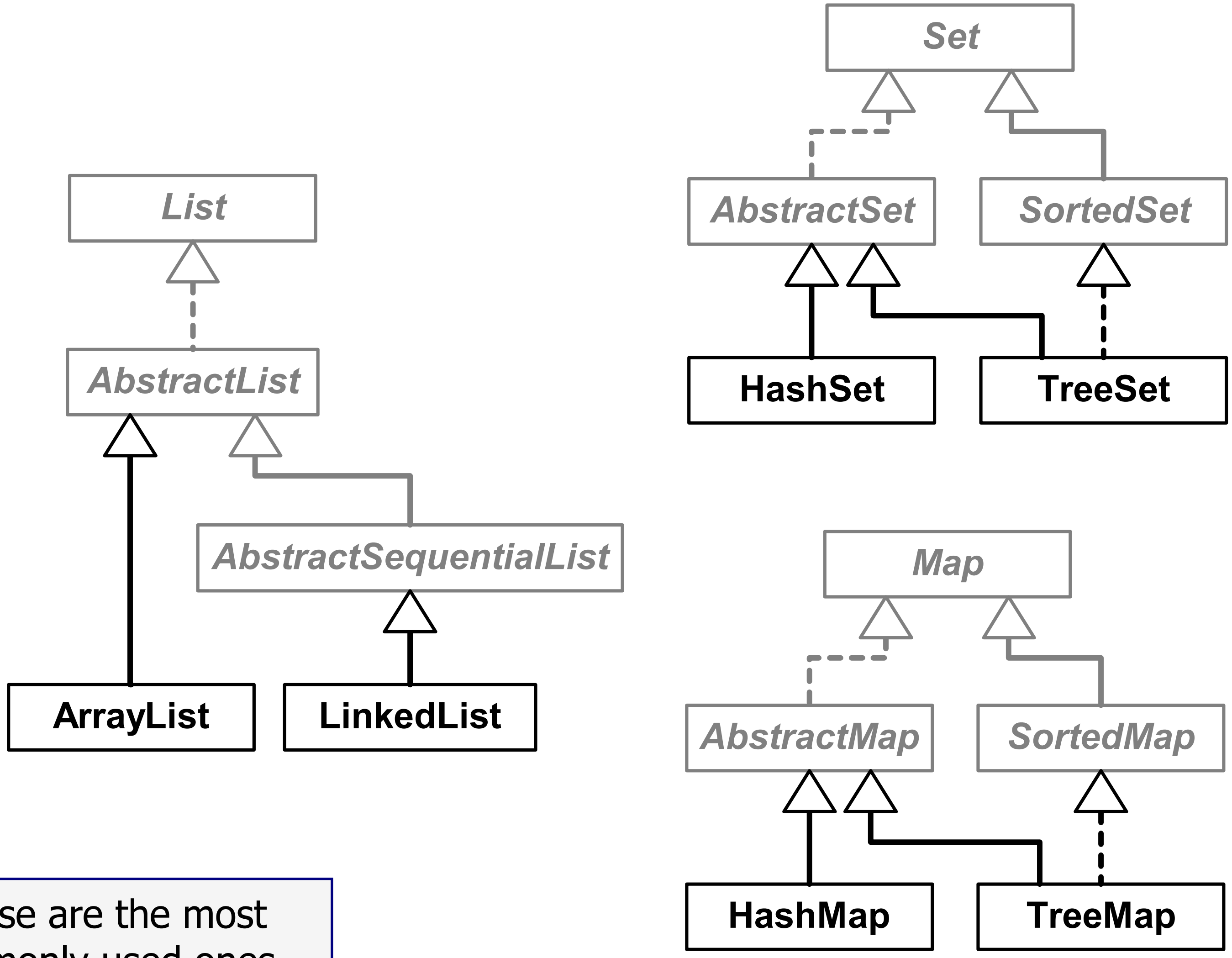
- Maps provide Collection views for iteration

```
Iterator iter = m.keySet().iterator();      // set of keys
Iterator iter = m.values().iterator();      // collection of values
Iterator iter = m.entrySet().iterator();    // set of Map.Entry
```

Collections Interface Hierarchy



General Purpose Implementations



These are the most commonly used ones.

