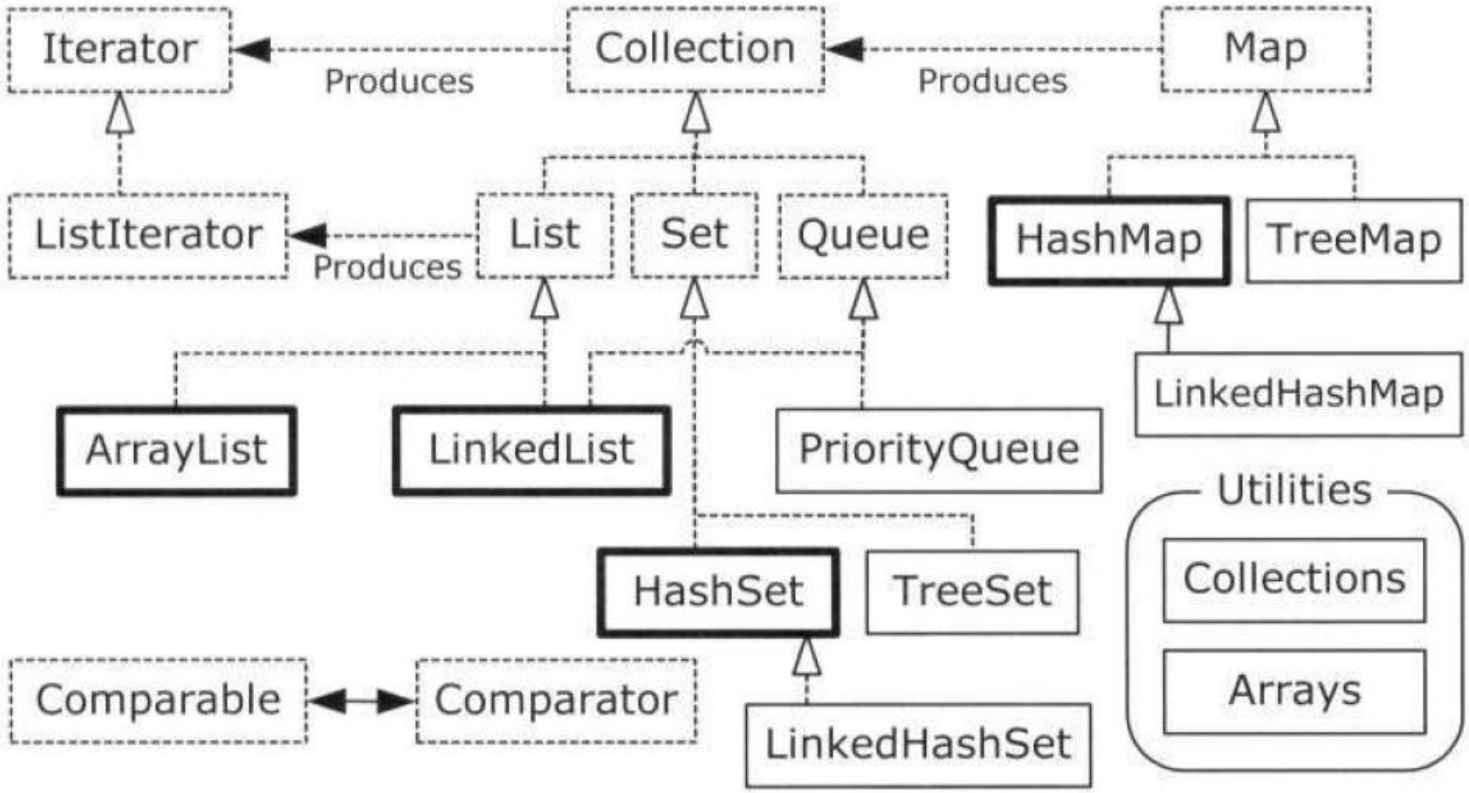
Holding Your Objects

Introduction

- It's a fairly simple program that only has a fixed quantity of objects with known lifetimes
- In general, your programs will always be creating new objects based on some criteria that will be known only at run time
  - Before then, you won't know the quantity or even the exact type of the objects you need
  - You can't rely on creating a named reference to hold each one of your objects
- An array is the most efficient way to hold a group of objects
  - But an array has a fixed size
- The java.util library has a reasonably complete set of *container classes* to solve this problem

Introduction (Cont.)

