



# Holding your Object

---

Container

Collection



# Array

---

- **Array initialization**
- `int[] a1;`
- `int a1[];`
- `int[] a1 = { 1, 2, 3, 4, 5 };`
- Well, it's possible to assign one array to another in Java, so you can say:
  - `int[] a2;`
- `a2 = a1;`



# Copy Array

---

- Well, it's possible to assign one array to another in Java, so you can say:
- `a2 = a1;`
- What you're really doing is copying a reference, as demonstrated here:



# Copy Array

---

- `// Arrays of primitives.`
- `public class Arrays {`
- `public static void main(String[] args) {`
- `int[] a1 = { 1, 2, 3, 4, 5 };`
- `int[] a2; a2 = a1;`
- `for(int i = 0; i < a2.length; i++)`
- `a2[i]++;`
- `for(int i = 0; i < a1.length; i++)`
- `System.out.println( "a1[" + i + "] = " + a1[i]);`
- `}`
- `} ///:~`



# Allocate Array size at run time

---

- `// Creating arrays with new.`
- `import java.util.*;`
- `public class ArrayNew {`
- `static Random rand = new Random();`
- `static int pRand(int mod) {`
- `return Math.abs(rand.nextInt()) % mod + 1;`
- `}`
- `public static void main(String[] args) {`
- `int[] a;`
- `}`



# Allocate Array size at run time

---

- `a = new int[pRand(20)];`
- `System.out.println(`
- `"length of a = " + a.length);`
- `for(int i = 0; i < a.length; i++)`
- `System.out.println(`
- `"a[" + i + "] = " + a[i]);`
- `}`
- `} ///:~`



# Initial Value of Array

---

- For numerics      zero
- and **char**      nothing
- and for **boolean false**
- **And for Object Null**



# Initial Value of Array

---

- `// Creating arrays with new.`
- `import java.util.*;`
- `public class InitialValue {`
- `public static void main(String[] args) {`
- `int[] intArray=new int[2];`
- `char[] charArray=new char[2];`
- `boolean[] booleanArray=new boolean[2];`
- `String[] stringArray=new String[2];`
- `System.out.println("intArray[0]="+intArray[0]);`
- `System.out.println("intArray[1]="+intArray[1]);`
- `System.out.println("charArray[0]="+charArray[0]);`
-



# Initial Value of Array

---

- `System.out.println("charArray[1]="+charArray[1]);`
- `System.out.println("booleanArray[0]="+booleanArray[0]);`
- `System.out.println("booleanArray[1]="+booleanArray[1]);`
- `System.out.println("stringArray[0]="+stringArray[0]);`
- `System.out.println("stringArray[1]="+stringArray[1]);`
- `}`
- `}`



# Initial Value of Array

---

- Output
- E:\COLLEC~1>java InitialValue
- intArray[0]=0
- intArray[1]=0
- charArray[0]=
- charArray[1]=
- booleanArray[0]=false
- booleanArray[1]=false
- stringArray[0]=null
- stringArray[1]=null



# Initialize arrays of objects

---

- `// Array initialization.`
- `public class ArrayInit {`
- `public static void main(String[] args) {`
- `Integer[] a = { new Integer(1), new Integer(2), new Integer(3), };`
- `Integer[] b = new Integer[] { new Integer(1), new Integer(2), new Integer(3),`  
`};`
- `}`
- `} ///:~`



# unknown element's types

---

- Since all classes are ultimately inherited from the common root class **Object** (a subject you will learn more about as this book progresses), you can create a method that takes an array of **Object** and call it like this:





# Unknown element's types

---

- At this point, there's not much you can do with these unknown objects, and this program uses the automatic **String** conversion to do something useful with each **Object**. RTTI, which covers *run-time type identification* (RTTI), you'll learn how to discover the exact type of such objects so that you can do something more interesting with them.



# Multidimensional arrays

---

- `import java.util.*;`
- `public class MultiDimArray {`
- `public static void main(String[] args) {`
- `int[][] a1 = {`
- `{ 1, 2, 3, },`
- `{ 4, 5, 6, },`
- `};`
- `for(int i = 0; i < a1.length; i++)`
- `for(int j = 0; j < a1[i].length; j++)`
- `System.out.println("a1[" + i + "][" + j +`
- `"] = " + a1[i][j]);`
- `System.out.println("a1.length="+a1.length);`
- `System.out.println("a[0].length="+a1[0].length);`
- `}`
- `}`



# Comparing arrays

---

- **Arrays** provides the overloaded method **equals()** to compare entire arrays for equality. Again, these are overloaded for all the primitives, and for **Object**. To be equal, the arrays must have the same number of elements and each element must be equivalent to each corresponding element in the other array, using the **equals()** for each element



# Initialization & re-assignment of arrays.

---

- See code `ArraySize`



# Summary

---

An array associates numerical indices to objects. It holds objects of a known type so that you don't have to cast the result when you're looking up an object. It can be multidimensional, and it can hold primitives. However, its size cannot be changed once you create it.



# Introduction to containers

---

- **container classes** are one of the most powerful tools for raw development because they significantly increase your programming muscle. The Java 2 containers represent a thorough redesign of the rather poor showings in Java 1.0 and 1.1. Some of the redesign makes things tighter and more sensible.



# Introduction to containers

---

- The Java 2 container library takes the issue of “holding your objects” and divides it into two distinct concepts:
  1. **Collection:** a group of individual elements, often with some rule applied to them. A **List** must hold the elements in a particular sequence, and a **Set** cannot have any duplicate elements. (A *bag*, which is not implemented in the Java container library since **Lists** provide you with enough of that functionality, has no such rules.)



# Introduction to containers

---

1. **Map**: a group of key-value object pairs. At first glance, this might seem like it ought to be a **Collection** of pairs, but when you try to implement it that way the design gets awkward, so it's clearer to make it a separate concept. On the other hand, it's convenient to look at portions of a **Map** by creating a **Collection** to represent that portion. Thus, a **Map** can return a **Set** of its keys, a **Collection** of its values, or a **Set** of its pairs. **Maps**, like arrays, can easily be expanded to multiple dimensions without adding new concepts: you simply make a **Map** whose values are **Maps** (and the values of *those Maps* can be **Maps**, etc.).



