Chapter 8

Iterators

Concepts:
- ▷ Iterators
- ▷ The `AbstractIterator` class
- ▷ `Vector` iterators
- ▷ Numeric iteration

One potato, two potato, three potato, four,
five potato, six potato, seven potato, more.
—A child's iterator

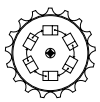
PROGRAMS MOVE FROM ONE STATE TO ANOTHER. As we have seen, this "state" is composed of the current value of user variables as well as some notion of "where" the computer is executing the program. This chapter discusses *enumerations* and *iterators*—objects that hide the complexities of maintaining the state of a traversal of a data structure.

Ah! Interstate programs!

Consider a program that prints each of the values in a list. It is important to maintain enough information to know exactly "where we are" at all times. This might correspond to a reference to the current value. In other structures it may be less clear how the state of a traversal is maintained. Iterators help us hide these complexities. The careful design of these *control structures* involves, as always, the development of a useful interface that avoids compromising the iterator's implementation or harming the object it traverses.

8.1 Java's Enumeration Interface

Java defines an interface called an `Enumeration` that provides the user indirect, iterative access to each of the elements of an associated data structure, exactly once. The `Enumeration` is returned as the result of calling the `elements` method of various container classes. Every `Enumeration` provides two methods:



`Enumeration`

```
public interface java.util.Enumeration
{
    public abstract boolean hasMoreElements();
    // post: returns true iff enumeration has outstanding elements

    public abstract java.lang.Object nextElement();
    // pre: hasMoreElements
    // post: returns the next element to be visited in the traversal
}
```

The `hasMoreElements` method returns `true` if there are unvisited elements of the associated structure. When `hasMoreElements` returns `false`, the traversal is finished and the `Enumeration` expires. To access an element of the underlying structure, `nextElement` must be called. This method does two things: it returns a reference to the current element and then marks it visited. Typically `hasMoreElements` is the predicate of a `while` loop whose body processes a single element using `nextElement`. Clearly, `hasMoreElements` is an important method, as it provides a test to see if the precondition for the `nextElement` method is met.

The following code prints out a catchy phrase using a `Vector` enumeration:

HelloWorld

```
public static void main(String args[])
{
    // construct a vector containing two strings:
    Vector<String> v = new Vector<String>();
    v.add("Hello");
    v.add("world!");

    // construct an enumeration to view values of v
    Enumeration i = (Enumeration)v.elements();
    while (i.hasMoreElements())
    {
        // SILLY: v.add(1,"silly");
        System.out.print(i.nextElement()+" ");
    }
    System.out.println();
}
```

When run, the following immortal words are printed:

```
Hello world!
```

There are some important caveats that come with the use of Java's `Enumeration` construct. First, it is important to avoid modifying the associated structure while the `Enumeration` is active or *live*. Uncommenting the line marked `SILLY` causes the following infinite output to begin:

```
Hello silly silly silly silly silly silly
```

A silly virus vector!

Inserting the string `"silly"` as the new second element of the `Vector` causes it to expand each iteration of the loop, making it difficult for the `Enumeration` to detect the end of the `Vector`.

Principle 9 *Never modify a data structure while an associated* `Enumeration` *is live.*

Modifying the structure behind an `Enumeration` can lead to unpredictable results. Clearly, if the designer has done a good job, the implementations of both

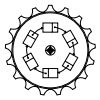
the Enumeration and its associated structure are hidden. Making assumptions about their interaction can be dangerous.

Another subtle aspect of Enumerations is that they do not guarantee a particular traversal order. All that is known is that each element will be visited exactly once before hasMoreElements becomes false. While we assume that our first example above will print out Hello world!, the opposite order may also be possible.

Presently, we develop the concept of an *iterator*.

8.2 The Iterator Interface

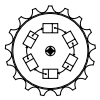
An Iterator is similar to an Enumerator except that the Iterator traverses an associated data structure in a predictable order. Since this is a *behavior* and not necessarily a characteristic of its *interface*, it cannot be controlled or verified by a Java compiler. Instead, we must assume that developers of Iterators will implement and document their structures in a manner consistent with the following interface:


Iterator

```
public interface java.util.Iterator
{
    public abstract boolean hasNext();
    // post: returns true if there is at least one more value to visit

    public abstract java.lang.Object next();
    // pre: hasNext()
    // post: returns the next value to be visited
}
```

While the Iterator is a feature built into the Java language, we will choose to implement our own AbstractIterator class.