- A ***Collection*** **holds single elements, and a** ***Map*** **holds associated pairs**



Generics and type-safe containers

- One of the problems of using pre-Java SE5 containers was that the compiler allowed you to insert an incorrect type into a container

```
1  import java.util.*;
2
3  class Apple {
4    private static long counter;
5    private final long id = counter++;
6    public long id() { return id; }
7  }
8
9  class Orange {}
10
11 public class ApplesAndOrangesWithoutGenerics {
12   @SuppressWarnings("unchecked")
13   public static void main(String[] args) {
14     ArrayList apples = new ArrayList();
15     for(int i = 0; i < 3; i++)
16       apples.add(new Apple());
17     // Not prevented from adding an Orange to apples:
18     apples.add(new Orange());
19     for(int i = 0; i < apples.size(); i++)
20       ((Apple)apples.get(i)).id();
21       // Orange is detected only at run time
22   }
23 } /* (Execute to see output) *///:~
```

Generics and type-safe containers (Cont.)

- **With generics, you're prevented, *at compile time*, from putting the wrong type of object into a container**
  - **For example, to define an *ArrayList* intended to hold Apple objects, you say *ArrayList<Apple>* instead of just *ArrayList***
  - **The angle brackets surround the type parameters (there may be more than one), which specify the type(s) that can be held by that instance of the container**

```
1  import java.util.*;
2
3  public class ApplesAndOrangesWithGenerics {
4    public static void main(String[] args) {
5      ArrayList<Apple> apples = new ArrayList<Apple>();
6      for(int i = 0; i < 3; i++)
7        apples.add(new Apple());
8      // Compile-time error:
9      // apples.add(new Orange());
10     for(int i = 0; i < apples.size(); i++)
11       System.out.println(apples.get(i).id());
12     // Using foreach:
13     for(Apple c : apples)
14       System.out.println(c.id());
15   }
16 }
```

Basic concepts

- **The Java container library takes the idea of "holding your objects" and divides it into two distinct concepts, expressed as the basic interfaces of the library:**
  - ***Collection*: a sequence of individual elements with one or more rules applied to them**
    - **A *List* must hold the elements in the way that they were inserted**
    - **A *Set* cannot have duplicate elements**
    - **A *Queue* produces the elements in the order determined by a queuing discipline**
  - ***Map*: a group of key-value object pairs, allowing you to look up a value using a key**
    - **A *map* allows you to look up an object using another object**
    - **It's also called an *associative array*, because it associates objects with other objects, or a *dictionary*, because you look up a value object using a key object just like you look up a definition using a word**

Basic concepts (Cont.)

- **Ideally, we use these interfaces and specify the precise type you're using is at the point of creation**

```
List<Apple> apples = new ArrayList<Apple>();
```

- **Advantage: Easy to change your implementation, all you need to do is change it at the point of creation**

```
List<Apple> apples = new LinkedList<Apple>();
```

- **Make an object of a concrete class, upcast it to the corresponding interface, and then use the interface throughout the rest of your code**
- **This approach won't always work**
  - **E.g., *LinkedList* has additional methods that are not in the *List* interface**

Basic concepts (Cont.)

- **A simple example**

```
1  import java.util.*;
2
3  public class SimpleCollection {
4    public static void main(String[] args) {
5      Collection<Integer> c = new ArrayList<Integer>();
6      for(int i = 0; i < 10; i++)
7        c.add(i); // Autoboxing
8      for(Integer i : c)
9        System.out.print(i + ", ");
10   }
11 } /* Output:
12 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
13 *///:~
```

Adding groups of elements

- There are utility methods in both the *Arrays* and *Collections* classes in *java.util* that add groups of elements to a *Collection*
  - *Arrays.asList()* takes either an array or a comma-separated list of elements (using varargs) and turns it into a *List* object
  - *Collections.addAll()* takes a *Collection* object and either an array or a comma-separated list and adds the elements to the *Collection*

```
1  import java.util.*;
2
3  public class AddingGroups {
4    public static void main(String[] args) {
5      Collection<Integer> collection =
6        new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
7      Integer[] moreInts = { 6, 7, 8, 9, 10 };
8      collection.addAll(Arrays.asList(moreInts));
9      // Runs significantly faster, but you can't
10     // construct a Collection this way:
11     Collections.addAll(collection, 11, 12, 13, 14, 15);
12     Collections.addAll(collection, moreInts);
13     // Produces a list "backed by" an array:
14     List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
15     list.set(1, 99); // OK -- modify an element
16     // list.add(21); // Runtime error because the
17                      // underlying array cannot be resized.
18   }
19 } ///:~
```

Adding groups of elements

```
1  import java.util.*;
2
3  class Snow {}
4  class Powder extends Snow {}
5  class Light extends Powder {}
6  class Heavy extends Powder {}
7  class Crusty extends Snow {}
8  class Slush extends Snow {}
9
10 public class AsListInference {
11   public static void main(String[] args) {
12     List<Snow> snow1 = Arrays.asList(
13       new Crusty(), new Slush(), new Powder());
14
15     // Won't compile:
16     // List<Snow> snow2 = Arrays.asList(
17     //   new Light(), new Heavy());
18     // Compiler says:
19     // found   : java.util.List<Powder>
20     // required: java.util.List<Snow>
21
22     // Collections.addAll() doesn't get confused:
23     List<Snow> snow3 = new ArrayList<Snow>();
24     Collections.addAll(snow3, new Light(), new Heavy());
25
26     // Give a hint using an
27     // explicit type argument specification:
28     List<Snow> snow4 = Arrays.<Snow>asList(
29        new Light(), new Heavy());
30   }
31 } ///:~
```

