Module 3

Searching Algorithms

Objectives:

1. Understand how linear searches and binary searches work, using simple arrays of items.
2. Understand the running time differences between linear and binary searches.
3. Be aware of the interpolation search variation of the binary search algorithm.
4. Know how search strategies can be adapted to deal with sorted arrays.

Textbook:

Java Interlude 1 (Generics)

Java Interlude 5 (More about Generics)

Java Interlude 8 (Generics Once Again)

Chapter 19, sections 19.1 - 19.25.

Labs:

Do the Binary Search Lab.

Assignment:

Complete the assignment for Module 3.

Quiz:

Complete the quiz for Module 3.

Comparing Objects Using Java's Comparable Interface

All of the searching and sorting algorithms discussed in this course are comparison-based algorithms. This simply means the algorithms compare elements when searching for a given element or sorting a list of elements.

If the elements are numbers, comparisons can be made very easily using the standard numerical comparison operators: <, <=, ==, >=, >. But what if the elements happen to be objects instead of numbers? For example, if you have a list of Rectangle objects, how do you know when you have found a particular Rectangle in the list? How would you sort a list of Rectangles?

You can't determine whether 2 Rectangles are the same by using the "==" equality operator. With objects, the "==" operator returns true if two variables refer to the same object in memory. For example, suppose 2 Rectangles are considered equal if they have the same dimensions. Consider the following code:

```
Rectangle rect1 = new Rectangle(10, 5);
Rectangle rect2 = new Rectangle(12, 7);
Rectangle rect3 = new Rectangle(10, 5);
Rectangle temp;

// prints false; rect1 and rect2 reference different Rectangle objects in memory
System.out.println(rect1 == rect2);

temp = rect2;

// prints true; temp and rect2 both reference the same Rectangle object in memory
System.out.println(temp == rect2);

// prints false; although rect3 and rect1 have same dimensions and would appear to
   be equal, they are two separate objects in memory
System.out.println(rect3 == rect1);
```

We would like for the third print statement to print `true`, since `rect1` and `rect3` have the same dimensions, but we have no operator that will do this. However, we can make use of the fact that every class we create in Java is implicitly derived from the base class `Object`, and thus inherits all methods in that class. One of those methods is the `equals` method. You have probably used it in the past to compare `String` objects. We can override the `equals` method for the `Rectangle` class and implement it so that it returns `true` if the dimensions of 2 Rectangles are the same, and false if the dimensions are different. Thus, the expression `rect1.equals(rect3)` would return `true`, but the expression `rect1.equals(rect2)` would return `false`.

Overriding the `equals` method will let us test equality, but for sorting we also need to be able to do less-than and greater-than comparisons. There are no methods we can override to do this, but Java does provide a way—the `Comparable` interface. The `Comparable` interface contains a single method:

```
public int compareTo(Object o) {}
```

This method is used similarly to the `equals` method, for example:

```
int comparison = x.compareTo(y).
```

The `compareTo` method returns a value `r`, such that:

- `r < 0` if `x < y`
- `r == 0` if `x == y`
- `r > 0` if `x > y`

For the equality condition, the Java API recommends, but does not strictly require, that `(x.compareTo(y) == 0) == (x.equals(y))`. Further, the inequality conditions should be transitive; i.e., `x.compareTo(y) == 0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

The exact value that is returned is not consistent between data types, but the above rule still holds. As an example, when Strings are compared, the value that is returned is the difference of the ASCII codes of the two letters. For instance, the letter "A" has ASCII code 65, and the letter "D" has ASCII code 68. Thus,

- `"A".compareTo("D")` returns -3, since 65 - 68 == -3.
- `"A".compareTo("A")` returns 0, since 65 - 65 == 0.
- `"D".compareTo("A")` returns 3, since 68 - 65 = 3.

(Incidentally, the above is valid Java syntax. You can type `"A".compareTo("D")`, and it will execute, giving the result above. The literal string values are automatically converted into `String` objects.).

So, if we want to do less-than or greater-than comparisons on object instances from a particular class, we would simply have that class implement the `Comparable` interface, as shown below:

```
public class Rectangle implements Comparable {}
```

Recall that when a class implements an interface it must provide an implementation for every method in that interface. Thus, in our `Rectangle` example we would have to add a `compareTo` method to the class definition and implement it according to however we choose to define less-than, greater-than and equals as they apply to Rectangles.

