Topic 4

Java Collection Framework and Generics/Parameterized Typing

Learning Outcomes

- **Describe** the general concept of a collection in computer programming
- **Describe and Document Diagrammatically** the OO design of the Java Collection Framework (JCF) and **apply** this framework in Java code
- **Explain** the difference between `List`, `Set` and `Map` and their implementations and when these are appropriate for a given programming problem. **Demonstrate** their use in Java code.
- **Demonstrate** the use of Iterator and **for-each** loop in Java and the JCF
- **Describe** the `java.util.Object equals()` and `hashCode()` methods and **demonstrate** how these are used in JCF code
- **Describe** the use of parameterized types/generics in Java and **demonstrate** how these are used in JCF code
- **Write** your own generic classes and methods in Java to facilitate code reuse and extensibility
- **Describe** and **apply** bounded generic parameters and generic methods in Java code

Java Collections

- Collection: An Object that groups multiple elements into a single unit
- Typically represents data items belonging to a natural group, such as Library Catalogue (a collection of library materials and borrow-status)
- To store, retrieve and manipulate data (e.g. through polymorphic operations), and to transmit data from one method to another

JAVA Collection Implementation in earlier versions (still supported)

1. `java.util.Vector` - collection of Java Objects (contents can be instantiated from different classes) without prior knowledge of the size
2. `java.util.Hastable` – collection of name-value pairs, like a dictionary, easy lookup
3. `java.util.Stack` – LIFO structure, extension of Vector with push(), pop() etc.
4. `Array` – collection of objects of same type (class or primitive data) with known fixed size

Static versus dynamic types and abstraction

- When we instantiate an object in Java we actually specify two types:

  `List myList=new ArrayList();`
- The type on the left of the '=' (instantiation operator) is called the *static (*)* or *declared* type (in the above example `List` i.e. the `java.util.List` interface)
  - This type is used by the compiler to type check any references where myList is used (e.g. you cannot assign a List to a `String`, you also cannot assign a List to an `ArrayList` since not all Lists are ArrayLists (although All `ArrayList` instances are Lists)
  - Only methods or attributes that exist in the *static/declared* type can be accessed (regardless of the type on the right of '=')
  - Try to use the most Abstract type possible (highest in class hierarchy) to favour reuse and choose interfaces over classes where possible. **This applies to parameter types as well**!
- The type on the right hand side is the *dynamic* or *runtime* type. It is the actual type of the object when instantiated during runtime and determines which actual implementation of methods are used (e.g. `ArrayList.add()` versus `LinkedList.add()`).
- Having collections of interfaces or abstract types facilitates polymorphism by placing different dynamic/runtime types in the collection

(*) Not to be confused with the static java keyword!

Java Collection Framework (JCF)

A 'Unified Architecture' for representing and manipulating collections

Collection Framework contains:

• **Interfaces**: abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation.

```
java.util.Collection etc.
```

• **Implementations**: concrete implementations of the collection interfaces. In essence, these are *reusable data structures*.

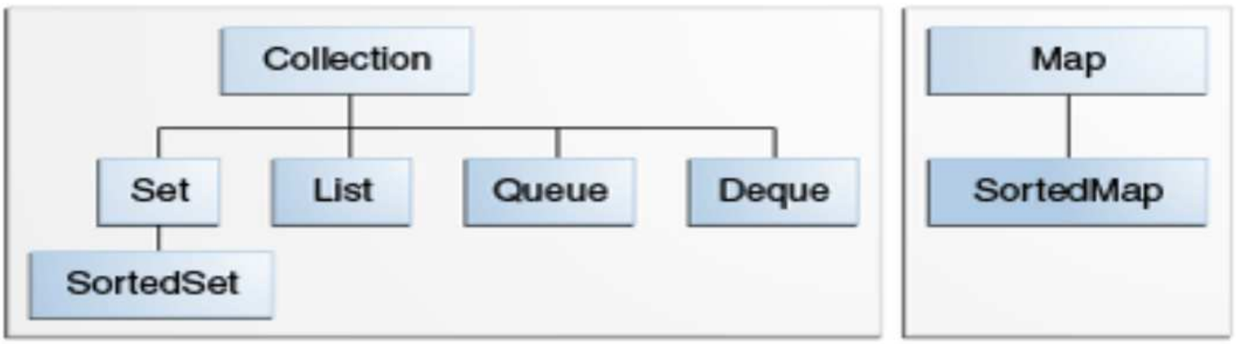
```
java.util.ArrayList etc.
```

• **Algorithms**: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces. These algorithms are said to be *polymorphic* because the same method can be used on many different implementations of the appropriate collections interface. In essence, algorithms are *reusable functionality*.

`java.util.Collections, java.util.Arrays` (note the 's' on the end!)

One of the basic features of OO Programming is *reusability*.

Collection Interface Hierarchy



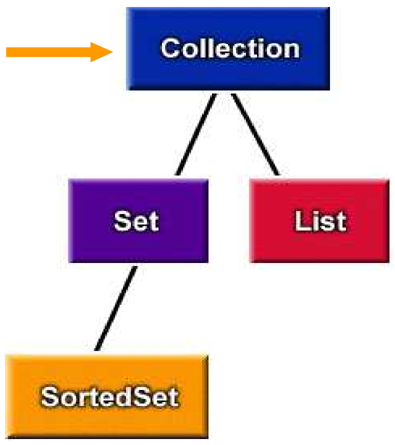
The core collection interfaces.

From: https://docs.oracle.com/javase/tutorial/collections/index.html

No unifying interface but can retrieve a Collection from a Map with Map.values()

Modification operations in each interface are designated *optional*: a given implementation may not support some of these operations. If an unsupported operation is invoked, a collection throws an UnsupportedOperationException

Interface : Collection



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. Often used for constructor param

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

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);     // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                     // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

What is Iterator?