# Printing containers

- 
- `// Containers print themselves automatically.`
  - `import java.util.*;`
  - `public class PrintingContainers {`
  - `static Collection fill(Collection c) {`
  - `c.add("dog");`
  - `c.add("dog");`
  - `c.add("cat");`
  - `return c;`
  - `}`
  - `static Map mfill(Map m) {`
  - `m.put("dog", "Bosco");`
  - `m.put("dog", "Spot");`
  - `m.put("cat", "Rags");`
  - `return m;`
  - `}`
  -



# Printing containers

---

- `public static void main(String[] args) {`
- `System.out.println(fill(new ArrayList()));`
- `System.out.println(fill(new HashSet()));`
- `System.out.println(mfill(new HashMap()));`
- `}`
- `}` ///:~



## Printing containers

---

- The **Collection** category only holds one item in each location
- The **Collection** category only holds one item in each location
- The **fill( )** and **mfill( )** methods fill **Collections** and **Maps**, respectively. If you look at the output, you can see that the default printing behavior (provided via the container's various **toString( )** methods) produces quite readable results, so no additional printing support is necessary as it was with arrays:
- `[dog, dog, cat] [cat, dog] {cat=Rags, dog=Spot}`



# Filling containers

---

- `// The Collections.fill() method.`
- `import java.util.*;`
- `public class FillingLists {`
- `public static void main(String[] args) {`
- `List list = new ArrayList();`
- `for(int i = 0; i < 10; i++)`
- `list.add("");`
- `Collections.fill(list, "Hello");`
- `System.out.println(list);`
- `}`
- `} ///:~`



## Container disadvantage: unknown type

---

- The “disadvantage” to using the Java containers is that you lose type information when you put an object into a container. This happens because the programmer of that container class had no idea what specific type you wanted to put in the container, and making the container hold only your type would prevent it from being a general-purpose tool. So instead, the container holds references to **Object**, which is the root of all the classes so it holds any type. (Of course, this doesn’t include primitive types, since they aren’t inherited from anything.) This is a great solution, except:



## Container disadvantage: unknown type

---

1. Since the type information is thrown away when you put an object reference into a container, there's no restriction on the type of object that can be put into your container, even if you mean it to hold only, say, cats. Someone could just as easily put a dog into the container.
2. Since the type information is lost, the only thing the container knows that it holds is a reference to an object. You must perform a cast to the correct type before you use it.



## Container disadvantage: unknown type

---

- On the up side, Java won't let you *misuse* the objects that you put into a container. If you throw a dog into a container of cats and then try to treat everything in the container as a cat, you'll get a run-time exception when you pull the dog reference out of the cat container and try to cast it to a cat.



## Container disadvantage: unknown type

---

- Here's an example using the basic workhorse container, **ArrayList**. For starters, you can think of **ArrayList** as “an array that automatically expands itself.” Using an **ArrayList** is straightforward: create one, put objects in using **add( )**, and later get them out with **get( )** using an index, just like you would with an array but without the square brackets. **ArrayList** also has a method **size( )** to let you know how many elements have been added so you don't inadvertently run off the end and cause an exception.



# Container disadvantage: unknown type

---

- `//: c09:Cat.java`
- `public class Cat {`
- `private int catNumber;`
- `Cat(int i) { catNumber = i; }`
- `void print() {`
- `System.out.println("Cat #" + catNumber);`
- `}`
- `}` `///:~`



## Container disadvantage: unknown type

---

- `//: c09:Dog.java`
- `public class Dog {`
- `private int dogNumber;`
- `Dog(int i) { dogNumber = i; }`
- `void print() {`
- `System.out.println("Dog #" + dogNumber);`
- `}`
- `} ///:~`



# Container disadvantage: unknown type

---

- **Cats** and **Dogs** are placed into the container, then pulled out:

```
//: c09:CatsAndDogs.java
```

- // Simple container example.

- import java.util.\*;

- public class CatsAndDogs {

- public static void main(String[] args) {

- ArrayList cats = new ArrayList();

- for(int i = 0; i < 7; i++)

- cats.add(new Cat(i));

- // Not a problem to add a dog to cats:

- cats.add(new Dog(7));

- for(int i = 0; i < cats.size(); i++)

- ((Cat)cats.get(i)).print();

- // Dog is detected only at run-time

- }

- } ///:~



# Iterators

---

- The concept of an *iterator* can be used to achieve this abstraction. An iterator is an object whose job is to move through a sequence of objects and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence. In addition, an iterator is usually what's called a “light-weight” object: one that's cheap to create. For that reason, you'll often find seemingly strange constraints for iterators; for example, some iterators can move in only one direction.



# Iterators

---

- The Java **Iterator** is an example of an iterator with these kinds of constraints. There's not much you can do with one except:
  1. Ask a container to hand you an **Iterator** using a method called **iterator( )**. This **Iterator** will be ready to return the first element in the sequence on your first call to its **next( )** method.
  2. Get the next object in the sequence with **next( )**.
  3. See if there *are* any more objects in the sequence with **hasNext( )**.
  4. Remove the last element returned by the iterator with **remove( )**.



# Iterators

- `// Simple container with Iterator.`
- `import java.util.*;`
- `public class CatsAndDogs2 {`
- `public static void main(String[] args) {`
- `ArrayList cats = new ArrayList();`
- `for(int i = 0; i < 7; i++)`
- `cats.add(new Cat(i));`
- `cats.add(new Dog(7));`
- `Iterator e = cats.iterator();`
- `while(e.hasNext())`
- `((Cat)e.next()).print();`
- `}`
- `} ///:~`



# Iterators

---

- As another example, consider the creation of a general-purpose printing method:
- // Using an Iterator.
- `import java.util.*;`
  
- `class Hamster {`
- `private int hamsterNumber;`
- `Hamster(int i) { hamsterNumber = i; }`
- `public String toString() {`
- `return "This is Hamster #" + hamsterNumber;`
- `}`
- `}`



# Iterators

---

- `class Printer {`
- `static void printAll(Iterator e) {`
- `while(e.hasNext())`
- `System.out.println(e.next());`
- `}`
- `}`



# Iterators

---

- `public class HamsterMaze {`
- `public static void main(String[] args) {`
- `ArrayList v = new ArrayList();`
- `for(int i = 0; i < 3; i++)`
- `v.add(new Hamster(i));`
- `Printer.printAll(v.iterator());`
- `}`
- `} ///:~`



# Iterators

---

- You must be careful when using the remove method. Calling remove removes the element that was returned by the last call to next. That makes sense if you want to remove a particular value- you need to see the element before you can decide that it is the one that should be removed. But if you want to remove an element by position, you first need to skip the element. For example, here is how you remove the first element in a collection.
- `Iterator it =c.iterator()`
- `it.next(); //skip over the first element`
- `it.remove(); //now remove it`



