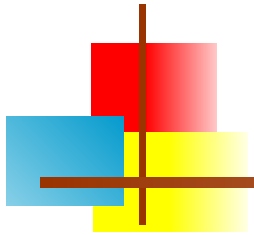
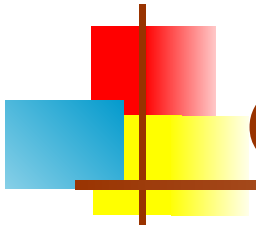


Control Structures

Processing and Java

- There is no difference between Processing syntax and Java syntax
  - Processing has it's own IDE and provides lots of methods to do drawing and animation
  - The main IDEs for Java are Eclipse (which we will use), NetBeans, and IntelliJ IDEA
- In Java, all code occurs within a class
  - Processing "hides" this level of complexity from you by putting your declarations and methods inside an "invisible" class
  - But it is still there, and the syntax is still Java

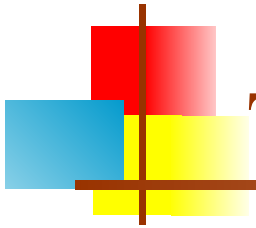


Compound statements

- Multiple statements can be grouped into a single statement by surrounding them with braces, { }
- Example:

```
if (score > 100) {
    score = 100;
    System.out.println("score has been adjusted");
}
```

- Unlike other statements, there is no semicolon after a compound statement
- Braces can also be used around a single statement, or no statements at all (to form an “empty” statement)
- Indentation and spacing should be as shown in the above example