Other J2SE Implementations

- Legacy (since 1.0)

```
java.util.Vector
java.util.Stack
java.util.Hashtable
java.util.Properties
```

- J2SE 1.2

```
java.util.WeakHashMap
```

- J2SE 1.4

```
java.util.LinkedHashSet
java.util.LinkedHashMap
java.util.IdentityHashMap
```

- J2SE 1.5

```
java.util.EnumSet
java.util.EnumMap
java.util.PriorityQueue
java.util.concurrent.*
```

Perceived Issues with Collections

```
Collection c = new ArrayList();
c.add(new DomainObject(1));
c.add(new DomainObject(2));
...
c.add(aDomainObject.getObjectId());
...
String ids = createDomainIdsString(c);
```

compiler provides no guarantee for element type

```
public String createDomainIdsString(Collection c) {
    StringBuffer buf = new StringBuffer();
    Iterator iter = c.iterator();
    while (iter.hasNext()) {
        DomainObject dObj = (DomainObject)iter.next();
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

ClassCastException

Using Documentation

```
Collection domainObjects = new ArrayList();
...
String ids = createDomainIdsString(domainObjects);
```

javadoc specifies element type,
but still not enforced by compiler

```
/**
 * Creates a comma-delimited string of domain object ids,
 * intended for SQL IN clauses.
 *
 * @param domainObjects collection of elements of type DomainObject
 * @return a string of the form: id1,id2,...,idn
 */
public String createDomainIdsString(Collection domainObjects) {
    StringBuffer buf = new StringBuffer();
    Iterator iter = domainObjects.iterator();
    while (iter.hasNext()) {
        DomainObject dObj = (DomainObject)iter.next();
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

Huh? What's javadoc?

```
Collection c = new ArrayList();
...
String ids = createDomainIdsString(c);
```

```
/**
 * auto-generated comment
 * Insert description here.
 *
 * @param c
 * @return String
 */
public String createDomainIdsString(Collection c) {
    StringBuffer buf = new StringBuffer();
    Iterator iter = c.iterator();
    while (iter.hasNext()) {
        DomainObject dObj = (DomainObject)iter.next();
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

Strengthening Type

```
DomainObject[] doArr = new DomainObject[...];
doArr[0] = new DomainObject(1);
doArr[1] = new DomainObject(2);
...
...
doArr[n] = aDomainObject.getObjectId();
...
String ids = createDomainIdsString(doArr);
```

this won't compile

compiler enforces element type

```
public String createDomainIdsString(DomainObject[] doArr) {
    StringBuffer buf = new StringBuffer();
    for (int i = 0; i < doArr.length; ++i) {
        buf.append(doArr[i].getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

Array/Collection Conversion

```
Collection c = new ArrayList();
c.add(new DomainObject(1));
c.add(new DomainObject(2));
...
c.add(aDomainObject.getObjectId());
...
...
DomainObject[] doArr = (DomainObject[])c.toArray(new DomainObject[c.size()]);
String ids = createDomainIdsString(doArr);
```

ArrayStoreException

```
public String createDomainIdsString(DomainObject[] doArr) {
    StringBuffer buf = new StringBuffer();
    for (int i = 0; i < doArr.length; ++i) {
        buf.append(doArr[i].getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

Collections & Generics

```
Collection<DomainObject> c = new ArrayList<DomainObject>();
c.add(new DomainObject(1));
c.add(new DomainObject(2));
...
c.add(aDomainObject.getObjectId());
...
String ids = createDomainIdsString(c);
```

this won't compile

compiler enforces element type

```
public String createDomainIdsString(Collection<DomainObject> c) {
    StringBuffer buf = new StringBuffer();
    Iterator<DomainObject> iter = c.iterator();
    while (iter.hasNext()) {
        DomainObject dObj = iter.next();
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

explicit cast not required

Generics in J2SE 1.5

- Supports specializing type when using classes
  - and generalizing type when implementing them
- Improves code clarity

```
Map cardsInSuits = new HashMap();
Map<Suit, Set<Card>> cardsInSuits = new HashMap<Suit, Set<Card>>();
```

