Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

# Dealing With Difference: Dispatch

## Chapter Overview

- How can I do different things at different times or under different circumstances?
- How can one method respond appropriately to many different inputs?

In previous chapters, we have looked at entities that respond to each input in roughly the same way. In this chapter, we will look at how an entity can respond differently depending on its input. In particular, we will look at how to build the central control loop of an entity whose job is to dispatch control to one of a set of internal "helper" procedures.

This chapter introduces several mechanisms for an entity to generate different behavior under different circumstances. Conditionals allow you to specify that a certain piece of code should only be executed under certain circumstances. This allows you to prevent potentially dangerous operations -- such as dividing by zero -- as well as to provide variant behavior.

The decision of how to respond often depends on the value of a particular expression. If there are a fixed finite number of possible values, and if the type of this expression is integral, we can use a special construct called a switch statement to efficiently handle the various options. A switch statement is often used together with symbolic constants, names whose most important property is that each one can be distinguished from the others.

Arrays are specialized collections of things. They allow you to treat a whole group of things uniformly. Arrays can be used to create conditional behavior under certain circumstances.

Procedural abstraction (covered in the next chapter) also plays a crucial role in designing good dispatch structures.

This chapter includes sidebars on the syntactic and semantic details of if, switch, and for statements, arrays, and constants. It is supplemented by portions of the reference chart on Java Statements.

## Conditional Behavior

The animate objects that we have seen so far generally execute the same instructions over and over. A clock ticks off the time. A stringTransformer reads a string, transforms it, and writes it out. A web browser receives a url request, fetches, and displays the web page. And so on. These entities repeatedly execute what we might call a central control loop, an infinitely repeated sequence of action.

In this chapter, we look instead at entities whose responses vary from one input to the next, based on properties of that input. The actual responses are not the subject of this chapter; instead, we will largely

assume that the object in question has methods to provide those behaviors. The topic of this chapter is how the central control loop selects among these methods. This function -- deciding how to respond by considering the value that you have been asked to respond to -- is called **dispatch**.

Imagine that we are building a calculator. One part of the calculator -- its graphical user interface, or GUI -- might keep a list of the buttons pressed, in order. The central controller might loop, each time asking the GUI for the next button pressed. The primary job of this central control loop would be to select the appropriate action to take depending on what kind of button was pressed, and then to dispatch control to this action-taker. For example, when a digit button is pressed, the calculator should display this digit, perhaps along with previously pressed numbers.[Footnote: Pressing 6 right after you turn on a calculator is different from pressing 6 after pressing 1 right after you turn on a calculator. In the first case, the calculator displays 6; in the second, it displays 16.] Pressing an arithmetic function key -- such as + or * -- means that subsequent digits should treated as a new number -- the second operand of the arithmetic operator -- rather than as additional digits on the first. Pressing = causes the calculator to do arithmetic. And so on.

In this example, the calculator's central control loop is behaving like a middle manager. It's not the boss, who gets to set direction. It's not the worker, who actually does what needs to be done. The dispatcher is there to see that the boss's directions (the button pressed) get translated into the appropriate action (the helper procedure). The dispatcher is simply directing traffic. This kind of behavior, in which different things happen under different circumstances, requires conditional behavior. We have already seen a simple kind of conditional behavior using Java's `if` statement. In this chapter, we explore several different means of achieving conditional behavior in greater detail.

Throughout this chapter, we will assume that we have methods that actually provide this behavior. For example, the calculator might have a `processDigitButton` method which would behave like exercise # in Chapter 7. Another method, `processOperatorButton`, would apply the appropriate operation to combine the value currently showing on the calculator's display with the number about to be entered. We will also use methods such as `isDigitButton` to test whether a particular buttonID corresponds to a number key. Separating the logic surrounding the use of these operations from their implementation is an important part of good design and the topic of much of the chapter on Encapsulation.

In this chapter, we are going to concern ourselves with what comes after the first line of the calculator's act method:

```
public void act()
{

    SomeType buttonID = this.gui.getButton();
    ....

}
```

The remainder of this method should contain code that calls, e.g., `processDigitButton` if `buttonID` corresponds to one of the buttons for digits 0 through 9, or `processOperatorButton` if `buttonID` corresponds to the button for addition. This chapter is about deciding which of these is the correct thing to do.

## If and else

We have already seen the `if` statement, Java's most general conditional. Almost every programming language has a similar statement type. An `if` statement is a compound statement involving a test expression and a body that can include arbitrary statements. Any conditional behavior that can be obtained in Java can be accomplished using (one or more) `if` statements. An `if` statement corresponds closely to normal use of conditional sentences in every-day language. For example, "If it is raining out, take an umbrella with you" is a sentence that tells you what to do when there's rain. Note that this sentence says nothing about what to do if there is no rain.

**Basic Form**

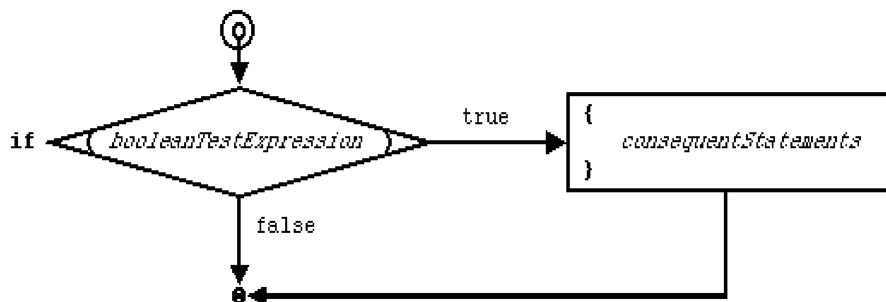
Every `if` statement involves two parts: the test expression and the consequent statement. The test expression represents the condition under which the consequent should be done. The test expression is some expression whose type *must* be `boolean`. In our example sentence, this boolean expression is "it is raining out". This expression is either true or false at any given time, [Footnote: Excluding that sort of grey dreary drippy weather that haunts London and certain times of the year in Maine, of course.] making it a natural language analog to a true-or-false boolean. In Java, this expression must be wrapped in parentheses.

When an if statement is executed, this conditional expression is evaluated, i.e., its value is computed. This value is either `true` or `false`. The evaluation of the boolean test expression is always the first step in executing an `if` statement. The rest of the execution of the `if` statement depends on whether this test condition is `true` or `false`.

In the English example above, if "it is raining out" is true -- i.e., if it is raining out at the time that the sentence is spoken -- then you should take an umbrella with you. That is, if the condition is true, you should do the next part of the statement. This part of the `if` statement -- the part that you do if the test expression's value is `true` -- is called the **consequent**.

In Java, execution of an `if` statement works the same way. First, evaluate the boolean test. If the value of the test expression is `true`, then execute the consequent. If the value of the test expression is `false`, the consequent is not executed. In this case, evaluating the test expression is the only thing that happens during the execution of the `if` statement. Note that the value of the expression that matters is its value *at the time of its evaluation*. If the test is executed at two different times, it may well have two different values at those times.

In Java, the consequent may be any arbitrary statement (including a block). In this book, we will always assume that the consequent is a block, i.e., a set of one or more statements enclosed in braces.



}

Figure #@@. The execution path of an `if` statement.

We could write pseudo-code for our English conditional as follows:

```
if ( currentWeather.isRaining() )
{
    take(umbrella);
}
```

This isn't runnable code, of course, but it does illustrate the syntax of a basic if statement: the keyword `if`, followed by a boolean expression wrapped in parentheses, followed by a block containing one or more statements. To execute it, we would first evaluate the (presumably boolean) expression `currentWeather.isRaining()` (perhaps by looking out the window) and then, depending on whether it *is* raining, either take an umbrella (i.e., execute `take( umbrella )`) or skip it.

A somewhat more realistic example is the following code to replace a previously defined number, `x`, with its absolute value:

```
if ( x < 0 ) {
    x = - x;
}
```

This code does nothing just in case x is greater than or equal to 0. [Footnote: It evaluates the expression `x < 0`, of course, but it "does nothing" that has any lasting effect.] If x happens to be less than 0, the value of x is changed so that x now refers to its additive inverse, i.e., its absolute value.

Note that the same `if` statement may be executed repeatedly, and the value of the boolean test expression may differ from one execution of the `if` statement to the next. (For example, it may be raining today but not tomorrow, so you should take your umbrella today but not tomorrow.) The value of the boolean test expression is checked exactly once each time the `if` statement is executed, as the first step of the statement's execution.