Using the `Comparable` interface can be very powerful. If a variable of type `Comparable` is declared, a reference to any class that implements the `Comparable` interface can be stored in that variable. This allows for the creation of generalized searching and sorting algorithms that can be used with virtually any object.

Sequential Search

A sequential search applied to an array or list simply begins at one end of the list and proceeds towards the other end, one item at a time. For a sequential search, the ordering of the elements in the array or list is unimportant. Whether the elements are sorted or not, a sequential search will return the position of the first occurrence of the specified search item, as determined by the starting point and direction of the search. Figure 3.1 shows a typical sequential search algorithm that searches an array of integers. If the search value is present in the array, the index of the first occurrence of the search value is returned. If the search value is not present in the array, a value of -1 is returned to indicate an unsuccessful search.

An analysis of the complexity of the sequential search algorithm in Figure 3.1, in terms of the number of statement executions, is shown in Figure 3.2. This analysis looks at the worst case, best case, and average case scenarios. In the worst case scenario, the search will be unsuccessful. The entire array will be searched, but the search value will not be found. Just as a point of distinction, if the search value happens to be the last item in the array, the result is a slightly better than worst case scenario. The analysis of statement executions shows that in the worst case, this particular sequential search algorithm is O(4N + 3), which is of order O(N). This makes sense, because traversing every element in the array amounts to a linear operation.

The best case scenario for the sequential search occurs when the search value is the first item that is examined. Since we're talking about the lower bound of the algorithm's performance, the complexity of the best case scenario is expressed using Big-Omega notation. As can be seen from the analysis in Figure 3.2, the algorithm's lower bound on complexity is O(4), which is of order O(1). The size of the problem is irrelevant here. Since the search value is always at the first position examined, there will never be more than one comparison made.

For a search algorithm, neither the worst case nor the best case scenarios are particularly interesting, since their frequency of occurrence will depend on the input data. For example, if you have an array of

integers, {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, and you search for the value 10, you are near the worst case scenario. If you happen to be searching for the value 1, you have the best case scenario. However, if the input array is instead {4, 2, 6, 3, 10, 1, 5, 9, 8, 7}, different values now represent the best and worst cases. For searches, we tend to be more interested in the average time it is likely to take to complete a search. Statistically, the median position of the array (not the median value) represents an average search, meaning that on average, half the array will need to be searched to find a given value. We would therefore predict the order of complexity of an average sequential search would be approximately O(N/2), but the analysis in Figure 3.2 would seem to contradict this prediction. How can this be? The explanation is that when we are talking about searching or sorting algorithms, the convention is to think only in terms of the number of comparisons that must be made, not the total number of statement executions, since the comparisons are what determine whether or not the search value has been found. If we look only at the number of comparisons, represented by the statement, `if (array[i] == searchValue)`, we see that indeed, the average running time is approximately O(N/2). Note that the actual number of statement executions in the average case varies slightly depending on whether the array contains an even number of elements or an odd number of elements. The variation is due to the fact that with an even number of elements there is no true median position, since N/2 yields a fractional value. The positions on either side of this fractional value therefore equally represent the median position, but in a sequential search, getting to one of these two positions will require an extra comparison, and extra increment of the loop variable, and an extra execution of the loop termination condition. For example, if you have the array, {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, positions 4 and 5 (corresponding to the values 5 and 6, respectively) equally represent the median position in the array.

We can summarize the analysis of the complexity of the sequential search algorithm as being O(4N + 3) and Ω(4). It may be tempting to use Big-Theta notation to describe the average case, but for the sequential search algorithm that would be incorrect. In order to use Big-Theta notation, the upper bound would have to be of the same order as the lower bound, which is not true for a sequential search, since the order of the upper bound is O(N) and the order of the lower bound is O(1). The upper and lower bounds must be of the same order to use Big-Theta notation. Therefore, the average running time would be expressed using Big-Oh notation.

```
public int sequentialSearch(int[] array, int searchValue)
{
    for (int i = 0; i < array.length; i++)
    {
        if (array[i] == searchValue)
        {
            return i;
        }
    }

    return -1;
}
```

Figure 3.1. An iterative sequential search algorithm.

