CHAPTER SUMMARY

- A Java variable can be one of the eight primitive data types. Anything that's not one of the eight primitive types is a reference to a Java `Object`.
- An assignment `x = y` is a **widening conversion** if the data type of `x` is capable of referring to a wider variety of things than the type of `y`. It is a **narrowing conversion** if the data type of `x` is capable of referring to a smaller variety of things than the type of `y`. Java always permits widening conversions, but narrowing conversions require a typecast.
- A **wrapper class** is a class in which each object holds a primitive value. Java provides wrapper classes for each of the eight primitive types. In many situations, Java will carry out automatic conversions from a primitive value to a wrapper object (**autoboxing**) or the other way (**auto-unboxing**).
- A **generic method** is similar to an ordinary method with one importnat difference: The definition of a generic method can depend on an underlying data type. The underlying data type is given a name, such as `T`, but `T` is not pinned down to a specific type anywhere in the method's implementation.
- When a class depends on an underlying data type, the class can be implemented as a **generic class**. Converting a collection class to a generic class that holds objects is usually a small task. For example, we converted the `IntArrayBag` to an `ArrayBag` by following the steps on page 258.
- An interface provides a list of methods for a class to implement. By writing a class that implements one of the standard Java interfaces, you make it easier for other programmers to use your class. There may also be existing programs already written that work with some of the standard interfaces.
- Java's `Iterator<E>` generic interface provides an easy way to step through all the elements of a collection class. A class that implements Java's `Iterator` interface must provide two methods:

  ```
  public boolean hasNext( )
  public E next( )
  ```

  An `Iterator` must also have a `remove` method, although if removal is not supported, then the `remove` method can simply throw an exception.
- Two classes in this chapter have wide applicability, and you'll find them useful in the future: (1) the `Node` class from Appendix E, which is a node from a linked list of objects; and (2) the `LinkedBag` class from Appendix F, which includes a method to generate an `Iterator` for its elements.
- Java provides several different standard collection classes that implement the `Collection` interface (such as `Vector`) and the `Map` interface (such as `TreeMap`).

Solutions to Self-Test Exercises

1. Yes, yes, yes.

2. This code has both a widening conversion (marked with the circle) and a narrowing conversion (marked with a triangle):

   ```
   String s = new String("Liberty!");
   Object obj;
   obj = s;●
   s = (String) obj;△
   ```

3. Here is one example that causes a class cast exception at runtime:

   ```
   String s = new String("Liberty!");
   Integer i;
   Object obj;
   obj = s;
   i = (Integer) obj;
   ```

4. ```
   Character example
      = new Character('w');
   ```

5. Advantage: When a primitive value is placed in a wrapper object, it can be treated just like any other Java `Object`. Disadvantage: A wrapper object can no longer use the primitive operations, such as the arithmetic operations.

6. First, `x` and `y` are unboxed. Then the double numbers are added, resulting in a double answer. This answer is boxed and assigned to `z`.

7. A generic method is a typesafe way to write a single method that can be used with a variety of different types of parameters.

8. The compiler can discover more type errors with a generic method than with an object method.

9. The generic type parameter first appears in angle brackets before the return type of the generic method. It may later appear within the return type, the parameter list, or the implementation of the generic method. Almost all situation require it to appear at least once in the method's parameter list.

10. Create a new object of that type, or create an array with that type of elements.

11. ```
    public static <E> int
    count(E[] data, E target)
    {
       int answer = 0;
       if (target == null)
       {
          for (E next : data)
          {
             if (next == null)
                answer++;
          }
       }
       else
       {
          for (E next : data)
          {
             if (target.equals(next))
                answer++;
          }
       }
       return answer;
    }
    ```

12. We use the `count` method from the previous exercise:

    ```
    static <S,T> bool most(
       S[] sa, S starget,
       T[] ta, T ttarget
       )
    {
       return count(sa, starget)
              > count(ta, ttarget);
    }
    ```

13. No. The only conversions from `int` to `Object` are the places where the data type refers to an element in the bag. For example, the return value from `size` remains an `int`.

14. Yes, we allow `add(null)`, but we also need special code in `countOccurrences` and `remove` to handle the null reference.

15. Here is the code:

```
ArrayBag numbers;
int i;
numbers = new ArrayBag( );
for (i = 1; i <= 10; i++)
   numbers.add(new Integer(i));
```

16. In the new `countOccurrences`, we use this for non-null targets:

```
target.equals(data[index])
```

17. It used `stdin.stringInput`, where `stdin` is an `EasyReader` from Appendix B.

18. A reference to the moving object was put in the bag. When the moving object changes it position, the original location is no longer in the bag. So, the first `countOccurrences` prints zero, and the second prints one.

19. To handle searching for a null target.

20. Suppose the data in the first node was equal to the data in the end node. Then the boolean expression will be false right at the start, and only one node will be copied.

21. The expression `(x == y)` is `true` if `x` and `y` refer to exactly the same node. The expression `(x.data.equals(y.data))` is `true` if the data in the two nodes is the same.

22. The methods are `charAt`, `length`, `subSequence`, and `toString`.

23.
```
public static <T> int countNull
(Node<T> head)
{
    int answer = 0;
    Lister<T> it =
       new Lister<T>(head);
    while (it.hasNext( ))
    {
        if (it.next( ) == null)
           count++;
    }
    return count;
}
```

24.
```
public static <T> T find
(Node<T> head, Comparable x)
{
    Lister<T> it =
       new Lister<T>(head);
    T d;
    while (it.hasNext( ))
    {
        d = it.next( );
        if (x.compareTo(d) == 0)
           return d;
    }
    return null;
}
```

25.
```
public Lister(Node<T> head)
{
   current = Node<T>.listCopy(head);
}
```

26.
```
(x instanceOf Comparable<Integer>)
```

27. None. The only purpose of the `Cloneable` interface to to allow a method to check whether an object is an instance of `Cloneable`.

28.
```
import java.util.Iterator;

public class ArrayIterator<E>
implements Iterator<E>
{
   private E[ ] array;
   private int index;

   public ArrayIterator
   (E[ ] things)
   {
      array = things;
      index = 0;
   }
   public boolean hasNext( )
   {
      return (index < array.length);
   }
   public E next( )
   {
      return array[index++];
   }
```

...See page 279 for remove...

```
}
```