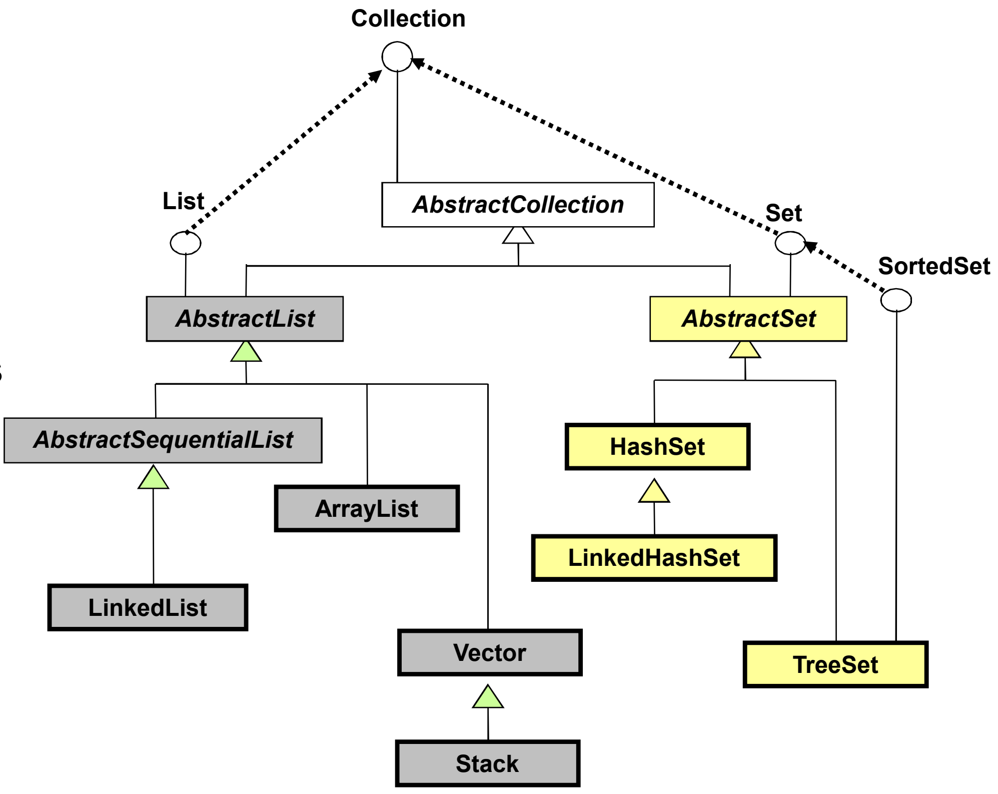
What about Iterable?

Collection Interface

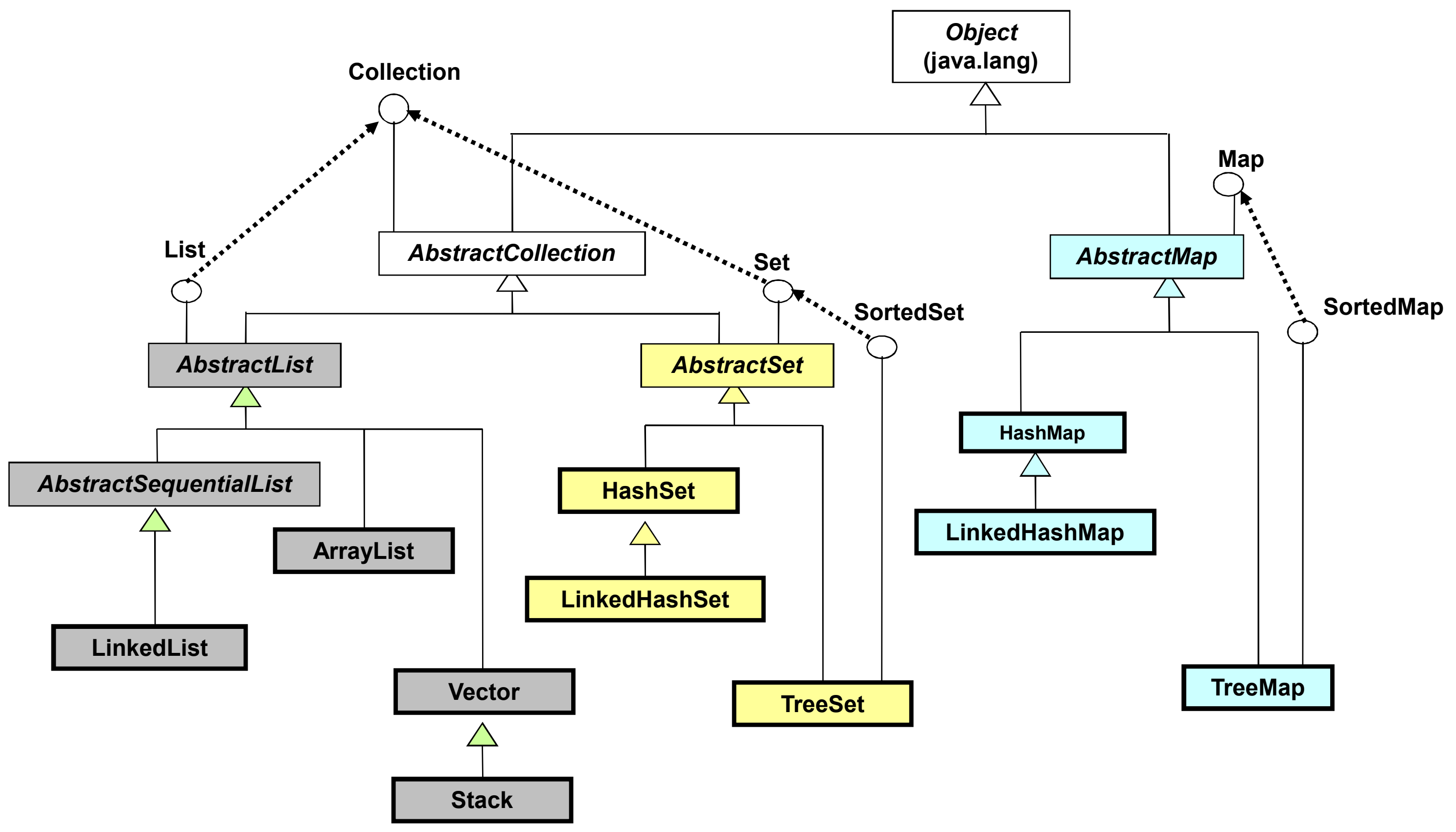
- The basic Collection interface provides operations for adding and removing elements
- The AbstractCollection class partially implements the Collection interface.