- Improves robustness/reliability

```
List<Integer> integers = new ArrayList<Integer>();
integers.add("1000");   // fails with compiler error
```

- Removes need for casts

```
Integer i = integers.get(0);
```

- Backwards compatible
  - *raw* type – using a generic type without a type argument
  - mixing generic code and legacy code
  - both are legal, but generate compiler warnings

Generic APIs

- Collections API
  - changed to support generics
  - legacy Collections code will continue to work
    - compiler generates warnings about unsafe/unchecked types
- Reflection API
  - changed to support generics

Basic Generics Syntax

- Two new forms of types
  - *parameterized types*

    `Collection<Integer>` `c = new ...`
  - *type variables*

    `interface Collection<`**`T`**`> {...`
- Enclosed by angle brackets – **<***type* | *type variable***>**
  - multiple types are comma-delimited – <*K*, *V*>
  - can be nested – **<**`String,` **<**`Collection`**<**`Integer`**>>**
- Type *variables* can be any unqualified legal identifier
  - can be referenced in (non-static context) enclosed code
    - class members
    - method arguments
    - return types
    - variable types

Basic Generics Syntax (cont'd)

- Type *variables* used to declare and reference generic types

```
class Pair<F,S> {
    F first;
    S second;
    public F getFirst() { return first; }
}
```

- Naming conventions for type variables
  - use upper case single letters
    - **T** for "**T**ype"

      ```
      public interface Comparator<T>
      ```

    - Collections API uses **E** for "**E**lement"

      ```
      public interface Set<E>
      ```

    - Collections API uses **K,V** for "**K**ey/**V**alue"

      ```
      public interface Map<K,V>
      ```

Basic Generics Syntax (cont'd)

- Type *parameters* for *defining* generic types/methods
  - class/interface definitions

```
public interface Comparable<T>
public class ArrayList<E> implements List<E>
```

  - method declarations

```
public ArrayList(Collection<? extends E> c) {...}
```

- Type *arguments* for *using* generic types
  - declaration and instantiation

```
public class DomainObject implements Comparable<DomainObject>
Collection<Integer> cInts = new ArrayList<Integer>();
Map<Id,DomainObject> m = new LinkedHashMap<Id,DomainObject>();
Map<ProductType, Collection<Plan>> m = ...
```

Basic Generics Syntax (cont'd)

- Generic type arguments cannot be primitives
  - this is not legal

```
Collection<int> ints = new ArrayList<int>();
```

  - but autoboxing allows this

```
Collection<Integer> ints = new ArrayList<Integer>();
ints.add(1234);
```

more on autoboxing later...

- Generics and exceptions
  - type parameters are allowed in throws classes
    - as long as they extend `Exception`, e.g. `<X extends Exception>`
  - parameterized types cannot be used in catch clauses
  - can be generic about what gets thrown
  - can be specific about what gets caught

Java Generics vs. C++ Generics

- Somewhat similar to C++ templates
  - define types used in a class generically
  - similar syntax
  - but little else in common
- But quite different than C++ templates
  - Java generics adds type bounds and wildcards
  - new Java classes are not created
    - no template instantiation
    - compiler performs erasure
      - more on this later
  - parameterized instances share classes
    - `HashSet`, `HashSet<String>`, `HashSet<Integer>` are all the same class
  - primitives not supported

A Simple Generic Class

```java
public class TTPair<T> {
    private T first;
    private T second;

    public TTPair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public T getSecond() {
        return second;
    }
}
```

```java
public class FSPair<F,S> {
    private F first;
    private S second;

    public FSPair(F first, S second) {
        this.first = first;
        this.second = second;
    }

    public F getFirst() {
        return first;
    }

    public S getSecond() {
        return second;
    }
}
```

```java
TTPair<String> name = new TTPair<String>("Dean", "Wette");

FSPair<Integer, DomainObject> entry =
    new FSPair<Integer, DomainObject>(domObj.getObjectId(), domObj);
```