Abstract-
Iterator

```
public abstract class AbstractIterator<E>
    implements Enumeration<E>, Iterator<E>, Iterable<E>
{
    public abstract void reset();
    // pre: iterator may be initialized or even amid-traversal
    // post: reset iterator to the beginning of the structure

    public abstract boolean hasNext();
    // post: true iff the iterator has more elements to visit

    public abstract E get();
    // pre: there are more elements to be considered; hasNext()
    // post: returns current value; ie. value next() will return

    public abstract E next();
    // pre: hasNext()
    // post: returns current value, and then increments iterator
```

```
    public void remove()
    // pre: hasNext() is true and get() has not been called
    // post: the value has been removed from the structure
    {
        Assert.fail("Remove not implemented.");
    }

    final public boolean hasMoreElements()
    // post: returns true iff there are more elements
    {
        return hasNext();
    }

    final public E nextElement()
    // pre: hasNext()
    // post: returns the current value and "increments" the iterator
    {
        return next();
    }

    final public Iterator<E> iterator()
    // post: returns this iterator as a subject for a for-loop
    {
        return this;
    }


}
```

This abstract base class not only meets the `Iterator` interface, but also implements the `Enumeration` interface by recasting the `Enumeration` methods in terms of `Iterator` methods. We also provide some important methods that are not part of general `Iterators`: `reset` and `get`. The `reset` method reinitializes the `AbstractIterator` for another traversal. The ability to traverse a structure multiple times can be useful when an algorithm makes multiple passes through a structure to perform a single logical operation. The same functionality can be achieved by constructing a new `AbstractIterator` between passes. The `get` method of the `AbstractIterator` retrieves a reference to the *current element* of the traversal. The same reference will be returned by the call to `next`. Unlike `next`, however, `get` does not push the traversal forward. This is useful when the current value of an `AbstractIterator` is needed at a point logically distant from the call to `next`.

The use of an `AbstractIterator` leads to the following idiomatic loop for traversing a structure:

HelloWorld

```
public static void main(String args[])
{
    // construct a vector containing two strings:
```

```
    Vector<String> v = new Vector<String>();
    AbstractIterator<String> i;
    v.add("Hello");
    v.add("world!");

    // construct an iterator to view values of v
    for (i = (AbstractIterator<String>)v.iterator(); i.hasNext(); i.next())
    {
        System.out.print(i.get()+" ");
    }
    System.out.println();
}
```

The result is the expected `Hello world!`

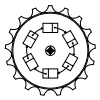
In Java 5 any type that has a method `iterator` that returns an `Iterator<T>` for traversing an object meets the requirements of the `Iterable<T>` interface. These classes can make use of a new form of the `for` loop that simplifies the previous idiom to:

```
Vector<String> v = new Vector<String>();
...
for (String word : v)
{
    System.out.print(word+" ");
}
System.out.println();
```

We will see this form of `for` loop used on many `structure` classes.

8.3 Example: Vector Iterators

For our first example, we design an `Iterator` to traverse a `Vector` called, not surprisingly, a `VectorIterator`. We do not expect the user to construct `VectorIterators` directly—instead the `Vector` hides the construction and returns the new structure as a generic `Iterator`, as was seen in the `HelloWorld` example. Here is the `iterator` method:

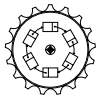


`Vector`

```
public Iterator<E> iterator()
// post: returns an iterator allowing one to
//       view elements of vector
{
    return new VectorIterator<E>(this);
}
```

When a `Vector` constructs an `Iterator`, it provides a reference to *itself* (`this`) as a parameter. This reference is used by the `VectorIterator` to recall which `Vector` it is traversing.

We now consider the interface for a `VectorIterator`:



`Vector-`
`Iterator`

```
class VectorIterator<E> extends AbstractIterator<E>
{
    public VectorIterator(Vector<E> v)
    // post: constructs an initialized iterator associated with v

    public void reset()
    // post: the iterator is reset to the beginning of the traversal

    public boolean hasNext()
    // post: returns true if there is more structure to be traversed

    public E get()
    // pre: traversal has more elements
    // post: returns the current value referenced by the iterator

    public E next()
    // pre: traversal has more elements
    // post: increments the iterated traversal
}
```

As is usually the case, the nonconstructor methods of `VectorIterator` exactly match those required by the `Iterator` interface. Here is how the `VectorIterator` is constructed and initialized:

```
protected Vector<E> theVector;
protected int current;

public VectorIterator(Vector<E> v)
// post: constructs an initialized iterator associated with v
{
    theVector = v;
    reset();
}

public void reset()
// post: the iterator is reset to the beginning of the traversal
{
    current = 0;
}
```

The constructor saves a reference to the associated `Vector` and calls `reset`. This logically attaches the `Iterator` to the `Vector` and makes the first element (if one exists) current. Calling the `reset` method allows us to place all the resetting code in one location.