**Else**

The `if` statement as described above either executes its consequent or doesn't, depending on the state of the boolean test expression at the time that the `if` statement is executed. Often, we don't want to decide whether (or not) to do something; instead, we want to decide which of two things to do. For example, if it's raining, we should take an umbrella; otherwise, we should take sunglasses. We could express this using two if statements:

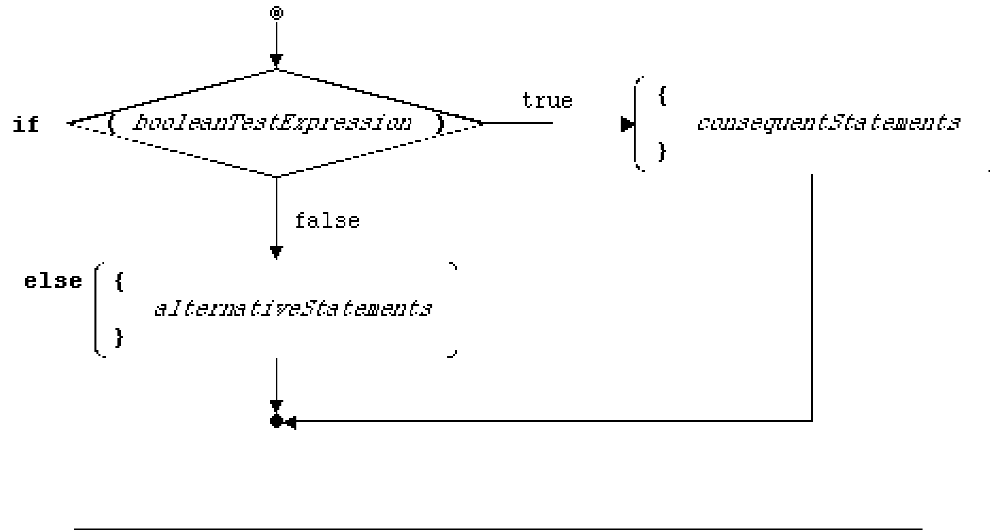
```
if ( currentWeather.isRaining() )
{
    take(umbrella);
}

if ( ! ( currentWeather.isRaining() ) )
```

```
        {
            take(sunglasses);
        }
```

Recall that ! is the Java operator whose value is the boolean opposite of its single argument. So if `currentWeather.isRaining()` is true, then `! (currentWeather.isRaining())` is false; if `currentWeather.isRaining()` is false, then `! (currentWeather.isRaining())` is true.

These two conditional statements, one after the other, are intended to express alternatives. But they don't, really. For example, the two statements each check the boolean condition `currentWeather.isRaining()`. This is like looking out the window twice. In fact, the answer in each of these cases might be different. If we don't get around to executing the second if statement (i.e., looking out the window the second time) for a little while, the weather might well have changed and we'd find ourselves without either umbrella or sunglasses (or with both). The weather doesn't usually change that often (except in New England), but there are plenty of things that your program could be checking that do change that quickly. And, since your program is a community, it is always possible that some other member of the community changed something while your back was turned. [Footnote: But see chapter 20, where we discuss mechanisms to prevent the wrong things from changing behind your back.]

Instead of two separate if statements, we have a way to say that these two actions are actually mutually exclusive alternatives. We use a second form of the `if` statement, the `if/else` statement, that allows us to express this kind of situation. An `if/else` statement has a single boolean test condition but two statements, the consequent and the **alternative**. Like the consequent, the alternative can be almost any statement but will in this book be restricted to be a block.




```
if ( booleanTestExpression )
{
    consequentStatements
}
else
{
    alternativeStatements
}
```

Figure #@@. Execution paths of an `if/else` statement.

Executing an `if/else` statement works mostly like executing a simple `if` statement: First the boolean test expression is evaluated. If its value is true, the consequent statement is executed and the `if/else` statement is done. The difference occurs when the boolean test expression's value is false. In this case, the consequent is skipped (as it would be in the simple `if`) but the alternative statement is executed in its place. So in an `if/else` statement, exactly one of the consequent statement or the alternative statement is *always* executed. Which one depends on the value of the boolean test expression.

The following code might appear in the calculator's act() method, as described above. It is looking at which button is pressed, just like a good manager, and deciding which helper procedure should handle it.

```
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
}
else
{
    this.processOperatorButton( buttonID );
}
```

This code presumes some helper functions. The method `isDigitButton` verifies that the buttonID corresponds to the keys 0 through 9. The `process...` methods actually implement the appropriate responses to these button types.

Because there is only one test expression in this statement, it is always the case that at the single time of its evaluation (per if statement execution), it will be either true or false. If the test expression is true, the consequent statement will be executed (and the alternative skipped). If it is false, the alternative statement will be executed (and the consequent skipped). Exactly one of the consequent or the alternative will necessarily be executed each time that the if statement is executed.

**Cascaded Ifs**

The if/else statement is a special case of a more general situation. Sometimes, it is sufficient to consider one test and decide whether to perform the consequent or the alternative. But the example we gave of determining whether the buttonID was a digit or not probably isn't one. After all, a non-digit might be an operator, but it also might, for example, be an =. We probably need to check more than one condition, although we know if any one of these conditions is true, none of the others is. This is a perfect situation for a cascaded if statement.[Footnote: The test for `isDigitButton`, etc., may seem mysterious right now, and indeed we will simply assume the existence of these boolean-returning predicates for now. An implementation is provided in the section on Symbolic Constants, below, and discussed further in the chapter on Encapsulation.]

```
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
}
else
{

     if ( this.isOperatorButton( buttonID ) )
     {
         this.processOperatorButton( buttonID );
     }
     else
     {
         this.processEqualsButton( buttonID );
     }

}
```

In fact, the situation is really even more complex:

```
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
}
else
{

     if ( this.isOperatorButton( buttonID ) )
     {
```

```
            this.processOperatorButton( buttonID );
        }
        else
        {

             if ( this.isEqualsButton( buttonID ) )
             {
             this.processEqualsButton( buttonID );
             }
             else
             {

                  // and so on until...
                     throw new NoSuchButtonException( buttonID );

             }

        }

    }
```

These `if`s inside `else`s can get to be quite difficult to read, not to mention the pressure that they put on the right margin of your code as each subsequent `if` is further indented. [Footnote: The final lines of such a sequence also contain an awful lot of closing braces.] In order to avoid making your code too complex -- and too right-handed -- there is an alternate but entirely equivalent syntax, called the cascaded `if` statement. In this statement, an else clause may take an `if` statement directly, rather than inside a block. Further, the consequent block of this embedded `if` statement is lined up with the consequent block of the original `if` statement. So the example above would now read

```
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
}
else if ( this.isOperatorButton( buttonID ) )
{
    this.processOperatorButton( buttonID );
}
else if ( this.isEqualsButton( buttonID ) )
{
    this.processEqualsButton( buttonID );
}
   // and so on until...
else
{
    throw new NoSuchButtonException( buttonID );
}
```

Note that instead of ending with many close braces in sequence, a cascaded `if` statement ends with a single `else` clause (generally without an `if` and test expression) followed by a single closing brace.

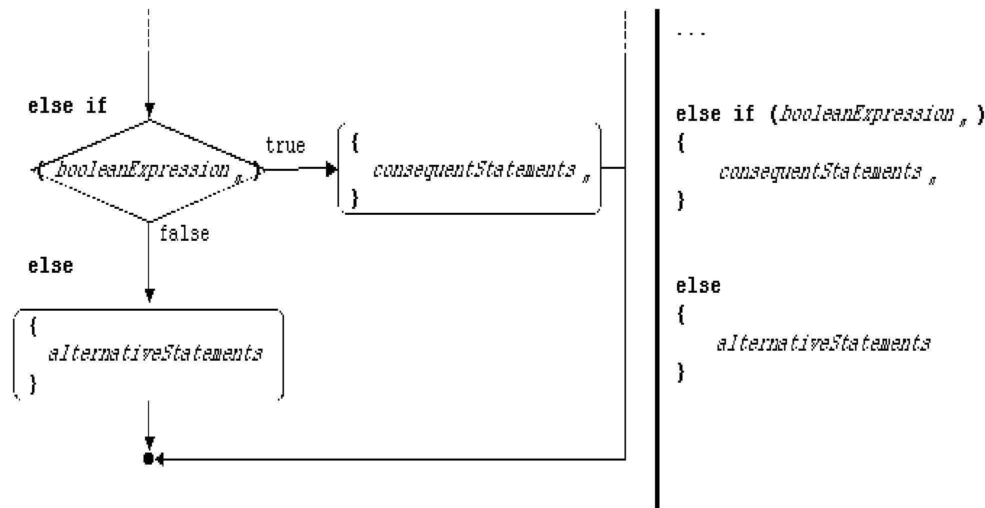



Figure #@@. Cascaded `if` statements.