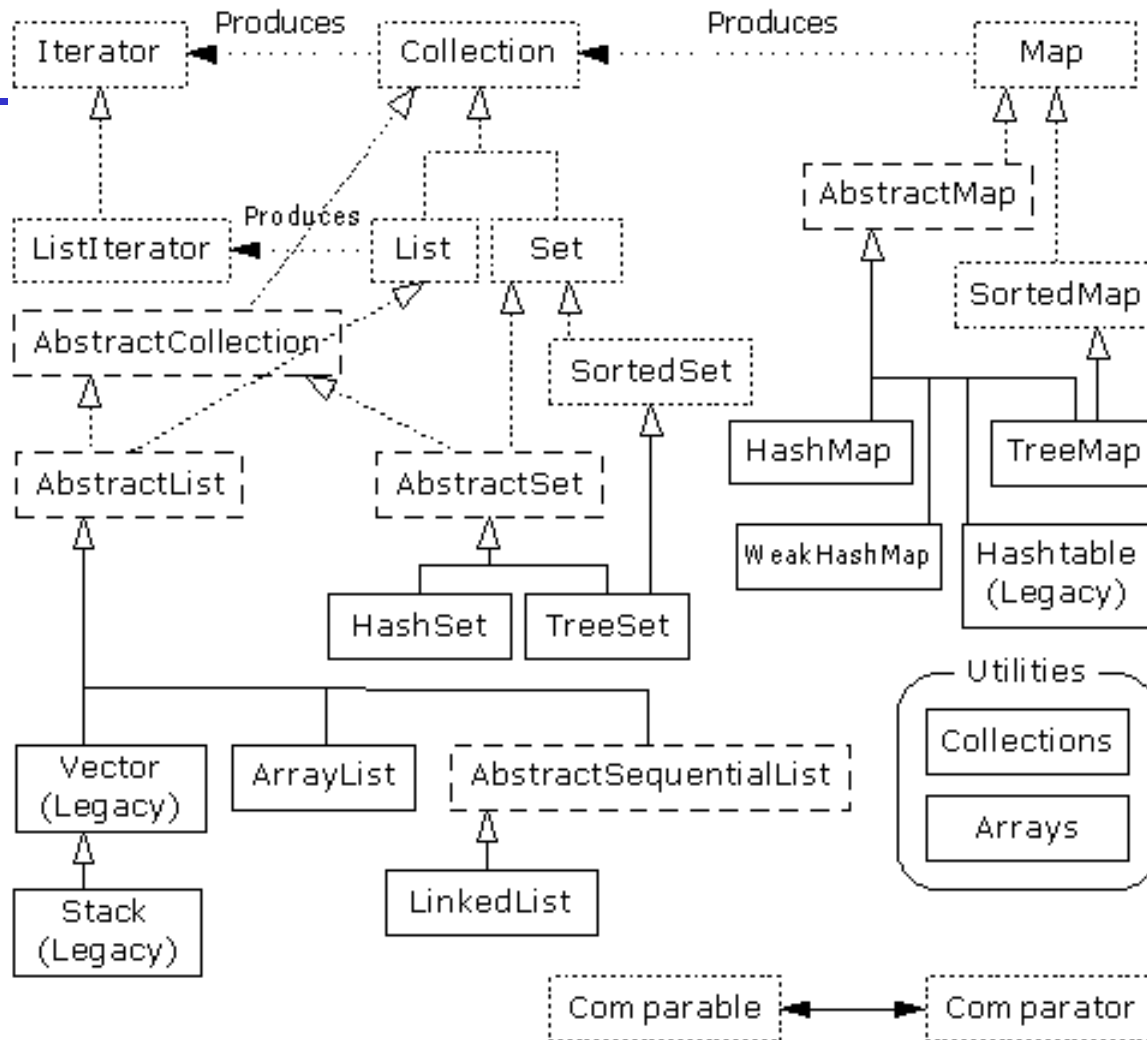
# Iterators

---

- If you want to remove two adjacent element, you cannot simply call:
  - `it.remove();`
  - `it.remove(); //error`
- You should call:
  - `it.remove();`
  - `it.next();`
  - `it.remove();`



# Container taxonomy





# Container taxonomy

---

- This diagram can be a bit overwhelming at first, but you'll see that there are really only three container components: **Map**, **List**, and **Set**, and only two or three implementations of each one (with, typically, a preferred version). When you see this, the containers are not so daunting.
- The dotted boxes represent **interfaces**, the dashed boxes represent **abstract** classes, and the solid boxes are regular (concrete) classes. The dotted-line arrows indicate that a particular class is implementing an **interface** (or in the case of an **abstract** class, partially implementing that **interface**). The solid arrows show that a class can produce objects of the class the arrow is pointing to. For example, any **Collection** can produce an **Iterator**, while a **List** can produce a **ListIterator** (as well as an ordinary **Iterator**, since **List** is inherited from **Collection**).



# Fills a Collection

---

- Here's a simple example, which fills a **Collection** (represented here with an **ArrayList**) with **String** objects and then prints each element in the **Collection**:  
// A simple example using Java 2 Collections.
- `import java.util.*;`
- `public class SimpleCollection {`
- `public static void main(String[] args) {`
- `// Upcast because we just want to`
- `// work with Collection features`
- `Collection c = new ArrayList();`
- `for(int i = 0; i < 10; i++)`
- `}`
- `}`



# Fills a Collection

---

- `c.add(Integer.toString(i));`
- `Iterator it = c.iterator();`
- `while(it.hasNext())`
- `System.out.println(it.next());`
- `}`
- `} ///:~`



# Collection functionality

- 
- **boolean add(Object)**
    - Ensures that the container holds the argument. Returns false if it doesn't add the argument. (This is an “optional” method, described later in the chapter.)
  - **boolean addAll(Collection)**
    - Adds all the elements in the argument. Returns true if any elements were added. (“Optional.”)
  - **void clear()**
    - Removes all the elements in the container. (“Optional.”)
  - **boolean contains(Object)**



# Collection functionality

---

- **true** if the container holds the argument.
- **boolean containsAll(Collection)**
- **true** if the container holds all the elements in the argument.
- **boolean isEmpty( )**
- **true** if the container has no elements.
- **Iterator iterator( )**
- Returns an **Iterator** that you can use to move through the elements in the container.



# Collection functionality

---

- **boolean remove(Object)**
  - If the argument is in the container, one instance of that element is removed. Returns **true** if a removal occurred. (“Optional.”)
- **boolean removeAll(Collection)**
  - Removes all the elements that are contained in the argument. Returns **true** if any removals occurred. (“Optional.”)
- **boolean retainAll(Collection)**
  - Retains only elements that are contained in the argument (an “intersection” from set theory). Returns **true** if any changes occurred. (“Optional.”)



# Collection functionality

---

- **int size( )**
  - Returns the number of elements in the container.
- **Object[] toArray( )**
  - Returns an array containing all the elements in the container.
- **Object[]  
toArray(Object[] a)**
  - Returns an array containing all the elements in the container, whose type is that of the array **a** rather than plain **Object** (you must cast the array to the right type).



## Collection functionality

---

- Collection interface declares quite a few useful methods that all implementing classes must supply.
- It is a bother if every class that implements the Collection interface has to supply so many routine methods. To make life easier for implementors, the class AbstractCollection leaves the fundamental methods (such as add and iterator) abstract but implements the routine methods in terms of them.



## Collection functionality

---

- A concrete collection class can now extend the `AbstractCollection` class. It is now up to the concrete collection class to supply an `add` method.
- This is a good design for a class framework. The users of the collection classes have a richer set of methods available in the generic interface, but the implementors of the actual data structures do not have the burden of implementing all the routine methods.



