



Collections



Core Java Day 5

Day 05

01 The Collection

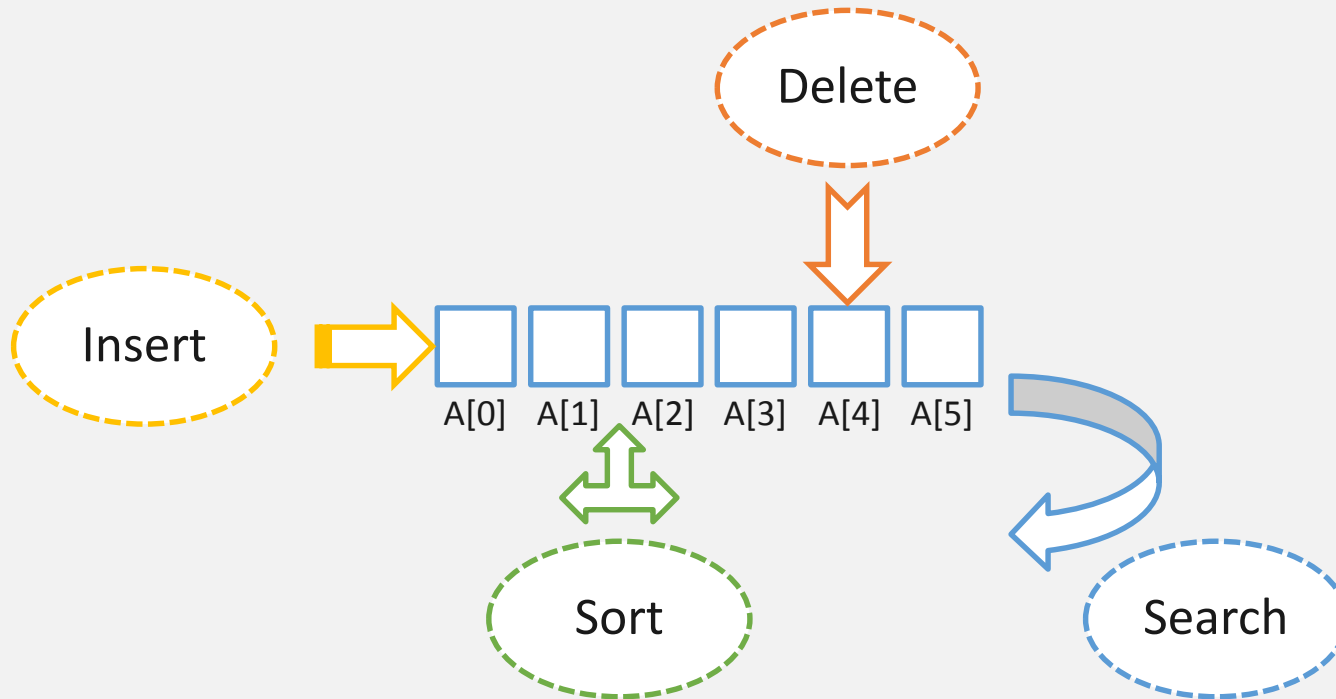


02 Generics and Annotations

01 The Collection

Arrays do not grow and they **do not have built-in methods** to perform operations such as add, remove, sort, or search.

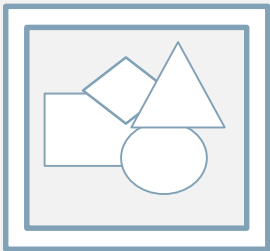
Inadequate support by Arrays while performing:



What is a Java **Collection**?



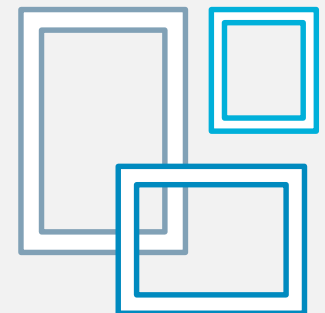
A **collection** is an object that contains a group of objects within it. These objects are called the elements of the collection.



- ▣ Collections can hold both **homogenous** and **heterogeneous** objects.
- ▣ The **elements** of a collection are objects of same class.
- ▣ Unlike Arrays, the Collections can **grow** to any size.

Collections supported by Java Collection Framework

- ▣ **Collections** : a sequence of individual elements.
- ▣ **Map** : a group of key-value object pairs.