Bounded Types

- Type arguments can be constrained by bounded type parameters (polymorphic parameterization)
  - `<T>` – type is unbounded
  - `<T,U>` – two types, both unbounded
  - `<T extends JButton>`
    - type is a `JButton` or a subclass of `JButton`
  - `<T extends Action>`
    - type implements the `Action` interface
  - `<T extends InputStream & ObjectInput>`
    - T is a subclass of `InputStream` and implements `ObjectInput`
  - `<T extends Comparable<T>>`
    - T implements the generic interface `Comparable` in terms of T
  - `<T, S super T>`
    - two types with S defined in terms of T, where S is a T or a superclass of T

Something A Bit More Complex

```
public class Pair<F extends Comparable<F> & Serializable,
                  S extends Comparable<S> & Serializable>
             implements Comparable<Pair<F,S>>, Serializable {

    private F first;
    private S second;

    public Pair(F first, S second) {
        this.first = first;
        this.second = second;
    }

    public F getFirst() { return first; }
    public S getSecond() { return second; }

    public int compareTo(Pair<F,S> that) {
        int result = this.first.compareTo(that.first);
        if (result == 0) {
            result = this.second.compareTo(that.second);
        }
        return result;
    }
}
```

`Pair<File,FileInputStream>` is now illegal

Generics & Inheritance

- Relationship between generics and inheritance can be counter-intuitive
  - for example

```
class DomainObject {...}
class SubDomainObject extends DomainObject {...}

Set<SubDomainObject> sdos = new HashSet<SubDomainObject>();
Set<DomainObject> dos = sdos;
```

  - is `Set<SubDomainObject>` a subclass of `Set<DomainObject>`?

Generics & Inheritance

- Relationship between generics and inheritance can be counter-intuitive
  - for example

```
class DomainObject {...}
class SubDomainObject extends DomainObject {...}

Set<SubDomainObject> sdos = new HashSet<SubDomainObject>();
Set<DomainObject> dos = sdos;
```

  - is `Set<SubDomainObject>` a subclass of `Set<DomainObject>`?

```
dos.add(new DomainObject(id));
SubDomainObject sdo = sdos.iterator().next();
```

Generics & Inheritance

- Relationship between generics and inheritance can be counter-intuitive
  - for example

```
class DomainObject {...}
class SubDomainObject extends DomainObject {...}

Set<SubDomainObject> sdos = new HashSet<SubDomainObject>();
Set<DomainObject> dos = sdos;    // compile error
```

  - is `Set<SubDomainObject>` a subclass of `Set<DomainObject>`?

```
dos.add(new DomainObject(id));
SubDomainObject sdo = sdos.iterator().next();
```

assign `DomainObject` to `SubDomainObject`

This Won't Compile

```
class DomainObject {...}
class SubDomainObject extends DomainObject {...}


Collection<SubDomainObject> sdos = new ArrayList<SubDomainObject>();
ac.add(new SubDomainObject(1));
ac.add(new SubDomainObject(2));
...
String ids = createDomainIdsString(sdos);
...

public String createDomainIdsString(Collection<DomainObject> c) {
    StringBuffer buf = new StringBuffer();
    Iterator<DomainObject> i
    while (iter.hasNext()) {
        DomainObject dObj =
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

```
createDomainIdsString(Collection<DomainObject>)
cannot be applied to (Collection<SubDomainObject>)
//  String ids = createDomainIdsString(sdos);
```

But This Will

```
class DomainObject {...}
class SubDomainObject extends DomainObject {...}


Collection<SubDomainObject> sdos = new ArrayList<SubDomainObject>();
sdos.add(new SubDomainObject(1));
sdos.add(new SubDomainObject(2));
...
String aIds = createDomainIdsString(sdos);
...

public <E extends DomainObject> String createDomainIdsString(Collection<E> c) {
    StringBuffer buf = new StringBuffer();
    Iterator<E> iter = c.i
    while (iter.hasNext())
        E dObj = iter.next();
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

Solved by making the type more flexible using type parameter bounds.
`E` is of type `DomainObject` or any subclass of `DomainObject`

Rules for Generics Inheritance

- If **`S`** is a subtype of **`T`**, and **`G`** is some generic type, it is **not** true that
  - **`G<S>`** is a subtype of **`G<T>`**