Statement	**Worst Case (unsuccessful search)**	**Best Case (successful search; search value is first item examined)**	**Average Case (successful search; search value is in the middle of the array)**		
			even # of elements		**odd # of elements**
`int i = 0;`	1	1	1	1	1
`for ( ; ; ) // as jump to loop start`	N	0	N/2 - 1	N/2	N/2 - 1
`i < array.length`	N + 1	1	N/2	N/2 + 1	N/2
`if (array[i] == searchValue)`	N	1	***N/2***	***N/2 + 1***	***N/2***
`return i;`	0	1	1	1	1
`i++`	N	0	N/2 - 1	N/2	N/2 - 1
`return -1;`	1	0	0	0	0
Complexity:	**4N + 3**	**4**	**2N**	**2N + 4**	**2N**

Figure 3.2. Analysis of the running time for the sequential search algorithm shown in Figure 3.1.

Binary Search

The time required to perform a sequential search can be prohibitive when the number of elements to be searched is large. If the items in an array are sorted, we can use the divide-and-conquer approach to greatly shorten the search time by repeatedly eliminating half of the array from consideration. It is important to realize that in contrast to the sequential search, the binary search algorithm only works on ordered lists or arrays. The binary search algorithm presented in the book uses recursion, but an iterative version can be written quite easily. One example of an iterative binary search algorithm is shown in Figure 3.3. Since we are repeatedly halving the array, and eliminating half the list from consideration each time, the running time for the binary search algorithm is on the order of $O(\log_2 N)$. Figure 3.4 shows the detailed analysis.

As you can see from this analysis, there are subtle differences in performance for the worst case, depending on whether the array has an even vs. an odd number of elements. In the best case, the search value is the first item that is compared, which will always be the item located at index `(low + high)/2`. In summary, the performance of the algorithm in Figure 3.3 is $O(5*\log_2 N + 11.5)$ and $\Omega(8)$, precisely, or $O(\log_2 N)$ and $\Omega(1)$, using order of magnitude estimation.

With search algorithms, and as we will see later, sorting algorithms, performance is often measured only in terms of the number of element comparisons that must be made. The time required for updating loop variables, etc., is largely ignored. It is difficult to see why when looking at an algorithm that uses only integers for data, since integer comparison are simple operations. If we consider more complex objects, however, determining equality may be considerably more complex than comparing two integer values. For example, when comparing two files for equality at the byte level, in the worst case it is necessary to perform individual comparisons on a number of bytes equal to the length of the smallest file. The worst case would be when the two files are equal except for the last byte compared. If the lengths of the files are on the order of megabytes, gigabytes, or higher, a single comparison between two objects will involve millions or billions of comparisons, respectively. Performances based on the numbers of comparisons only, are also shown in Figure 3.4.

A few other interesting points to mention about the algorithm are:

- The algorithm shown in Figure 3.3 can only work when the elements sorted in ascending order, but it can be easily modified to work with elements sorted in descending order.
- If there are multiple occurrences of the search value in the array, the only guarantee offered by binary search is that the index of one of those occurrences will be returned.
- Binary search will always perform better than sequential search unless the array being searched is trivially small (around 5 elements).
- The analyses shown here assume the entire array being searched resides in main memory. If external disk accesses are required, the binary search can perform much worse than a sequential search, due to the random access nature imposed by the repeated halving; i.e., the next element to be compared may be located in a different sector on the hard drive.

It is possible to improve the worst case performance of the algorithm in Figure 3.3. Since there are two comparison statements inside the while loop, in some cases both comparisons must be performed during the same loop iteration. When the two array pointers are equal, the element they reference is either the search value or it isn't. We can adjust the algorithm in Figure 3.3 by changing the loop termination condition to move the comparison for equality outside the loop. This is shown in the algorithm in Figure 3.5, which is implemented to allow comparisons between any two objects that implement the `Comparable` interface. For the sake of simplicity, I am treating the calls to the `compareTo` method as simple statements, since the implementation details of the method will vary depending on the object. Figure 3.6 shows the detailed analysis of the algorithm's complexity, and shows that the performance in the worst case scenario is improved uniformly to $O(5*\log_2 N + 6)$, with a reduction of 1-2 comparisons. This is slightly faster than the algorithm shown in Figure 3.3. However, the improvement in the worst case scenario comes with a cost. The running time for the algorithm in Figure 3.3 can vary from case to case (from 1 to even 18 statement executions in the non-best case for an array of 16 items). The algorithm in Figure 3.5

involves $5*\log_2 N + 6$ statement executions regardless of the case—including the best case scenario. What this means is that the algorithm in Figure 3.3 will have a better average running time than the algorithm in Figure 3.5.