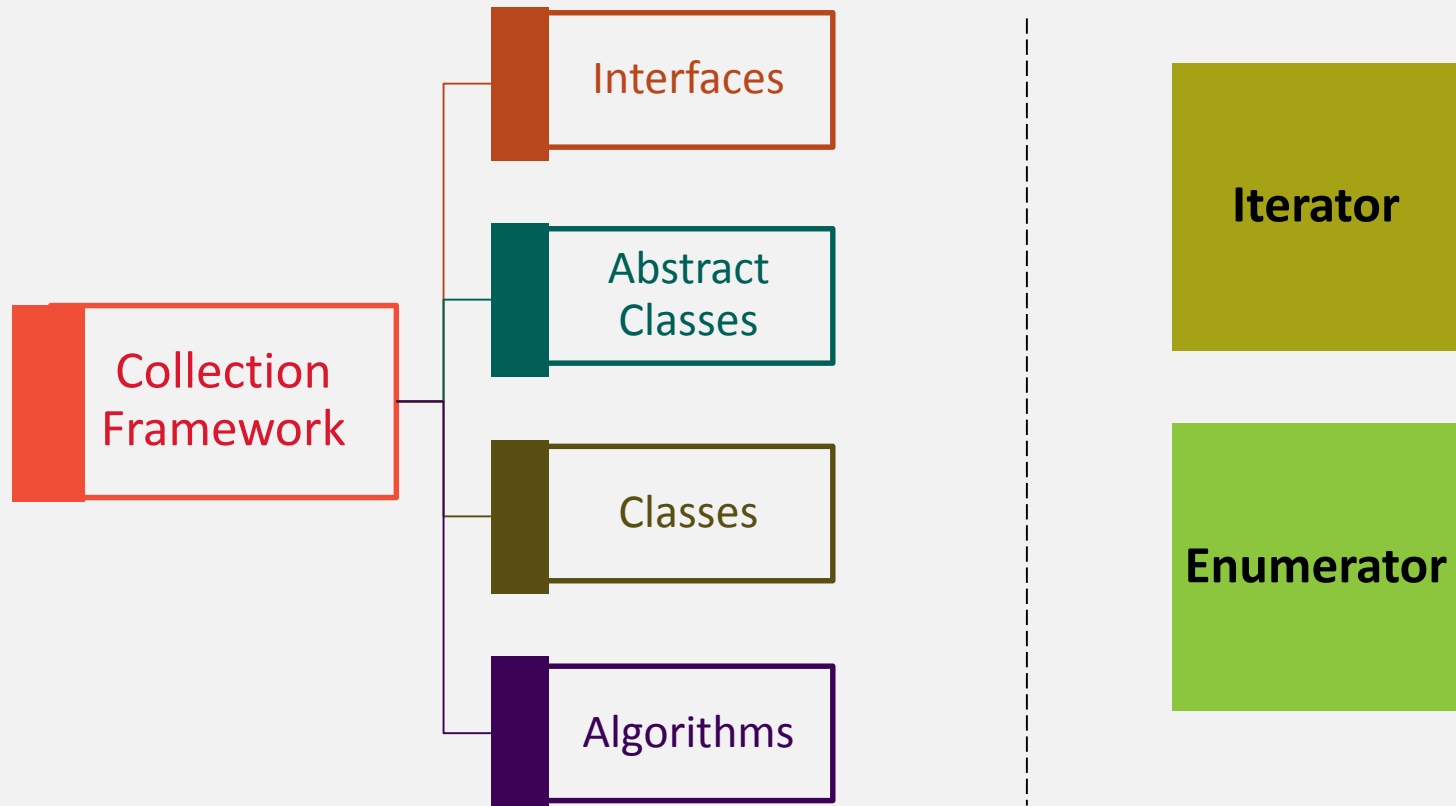


Java ad hoc classes were quite useful to store and manipulate objects. However, they lacked a design to extend. This was overcome by the **Collection** framework.

Collections Framework was designed to meet several goals:

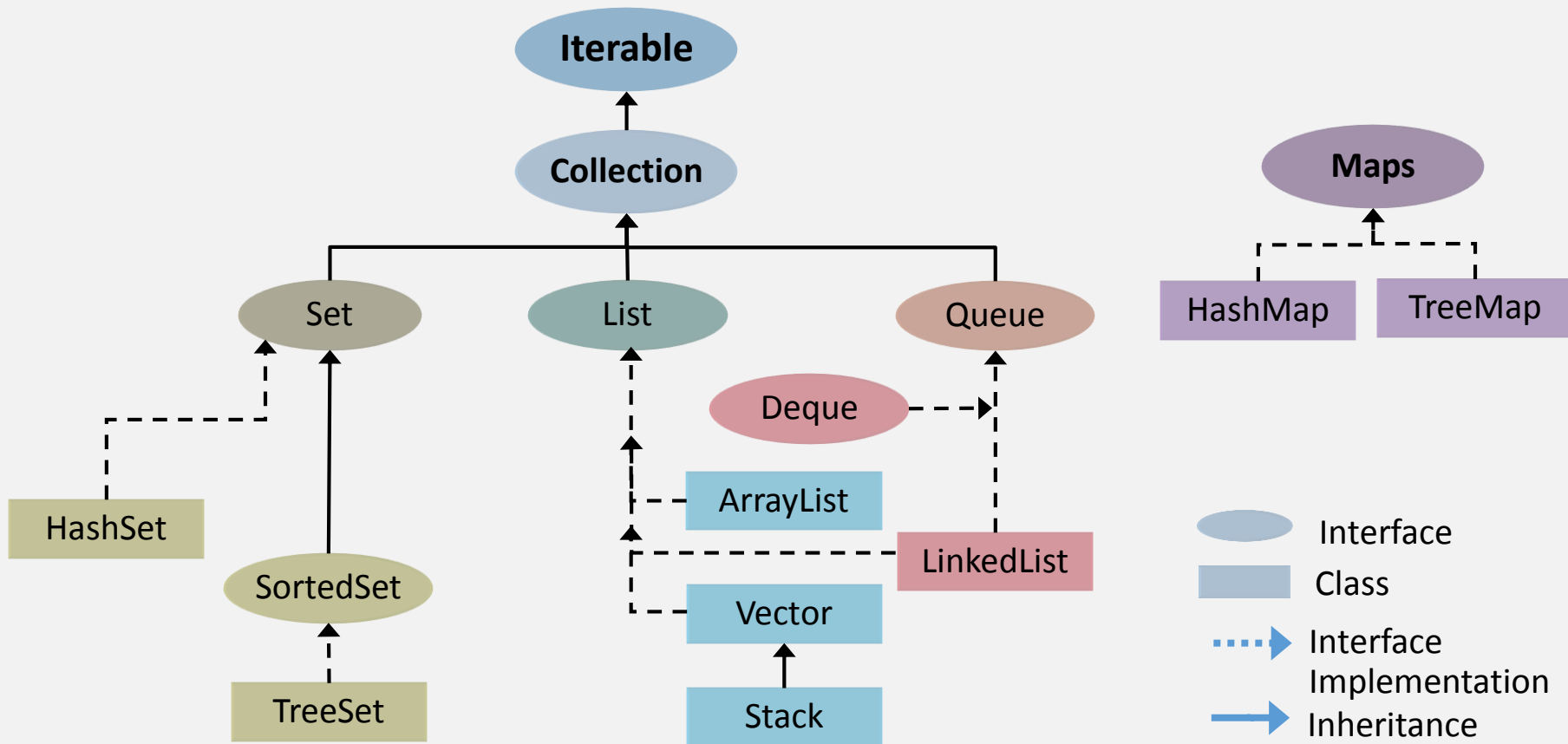
Collections Framework

- First, the framework had to be high performance.
- Second, the framework had to allow different types of collections to work in a similar manner, and with a high degree of interoperability.
- Third, extending or adapting a collection had to be easy.



Collection **Framework** provides a unified system for organizing and handling collections. In addition to collections, the framework defines several map interfaces and classes.

The **java.util** package contains all the classes and interfaces for Collection framework.



