CSE 331 Software Design and Implementation

Handout C7: Rep Invariants and Abstraction Functions

Contents:

Introduction

This handout describes how to document a class's implementation. We assume that you have already read and understood Class and Method Specifications.

We begin with brief definitions of the concepts this document discusses. As an example, we will use the same class, `Line` as from Class and Method Specifications. Notice that we are now showing `Line`'s implementation because this handout covers documenting a class's implementation as opposed to its specification.

```
/**
 * This class represents the mathematical concept of a line segment.
 *
 * Specification fields:
 *  @specfield start-point : point  // The starting point of the line.
 *  @specfield end-point   : point  // The ending point of the line.
 *
 * Derived specification fields:
 *  @derivedfield length : real     // length = sqrt((start-point.x - end-point.x)^2 +
(start-point.y - end-point.y)^2)
 *                                  // The length of the line.
 *
 * Abstract Invariant:
 *  A line's start-point must be different from its end-point.
 */
public class Line {

  /** The x-coordinate of the line's starting point. */
  private int startX;

  /** The y-coordinate of the line's starting point. */
  private int startY;

  /** The x-coordinate of the line's ending point. */
  private int endX;

  /** The y-coordinate of the line's ending point. */
  private int endY;

  // Representation Invariant:
  //   ! (startX == endX && startY == endY)
  //
  // Abstraction Function:
  //  AF(r) = a line, l, such that
  //    l.start-point = (r.startX, r.startY)
  //    l.end-point = (r.endX, r.endY)

 /**
  * @requires p != null && ! p.equals(start-point)
  * @modifies this
  * @effects Sets end-point to p
  */
  public void setEndPoint(Point p) {
    ...
  }

  ...

}
```

Concrete Representation

How the abstract state of a class is represented within a Java object. For example, `line` uses four fields with type `int`.

Representation Invariant

A condition that must be true over all valid concrete representations of a class. The representation invariant also defines the domain of the abstraction function. `Line` requires that it is never the case that `r.startX == r.endX` and `r.startY == r.endY` . This condition is required by the abstract invariant.

Abstraction Function

A function from an object's concrete representation to the abstract value it represents. The abstraction function for `Line` is trivial: A concrete `Line` instance, `r`, is mapped to a line, `l`, having `l.start-point` equal to `(r.xStart, r.yStart)` and `l.end-point` equal to `(r.xEnd, r.yEnd)`.

Representation invariants and abstraction functions are internal documentation of a class's implementation details. A client should not need any of this information to properly use the class. This information is recorded to help with implementing, testing, debugging, modifying, and extending the class. Therefore, abstraction functions and representation invariants should not appear in a class's description or specification (Javadoc). Instead, the information should appear as internal comments in the class's body generally, using "//").

This document starts by telling how to relate a class's concrete representation to what it abstractly represents through the use of representation invariants and abstraction functions. Then it provides two examples of these concepts and the concepts presented in Class and Method Specifications. Finally, it ends with some general hints for writing understandable documentation.

Rep Invariants

A rep invariant RI maps the concrete representation to a Boolean (true or false). Formally,

RI: *R => boolean*

where R is the set of rep values. The rep invariant describes whether a rep value is a well-formed instance of the type.

The comment describing a rep invariant may explicitly emphasize the functional aspect of RI:

```
 // Rep invariant is
```

```
//    RI(r) = r.name != null && r.balance >= 0
```

More commonly, however, we just drop the RI(r) detail and simply write the rep invariant as a predicate that must be true of `this`:

```
 // Rep invariant is
//    name != null && balance >= 0
```

A rep invariant may mix concrete Java syntax (rep field references, method calls, `instanceof`, `!=`, `==`) with abstract mathematical syntax (sequence/set/tuple construction, for all, there exists, summation, `=`). A rep invariant may also be simple English, of course, as long as it is unambiguous. Here are some examples:

```
 //    for all i, transactions[i] instanceof Trans

 //    suit in {Clubs,Diamonds,Hearts,Spades}

 //    string contains no duplicate characters

 //    size = number of non-null entries in array
```