Like a simple `if/else` statement, exactly one block of a cascaded `if` statement is executed. Once that block executes, the entire statement is finished. The difference is that if the first expression's value is false, the next condition is evaluated, and then the next, and so on, until either

- one test expression evaluates to true, in which case the corresponding body is executed and execution of the statement is then terminated, or
- an `else` without an `if` and test is reached, in which case the corresponding body is executed, or
- the end of the statement is reached, in which case its execution is complete.

Since an `else` with no `if` and test is always executed, such an `else` must be the last clause of the cascaded `if`.

### Many Alternatives

A conditional is a very general statement. With it, it is possible to write extremely convoluted programs. In order to make your program as easy to understand as possible, it is a good idea to keep your conditionals clean. A reasonable rule of thumb is that you should be able to explain the logic of your if statement easily to a friend. If you have to resort to pen and paper, your conditional expression may be too complex. If you have to write down more than two or three things, your conditional logic is most likely out of control.

For example, you should not test too many things simultaneously in one test expression. If you have a complex condition to test, use a boolean-returning method (a **predicate**) to keep the test expression simple. By naming the predicate appropriately, you can actually make your code much easier to read, as we did with isDigitButton and isOperatorButton, above. We will return to this point in the section on Procedural Abstraction in the chapter on Encapsulation.

As we have seen, you can embed if statements. In the example that we gave above, the embedded statements were actually mutually exclusive alternatives in the same set of tests: the button is either a digit

or an operator or the equals button or.... In this case, you should use the cascaded if syntax with which we replaced our embedded ifs.

But sometimes it is appropriate to embed conditionals. For example, in the calculator's act() method, inside the isOperatorButton block, we might further test whether the operation was addition or subtraction or multiplication or division.

```
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
}
else if ( this.isOperatorButton( buttonID ) )
{

     if ( this.isPlusButton( buttonID )
     {
         this.handlePlus();
     }
     else if ( this.isMinusButton( buttonID ) )
     {
         this.handleMinus();
     }
     else if ( this.isTimesButton( buttonID ) )
     {
         this.handleTimes();
     }
     else if ( this.isDivideButton( buttonID ) )
     {
         this.handleDivide();
     }
     else
     {
         throw new NoSuchOperatorException( buttonID );
     }

}
else if ( this.isEqualsButton( buttonID ) )
{
   // etc.
```

In this case, these further tests are a part of deciding how to respond to an operator button, including an operator-specific exception-generating clause. Note that the additional tests appear inside an if body, not inside an unconditional else. Using an embedded conditional to further refine a tested condition is a reasonable design strategy.

**B**eware of multiply evaluating an expression whose value might change. Instead, evaluate the expression once, assigning this value to a temporary variable whose value, once assigned, will not change between repeated evaluations.

The example above of looking out the window to check the weather may work well in southern California, but it is ill-advised in New England, where the weather has been known to change at the drop of a hat. Similarly, repeated invocation of a method returning the current time can be expected to produce different

values. So can repeated invocations of a Counting's getValue method. If we execute the following conditional

```
if ( theCounter.getValue() > 1 )
{
    Console.println( "My, there sure are a lot of them!" );
}
else if ( theCounter.getValue() == 1 )
{
    Console.println( "A partridge in a pear tree!" );
}
else if ( theCounter.getValue() = 0 )
{
    Console.println( "Not much, is it?" );
}
else if ( theCounter.getValue() < 0 )
{
    Console.println( "I'm feeling pretty negative" );
}
else
{
    Console.println( "Not too likely, is it?" );
}
```

it is possible that the counter will be incremented in just such a way that "Not too likely" might be printed.

**Q**. Describe how the process of executing this conditional might be intertwined with the incrementing of the counter to result in each of the five different values being printed. How might no value be printed?

# If Statement Syntax

An if statement consists of the following parts:

- The keyword if, followed by
- an expression of type boolean, enclosed in parentheses, followed by
- a (block) statement.

This may optionally be followed by an else clause. An else clause consists of the following parts:

- The keyword else, followed by either
- a (block) statement

  or

- the keyword if, followed by
- an expression of type boolean, enclosed in parentheses, followed by
- a (block) statement, optionally followed by
- another else clause.

Execution of the if statement proceeds as follows:

First, the test expression of the if is executed. If its value is true, the (block) statement immediately following this test is executed. When this completes, execution continues after the end of the entire if statement, i.e., after the final else clause body (if any).

If the value of the first if test is false, execution continues at the first else clause. If this else clause does not have an if and condition, its body (block) is executed and then the if statement terminates. If the else clause does have an if test, execution proceeds as though this if were the first test of the statement, i.e., at the beginning of the preceding paragraph.

## Limited Options: Switch

An if statement is a very general conditional. Often, the decision of what action to take depends largely or entirely on the value of a particular expression. For example, in the calculator, the decision as to what action to take when a user presses a button can be made based on the particular button pressed. What we really want to do is to see which of a set of known values (all of the calculator's buttons) matches the particular value (the actual button pressed). This situation is sometimes called a **dispatch on case**.

There is a special statement designed to handle just such a circumstance. In Java, this is a switch statement. A switch statement matches a particular expression against a list of known values.

Before we look at the switch statement itself, we need to look briefly at the list of known values. In a Java switch statement, these values must be **constant expressions**.

### Constant Values

When we are choosing from among a fixed set of options, we can represent those options using symbolic constants. A symbolic constant is a name associated with a fixed value. For example, it would be lovely to write code that referred to the calculator's `PLUS_BUTTON`, `TIMES_BUTTON`, etc. But what values would we give these names? For that matter, what is the type of the calculator's buttonID?

The answer is that it doesn't matter. At least, it doesn't matter as long as `PLUS_BUTTON` is distinct from `TIMES_BUTTON` and every other buttonID on the calculator. We don't want to add `PLUS_BUTTON` to `TIMES_BUTTON` and find out whether the value is greater or less than `EQUALS_BUTTON`, or to concatenate `PLUS_BUTTON` and `EQUALS_BUTTON`. But we do want to check whether `buttonID == PLUS_BUTTON`, and the value of this expression ought to be (guaranteed to be) different from the value of `buttonID == TIMES_BUTTON` (unless the value of `buttonID` has changed). Contrast this with a constant such as `Math.PI`, whose value is at least as important as its name.

These symbolic constants, then, must obey a simple contract. A particular symbolic constant must have the same value at all times (so that `EQUALS_BUTTON == EQUALS_BUTTON`, always), and its value must be distinct from that of other symbolic constants in the same group (`PLUS_BUTTON != EQUALS_BUTTON`). These are the ONLY guaranteed properties, other than the declared type of these names.

#### Symbolic Constants

It is common, though not strictly speaking necessary, to declare symbolic constants in a class or interface rather than on a per instance basis. It makes sense for them to appear in an interface when they form part of the contract that two objects use to interact. For example, you might communicate with me by passing me one of a fixed set of messages -- `MESSAGE_HELLO`, `MESSAGE_GOODBYE`, etc. -- and the interface might declare these constants as a part of defining the messages that we both are expected to understand and use. This means that these symbolic constants are declared `static`.

It makes sense that a name such as this, which is part of a contract, might be declared `public`. This allows it to be used by any objects that need to interact with the symbolic constant's declaring object. Symbolic constants like this need not be public, but they often are. (Private symbolic constants would be used only for internal purposes. Package-level or protected symbolic constants might be used in a restricted way.)

In Java, a name is declared `final` to indicate that its value cannot change. This is one of the properties that we want our symbolic constants to have: unchanging value. A value declared final cannot be modified, so you need not worry that extra visibility will allow another object to modify a constant inappropriately.

It is common, though somewhat arbitrary, to use `ints` for these constants. There are some advantages to this practice, and it does simplify accounting. For example, by defining a set of these constants in sequence one place in your code, it is relatively easy to keep track of which values have been used or to add new values.

```
public static final int ...
                         PLUS_BUTTON = 10,
                         MINUS_BUTTON = 11,
                         TIMES_BUTTON = 12,
                         ...
```

Of course, you should *never* depend on the particular value represented by a symbolic constant (such as `EQUALS_BUTTON`), since adding a new symbolic name to the list might cause renumbering. The particular value associated with such a name is not important.