- A limitation of ***Arrays.asList()*** is that it takes a best guess about the resulting type of the ***List***, and doesn't pay attention to what you're assigning it to. Sometimes this can cause a problem
- ***Collections.addAll()*** works fine because it knows from the first argument what the target type is
- it's possible to insert a "hint" in the middle of ***Arrays.asList( )***
  - This is called *an explicit type argument specification*

Printing containers

- **The containers print nicely *without any help***
  - **You must use *Arrays.toString()* to produce a printable representation of an array**

```
1  import java.util.*;
2  import static net.mindview.util.Print.*;
3
4  public class PrintingContainers {
5    static Collection fill(Collection<String> collection) {
6      collection.add("rat");
7      collection.add("cat");
8      collection.add("dog");
9      collection.add("dog");
10     return collection;
11   }
12   static Map fill(Map<String,String> map) {
13     map.put("rat", "Fuzzy");
14     map.put("cat", "Rags");
15     map.put("dog", "Bosco");
16     map.put("dog", "Spot");
17     return map;
18   }
19   public static void main(String[] args) {
20     print(fill(new ArrayList<String>()));
21     print(fill(new LinkedList<String>()));
22     print(fill(new HashSet<String>()));
23     print(fill(new TreeSet<String>()));
24     print(fill(new LinkedHashSet<String>()));
25     print(fill(new HashMap<String,String>()));
26     print(fill(new TreeMap<String,String>()));
27     print(fill(new LinkedHashMap<String,String>()));
28   }
29 }
```

```
/* Output:
[rat, cat, dog, dog]
[rat, cat, dog, dog]
[dog, cat, rat]
[cat, dog, rat]
[rat, cat, dog]
{dog=Spot, cat=Rags, rat=Fuzzy}
{cat=Rags, dog=Spot, rat=Fuzzy}
{rat=Fuzzy, cat=Rags, dog=Spot}
*///:~
```

List

- **There are two types of List:**
  - **The basic *ArrayList*, which excels at randomly accessing elements, but is slower when inserting and removing elements in the middle of a *List***
  - **The *LinkedList*, which provides optimal sequential access, with inexpensive insertions and deletions from the middle of the *List*. A *LinkedList* is relatively slow for random access, but it has a larger feature set than the *ArrayList***

List (Cont.)

```java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ListFeatures {
  public static void main(String[] args) {
    Random rand = new Random(47);
    List<Pet> pets = Pets.arrayList(7);
    print("1: " + pets);
    Hamster h = new Hamster();
    pets.add(h); // Automatically resizes
    print("2: " + pets);
    print("3: " + pets.contains(h));
    pets.remove(h); // Remove by object
    Pet p = pets.get(2);
    print("4: " +  p + " " + pets.indexOf(p));
    Pet cymric = new Cymric();
    print("5: " + pets.indexOf(cymric));
    print("6: " + pets.remove(cymric));
    // Must be the exact object:
    print("7: " + pets.remove(p));
    print("8: " + pets);
    pets.add(3, new Mouse()); // Insert at an index
    print("9: " + pets);
    List<Pet> sub = pets.subList(1, 4);
    print("subList: " + sub);
    print("10: " + pets.containsAll(sub));
    Collections.sort(sub); // In-place sort
    print("sorted subList: " + sub);
```

/* Output:
1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster]
3: true
4: Cymric 2
5: -1
6: false
7: true
8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
subList: [Manx, Mutt, Mouse]
10: true
sorted subList: [Manx, Mouse, Mutt]