If the rep uses other ADTs, it may refer to them either by their spec fields or by their operations. For example, suppose the `Trans` type has a spec field `amount` that is accessible by the operation `getAmount`. Then the rep invariant for `Account` (which uses `Trans` objects) might look like this:

```
(1)   //    balance == sum (for all i) of transactions[i].amount
```

or this:

```
(2)   //    balance == sum (for all i) of transactions[i].getAmount()
```

or even this (since `transactions` is an instance of an ADT with a `get` operation):

```
(3)   //    balance == sum (for all i) of transactions.get(i).getAmount()
```

All these are equivalent. But it's important to keep in mind that `amount` in (1) above refers to a *spec field* in `Trans`, not to a concrete field. Unless `Account` and `Trans` are cooperating very closely, it isn't appropriate for `Account` to break the abstraction barrier and refer to the rep fields of `Trans` directly. By contrast, it's perfectly normal for `Account`'s rep invariant to refer directly to `Account`'s rep fields.

Choosing the Rep Invariant

Writing down a rep invariant is tremendously useful for testing, debugging, and writing correct code. It's essential for maintainers: programmers who come back to fix or enhance the code later. The most common problem with rep invariants is *incompleteness* - leaving out something important. (Leaving out the rep invariant entirely is probably the most common example of this problem!) So here are some hints that will help you fill out your rep invariants.

Look for rep values on which the abstraction function has no meaning. The abstraction function is often a *partial* function, meaning that it isn't defined for some values in *R*. The rep invariant must exclude any such values. Recall the `Card` example from above:

```
public class Card {
    private int index;
    // Abstraction function is
    //    suit = S(index div 13)   where S(0)=Clubs, S(1)=Diamonds, S(2)=Hearts, S(3)=Spades
    //    value = V(index mod 13)  where V(1)=Ace, V(2)=2, ..., V(10)=10, V(11)=Jack,
    //                                   V(12)=Queen, V(0)=King
    ...

}
```

In this case, the abstraction function isn't defined when the suit number, `index div 13`, is anything but 0, 1, 2, or 3. So that rules out values of `index` less than 0 or greater than 51:

```
    // Rep invariant is
    //    0 <= index <= 51
```

Make sure you don't restrict the rep invariant so much that the abstraction function no longer covers the entire abstract value space *A*! If you do that, you'll no longer be implementing the abstract type, since some abstract values will be unrepresentable. Here, a little thought convinces us that `index` still represents 52 cards, all distinct.

Look for rep values on which your methods would produce the wrong abstract value. Consider a class `CharSet` that represents a set of characters using a mutable string:

```
/** CharSet represents a mutable set of characters. */
public class CharSet {
    private StringBuffer chars;

    // Abstraction function is
    //    { chars[i] | 0 <= i < chars.size }
    ...

    /** @modifies this
     *  @effects this_post = this - {c}
     */
    public void remove (char c) {
        int i = chars.indexOf(Character.toString(c));
        if (i != -1) chars.deleteCharAt(i);
    }
}
```

This implementation of the `remove` method only works if there are no duplicates in the string, because it only deletes the first occurrence of `c` that it finds. Notice that the abstraction function is fully defined on *R* here - it doesn't care whether or not there are duplicate characters, because the set construction syntax implicitly ignores them. But we need the rep invariant to ensure that `remove` always results in the correct abstract value, i.e. a set that does not contain `c.`

Look for constraints required by the data structures or algorithms you've chosen. For example, an array must be sorted in order to use binary search. A tree cannot have cycles, and two nodes cannot share the same child.

Look for fields that need to stay synchronized with each other. In a bank account, for example, the balance and the sum of the transaction amounts should always be in synch. In a linked list, the `size` field always needs to accurately reflect the number of nodes in the list.

Look for rep values that would cause your code to throw unexpected exceptions. A conventional part of every rep invariant is a set of fields that shouldn't be null, so you won't have any `NullPointerException`s when you use them later:

```
    //   name != null && transactions != null
```

Make sure your constructors and producers actually establish the rep invariant, though.

That means, if any of these fields are initialized from parameters, you have to check for null before you put them in the rep.

Some other exceptional conditions you should think about:

- divide by zero
- index out of range
- class cast exception

Look for constraints imposed by the application domain (on abstract values). In a bank account, the balance should always be non-negative. (Or even stronger: the partial sums of the transaction amounts must all be non-negative, to guarantee that the balance never went negative in the past.) In chess, one white bishop should always be on a white square, while the other white bishop should be on a black square.