  `Set<SubDomainObject>` is **not** a subtype of `Set<DomainObject>`
- A generic type G_S is a subtype of G_T, if and only if
  - the type arguments are identical
  - the raw type of G_S is a subtype of the raw type of G_T

  `HashSet<DomainObject>` **is** a subtype of `Set<DomainObject>`
- Subtype guarantee
  - any method call you can make on **`T`** you can make on **`S`**
  - its why polymorphism works, and the following doesn't...

```
Collection<Number> numbers = new ArrayList<Integer>();
numbers.add(new Double(1.0)); // broken virtual call
```

Wildcards

- Easy to learn the basics, harder to use effectively
- Adds flexibility (and complexity) to type parameter bounding
  - used when a variety of types are expected to match the type parameter
- Addresses the issue of parameterization & inheritance
- Used instead of a type variable in a type parameter
  - designated with a '**?**'
  - indicates an unknown type
  - can be used wherever a type parameter can be used
    - field, method, variable declaration
    - by itself or with parameter bounding
  - unlike a type variable, cannot reference wildcard in code

Wildcards (Cont'd)

- Example (for declarations)
  - a `List` of `Number` objects: **`List<Number>`**
  - a `List` of any subclass of `Number`: **`List<? extends Number>`**
    - such as `List<Integer>` or `List<Double>`
  - a `Collection` of whatever: **`Collection<?>`**
    - not the same as `Collection<Object>`
- With wildcards
  - can define more flexible parameterized types
  - allows assignment of generic types to fields
  - simplifies (somewhat) use of generic types as method arguments

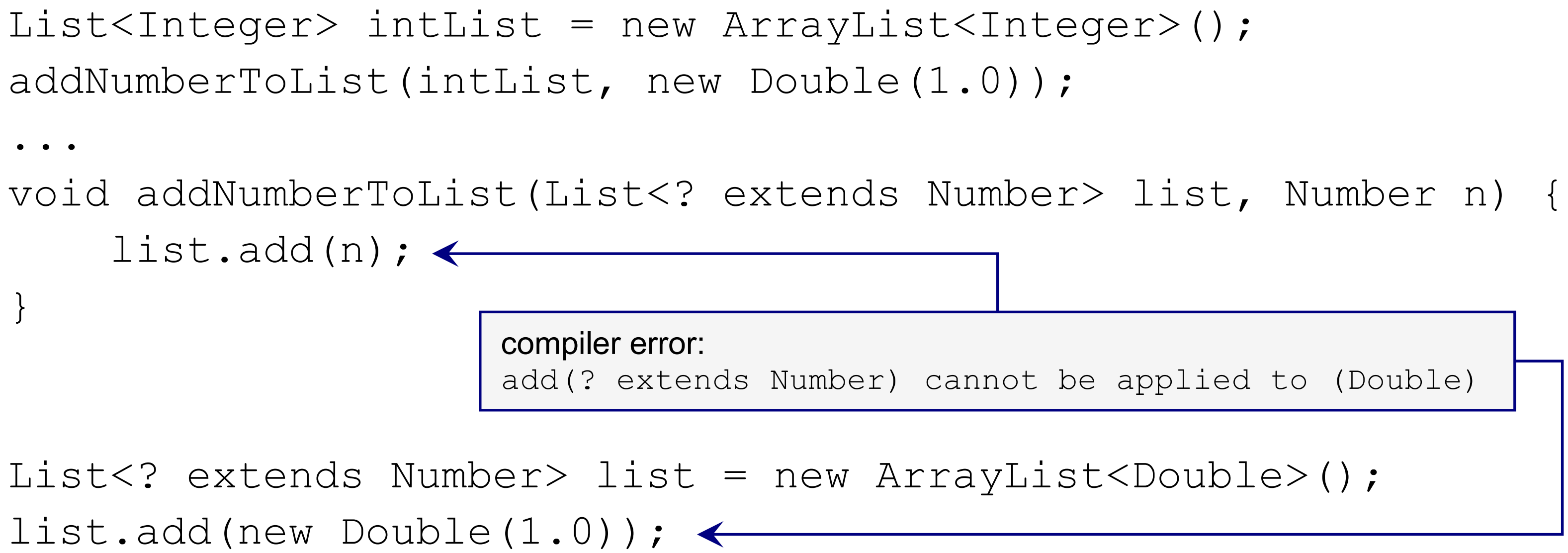
```
<T extends Foo> void bar(Collection<T> c)

void bar(Collection<? extends Foo> c)
```