So symbolic constants are often `public static final ints`.

# final

In Java, a name may be declared with the modifier `final`. This means that the value of that name, once assigned, cannot be changed. Such a name is, in effect, **constant**.

The most common use of this feature is in declaring final fields. These are object properties that represent constant values. Often, these fields are static as well as final, i.e., they belong to the class or interface object rather than to its instances. Static final fields are the only fields allowed in interfaces.

In addition to final fields, Java parameters and even local variables can be declared final. A final parameter is one whose value may not be changed during execution of the method, though its value may vary from one invocation of the method to the next. A final variable is one whose value is unchanged during its scope, i.e., until the end of the enclosing block. [Footnote: final fields and parameters are not strictly

speaking necessary unless you plan to use inner classes. They may, however allow additional efficiencies for the compiler or clarity for the reader of your code.]

Java methods may also be declared final. In this case, the method cannot be overridden in a subclass. Such methods can be inlined (i.e., made to execute with especially little overhead) by a sufficiently intelligent compiler.

Java classes declared final cannot be extended (or subclassed).

#### Using Constants

Properties such as the button identifiers are common to all instances of Calculators. In fact, they are reasonably understood as properties of the Calculator type rather than of any particular Calculator instance. They can (and should) be used in interactions between Calculator's implementors and its users. In general, symbolic names (and other constants) can be a part of the contract between users and implementors.

This means that it is often useful to declare these static final fields in an interface, i.e., in the specification of the type and its interactions. In fact, static final fields are allowed in interfaces for precisely this reason. Thus, the definition of interfaces in chapter 4 is incomplete: interfaces can contain (only) abstract methods *and* static final data members.

For example, the Calculator's interface might declare the button identifiers described above:

```
public interface Calculator
{

     public static final int PLUS_BUTTON = 10,
                             MINUS_BUTTON = 11,
                             TIMES_BUTTON = 12,
                             ...
                             EQUALS_BUTTON = 27;

}
```

Now any user of the Calculator interface can rely on these symbolic constants as a part of the Calculator contract. For example, the isOperatorButton predicate might be implemented as

```
public boolean isOperatorButton( int buttonID )
{
    return ( buttonID == PLUS_BUTTON )
           || ( buttonID == MINUS_BUTTON )
           || ( buttonID == TIMES_BUTTON )
           || ( buttonID == DIVIDE_BUTTON );
}
```

[Footnote: Note the absence of any explicit conditional statement here. Using an if to decide which boolean to return would be redundant when we already have boolean values provided by == and by ||. See the Sidebar on Using Booleans in the chapter on Statements.]

If we choose our numbering scheme carefully, the predicate isDigitButton could be implemented as

```
public boolean isDigitButton( int buttonID )
{
    return ( 0 <= buttonID ) && ( buttonID < 10 ) ;
}
```

Of course, this is taking advantage of the idea that the digit buttons would be represented by the corresponding ints. This is a legitimate thing to do, but ought to be carefully documented, both in the method's documentation and in the declaration of the symbolic constants:

```
/**
 *  Symbolic constants representing calculator button IDs.
 *  The values 0..9 are reserved for the digit buttons,
 *  which do not have symbolic name equivalents.
 */
   public static final int PLUS_BUTTON = 10,
                          MINUS_BUTTON = 11,
                          TIMES_BUTTON = 12,
                          ...
                          EQUALS_BUTTON = 27;
```

and

```
/**
 *  Assumes that the digit buttons 0..9 will be represented by
 *  the corresponding ints. These values should not be used for
 *  other buttonID constants.
 */
   public boolean isDigitButton( int buttonID )
{
    return ( 0 <= buttonID ) && ( buttonID < 10 ) ;
}
```

**Style Sidebar**

# Use Named Constants

A constant is a name associated with a fixed value. Constants come in two flavors: constants that are used for their value, and symbolic constants, used solely for their names and uniqueness. `Calculator.PLUS_BUTTON` (whose value is meaningless) is a symbolic constant, while Math.PI (whose value is essential to its utility) is not. But constants -- named values -- are a good idea whether the value matters or not.

Introducing a numeric literal into your code is generally a bad idea. One exception is 0, which is often used to test for the absence of something or to start off a counting loop. Another exception is 1 when it is used to increment a counter. But almost all other numeric literals are hard to understand. In these cases, it is good style to introduce a name that explains what purpose the number serves.

Numbers that appear from nowhere, with no explanation and without an associated name, are sometimes called magic numbers (because they appear by magic). Like magic, it is difficult to know what kind of

stability magic numbers afford. It is certainly harder to read and understand code that uses magic numbers.

In contrast, when you use a static final name, you give the reader of your code insight into what the value means. Contrast, for example, `EQUALS_BUTTON` vs. 27. You also decouple the actual value from its intended purpose. Code containing the name `EQUALS_BUTTON` would still work if `EQUALS_BUTTON` were initially assigned 28 instead of 27; it relies only on the facts that its value is unchanging and it is distinct from any other buttonID.

## Syntax

We turn now to a switch statement. A switch statement begins by evaluating the expression whose value is to be compared against the fixed set of possibilities. This expression is evaluated exactly once, at the beginning of the execution of the switch statement. Then, each possibility is compared until a match is found. If a match is found, "body" statements are executed. A switch statement may also contain a default case that always matches. In these ways, a switch statement is similar to, but not the same as, a traditional conditional.

#### Basic Form

A simple switch statement looks like this:

```
switch ( integralExpression )
{
    case integralConstant:
                               actionStatement;
                               break;
    case anotherIntegralConstant:
                               anotherActionStatement;
                               break;
}
```

To execute it, first the integralExpression is evaluated. Then, it is compared to the first integralConstant. If it matches, the first actionStatement is executed. If integralExpression doesn't match the first integralConstant, it is compared to anotherIntegralConstant instead. The result is to execute the first actionStatement whose integralConstant matches, then jumps to the end of the switch statement.

For example, we might implement the calculator's act method like this:

```
switch ( buttonID )
{
    case Calculator.PLUS_BUTTON:
                               this.handlePlus();
                               break;
    // ...
    case Calculator.EQUALS_BUTTON :
                               this.handleEquals();
                               break;
}
```

The presence of the break statements as the last statement of each set of actions is extremely important. They are not required in a switch statement, but without them the behavior of the switch statement is quite different. See the Switch Statement Sidebar for details.

# Break and Continue Statements

The `break` statement used here is actually more general that just its role in a `switch` statement.

A `break` statement is a general purpose statement that exits the innermost enclosing `switch`, `while`, `do`, or `for` block.

A variant form, the labelled `break` statement, exits all enclosing blocks until a matching label is found. A labelled `break` does not exit a method, however. The labelled form of the `break` statement looks like this:

```
label:

     blockStatementText
                {

          // body text
                         break label;
                         // more body text

     } endBlockStatementText
```

One or both of *`blockStatementText`* or *`endBlockStatementText`* may be present; for example, this block may be a while loop, in which case *`blockStatementText`* would be the code fragment `while ( expr )` and there would be no *`endBlockStatementText`*. [Footnote: The labelled block may be any statement containing a block, including a simple sequence statement. The body text may contain any statements, including -- in the case of a labelled `break` -- other blocks, so that a labelled break may exit multiple embedded blocks.]

This code is equivalent to [Footnote: Here, *`LabelBreakException`* is a unique exception type referring to this particular labelled break statement.]

```
try
{

     blockStatementText
                {

          // body text
                         throw new LabelBreakException();
                         // more body text

     } endBlockStatementText

}
catch ( LabelBreakException e )
```

```
{
}
```

That is, the labelled break statement causes execution to continue immediately after the end of the corresponding labelled block.