Assuming they refer only to properties of the abstract values (like spec fields), these kinds of constraints do not just apply to the rep. They are in fact *abstract invariants*. Abstract invariants should be documented for the client of the data type, so put them in your class overview. But they're also part of the representation invariant. If you omitted them from the rep invariant, then your type would represent abstract values that aren't well-formed.

Checking the Rep Invariant at Run Time

Many rep invariants can be translated straightforwardly into code. If that's the case, do it! Having an executable rep invariant, and using it at run time, not only helps find bugs in the code quickly; it also checks for mistakes in the rep invariant. Sometimes the rep invariant you've written is *too* strong, making assumptions that are unwarranted and unnecessary. Actually testing the invariant against running code is a good sanity check.

The rep invariant checker can be coded as a method `checkRep` of no arguments that throws an exception if the rep invariant is violated, preferably with a message indicating which constraint was broken. (However, a specification never mentions the representation invariant; from a client's point of view, the method never has any effects.) Here's an example:

```
private void checkRep () {
    if (balance < 0)
        throw new RuntimeException ("balance should be >= 0");

    if (name == null)
        throw new RuntimeException ("name should be non-null");

    // for all j, sum (i=0..j) transactions[i].amount >= 0
    int sum = 0;
    for (Trans t : transactions) {
        sum += t.getAmount();
        if (sum < 0)
           throw new RuntimeException("balance went negative");
    }
    // balance = sum (for all i) transactions[i].amount
    if (balance != sum)
        throw new RuntimeException ("balance should equal sum of transactions[i].amount");
}
```

The best place to put `checkRep` in your class is right after your fields. You can either write your rep invariant as a separate comment, or intersperse the constraints of your rep invariant as exception messages in `checkRep` (as was done above). The latter approach is probably better, because it makes it more likely that the comment will be kept current with the code in `checkRep`. Any part of the rep invariant that you can't write as executable code, just leave as a comment.

Assertions provide an even cleaner way to write `checkRep`, because they handle the test and exception for you. Here are two ways to do it:

```
      // using Java assert syntax
      assert balance >= 0 : "balance should be >= 0";

// using junit.framework.Assert (you don't have to be writing a unit test to use this class)
     Assert.assertTrue ("balance should be >= 0", balance >= 0);
```

Calls to `checkRep` should be placed at the start and end of every public method, and at the end of every constructor. Put a call to `checkRep` at the end of observers, even if you think they don't change the representation (since you may be wrong). Private methods generally don't call `checkRep`, because private methods may be designed to be called while the rep is in an intermediate state that doesn't satisfy the rep invariant.

If part of the rep invariant is very expensive to check, you may want to turn that part off in the release version. Otherwise it makes sense to leave it in. Be careful how you judge performance here. Novices are often much too ready to worry about performance improvements that turn out to be negligible. Before dropping a check, you should have some evidence that it's expensive, such as an analysis with a profiler showing that indeed the check is a hotspot, or a theoretical argument, for example that the check turns a constant time operation into a linear time one. Checking fields against null is always a constant time operation; there's no reason to drop this check except in absolutely performance-critical code. Summing all the transactions in an account is linear in the number of transactions, which you might not want to do for every method call in production code. But even expensive `checkRep`s are justified during testing and debugging; they'll more than pay for themselves in reduced debugging time. An easy way to enable/disable the `checkRep` invocation is to add a static boolean variable named `debug` to the class, and either check that variable within `checkRep` or write calls like `if (debug) checkRep();`. Then, you can disable all the debugging checks by just changing the variable's value.

The `checkRep` method can also be made public, so that unit tests can call it during testing. `checkRep` is normally private, since the representation is not part of the specification; clients aren't supposed to be aware that there's a rep under the covers that might be broken. (Horrors!) But it doesn't expose the rep if you make it public. If your class is working properly, then from the client's point of view, `checkRep` is just a no-op. If you make `checkRep` public, you should probably specify it that way, so that clients don't bother calling it unless they're paranoid: "This operation does nothing unless there's a bug, in which case it (sometimes) throws an exception." Needless to say, don't put the rep invariant itself in the spec for `checkRep`. That's representation-dependent.