Q. Which classes allow:

- Duplicate elements?
- Elements to be ordered?
- Mixed types?



Relationship between Java Interfaces and Classes



Iterator

Iterator Interface (current)

Iterator also allows the caller to **remove** elements from the underlying collection during the iteration with well-defined semantics.

See
`sadi.topic4.iterator.IteratorTest`

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

Enumeration Interface (old)

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series. Eg, StringTokenizer

Two Methods:

- `boolean hasMoreElements()`
- `Object nextElement()`

An Iterator Example

```
c1 = new LinkedList();
// Process the linked list
Iterator iter = c1.iterator();
while (iter.hasNext()) {
   Object obj = iter.next();
   //do what you want to obj
}
c2 = new HashSet();
// Process the hash set
iter = c2.iterator();
while (iter.hasNext()) {
   Object obj = iter.next();
   //do what you want to obj
}
```

Note identical code for generating the sequence of objects

Without generics we must usually cast the object to use it properly

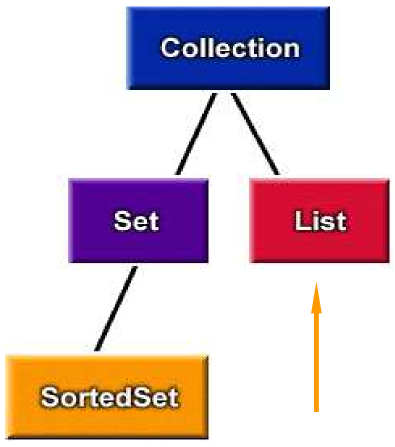
Java 1.5+: Use of for-each loop instead of an Iterator

```
c1 = new LinkedList();
// Process the linked list
Iterator iter = c1.iterator();
while (iter.hasNext()) {
   Object obj1 = iter.next();
   //do what you want to obj1
}
c2 = new HashSet();
// Process the hash set
iter = c2.iterator();
while (iter.hasNext()) {
   Object obj2 = iter.next();
   //do what you want to obj2
}
```

```
c1 = new LinkedList();
// Process the linked list
for(Object obj1: c1)
   //do what you want to obj1
}

c2 = new HashSet();
// Process the hash set
for(Object obj2: c2) {
   //do what you want to obj2
}
```

Classes that implement java.lang.Iterable interface and arrays can be used in foreach as well as Arrays

Interface: List



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

In addition to the operations inherited from Collection, the List interface includes operations for:

1. **Positional Access (indexed)**: manipulate elements based on their numerical position in the list.
2. **Search**: search for a specified object in the list and return its numerical position.
3. **List Iteration**: extend Iterator semantics to take advantage of the list's sequential nature.
4. **Range-view**: perform arbitrary *range operations* on the list.

JDK Standard Implementation (examples)
ArrayList – best general purpose and stored in a resizable array
LinkedList - better performance under certain circumstances and implements Deque interface
Vector – retrofitted to implement List (is also Thread safe)

Interface: List (Continued…)

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

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

   // Iteration
   ListIterator listIterator();
   ListIterator listIterator(int index);

   // Range-view
   List subList(int from, int to);
}
```

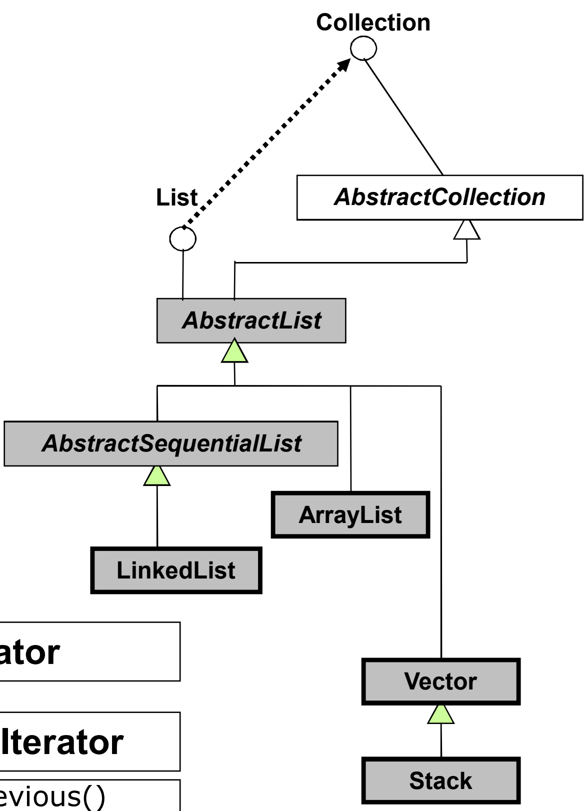
```
public interface ListIterator
   extends Iterator {

   boolean hasNext();
   Object next();

   boolean hasPrevious();
   Object previous();

   int nextIndex();
   int previousIndex();

   void remove();          // Optional
   void set(Object o);     // Optional
   void add(Object o);     // Optional
}
```

Most of the polymorphic algorithms in the **JCF** classes (e.g. java.util.Collections) apply specifically to List not Collection.

- `sort(List)`
- `shuffle(List)`
- `reverse(List)`
- `fill(List, Object)`
- `copy(List dest, List src)`
- `binarySearch(List, Object)`

The List Interface

- The List interface provides an ordered collection
- Allows duplicate elements (unlike sets)
- Allows elements to be stored/retrieved using their position or index.
- Has index operations like add(int idx, Object o)
- Provides a special ListIterator for traversing in both directions.



java.util.Collection

java.util.List

```
boolean add(int index, Object e)
Object get(int index)
Int indexOf(Object element)
ListIterator listIterator()
ListIterator listIterator(int startIndex)
List subList(int from, int to)
Object remove(int index)
```

java.util.Iterator

java.util.ListIterator