Side Effect of Wildcards

- Introduces partially immutable collections

```
List<? extends Number> list = new ArrayList<Integer>()
```

- cannot add elements to a collection declared using wildcards



```
List<Integer> intList = new ArrayList<Integer>();
addNumberToList(intList, new Double(1.0));
...
void addNumberToList(List<? extends Number> list, Number n) {
    list.add(n);
}
```

compiler error:

```
add(? extends Number) cannot be applied to (Double)
```

```
List<? extends Number> list = new ArrayList<Double>();
list.add(new Double(1.0));
```

- but you can remove items

```
boolean remove(Object element);  //  Collection interface
```

Generic Collection Interface

- All collections are redefined in terms of generics

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();

    boolean containsAll(Collection c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

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

supports *enhanced for* loop more later...

Generic Iterator Interface

- Iterators are also generic

```
public interface Iterator<E> {
   int size();
   boolean hasNext();
   E next();
   void remove();
}
```

Generic Map Interface

- All maps are redefined in terms of generics

```
public interface Map<K,V> {
    Set<Map.Entry<K,V>> entrySet();
    V get(Object key);
    Set<K> keySet();
    V put(K key, V value);
    void putAll(Map<? extends K, ? extends V> map);
    V remove(Object key);
    Collection<V> values();

    public interface Entry<K,V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

non-generic
methods omitted

Generic Methods

- Methods can be defined in terms of generics

```
[modifiers] [<typeParams>] returnType methodName([argList])
    [throwsClause]
```

- New type variables are declared in the `typeParams` clause
  - i.e. those not defined by the enclosing class/interface

```
interface Collection<E> {
    boolean add(E element);
    boolean addAll(Collection<? extends E> c);
    <T> T[] toArray(T[] t);
}
```

  - if a type *variable* in a type *parameter* is not otherwise referenced, a wildcard can be used instead...

Wildcards vs. Type Variable

```
public <E extends DomainObject> String createDomainIdsString(Collection<E> c)
    StringBuffer buf = new StringBuffer();
    Iterator<E> iter = c.iterator();
    while (iter.hasNext()) {
        E dObj = iter.next();
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

type variable can be used in method body

```
public String createDomainIdsString(Collection<? extends DomainObject> c) {
    StringBuffer buf = new StringBuffer();
    Iterator<DomainObject> iter = c.iterator();
    while (iter.hasNext()) {
        DomainObject dObj = iter.next();
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

wildcards cannot be referenced

Arrays & Generics

- Arrays can be declared using a type parameter
- Arrays cannot be created if element type is generic

```
<T> T[] toArray(T[] a) {                // OK
    T[] tArr = new T[n];                // error
    ...
}
```

- The element type of an array cannot be parameterized
  - unless using an unbounded wildcard

```
List<?>[] listArr = new List<?>[16];                // OK
List<String>[] strListArr = new List<String>[16];   // error
```

Arrays & Generics (Cont'd)

- Implementation of
  `<T> T[] java.util.ArrayList.toArray(T[] a)`

```
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        a = (T[])java.lang.reflect.Array.
           newInstance(a.getClass().getComponentType(), size);

    System.arraycopy(elementData, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

Comparable & Comparator

- Redefined in terms of generics
- Old definition

```
public interface Comparable {
    public int compareTo(Object o);
}
public interface Comparator {
    public int compare(Object o1, Object o2);
}
```

- for example

```
public class MessageComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Message m1 = (Message)o1;
        Message m2 = (Message)o2;
        return m1.getText().compareTo(m2.getText());
    }
}
```

runtime error: throws `ClassCastException` if type is wrong

Comparable & Comparator (cont)

- New definition

```
public interface Comparable<T> {
    public int compareTo(T o);
}
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

- Avoids problems comparing wrong types
  - methods implemented in terms of type argument
  - compiler checks type arguments
    - won't throw `ClassCastException`