# List and LinkedList

---

- public interface List extends Collection
  - void add(int index, Object element)
  - boolean add(Object o)
  - boolean addAll(Collection c)
  - boolean addAll(int index, Collection c)
  - void clear()
  - boolean contains(Object o)
  - boolean containsAll(Collection c)
  - Object remove(int i)
  - Object set(int i, Object element)
  - and more



# List and LinkedList

---

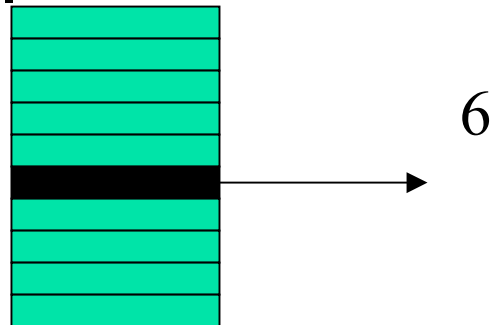
- public class LinkedList extends AbstractSequentialList implements List, Cloneable, Serializable
- LinkedList()
  - Constructs an empty list. LinkedList(Collection c)
  - Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator
  - Void addFirst(Object element).
  - Void addLast(Object element).
  - Object getFirst()
  - Object getLast()
  - Object removeFirst()
  - Object removeLast()



## Concrete Collection: Linked lists

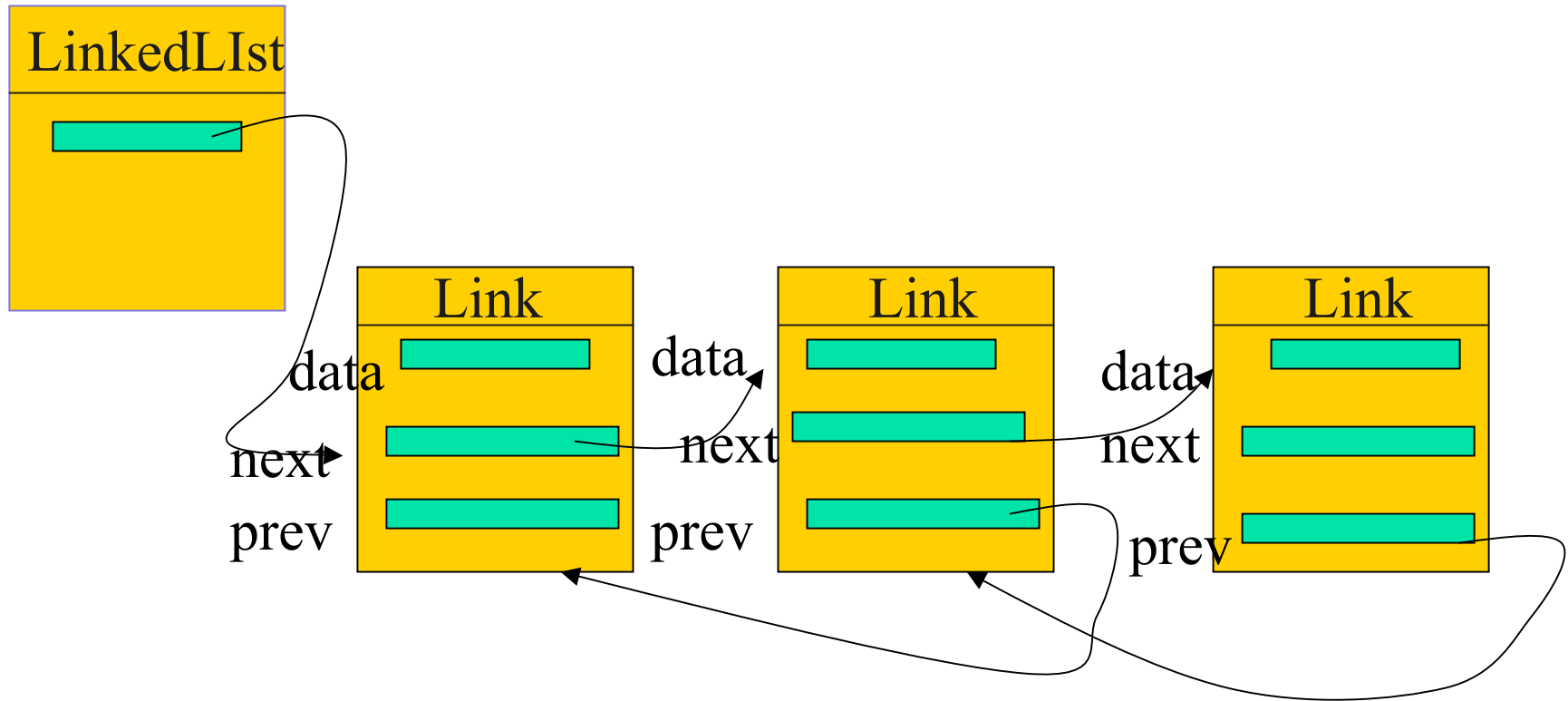
---

- Linked Lists:
- Arrays/Vector suffer from a major drawback, Removing an element from the middle of an array is very expensive since all array elements beyond the removed one must be moved toward the beginning of the array. The same is true for inserting elements in the middle.