To see if the traversal is finished, we invoke `hasNext`:

```
public boolean hasNext()
// post: returns true if there is more structure to be traversed
{
    return current < theVector.size();
}
```

This routine simply checks to see if the current index is valid. If the index is less than the size of the `Vector`, then it can be used to retrieve a current element from the `Vector`. The two value-returning methods are `get` and `next`:

```
public E get()
// pre: traversal has more elements
// post: returns the current value referenced by the iterator
{
    return theVector.get(current);
}

public E next()
// pre: traversal has more elements
// post: increments the iterated traversal
{
    return theVector.get(current++);
}
```

The `get` method simply returns the current element. It may be called arbitrarily many times without pushing the traversal along. The `next` method, on the other hand, returns the same reference, but only after having incremented `current`. The next value in the `Vector` (again, if there is one) becomes the current value.

Since all the `Iterator` methods have been implemented, Java will allow a `VectorIterator` to be used anywhere an `Iterator` is required. In particular, it can now be returned from the `iterator` method of the `Vector` class.

Observe that while the user cannot directly construct a `VectorIterator` (it is a nonpublic class), the `Vector` can construct one on the user's behalf. This allows measured control over the agents that access data within the `Vector`. Also, an `Iterator` is a Java interface. It is not possible to directly construct an `Iterator`. We can, however, construct any class that implements the `Iterator` interface and use that as we would any instance of an `Iterator`.

Since an `AbstractIterator` implements the `Enumeration` interface, we may use the value returned by `Vector`'s `iterator` method as an `Enumeration` to access the data contained within the `Vector`. Of course, treating the `VectorIterator` as an `Enumeration` makes it difficult to call the `AbstractIterator` methods `reset` and `get`.

8.4 Example: Rethinking Generators

In Section 7.2 we discussed the construction of a class of objects that generated numeric values. These `Generator` objects are very similar to `AbstractIterators`—they have `next`, `get`, and `reset` methods. They lack, however, a `hasNext` method, mainly because of a lack of foresight, and because many sequences of integers are infinite—their `hasNext` would, essentially, always return `true`.

`Generators` are different from `Iterators` in another important way: `Generators` return the `int` type, while `Iterators` return `Objects`. Because of this,

the `Iterator` interface is more general. Any `Object`, including `Integer` values, may be returned from an `Iterator`.

In this section we experiment with the construction of a numeric iterator—a `Generator`-like class that meets the `Iterator` interface. In particular, we are interested in constructing an `Iterator` that generates prime factors of a specific integer. The `PFIterator` accepts the integer to be factored as the sole parameter on the constructor:

PFGenerator

```
import structure5.AbstractIterator;
public class PFGenerator extends AbstractIterator<Integer>
{
    // the original number to be factored
    protected int base;

    public PFGenerator(int value)
    // post: an iterator is constructed that factors numbers
    {
        base = value;
        reset();
    }
}
```

The process of determining the prime factor involves reducing the number by a factor. Initially, the factor `f` starts at 2. It remains 2 as long as the reduced value is even. At that point, all the prime factors of 2 have been determined, and we next try 3. This process continues until the reduced value becomes 1.

Because we reduce the number at each step, we must keep a copy of the original value to support the `reset` method. When the iterator is reset, the original number is restored, and the current prime factor is set to 2.

```
// base, reduced by the prime factors discovered
protected int n;
// the current prime factor
protected int f;

public void reset()
// post: the iterator is reset to factoring the original value
{
    n = base;
    // initial guess at prime factor
    f = 2;
}
```