So, why would we be interested in writing an algorithm that performs better in the worst case, but worse on average? If we have a problem that we know is the worst case scenario, or if we know the worst case scenario is likely to appear with an unusually high frequency, it may be worth sacrificing better average performance for better worst case performance. We can have a library of algorithms, all of which perform the same function but with differing efficiencies, and if we have a way to efficiently screen problems beforehand, we can selectively use the algorithm that is best suited to the problem at hand. Granted, the difference in performance between the two binary search algorithms I've presented is negligible, but for other algorithms that perform more complex functions, the difference may be significantly more pronounced.

```
public int binarySearch(int[] array, int searchValue)
{
    int low = 0;
    int high = array.length - 1;
    int mid = 0;

    while (low <= high)
    {
        mid = (low + high) / 2;

        if (array[mid] < searchValue)
        {
            low = mid + 1;
        }
        else if (array[mid] > searchValue)
        {
            high = mid - 1;
        }
        else
        {
            return mid;
        }
    }

    return -1;
}
```

Figure 3.3. A straightforward, iterative, binary search algorithm.

Statement	Worst Case (always an unsuccessful search)		Best Case(successful search; search value is at position (low + high)/2)
	even # of elements search value > last item in array	odd # of elements search value < first item in array	
`int low = 0;`	1	1	1
`int high = array.length - 1;`	1	1	1
`int mid = 0;`	1	1	1
`while () // as jump to loop start`	$\log_2 N + 1$	$\log_2 N$	0
`low <= high`	$\log_2 N + 1 + 1$	$\log_2 N + 1$	1
`mid = (low + high) / 2`	$\log_2 N + 1$	$\log_2 N$	1
`if (array[mid] < searchValue)`	$\log_2 N + 1$	$\log_2 N$	1
`low = mid + 1;`	$\log_2 N + 1$	0	0
`else if (array[mid] > searchValue)`	0	$\log_2 N$	1
`high = mid - 1;`	0	$\log_2 N$	0
`return mid;`	0	0	1
`return -1;`	1	1	0
Complexity:	**$5\log_2 N + 10$**	**$6\log_2 N + 5$**	**8**
Complexity (element comparisons only):	**$\log_2 N + 1$**	**$2\log_2 N$**	**2**

Figure 3.4. Analysis of the running time for the binary search algorithm shown in Figure 3.3. Values were corroborated using an even-numbered array consisting of `[1, 5, 10, 12, 15, 17, 20, 23]` and `searchValue == 25` for the even-numbered worst case. For the odd-numbered worst case, the array `[1, 5, 10, 12, 15, 17, 20]` was used, with `searchValue == 0`. Slightly better performances were observed when `searchValue == 0` for the even-numbered array, and for `searchValue == 25` for the odd-numbered array.

```
public int search(Comparable[] objArray, Comparable searchObj)
{
    int low = 0;
    int high = objArray.length - 1;
    int mid = 0;

    while (low < high)
    {
        mid = (low + high) / 2;

        if (objArray[mid].compareTo(searchObj) < 0) // searchObj > objArray[mid]
        {
            low = mid + 1;
        }
        else // searchObj <= objArray[mid]
        {
            high = mid;
        }
    }

    if (objArray[low].compareTo(searchObj) == 0)
    {
        return low;
    }

    return -1;
}
```

Figure 3.5. An alternative, iterative, binary search algorithm.