```
boolean hasPrevious()
Object previous()
int previousIndex()
int nextIndex()
```

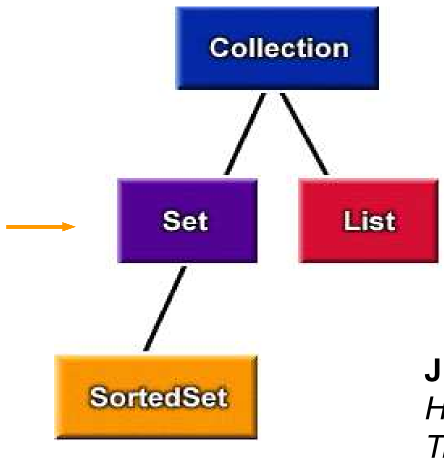
LinkedList Class

- Grows dynamically
- Implements List methods
- *Sequencial* access to data
- Faster at inserting (deleting) elements anywhere in the list.
- Slow at accessing elements because the list has to be traversed sequentially.

ArrayList Class

- Grows dynamically
- Implements List methods
- *Random* access to data
- Faster at accessing elements anywhere in the list.
- Slow at inserting (deleting) elements in the middle due to copying overhead.

Interface : Set



- 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.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations (inherited from Object), allowing Set objects with different implementation types to be compared meaningfully. Two Set objects are equal if they contain the same elements.

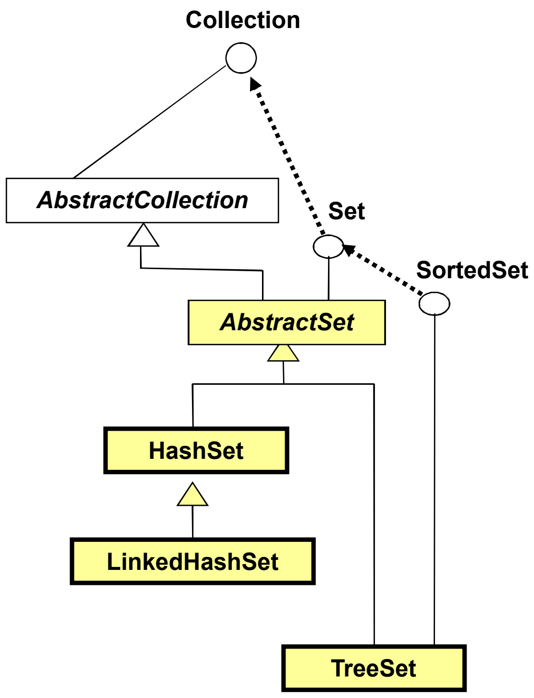
JDK Standard Implementations
HashSet – best performing and stored in hash table but random ordering
TreeSet - guarantees ordering and stored in red-black tree
LinkedHashSet – ordered by insertion order

What are the common Set Operations?

Subset	`s1.containsAll(s2)`
Union	`s1.addAll(s2)`
Intersection	`s1.retainAll(s2)`
Set Difference	`s1.removeAll(s2)`

Sets

- Set extends Collection but has no new methods
  - But classes implementing Set interface must ensure no duplicates, i.e. they cannot contain e1 and e2 if e1.equals(e2) is true
- HashSet provides a concrete implementation of Set. Elements in HashSet are not ordered
- LinkedHashSet extends HashSet with a linked list and supports an ordering. Elements can be retrieved in order of insertion
- TreeSet class provide a concrete implementation of SortedSet. It guarantees elements are sorted. Objects inserted must be *comparable* with each other



e.g. Set vs Collection add() method contract

java.util.Collection

boolean add(Object e)

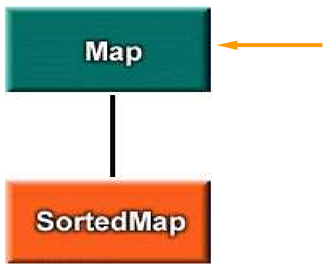
Ensures that this collection contains the specified element (optional operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)

java.util.Set

boolean add(Object e)

Adds the specified element to this set if it is not already present (optional operation). More formally, adds the specified element e to this set if the set contains no element e2 such that (e==null ? e2==null : e.equals(e2)). If this set already contains the element, the call leaves the set unchanged and returns false. In combination with the restriction on constructors, this ensures that sets never contain duplicate elements.

Interface: Map



A Map is a collection that maps keys to values.

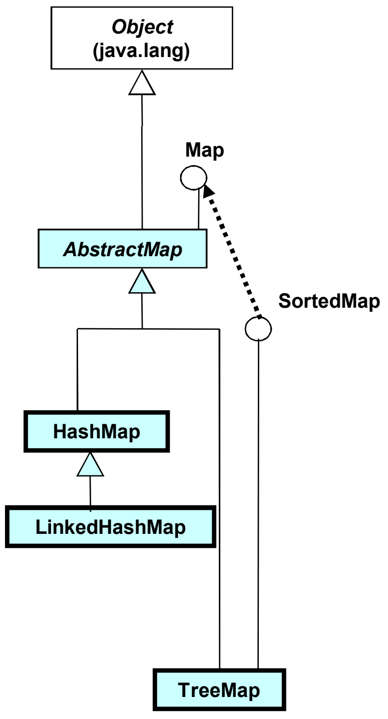
A map cannot contain duplicate keys: Each key can map to at most one value.

JDK Standard Implementation (examples)
HashMap – best performing and stored in hash table.
TreeMap – guarantees ordering and stored in red-black tree
Hashtable – retrofitted to implement Map