What Not to Write in the Rep Invariant

Your rep invariant does not have to mention facts that are impossible in the rep. You are allowed to rely on the guarantees of the ADTs in your concrete fields. For example, if one of your concrete fields is an `int`, your rep invariant shouldn't mention that its value is less than or equal to `Integer.MAX_VALUE` or that it is not `null`. Likewise, if one of your concrete fields is a set, your rep invariant needn't mention that it contains no duplicates.

This is similar to the way that a method precondition does not need to mention properties that are guaranteed by the ADTs of the formal parameters.

The rep invariant should be expressible as a `checkRep` method. Don't write anything in the rep invariant that cannot be checked by just examining the concrete representation. Checking the rep invariant should not require any knowledge of what the representation means nor of what operations were performed to create the representation.

Abstraction Functions

An abstraction function AF maps the concrete representation of an abstract data type to the abstract value that the ADT represents. Formally,

AF: *R* => *A*

where R is the set of rep (representation) values, and A is the set of abstract values. You can think of an element in *R* as the Java object. *A*, on the other hand, exists only in our imagination, and has no existence inside the computer. For example, if this is the ADT:

```
class Complex {
  private double real;
  private double imag;
  ...
}
```

then the rep space *R* is the set of `Complex` objects, and the abstract space *A* is the set of complex numbers.

Where Are R and A Defined?

The rep space *R* is obvious: it's defined by the fields you put in your class. You can't possibly implement an abstract data type without fields, so you'll never forget this. The abstract value space *A*, however, is not represented in code. It should be documented in your class overview (see Abstract State), because both clients and implementors want to know what abstract type the class represents:

```
/**
 * Complex represents an immutable complex number.    <- this is A
 */
public class Complex {
  private double real;     <- these fields form R
  private double imag;
  ...
}
```

Whether the type is mutable or immutable is a crucial property that should be mentioned in the class overview.

AF(r)

Technically, an abstraction function is a *function*. This is why you may see it written using functional notation, AF(r):

```
/**
 * Complex represents an immutable complex number.
 */
public class Complex {
  private double real;
  private double imag;

  // The abstraction function is
  //      AF(r) = r.real + i * r.imag
  ...
}
```

The *r* in AF(r) represents an element in the rep space. In other words, it's a reference to a `Complex` object. So we can refer to fields of the object *r* on the right-hand side of the abstraction function. (Incidentally, *r* stands for *representation*.)

The functional notation is essential when the abstraction function is recursive, since we need some way to refer to it on the right-hand side:

```
/**
 * Cons represents an immutable sequence of objects.
 */
public class Cons {
  private Object car;
  private Cons cdr;

  // The abstraction function is
  //      AF(r) = [] if r = null
  //              [r.car] : AF(r.cdr) if r != null
  ...
}
```

(The square brackets and colons are sequence construction syntax.)

However, usually the abstraction function isn't recursively defined. Then it's more readable just to write the right-hand side. We further assume that the rep object *r* represents `this`, and adopt the convention of dropping references to `this` when writing the abstraction function:

```
/**
 * Complex represents an immutable complex number.
 */
public class Complex {
  private double real;
  private double imag;

  // The abstraction function is
  //      real + i * imag
  ...
}
```

This is simple and clear, but remember that it's just shorthand for AF(r).

For ADTs with trivial reps, the spec fields may correspond one-to-one with rep fields:

```
public class Line {
    private Point start;
    private Point end;

    // Abstraction function is
    //    AF(r) = line l such that
    //        l.start = r.start
    //        l.end = r.end
    ...
}
```

But this abstraction function is hardly worth writing down. Here's a more interesting rep for the same ADT:

```
public class Line {
    private Point start;
    private double length;
    private double angle;

    // Abstraction function is
    //    AF(r) = line l such that
    //        l.start = r.start
    //        l.end.x = r.start.x + r.length * cos(r.angle)
    //        l.end.y = r.start.y + r.length * sin(r.angle)
    ...
}
```

Note that `x` and `y` are spec fields of the `Point` type. It was convenient to define the point `end` in terms of its spec fields as well.