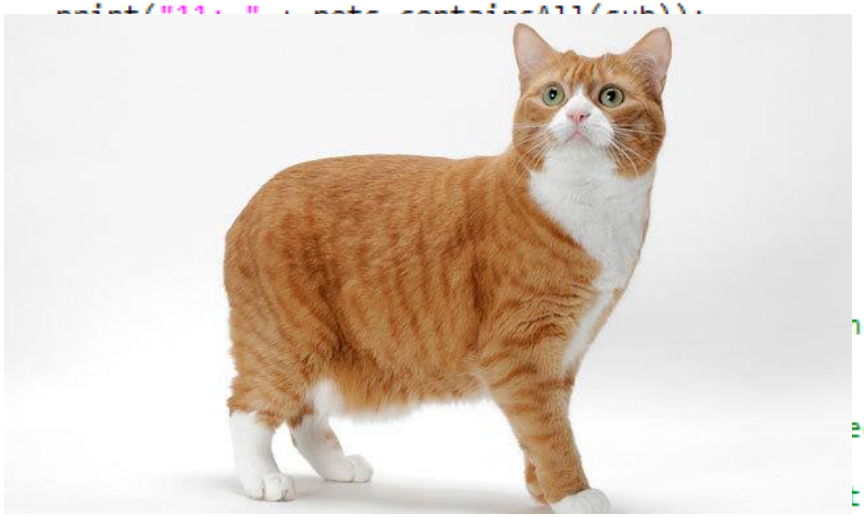
List (Cont.)

```
11: true
shuffled subList: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Manx, Cymric, Rat, EgyptianMau]
22: EgyptianMau
23: 14
*///:~
```

```
30      // Order is not important in containsAll():
31      print("11: " + pets.containsAll(sub));
32      Collections.shuffle(sub, rand); // Mix it up
33      print("shuffled subList: " + sub);
34      print("12: " + pets.containsAll(sub));
35      List<Pet> copy = new ArrayList<Pet>(pets);
36      sub = Arrays.asList(pets.get(1), pets.get(4));
37      print("sub: " + sub);
38      copy.retainAll(sub);
39      print("13: " + copy);
40      copy = new ArrayList<Pet>(pets); // Get a fresh copy
41      copy.remove(2); // Remove by index
42      print("14: " + copy);
43      copy.removeAll(sub); // Only removes exact objects
44      print("15: " + copy);
45      copy.set(1, new Mouse()); // Replace an element
46      print("16: " + copy);
47      copy.addAll(2, sub); // Insert a list in the middle
48      print("17: " + copy);
49      print("18: " + pets.isEmpty());
50      pets.clear(); // Remove all elements
51      print("19: " + pets);
52      print("20: " + pets.isEmpty());
53      pets.addAll(Pets.arrayList(4));
54      print("21: " + pets);
55      Object[] o = pets.toArray();
56      print("22: " + o[3]);
57      Pet[] pa = pets.toArray(new Pet[0]);
58      print("23: " + pa[3].id());
59    }
60  }
```

List (Cont.)

```
11
sh
12
sul
13
14
15
16
17
18: fal
19: []
20: tru
21: [M
22: Eg
23: 14
*///:~
```



```
30      // Order is not important in containsAll():
                                                    copy
                                                    ects
                                                    t
print("16: " + copy);
                                                    ile
47
48
49
50
51
52
53
54
55
56
57
58
59
60  }
```

List (Cont.)

```
11: true
shuffled subList: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Manx, Cymric, Rat, EgyptianMau]
22: EgyptianMau
23: 14
*///:~
```

```
30      // Order is not important in containsAll():
31      print("11: " + pets.containsAll(sub));
32      Collections.shuffle(sub, rand); // Mix it up
33      print("shuffled subList: " + sub);
34      print("12: " + pets.containsAll(sub));
35      List<Pet> copy = new ArrayList<Pet>(pets);
36      sub = Arrays.asList(pets.get(1), pets.get(4));
37      print("sub: " + sub);
38      copy.retainAll(sub);
39      print("13: " + copy);
40      copy = new ArrayList<Pet>(pets); // Get a fresh copy
41      copy.remove(2); // Remove by index
42      print("14: " + copy);
43      copy.removeAll(sub); // Only removes exact objects
44      print("15: " + copy);
45      copy.set(1, new Mouse()); // Replace an element
46      print("16: " + copy);
47      copy.addAll(2, sub); // Insert a list in the middle
48      print("17: " + copy);
49      print("18: " + pets.isEmpty());
50      pets.clear(); // Remove all elements
51      print("19: " + pets);
52      print("20: " + pets.isEmpty());
53      pets.addAll(Pets.arrayList(4));
54      print("21: " + pets);
55      Object[] o = pets.toArray();
56      print("22: " + o[3]);
57      Pet[] pa = pets.toArray(new Pet[0]);
58      print("23: " + pa[3].id());
59    }
60  }
```

Iterator

- An *iterator* is an object whose job is to move through a sequence and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence
- An iterator is usually what's called a *lightweight object*: one that's cheap to create
- Limitation: *Iterator* can move in only *one direction*
- There's not much you can do with an Iterator except:
  1. Ask a Collection to hand you an Iterator using a method called *iterator( )*. That Iterator will be ready to return the first element in the sequence
  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( )*

Iterator (Cont.)