```
public interface Map
{
    // Basic Operations                        // Collection Views
    Object put(Object key, Object value);      public Set keySet();
    Object get(Object key);                    public Collection values();
    Object remove(Object key);                 public Set entrySet();
    boolean containsKey(Object key);
    boolean containsValue(Object value);       // Interface for entrySet elements
    int size();                                public interface Entry
    boolean isEmpty();                         {
                                                   Object getKey();
    // Bulk Operations                             Object getValue();
    void putAll(Map t);                            Object setValue(Object value);
    void clear();                              }
                                           }
```

Implementation of Map Interface

- HashMap
  - Efficient for locating a value, inserting/deleting mapping
  - Entries not ordered
  - Pre JDK 1.2 equivalent is HashTable (redesigned to fit JCF but methods retained for compatibility)
- LinkedHashMap
  - Extends HashMap with a linked list
  - Can be accessed in order of insertion/access
- TreeMap
  - Implements the interface SortedMap and efficient for accessing maps in sorted order
  - Comparison can be done using the compareTo() method (default) or by passing a Comparator to the TreeMap constructor



Queue and Deque

- `java.util.Queue` - typically a FIFO (first-in, first-out) structure but may be ordered arbitrarily, for example by priority
  - Removal is always from the head of the queue via a call to `remove()` or `poll()`
  - In a FIFO queue elements are inserted at the tail of the queue but other kinds of queues may use different placement rules
  - Implementations: see for example `LinkedList`, `PriorityQueue`
- `java.util.Deque` - can be used as both FIFO (first-in, first-out) and LIFO (last-in, first-out).
  - extends the `Queue` interface
  - new elements can be inserted, retrieved and removed at both ends
  - Implementations: see for example `LinkedList`, `ArrayDeque`, `LinkedBlockingDeque`

java.util.Object equals() and hashCode() methods

- `java.util.Collection` operations use the `java.lang.Object.equals(...)` and `hashCode()` methods
- e.g. for a `Map` it is the hash of the key that is stored not the key itself
- if object1 and object2 are equal according to their equals() method, they must also have the same hash code
- If object1 and object2 have the same hash code they do NOT have to be equal (but may be)
- described further in `java.lang.Object` API docs
- the default Object implementations give you a very strict interpretation .. only equal if comparing the same object
- Most Java API classes (e.g. String) already provide suitable implementations
- as a rule if you override one you should override both however especially if you are using a class as a key you must override these methods together correctly (**or more <u>simply</u> use both Object defaults for strict comparison**)

Comparable and Comparator Interface

If a sorted Collection consists of String elements, it will be sorted into lexicographic (alphabetical) order. If it consists of Date elements, it will be sorted into chronological order. How?

Both String and Date implements the ***<u>Comparable</u>*** interface that provides a ***natural ordering*** for a class, which allows objects of that class to be sorted automatically.

- for a list whose elements do not implement Comparable, Collections.sort(list) will throw a <u>ClassCastException</u>.

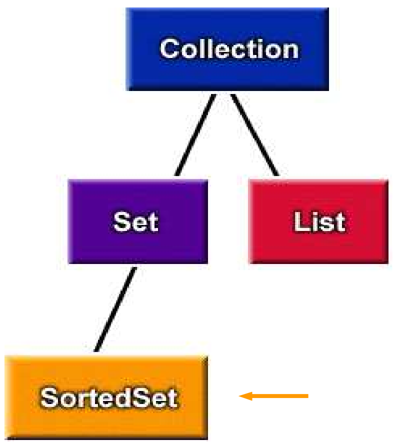
- for a list whose elements cannot be compared *to one another*, Collections.sort will throw a <u>ClassCastException</u>.

- Elements that can be compared to one another are called *mutually comparable*. While it is possible to have elements of different types be mutually comparable, none of the JDK types (Byte, Character, Long, Integer, Short, Double, Float, String, Date) permit inter-class comparison.

```
public interface Comparable
{
   public int compareTo(Object o);
}
```

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

Interface: SortedSet



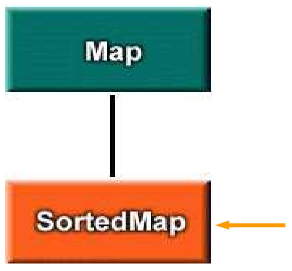
A **SortedSet** is a Set that maintains its elements in ascending order, sorted according to the elements' ***natural order***, or according to a Comparator provided at SortedSet creation time.

In addition to the normal Set operations, the SortedSet interface provides operations for:

Range-view: Performs arbitrary *range operations* on the sorted set.
Endpoints: Returns the first or last element in the sorted set.
Comparator access: Returns the Comparator used to sort the set

```
public interface SortedSet extends Set {
    // Range-view
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet headSet(Object toElement);
    SortedSet tailSet(Object fromElement);

    // Endpoints
    Object first();
    Object last();

    // Comparator access
    Comparator comparator();
}
```

Interface: SortedMap



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

In addition to the normal Map operations, the SortedMap interface provides operations for:

Range-view: Performs arbitrary *range operations* on the sorted map.
Endpoints: Returns the first or last key in the sorted map.
Comparator access: Returns the Comparator used to sort the map

```
public interface SortedMap extends Map {
    // Range-view
    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    // Endpoints
    Object first();
    Object last();

   // Comparator access
    Comparator comparator();

}
```

Implementation

Implementations are the actual data objects used to store collections, which implement the *core collection interfaces*

General-purpose Implementations
General-purpose implementations are the public classes that provide the primary implementations of the core collection interfaces. e.g. ArrayList, HashMap

Convenience Implementations
Convenience implementations are mini-implementations, typically made available via *static factory methods* that provide convenient, efficient alternatives to the general-purpose implementations for special collections e.g. Collections.singletonList().

Wrapper Implementations
Wrapper implementations are used in combination with other implementations (often the general-purpose implementations) to provide added functionality. e.g. Collections.unmodifiableCollection();

General Purpose Implementations

Two implementations for each interface except Collection has been provided.

		Implementations			
		Hash Table	***Resizable Array***	***Balanced Tree***	***Linked List***
Interfaces	***Set***	*HashSet*		*TreeSet*	
	List		*ArrayList, Vector*		*LinkedList*
	Map	*HashMap*		*TreeMap*	

(Reference : Java Tutorial – Joshua Bloch, collection trail)

The Big-O Notation

A common way of assessing algorithm complexity (performance) is **Big-O** notation. This is an evaluation of how processing time or memory usage increases as the volume of data (N) increases.

A crude Linear Search may examine *every* item in the collection. Therefore, running time has a direct linear relationship to volume of data. We call this a performance of **O(N)**, where N is the volume of data. This means that the complexity of the search is linearly related to the size of the data. Examples of Big-O(N): N, 2N, 1/2 N, N + 3, but not N^2, N^3, Log(N), 10^N

Big-O means that the complexity *is an approximation of the specified N*

A Linear Search should stop when the correct key is found.

- ***Best*** case is when the key is in the first position. A performance where only one operation is needed is **O(1)**.
- ***Worst*** case is when the key is in the last position and all items must be examined: **O(N)**.
- ***Average*** case is when the key is about halfway, giving a performance of **O(N/2)**.

Java API Arrays.sort() algorithm uses Merge Sort which has **O(N * log(N))** performance

See http://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions

Performance Summary – Big O

List implementations:

	get	add	contains	next	remove(0)	iterator.remove
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWrite-ArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

Set implementations:

	add	contains	next	notes
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

https://stackoverflow.com/questions/559839/big-o-summary-for-java-collections-framework-implementations

(From Philip Wadler, Maurice Naftalin, *Java Generics and Collections - Speed Up the Java Development Process*, O'Reilly Media, 2009)

Performance Summary – Big O (contd.)

Map implementations:

	get	containsKey	next	Notes
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

Queue implementations:

	offer	peek	poll	size
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)
DelayQueue	O(log n)	O(1)	O(log n)	O(1)
LinkedList	O(1)	O(1)	O(1)	O(1)
ArrayDeque	O(1)	O(1)	O(1)	O(1)
LinkedBlockingDeque	O(1)	O(1)	O(1)	O(1)

Pros and Cons of JCF

Pros

1. Reduces Programming Effort
2. Increases Speed and Quality (?)
3. Allows interoperability among unrelated code
4. Reduces learning effort (?)
5. Reduces design effort
6. Fosters software reuse

Cons

1. Complexity (?)

Use of Generic classes

- Java 5+ allow the use of Generic classes and methods
- Traditionally Java programmers have used inheritance & polymorphism to create flexible classes.
- A flexible Stack class can be written to store any object by creating an array of Object references
- However it does not prevent the wrong type of object being added to the stack at compile time – resulting in a runtime error at a later time.
- It also requires casting when an object is retrieved.