Collection Interface is the **root interface** for:

- Storing a collection of objects
- Processing a collection of objects

```
+add(element: Object): boolean
+addAll(collection: Collection): boolean
+clear(): void
+contains(element: Object): boolean
+containsAll(collection: Collection): boolean
+equals(object: Object): boolean
+hashCode(): int
+isEmpty(): boolean
+iterator(): Iterator
+remove(element: Object): boolean
+removeAll(collection: Collection): boolean
+retainAll(collection: Collection): boolean
+size(): int
+toArray(): Object[]
+toArray(array: Object[]): Object[]
```

○ Insert element into collection

```
boolean add (object o)
```

○ Remove elements from collection

```
void clear()
```

○ Match two collections

```
boolean equals (object o)
```

○ Returns number of elements

```
int size()
```

○ Insert specified collection elements

```
boolean addAll (Collection c)
```

○ To check if specified element is available

```
boolean contains (object o)
```

○ To check if collection contains all elements

```
boolean containsAll (Collection c)
```

○ To remove first occurrence of element

```
boolean remove (object o)
```

The **List** interface extends **Collection** interface, and declares the behavior of a collection that stores a sequence of elements.

```
interface List {  
    public void put (Policy policy, int position);  
    public Policy get(int position);  
}
```

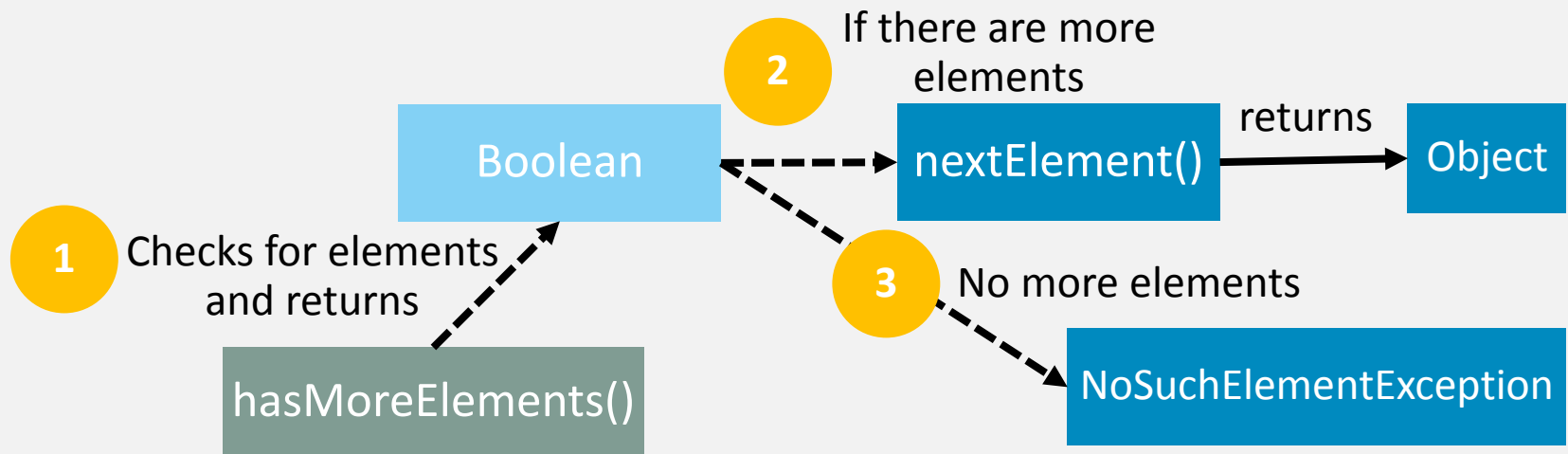
Some features of the List Interface:

- List interface allows to create a list of elements.
- The list may contain duplicate elements.
- A list also allows to specify where the element is to be stored.
- The user can access the element by index.
 - Elements can be inserted or accessed by their position in the list, using a zero-based index.

A List interface has the following methods:

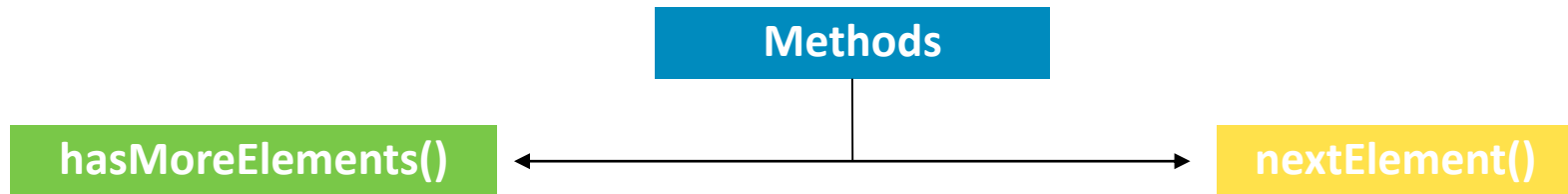
```
+add(index: int, element: Object) : void
+addAll(index: int, collection: Collection) : boolean
+get(index: int) : Object
+indexOf(element: Object) : int
+lastIndexOf(element: Object) : int
+listIterator() : ListIterator
+listIterator(startIndex: int) : ListIterator
+remove(index: int) : Object
+set(index: int, element: Object) : Object
+subList(fromIndex: int, toIndex: int) : List
```

Enumeration interface defines a way to traverse all the members of a collection of objects.



Note: Enumeration is only used in legacy classes such as Vector and Properties, this interface has been super ceded by **Iterator**.

Enumeration interface defines a way **to traverse** all the members of a **collection of objects**.



- Checks to see if there are more elements
- Then returns a Boolean

- Checks to see if there are more elements
- If yes, then returns the next element as an object

Note:

If there aren't any more elements when `nextElement()` is called, the runtime **NoSuchElementException** will be thrown.