- With an Iterator, you don't need to worry about the number of elements in the container. That's taken care of for you by *hasNext( )* and *next( )*
  - iterators unify access to containers

```
1  import typeinfo.pets.*;
2  import java.util.*;
3
4  public class SimpleIteration {
5    public static void main(String[] args) {
6      List<Pet> pets = Pets.arrayList(12);
7      Iterator<Pet> it = pets.iterator();
8      while(it.hasNext()) {
9        Pet p = it.next();
10       System.out.print(p.id() + ":" + p + " ");
11     }
12     System.out.println();
13     // A simpler approach, when possible:
14     for(Pet p : pets)
15       System.out.print(p.id() + ":" + p + " ");
16     System.out.println();
17     // An Iterator can also remove elements:
18     it = pets.iterator();
19     for(int i = 0; i < 6; i++) {
20       it.next();
21       it.remove();
22     }
23     System.out.println(pets);
24   }
25 }
```

```
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric
6:Pug 7:Manx 8:Cymric 9:Rat 10:EgyptianMau
11:Hamster
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric
6:Pug 7:Manx 8:Cymric 9:Rat 10:EgyptianMau
11:Hamster
[Pug, Manx, Cymric, Rat, EgyptianMau,
Hamster]
*///:~
```

ListIterator

- The ***ListIterator*** is a more powerful subtype of ***Iterator*** that is produced only by List classes
  - ***ListIterator*** is ***bidirectional***

```
1  import typeinfo.pets.*;
2  import java.util.*;
3
4  public class ListIteration {
5    public static void main(String[] args) {
6      List<Pet> pets = Pets.arrayList(8);
7      ListIterator<Pet> it = pets.listIterator();
8      while(it.hasNext())
9        System.out.print(it.next() + ", " + it.nextIndex() +
10         ", " + it.previousIndex() + "; ");
11     System.out.println();
12     // Backwards:
13     while(it.hasPrevious())
14       System.out.print(it.previous().id() + " ");
15     System.out.println();
16     System.out.println(pets);
17     it = pets.listIterator(3);
18     while(it.hasNext()) {
19       it.next();
20       it.set(Pets.randomPet());
21     }
22     System.out.println(pets);
23   }
24 }
```

```
/* Output:
Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2;
Mutt, 4, 3; Pug, 5, 4; Cymric, 6, 5;
Pug, 7, 6; Manx, 8, 7;
7 6 5 4 3 2 1 0
[Rat, Manx, Cymric, Mutt, Pug,
Cymric, Pug, Manx]
[Rat, Manx, Cymric, Cymric, Rat,
EgyptianMau, Hamster,
EgyptianMau]
*///:~
```

LinkedList

- The *LinkedList* implements the basic List interface like *ArrayList* does
  - Operations of insertion and removal in the middle of the List more efficiently than does *ArrayList*
- The *LinkedList* can be used as a stack, a Queue or a double-ended queue (deque)
  - *getFirst( )* and *element( )* are identical—they return the head (first element) of the list without removing it, and throw NoSuchElementException if the List is empty
  - *peek( )* is a variation of those two that returns null if the list is empty
  - *removeFirst( )* and *remove( )* are also identical—remove and return the head of the list, and throw *NoSuchElementException* for an empty list
  - *poll( )* is a variation that returns null if this list is empty
  - *addFirst( )* inserts an element at the beginning of the list
  - *offer( )* is the same as *add( )* and *addLast( )*. They all add an element to the tail (end) of a list.
  - *removeLast( )* removes and returns the last element of the list

LinkedList (Cont.)

```
1  import typeinfo.pets.*;
2  import java.util.*;
3  import static net.mindview.util.Print.*;
4
5  public class LinkedListFeatures {
6    public static void main(String[] args) {
7      LinkedList<Pet> pets =
8        new LinkedList<Pet>(Pets.arrayList(5));
9      print(pets);
10     // Identical:
11     print("pets.getFirst(): " + pets.getFirst());
12     print("pets.element(): " + pets.element());
13     // Only differs in empty-list behavior:
14     print("pets.peek(): " + pets.peek());
15     // Identical; remove and return the first element:
16     print("pets.remove(): " + pets.remove());
17     print("pets.removeFirst(): " + pets.removeFirst());
18     // Only differs in empty-list behavior:
19     print("pets.poll(): " + pets.poll());
20     print(pets);
21     pets.addFirst(new Rat());
22     print("After addFirst(): " + pets);
23     pets.offer(Pets.randomPet());
24     print("After offer(): " + pets);
25     pets.add(Pets.randomPet());
26     print("After add(): " + pets);
27     pets.addLast(new Hamster());
28     print("After addLast(): " + pets);
29     print("pets.removeLast(): " + pets.removeLast());
30   }
31 }
```

```
/* Output:
[Rat, Manx, Cymric, Mutt, Pug]
pets.getFirst(): Rat
pets.element(): Rat
pets.peek(): Rat
pets.remove(): Rat
pets.removeFirst(): Manx
pets.poll(): Cymric
[Mutt, Pug]
After addFirst(): [Rat, Mutt, Pug]
After offer(): [Rat, Mutt, Pug, Cymric]
After add(): [Rat, Mutt, Pug, Cymric, Pug]
After addLast(): [Rat, Mutt, Pug, Cymric,
 Pug, Hamster]
pets.removeLast(): Hamster
*///:~
```