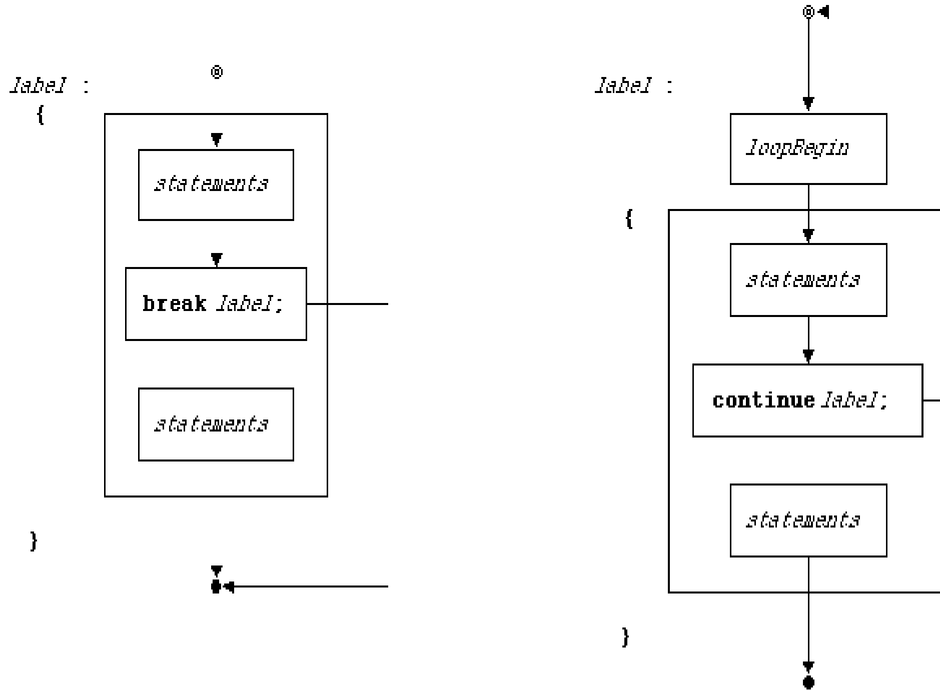



Figure #@@. Control flow diagrams for `break` and `continue` statements.

A similar statement, `continue`, also exists in unlabelled and labelled forms.

An unlabelled `continue` statement terminates the particular body execution of the (`while`, `do`, or `for`) loop it is executing and returns to the (increment and) test expression.

The labelled `continue` statement works similarly, except that it continues at the test expression of an enclosing labelled `while`, `do`, or `for` loop. The labelled `continue` statement

```
label:

    blockStatementText
                {

        // body text
                        continue label;
                        // more body text

    } endBlockStatementText
```

is equivalent to

```
blockStatementText
            {

      try
      {

           // body text
                                throw new LabelContinueException();
                                // more body text

      }
      catch ( LabelContinueException e )
      {
      }

} endBlockStatementText
```

**The Default Case**

In an `if` statement, if none of the test expressions evaluates to true, a final `else` clause without an `if` and test expression may be used as the default behavior of the statement. Such an `else` clause is always executed whenever it is reached.

In a `switch` statement, a similar effect can be achieved with a special case (without a comparison value) labelled `default`:

```
switch ( buttonID )
{
    case Calculator.PLUS_BUTTON:
                                this.handlePlus();
                                break;
    // ...
    case Calculator.EQUALS_BUTTON :
                                this.handleEquals();
                                break;
    default :
                                throw new NoSuchButtonException( buttonID );
}
```

If no preceding case matches the value of the test expression, the `default` will always match. It is therefore usual to make the `default` the final case test of the switch statement. (No case after the default will be tested.) When the `default` clause is the last statement of your `switch`, it is not strictly speaking necessary to end it with a `break` statement, though it is not a bad idea to leave it in anyway. The final `break;` statement is omitted in this example because it would never be reached after the throw. (Any instruction follower executing the throw would exit the switch statement at that point.)

It is often a good idea to include a default case, even if you believe that it is unreachable. You would be amazed at how often "impossible" circumstances arise in programs, usually because an implicit assumption is poorly documented or because a modification made to one part of the code has an unexpected effect on another.

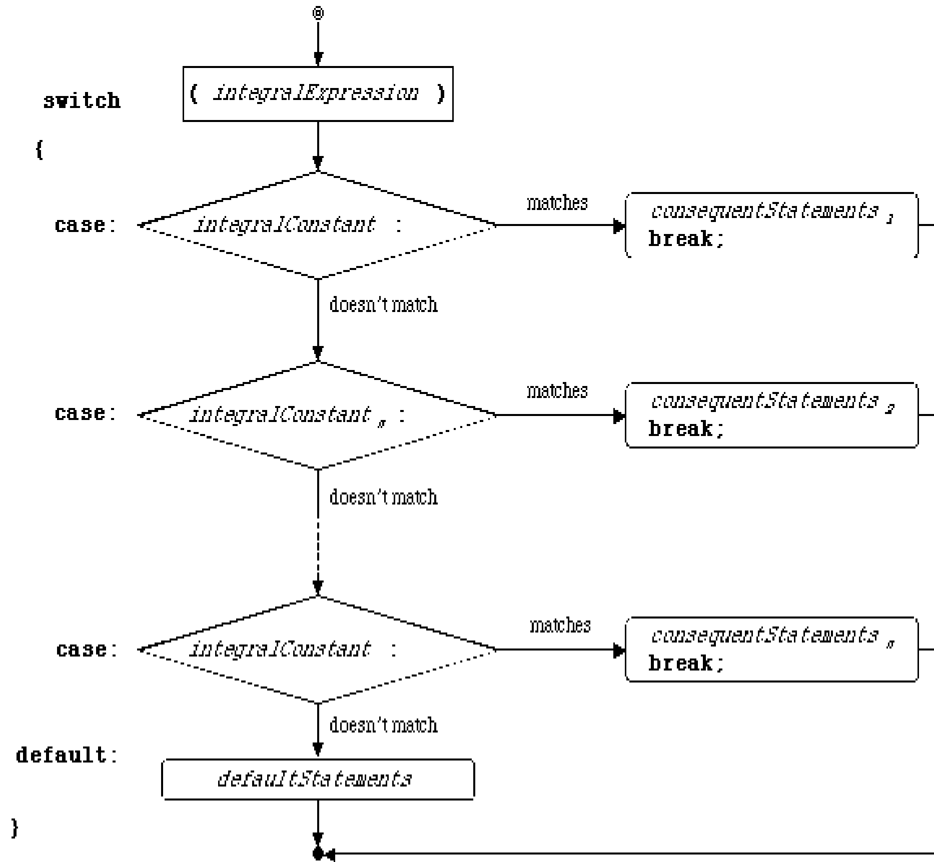



Figure #@@. Control flow diagram for a `switch` statement.

#### Variations

It is possible to write a switch statement without using breaks. In this case, when a case matches, not only its following statements but all statements within the switch and up to a break or the end of the switch statement will be executed. This can be useful when the action for one case is a subset of the action for a second case.

Beware of accidentally omitted `break` statements in a switch. Because omitting the break is sometimes what you want, it is legal Java and the compiler will not complain. Omitting a break statement will cause the statements of the following case(s) to be executed as well.

If two (or more) cases have the same behavior, you can write their cases consecutively and the same statements will be executed for both. This is, in effect, giving the first case no statements (and no break) and letting execution "drop through" to the statements for the second case. For example:

```
switch ( buttonID )
{
    case Calculator.PLUS_BUTTON:
    case Calculator.MINUS_BUTTON:
    case Calculator.TIMES_BUTTON:
    case Calculator.DIVIDED_BY_BUTTON:
```

```
                                        this.handleOperator( buttonID );
                                        break;
        // ....
```

In this case statement, the same action would be taken for each of the four operator types. The buttonID pressed is passed along to the operator handler to allow it to figure out which operator is needed.

**Switch Statement Pros and Cons**

A switch statement is very useful when dispatch is based on the value of an expression and the value is drawn from a known set of choices. The switch expression must be of an integral type and the comparison case values must be constants (i.e., literals or final names) rather than other variable names. When a switch statement is used, the switch expression is evaluated only once.

A switch statement cannot be used when the dispatch expression is of an object type or when it is a floating point number. It also cannot be used with a boolean, but since the boolean expression has only two possible values, an if statement with a single alternative makes at least as much sense in that case.

The requirement that a switch expression must be of integral type is one reason why `static final int`s are often used as symbolic constants. `int` is a convenient integral type and symbolic constants are naturally compatible with switch statements.

A switch statement cannot be used when the comparison values are variable or drawn from a non-fixed set. That is, if the dispatch expression must be compared against other things whose values may change, the switch statement is not appropriate. For example, you wouldn't want to use a switch statement to compare a number against the current ages of the employees of your company, because these are changing values.