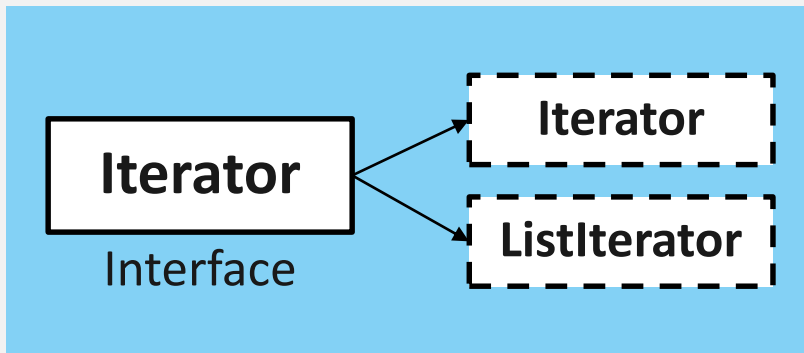
This example illustrates use of enumeration:

```
import java.util.Enumeration;
import java.util.Vector;

public class MainClass {
    public static void main(String args[]) throws Exception {
        Vector v = new Vector();
        v.add("a");
        v.add("b");
        v.add("c");

        Enumeration e = v.elements();
        while (e.hasMoreElements()) {
            Object o = e.nextElement();
            System.out.println(o);
        }
    }
}
```

Iterator is a special object to provide a way to access the elements of a collection sequentially.



- **boolean hasNext()** - Returns true if the iteration has more elements
- **Object next()** - Returns the next element in the iteration
- **void remove()** - Removes from the underlying collection the last element returned by the iterator

Iteration

- ❑ Allows manipulation of the objects
- ❑ More secure and safe because it does not allow other thread to modify the collection object

Enumeration

- ❑ Methods only to traverse and fetch the objects
- ❑ Not secure since it can modify the collection object while some thread is iterating over it


```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class MainClass {
    public static void main(String[] a) {
        Collection c = new ArrayList();
        c.add("1");
        c.add("2");
        c.add("3");
        Iterator i = c.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

ArrayList and **LinkedList** are two implementations of List interface.

Similarities between ArrayList and LinkedList

ArrayList

Beta
Alpha
Omega
Gamma
Sigma

LinkedList





Sigma
Alpha
Omega
Gamma
Beta

- Both classes are non-synchronized
- Can be made synchronized explicitly by using `Collections.synchronizedList` method
- Both maintain elements in order of insertion
- Iterator and listIterator returned by the classes are fail-fast

What does Fail-Fast mean?



- Insertions are easy and fast in **LinkedList**, as compared to **ArrayList**, because there is no risk of resizing an array and copying content to new array if the array gets full.
- ArrayList needs to update its index if you insert something anywhere except at the end of the array.
- LinkedList has more memory overhead than the ArrayList.

A r Index based data-structure	 Does not provide random/index based access
A r Searching or getting element with index is pretty fast	 Iteration over linked list is required, to retrieve any element
A r Need to update index when performing insertion	 Insertion and Removal is easy
A r Each Index only holds actual object	 Each Node holds both Data and Address of previous and next node

Implementation of List Interface (3 of 3)



Find the output using **ArrayList** and **LinkedList** for implementation of the interface.

ArrayList

```
ArrayList = new  
ArrayList();  
arrayList.add(new Integer(1));  
arrayList.add(new Integer(2));  
arrayList.add(new Integer(3));  
arrayList.add(0, new Integer(10));  
arrayList.add(3, new Integer(20));  
System.out.println(arrayList);
```

Array output

LinkedList

```
LinkedList = new  
LinkedList(arrayList);  
linkedList.add(1, "A");  
linkedList.removeLast();  
linkedList.addFirst("B");  
System.out.println(linkedList);
```

Output

??

Code of LinkedList implementation:

```
import java.util.*;
public class LinkedListDemo {
    public static void main(String args[]) {
        // Create a linked list
        LinkedList lnkList = new LinkedList();
        // Adding elements to the linked list
        lnkList.add("M");
        lnkList.add("I");
        lnkList.add("N");
        lnkList.add("D");
        lnkList.add("I");
        lnkList.addLast("A");
        lnkList.addFirst("I");
        lnkList.add(1, "B");
        System.out.println("The value of LinkedList: " + lnkList);
        // Remove first and last elements
        lnkList.removeFirst();
        lnkList.removeLast();
        System.out.println("Value after deleting first and last: "
            + lnkList);
    }
}
```

Code of FileList implementation:

```
class FileList implements List{  
    //Data Members  
    public void put (Policy policy, int position){  
        //Stores the Policy object  
    }  
    public Policy get(int position){  
        //Retrieves the Policy object  
    }  
}
```

List

Vector

```
+addElement(element: Object) : void
+capacity() : void
+copyInto(anArray: Object[]) : void
+elementAt(index: int) : Object
+elements() : Enumeration
+ensureCapacity(int minCapacity) : void
+firstElement() : int
+insertElementAt(index: int) : void
+lastElement() : Object
+removeAllElements() : void
+removeElement(element: Object) : void
+removeElementAt(index: int) : void
+setElementAt(element: Object, index: int) : void
+setSize(newSize: int) : void
+trimToSize() : void
```

The Vector class implements an expandable array of objects. Like an array, it contains components that can be accessed using an integer index.

- Vector can grow or shrink, as needed to accommodate adding and removing items after the Vector has been created.
- Vector maintains insertion order of element.
- It is similar to ArrayList, but with two differences:
 - Vector is synchronized.
 - Vector contains some legacy methods that are not part of the collections framework.

A c t i v i t y

What will be the initial size and capacity of the **vector**, how will it change after four additions?

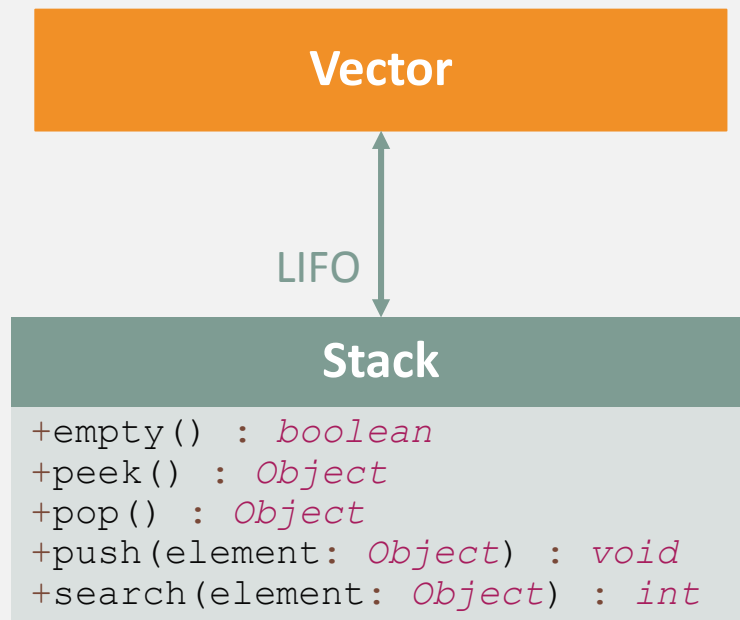
```
public static void main(String args[]) {  
    Vector v = new Vector(3, 2);  
    System.out.println("Initial size: " +  
v.size());  
    System.out.println("Initial capacity:  
" + v.capacity());  
    v.addElement(new Integer(1));  
    v.addElement(new Integer(2));  
    v.addElement(new Integer(3));  
    v.addElement(new Integer(4));  
    System.out.println("Capacity after  
four additions: " + v.capacity());  
}
```


Difference between **ArrayList** and **Vector**



- ArrayList and Vector both implement the List interface and maintain insertion order.

ArrayList	Vector
ArrayList is not synchronized.	Vector is synchronized.
ArrayList increments 50% of current array size if number of element exceeds from its capacity.	Vector increments 100% means doubles the array size if total number of element exceeds than its capacity.
ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
ArrayList uses Iterator interface to traverse the elements	Vector uses Enumeration interface to traverse the elements. But it can use Iterator also.



Stack is a subclass of Vector that implements a standard last-in, first-out stack (LIFO).

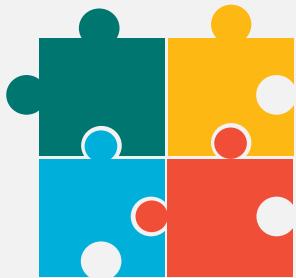
- The elements are accessed only from the top of the stack.
- You can retrieve, insert, or remove an element from the top of the stack.

A c t i v i t y

Explain how does the **stack** add and remove take place. What will be the final result?