If, at any point, the number `n` has not been reduced to 1, prime factors remain undiscovered. When we need to find the current prime factor, we first check to see if `f` divides `n`—if it does, then `f` is a factor. If it does not, we simply increase `f` until it divides `n`. The `next` method is responsible for reducing `n` by a factor of `f`.

```
public boolean hasNext()
// post: returns true iff there are more prime factors to be considered
{
    return f <= n;          // there is a factor <= n
}

public Integer next()
// post: returns the current prime factor and "increments" the iterator
{
    Integer result = get();  // factor to return
    n /= f;                  // reduce n by factor
    return result;
}

public Integer get()
// pre: hasNext()
// post: returns the current prime factor
{
    // make sure f is a factor of n
    while (f <= n && n%f != 0) f++;
    return f;
}
```

We can now write a program that uses the iterator to print out the prime factors of the values presented on the command line of the Java program as it is run:

```
public static void main(String[]args)
{
    // for each of the command line arguments
    for (int i = 0; i < args.length; i++)
    {
        // determine the value
        int n = Integer.parseInt(args[i]);
        PFGenerator g = new PFGenerator(n);
        System.out.print(n+": ");
        // and print the prime factors of n
        while (g.hasNext()) System.out.print(g.next()+" ");
        System.out.println();
    }
}
```

For those programmers that prefer to use the `hasMoreElements` and `nextElement` methods of the `Enumeration` interface, those methods are automatically provided by the `AbstractIterator` base class, which `PFGenerator` extends.

Exercise 8.1 *The $3n + 1$ sequence is computed in the following manner. Given a seed n, the next element of the sequence is $3n + 1$ if n is odd, or $n/2$ if n is even. This sequence of values stops whenever a 1 is encountered; this happens for all*

seeds ever tested. Write an `Iterator` *that, given a seed, generates the sequence of values that ends with* 1.

8.5 Example: Filtering Iterators

We now consider the construction of a *filtering iterator*. Instead of traversing structures, a filtering iterator traverses another iterator! As an example, we construct an iterator that returns the unique values of a structure.

Before we consider the implementation, we demonstrate its use with a simple example. In the following code, suppose that `data` is a `Vector` of `Strings`, some of which may be duplicates. For example, the `Vector` could represent the text of the Gettysburg Address. The `iterator` method of `data` is used to construct a `VectorIterator`. This is, in turn, used as a parameter to the construction of a `UniqueFilter`. Once constructed, the filter can be used as a standard `Iterator`, but it only returns the first instance of each `String` appearing in the `Vector`:

UniqueFilter

```
Vector<String> data = new Vector<String>(1000);
  ...
AbstractIterator<String> dataIterator =
    (AbstractIterator<String>)data.iterator();
AbstractIterator<String> ui = new UniqueFilter(dataIterator);
int count=0;

for (ui.reset(); ui.hasNext(); ui.next())
{
    System.out.print(ui.get()+" ");
    if (++count%7==0) System.out.println();
}
System.out.println();
```

The result of the program, when run on the Gettysburg Address, is the following output, which helps increase the vocabulary of this textbook by nearly 139 words:

```
four score and seven years ago our
fathers brought forth on this continent a
new nation conceived in liberty dedicated to
the proposition that all men are created
equal now we engaged great civil war
testing whether or any so can long
endure met battlefield of have come dedicate
portion field as final resting place for
those who here gave their lives might
live it is altogether fitting proper should
do but larger sense cannot consecrate hallow
ground brave living dead struggled consecrated far
```

```
above poor power add detract world will
little note nor remember what say itcan
never forget they did us rather be
unfinished work which fought thus nobly advanced
task remaining before from these honored take
increased devotion cause last full measure highly
resolve shall not died vain under God
birth freedom government people by perish earth
```

Fans of compact writing will find this unique.

The `UniqueFilter` provides the same interface as other iterators. Its constructor, however, takes a "base" `Iterator` as its parameter:

```
protected AbstractIterator<T> base; // slave iterator
protected List<T> observed;  // list of previous values

public UniqueFilter(AbstractIterator<T> baseIterator)
// pre: baseIterator is a non-null iterator
// post: constructs unique-value filter
//       host iterator is reset
{
    base = baseIterator;
    reset();
}

public void reset()
// post: master and base iterators are reset
{
    base.reset();
    observed = new SinglyLinkedList<T>();
}
```

When the filter is reset using the `reset` method, the base iterator is reset as well. We then construct an empty `List` of words previously observed. As the filter progresses, words encountered are incorporated into the `observed` list.