Stack

- A stack is sometimes referred to as a "last-in, first-out" (LIFO) container

```java
// Making a stack from a LinkedList.
package net.mindview.util;
import java.util.LinkedList;

public class Stack<T> {
  private LinkedList<T> storage = new LinkedList<T>();
  public void push(T v) { storage.addFirst(v); }
  public T peek() { return storage.getFirst(); }
  public T pop() { return storage.removeFirst(); }
  public boolean empty() { return storage.isEmpty(); }
  public String toString() { return storage.toString(); }
} ///:~
```

```java
import net.mindview.util.*;

public class StackTest {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.empty())
      System.out.print(stack.pop() + " ");
  }
}
```

/* Output:
fleas has dog My
***///:~**

```java
import net.mindview.util.*;

public class StackCollision {
  public static void main(String[] args) {
    net.mindview.util.Stack<String> stack =
      new net.mindview.util.Stack<String>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.empty())
      System.out.print(stack.pop() + " ");
    System.out.println();
    java.util.Stack<String> stack2 =
      new java.util.Stack<String>();
    for(String s : "My dog has fleas".split(" "))
      stack2.push(s);
    while(!stack2.empty())
      System.out.print(stack2.pop() + " ");
  }
}
```

/* Output:
fleas has dog My
fleas has dog My
***///:~**

Set

- A *Set* refuses to hold more than one instance of each object value
  - The *Set* prevents duplication
  - The most common use for a *Set* is to test for membership
- Lookup is typically the most important operation for a Set, so you'll usually choose a *HashSet* implementation, which is optimized for rapid lookup

```
1  import java.util.*;
2
3  public class SetOfInteger {
4    public static void main(String[] args) {
5      Random rand = new Random(47);
6      Set<Integer> intset = new HashSet<Integer>();
7      for(int i = 0; i < 10000; i++)
8        intset.add(rand.nextInt(30));
9      System.out.println(intset);
10   }
11 }
```

/* Output:
[15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 29, 14, 24, 4,
19, 26, 11, 18, 3, 12, 27, 17, 2, 13, 28, 20, 25,
10, 5, 0]
***///:~**

Set (Cont.)

- The order maintained by a *HashSet* is different from a *TreeSet* or a *LinkedHashSet*
  - https://www.cs.usfca.edu/~galles/visualization/OpenHash.html
- *TreeSet* keeps elements sorted into a red-black tree data structure, whereas *HashSet* uses the hashing function
  - https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
- *LinkedHashSet* also uses hashing for lookup speed, but appears to maintain elements in insertion order using a linked list

```
1  import java.util.*;
2
3  public class SortedSetOfInteger {
4    public static void main(String[] args) {
5      Random rand = new Random(47);
6      SortedSet<Integer> intset = new TreeSet<Integer>();
7      for(int i = 0; i < 10000; i++)
8        intset.add(rand.nextInt(30));
9      System.out.println(intset);
10   }
11 }
```

```
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29]
*///:~
```

Set (Cont.)

- One of the most common operations you will perform is a test for set membership using ***contains( )***

```
1  import java.util.*;
2  import static net.mindview.util.Print.*;
3
4  public class SetOperations {
5    public static void main(String[] args) {
6      Set<String> set1 = new HashSet<String>();
7      Collections.addAll(set1,
8        "A B C D E F G H I J K L".split(" "));
9      set1.add("M");
10     print("H: " + set1.contains("H"));
11     print("N: " + set1.contains("N"));
12     Set<String> set2 = new HashSet<String>();
13     Collections.addAll(set2, "H I J K L".split(" "));
14     print("set2 in set1: " + set1.containsAll(set2));
15     set1.remove("H");
16     print("set1: " + set1);
17     print("set2 in set1: " + set1.containsAll(set2));
18     set1.removeAll(set2);
19     print("set2 removed from set1: " + set1);
20     Collections.addAll(set1, "X Y Z".split(" "));
21     print("'X Y Z' added to set1: " + set1);
22   }
23 }
```

```
/* Output:
H: true
N: false
set2 in set1: true
set1: [D, K, C, B, L, G, I, M, A, F, J, E]
set2 in set1: false
set2 removed from set1: [D, C, B, G,
 M, A, F, E]
'X Y Z' added to set1: [Z, D, C, B, G,
 M, A, F, Y, X, E]
*///:~
```

Map

- **The ability to map objects to other objects can be an immensely powerful way to solve programming problems**
  - **For example, consider a program to examine the randomness of Java's *Random* class**

```
1  // Simple demonstration of HashMap.
2  import java.util.*;
3
4  public class Statistics {
5    public static void main(String[] args) {
6      Random rand = new Random(47);
7      Map<Integer,Integer> m =
8        new HashMap<Integer,Integer>();
9      for(int i = 0; i < 10000; i++) {
10       // Produce a number between 0 and 20:
11       int r = rand.nextInt(20);
12       Integer freq = m.get(r);
13       m.put(r, freq == null ? 1 : freq + 1);
14     }
15     System.out.println(m);
16   }
17 }
```

/* Output:
{15=497, 4=481, 19=464, 8=468,
11=531, 16=533, 18=478, 3=508,
7=471, 12=521, 17=509, 2=489,
13=506, 9=549, 6=519, 1=502,
14=477, 10=513, 5=503, 0=481}
***///:~**

Map (Cont.)