```
public static void main(String args[]) {  
    // creating stack  
    Stack st = new Stack();  
  
    // populating stack  
    st.push("Java");  
    st.push("Source");  
    st.push("code");  
  
    // removing top object  
    System.out.println("Removed object is: "  
        +st.pop());  
  
    // elements after remove  
    System.out.println("Elements after remove: "  
        +st);  
}
```

Map interface represents a mapping between a **key** and a **value**. The Map interface is not a subtype of the Collection interface.



- A Map is a storage that maps keys to values. There cannot be duplicate keys in a Map and each key maps to at most one value.
- The Map interface is not an extension of the Collection interface. Instead the interface starts off its own interface hierarchy.
- For maintaining key-value associations, the interface describes a mapping from keys to values, without duplicate keys, by definition.
- The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.

The important methods of Map interface are:

- **void clear()**
 - Removes all mappings from this map (optional operation).
- **boolean containsKey(Object key)**
 - Returns true if this map contains a mapping for the specified key.
- **boolean containsValue(Object value)**
 - Returns true if this map maps one or more keys to the specified value.
- **Set<Map.Entry<K,V>> entrySet()**
 - Returns a set view of the mappings contained in this map.
- **boolean equals(Object o)**
 - Compares the specified object with this map for equality.
- **V get(Object key)**
 - Returns the value to which this map, maps the specified key.
- **int hashCode()**
 - Returns the hash code value for this map.

- **boolean isEmpty()**
 - Returns true if this map contains no key-value mappings.
- **Set<K> keySet()**
 - Returns a set view of the keys contained in this map.
- **V put(K key, V value)**
 - Associates the specified value with the specified key in this map (optional operation).
- **void putAll(Map<? extends K,? extends V> t)**
 - Copies all of the mappings from the specified map to this map (optional operation).
- **V remove(Object key)**
 - Removes the mapping for this key from this map if it is present (optional operation).
- **int size()**
 - Returns the number of key-value mappings in this map.
- **Collection<V> values()**
 - Returns a collection view of the values contained in this map.

01

Which method compares the specified object with the map for equality?

A

`boolean equals(Object o)`

B

`Set<K> keySet()`

C

`Collection<V> values()`

D

`boolean containsKey(Object key)`

HashMap and **TreeMap** classes are two concrete implementations of the Map interface.



HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping. It returns the unordered values.



TreeMap class implementing SortedMap, is efficient for traversing the keys in a sorted order.

Similarities:

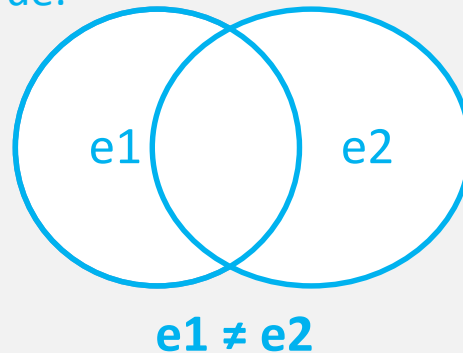
- Both HashMap and TreeMap require key and value. The keys must be unique.
- Allows one null key and many null values

Differences:

- TreeMap is slower than HashMap
- TreeMap allows us to specify an optional Comparator object during its creation. This comparator decides the order, by which the keys need to be sorted.

Set interface is used to represent a collection, which does not contain duplicate elements.

- The classes that implement set must ensure that no duplicate elements can be added to the set. For example, the elements $e1$ and $e2$ cannot be in the set such that $[(e1)eq(e2)]$ is true.



- External synchronization or wrapping is performed if multiple threads access a set concurrently, where the thread could modify the set.

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

A c t i v i t y

Review the **Interface** class shown below. How does each operations differ from one another.

```
public interface Set<E> extends Collection<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);           //optional  
    boolean remove(Object element);       //optional  
    Iterator<E> iterator()  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    //optional  
    boolean removeAll(Collection<?> c); //optional  
    boolean retainAll(Collection<?> c); //optional  
    void clear();                          //optional  
    // Array Operations  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

The Java platform contains three general purpose set implementations: **HashSet**, **TreeSet**, and **LinkedHashSet**.

#

HashSet

An unsorted, unordered set chosen when order of the elements are not important.



TreeSet

A sorted set, where elements are in ascending order.



LinkedHashSet

An ordered set, where elements are linked to one another (double-linked) to maintain the list order in which they were inserted.