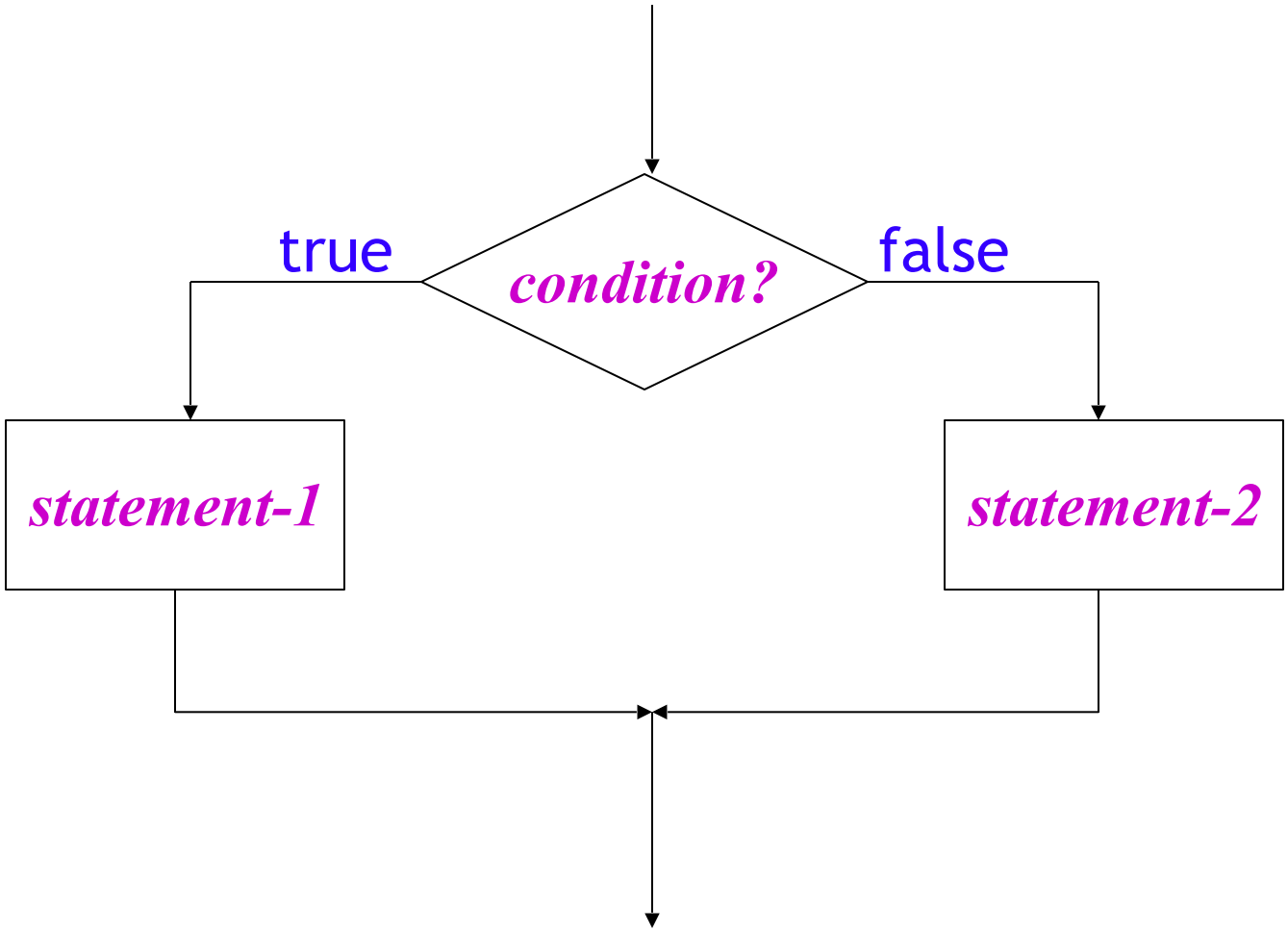
The if-else statement

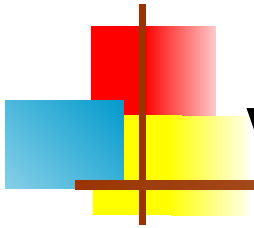
- The if-else statement chooses which of *two* statements to execute
- The if-else statement has the form:

  if (*condition*) *statement-to-execute-if-true* ;
  else *statement-to-execute-if-false* ;

- Either statement (or both) may be a compound statement
- Notice the semicolon after *each* statement
- The else part is optional

Flowchart for the if-else statement





while loops

- A while loop will execute the enclosed statement as long as a boolean condition remains true
  - **Syntax:** while (*boolean_condition*) {
    statement;
    }
  - **Example:**

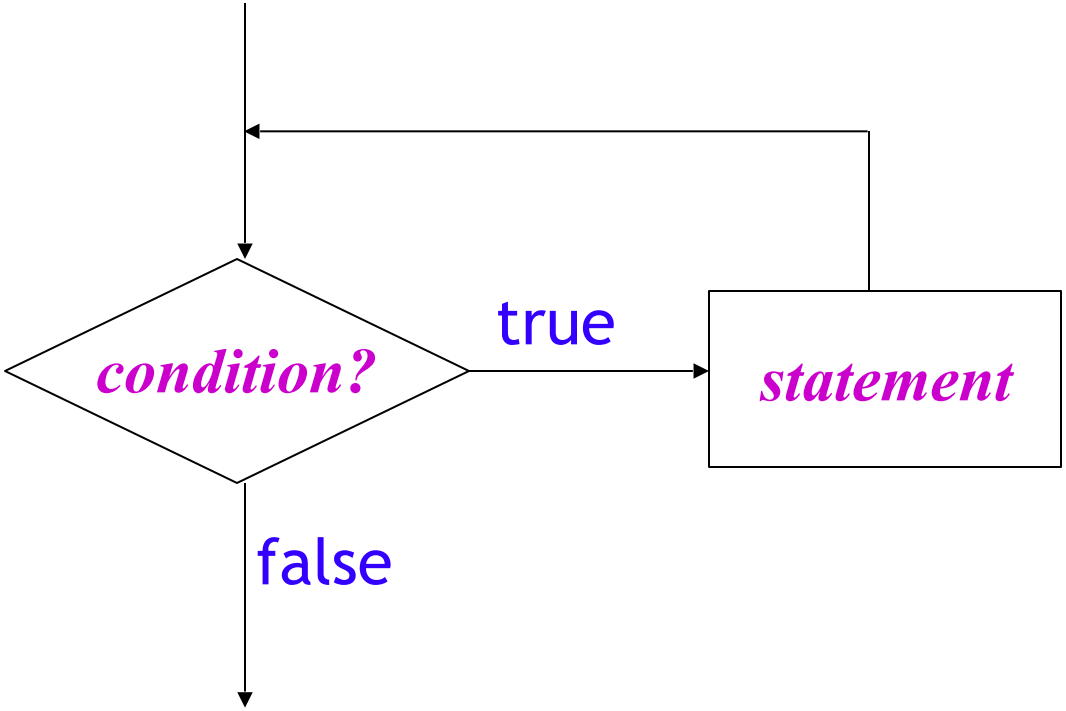
```
n = 1;
while (n < 4) {
    System.out.println(n + " squared is " + (n * n));
    n = n + 1;
}
```

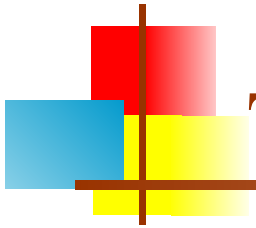
  - **Result:**

```
1 squared is 1
2 squared is 4
3 squared is 9
```