- **Here's an example that allows you to use a *String* description to look up Pet objects**
  - **It shows how you can test a *Map* to see if it contains a key or a value with *containsKey( )* and *containsValue( )***

```
1  import typeinfo.pets.*;
2  import java.util.*;
3  import static net.mindview.util.Print.*;
4
5  public class PetMap {
6    public static void main(String[] args) {
7      Map<String,Pet> petMap = new HashMap<String,Pet>();
8      petMap.put("My Cat", new Cat("Molly"));
9      petMap.put("My Dog", new Dog("Ginger"));
10     petMap.put("My Hamster", new Hamster("Bosco"));
11     print(petMap);
12     Pet dog = petMap.get("My Dog");
13     print(dog);
14     print(petMap.containsKey("My Dog"));
15     print(petMap.containsValue(dog));
16   }
17 }
```

```
/* Output:
{My Cat=Cat Molly,
My Hamster=Hamster Bosco,
My Dog=Dog Ginger}
Dog Ginger
true
true
*///:~
```

Queue

- A *queue* is typically a "*first-in, first-out*" (FIFO) container
  - https://www.cs.usfca.edu/~galles/visualization/QueueLL.html
- Queues are commonly used as a way to reliably transfer objects from one area of a program to another
- *LinkedList* has methods to support queue behavior and it implements the *Queue* interface, so a *LinkedList* can be used as a *Queue* implementation

```
1  // Upcasting to a Queue from a LinkedList.
2  import java.util.*;
3
4  public class QueueDemo {
5    public static void printQ(Queue queue) {
6      while(queue.peek() != null)
7        System.out.print(queue.remove() + " ");
8      System.out.println();
9    }
10   public static void main(String[] args) {
11     Queue<Integer> queue = new LinkedList<Integer>();
12     Random rand = new Random(47);
13     for(int i = 0; i < 10; i++)
14       queue.offer(rand.nextInt(i + 10));
15     printQ(queue);
16     Queue<Character> qc = new LinkedList<Character>();
17     for(char c : "Brontosaurus".toCharArray())
18       qc.offer(c);
19     printQ(qc);
20   }
21 }
```

```
/* Output:
8 1 1 1 5 14 3 1 0 1
B r o n t o s a u r u s
*///:~
```

Queue (Cont.)

- ***offer( )*** **inserts an element at the tail of the queue if it can, or returns false**
- **Both *peek( )* and *element( )* return the head of the queue without removing it**
  - ***peek( )*** **returns *null* if the queue is empty**
  - ***element( )*** **throws *NoSuchElementException***
- **Both *poll( )* and *remove( )* remove and return the head of the queue**
  - ***poll( )*** **returns *null* if the queue is empty**
  - ***remove( )*** **throws *NoSuchElementException***

PriorityQueue

- ***First-in, first-out (FIFO)*** **describes the most typical** ***queuing discipline***
  - **First-in, first-out says that the next element should be the one that was waiting the longest**
- ***A priority queue*** **says that the element that goes next is the one with the greatest need (the highest priority)**
- **When you** ***offer( )*** **an object onto a** ***PriorityQueue*****, that object is sorted into the queue**
  - **The default sorting uses the** ***natural order*** **of the objects in the queue, but you can modify the order by providing your own** ***Comparator***
- **The** ***PriorityQueue*** **ensures that when you call** ***peek( )*****,** ***poll( )*** **or** ***remove( )*****, the element you get will be the one with the highest priority**

PriorityQueue (Cont.)

```
1  import java.util.*;
2
3  public class PriorityQueueDemo {
4    public static void main(String[] args) {
5      PriorityQueue<Integer> priorityQueue =
6        new PriorityQueue<Integer>();
7      Random rand = new Random(47);
8      for(int i = 0; i < 10; i++)
9        priorityQueue.offer(rand.nextInt(i + 10));
10     QueueDemo.printQ(priorityQueue);
11
12     List<Integer> ints = Arrays.asList(25, 22, 20,
13       18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
14     priorityQueue = new PriorityQueue<Integer>(ints);
15     QueueDemo.printQ(priorityQueue);
16     priorityQueue = new PriorityQueue<Integer>(
17         ints.size(), Collections.reverseOrder());
18     priorityQueue.addAll(ints);
19     QueueDemo.printQ(priorityQueue);
20
21     String fact = "EDUCATION SHOULD ESCHEW OBFUSCATION";
22     List<String> strings = Arrays.asList(fact.split(""));
23     PriorityQueue<String> stringPQ =
24       new PriorityQueue<String>(strings);
25     QueueDemo.printQ(stringPQ);
26     stringPQ = new PriorityQueue<String>(
27       strings.size(), Collections.reverseOrder());
28     stringPQ.addAll(strings);
29     QueueDemo.printQ(stringPQ);
30
31     Set<Character> charSet = new HashSet<Character>();
32     for(char c : fact.toCharArray())
33       charSet.add(c); // Autoboxing
34     PriorityQueue<Character> characterPQ =
35       new PriorityQueue<Character>(charSet);
36     QueueDemo.printQ(characterPQ);
37   }
38 }
```