Example for **Set** Interface implementation



```
import java.util.*;

public class SetInterfaceEx{
    public static void main(String[] args) {
        int a[] = {5,2,9,4,1};

        Set <Integer> hs = new HashSet<Integer>();
        for(int i=0;i<a.length;i++)    hs.add(new Integer(a[i]));
        System.out.println(hs.size() + " The HashSet is " + hs);




        Set <Integer> lhs = new LinkedHashSet<Integer>();
        for(int i=0;i<a.length;i++)    lhs.add(new Integer(a[i]));
        System.out.println(hs.size() + " The LinkedHashSet is " + lhs);

        Set <Integer> ts = new TreeSet<Integer>();
        for(int i=0;i<a.length;i++)    ts.add(new Integer(a[i]));
        System.out.println(hs.size() + " The TreeSet is " + ts);
    }
}
```

What **Set** to choose and when?



- **HashSet** is a good choice for representing sets if you don't care about element ordering. But if ordering is important, then **LinkedHashSet** or **TreeSet** are better choices. However they come with an additional speed and space cost.
- Iteration over a **LinkedHashSet** is generally faster than iteration over a **HashSet**.
- **Tree-based** data structures get slower as the number of elements get larger.
- **HashSet** and **LinkedHashSet** do not represent their elements in sorted order.
- Because **TreeSet** keeps its elements sorted, it can offer other features, such as the first and last methods, that is, the lowest and highest elements in a set, respectively.

	 HashSet	 LinkedHashSet	 TreeSet
Element ordering:	Random	Insertion	Sorted
Iteration speed:	Normal	Fast	Slow

- **SortedSet** is a sub interface of Set, which guarantees that the elements in the set are sorted.
- **TreeSet** is a concrete class that implements the SortedSet interface. You can use an iterator to traverse the elements in the sorted order.

Two ways to sort the Elements:

- **Natural order approach:** Comparable interface is used, objects can be compared using the **compareTo()** method.
- **Order by comparator approach:** Specify a comparator for the elements in the set if the class for the elements does not implement the Comparable interface, or you don't want to use the compareTo() method.

A c t i v i t y

Employ an idea to write a simple **TreeSet** code to arrange the output from HashSet.

HashSet ()

```
Set hashSet = new HashSet();  
hashSet.add("Yellow");  
hashSet.add("White ");  
hashSet.add("Green");  
hashSet.add("Orange");  
System.out.println("An unsorted set of strings");  
System.out.println(hashSet + "\n");
```

Output

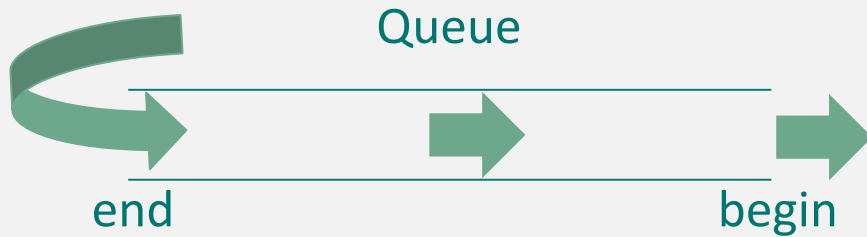
An unsorted set of strings Orange
Green
White
Yellow

TreeSet ()



Result

Green
Orange
White
Yellow

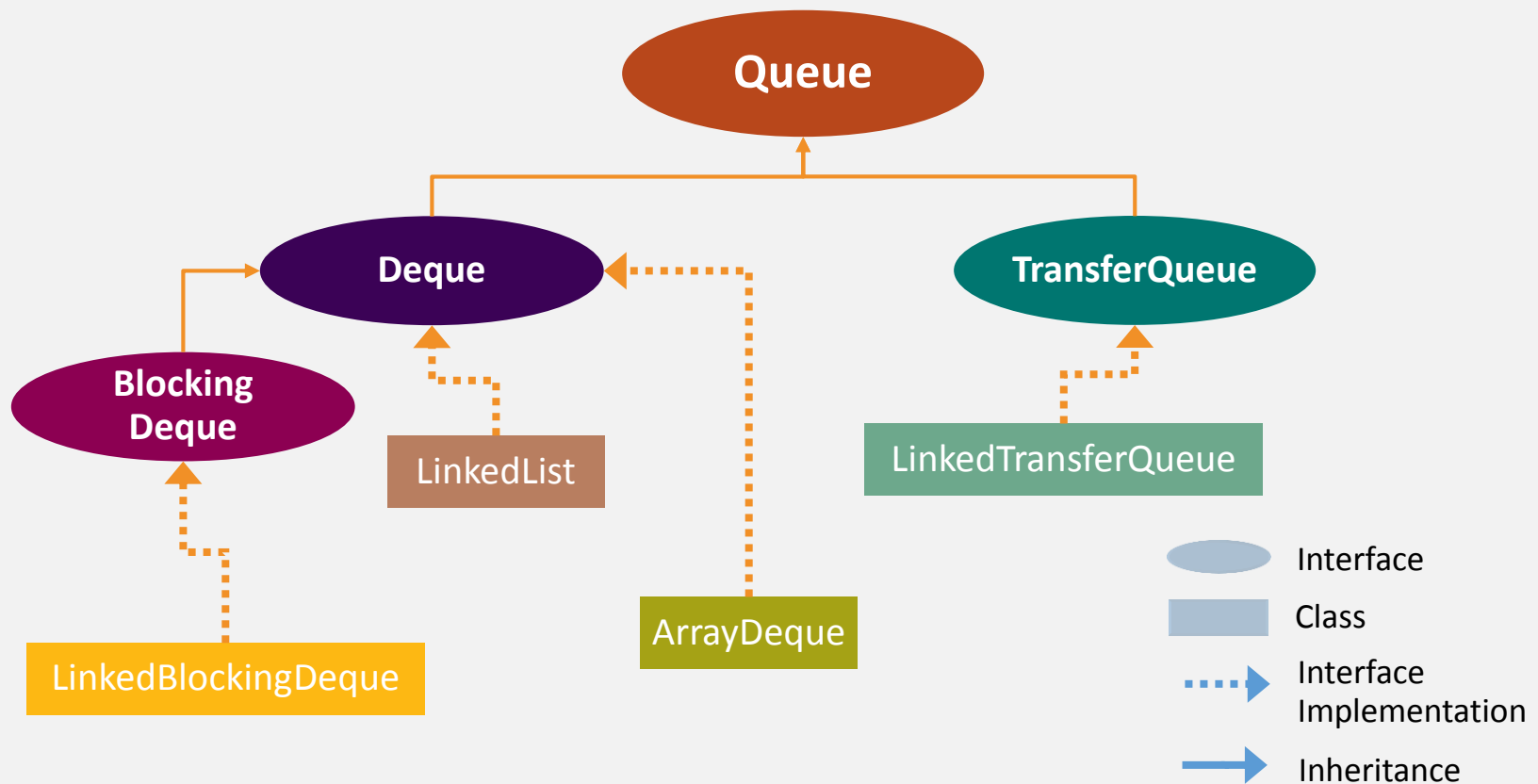


A **queue** is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue in a FIFO manner.