  - Python programmers: The parentheses are required
  - C programmers: The condition *must* be boolean
- Danger: If the condition never becomes false, the loop never exits, and the program never stops--this is called an infinite loop

Flowchart for the while loop





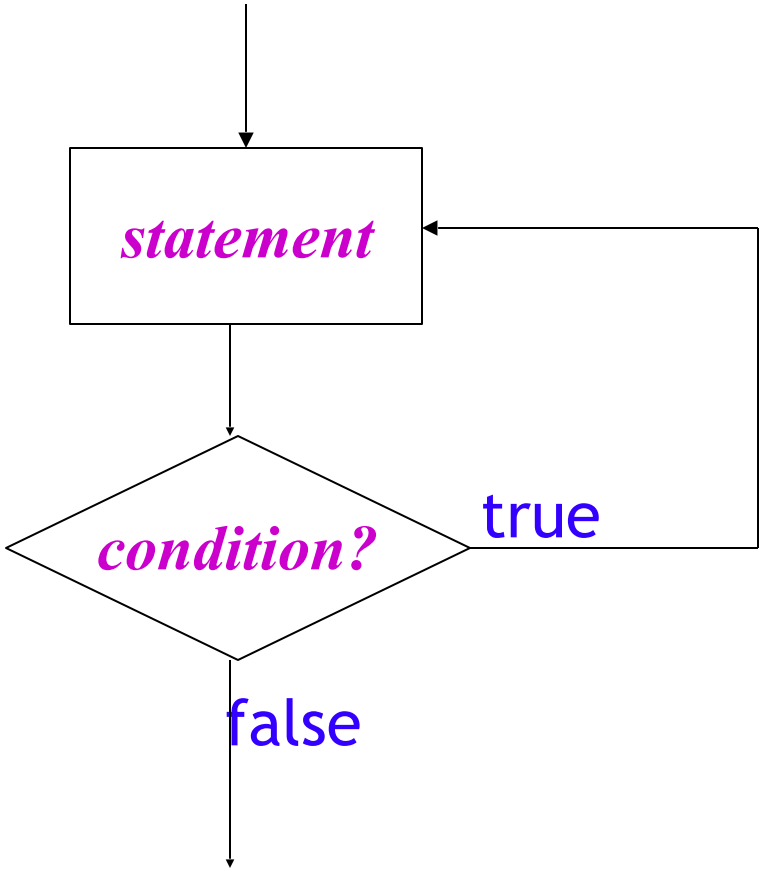
The do-while loop

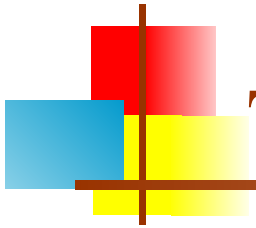
- The syntax for the do-while is:

```
do {
    …any number of statements…
} while (condition) ;
```

- The while loop performs the test first, before executing the statement
- The do-while statement performs the test *afterwards*
- As long as the test is true, the statements in the loop are executed again

Flowchart for the do-while loop





The increment operator

- ++ adds 1 to a variable
  - It can be used as a statement by itself, or within an expression
  - It can be put *before* or *after* a variable
  - If before a variable (preincrement), it means to add one to the variable, then use the result
  - If put after a variable (postincrement), it means to use the current value of the variable, then add one to the variable

Examples of ++

```
int a = 5;
a++;
// a is now 6

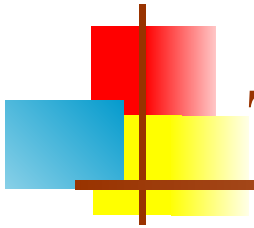
int b = 5;
++b;
// b is now 6

int c = 5;
int d = ++c;
// c is 6, d is 6
```

```
int e = 5;
int f = e++;
// e is 6, f is 5

int x = 10;
int y = 100;
int z = ++x + y++;
// x is 11, y is 101, z is 111
```

Confusing code is bad code, so this is very poor style



The decrement operator

- -- subtracts 1 from a variable
  - It can be used as a statement by itself, or within an expression
  - It can be put *before* or *after* a variable
  - If before a variable (predecrement), it means to subtract one from the variable, then use the result
  - If put after a variable (postdecrement), it means to use the current value of the variable, then subtract one from the variable

Examples of --

```
int a = 5;
a--;
// a is now 4

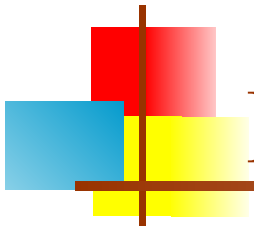
int b = 5;
--b;
// b is now 4

int c = 5;
int d = --c;
// c is 4, d is 4
```

```
int e = 5;
int f = e--;
// e is 4, f is 5

int x = 10;
int y = 100;
int z = --x + y--;
// x is 9, y is 99, z is 109
```

Confusing code is bad code, so this is very poor style



Example of confusing code

- Question: Do these two statements do the same thing?

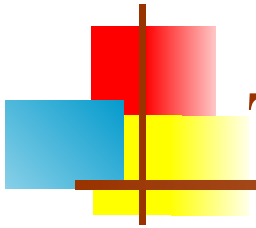
```
x = ++x;
x = x++;
```

- Let's try them:

```
int x = 5;
System.out.println(x);
x = ++x;
System.out.println(x);
x = x++;
System.out.println(x);
```

- Here are the results:

```
5
6
6
```

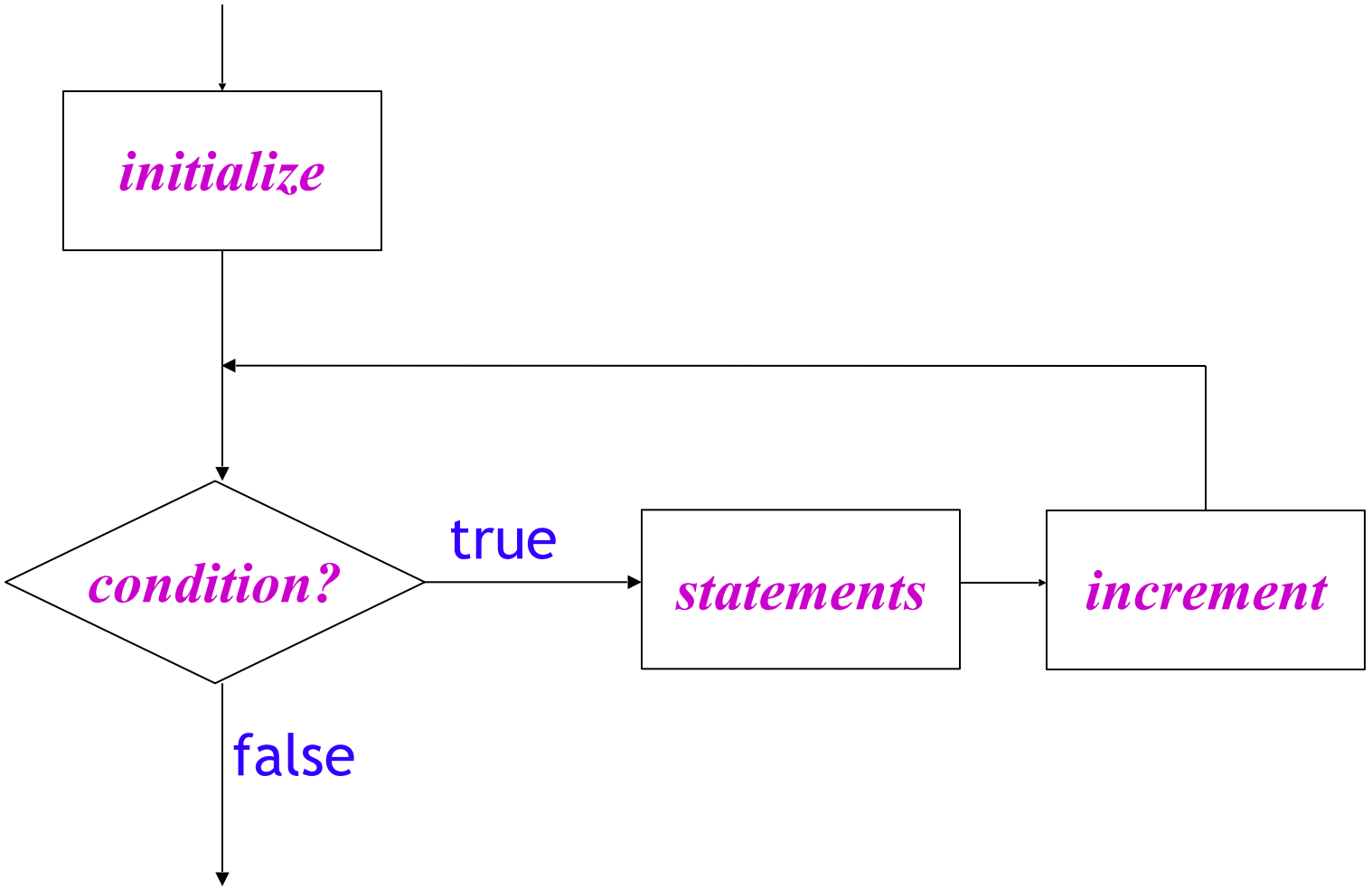


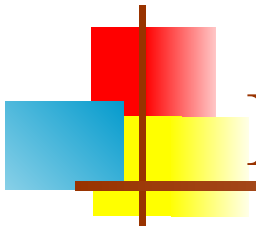
The for loop

- The for loop is complicated, but *very* handy
- Syntax:

  for (***initialize*** ; ***test*** ; ***increment***) ***statement*** ;
  - Notice that there is no semicolon after the ***increment***
- Execution:
  - The ***initialize*** part is done first and only once
  - The ***test*** is performed; as long as it is true,
    - The ***statement*** is executed
    - The ***increment*** is executed

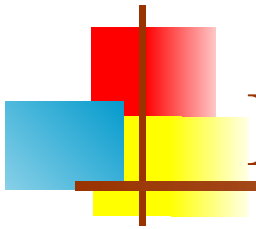
Flowchart for the for loop





Parts of the for loop

- *Initialize:* In this part you define the loop variable with an assignment statement, or with a declaration and initialization
  - Examples: `i = 0` `int i = 0` `i = 0, j = k + 1`
- *Test, or condition:* A boolean condition
  - Just like in the other control statements we have used
- *Increment:* An assignment to the loop variable, or an application of `++` or `--` to the loop variable



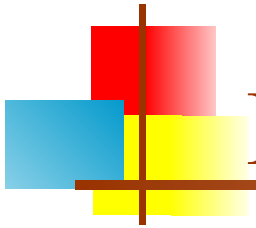
Example for loops

- Print the numbers 1 through 10, and their squares:

```
for (int i = 1; i < 11; i++) {
    System.out.println(i + "  " + (i * i));
}
```

- Print the squares of the first 100 integers, ten per line:

```
for (int i = 1; i < 101; i++) {
    System.out.print("  " + (i * i));
    if (i % 10 == 0) System.out.println();
}
```



Enhanced for loop

- The syntax of the new statement is

  for(*type var* : *array*) {…}

  or for(*type var* : *collection*) {…}

- Example:

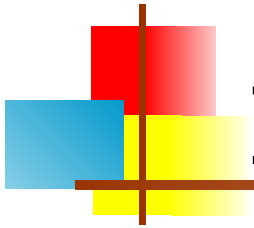
```
for(float x : myRealArray) {
    myRealSum += x;
}
```

- For a collection class that has an Iterator, instead of

```
for (Iterator iter = c.iterator(); iter.hasNext(); )
    ((TimerTask) iter.next()).cancel();
```

  you can now say

```
for (TimerTask task : c)
    task.cancel();
```



Iterators

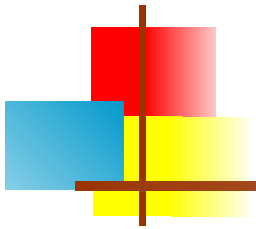
- Iterators are useful for stepping through collections, such as an ArrayList

```
public interface Iterator {

    boolean hasNext( );
        // true if there is another element

    Object next( );
        // returns the next element (advances the iterator)

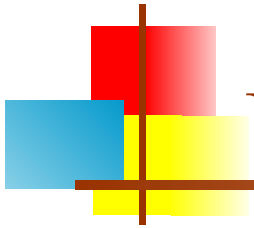
    void remove( );    // Optional
      // removes the element returned by next
}
```



Using an Iterator with a while loop

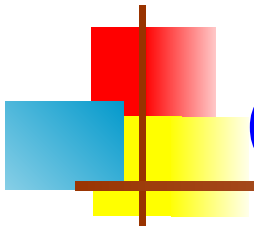
```
static void printAll (Collection coll) {
    Iterator iter = coll.iterator( );
    while (iter.hasNext( )) {
       System.out.println(iter.next( ) );
    }
}
```

- hasNext() just checks if there are any more elements
- next() returns the next element and advances in the collection
- Note that this code is polymorphic—it will work for *any* collection



Using an Iterator with a for loop

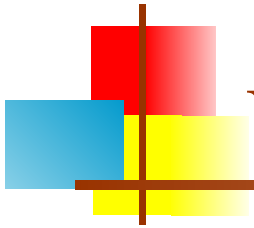
- Suppose you have
  - List<String> listOfStrings = new LinkedList<String>();
- You can print the strings this way:
  - for (Iterator<String> i = listOfStrings.iterator(); i.hasNext();) {
    String s = i.next();
    System.out.println(s);
    }
- Or better:
  - List<String> listOfStrings = new LinkedList<String>();
    ...
    for (String i : listOfStrings) {
    System.out.println(i);
    }



ConcurrentModificationException

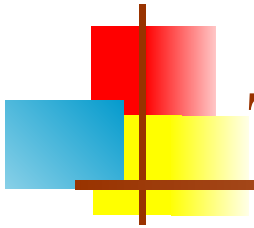
```
static void printAll (Collection coll) {
    Iterator iter = coll.iterator( );
    // When you create an iterator, a "fingerprint"
    // of the collection (list or array) is taken
    while (iter.hasNext( )) {
       System.out.println(iter.next( ) );
       // Both hasNext and next check to make sure
       // the collection hasn't been altered, and will
       // throw a ConcurrentModificationException
       // if it has
    }
}
```

- This means you cannot add or remove elements from the collection within the loop, or any method called from within the loop, *or from some other Thread* that has nothing to do with the loop



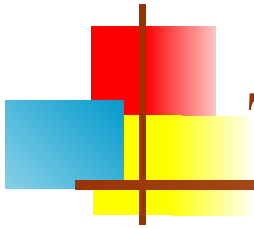
When do you use each loop?

- Use the enhanced for loop if you want to process every element of an array or collection, but you don't care about its index
  - Example: Print a 12-month calendar
- Use the for loop if you know ahead of time how many times you want to go through the loop, and need to know the index
  - Example: Stepping through an array
- Use the while loop in almost all other cases
  - Example: Compute the next step in an approximation until you get close enough
- Use the do-while loop if you must go through the loop at least once before it makes sense to do the test
  - Example: Ask for the password until user gets it right



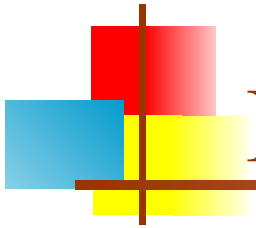
The break statement

- Inside any loop, the break statement will immediately get you out of the loop
  - If you are in nested loops, break gets you out of the *innermost* loop
- It doesn't make any sense to break out of a loop unconditionally—you should do it only as the result of an if test
- Example:
  - ```
    for (int i = 1; i <= 12; i++) {
        if (badEgg(i)) break;
    }
    ```
- break is not the normal way to leave a loop
  - Use it when necessary, but don't overuse it
    ```



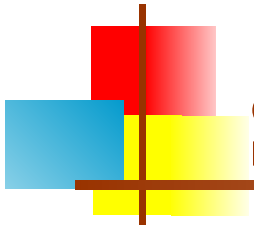
The continue statement

- Inside any loop, the continue statement will start the next pass through the loop
  - In a while or do-while loop, the continue statement will bring you to the test
  - In a for loop, the continue statement will bring you to the increment, *then* to the test



Multiway decisions

- The if-else statement chooses one of two statements, based on the value of a boolean expression
- The switch statement chooses one of several statements, based on the value on an integer (int, byte, short, or long) or a char expression
  - The value can also be an enum



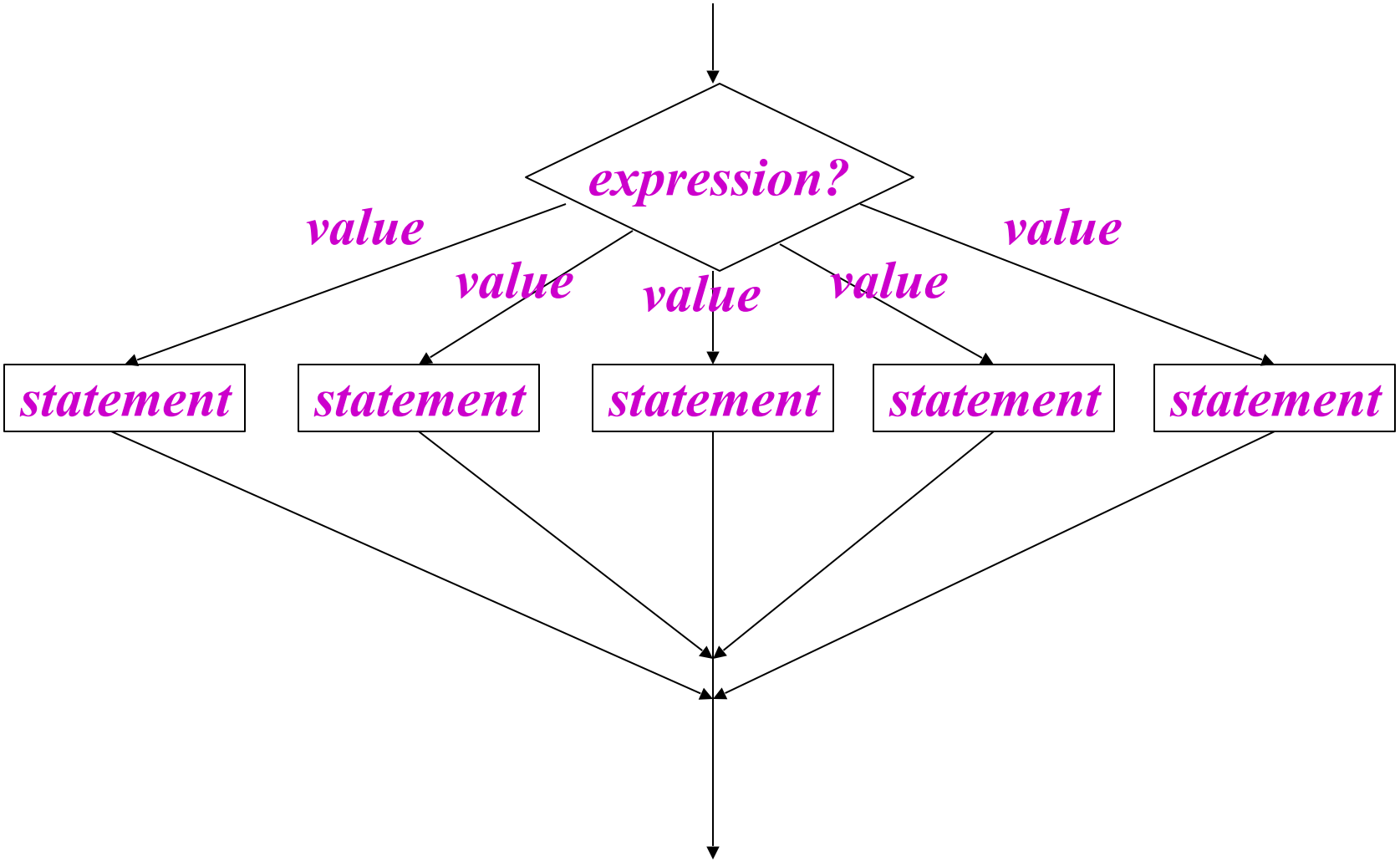
Syntax of the switch statement

- The syntax is:

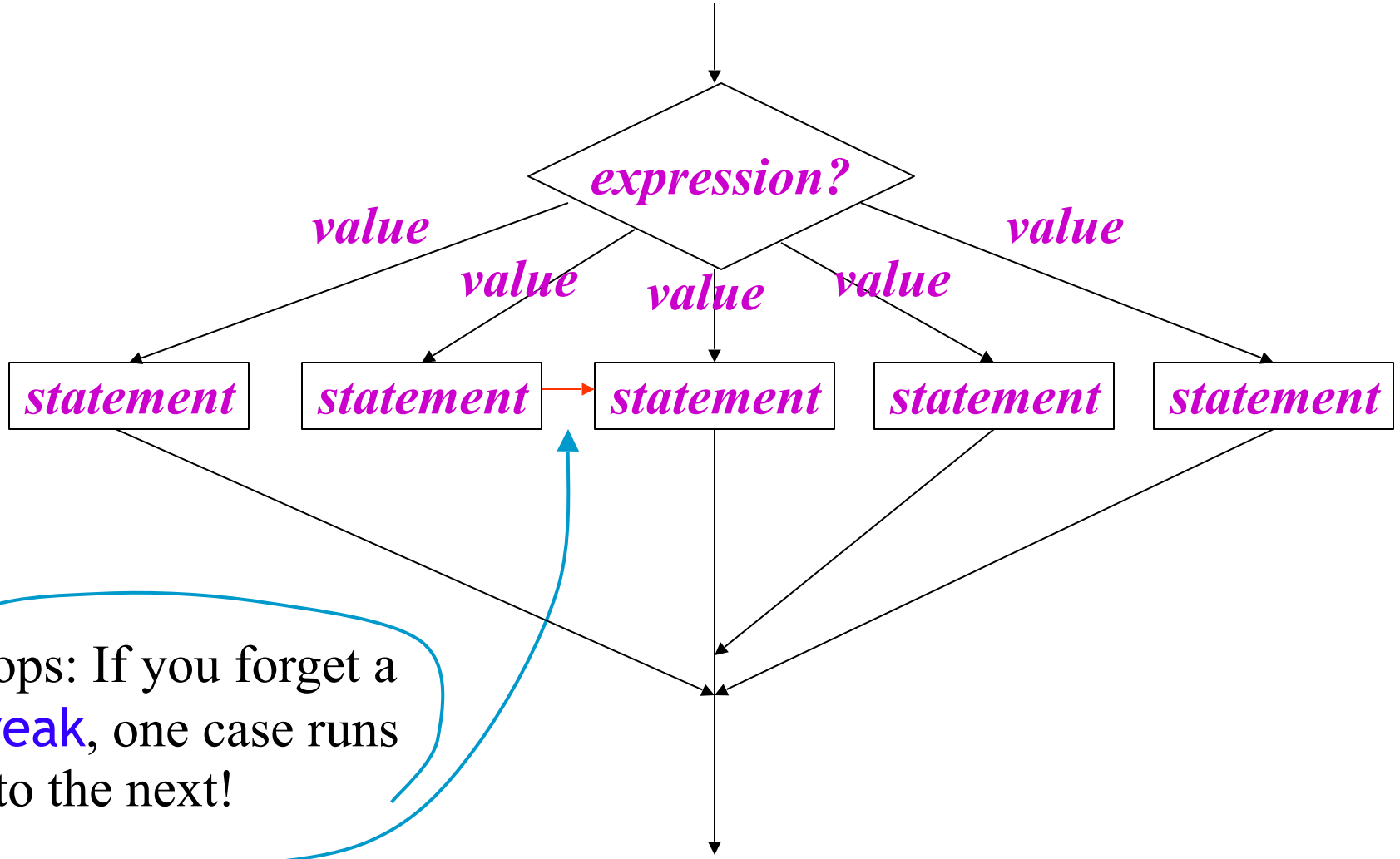
```
switch (expression) {
   case value1 :
      statements ;
      break ;
   case value2 :
      statements ;
      break ;
   ...(more cases)...
   default :
      statements ;
      break ;
}
```

- The *expression* must yield an integer or a character
- Each *value* must be a literal integer or character or enum
- Notice that colons (:) are used as well as semicolons
- The last statement in every case should be a break;
  - I even like to do this in the *last* case
- The default: case handles every value not otherwise handled

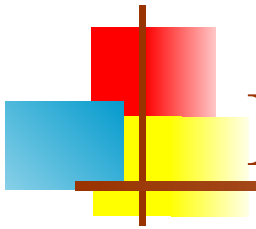
Flowchart for switch statement



Flowchart for switch statement

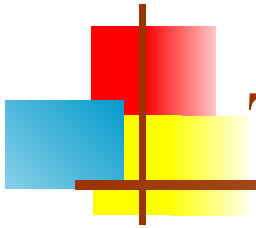


Oops: If you forget a break, one case runs into the next!



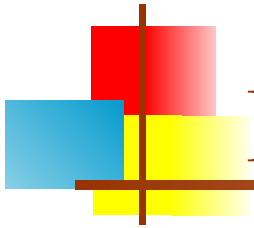
Example switch statement