The switch statement is also not appropriate for expressions that may take on any of a large range of values. ("Large" is subjective, but if you wouldn't want to write out all of the cases, that's a good indication that you don't want a switch statement.) For example, you wouldn't want to do a dispatch on a the title of a returned library book, testing it against every book name in the card catalog, even if you represented names as symbolic constants rather than as Strings.[Footnote: Of course, if you represented the names as Strings, you couldn't use a switch statement because String is an object type.]

## Switch Statement Syntax

A switch statement contains a test expression and at least one case clause. After that, the switch statement may contain any number of case clauses or statements in any order:

```
switch ( integralExpression )

    {
        caseClause
                    caseClauses or statements
                }
```

The *integralExpression* is any expression whose type is an integral type: `byte`, `short`, `int`, `long`, or `char`.

A *caseClause* may be either

```
case constantExpression :
```

or

```
default :
```

If the *caseClause* contains a *constantExpression*, this must be an expression of an integral type whose value is known at compile time. Such an expression is typically either a literal or a name declared `final`, although it may also be an expression combining other constant expressions (e.g., the product of a literal and a name declared `final`).

Note that each *caseClause* must end with a colon.

The embedded statements may be any statement type [!!?!].

Typically, the actual syntax of a switch statement is

```
switch ( integralExpression )
{

     caseClauses
                         statements ending with break;
                    caseClauses
                         statements ending with break;
     ...
     default :
          statements optionally ending with break;

}
```

where *caseClauses* is one or more case clauses.

## Arrays

Sometimes, what we really want to do when dispatching is to translate from one representation to another. For example, in constructing a Calculator, we might want to move from the symbolic constants used to identify buttons above to the actual labels appearing on those buttons. We might even want to move between the labels on buttons and the buttons themselves. If our collection of objects is indexed using an integral type -- either because it is naturally indexed or because we have used ints as symbolic constants -- we can often accomplish this conveniently using arrays.

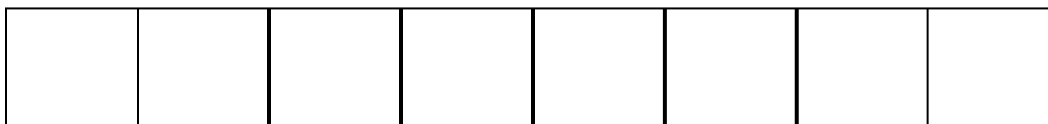


Figure #@@. An array is like a wall of numbered mailboxes.

**What is an Array?**

An array is an integrally indexed grouping of shoeboxes or labels. You can think of it sort-of like a wall full of numbered mailboxes. In identifying a mailbox, you need to use both a name corresponding to the whole group ("the mailboxes in the lobby") and an index specifying which one ("mailbox 37").Similarly, an array itself is a thing that can be named -- like the group of mailboxes -- and it has members -- individual mailboxes -- named using both the array name *and* the index, in combination. For example, my own particular individual mailbox might be named by `lobbyMailboxes[37]`.

An array has an associated type that specifies what kind of thing the individual names within the array can be used to refer to. This type is sometimes called the **base type** of the array. For example, you can have an array of chars or an array of Strings or an array of Buttons. The individual names within the array are all of the same type, say `char` or `String` or `Button`.

That is, an array is a collection of nearly-identical names, distinguished only by an int index. An array of shoebox-type--for example, an array of chars--really is almost like a set of mailboxes, each of which is an individual shoebox-name. To identify a particular shoebox, you give its mailbox number. For example, you can look and see what (char) is in mailbox 32 or put an appropriately typed thing (char) in mailbox 17. Label-type arrays work similarly, though it's hard to find an analogously appropriate analogy. (A set of dog-tags or post-it notes is along the right lines, but it is harder to visualize these as neatly lined up and numbered.) A label-type array -- such as an array of Buttons - is an indexed collection of labels suitable for affixing on things of the appropriate type -- such as Buttons. The names affixed on individual Buttons are names like `myButtons[8]`, the ninth button in my array.[Footnote: Yes, that's right, `myButtons[8]`, the ninth button. Array elements, like the characters in Strings, are numbered starting from 0.]

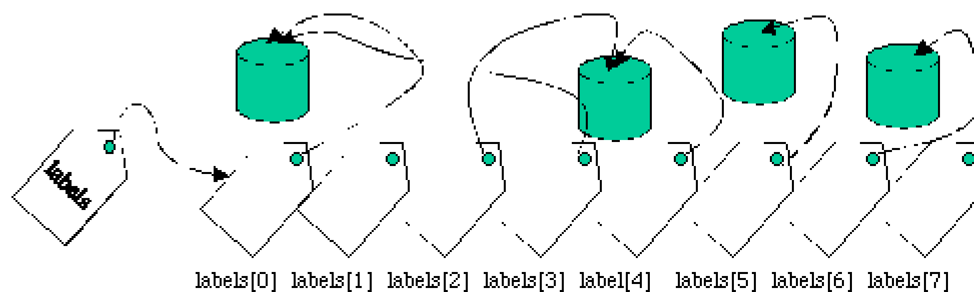



Figure #@@. This array, named `labels`, has eight elements, named `labels[0]` through `labels[7]`. Note the difference between what's attached to `labels` and what's attached to `labels[0]`.

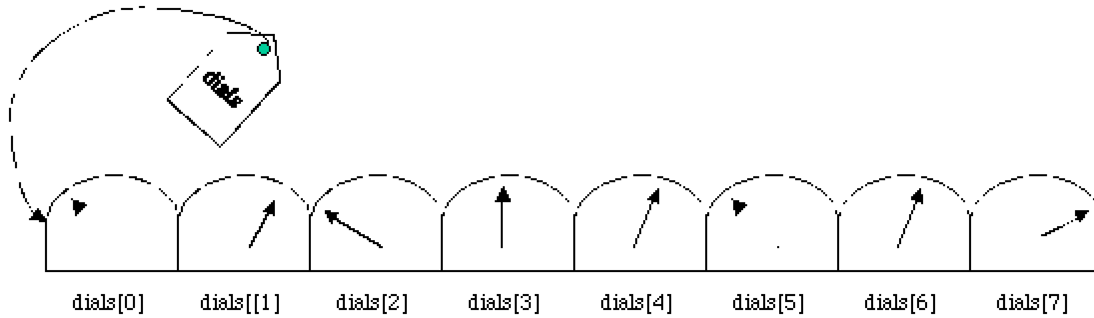



Figure #@@. Arrays of primitive (dial) types are similar, but not exactly the same as label-type arrays. This array, named `dials`, has eight elements, named `dials`[0] through `dials`[7]. Note the difference between the value of `dials` (which is actually a label!) and the value of `dials`[0].

**Array Declaration**

An array type is written just like the type it is intended to hold, followed by square braces. For example, the type of an array of chars is `char[ ]` and the type of an array of Buttons is `Button[ ]`. Note that, like `char` and `Button`, `char[ ]` and `Button[ ]` denote types, not actual Things. So, for example,

```
char[] initials;
```

makes the name `initials` suitable for sticking on things of type `char[ ]`; it doesn't *create* anything of type `char[ ]` or otherwise affix `initials` to some Thing. Similarly,

```
Button[] pushButtons;
```

creates a label, pushButtons, suitable for attaching to a `Button[ ]`, and nothing more. Note that both initials and pushButtons are *label* names, not*shoebox* names. The names of array types are always label types, although a particular array may itself be suitable either for holding shoebox (e.g., `char`) or label (e.g. `Button`) types.

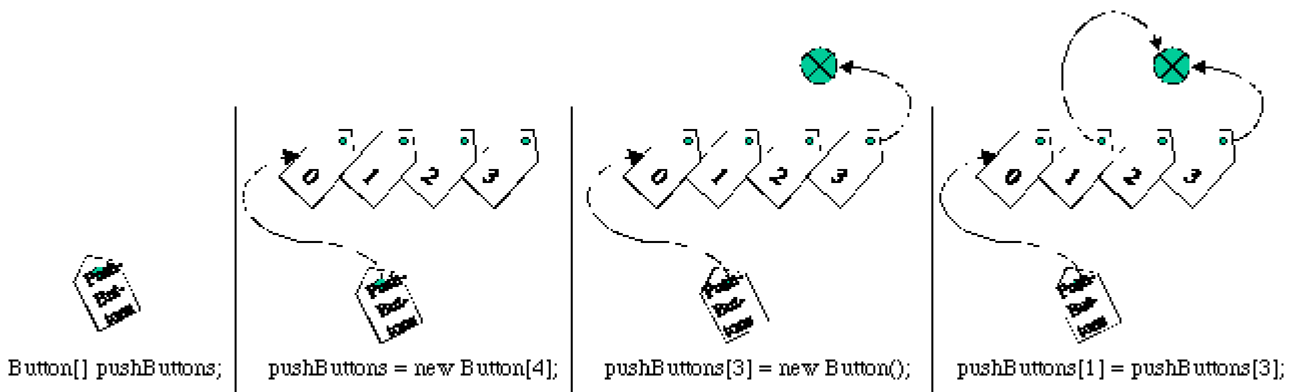



Figure #@@. Declaring, constructing, and using an array.

#### Array Construction

To actually create a `char[]` or `Button[]`,[Footnote: Pronounced "Button array" or "array of Buttons".] you need an array construction expression. This looks a bit like a class instantiation expression, but it is actually not quite the same. An array construction expression consists of the keyword `new` followed by the array type with an array size inside the square braces. For example,

```
new char[26]
```

is an expression that creates 26 char-sized mailboxes, numbered 0 through 25. Similarly,

```
new Button[ 518 ]
```

is an expression whose value is a brand new array of 518 Button-sized labels. Note that arrays are indexed starting at 0, so the last index of a member of this array will be 517, one less than the number supplied to the array construction expression. [Footnote: An array construction expression can be passed any expression with integral type (byte, short, int, long, or char) and its size and indexing will be set accordingly.]

The expression

```
pushButtons = new Button[ numButtons ]
```

makes the name `pushButtons` refer to a new array of `Button`-sized labels. How many? That depends on the value of `numButtons` at the time that this statement is executed.

The statement

```
String[] buttonLabels = new String[16];
```

combines all of these forms, creating a name (buttonLabels) suitable for labeling an array of Strings (String[]), constructing a 16-String array, and then attaching the name buttonLabels to that array. Note that the text String[] appears *twice* in this definition, once as the type and once (with an integral argument between the brackets) in the array construction expression.

#### Array Elements

To access a particular member of the array, you need an expression that refers to the array (such as its name), followed by the index of the particular member inside square braces. For example,

```
buttonLabels[2]
```

is an expression of type String that refers to the element at index 3 of the String array named by buttonLabels. Recall that, since the indices of buttonLabels run from 0 to 15, buttonLabels[2] is the *third* element of the array.

This expression behaves very much as though it were a name expression. Like a name, an array element expression of label type may be stuck on something, or may be null. An array element of shoebox type (e.g., initials[6]) behaves like a shoebox name.

You can use these array member expressions in any place you could use a name of the same type. So, for example, you can say any of the following things:

```
buttonLabels[2] = "Hi there";

String firstString = buttonLabels[0];

buttonLabels[7] = buttonLabels[6] + buttonLabels[5];

Console.println( buttonLabels[ Calculator.PLUS_BUTTON ] );

if ( buttonLabels[ currentIndex ] == null ) ...
```

(assuming of course that `Calculator.PLUS_BUTTON` and `currentIndex` are both int names).

# Array Syntax

#### Array Type

An array is a label name whose type is any Java type followed by []. The array is an array *of that type*. Admissible types include shoebox (primitive) types, label (object) types, and other array types. An array is declared like any other Java name, but using an array type. For example, if *`baseType`* is any Java type, then the following declaration creates a label, *`arrayName`*, suitable for affixing on an *array of* *`baseType`*:

```
baseType[] arrayName;
```

#### Array Initialization

By default, an array name's value is null. An array name may be defined at declaration time using an array literal. This consists of a sequence of comma-separated constant expressions enclosed in braces:

```
baseType[] arrayName = { const0, const1, ...  constN };
```

[Check this: literals only, or also symbolic constants? Can this be done w/non-String object types?]

#### Array Construction

Unless an array initialization expression is used in the declaration, an array must be constructed explicitly using the array construction expression

```
new baseType[ size ]
```

Here, *`baseType`* is the base type of the array (i.e., this expression constructs an *array of* *`baseType`*) and *`size`* is any non-negative integral expression.

#### Array Access

The expression `arrayName[index ]` behaves as a "regular" Java name. Its type is the array's base type.

Arrays are numbered from 0 to `arrayName.length - 1`. Attempting to access an array with an index outside this range throws an `ArrayOutOfBoundsException`.

**Manipulating Arrays**

The particular names associated with individual members of an array behave like ordinary (shoebox or label) names. What is unusual about them is how you write the name -- `arrayName[ index ]` -- and not any of how they actually behave.

You can find out how many elements are in a particular array with the expression `arrayName.length`. Note that there are no parentheses after the word length in this expression. Technically, this is not either a field access or a method invocation expression, although it looks like one and behaves like the other.

Note also that the value of the expression `arrayName.length` is *not* the index of the last element of the array. It is in fact one more than the final index of the array, because the array's indices start at 0. Attempting to access an array element with a name smaller than 0 or greater than or equal to its length is an error. In this case, Java will throw an ArrayOutOfBoundsException.

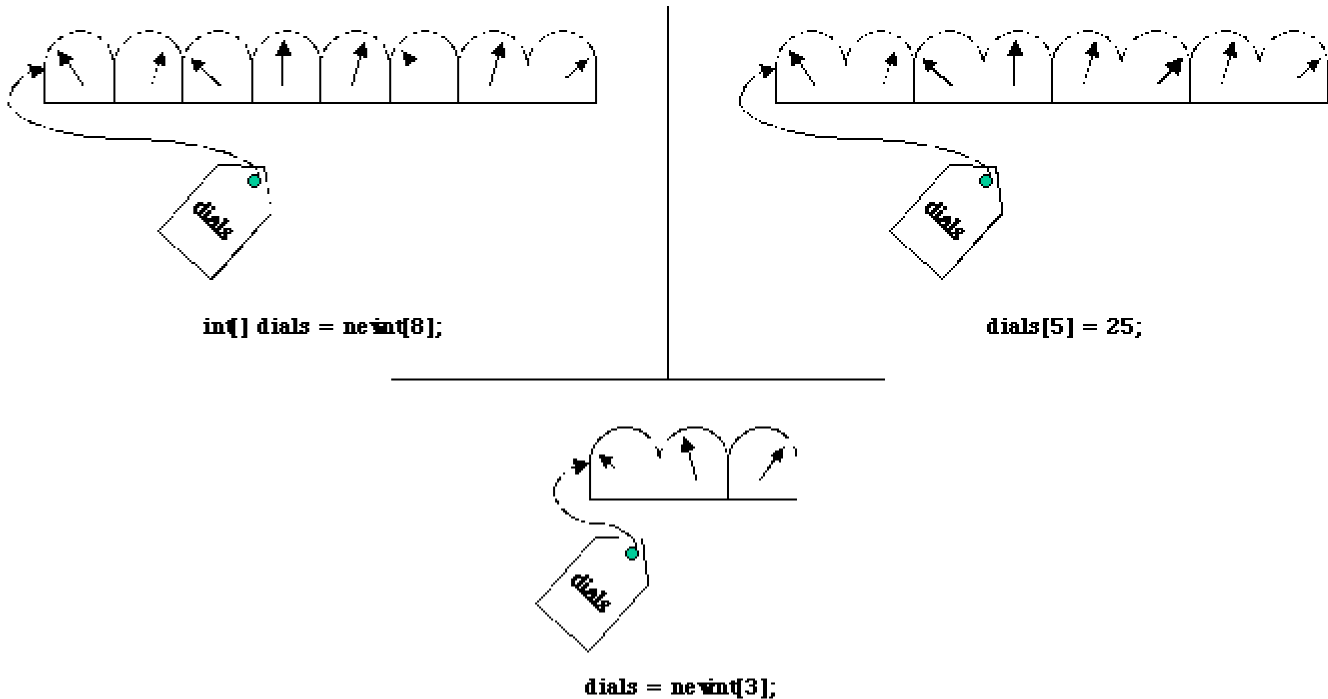



Figure #@@. In the upper left, an array of five `ints` is constructed (using `new int[5]`) and the label `dials` is attached to this array. In the upper right, the value of one of element of this array -- `dials`[5] -- is modified. In the bottom figure, the label `dials` is stuck on an entirely new array, this one of three `ints`.