/* Output:

0 1 1 1 1 1 3 5 8 14

1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25

25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1

A A B C C C D D E E E F H H I I L N N O O O

O S S S T T U U U W

W U U U T T S S S O O O O N N L I I H H F E E E

D D C C C B A A

A B C D E F H I L N O S T U W

***///:~**

Collection vs. Iterator

- **Collection is the root interface that describes what is common for all sequence containers**
  - **One argument for having an interface is that it allows you to create more generic code**
- **The Standard C++ Library has no common base class for its containers—all commonality between containers is achieved through *iterators***
- **In Java, it expresses commonality between containers using an *iterator* rather than a *Collection***
- **The two approaches are bound together, since implementing *Collection* also means providing an *iterator( )* method**

Collection vs. Iterator (Cont.)

```
1  import typeinfo.pets.*;
2  import java.util.*;
3
4  public class InterfaceVsIterator {
5    public static void display(Iterator<Pet> it) {
6      while(it.hasNext()) {
7        Pet p = it.next();
8        System.out.print(p.id() + ":" + p + " ");
9      }
10     System.out.println();
11   }
12   public static void display(Collection<Pet> pets) {
13     for(Pet p : pets)
14       System.out.print(p.id() + ":" + p + " ");
15     System.out.println();
16   }
17   public static void main(String[] args) {
18     List<Pet> petList = Pets.arrayList(8);
19     Set<Pet> petSet = new HashSet<Pet>(petList);
20     Map<String,Pet> petMap =
21       new LinkedHashMap<String,Pet>();
22     String[] names = ("Ralph, Eric, Robin, Lacey, " +
23       "Britney, Sam, Spot, Fluffy").split(", ");
24     for(int i = 0; i < names.length; i++)
25       petMap.put(names[i], petList.get(i));
26     display(petList);
27     display(petSet);
28     display(petList.iterator());
29     display(petSet.iterator());
30     System.out.println(petMap);
31     System.out.println(petMap.keySet());
32     display(petMap.values());
33     display(petMap.values().iterator());
34   }
35 }
```

```
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric
6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx
2:Cymric 0:Rat
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric
6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx
2:Cymric 0:Rat
{Ralph=Rat, Eric=Manx, Robin=Cymric,
Lacey=Mutt, Britney=Pug, Sam=Cymric,
Spot=Pug, Fluffy=Manx}
[Ralph, Eric, Robin, Lacey, Britney, Sam, Spot,
Fluffy]
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric
6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric
6:Pug 7:Manx
*///:~
```

Foreach and iterators

- The *foreach* syntax has been primarily used with arrays, but it also works with any *Collection* object

```
1  // All collections work with foreach.
2  import java.util.*;
3
4  public class ForEachCollections {
5    public static void main(String[] args) {
6      Collection<String> cs = new LinkedList<String>();
7      Collections.addAll(cs,
8        "Take the long way home".split(" "));
9      for(String s : cs)
10       System.out.print("'" + s + "' ");
11   }
12 }
```

```
/* Output:
'Take' 'the' 'long' 'way' 'home'
*///:~
```

Foreach and iterators (Cont.)

- The reason that this works is that Java SE5 introduced a new interface called *Iterable* which contains an *iterator( )* method to produce an *Iterator*
  - if you create any class that implements *Iterable*, you can use it in a foreach statement

```
1  // Anything Iterable works with foreach.
2  import java.util.*;
3
4  public class IterableClass implements Iterable<String> {
5    protected String[] words = ("And that is how " +
6      "we know the Earth to be banana-shaped.").split(" ");
7    public Iterator<String> iterator() {
8      return new Iterator<String>() {
9        private int index = 0;
10       public boolean hasNext() {
11         return index < words.length;
12       }
13       public String next() { return words[index++]; }
14       public void remove() { // Not implemented
15         throw new UnsupportedOperationException();
16       }
17     };
18   }
19   public static void main(String[] args) {
20     for(String s : new IterableClass())
21       System.out.print(s + " ");
22   }
23 }
```

```
/* Output:
And that is how we know the
Earth to be banana-shaped.
*///:~
```

Thank you

zhenling@seu.edu.cn