How do we know what comments to write?

How do we know when to write comments?

The only point of agreement seems to be that comments should describe why the code does what it does.
In rare cases, the code is hard to understand and cannot be written more clearly. In this case, the comments do need to describe how the code does what it does.
Consider the following code which is taken from
http://rosettacode.org/wiki/Sorting_algorithms/
Bubble_sort#Java

This method sorts an array of ints
public static void bubbleSort(int[] toSort) {
    boolean changed = false;
    do {
        changed = false;
        for (int a = 0; a < toSort.length - 1; a++) {
            if (toSort[a] > toSort[a + 1]) {
                int tmp = toSort[a];
                toSort[a] = toSort[a + 1];
                toSort[a + 1] = tmp;
                changed = true;
            }
        }
    } while (changed);
}
How can we better comment the code?

Bubble sort

http://www.youtube.com/watch?v=lyZQPjUT5B4&t=0m51s

First we can add pre and post conditions:
Good (in my opinion) method comments:

```plaintext
/**
 * Pre: The argument to bubblesort is an array of ints.
 * Post: The array is in sorted order from least to greatest.
 * Bubblesort is slow to sort large arrays.
 * Note that no value is returned. This method changes the original array.
 * Source:
 * http://rosettacode.org/wiki/Sorting_algorithms/Bubblesort
 */
```
Bad method comments:

This is insufficient.

```cpp
/**
 * sorts the array
 */
```

Why is this bad?
Why is this bad?

At the very least we need to say that the array is sorted from least to greatest. I think it’s also important to comment on the return type, which is perhaps surprising. It is well known that bubbleSort is slow to sort large arrays so putting this in the comments may be unnecessary.
Bad pre method comments:

Gives how information, not why.

```c
/**
 * This method declares a boolean variable called changed, then enters a do-while loop. changed is set to false. For each element of the array toSort, consecutive elements are compared. If the first element is greater than the second then the two elements are swapped using a temporary variable named tmp and changed is set to true. While changed is true, the loop repeats.
 */
```

Why is this bad?
Why is this bad?

First of all, it describes what the program does but not why.
A reader of this comment hasn’t even been told that the method sorts its input.
On the other hand, these comments may help you understand what the code is doing.
Q. How do we decide what comments to write?
A. It depends on your audience (the people who will read your code).
In this class, I am your audience and I demand excessively commented code, so there you go.
You may find yourself in a business or other formal setting in which you are required to comment code in a specific way. Consider this practice for such a situation.
If your audience is just yourself then how many comments do you need?

- You need enough comments to quickly understand what you wrote and why.
- If you return to your code later, but are confused reading it, then you wrote bad code and too few comments.
- If you write zero comments and you understand your code perfectly, then you wrote enough comments.
Let’s comment this method **inline**. The following slide shows the comments I want to see. I will justify each comment in turn. The code no longer fits on one slide so it’s split across two.
public static void bubbleSort(int[] toSort) {
    /* If we iterate through the array and don’t
     * change anything then the array is sorted
     * and we can stop sorting it. Use the
     * boolean ‘changed’ to track if something
     * was changed. */
    boolean changed = false;
    do {
        changed = false;
        /* Subtract one because we will be
         * comparing element a and a+1 at each
         * loop iteration and don’t want to run
         * off the end of the array. */
        for (int a=0; a < toSort.length-1; a++) {
            /* If two consecutive elements are out
             * of order then swap them and note
             * that we had to change the array,
             * forcing us to loop through at least
             * once more. */
            ...
if (toSort[a] > toSort[a + 1]) {
    int tmp = toSort[a];
    toSort[a] = toSort[a + 1];
    toSort[a + 1] = tmp;
    changed = true;
}
} while (changed);
//the array toSort is now sorted.
This comment explains two non-obvious features of the algorithm.

1. do-while loops are not very common.
2. It’s not clear what the boolean variable `changed` is all about.
/* If we iterate through the array and don’t
 * change anything then the array is sorted
 * and we can stop sorting it. Use the
 * boolean ‘changed’ to track if something
 * was changed. */

I think this comment nicely summarizes the purpose, the why, of the variable *changed* and the do-while loop.
If your audience is not familiar with bubble sort you may want an additional comment about why we need a do-while instead of a while. The reason is that we need to check the array at least once to see if it is in sorted order.
The most common for loop looks like this:

```java
for (int a=0; a < toSort.length; a++) {
}
```

but our method’s for loop looks like this:

```java
for (int a=0; a < toSort.length-1; a++) {
}
```
Some explanation of the \(-1\) is needed. I chose to explain it in terms of the consequences if it is ommitted.

/* Subtract one because we will be
 * comparing element a and a+1 at each
 * loop iteration and don’t want to run
 * off the end of the array. */
Under normal circumstances we don’t need to explain that we are iterating over the elements of the array `toSort`. This should be obvious from the code. We could explain why we are iterating the elements: To find out-of-order pairs and put them in order.
/* If two consecutive elements are out of order then swap them and note that we had to change the array, forcing us to loop through at least once more. */

if (toSort[a] > toSort[a + 1]) {
    int tmp = toSort[a];
    toSort[a] = toSort[a + 1];
    toSort[a + 1] = tmp;
    changed = true;
}

Unless you see a lot of swap code, it’s not obvious what is happening above.
I describe what is happening in English: *If two consecutive elements are out of order....*
I don’t say that *changed = true*, because that’s obvious from the code, but I describe the consequence of the code *forcing us to loop through at least once more.*
//the array toSort is now sorted.

Since there is nothing returned by the method, I feel that it is important to describe what has changed.
What comments would you add?
Which comments would you remove or change?
What if your audience is just yourself?
What if your audience doesn’t know programming?
What if your audience is a veteran programmer?
/**
 * Pre: The argument to bubblesort is an array of ints.
 * Post: The array is in sorted order from least to greatest.
 * Bubblesort is slow to sort large arrays.
 * Note that no value is returned. This method changes the original array.
 * Source:
 * http://rosettacode.org/wiki/Sorting_algorithms/Bubble_sort
 */
public static void bubbleSort(int[] toSort) {
    /* If we iterate through the array and don’t
     * change anything then the array is sorted
     * and we can stop sorting it. Use the
     * boolean ‘changed’ to track if something
     * was changed. */
    boolean changed = false;
    do {
        changed = false;
        /* Subtract one because we will be
         * comparing element a and a+1 at each
         * loop iteration and don’t want to run
         * off the end of the array. */
        for (int a = 0; a < toSort.length - 1; a++) {
            /* If two consecutive elements are out
             * of order then swap them and note
             * that we had to change the array,
             * forcing us to loop through at least
             * once more. */
        }
    }
}
if (toSort[a] > toSort[a + 1]) {
    int tmp = toSort[a];
    toSort[a] = toSort[a + 1];
    toSort[a + 1] = tmp;
    changed = true;
}
}
} while (changed);

//the array toSort is now sorted.