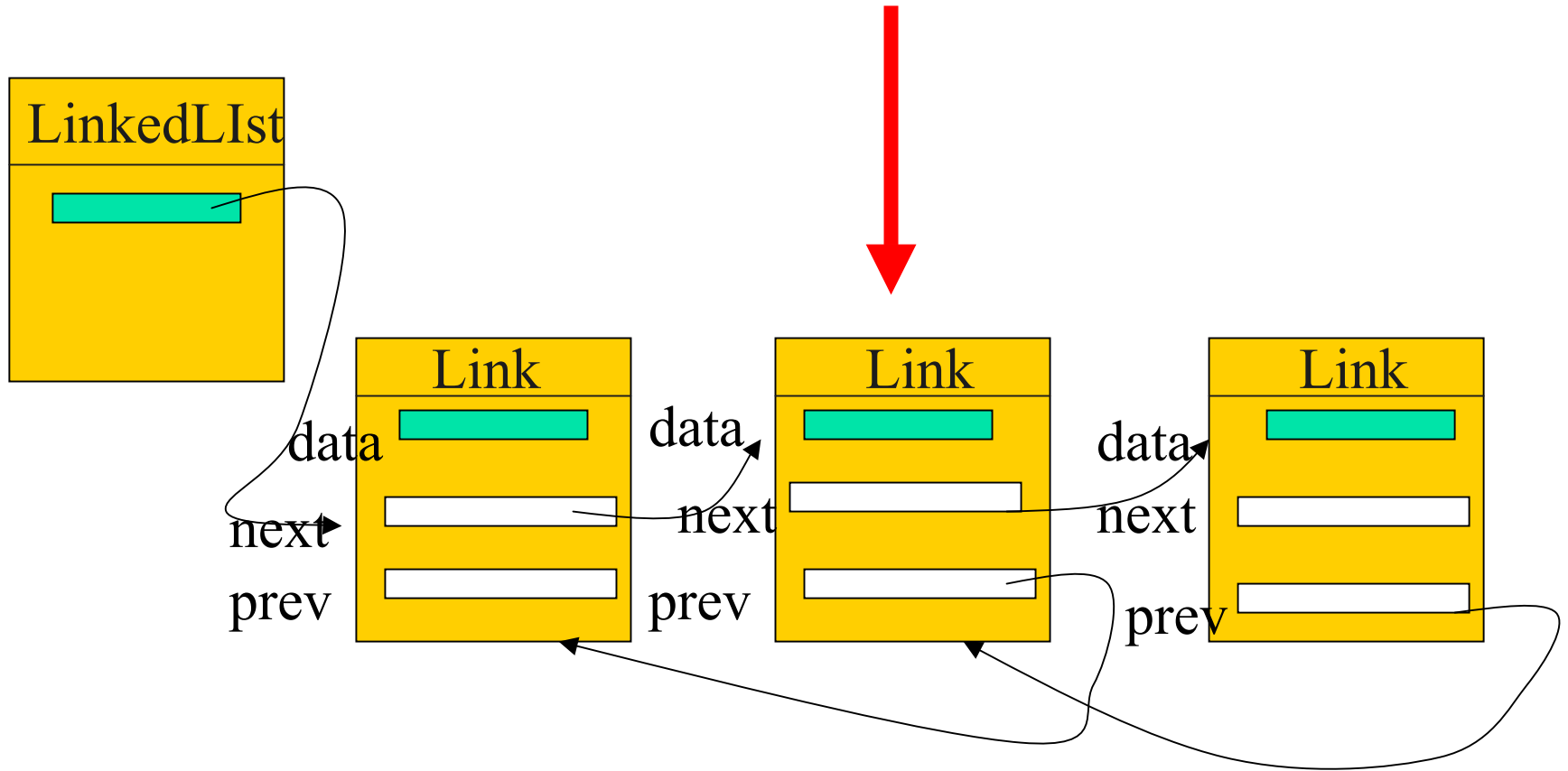


## Concrete Collection: Linked lists



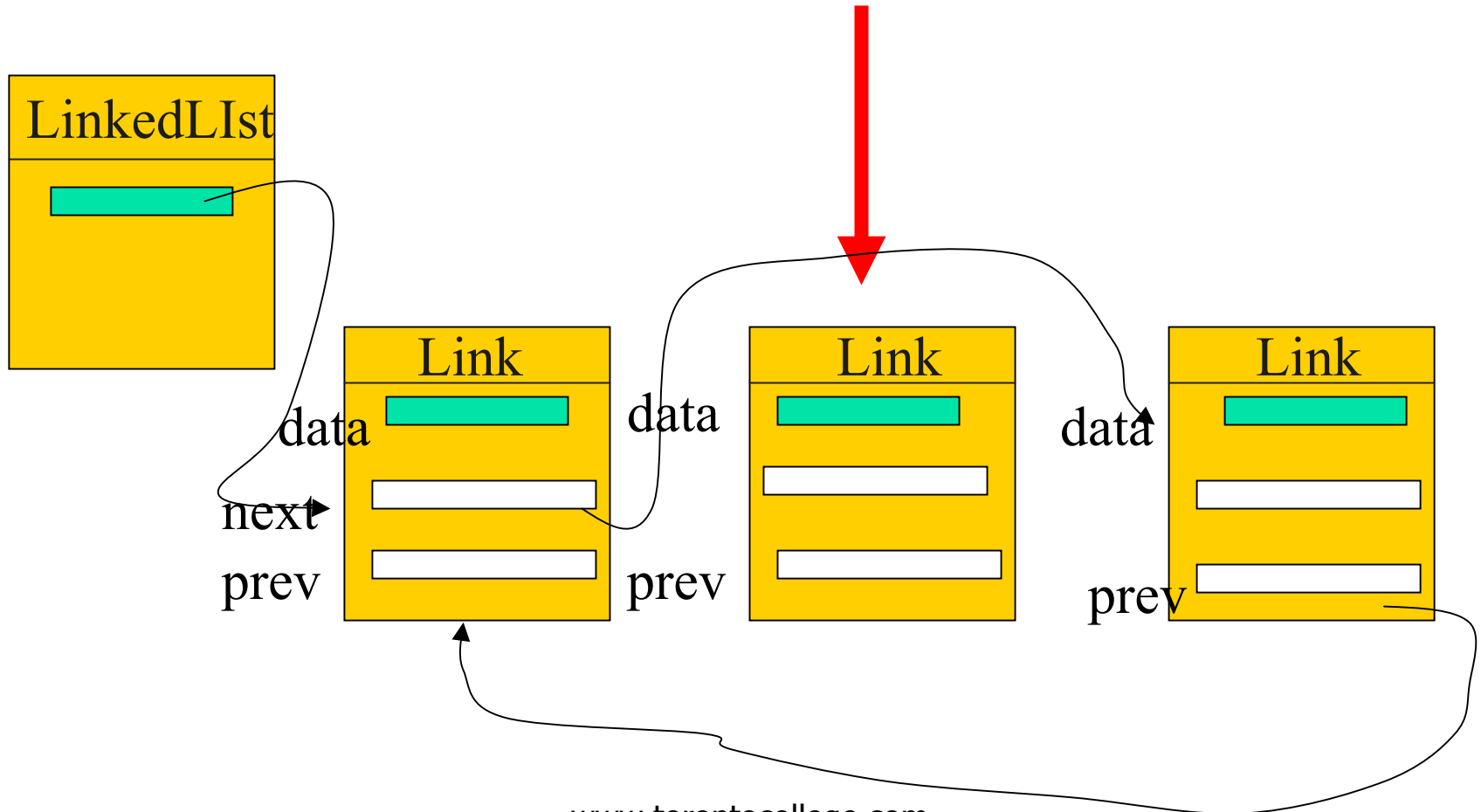


# Concrete Collection: Linked lists





# Concrete Collection: Linked lists





## Concrete Collection: Linked lists

---

- `LinkedList staff=new LinkedList();`
- `Staff.add("Angela");`
- `Staff.add("bob");`
- `Staff.add("carl");`
- `Iterator iter=staff.iterator();`
- `For (int I=0;I<3;I++)`
- `System.out.println("iter.next);`
- `iter.remove(); //remove last visited element`



## Concrete Collection: Linked lists

---

- There is an important difference between linked lists and generic collections. A linked list is an ordered collection where the position of the object matters. The `LinkedList.add` method adds the object to the end of the list.



# ListIterator and Iterator

---

## ■ Iterator

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



# ListIterator and Iterator

---

- ListIterator

**interface ListIterator extends Iterator**

- boolean hasNext()  
Returns true if this list iterator has more elements when traversing the list in the reverse direction.
- void add(Object o)  
Inserts the specified element into the list (optional operation).
- int nextIndex()  
Returns the index of the element that would be returned by a subsequent call to next



# ListIterator and Iterator

---

- `int previousIndex()`  
Returns the index of the element that would be returned by a subsequent call to `previous`.
- `void set(Object o)`  
Replaces the last element returned by `next` or `previous` with the specified element (optional operation).



# ListIterator and Iterator

---

- The add method adds the new element before the iterator position
  - `LinkedList staff=new LinkList();`
  - `ListIaterator iter=staff.listIterator();`
  - `Iter.next();`
  - `Iter.add("Juliet");`
- The set method replaces the last element returned by call to next or previous with a new element
  - `ListIterator iter=list.listIterator();`
  - `Object oldValue=iter.next(); //returns first element`
  - `Iter.set(newValue); //sets first element to newValue`



# Example: LinkedListTest

---

```
■ import java.util.*;

■ public class LinkedListTest
■ {   public static void main(String[] args)
■     {   List a = new LinkedList();
■         a.add("Angela");
■         a.add("Carl");
■         a.add("Erica");

■         List b = new LinkedList();
■         b.add("Bob");
■         b.add("Doug");
■         b.add("Frances");
■         b.add("Gloria");
```



## Example: LinkedListTest

---

- `// merge the words from b into a`
- `ListIterator aIter = a.listIterator();`
- `Iterator bIter = b.iterator();`
- `while (bIter.hasNext())`
- `{ if (aIter.hasNext()) aIter.next();`
- `aIter.add(bIter.next());`
- `}`
- `System.out.println(a);`



# Example: LinkedListTest

---

- `// remove every second word from b`
  
- `bIter = b.iterator();`
- `while (bIter.hasNext())`
- `{ bIter.next(); // skip one element`
- `if (bIter.hasNext())`
- `{ bIter.next(); // skip next element`
- `bIter.remove(); // remove that element`
- `}`
- `}`
  
- `System.out.println(b);`



## Example: LinkedListTest

---

- `// bulk operation: remove all words in b from a`
- `a.removeAll(b);`
- `System.out.println(a);`
- `}`
- `}`



# ArrayLists

---

- ArrayList implements the List interface. An ArrayList is similar to a vector: it encapsulates a dynamically reallocated Object[] array.
- Why use an ArrayList instead of a Vector? All methods of the Vector class are synchronized. It is safe to access a Vector object from two threads. But if you only access a vector from a single thread-by-fat the more common case  
--your code wastes quite a bit of time with synchronization. In contrast, the ArrayList methods are not synchronized



# Hash Functions

---

- Linked lists and arrays let you specify in which order you want to arrange the elements. However, if you are looking for a particular element and you do not remember its position, then you need to visit all elements until you find a match.
- A well-known data structure for finding objects quickly is the hash table. A hash table computes an integer, called the hash code, for each object.



# Hash Functions

---

- **hash function**

- Function which, when applied to the key, produces a integer which can be used as an address in a hash table.

- **hash table**

- Tables which can be searched for an item in  **$O(1)$**  time using a hash function to form an address from the key.



# Hash Functions

---

- **hashCode**
- A Java method which defined in the Object class. Every object has a default hashcode.
- A hash code is derived from the object's memory address. In general, the default hash function is not very useful because objects with identical contents may yield different hash codes.



# Hash Functions: hashCode

---

- `public class hashCodeTest{`
- `public static void main(String[] args){`
- `String s="ok";`
- `StringBuffer sb=new StringBuffer(s);`
- `System.out.println("s.hashCode()="+s.hashCode());`
- `System.out.println("sb.hashCode()="+sb.hashCode());`
- `String t="ok";`
- `StringBuffer st=new StringBuffer(t);`
- `System.out.println("t.hashCode()="+t.hashCode());`
- `System.out.println("st.hashCode()="+st.hashCode());`
- `}`
- `}`



# Hash Functions: hashCode

---

- `s.hashCode()`=3548
- `sb.hashCode()`=7474923
- `t.hashCode()`=3548
- `st.hashCode()`=3242435



# Hash Functions: hashCode

---

- Note that the strings `s` and `t` have the same hash value because, for strings, the hash values are derived from their contents. The string buffers `sb` and `tb` have different hash values because a special hash function has been defined for the `StringBuffer` class and the default hash code function in the object class derives the hash code from the object's memory address.



# Hash Functions: hashCode

---

- You should always define the hashCode method for objects that you insert into a hash table. This method should return an integer.
- The hash table code will later reduce the integer by dividing by the bucket count and taking the remainder.



# Hash Functions: hashCode

---

- For example, suppose you have the class Item for inventory items, An item consists of a description string and a part number.

```
anItem=new Item("Toaster", 49954);
```

If you want to construct a hash set of items, you need to define a hash code.

```
class Item{  
    private String description;  
    private String partNumber;  
    public int hashCode(){  
        return 13*description.hashCode()+17*partnumber;  
    }  
}
```

.....



# Object equals

---

- The object class defines equals method, but this method only tests whether or not two objects are identical. If you do not redefine equals, then every new object that you insert into table will be considered a different object.



# Object equals

---

- You need to redefine equals to check for equal contends
- class Item {
- public boolean equals(Object other){
- if (other !=null && getClass()==other.getClass()){
- Item otherItem=(Item)other;
- Return description.equals(otherItem.description) && partNumber==otherItem.partNumber;
- }
- Else
- return false;
- ..
- }



# Object equals

---

- `public class EqualsTest{`
- `public static void main(String[] args){`
- `String s="ok";`
- `StringBuffer sb=new StringBuffer(s);`
- 
- `String t="ok";`
- `System.out.println("-----");`
- `System.out.println("s.equals(t) "+s.equals(t));`
-



# Object equals

---

- `StringBuffer st=new StringBuffer(t);`
- `System.out.println("-----");`
- `System.out.println("sb.equals(st) "+sb.equals(st));`
  
- `System.out.println("-----");`
- `System.out.println("s==t "+(s==t));`
- `System.out.println("-----");`
- `System.out.println("sb==st "+(sb==st));`



# Object equals

---

- `String newS=new String("ok");`
- `String newT=new String("ok");`
- `System.out.println("-----");`
- `System.out.println("newS==newT "+(newS==newT));`
- `System.out.println("-----");`
- `System.out.println("newS.equals(newT)  
"+newS.equals(newT));`
- `newS=newT;`
- `System.out.println("-----");`
- `System.out.println("newS==newT "+(newS==newT));`
- `System.out.println("-----");`
- `System.out.println("newS.equals(newT)  
"+newS.equals(newT));`
- `}`
- `}`



# Object equals

---

- **System.out.println("s.equals(t) "+s.equals(t));**
- **Here s and t are String. String equals method tests the content equal or not.**
- **System.out.println("sb.equals(st) "+sb.equals(st));**
- Here sb and st are StringBuffer object, the equals method derived from Object class test the two object are some object or not(same memory location).
- == It only determines whether or not the strings are stored in the same location.



# Hash Set

---

- `public class HashSet extends AbstractSet`
- This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.



# Hash Set

---

- This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.



# Hash Set

---

- A set is a collection of elements without duplicates. The `add` method of a set tries to find the object to be added and only adds it if it is not yet present
- `HashSet()` Constructs a new, empty set; the backing `HashMap` instance has default capacity and load factor, which is 0.75.
- `HashSet(Collection c)` Constructs a new set containing the elements in the specified collection.
- `HashSet(int initialCapacity)` Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and default load factor, which is 0.75



# Hash Set

---

- `HashSet(int initialCapacity, float loadFactor)` Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and the specified load factor.



# Hash Set: example

---

- SetTest.java



# Tree Sets

---

- The TreeSet class is similar to the hash set, with one added improvement. A tree set is a sorted collection. You insert elements into the collection in any order. When you iterate through the collection, the values are automatically presented in sorted order.
- Adding an element to a tree is slower than adding it to a hash table, but it is still much faster than adding it into right place in an array or linked list.
- If the tree contains  $n$  elements, then an average of  $\log_2 n$  comparisons are required to find the correct position for the new element. For example, if the tree already contains 1,000 elements, then adding a new element requires about 10 comparisons.



# Tree Sets: Object comparison

---

- How does the TreeSet know how you want the elements sorted? By default, the tree set assumes that you insert elements that implement the Comparable interface. That interface defines a single method:

```
int compareTo(Object other)
```

- The call `a.compareTo(b)` must return 0 if a and b are equal, a negative integer if a comes before b in the sort order, and a positive integer if a comes after b.
-



# Tree Sets: Object comparison

---

- If you insert your own objects, you need to define a sort order yourself by implementing the Comparable interface. There is no default implementation of compareTo in the Object class
- Class Item implements Comparable {
- Public int compareTo(Object other){
- Item otherItem=(Item)other;
- return partnumber-otherItem.partNumber;
- }
- ..
- }



# Tree Sets: Object comparison

---

- Using the Comparable interface for defining the sort order has obvious limitations. You can only implements the interface once. But what can you do if you need to sort a bunch of items by part number in one collection and by description in another?
- In these situations, you can define a class that implements the Comparator interface
- To sort items by their description, simply define a class that implements the Comparator interface



# Tree Sets: Object comparison

---

- To sort items by their description, simply define a class that implements the Comparator interface
- Class ItemComparator implements Comparator{
- public int compare(Object a, Object b) {
- Item itemA=(Item)a;
- Item itemB=(Item)b;
- String descrA=itemA.getDescription();
- String descrB=itemB.getDescription();
- Return descrA.compareTo(descrB);
- }
- }



# Tree Sets: Object comparison

---

- You then pass an object of this class to the tree set constructor:

```
ItemComparator comp=new ItemComparator();  
TreeSet sortByDescription=new TreeSet(comp);
```



# Tree Sets: Object comparison

---

- Note that this item comparator has no data. It is just a holder for the comparison method. Such an object sometimes called a function object.
- Function objects are commonly defined “on the fly” as instances of anonymous inner classes



# Tree Sets: Object comparison

---

- `TreeSet sortByDescription=new TreeSet(`
- `new Comparator{`
- `public int compare(Object a, Object b) {`
- `Item itemA=(Item)a;`
- `Item itemB=(Item)b;`
- `String descrA=itemA.getDescription();`
- `String descrB=itemB.getDescription();`
- `return descrA.compareTo(descrB);`
- `}`
- `}`



# Tree Sets: Example

---

- TreeSetTest.java



# Maps

---

- A map stores key/value pairs. You can find a value if you provide the key. Key must be unique. You cannot store two values with the same key. If you call the put method twice with the same key, then the second value replace the first one. The remove method removes an element from the map. The size() method returns the number of entries in the map.



# Maps

---

- Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.



# Maps

---

- This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.



# Maps

---

- An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method.



# Maps

---

- As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs.
- Note that this implementation is not synchronized. If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the `Collections.synchronizedMap` method. This is best done at creation time, to prevent accidental unsynchronized access to the map: `Map m = Collections.synchronizedMap(new HashMap(...));`



# Maps

---

- `HashMap()` Constructs a new, empty map with a default capacity and load factor, which is 0.75.
- `HashMap(int initialCapacity)` Constructs a new, empty map with the specified initial capacity and default load factor, which is 0.75.
- `HashMap(int initialCapacity, float loadFactor)` Constructs a new, empty map with the specified initial capacity and the specified load factor.
- `HashMap(Map t)` Constructs a new map with the same mappings as the given map.



# Maps: View of map

---

- View of map: objects that implement the Collection interface or one of its subinterfaces.
- There are three views:
  - Set of keys: `set keySet()`
  - Collection of values: Collection values
  - The set of key/values: `Set entrySet()`



# Maps: View of map

---

- For example, you can enumerate all keys of a map:
  - Set keys=map.keySet();
  - Iterator iter=keys.iterator();
  - While (iter.hasNext()) {
  - {Object key=iter.next();
  - Do something with key
  - }



# Maps: View of map

---

- MapTest.java



# Maps:WeakHashMap

---

- public class WeakHashMap extends Abstract Map implements Map
- A hashtable-based Map implementation with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently than other Map implementations.



# Maps: TreeMap

---

- public class TreeMap extends AbstractMap implements SortedMap, Cloneable, Serializable
- Red-Black tree based implementation of the SortedMap interface. This class guarantees that the map will be in ascending key order, sorted according to the natural order for the key's class (see Comparable), or by the comparator provided at creation time, depending on which constructor is used. This implementation provides guaranteed  $\log(n)$  time cost for the containsKey, get, put and remove operations.



# Views and Wrappers

---

- You might think it overkill to have six interfaces and five abstract classes to implement six concrete collection classes. However, these figures do not tell the whole story. By using views, you can obtain other objects that implement the Collection and Map interface
- For example, you can enumerate all keys of a map:
  - `Set keys=map.keySet();`
  - `Iterator iter=keys.iterator();`
  - `While (iter.hasNext()) {`
  - `{Object key=iter.next();`
  - `Do something with key`
  - `}`
- .



# Views and Wrappers

---

- Recall that the methods of the Vector class are synchronized, which is unnecessarily slow if the vector is only accessed from a single thread. For that reason, we recommended the use of the ArrayList instead of the vector class. However, if you do access a collection from multiple threads, it is very important that the methods are synchronized.
- Java supplied a mechanism that produces synchronized views for all interfaces. For example, the static synchronizedMap method in the Collections class can turn any Map into a Map with synchronized access methods



# Views and Wrappers

---

- For example:
  - `HashMap hashMap=new HashMap();`
  - `Map=Collections.synchronizedMap(hashMap);`  
Now, you can access the map object from multiple threads.
- There are six methods to obtain synchronized collections
  - `Collections.synchronizedMap`
  - `Collections.synchronizedSortedMap`
  - `Collections.synchronizedSortedSet`
  - `Collections.synchronizedSet`
  - `Collections.synchronizedList`
  - `Collections.synchronizedCollection`
- The views that are returned by these methods are sometimes called wrapper



# Views and Wrappers

---

- Subranges:
- You can form subrange views for a number of collections.  

```
List group2=staff.subList(10,20);
```
- You can apply any operation to the subrange, and they automatically reflect the entire list. For example, you can erase the entire subrange
  - `Group2.clear(); //staff reduction`



# Unsupported operations

---

- // Sometimes methods defined in the
- // Collection interfaces don't work!
- `import java.util.*;`
  
- `public class Unsupported {`
- `private static String[] s = {`
- `"one", "two", "three", "four", "five",`
- `"six", "seven", "eight", "nine", "ten",`
- `};`
- `static List a = Arrays.asList(s);`
- `static List a2 = a.subList(3, 6);`



# Unsupported operations

---

- `public static void main(String[] args) {`
- `System.out.println(a);`
- `System.out.println(a2);`
- `System.out.println(`
- `"a.contains(" + s[0] + ") = " +`
- `a.contains(s[0]));`
- `System.out.println(`
- `"a.containsAll(a2) = " +`
- `a.containsAll(a2));`
- `System.out.println("a.isEmpty() = " +`
- `a.isEmpty());`
- `System.out.println(`
- `"a.indexOf(" + s[5] + ") = " +`
- `a.indexOf(s[5]));`