```
public class MessageComparator implements Comparator<Message> {
    public compare(Message m1, Message m2) {
        return m1.getText().compareTo(m2.getText());
    }
}
```

compile error if type is wrong

Erasure of Generics

- Java compiler performs *erasure* of all generic type info
  - everything between **< >** is thrown away
  - remaining uses of type variable replaced by type of upper bound
    - `<T>` replaced by `Object`
    - `<T extends Serializable>` replaced by `Serializable`
  - casts inserted to make source type correct (compilable)
- Intended to support backwards compatibility
  - so generics can interoperate with non-generic legacy code – raw types
- Casts and `instanceof`
  - testing instanceof on a generic instance results in compiler error

```
if (list instanceof List<Integer>)      // illegal
```

  - casting to generic type results in "unchecked" warning

```
(List<Integer>)list                     // warning
```

Consequences of Erasure

- Different than C++ templates
  - `List<String>` and `List<Integer>` are same `List` class
- Cannot use type variables in static context
  - i.e. can't define static members in terms of class type parameter

```
class Erased<T> {
    static T staticField;                      // error
    static Collection<T> ct;                   // error
    static Collection<String> cs;              // OK
    T objectField;                             // OK
    static T getSF() { return staticField; }   // error
    static void do(List<T> lt) { ... }         // error
    static <S> void do(S s) { ... }            // OK
    T getOF() { return objectField; }          // OK
}
```

Consequences of Erasure (cont'd)

- Type parameters cannot be overloaded
  - class conflict

    ```
    // error - compile time conflict
    class Pair<T> {...}
    class Pair<F,S> {...}
    ```

  - method conflict

    ```
    class Pair<F,S> {
        // error - compile time conflict
        void set(F f) {...}
        void set(S s) {...}
    }
    ```

- Erasure can also introduce unintended overrides
  - compiler enforces certain rules about this
  - see specification for more details about this and other issues

Bridging

- During erasure, the compiler may also insert bridge methods
  - necessary to make overriding work

    ```
    class DomainObject implements Comparable<DomainObject> {
        public int compareTo(DomainObject obj) { ... }
    }
    ```

    - erases to

    ```
    class DomainObject implements Comparable {
        public int compareTo(DomainObject obj) { ... }
        public int compareTo(Object obj) {
            return compareTo((DomainObject)obj);
        }
    }
    ```

- Bridge methods also used to support covariant return types
  - for returning a subtype of an overridden method's return type

Enhanced `for` Loop

- Simplifies pattern for iteration over collections and arrays

1. existing pattern

```
Collection c = ...
Iterator iter = c.iterator();
while (iter.hasNext()) {
    JButton b = (JButton)iter.next();
    b.addActionListener(this);
}
```

2. improvement using generics

```
Collection<JButton> c = ...
Iterator<JButton> iter = c.iterator();
while (iter.hasNext()) {
    iter.next().addActionListener(this);
}
```

3. using "enhanced for" syntax

```
Collection<JButton> c = ...
for (JButton b : c) {
    b.addActionListener(this);
}
```

4. also works for primitive and object arrays

```
int[] ints = ...
int sum = 0;
for (int i : ints) {
    sum += i;
}
```

Maps & Enhanced `for`

```
Map<Integer,DomainObject> m = ...

for (Integer i : m.keySet()) {
    ids += i + ",";
}

for (DomainObject dObj : m.values()) {
    int objectId = dObj.getObjectId();
}

for (Map.Entry<Integer,DomainObject> entry : m.entrySet()) {
    Integer i = entry.getKey();
    DomainObject dObj = entry.getValue();
}
```

Enhanced `for` (cont'd)

```
public <E extends DomainObject> String createDomainIdsString(Collection<E> c) {
    StringBuffer buf = new StringBuffer();
    Iterator<E> iter = c.iterator();
    while (iter.hasNext()) {
        E dObj = iter.next();
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}

public <E extends DomainObject> String createDomainIdsString(Collection<E> c) {
    StringBuffer buf = new StringBuffer();
    for (E dObj : c) {
        buf.append(dObj.getObjectId()).append(",");
    }
    return buf.substring(0, buf.length() - 1);
}
```

Iterable Interface

- Any type can be target of enhanced **`for`** loop
  - implement the `java.lang.Iterable` interface

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

`Iterable` is a super interface of `java.util.Collection`

- Example