```
class Stack
{
   private Object elems[];
   void push(Object o) {…}
   Object pop() {…}
}

Stack custStack = new Stack(10);

custStack.push(new Customer(…));
custStack.push(new Customer(…));
custStack.push(new Account(…));
…
Cutomer c = (Customer)
custStack.pop();
…
```

Using JCF Generic classes (Parameterized Types)

- All the JCF classes we have seen before are created to be used as generic classes
- For example, an ArrayList instance can be created to store only Account objects by passing the type (Account) to the constructor and the reference as in:

```
List<Account> accList = new ArrayList<Account>();
```

- Similarly to map a customer name (String) to Account objects we can use:

```
Map<String,Account> hashMap = new
   HashMap<String,Account>();
```

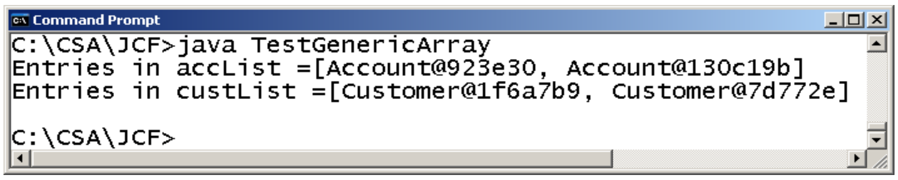
- The same classes when used without specifying any type information reduces to raw type which is equivalent to:

```
List<Object> accList = new ArrayList<Object>();
Map<Object,Object> hashMap = new
   HashMap<Object,Object>();
```

Example using ArrayList

```
import java.util.*;
class Account {  }
class Customer { }

public class TestGenericList {
   public static void main(String[] args) {
      List<Account> accList = new ArrayList<Account>();
      accList.add( new Account() );
      accList.add( new Account() );

      List<Customer> custList = new ArrayList<Customer>();
      custList.add( new Customer() );
      custList.add( new Customer() );
//    custList.add( new Account() );

      System.out.println("Entries in accList =" + accList);
      System.out.println("Entries in custList =" + custList);

   }
}
```

compiler detects type mismatch

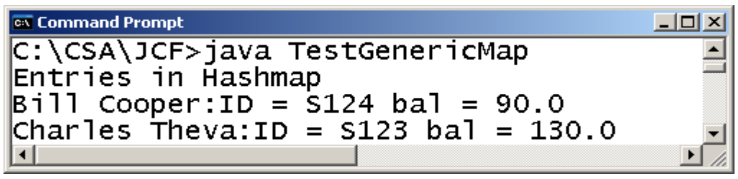


```
C:\CSA\JCF>java TestGenericArray
Entries in accList =[Account@923e30, Account@130c19b]
Entries in custList =[Customer@1f6a7b9, Customer@7d772e]

C:\CSA\JCF>
```

Example using HashMap

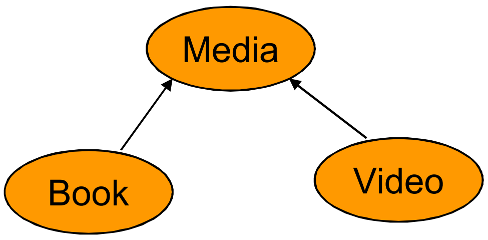
```
import java.util.*;
Customer Account { …
public class TestGenericMap {
  public static void main(String[] args) {
     Map<String,Account> hashMap =
                  new HashMap<String,Account>();
      hashMap.put("Charles Theva", new Account("S123",130.0));
      hashMap.put("Bill Cooper", new Account("S124",90.0));
//    hashMap.put(1234, new Account("S126",220.0));
      System.out.println("Entries in Hashmap");
      displayMap(hashMap);
   }
   public static void displayMap(Map<Integer, Account> m)
   {
      Set<Integer> keySet = m.keySet();
      Iterator<Integer> iterator = keySet.iterator();
      while (iterator.hasNext())
      {
         Integer key = iterator.next();
         System.out.println(key + ":"
                + m.get(key));
      }
   }
}
```

Compiler detects type mismatch



```
C:\CSA\JCF>java TestGenericMap
Entries in Hashmap
Bill Cooper:ID = S124 bal = 90.0
Charles Theva:ID = S123 bal = 130.0
```

Consider a simple library system



```
import java.util.*;
class Library {
   private List resources =
             new ArrayList();
   public void add(Media x) {
      resources.add(x);
   }
   public Media retrieveLast() {
      int size = resources.size();
      if (size > 0) {
         return (Media)
           resources.get(size-1);
      }
      return null;
   }
}
```

```
public class TestLibrary
{  public static void main(String  args[])
   {  Library myBooks =  new Library();
      myBooks.add(new Book());
      myBooks.add(new Book());
//    myBooks.add(new Video());
      Book lastBook = (Book) myBooks.retrieveLast();
      lastBook.print();
   }
}
```

For storing Book objects

Cannot detect at compile time resulting in runtime error when retrieving

Explicit cast needed

A Parameterized Library class

The generic library class has a single type parameter E (for Element type), allowing it to store objects of type E. 'T' for type is another common variable name for generic type parameters.