```
switch (cardValue) {
    case 1:
        System.out.print("Ace");
        break;
    case 11:
        System.out.print("Jack");
        break;
    case 12:
        System.out.print("Queen");
        break;
    case 13:
        System.out.print("King");
        break;
    default:
        System.out.print(cardValue);
        break;
}
```



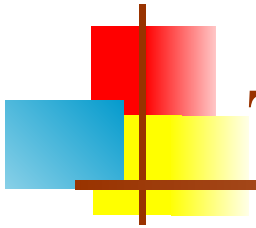
The assert statement

- The purpose of the assert statement is to document something you believe to be true
- There are two forms of the assert statement:
  1. assert ***booleanExpression***;
     - This statement tests the boolean expression
     - It does nothing if the boolean expression evaluates to true
     - If the boolean expression evaluates to false, this statement throws an AssertionError
  2. assert ***booleanExpression*** : ***expression***;
     - This form acts just like the first form
     - In addition, if the boolean expression evaluates to false, the second expression is used as a detail message for the AssertionError
     - The second expression may be of any type except void



Enabling assertions

- In Processing, assertions are enabled
- By default, Java has assertions *disabled*—that is, it *ignores* them
  - This is for efficiency
  - Once the program is completely debugged and given to the customer, nothing more will go wrong, so you don't need the assertions any more
    - Yeah, right!
- You can change this default
  - Open Window → Preferences → Java → Installed JREs
  - Select the JRE you are using (should be 1.6.*something*)
  - Click Edit…
  - For Default VM Arguments, enter `-ea` (enable assertions)
  - Click OK (twice) to finish



The End

"I think there is a world market for maybe five computers."

—Thomas Watson, Chairman of IBM, 1943