```
class DeadMessageQueue implements Iterable<Message> {
    public Iterator<Message> iterator() { ... }
}
...
DeadMessageQueue deadMsgs = new DeadMessageQueue();
...
for (Message m : deadMsgs) {
    purge(m);
}
```

Autoboxing

makes using collections easier

- Specified as part of JSR 201
  - changes to the Java Language Specification
  - also includes enumerations, enhanced for loop, static imports
- Replaces explicit conversion of primitives with implicit conversions performed by compiler
- Boxing Conversion
  - if **p** is a value of type *primitive*, then
    - convert **p** into object reference **r** of type *WrapperClass*, such that
    - `r.value() == p`
- Unboxing Conversion
  - if **r** is an object reference of type *WrapperClass*, then
    - convert **r** into value **p** of type *primitive*, such that
    - `p == r.value()`

Autoboxing (cont'd)

- Specification also details rules for
  - forbidden conversions
  - assignment conversion
  - casting conversion
  - method invocation conversion
- Examples

```
Integer iObj = -1;
int[] iArr = { 1, 2, 3, 4, 5 };
Collection<Integer> iColl = new ArrayList<Integer>();
iColl.add(++iObj);
for (int i : iArr) iColl.add(i);
for (Integer i : iColl) {
    out.printf("Incremented %2$s is %1$d%n",
               ++i, i.getClass().getName());
}
```

```
Incremented java.lang.Integer is 1
Incremented java.lang.Integer is 2
Incremented java.lang.Integer is 3
Incremented java.lang.Integer is 4
Incremented java.lang.Integer is 5
Incremented java.lang.Integer is 6
```

Miscellaneous New Features

- Collections class
  - note: all methods are `public static`
  - existing and new methods are redefined in terms of generics
- wrappers for creating dynamic type-safe checked collections

```
<E> Collection<E> checkedCollection(Collection<E>, Class<E> type)
```

- `reverseOrder()` overloaded for specified `Comparator`

```
<T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

- miscellaneous

```
<T> boolean addAll(Collection<? super T> c, T[] a)
boolean disjoint(Collection<?> c1, Collection<?> c2)
int frequency(Collection<?> c, Object o)
```

Misc. New Features (cont'd)

- New interface: `java.util.Queue`

```
public interface Queue<E> extends Collection<E> {
    /** attempt to insert specified element */
    boolean offer(E o);
    /** retrieve and remove head element */
    E poll();
    /** retrieve head element without removing it */
    E peek();
    /** retrieve and remove head element */
    E remove();
    /** retrieve head element without removing it */
    E element();
}
```

`poll()` & `peek()` return `null` if queue is empty, `remove()` & `element()` throw `NoSuchElementException`

- `LinkedList` now implements `Queue`
- No `Collections` wrapper factory methods for `Queue`
- Several `Queue` classes in `java.util.concurrent`
  - also `BlockingQueue` subinterface

Selected References

- Bracha, Gilad. "Generics in the Java Programming Language."
  - http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
- Bracha, Gilad, et al. "JSR 14: Adding Generics To The Java Programming Language."
  - http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html
- Grosso, William. "Explorations: Generics, Erasure, and Bridging."
  - http://today.java.net/pub/a/today/2003/12/02/explorations.html
- Grosso, William. "Explorations: Wildcards in the Generics Specification."
  - http://today.java.net/pub/a/today/2004/01/15/wildcards.html
- Smith, Rob. "Generics In Java." *Java News Brief*. Object Computing, Inc. July 2003.
  - http://ociweb.com/jnb/jnbJul2003.html
- Sun Microsystems. "Java 2 SDK, Standard Edition Documentation. Version 1.5.0 Beta 1."
  - http://java.sun.com/j2se/1.5.0/docs/index.html

Collections & Generics – Q & A

Dean Wette
Principal Software Engineer, Instructor
Object Computing, Inc.
St. Louis, MO