```
import java.util.*;

class Library<E>
{
   private List<E> resources = new ArrayList<E>();

   public void add(E x)
   {
      resources.add(x);
   }

   public E retrieveLast() {
      int size = resources.size();
      if (size > 0) {
         return resources.get(size-1);
      }
      return null;
   }
}
```

Using parameterized type E

Uses the services of parameterized ArrayList reference

Constructor

Allows any object of type E to be added

Objects retrieved are of type E

Using the Parameterized Library

- When using the parameterized (generic) Library class a type must be passes to the type parameter E.
- Note the element extracted from the parameterized Library need not be cast.

Creating a Library to store Book objects

```
public class TestLibrary {
   public static void main(String args[]) {
      Library<Book> myBooks =  new Library<Book>();
      myBooks.add(new Book());
      myBooks.add(new Book());
      Book lastBook = myBooks.retrieveLast();
      lastBook.print();

      Library<Video> myVideos =  new Library<Video>();
      myVideos.add(new Video());
      myVideos.add(new Video());
      myVideos.add(new Video());
      Video lastVideo = myVideos.retrieveLast();
      lastVideo.print();
   }
}
```

No casting needed

Creating a Library to store Video objects

Things to note when creating and using Parameterized classes

- Primitive types cannot be passed as parameters.

```
List<Integer> numbers = new ArrayList<Integer>();   ✓
List<int> numbers = new ArrayList<int>();           ✗
```

- When a class uses parameterized type T, the type parameter T can be used as a reference but not for constructing.

```
T object = …      ✓
T[] a = …         ✓
         = new T();      ✗
         = new T[10];    ✗
```

- Though the parameter cannot be used as a constructor it can be used for casting

```
E e2=(E) new Object();        ✓
E[] e3 = (E[])new Object[10]; ✓
```

- Generic classes cannot be array base type (but can be a parameterized collection)

```
Library<Video>[] videoLibs =  new Library<Video>[10];   ✗
List<Video>[] vidLibs =  new ArrayList<Video>[10];      ✗
```

See https://docs.oracle.com/javase/tutorial/java/generics/ for more info.

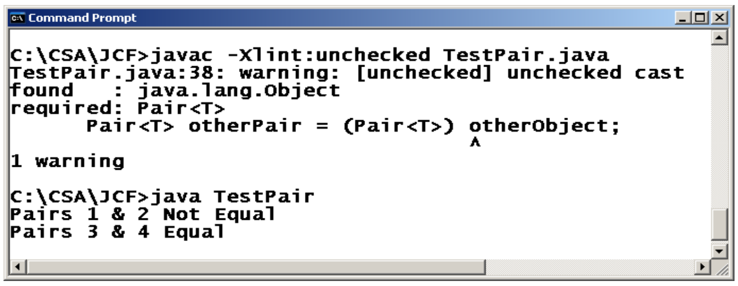
Comparison of Parametric classes

- The next program shows how a generic class could provide a method to compare two objects for equality. The Pair class keeps a pair of objects of the same type. It takes that type as a parameter.
- The equals method takes an Object reference to the other object. The Object reference is cast to the type of the current object using:
  - `Pair<T> otherPair = (Pair<T>) otherObject;`
- A cast such as this generates compiler warnings. To see the details of warnings compile with
  - `javac -Xlint:unchecked TestPair.java`
- The equals method first verifies it is an objects of the same class before comparing the parts that make up the two objects.

```
class Pair<T>
{
   private T first;
   private T second;
   public Pair()
   {
      first = null;
      second = null;
   }
   public Pair(T first, T second)
   {
      this.first = first;
      this.second = second;
   }
   @Override
   public boolean equals(Object otherObject)
   {
      if (otherObject == null) return false;
      if (getClass() != otherObject.getClass())
         return false;
      Pair<T> otherPair = (Pair<T>) otherObject;
      return (first.equals(otherPair.first)
               && second.equals(otherPair.second));
   }
}
```

Testing the Pair class

```
public class TestPair
{  public static void main(String args[])
   {
      Pair<String> pair1 = new Pair<String>("10+5", "20+5");
      Pair<String> pair2 = new Pair<String>("15","25");
      if (pair1.equals(pair2))
         System.out.println("Pars 1 & 2 Equal");
      else System.out.println("Pairs 1 & 2 Not Equal");
      Pair<Integer> pair3 = new Pair<Integer>(10+5, 20+5);
      Pair<Integer> pair4 = new Pair<Integer>(15,25);
      if (pair3.equals(pair4))
         System.out.println("Pairs 3 & 4 Equal");
      else System.out.println("Pairs 3 & 4 Not Equal");
   }
}
```



```
C:\CSA\JCF>javac -Xlint:unchecked TestPair.java
TestPair.java:38: warning: [unchecked] unchecked cast
found    : java.lang.Object
required: Pair<T>
      Pair<T> otherPair = (Pair<T>) otherObject;
                                    ^
1 warning

C:\CSA\JCF>java TestPair
Pairs 1 & 2 Not Equal
Pairs 3 & 4 Equal
```

Bounds for type parameters

- Parametric classes may not make sense for all possible types
- In the parameterized library class we may want restrict the parameter type to Media or its subclasses.
- We may expect all items in the library to have a catalogue number or a method to get the expiry date.
- We can specify ***upper*** bounds for the type of parameters by using `extends` clause (regardless of whether referring to *interface* or *class*) as in:
  - `class Library<E extends Media> { …`

Bounds for type parameters (contd.)

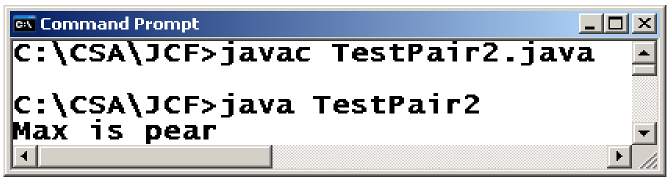
- As another example we extend the Pair class by incorporating a method named `max()` to return the bigger of its two objects.
- Java provides the `Comparable<T>` interface with a method public `int compareTo(T other)` that returns zero, negative or positive value depending on which object is larger.
- The Pair objects can be compared using this method if we specify bounds for the Parametric Pair class.
  - `class Pair<T extends Comparable<T>>`
- We can also specify ***lower*** bounds for the type of parameters by using `super` clause as in:
  - `public static <T> boolean addAll(Collection<? super T> c, T... elements) (from java.util.Collections)`
  - useful if you are doing only assignment since a `T` can technically be assigned to anything up to `Object` (but cannot be downcast below `T`) but this usage is fairly specialised

Pair class with extends bound (see ComparablePair code example)