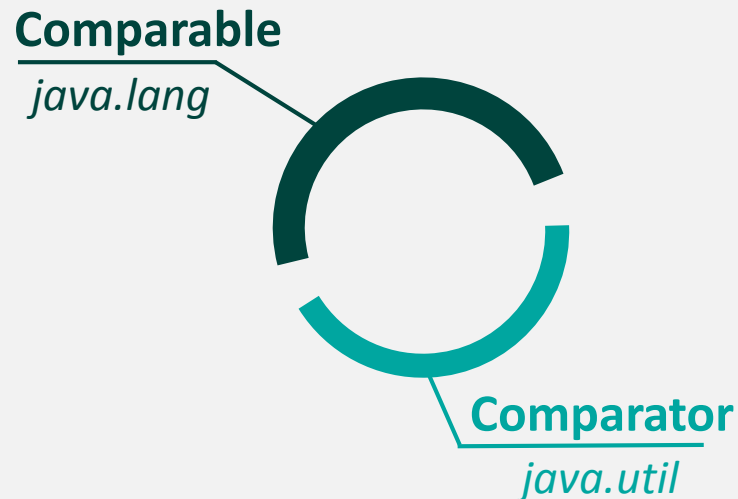
The Queue Interface extends Collection.

- Designed for holding elements prior to processing.
- Typically ordered in a first-in, first-out (FIFO) manner.
- Main Methods:
 - **remove()** - When empty, the remove() method throws an exception
 - **poll()** – When empty poll() returns null.
- Both the methods return the head of the queue.

Queue hierarchy shows the relation between the interfaces and their class.



To provide for ordering, or sorting of the objects in an application, the Java API provides two interfaces – **Comparable** and **Comparator**.



Implementing Comparable can be done either as an interface, or a logic to compare object, which can throw an exception.

```
public class Student implements Comparable {  
    public Student(String name, int score) {...}  
  
    public int compareTo(Object o)  
        throws ClassCastException {...}  
    public static void main(String args[]) {...}  
}
```

Same Constructor is used for both the methods to sort Students according to Score

```
public Student(String name, int score) {  
    this.name = name;  
    this.score = score;  
}
```

A c t i v i t y

Please study the below code. What will be the result of Iterator <Student>?

```
public static void main(String args[]) {  
    TreeSet<Student> set = new TreeSet<Student>();  
  
    set.add(new Student("Ann", 87));  
    set.add(new Student("Bob", 83));  
    set.add(new Student("Cat", 99));  
    set.add(new Student("Dan", 25));  
    set.add(new Student("Eve", 76));  
    Iterator<Student> iter = set.iterator();  
    while (iter.hasNext()) {  
        Student s = iter.next();  
        System.out.println(s.name + " " +  
s.score);  
    }  
}
```

In order to implement this interface, our parameter must also be an object.

```
public class Student implements Comparable
//This means it must implement the method
    public int compareTo(Object o)
//Notice that the parameter is an Object
    public int compareTo(Object o) throws
ClassCastException {
    if (o instanceof Student)
        return score - ((Student)o).score;
    else
        throw new ClassCastException("Not a Student!");
}
```

Since casting an arbitrary object to a Student may throw a `ClassCastException` for us, we don't need to throw it explicitly.

```
public int compareTo(Object o) throws ClassCastException {  
    return score - ((Student)o).score;  
}
```

The **ClassCastException** is a subclass of `RuntimeException`, we don't need to declare when we throw one.

```
public int compareTo(Object o) {  
    return score - ((Student)o).score;  
}
```

- Sorts only by the score
- Only one method of signature

Comparable

java.lang



Comparator

java.util

- Sorts more than one variable
- Requires definition of compare and equal methods

If objects are to be sorted in other ways, its not possible through **Comparable**.

Instead, Comparator can be used to provide sorting by more than one variable.

Comparator interface is used to order the objects of user-defined class. It provides multiple sorting sequence.

The two methods used by Comparator are:

- `int compare (T object1, T object2)`
 - Compares its two arguments for order
- `boolean equals(Object obj)`
 - Indicates whether some other object is "equal to" this comparator

```
import java.util.*;
public class StudentComparator
implements Comparator<Student> {
public int compare(Student s1, Student s2) {...}
public boolean equals(Object o1) {...}
}
```


- When using **Comparator** we don't need the compareTo method in the Student class. Making use of generics it can take Student arguments, instead of just Object arguments.

The **compare** method takes both objects that needs to be compared.


- This differs from `compareTo(Object o)` in these ways:

- The name is different
- It takes both objects as parameters, not just one
- We have to either use generics, or check the type for both objects
- If our parameters are Objects, they have to be cast to Students

- The main method remains almost same, but it includes the additional line of `Comparator`.



```
public int compare(Student  
s1, Student s2) {  
    return s1.score - s2.score;  
}
```



```
Comparator<Student> comp = new  
StudentComparator();  
  
TreeSet<Student> set = new  
TreeSet<Student>(comp);
```

A c t i v i t y

Write a program to implement adding a list of students, and sorting them according to their name and score, using compare method.

```
...  
    list.add(new Student("Ajay", 55));  
    list.add(new Student("Vijay", 90));  
    list.add(new Student("Sujoy", 60));  
    list.add(new Student("Ranajay", 99));  
  
    Collections.sort(list);  
    for(Student a: list)  
        System.out.print(a.getStudentName() + ", ");  
  
    Collections.sort(list, new Student());  
    for(Student a: list)  
        System.out.print(a.getStudentName() + " : "+  
                           a.getStudentscore() + ", ");  
...
```

The `comparator.equals` Method



The `comparator.equals` method is used to compare two comparators.