Statement	# Statement Executions, Worst Case (always an unsuccessful search) OR Best Case	
	even # of elements search value < first item in array OR search value > last item in array	**odd # of elements search value < first item in array**
`int low = 0;`	1	1
`int high = array.length - 1;`	1	1
`int mid = 0;`	1	1
`while() // jump to loop start`	$\log_2 N$	$\log_2 N$
`low < high`	$\log_2 N + 1$	$\log_2 N + 1$
`mid = (low + high) / 2`	$\log_2 N$	$\log_2 N$
`if (objArray[mid].compareTo(searchObj) < 0)`	$\log_2 N$	$\log_2 N$
`low = mid + 1;`	$\log_2 N$	0
`high = mid;`	0	$\log_2 N$
`if (objArray[low].compareTo(searchObj) == 0)`	1	1
`return low;`	0	0
`return -1;`	1	1
Complexity:	**$5(\log_2 N) + 6$**	**$5(\log_2 N) + 6$**
Complexity (element comparisons only):	**$\log_2 N + 1$**	**$\log_2 N + 1$**

Figure 3.6. Analysis of the running time for the alternative binary search algorithm shown in Figure 3.5. Values were corroborated using an even-numbered array consisting of `[1, 5, 10, 12, 15, 17, 20, 23]` and `searchValue = 25` for the even-numbered worst case. For the odd-numbered worst case, the array `[1, 5, 10, 12, 15, 17, 20]` was used, with `searchValue == 0`. Slightly better performance was observed when `searchValue = 25` for the odd-numbered array. No performance increase was observed for `searchValue = 25` for the even-numbered array.

Interpolation Search

With a binary search, the array is always divided into two equal halves, and the next item to be compared with the search item is at the midpoint. If the search item happens to be very near the front or back of a very large array, it can take even the binary search algorithm a while to get to the right region. What if we could statistically predict where the search item should be located? We should be able to make an educated guess as to where to jump following an unsuccessful comparison.

A variant of the binary search algorithm, known as the interpolation search, does just that. It works similarly to a binary search, except that:

- The array needs to store numerical keys that map to the data items (the parameter `searchObj` would also be a key instead of an actual data item).
- Instead of computing the midpoint, the interpolation search computes the index of the next value to be compared with the search item as follows:

```
next = low + (searchObj - a[low])*(high - low - 1)/(a[high] - a[low])
```

You may also see this formula without the "`- 1`" in the factor, `(high - low - 1)`. The subtraction of 1 has been shown to offer slightly better performance in practice. This algorithm has a bad worst case of O(N), which occurs when the keys are not evenly distributed. Even distribution, in this context, means that the difference between any two consecutive keys k1 and k2 is approximately the same as the difference between any two other consecutive keys k3 and k4. It is easy to spot perfect distribution (k2 - k1 == k4 - k3, for all consecutive key pairs <k1, k2>, and <k3, k4>), and extremely uneven distribution, but for cases between these extremes, some arbitrary judgment is required when labeling them as evenly distributed or unevenly distributed. Some examples are helpful here:

Even distribution:

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- 5, 10, 15, 20, 25, 30, 35, 40
- 2, 4, 6, 9, 12, 14, 16, 19, 21 (not perfectly even distribution, but close)

Uneven distribution:

- 1, 2, 3, 50, 55, 75, 123, 178
- 1, 10, 100, 1000, 10000, 100000
- 23, 89, 125, 290, 509, 1035, 8463

The average running time of interpolation search can be as good as O(log log N), meaning it can be faster than the standard binary search.

In practice, the interpolation search algorithm is most useful when the items are stored on a disk or network device as opposed to the computer's main memory. Since accessing items externally usually tends to be very slow, the time lost doing the extra computations to find the next index is more than offset by the time saved having to access fewer items for comparison.

Secure Coding in Java

This is the first of many sidebars throughout the lectures that will cover selected aspects of secure coding practices in Java. Some of the practices presented in each sidebar will be directly related to the topics of the lecture, while others will be general in nature. These recommendations come from the two sources listed below, and I have included the reference number for each recommendation. The simple numeric references; e.g., 52, are from the Java Coding Guidelines reference, while the alphanumeric references; e.g., EXP03-J, are from the *CERT® Oracle Secure Coding Standard for Java reference*.

