„Generics and collections”

Generics

- From JDK 1.5.0
- They are similar to C++ templates
- They allow to eliminate runtime exceptions related to improper casting (ClassCastException)

Traditional approach

```
public class Box {
private Object object;
public void add(Object object) {
this.object = object;
}
public Object get() {
return object; }
}
```

```
public class BoxDemo1 {
public static void main(String[] args) {
 // Box for integers (?)
Box integerBox = new Box();
integerBox.add(new Integer(10));
// we are casting to Integer. Why?
Integer someInteger =
      (Integer)integerBox.get();
System.out.println(someInteger);
 } }
```

Traditional approach (2)

```
public class BoxDemo2 {
    public static void main(String[] args) {
        // Box for integers (?)
        Box integerBox = new Box();
        // In large application modified by one programmer:
        integerBox.add("10"); // note how the type is now String
        // And in the second written by a different programmer
        // Checked exception or runtime exception ?
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

Generics approach

```
public class Box<T> {
    private T t;
    // T stands for "Type"
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

```
public class BoxDemo3 {
public static void main(String[] args) {
        Box<Integer> integerBox = new
        Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get();
        // no cast!
        System.out.println(someInteger);
} }
```

Generics approach (2)

In case of adding an incompatible type to the box:

```
BoxDemo3.java:5: add(java.lang.Integer) in Box<java.lang.Integer>
    cannot be applied to (java.lang.String)
        integerBox.add("10");
                  ^
    1 error
```

The Diamond

From Java SE 7:

```
Box<Integer> integerBox = new Box<>();
```

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U – other types

Bounded type parameters

```
public class Box<T> {
   …
   public <U extends Number> void inspect(U u){
      System.out.println("T: " + t.getClass().getName());
      System.out.println("U: " + u.getClass().getName());
   }
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      integerBox.add(new Integer(10));
      integerBox.inspect("some text"); // error: this is still String!  } }
    // extends is used in a general sense to mean either "extends" (as in
    classes) or "implements" (as in interfaces)
```

Bounded type parameters (2)

```
public class NaturalNumber<T extends Integer> {
      private T n;
      public NaturalNumber(T n) {
            this.n = n;
      }
      public boolean isEven() {
            return n.intValue() % 2 == 0;
      } // ... }
```

The isEven method invokes the intValue method defined in the Integer class through n.

In this simple case declaration could be also just NaturalNumber<Integer>, but extends can be applied also to interfaces as well it can have many arguments, e.g. **extends A & B & C. Question: What are A, B, C?**

Answer: Only one of them can be class and in this case it must be first argument of extends.

1)
```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger; // OK ???
```

2)
```
public void someMethod(Number n) { /* ... */ }
someMethod(new Integer(10)); // OK ???
someMethod(new Double(10.1)); // OK ???
```

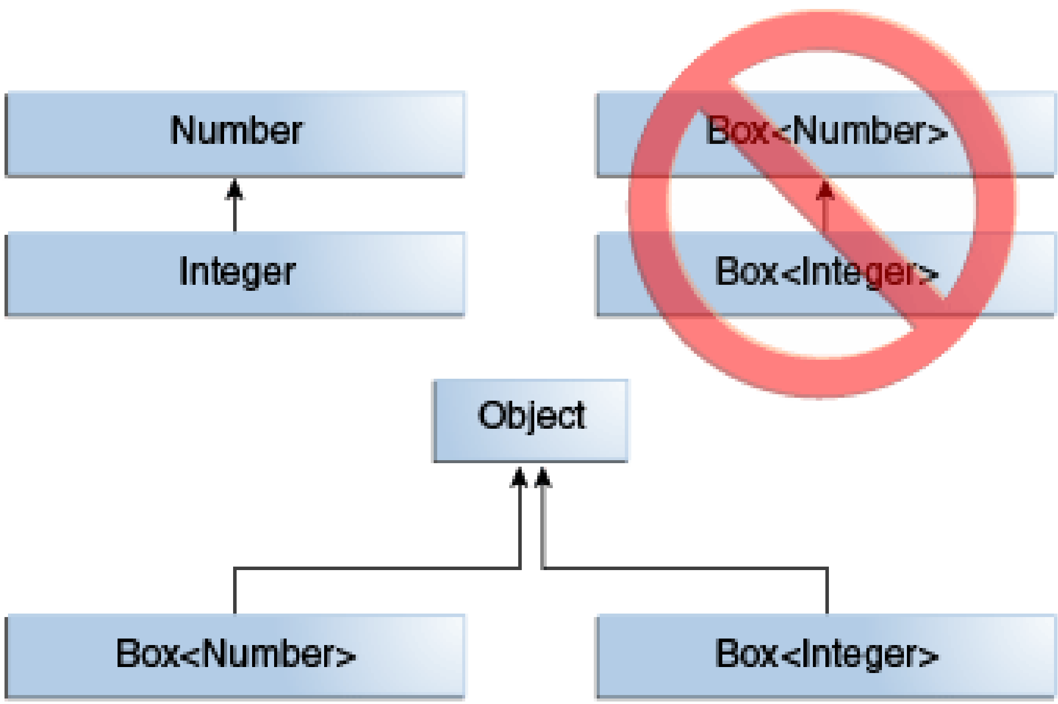
3)
```
Box<Number> box = new Box<Number>();
 box.add(new Integer(10)); // OK ???
box.add(new Double(10.1)); // OK ???
```