- It indicates whether some object is "equal to" this comparator.
- This method must obey the general contract of **`Object.equals(Object)`**.
- Even though it is part of the Comparator interface, you actually do not need to override it.
- The purpose is efficiency—you can replace one Comparator with an equal yet faster one.

Ignore this method!

Comparator 1
java.util



java.util
Comparator 2

When to use **Comparator** or **Comparable**



Comparable

- The interface is simpler and requires less work:
 - Your class implements Comparable
 - You provide a public `int compareTo(Object o)` method
 - Use no argument in your `TreeSet` or `TreeMap` constructor
 - You will use the same comparison method every time

Comparator

- The interface is more flexible but involves slightly more work:
 - Create as many different classes that implement `Comparator` as you like
 - You can sort the `TreeSet` or `TreeMap` differently with each
 - Also, construct `TreeSet` or `TreeMap` using the comparator you want; for example, sort `Students` by score or by name

Comparable

java.lang

Used to perform same comparison method each time

Used to compare more than one attribute

Comparator

java.util

```
Comparator<Student> myStudentNameComparator = new  
MyStudentNameComparator();  
  
TreeSet studentsByName = new  
TreeSet(myStudentNameComparator);  
  
studentsByName.addAll(studentsByScore);
```

- Suppose you have students already sorted by score using Comparable, in a TreeSet you call **studentsByScore**
- If you want to sort them again through comparator, use **studentsByName**

A c t i v i t y

How do you iterate to sort Students **by name** using Comparator, rather than by score? What changes are required here?

```
public static void main(String args[]) {  
    TreeSet<Student> set = new TreeSet<Student>();  
  
    set.add(new Student("Ann", 87));  
    set.add(new Student("Bob", 83));  
    set.add(new Student("Cat", 99));  
    set.add(new Student("Dan", 25));  
    set.add(new Student("Eve", 76));  
    Iterator<Student> iter = set.iterator();  
    while (iter.hasNext()) {  
        Student s = iter.next();  
        System.out.println(s.name + " " +  
s.score);  
    }  
}
```

02

What is the purpose of using the **comparator.equals** method?

A

Compare object and the comparators

B

Replace one comparator with another equal yet faster one

C

To equal the student score

D

To compare if sum of integer is same using the Streams



Questions?

02 Generics and Annotations

Generics allows "a type or method to operate on objects of various types while providing compile-time type safety."

Generics



- Ensures stored or retrieved type of objects
- Check done at compile-time to avoid nasty casting surprises, during runtime
- Generics are removed from compiled classes (backward compatibility), no additional code
- Can be used in legacy code to check for place where incorrect type is inserted
- Access to "template" class methods without casting

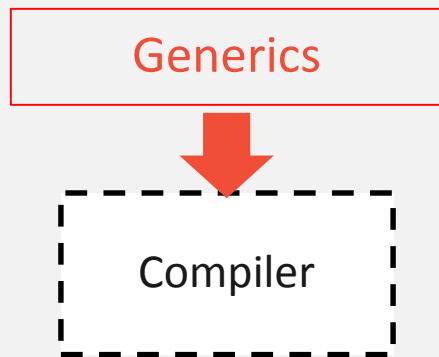
One big advantage of using Generics is that the collections are now type safe. For example, a Stack that holds only Strings having a **type-safe** collection.

Creating a Stack that holds only Strings

```
Stack<String> names = new Stack<String>();
```

You can write methods that require a type-safe collection as follows:

```
void printNames(Stack<String> names) {  
    String nextName = names.pop(); // no casting needed!  
    names.push("Hello"); // works just the same as before  
    names.push(Color.RED); // compile-time error!  
}
```



Generics are instructions passed only to the compiler. Generics also act as erasures during runtime.

In Java 1.4:

- `String s = myStack.pop();`
 - will not compile
- `String s = (String)myStack.pop();`
 - compiles with runtime check
- `myStack.push(Color.RED);`
 - compiles with no complaint

In Java 5

- `String s = myStack.pop();`
 - Compiles with no runtime check if `myStack` was declared as `Stack<String>`
 - Does not compile if `myStack` was declared any other way
- `myStack.push(Color.RED);`
 - is a compiler error (= syntax error)

A little more explanation ...

- Java 5 is upwardly compatible with Java 1.4
- Old java programs must continue to work
- Hence you can have non-generic stacks (and stack assignment)
- When you use a generic collection, you should make it generic everywhere, not just in the places that Java would otherwise report an error
- Eclipse will provide warnings for many unsafe cases, so pay close attention to those warnings

Creating a Stack that is type-safe

```
Stack<Integer> stack1 = new Stack<Integer>();  
Stack stack2 = stack1; // stack2 is alias of stack1  
stack2.push(Color.RED); // legal--stack2 is a plain stack  
Integer xxx = stack1.pop(); // ClassCastException!
```

Diamond operator (<>) is the type Inference for Generic Instance Creation.

This allows us to replace type arguments required to invoke the constructor of a generic class, with an empty set of type parameters (<>), as long as the compiler can infer the type arguments from the context.

Example:

```
Map<String, List<String>> myMap = new  
HashMap<String, List<String>>();
```



Java SE 7 allows substitution of the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>();
```

Annotations (metadata) are added using @ which can:

- Annotate types, methods, fields for documentation, code generation, runtime services
- Provide built-in & custom annotations
 - Built-in notations are @Override, @Deprecated, @SuppressWarnings, @SafeVarargs, @FunctionalInterface, @Retention, @Documented, @Target, @Inherited, @Repeatable
- Control availability of annotations
 - Source code, class file, runtime in JVM

Example:

```
/* @author Jack */  
public final class AnnotationsTest {  
    @Override  
    public String toString(int i) { return " x "; }  
}
```



03

Why are **generics** used?

A

Generics make code really fast.

B

Generics make code more optimized and readable.

C

Generics are instructions passed to the compiler.

D

Generics are not type safe.



Questions?

A c t i v i t y

Day 05 Practice Exercise:

1. Write a program that sorts the students by default of name. Create comparators to sort by age, date of birth and college.

Refer to the Additional Exercises handout / student guide for more details.