1. *The CERT® Oracle Secure Coding Standard for Java* by Fred Long, et al. (Part of the SEI Series in Software Engineering), 1st Edition (2012), Pearson Education, Inc., ISBN# 978-0-321-80395-5
2. *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* by Fred Long, et al. (Part of the SEI Series in Software Engineering), 1st Edition (2014), Pearson Education, Inc., ISBN# 978-0-321-93315-7

EXP03-J. Do not use the equality operators when comparing values of boxed primitives

Recall that when using the equality operators, `==` and `!=`, with primitive values, the values themselves are compared. For example, `7 == 7` would return `true`, and `7 != 7` would return `false`. However, when two objects are compared, these operators compare the reference addresses of the objects. That is, two object references are equal if they point to the same object in memory. This may not be what is desired, since there may be times when it's necessary to ensure equality between two different instances of the same class based on the values of one or more of the class' attributes. In this case, the `equals()` method must be used to determine equality between the objects, and not the primitive equality operators.

The same reasoning applies to boxed primitives. Boxed primitives are primitive data values that have been encapsulated in an appropriate Java wrapper object using Java's autoboxing mechanism. Thus, boxed primitives are really objects that represent primitive data values. For example, the following statement creates two boxed integer values:

```
Integer seven = new Integer(7);
Integer anotherSeven = new Integer(7);
```

In this example, `seven` and `anotherSeven` are two different objects, but they both represent the integer value 7 so on that basis the two objects should be considered equal. Using the primitive equality operators will not give the desired result, thus the `equals()` method should be used instead.

There is another reason to avoid using the equality operators with boxed primitives in Java. During autoboxing in Java, some primitive values are memoized; i.e., cached, for optimization of performance. The primitive Boolean values, true and false, all byte values, char values in the range `\u0000` to `\u007f`, and values of type int or short in the range `–128` to `127` are memoized. Primitives outside these values may or may not be memoized during autoboxing. As a result, using the equality operators can sometimes give the result for boxed primitives that would be expected for unboxed primitives.

While the equality operators should not be used to compare boxed primitives, it is allowable to use the comparison operators, `<`, `>`, `<=`, and `>=`. These operators cause boxed primitives to be automatically unboxed, and thus compared as primitives instead of object references.

EXP04-J. Ensure that autoboxed values have the intended type

Unexpected behavior can result when a non-specific reference type, such as `Number` or `Object`, that causes a boxing conversion tries to convert a value that results from an expression that mixes primitive and numeric types. For example, consider the following code:

```
public class ShortSet
{
      public static void main(String[] args)
      {
            HashSet<Short> s = new HashSet<Short>();

            for (short i = 0; i < 100; i + +)
            {
                  s.add(i);
                  s.remove(i - 1); // tries to remove an Integer
            }

            System.out.println(s.size());
      }
}
```

This code adds the `short` values 0 to 99 to a `HashSet` that has been correctly parameterized to contain `Short` values, which are the boxed form of primitive short types. Immediately after adding a value, an attempt is made to remove the value that was added in the previous iteration of the loop. It seems like this should work, and that after the loop terminates the size of the `HashSet` should be 1 (since the very last value added would not be removed). However, the size of the `HashSet` is displayed as 100. The reason is because the literal value, 1, in the expression that forms the argument to the `remove` method is treated by Java as an `int` value, thus causing a mixture of a `short` value, `i`, and an `int` value, 1. Because Java's numeric promotion rules will cause the expression to be converted to an `int` value, which will then be autoboxed as an `Integer`, the remove method attempts to remove an `Integer` from a `HashSet` that was defined to contain only `Short` values.

The data type mismatch will cause the `remove` method to do nothing, thus no items are ever removed.

To avoid this problem, in this particular example, the expression inside the `remove` method should be cast to the approptiate type:

```
s.remove((short)(i - 1)); // will now try to remove an Short
```

OBJ03-J. Do not mix generic with non-generic raw types in new code

When Java rolled out generics in Java 1.5, there was already a lot of legacy Java code in operation that wasn't written to handle parameterized types. In order to preserve compatibility, the Java compiler allows generic and non-generic types to be mixed. As long as one is very careful, this can be done without causing problems, and as a practice it is acceptable to mix generic and non-generic types in legacy code. There are two main problems that can occur when generic and non-generic types are mixed.

1. When generic types are declared without specifying a type — which is allowed in Java, an exception thrown at one statement may in fact have been caused by a problem in a statement that was executed earlier. Methods that return values indicating the success or failure of an operation, rather than throwing exceptions, can appear to have succeeded even though they failed. Methods that use or return specific types, however, may throw exceptions because of errors in casting from one class to another. Depending on the order in which methods are called, and where those methods lie in the code, allowing the mixing of generic and non-generic types can result in debugging nightmares.

2. The other problem that can occur when mixing generic and non-generic types is known as "heap pollution". Heap pollution basically means that there are variables stored in the heap that do not point to the correct reference types. One way this can happen is when a parameterized class is instantiated without specifying a type for the parameter, and later an attempt is made to treat that object as though it were a parameterized type. For example:

```
List l = new ArrayList();
List<String> ls = l;
```

Although the Java compiler can generate unchecked warnings when this occurs, it is not wise to rely on them, especially when system security is of major importance. Additionally, these warnings can be suppressed by including the appropriate annotation in the code.

The ideal solution is to never mix generic and non-generic types, and this is the recommended practice when legacy code is not involved. If legacy code that does not support generics must be used, it is incumbent upon the programmer to ensure there will be no type mismatches during execution. The most commonly parameterized types are data structures, and Java provides methods in the `Collections` class for dynamically returning typesafe views of the data structures in the `java.util package`. These methods all follow the name format, `checked`<data structure type>; e.g., `checkedList(List<E> list, Class<E> type)`.

MET08-J. Ensure objects that are equated are equatable

Often, user-defined classes in Java must override the `equals()` method in order to allow for comparing the equality of two instances of the same user-defined class. Determining equality can become complicated if composition or inheritance are used to derive subclasses of a user-defined class. As long as the equality of the subclass depends only on the state of the parent class, the subclass can simply inherit the parent's `equals()` method, and nothing further need be done. If the equality of the subclass does depend on the new attributes of the subclass, the subclass must override the `equals()` method it inherits from the parent class. When overriding the `equals()`

method in any class, it is important to ensure the contract of the `equals()` method is satisfied. This contract has 5 points, as taken directly from the Java API documentation:

1. It is reflexive: For any reference value `x`, `x.equals(x)` must return `true`.

2. It is symmetric: For any reference values `x` and `y`, `x.equals(y)` must return `true` if and only if `y.equals( x)` returns `true`.

3. It is transitive: For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.

4. It is consistent: For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals()` comparisons on the object is modified.

5. It correctly handles comparisons with the `null` reference: For any non-`null` reference value `x`, `x.equals(null)` must return `false`.

When overriding the `equals()` method, all 5 points of the contract must be satisfied. When extending a class and adding attributes to the subclass that are important for comparing equality, it is impossible to satisfy all 5 points. Usually the transitive property will be unsatisfiable since the `equals()` method for two non-equal instances of a subclass must return `false`, yet must return `true` when either subclass instance is treated as though it were an instance of the parent class. One solution to this is to use composition to have the subclass maintain a private instance of its parent class, and compare parent classes by comparing those private instances.

If it is known with certainty that incompatible types resulting from composition or inheritance will never be compared for equality, it is acceptable to allow the `equals()` contract to be unsatisfied. However, care must be taken to ensure that the appropriate changes are made if, at a later time, the code is modified to allow such equality comparisons, as shown in the following example.

```
class XCard
{
        private String type;
        private Card card;     // Composition

        public XCard(int number, String type)
        {
                card = new Card(number);
                this.type = type;
        }

        public Card viewCard()
        {
                return card;
        }

        public boolean equals(Object o)
        {
                if (!(o instanceof XCard))
                {
                        return false;
                }

                XCard cp = (XCard) o;
                return cp.card.equals(card) && cp.type.equals(type);
        }

        public int hashCode()
        {
                /* ... */
        }
}
```

MET09-J. Classes that define an equals() method must also define a hashCode() method

In addition to the 5 points of the contract for the `equals()` method listed above, the Java API also recommends that the `hashcode()` method also be overridden so as to maintain the general contract for the `hashcode()` method, which states:

> *If two objects are equal according to the equals(Object) method, then calling the hashcode method on each of the two objects must produce the same integer result.*

This makes sense from the standpoint that if two objects are logically equal, as determined by the `equals()` method, they should also return the same value for `hashcode()`. Otherwise, equality of objects may be determined based on object references. An example of where this can occur is when items are added to hashing data structures. If the `hashcode()` method is not overridden, two different instances of the same class that are logically equal will be treated as different objects when adding them to a hashing data structure.