Let's simplify this with some more shorthand. We'll drop the AF(r), as we did earlier. We'll assume that the rep value *r* and the abstract value *l* both represent `this` - just different aspects of `this` - and adopt the convention of dropping references to `this`. The effect of all this shorthand is just a list of equations defining each spec field in terms of the concrete fields:

```
    // Abstraction function is
    //       start = start
    //       end.x = start.x + length * cos(angle)
    //       end.y = start.y + length * sin(angle)
```

Keep in mind that on the left-hand side, `start` refers to a spec field; on the right-hand side, `start` refers to a concrete field. (`x` and `y` *always* refer to spec fields here, because the rep of the `Point` type isn't visible to us.) Isn't there a danger of confusion between the two `starts`? Not really. We gave the spec field and the concrete field the same name for good reason - because they are equated by the abstraction function. Do we really have to say `start=start` explicitly in the abstraction function? Probably not. If the spec field `start` and the rep field `start` weren't equated by the abstraction function, then we should have given them different names. Remember that your goal in these kinds of specifications is not formal communication with a machine, but clear and unambiguous communication with other human beings (Not only the author(s)!). Names matter. Sometimes abbreviations help, and sometimes they do not.

Here's another example. Suppose we want to represent a `Card` data type, in a poker game, using a single integer in a field `index`. We might have two specification fields, `suit` and `value`:

```
/**
 * Card represents an immutable playing card.
 * @specfield suit : {Clubs,Diamonds,Hearts,Spades} // card suit
 * @specfield value : {Ace,2,...,10,Jack,Queen,King} // card rank
 */
```

The abstraction function then describes how to peel apart the `index` field into a suit and a value:

```
public class Card {
    private int index;

    // Abstraction function is
    //    suit = S(index div 13)   where S(0)=Clubs, S(1)=Diamonds, S(2)=Hearts, S(3)=Spades
    //    value = V(index mod 13)  where V(1)=Ace, V(2)=2, ..., V(10)=10,

    //                                     V(11)=Jack, V(12)=Queen, V(0)=King
      //    (div and mod refer to the integer division and remainder operations)

      //    for example, 3 => Three of Clubs, 14 => Ace of Diamonds
      ...
}
```

This abstraction function maps each representation object to a pair `(suit,value)`, but rather than writing it as a single function, we've specified it as two separate ones, one for each specification.

(Two incidental points: you may wonder why Ace is V(1) instead of V(0). This was done to make the abstraction function more direct on the numbered cards, so that V(*i*) = *i* for 2 through 10. But it may not be ideal; since King is V(0), we can't compare cards by comparing the `index` field directly. Second, this representation of `Card` is tightly coupled to the set of suits and the set of values. If we expect those types to change in the future, e.g., adding or removing suits, then we would have to change the representation of `Card`.

For some applications, however, the compactness of the representation may be worth the disadvantages of greater coupling.)

Example 1: Card

Here's a complete example of a simple class with an abstraction function and a rep invariant, so you can see one way you might structure your code.

```
/**
 * Card represents an immutable playing card.
 * @specfield suit : {Clubs,Diamonds,Hearts,Spades} // card suit
 * @specfield value : {Ace,2,...,10,Jack,Queen,King} // card rank
 */
public class Card {

    private int index;

    // Abstraction function is
    //    suit = S(index div 13)   where S(0)=Clubs, S(1)=Diamonds, S(2)=Hearts, S(3)=Spades
    //    value = V(index mod 13)  where V(1)=Ace, V(2)=2, ..., V(10)=10, V(11)=Jack,
    //                                   V(12)=Queen, V(0)=King


    // Rep invariant is
    //    0 <= index <= 51

    /**
     * Check the rep invariant.
     * @effects: nothing if this satisfies rep invariant;
     *           otherwise throws an exception
     */
    private void checkRep() {
        if (index < 0 || index > 51)
            throw new RuntimeException ("card index out of range");
    }

    /**
     * @effects makes a new playing card with given suit and value
     */
    public Card(CardSuit suit, CardValue value) {
        ... // initialize Card
        checkRep ();
    }

    /**
     * @effects returns this.suit   <- "suit" refers to the spec field
     */
    public CardSuit getSuit() {
        checkRep ();
        try {
            CardSuit s = ... // decode suit
            return s;
        } finally {
            checkRep ();
        }
    }

    /**
     * @effects returns this.value  <- "value" is the spec field
     */
    public CardValue getValue() {
        checkRep ();
        try {
            CardValue v = ... // decode value
            return v;
        } finally {
            checkRep ();
        }
    }

    ...
}
```

The calls to checkRep() in all the observers are probably unnecessary in this case, since the class is simple, immutable, and clearly has no rep exposure. They're included anyway to illustrate what you would want to do in a more complex class.

Example 2: Stack

Suppose we wanted to implement a Stack ADT with an array.

The Representation Invariant

Here are some possible representation invariants:

```
public class Stack {
    private int[] elements;

    // Abstraction function:
    //    The Nth element from the bottom of the stack = elements[N-1]
    //    where the 1st element is at the bottom

    // Rep Invariant:
    //    For any index i such that 0 <= i < size, elements[i] != null

    // OR

    // Rep Invariant:
    //    elements never contains a null value
}
```

One can imagine a slight change if the RI was for a SortedStack:

```
public class SortedStack {
    private int[] elements;

    // Abstraction function:
    //    The value considered 'least' in the stack = elements[size-1]
    //    ...
    //    The value considered 'greatest' in the stack = elements[0]

    // Rep Invariant:
    //    For any index i such that 0 <= i < size: elements[i] != null
    //    For any index j such that 1 <= j < size: elements[j] <= elements[j-1]
}
```

In this case, our RI is distinguishing limitations on the internal state of our ADT.

The purpose of the RI: to define valid and invalid internal states for this ADT object (a SortedStack should always be sorted and never contain a null value). The `checkRep()` method should mirror the RI.

The Abstraction Function

Here are some example AFs.

```
// For a naive implementation with poor run-time performance:
public class Stack {
    private int[] elements;
    private int size;

    // Abstraction function:
    //    The top element of the stack = elements[0]
    //    The second-from-the-top element of the stack = elements[1]
    //    ...
    //    The bottom element of the stack = elements[size-1]

    // OR

    // Abstraction function:
    //    The Nth element from the top of the stack = elements[N-1]
    //    where the 1st element is at the top
}
```

```
// For a tweaked implementation with great run-time performance:
public class Stack {
    private int[] elements;
    private int size;

    // Abstraction function:
    //    The top element of the stack = elements[size-1]
    //    The second-from-the-top element of the stack = elements[size-2]
    //    ...
    //    The bottom element of the stack = elements[0]

    // OR

    // Abstraction function:
    //    The Nth element from the bottom of the stack = elements[N-1]
    //    where the 1st element is at the bottom
}
```

Notice that the signature and fields of both examples are identical, but each abstraction function suggests different implementations with dramatically different run-time performance (If its unclear why the run-time performance is different, think about what needs to happen to the elements in each implementation's array when pushing and popping from the stack).

The purpose of the AF: to give a mapping of an ADT's abstract (specification) representation (in this case, a Stack containing values) **to the concrete (implementation) representation** (how the implementation stores those values in its array).

Subclasses

Abstraction functions and representation invariants are implementation-specific. Therefore, it does not make sense to inherit them from a superclass. When you write a subclass, you should define its abstraction function and representation invariant from scratch and write it out in full.

General Hints on Readability

Remember that the reader of your specifications, abstraction functions, and rep invariants is most likely going to be a *human being*, not a program. It might be a teammate trying to find a bug in the code; a maintainer charged with updating the software after you've left the company; or even yourself, six months (or days!) later, trying to remember how this program worked.

So formal syntactic correctness is actually *less* important than simplicity and clarity. That doesn't mean you should sacrifice semantic precision, or leave things ambiguous or undefined. But it does mean that you should think twice about writing something like this:

```
    // Abstraction function is
    //    <[x,y],s^[for all 0<i?chars.size(), chars[chars.size()-i]]> | x=front.cdr,
    //     y=back.car, s=n.toString()>
```

A compiler might enjoy reading this (if the compiler were actually reading your abstraction function). A human would stare at it and curse.

Here are some tips for making your abstraction functions and rep invariants more readable:

Introduce new names where they're useful:

```
    AF(r) = list l such that...

    AF(r) = c_0 + c_1*x + c_2*x^2 + ...

        where c_i = ...
```

- Use spec fields:

```
    suit = ...
    value = ...
```

- Provide examples:

```
     for example, 3 => Three of Clubs
```

- Introduce new functions where they're useful:

```
     reverse(back)
```

- Use plain language wherever it's unambiguous and clearer or more concise than formal syntax:

```
    chars contains no duplicates
```

Contrast this with the formal alternative:

```
    for all 0?i<j<chars.size, chars[i] != chars[j]
```

Back to Conceptual Info
Back to the CSE 331 home page.