Once you construct an array, the number of elements in that array does not change. However, this immutable value is the number of elements in the array itself, not the number of elements associated with the name. If the name is used to refer to a different array later, it may have a different set of legal indices. For example:

```
char[] firstInitials = new char[ 10 ];
   // firstInitials[3] would be legal, but firstInitials[12] would not.

firstInitials[ 5 ] = 'f';
firstInitials[ 5 ] = 'g';
   // changes the value associated with a particular mailbox

firstInitials = new char[ 2 ]
   // changes the whole set of mailboxes
// now pushButtons[3] isn't legal either!
```

#### Stepping through An Array Using a for Statement

One common use of arrays is as a way to step through a collection of objects. If you are going to work your way through the collection, one by one, it is common to do so using a counter and a loop.

We can write this with a while loop:

```
int index = 0;

while (index < array.length)
{
    // do something
       index = index + 1;
}
```

Note that `index` can't be initialized inside the while statement or it wouldn't be bound in the test expression. Local (variable) names have scope only from their declarations until the end of their enclosing blocks.

This is so common, there's a special statement for it. The while statement above can be replaced by

```
for (int index = 0; index < array.length; index = index + 1)
{
    // do something
   }
```

Note that the for loop also includes the declaration of index, but that index only has scope inside the for loop. It is as though index's definition plus the while loop were enclosed in a block.

For additional detail on for statements, refer to the sidebar.

# For Statement Syntax

The syntax

```
for ( initStatement; testExpression; incrementStatement )
{
    body
             }
```

is the same as

```
{

     initStatement;

     while ( testExpression )
     {
         body
                      incrementStatement;
     }

}
```

The expression *`testExpression`* is any single boolean expression. It falls within the scope of any declarations made in *`initStatement`*.

Both *initStatement* and *incrementStatement* are actually allowed to be multiple statements separated by commas:

e.g.

```
i = i + 1, j = j + i
```

Note that *initStatement*,*testExpression* and *incrementStatement* are separated by semicolons, but that individual statements within *initStatement* and *incrementStatement* are separated by commas. There is no semicolon at the end of *incrementStatement.*

**Using Arrays for Dispatch**

In addition to their use as collection objects, arrays can be used as a mechanism for dispatch. This is because the same variable can be used to index into multiple arrays or be passed to appropriate methods. We are not going to use an array to do the calculator's central dispatch job right now. Instead, we will consider the problem of constructing actual GUI Button objects that will appear on the screen. There should be one Button corresponding to each of the symbolic constants described above. Each of these Buttons will need an appropriate label, to be passed into the Button constructor. We might create a method,

```
String getLabel( int buttonID )
```

for this purpose.

We could use our getLabel to say

```
new Button( this.getLabel( buttonID ) )
```

or even

```
gui.add( new Button( this.getLabel( buttonID ) ) )
```

Such a getLabel method, which could translate from buttonIDs to labels, would also be useful for generating Strings suitable for printing to the Console, e.g., for debugging purposes.

One way to implement this method would be with an if statement. In this case, the body of the method might say:

```
if ( buttonID == Calculator.PLUS_BUTTON )
{
    return "+";
}
else if ( buttonID == Calculator.MINUS_BUTTON )
{
    return "-";
}
else if ( buttonID == Calculator.TIMES_BUTTON )
{
       // and so on....
```

Of course, this would get rather verbose rather quickly.

Because we are really doing a dispatch on the value of buttonID, and because we've cleverly chosen to implement these symbolic constants as ints, we could opt instead to use a switch statement:

```
switch ( buttonID )
{
    case Calculator.PLUS_BUTTON :
                    return "+";

    case Calculator.MINUS_BUTTON :
                    return "-";
       // and so on....
```

This may be somewhat shorter, but not much. It does have the advantage of making the dispatch on buttonID more explicit. But we can do still better.

**Q**. In the immediately preceding switch statement, why are there no break statements?

If we create an array containing the button labels, in order, corresponding to the buttonID symbolic constants, then we can use the buttonID to select the label:

```
String[] buttonLabels = { "0", "1", "2", "3", "4",
                          "5", "6", "7", "8", "9",
                          "+", "-", "*", "/",
                             // and so on...up to
                          "="};
```

In this case, the entire body of our getLabel method might say simply

```
return this.buttonLabels[ buttonID ];
```

This example is relatively simple, but in general arrays can be used whenever there is an association from an index set (such as the buttonIDs) to other values. The idea is that the index pulls out the correct information for that particular value. This is a very simple form of a very powerful idea, which we shall revisit in the chapter on Object Dispatch.

## When to Use Which Construct

Arrays are in many ways the most limited of the dispatch mechanisms. They work well when the action is uniform up to some integrally indexed decisions, e.g., some integrally indexed variables need to be supplied. Setting up the array appropriately allows for very concise code. This is not always possible, though, either because there isn't an obvious index set, because the index set is not integral, because it is not possible to set up the necessary association, or because the needed responses are nonuniform.

Switch statements also rely on integrally indexed decisions on a single expression, but they are otherwise quite general in the action(s) that can take place. They are useful any time the decision is made by testing the expression against a pre-known fixed set of constants. In other words, a switch statement can be used whenever an array is appropriate, though it may be more verbose. A switch statement can also be used in cases of nonuniform response, where an array would not be appropriate.

Ifs are very general. You can do anything with them. You should use them when none of the other mechanisms are appropriate.

In a subsequent chapter, we will see an additional dispatch mechanism, object dispatch, that resembles the implicit nature of array-based dispatch, but without many of its restrictions.

## Chapter Summary

- Dispatch is the process of deciding what action needs to be taken based on one's input. It is essentially a middle management function.
- Conditional statements are used when a piece of code should be executed under some but not all circumstances.
  - An if statement may consist only of a single boolean test expression and a body. This body is executed only if the test expression's value is true.
  - An if statement may optionally have an else clause with a body that is executed only when the if's test expression has the value false.
  - The else clause of an if statement may itself be an if statement. In this case, it is preferable to use cascaded rather than embedded ifs.
  - Each test expression is evaluated independently as it is reached.
- Numbers generally should not appear in code. Instead, use symbolic constants with descriptive names.
- A switch statement is used when different actions must be taken depending on the value of a single expression.
  - This expression is evaluated only once. Its type must be integral.
  - In a switch expression, the value is compared against different cases, which must be constants. Once a case matches, the statements of the switch body are executed until either a break or the end of the switch body is reached.
  - Switch has a specialized case, default, which always matches.
- An array is a uniformly typed collection of names.
  - The type of the array member names is the array's base type. The array member names may be either shoebox names or label names, depending on the base type.
  - The type of the array is "array of *base type*". The array name is a label name.
  - The names of array members are written using the array name followed by an integral index enclosed in square brackets.
  - The indices of an array run from 0 to `arrayName.length - 1`.
  - Like an object, an array must be explicitly created using new.

## Exercises

1. In the section entitled "Many Alternatives", there is an example of a counter whose getValue() method is invoked repeatedly.

a. Describe an execution sequence in which the value printed would be "My, there sure are a lot of them!"
2. Describe an execution sequence in which the value printed would be "A partridge in a pear tree!"
3. Describe how the process of executing this conditional might be intertwined with the incrementing of the counter to result in the printing of none of the messages.

of the five different values being printed.

2. Convert the following to a for loop:

```
int sum = 0;
int i = 1;
while ( i < MAXIMUM )
{
    sum = sum + i;
    i = i + 2;
}
```

3. Write a method that takes an array of ints and returns the sum of these ints.

4. Suppose that you have access to an array of StringTransformers, each of which has a method satisfying String transform( String ). Write a method, produceAllTransformations, that takes in a String and returns an array of Strings. The first element of the returned array should correspond to the transformation of the argument String by the first transformer, the second to the transformation of the argument String by the second transformer, and so on. You may assume that the name of the array of StringTransformers is `transformerFunctions`.

5. Consider the following code, excerpted from the definition of class EmotionalSpeaker.

```
public String transformEmotionally( Type emotion, String what )
{

     switch ( emotion )
     {
         case HAPPY:  return sayHappily( what );
         case SAD:    return saySadly( what );
         case ANGRY:  return sayAngrily( what );
     }

}
```

Where, e.g.,

```
private String sayHappily( String what )
{
    return "I'm so happy that ";
}
```

(You may assume similar definitions for the other emotions, with appropriate modifications.)

Define the symbolic constants HAPPY, SAD, and ANGRY, and provide a type for emotion.

6. In the previous exercise, the switch statement contains no breaks. What happens when we invoke `transformEmotionally( SAD, "I am here." )`?

7. Using an array, modify the code for transformEmotionally so that it fits in a single line. The array definition need not fit on that line.

**© 2003 Lynn Andrea Stein**

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>