To ensure this security recommendation is satisfied, any user-defined class that overrides the `equals()` method must also override the `hashcode()` method to ensure that two different instances of the same class will return the same value for `hashcode()`. This can be done by exploiting one or more attributes of the class when computing the hashcode.

MET10-J. Follow the general contract when implementing the compareTo() method

Classes that implement the `Comparable` interface must provide an implementation of the `compareTo()` method. It is essential that the implementation of the `compareTo()` method obeys the contract specified in the Java API. There are 4 points to this contract, as taken from the Java API documentation:

1. The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.) The notation `sgn(`*expression*`)` designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of *expression* is negative, zero or positive.

2. The implementor must also ensure that the relation is transitive: `(x.compareTo(y) > 0 && y.compareTo(z )> 0)` implies `x.compareTo(z) > 0`.

3. Finally, the implementor must ensure that `x.compareTo(y) == 0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

4. It is also strongly recommended, but not strictly required that `(x.compareTo(y)==0) == (x.equals(y))`, to ensure consistency of equality when measured by either `compareTo()` or `equals()`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "*Note: this class has a natural ordering that is inconsistent with equals.*"

Implementations of `compareTo()` must never violate any of the first three conditions. The fourth condition can technically be violated since it is only a recommendation of the Java API, but it would be wise to conform to all four conditions.

MET11-J. Ensure that keys used in comparison operations are immutable

Certain data structures map items to keys, which are used for retrieving those items from the data structure. Keys can be primitive data values, or they can be objects. When objects are used as keys it is essential that the objects are immutable; otherwise, the mapping of keys to items will be corrupted, and the data structures in java.util do not specify anything about the behavior should mutable keys be used. When keys are used in comparisons, the guidelines for following the contracts of the `equals()`, `compareTo()`, and `hashcode()` methods are all obeyed. If some attributes of an object key must be mutable, there must also be at least one immutable attribute, and comparisons of keys must use only immutable attributes.

Java does provide a mechanism for declaring attributes of a class as immutable. Using the `final` keyword in the declaration of an attribute ensures that the value of the attribute can be set once, but can never be modified thereafter.

Secure coding guideline #52. Avoid in-band error indicators

An in-band error indicator is a value returned by a method that indicates either a legitimate return value or an illegitimate value that denotes an error. In other words, of the entire range of possible values a method might return, if one or more of those values indicates an error state, and there is only a single return statement, then the return statement can return in-band error indicators. Values that indicate an error state can be special, user-defined values, such as -1 or "EOF". The `null` reference is also considered an in-band error indicator if it is used in this way. The following code shows an example of this:

```
static final int MAX = 21;
static final int MAX_READ = MAX - 1;
static final char TERMINATOR = '\\';
int read; char[] chBuff = new char[MAX];
BufferedReader buffRdr;
// Set up buffRdr
read = buffRdr.read(chBuff, 0, MAX_READ);
chBuff[read] = TERMINATOR;
```

The problem lies in the assignment of the value for the variable, `read`. The `read(char[] buf, int off, int len)` method of the `BufferedReader` class returns one of two values: -1 if the end-of-file has been reached (an error code), or the number of characters read if the end-of-file has not been reached. If the method returns -1, an `ArrayIndexOutOfBounds` exception will be thrown in the subsequent statement, which attempts to add the `TERMINATOR` character to the end of the array, `chBuff`.

One solution to this problem is to wrap the call to the read method in another method that pe forms a check to see whether the end-of-file has been reached. If the end-of-file has been reached, an exception is thrown; otherwise, the wrapper method simply returns the number of characters read. This code is shown below:

```
public static int readSafe(BufferedReader buffer, char[] cbuf, int off, int len)
                        throws IOException
{
        int read = buffer.read(cbuf, off, len);

        if (read = = -1)
        {
                throw new EOFException();
        }
        else
        {
                return read;
        }
}
```