```
class Pair2<T extends Comparable<T>> {
   private T first;
   private T second;

   public Pair() {
      first = null;
      second = null;
   }
   public Pair(T first, T second) {
      this.first = first;
      this.second = second;
   }
   public T max() {
      if (first.compareTo(second) >= 0)
         return first;
      else
         return second;
   }
}
```

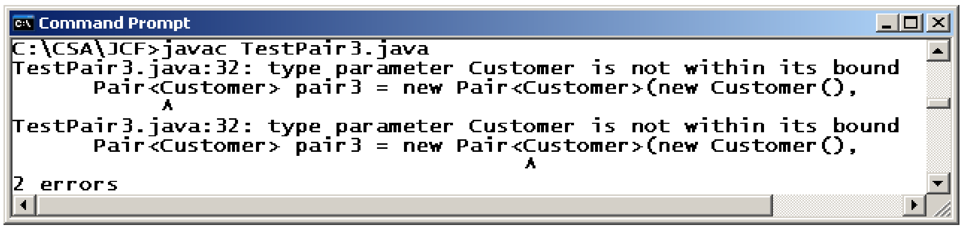
Testing the Bounded Pair class

```
public class TestPair2 {
  public static void main(String args[]) {
     Pair2<String> pair1 = new Pair2<String>("apple", "pear");
     System.out.println("Max is " +  pair1.max());
   }
}

class Customer {}

public class TestPair3 {
  public static void main(String args[]) {
      Pair2<Customer> pair3 = new Pair2<Customer>(new Customer(),
                                new Customer());
      System.out.println("Max is " +  pair3.max());
   }
}
```



```
C:\CSA\JCF>javac TestPair2.java

C:\CSA\JCF>java TestPair2
Max is pear
```

Customer does not implement Comparable



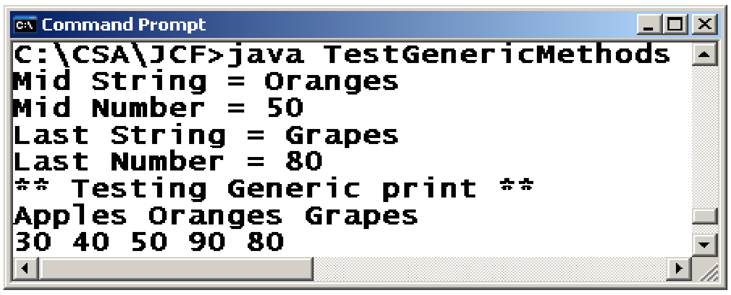
```
C:\CSA\JCF>javac TestPair3.java
TestPair3.java:32: type parameter Customer is not within its bound
      Pair<Customer> pair3 = new Pair<Customer>(new Customer(),
           ^
TestPair3.java:32: type parameter Customer is not within its bound
      Pair<Customer> pair3 = new Pair<Customer>(new Customer(),
                                      ^
2 errors
```

Generic Methods

- Generic methods can be members of generic classes or normal (ordinary) classes
- In the next example we have created a utility class that has three generic methods all declared as static.
- The method getMid() takes an array of elements of any type and returns the middle one.
- The method getLast() returns the last one.
- The print() method takes an array of elements of any type and prints them in a row.
- The generic type can by stated explicitly (className.<T> or inferred)
- **java.lang.Collections** class (note ‘s’!) contains many generic methods

```
class GenericMethods
{
   public static <T> T getMid(T[] a)
   {
      return a[a.length/2];
   }
   public static <T> T getLast(T[] a)
   {
      return a[a.length-1];
   }
   public static <T> void print(T[] a)
   {
      for (int i=0; i<a.length; i++)
         System.out.print(a[i] + " ");
      System.out.println();
   }
}
```

```
public class TestGenericMethods {
   public static void main(String args[]) {
      String s[] = { "Apples", "Oranges", "Grapes" };
      Integer nums[] = { 30 , 40 , 50, 90 , 80};

      String midS = GenMethods.<String>getMid(s);
      Integer midN = GenMethods.<Integer>getMid(nums);
      String lastS = GenMethods.<String>getLast(s);
      Integer lastNum = GenMethods.<Integer>getLast(nums);

      System.out.println("Mid String = " + midS);
      System.out.println("Mid Number = " + midN);
      System.out.println("Last String = " + lastS);
      System.out.println("Last Number = " + lastNum);
      System.out.println("** Testing Generic print **");
      GenMethods.<String>print(s);
      GenMethods.<Integer>print(nums);
   }
}
```



```
C:\CSA\JCF>java TestGenericMethods
Mid String = Oranges
Mid Number = 50
Last String = Grapes
Last Number = 80
** Testing Generic print **
Apples Oranges Grapes
30 40 50 90 80
```

Advanced

- In the Java language, arrays are covariant
  - since an Integer subclasses Number, an array of Integer is also an array of Number
- Generics are not covariant
  - a `List<Integer>` is *not* a `List<Number>`
- If covariance is required we can use a bounded wildcard for operations requiring covariance of parameterised types
  - `<? extends T>` or `<? super T>`
  - e.g. `java.lang.Collection.addAll(Collection<? extends E> c)`
  - `e.g. public static <T> boolean addAll(Collection<? super T> c, T... elements) (from java.util.Collections)`
- See `sadi.topic3.generic.TestGenericsAdvanced`
- **Further Study**: Aggregate operations on Collections using Streams
  - https://docs.oracle.com/javase/tutorial/collections/streams/