# Unsupported operations

---

- `// Traverse backwards:`
- `ListIterator lit = a.listIterator(a.size());`
- `while(lit.hasPrevious())`
- `System.out.print(lit.previous() + " ");`
- `System.out.println();`
- `// Set the elements to different values:`
- `for(int i = 0; i < a.size(); i++)`
- `a.set(i, "47");`
- `System.out.println(a);`
- `// Compiles, but won't run:`
- `lit.add("X"); // Unsupported operation`
- `a.clear(); // Unsupported`



# Unsupported operations

---

- `a.add("eleven"); // Unsupported`
- `a.addAll(a2); // Unsupported`
- `a.retainAll(a2); // Unsupported`
- `a.remove(s[0]); // Unsupported`
- `a.removeAll(a2); // Unsupported`
- `}`
- `} ///:~`



# Unsupported operations

---

- The “unsupported operation” approach achieves an important goal of the Java containers library: the containers are simple to learn and use;



# Unsupported operations

---

- unsupported operations are a special case. For this approach to work, however:  
The **UnsupportedOperationException** must be a rare event. That is, for most classes all operations should work, and only in special cases should an operation be unsupported. This is true in the Java containers library, since the classes you'll use 99 percent of the time—**ArrayList**, **LinkedList**, **HashSet**, and **HashMap**, as well as the other concrete implementations—support all of the operations. The design does provide a “back door” if you want to create a new **Collection** without providing meaningful definitions for all the methods in the **Collection interface**, and yet still fit it into the existing library.



# Algorithms: Collections

---

- public class Collection extends Object
- This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.



# Algorithms: Collections

---

- `static int binarySearch(List list, Object key)` Searches the specified list for the specified object using the binary search algorithm.
- `static Object max(Collection coll)` Returns the maximum element of the given collection, according to the natural ordering of its elements.
- `static Object min(Collection coll)` Returns the minimum element of the given collection, according to the natural ordering of its elements.
- `static void reverse(List l)` Reverses the order of the elements in the specified list.



# Algorithms: Collections

---

- `static voidsort(List list)`      Sorts the specified list into ascending order, according to the natural ordering of its elements.
- `static voidreverse(List l)`      Reverses the order of the elements in the specified list.



# Property Sets

---

- A property set is a map structure of a very special type.
- Properties class implements property set  
public class Properties extends Hashtable  
The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string.



# Property Sets

---

- Property sets are useful in specifying configuration options for programs
- Example:
  - `public class SystemInfo{`
  - `public static void main(String args[]){`
  - `Properties systemProperties=System.getProperties();`
  - `Enumeration enum=systemProperties.propertyNames();`
  - `While(enum.hasMoreElements()){`
  - `String key=(String)enum.nextElement();`
  - `System.out.println(key+"="+systemProperties.getProperty(key));`



# Property Sets

---

- Example: CustomWorld.java



## Collection:Lab

---

- 1. Write a program that prints out its arguments in lexicographic (alphabetical) order.
- For example:
- `% java Sort i walk the line`
- `[i, line, the, walk]`
- 
-



## Collection:Lab

---

- `import java.util.*;`
  
- `public class Sort {`
- `public static void main(String args[]) {`
- `List l = Arrays.asList(args);`
- `Collections.sort(l);`
- `System.out.println(l);`
- `}`
- `}`



## Collection:Lab

---

- 2. Quicksort is a very efficient sorting algorithm invented by C.A.R. Hoare. It has two phases:
  - the partition phase and
  - the sort phase.
  - Thus the conquer part of the quicksort routine looks like this:  
  

```
quicksort( void *a, int low, int high ) {  
    int pivot; /* Termination condition! */  
    if ( high > low ) {  
        pivot = partition( a, low, high );  
        quicksort( a, low, pivot-1 );  
        quicksort( a, pivot+1, high );  
    }  
}
```



## Collection:Lab

---

- Write a Java program to implement Quick Sort.



# An example of the use of a **HashMap**

---

- `// Simple demonstration of HashMap.`
- `import java.util.*;`
  
- `class Counter {`
- `int i = 1;`
- `public String toString() {`
- `return Integer.toString(i);`
- `}`
- `}`



# An example of the use of a **HashMap**

---

```
■ class Statistics {
■     public static void main(String[] args) {
■         HashMap hm = new HashMap();
■         for(int i = 0; i < 10000; i++) {
■             // Produce a number between 0 and 20:
■             Integer r =
■                 new Integer((int)(Math.random() * 20));
■             if(hm.containsKey(r))
■                 ((Counter)hm.get(r)).i++;
■             else
■                 hm.put(r, new Counter());
■         }
■         System.out.println(hm);
■     }
■ } ///:~
```



# Java 1.0/1.1 containers

---

- **Vector & Enumeration**
  - **Vector** replaced by **ArrayList**
  - **Hashtable**
  - As you've seen in the performance comparison in this chapter, the basic **Hashtable** is very similar to the **HashMap**, even down to the method names. There's no reason to use **Hashtable** instead of **HashMap** in new code.
- Stack**
- Replaced by **LinkedList**.



# Java 1.0/1.1 containers

---

- // Demonstration of Stack Class.
- import java.util.\*;
- public class Stacks {
- static String[] months = {
- "January", "February", "March", "April",
- "May", "June", "July", "August", "September",
- "October", "November", "December" };
- public static void main(String[] args) {
-



# Java 1.0/1.1 containers

---

- Stack stk = new Stack();
- for(int i = 0; i < months.length; i++)
- stk.push(months[i] + " ");
- System.out.println("stk = " + stk);
- // Treating a stack as a Vector:
- stk.addElement("The last line");
- System.out.println(
- "element 5 = " + stk.elementAt(5));
- System.out.println("popping elements:");
- while(!stk.empty())
- System.out.println(stk.pop());
- }
- } ///:~



# Summary

---

- To review the containers provided in the standard Java library:
  1. An array associates numerical indices to objects. It holds objects of a known type so that you don't have to cast the result when you're looking up an object. It can be multidimensional, and it can hold primitives. However, its size cannot be changed once you create it.
  2. A **Collection** holds single elements, while a **Map** holds associated pairs.
  3. Like an array, a **List** also associates numerical indices to objects—you can think of arrays and **Lists** as ordered containers. The **List** automatically resizes itself as you add more elements. But a **List** can hold only **Object references**, so it won't hold primitives and you must always cast the result when you pull an **Object** reference out of a container.
  4. Use an **ArrayList** if you're doing a lot of random accesses, and a **LinkedList** if you will be doing a lot of insertions and removals in the middle of the list.



# Summary

- 
1. The behavior of queues, deques, and stacks is provided via the **LinkedList**.
  2. A **Map** is a way to associate, not numbers, but *objects* with other objects. The design of a **HashMap** is focused on rapid access, while a **TreeMap** keeps its keys in sorted order, and thus is not as fast as a **HashMap**.
  3. A **Set** only accepts one of each type of object. **HashSets** provide maximally-fast lookups, while **TreeSets** keep the elements in sorted order.
  4. There's no need to use the legacy classes **Vector**, **Hashtable** and **Stack** in new code.
- The containers are tools that you can use on a day-to-day basis to make your programs simpler, more powerful and more effective.