The current value is fetched by the `get` method. It just passes the request along to the base iterator. A similar technique is used with the `hasNext` method:

```
public boolean hasNext()
// post: returns true if there are more values available
//       from base stream
{
    return base.hasNext();
}

public T get()
// pre: traversal has more elements
// post: returns the current value referenced by the iterator
{
```

```
    return base.get();
}
```

Finally, the substance of the iterator is found in the remaining method, `next`:

```
public T next()
// pre: traversal has more elements
// post: returns current value and increments the iterator
{
    T current = base.next();
    // record observation of current value
    observed.add(current);
    // now seek next new value
    while (base.hasNext())
    {
        T possible = base.get();
        if (!observed.contains(possible))
        {   // new value found! leave
            break;
        } else {
            // old value, continue
            base.next();
        }
    }
    return current;
}
```

Because this routine can only be called if there is a current value, we record the current value in the `observed` list. The method then increments the base iterator until a new, previously unobserved value is produced, or the base iterator runs dry.

Some subtle details are worth noting here. First, while we have used a `VectorIterator` on a `Vector` of `String`s, the `UniqueFilter` can be applied, as is, to any type of iterator and can deliver any type of value. All that is required is that the base type support the `equals` method. Secondly, as the filter iterator progresses, it forces the base iterator to progress, too. Because of this, two filters are usually not applied to the same base iterator, and the base iterator should never be modified while the filter is running.

8.6 Conclusions

We have seen that data structures can sometimes be used to control the way programs focus on and access data. This is made very explicit with Java's `Enumeration` construct that facilitates visiting all the elements of a structure.

When we wish to traverse the elements of a data structure in a predetermined order, we use an `Iterator`. The `Iterator` provides access to the elements of a structure using an interface that is similar to that of an `Enumeration`.

The abstract base class `AbstractIterator` implements both the `Iterator` and `Enumeration` interfaces, and provides two new methods—`get` and `reset`—as well. We have also seen that there are weaknesses in the concept of both of these constructs, because they surrender some of the data hiding and access controls that are provided by the associated structure. Careful use of these controlling structures, however, can yield useful tools to make traversal of structures simpler.

Self Check Problems

Solutions to these problems begin on page 445.

8.1 Suppose `e` is an `Enumeration` over some data structure. Write a loop using `e` to print all the values of the data structure.

8.2 Suppose `i` is an `Iterator` over some data structure. Write a loop using `i` to print all the values of the data structure.

8.3 Suppose that `v` is a `Vector` of `Integer` values. Write a loop that will use an `Iterator` to print those `Integer` values that are even.

8.4 It is possible to write down the integers 1 through 15 in an order such that each adjacent pair of integers sums to a perfect square. Write a loop that prints `Perfect!` only if the adjacent `Integer` values generated by the `Iterator` `g` sum to perfect squares. (You needn't verify the number or range of values.)

Problems

Solutions to the odd-numbered problems begin on page 467.

8.1 Since the `get` method is available to the `AbstractIterator`, the `next` method does not appear to need to return a value. Why does our implementation return the value?

8.2 Write an `Iterator` that works on `Strings`. Each value returned should be an object of type `Character`.

8.3 Write an `Iterator` that returns a stream of `Integers` that are prime. How close is it to the `Generator` implementation of Section 7.2?

8.4 Write a filtering iterator, `ReverseIterator`, that reverses the stream of values produced by another `Iterator`. You may assume that the base `Iterator` will eventually have no more elements, but you may not bound the number.

8.5 Write a filtering iterator, `OrderedIterator`, that sorts the stream of values produced by another `Iterator`. You may assume that the base `Iterator` will eventually have no more elements, but you may not bound the number.

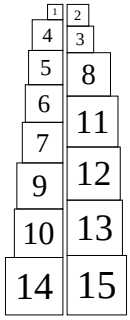
8.6 Write a filtering iterator, `ShuffleIterator`, that shuffles the stream of values produced by another `Iterator`. You may assume that the base `Iterator` will eventually have no more elements, but you may not bound the number.

8.7 Write a filtering iterator that takes a base iterator and an `Object` (called `predicate`) with a static `select` method defined. This iterator passes along only those values that generate `true` when passed to the `select` method of the `predicate Object`.

8.7 Laboratory: The Two-Towers Problem

Objective. To investigate a difficult problem using `Iterators`.