Answer:
Everything is ok!

Inheritance and subtypes (2)

Consider this:
public void boxTest(Box<Number> n) { /* ... */ }

Can you pass argument Box<Integer>? **Answer: No.**



Everything is an object!

Method	Description
`boolean equals (Object obj)`	Decides whether two objects are meaningfully equivalent
`void finalize()`	Called by the garbage collector when the garbage collector sees that the object cannot be referenced
`int hashCode()`	Returns a hashcode `int` value for an object so that the object can be used in Collection classes that use hashing, including `Hashtable`, `HashMap`, and `HashSet`
`final void notify()`	Wakes up a thread that is waiting for this object's lock
`final void notifyAll()`	Wakes up *all* threads that are waiting for this object's lock
`final void wait()`	Causes the current thread to wait until another thread calls `notify()` or `notifyAll()` on this object
`String toString()`	Returns a "text representation" of the object

Overriding equals() method

If you want objects of your class to be used as keys for a hashtable (or as elements in any data structure that uses equivalency for searching for—and/or retrieving—an object), then you must override `equals()` so that two different instances can be considered the same.

The equals() Contract

Pulled straight from the Java docs, the `equals()` contract says:

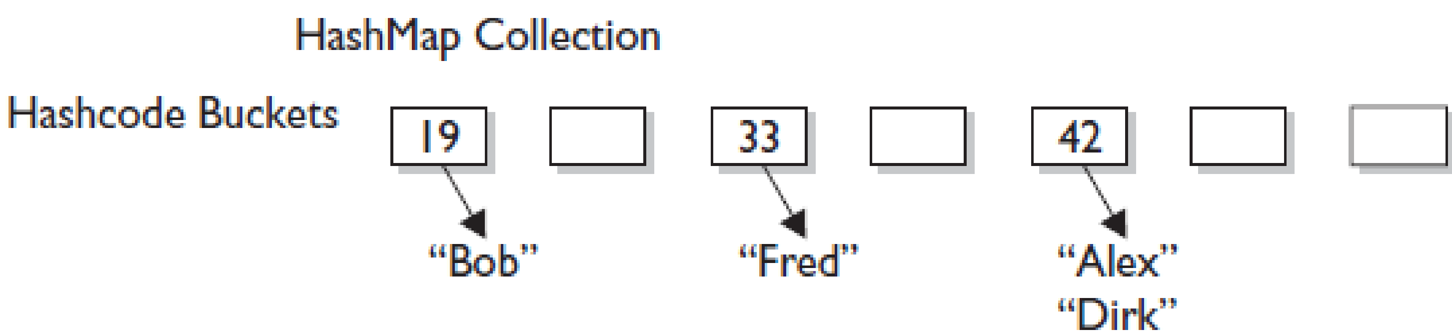
- It is **reflexive**. For any reference value `x`, `x.equals(x)` should return `true`.
- It is **symmetric**. For any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is **transitive**. For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.

The equals() Contract

- It is **consistent**. For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`
- For any non-`null` reference value `x`, `x.equals(null)` should return `false`.

Understanding Hashcodes

Key	Hashcode Algorithm	Hashcode
Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) + I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + D(4)	= 33



Now that we know that two equal objects must have identical hashcodes, is the reverse true? Do two objects with identical hashcodes have to be considered equal?

The hashCode() Contract

`hashCode()` contract:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently `return` the same `integer,` provided that no information used in `equals()` comparisons on the object is modified.

The hashCode() Contract

- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same `integer` result.
- It is NOT required that if two objects are unequal according to the equals(`java.lang.Object`) method, then calling the `hashCode()` method on each of the two objects must produce distinct `integer` results.

Condition	Required	Not Required (But Allowed)
`x.equals(y) == true`	`x.hashCode() == y.hashCode()`	
`x.hashCode() == y.hashCode()`		`x.equals(y) == true`
`x.equals(y) == false`		No `hashCode()` requirements
`x.hashCode() != y.hashCode()`	`x.equals(y) == false`	

Problem

1. Is the following implementation of hashCode valid:

```
int hashCode() { return 1; }
```

Yes, it is. But it is slooooooow.

Problem

1. Give an object some state (assign values to its instance variables).
2. Put the object in a `HashMap`, using the object as a key.
3. Save the object to a file using serialization without altering any of its state.
4. Retrieve the object from the file through deserialization.
5. Use the deserialized (brought back to life on the heap) object to get the object out of the `HashMap`.

Collections overview

- A collection (sometimes called a *container*) is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- Collections typically represent data items that form a natural group, like a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a collection of name-to-phone-number mappings).

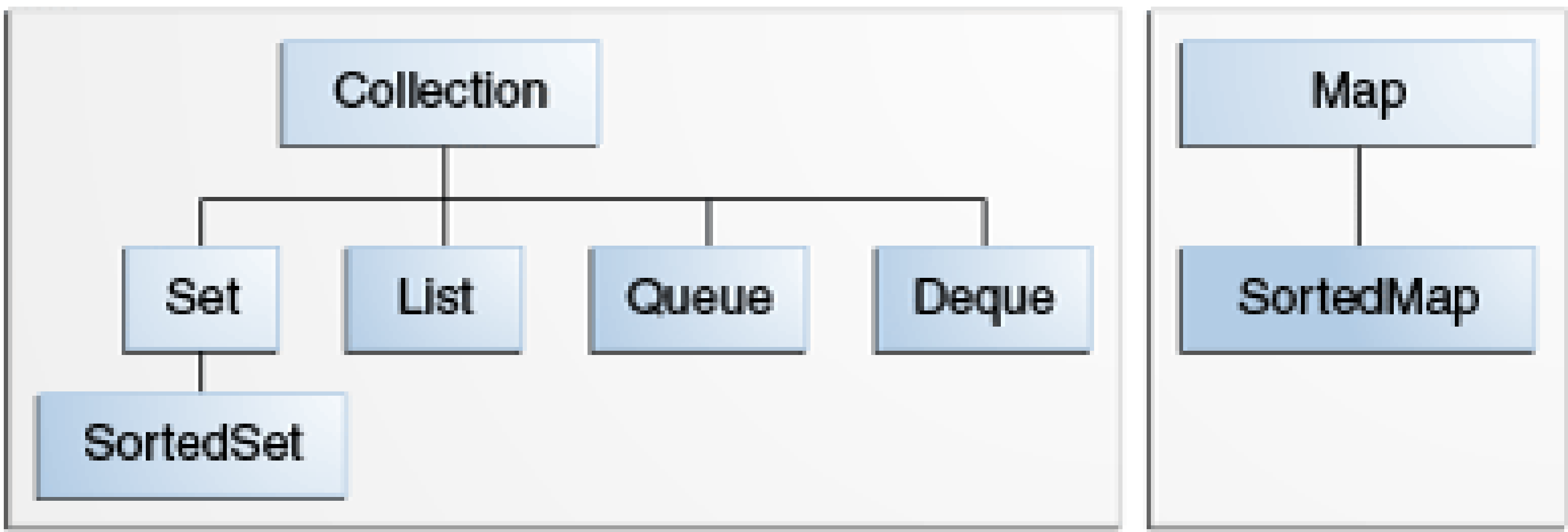
Collections framework

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain three things:

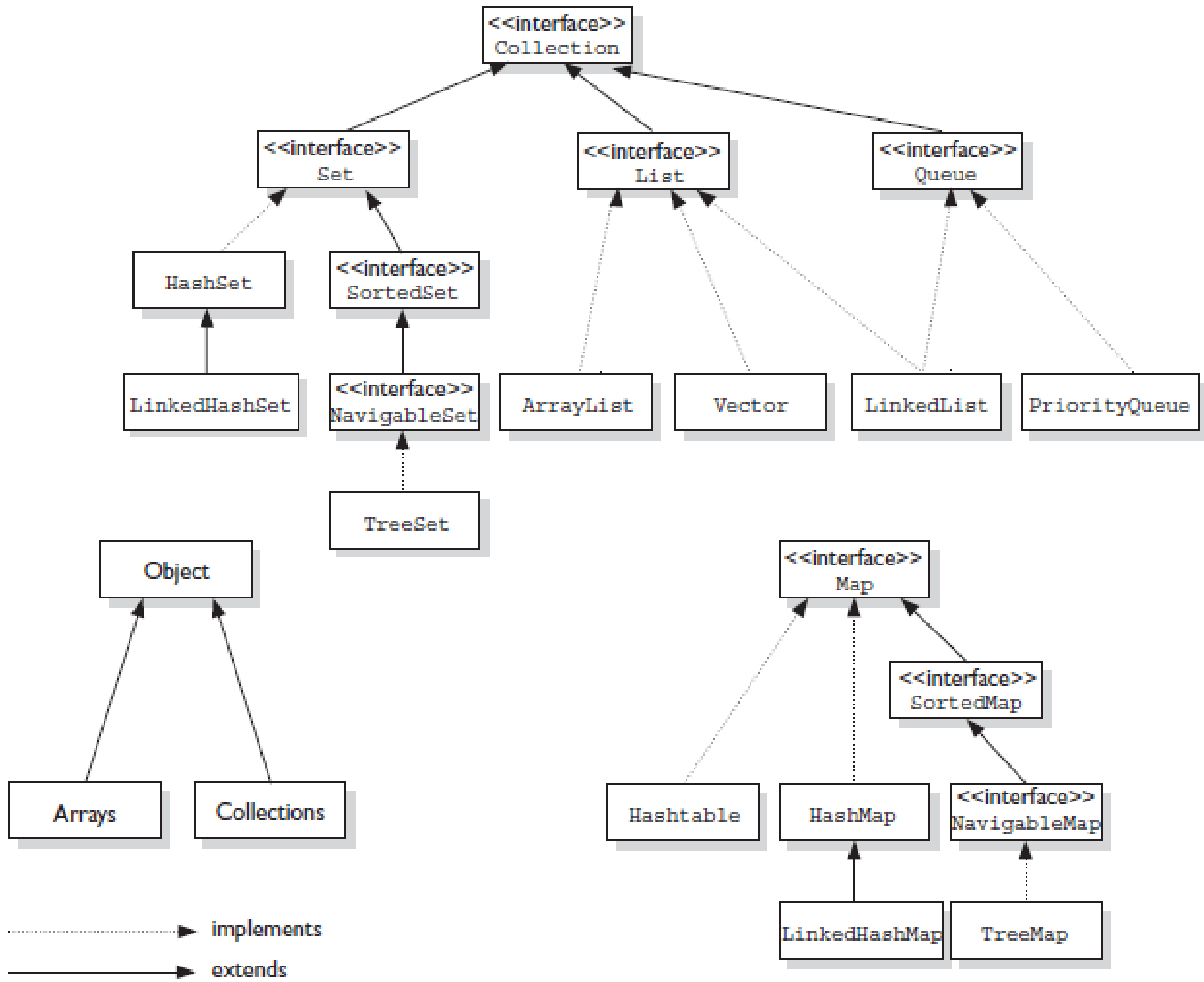
- **Interfaces**: abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation.
- **Implementations**: concrete implementations of the collection interfaces.
- **Algorithms**: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.

Core collections interfaces

The **core collection interfaces** are the interfaces used to manipulate collections, and to pass them from one method to another. The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation:



The interface and class hierarchy for collections



Optional operations

- To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type.
- Instead, the modification operations in each interface are designated optional — a given implementation may elect not to support all operations.
- If an unsupported operation is invoked, a collection throws an UnsupportedOperationException.
- Implementations are responsible for documenting which of the optional operations they support.
- All of the Java platform's general-purpose implementations support all of the optional operations.

The Collection interface

A Collection represents a group of objects, known as its elements. The primary use of the Collection interface is to pass around collections of objects where maximum generality is desired. The Collection interface is shown below:

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

The Collection interface(2)

```
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c); //optional
   boolean removeAll(Collection<?> c);       //optional
   boolean retainAll(Collection<?> c);       //optional
// Removes from the target Collection all of its elements that are not also
      contained in the specified Collection.
   void clear();                             //optional
   Object[] toArray();
   <T> T[] toArray(T[] a);
}
```

Iterator

Iterator is very similar to an Enumeration , but allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics. The Iterator interface:

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

Traversing collections:

```
for (Object o : collection)
  System.out.println(o);
```

Iterator - example

```
static void filter(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (!cond(i.next()))
                i.remove();
    } //Another example:
java.util.Map result; //Creation somewhere else...
  if (result!=null) {
        java.util.Iterator i=result.entrySet().iterator();
        while(i.hasNext()) {
          java.util.Map.Entry entry=(java.util.Map.Entry)i.next();
          debug(entry.getKey()+" => "+entry.getValue()); }}
```

The Set Interface

A Set is a Collection that cannot contain duplicate elements. Set models the mathematical set abstraction. The Set interface extends Collection and contains no methods other than those inherited from Collection. It adds the restriction that duplicate elements are prohibited.Two Set objects are equal if they contain the same elements.

Usage of Set example:

Suppose you have a Collection, c, and you want to create another Collection containing the same elements, but with all duplicates eliminated. The following one-liner does the trick:

Collection<T> noDups = new HashSet<T>(c);

The Set Interface usage example

```
public class FindDuplicates {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate detected: " + a);
        System.out.println(s.size() + " distinct words: " + s);
    }
}
```

The List Interface

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements.

The JDK contains two general-purpose List implementations. ArrayList , which is generally the best-performing implementation, and LinkedList which offers better performance under certain circumstances.

Two List objects are equal if they contain the same elements in the same order.

The List Interface(2)

```
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
    Collection<? extends E> c); //optional
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int from, int to);
}
```

The queue interface

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

The Deque interface

Deque is a double-ended-queue:

Type of Operation	***First Element (Beginning of the Deque instance)***	***Last Element (End of the Deque instance)***
Insert	addFirst(e) offerFirst(e)	addLast(e) offerLast(e)
Remove	removeFirst() pollFirst()	removeLast() pollLast()
Examine	getFirst() peekFirst()	getLast() peekLast()

The queue interface (2)

Each Queue method exists in two forms:

(1) one throws an exception if the operation fails

(2) the other returns a special value if the operation fails (either null or false, depending on the operation).

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

The Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. Two Map objects are equal if they represent the same key-value mappings.

The most useful methods:

```
public interface Map<K,V> {
V put(K key, V value);
V get(Object key);
V remove(Object key);
boolean containsKey(Object key);
boolean containsValue(Object value);
…}
```

The Comparable Interface

The Comparable interface consists of a single method:

```
public interface Comparable<T> {
public int compareTo(T o);
}
```

The **compareTo** method compares the receiving object with the specified object, and returns a negative integer, zero, or a positive integer as the receiving object is less than, equal to, or greater than the specified Object.

The Comparator Interface

A Comparator is an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method:

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

The compare method compares its two arguments, returning a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

SortedSet and SortedMap

A SortedSet is a Set that maintains its elements in ascending order, sorted according to the elements natural order, or according to a Comparator provided at SortedSet creation time.

A SortedMap is a Map that maintains its entries in ascending order, sorted according to the **keys** natural order, or according to a Comparator provided at SortedMap creation time.

Implementations

JDK provides two implementations of each interface (with the exception of Collection).

All implementations permit null elements, keys and values.

All are Serializable, and all support a public clone method.

Each one is unsynchronized.

If you need a synchronized collection, the **synchronization wrappers** allow any collection to be transformed into a synchronized collection.

HashSet and TreeSet

- The two general purpose Set implementations are HashSet and TreeSet (and LinkedHashSet which is between them)
- HashSet is much faster but offers no ordering guarantees.
- If in-order iteration is important use TreeSet.
- Iteration in HashSet is linear in the sum of the number of entries and the capacity. It's important to choose an appropriate initial capacity if iteration performance is important. The default initial capacity is 101. The initial capacity may be specified using the int constructor. To allocate a HashSet whose initial capacity is 17:

Set s= new HashSet(17);

ArrayList and LinkedList

The two general purpose List implementations are ArrayList and LinkedList . **ArrayList** offers constant time positional access, and it's just plain fast, because it does not have to allocate a node object for each element in the List, and it can take advantage of the native method System.arraycopy when it has to move multiple elements at once.

If you frequently add elements to the beginning of the List, or iterate over the List deleting elements from its interior, you might want to consider **LinkedList**. These operations are constant time in a LinkedList but linear time in an ArrayList. Positional access is linear time in a LinkedList and constant time in an ArrayList.

HashMap and TreeMap

The two general purpose Map implementations are HashMap and TreeMap . And LinkedHashMap (similar to LinkedHashSet)

The situation for Map is exactly analogous to Set.

If you need SortedMap operations you should use TreeMap; otherwise, use HashMap.

Synchronization wrappers

The **synchronization wrappers** add automatic synchronization (thread-safety) to an arbitrary collection. There is one static factory method for each of the six core collection interfaces:

public static Collection synchronizedCollection(Collection c);

public static Set synchronizedSet(Set s);

public static List synchronizedList(List list);

public static Map synchronizedMap(Map m);

public static SortedSet synchronizedSortedSet(SortedSet s);

public static SortedMap synchronizedSortedMap(SortedMap m);

Each of these methods returns a synchronized (thread-safe) Collection backed by the specified collection.

Unmodifiable wrappers

Unmodifiable wrappers take away the ability to modify the collection, by intercepting all of the operations that would modify the collection, and throwing an **UnsupportedOperationException.** The unmodifiable wrappers have two main uses:

- To make a collection immutable once it has been built.
- To allow "second-class citizens" read-only access to your data structures. You keep a reference to the backing collection, but hand out a reference to the wrapper. In this way, the second-class citizens can look but not touch, while you maintain full access.

Unmodifiable wrappers(2)

There is one static factory method for each of the six core collection interfaces:

public static Collection unmodifiableCollection(Collection c);

public static Set unmodifiableSet(Set s);

public static List unmodifiableList(List list);

public static Map unmodifiableMap(Map m);

public static SortedSet unmodifiableSortedSet(SortedSet s);

public static SortedMap unmodifiableSortedMap(SortedMap m);

Java 1.0/1.1 containers

A lot of code was written using the Java 1.0/1.1 containers, and even new code is sometimes written using these classes. So although you should never use the old containers when writing new code, you'll still need to be aware of them. Here are older container classes:

Vector (~=ArrayList)

Enumeration (~= Iterator)

Hashtable (~=HashMap)

Stack (~=LinkedList)

All of them are synchronized (and slower).

Questions

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)
            ++count;
    return count;
}
```

Will it compile?

No. So how to repair it?

```
public interface Comparable<T> {
public int compareTo(T o); }
```

```
public static <T extends Comparable<T>> int countGreaterThan(T[]
anArray, T elem) {
... if (e.compareTo(elem) > 0) ...
```

Questions

Given the following classes:

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
class Node<T> { /* ... */ }
```

Will the following code compile?

```
Node<Circle> nc = new Node<>();
Node<Shape> ns = nc;
```

Answer: No. Because Node<Circle> is not a subtype of Node<Shape>.