Discussion. Suppose that we are given n uniquely sized cubic blocks and that each block has a face area between 1 and n. Build two towers by stacking these blocks. How close can we get the heights of the two towers? The following two towers built by stacking 15 blocks, for example, differ in height by only 129 millions of an inch (each unit is one-tenth of an inch):



Still, this stacking is only the *second-best* solution! To find the best stacking, we could consider all the possible configurations.

We *do* know one thing: the total height of the two towers is computed by summing the heights of all the blocks:

$$h = \sum_{i=1}^{n} \sqrt{i}$$

If we consider all the *subsets* of the n blocks, we can think of the subset as the set of blocks that make up, say, the left tower. We need only keep track of that subset that comes closest to $h/2$ without exceeding it.

In this lab, we will represent a set of n distinct objects by a `Vector`, and we will construct an `Iterator` that returns each of the 2^n subsets.

Procedure. The trick to understanding how to generate a subset of n values from a `Vector` is to first consider how to generate a subset of indices of elements from 0 to $n-1$. Once this simpler problem is solved, we can use the indices to help us build a `Vector` (or subset) of values identified by the indices.

There are exactly 2^n subsets of values 0 to $n-1$. We can see this by imagining that a coin is tossed n times—once for each value—and the value is added to the subset if the coin flip shows a head. Since there are $2 \times 2 \times \cdots \times 2 = 2^n$ different sequences of coin tosses, there are 2^n different sets.

We can also think of the coin tosses as determining the place values for n different digits in a binary number. The 2^n different sequences generate binary numbers in the range 0 through $2^n - 1$. Given this, we can see a line of attack:

count from 0 to $2^n - 1$ and use the binary digits (*bits*) of the number to determine which of the original values of the `Vector` are to be included in a subset.

Computer scientists work with binary numbers frequently, so there are a number of useful things to remember:

- An `int` type is represented by 32 bits. A `long` is represented by 64 bits. For maximum flexibility, it would be useful to use `long` integers to represent sets of up to 64 elements.
- The *arithmetic shift* operator (`<<`) can be used to quickly compute powers of 2. The value 2^i can be computed by shifting a unit bit (1) i places to the left. In Java we write this `1<<i`. This works only for nonnegative, integral powers. (For long integers, use `1L<<i`.)
- The *bitwise and* of two integers can be used to determine the value of a single bit in a number's binary representation. To retrieve bit `i` of an integer `m` we need only compute `m & (1<<i)`.

Armed with this information, the process of generating subsets is fairly straightforward. One line of attack is the following:

1. Construct a new extension to the `AbstractIterator` class. (By extending the `AbstractIterator` we support both the `Iterator` and `Enumeration` interfaces.) This new class should have a constructor that takes a `Vector` as its sole argument. Subsets of this `Vector` will be returned as the `Iterator` progresses.
2. Internally, a `long` value is used to represent the current subset. This value increases from 0 (the empty set) to $2^n - 1$ (the entire set of values) as the `Iterator` progresses. Write a `reset` method that resets the subset counter to 0.
3. Write a `hasNext` method that returns `true` if the current value is a reasonable representation of a subset.
4. Write a `get` method that returns a new `Vector` of values that are part of the current subset. If bit `i` of the current counter is 1, element `i` of the `Vector` is included in the resulting subset `Vector`.
5. Write a `next` method. Remember it returns the current subset *before* incrementing the counter.
6. For an `Iterator` you would normally have to write a `remove` method. If you extend the `AbstractIterator` class, this method is provided and will do nothing (this is reasonable).

You can now test your new `SubsetIterator` by having it print all the subsets of a `Vector` of values. Remember to keep the `Vector` small. If the original values are all distinct, the subsets should all have different values.

To solve the two-towers problem, write a `main` method that inserts the values $\sqrt{1}$, $\sqrt{2}$,...,$\sqrt{n}$ as `Double` objects into a `Vector`. A `SubsetIterator` is then used to construct 2^n subsets of these values. The values of each subset are summed, and the sum that comes closest to, but does not exceed, the value $h/2$ is remembered. After all the subsets have been considered, print the best solution.

Thought Questions. Consider the following questions as you complete the lab:

1. What is the best solution to the 15-block problem?
2. This method of exhaustively checking the subsets of blocks will not work for very large problems. Consider, for example, the problem with 50 blocks: there are 2^{50} different subsets. One approach is to repeatedly pick and evaluate random subsets of blocks (stop the computation after 1 second of elapsed time, printing the best subset found). How would you implement `randomSubset`, a new `SubsetIterator` method that returns a random subset?

Notes: