Selection

Chapter 7
Chapter Contents

Objectives
7.1 Introductory Example: The Mascot Problem
7.2 Selection: The if Statement Revisited
7.3 Selection: The switch Statement
7.4 Selection: Conditional Expressions
7.5 Graphical/Internet Java: Event–Driven Programming

Part of the Picture: Boolean Logic and Digital Design
Part of the Picture: Computer Architecture
Chapter Objectives

- Expand concepts of selection begun in Chapter 4
- Examine the \texttt{if} statement in more detail
- Study the \texttt{switch} statement, multialternative selection
- Introduce conditional expressions
- Use event-driven program in GUls
- See Boolean expressions used to model logical circuits
- Look at architecture of computer systems
7.1 The Mascot Problem

We seek a method, mascot()
given name of a Big 10 university
returns the mascot

Objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Univ Name</td>
<td>String</td>
<td>varying</td>
<td>received</td>
<td>university</td>
</tr>
<tr>
<td>Mascot</td>
<td>String</td>
<td>varying</td>
<td>returned</td>
<td>none</td>
</tr>
</tbody>
</table>
Design

Class declaration

class Big10 {
    public static String mascot (String university) {
        ... }
}
Operations

- Compare *university* to "Illinois"; if equal, return "Fighting Illini"
- Compare *university* to "Indiana"; if equal return "Hoosiers"

... 

- An *if-else-if* ... structure can be used
Coding

- Note method source code, Figure 7.1 in text -- Driver program, Figure 7.2
- Note use of
  `school = theKeyboard.readLine()` instead of `.readWord()`
- `.readLine()` reads entire line of input, including blanks
- needed for schools like “Ohio State”
- Note also the final `else`
- returns an error message for a non Big-10 name
7.2 Selection: The *if* Statement Revisited

1. Single-branch
   
   ```
   if (Boolean_expression)
       statement
   ```

2. Dual-branch
   
   ```
   if (Boolean_expression)
       statement
   else
       statement
   ```

3. Multi-branch
   
   ```
   if (Boolean_expression)
       statement
   else if (Boolean_expression)
       statement . . .
   ```

Recall the three forms of the *if* statement from Chapter 4
Multibranch if

- The if–else–if is really of the form:
  ```java
  if (booleanExpression)
      statement1
  else
      statement2
  ```

- Where `statement2` is simply another `if` statement.

- Thus called a “nested” `if`
The Dangling-else Problem

Consider

```java
if (x > 0)
    if (y > 0)
        z = Math.sqrt(x) + Math.sqrt(y);
    else
        System.err.println("Cannot compute z");
```

Which if does the else go with?

In a nested if statement, an `else` is matched with the nearest preceding unmatched `if`
The Dangling-else Problem

What if we wish to force the else to go with the first if?

```java
if (x > 0)
  if (y > 0)
    z = Math.sqrt(x) + Math.sqrt(y);
  else
    System.err.println("Cannot compute z");
```

Enclose the second if statement in curly braces `{}`. The else must then associate with the outer if.
Using Relational Operators with Reference Types

Recall that reference types have "handles" that point to memory locations

```java
String s1 = new String("Hi");
    s2 = new String("Hi");
    s3 = s2;
```

Thus `s1 == s2` is false.

- they point to different locations in memory

But `s3 == s2` is true.

- they point to the same location in memory
Using Relational Operators with Reference Types

- When we wish to compare values instead of addresses
- use comparison methods provided by the classes

```java
if (s1.equals(s2))
    aScreen.println("strings are equal");
else
    aScreen.println("strings are different");
```
7.3 Selection: The `switch` Statement

- The `if-else-if` is a multialternative selection statement.
- The `switch` statement can be a more efficient alternative.
- Consider our Temperature class.
- The user may wish to specify which scale the temperature value to be displayed.
Object–Centered Design

Behavior:
- Program displays menu of possible conversions
- Read desired conversion from keyboard
- Prompt for temp, read temp from keyboard
- Display result

To convert temps, choose:
A. To Fahrenheit
B. To Celsius
C. To Kelvin
Q. Quit

Enter choice -> A
Enter temp -> _
## Problem Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Types</th>
<th>Kind</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sceeen</td>
<td>Screen</td>
<td>varying</td>
<td>theScreen</td>
</tr>
<tr>
<td>Menu</td>
<td>String</td>
<td>constant</td>
<td>MENU</td>
</tr>
<tr>
<td>Prompt</td>
<td>String</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>Conversion</td>
<td>char</td>
<td>varying</td>
<td>menuChoice</td>
</tr>
<tr>
<td>Keyboard</td>
<td>Keyboard</td>
<td>varying</td>
<td>theKeyboard</td>
</tr>
<tr>
<td>temperature</td>
<td>Temperature</td>
<td>varying</td>
<td>temp</td>
</tr>
<tr>
<td>result</td>
<td>Temperature</td>
<td>varying</td>
<td></td>
</tr>
</tbody>
</table>
Operations

1. Send `theScreen` messages to display `MENU` and a prompt
2. Send `temp` a message to read a Temperature from `theKeyboard`
3. Send `theKeyboard` a message to read a char and store it in `menuChoice`
4. Send `temp` the conversion message corresponding to `menuChoice`
Algorithm

Loop
1. Display MENU, read choice, terminate if choice == 'Q'
2. Prompt, receive input for temp
3. If menuChoice is 'A' or 'a'
   a. Send temp message to convert to Fahrenheit
   b. Tell theScreen to display result
      Otherwise if menuChoice is 'B' or 'b'
   c. Send temp a message to convert to Celsius
   d. Tell theScreen to display result
   ...
End Loop
Instead of if-else-if selection, use `switch`

```
switch(menuChoice) {
    case 'A': case 'a':
        theScreen.println( ... );
        break;
    case 'B': case 'b':
        theScreen.println( ... );
        break;
    case 'C': case 'c':
        ...
    default:
        System.err.println( ... );
}
```
The `switch` Statement

- Evaluated expression must be of type `char`, `byte`, `short`, or `int` (no `float` or `String`)
- Syntax in case list:
  ```java
  case constantValue:
  ```
  - type of `constantValue` must match evaluated expression
  - The `default` clause is optional
  - Once a `constantValue` is matched, execution proceeds until …
    - `break` statement
    - `return` statement
    - end of `switch` statement
The **break** statement

- Note each statement list in a **switch** statement usually ends with a **break** statement.
- This transfers control to the first statement following the **switch** statement.
- **Drop-through behavior**
  - If **break** is omitted, control drops through to the next statement list.
Example: Converting Numeric Codes to Names

We seek a method which receives a numeric code (1 – 5) for the year in college
returns the name of the year (Freshman, Sophomore, …, Graduate)
could be used in a class called AcademicYear
We use a switch statement to do this conversion
public static String academicYear(int yearCode)
{
    switch (yearCode) {
    case 1: return "Freshman";
    case 2: return "Sophomore";
    . . .
    default: System.err.println( ... );
             return;
    }
}
Cases with No Action

- Occasionally no action is required for specified values of the expression
- that feature not yet implemented
- that value simply meant to be ignored

In that situation

- insert the `break` or `return` statement after the case list constant
Choosing the Proper Selection Statement

*switch* statement preferred over *if-else-if* when all of the following occur:

1. equality `==` comparison is used
2. same expression (such as `menuChoice`) is compared for each condition
3. type of expression being compared is `char`, `byte`, `short`, or `int`
Examples:

- Consider a class called AcademicYear:
  ```java
  class AcademicYear
  {
      // constructor methods
      private String myName;
  }
  ```

- If a constructor takes an `int` parameter (1 - 5) to initialize `myName` use `switch`

- Another constructor might take a `String` parameter to initialize `myName` here we cannot use `switch`
7.4 Selection: Conditional Expressions

This is a trinary operator

It takes three operands

Syntax:

```
condition ? expression1 : expression2
```

Where:

- `condition` is a Boolean expression
- `expression1` and `expression2` are of compatible types
Example:

To return the larger of two numbers:

```java
public static largerOf(int v1, int v2 )
{
    return ( ( v1 > v2) ? v1 : v2);
}
```
Traditional programming consists of:
- Input
- Processing
- Output

GUI programs act differently
- They respond to different events
  - mouse clicks, dragging
  - keys pressed on keyboard
- Hence it is called “event driven” programming
Example: A GUI Big-10-Mascot Program

**Behavior**
- Construct window with prompt for university name
- User enters name in a text field
- Program responds with proper mascot or error message

<table>
<thead>
<tr>
<th>University</th>
<th>Mascot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ohio State</td>
<td>Buckeyes</td>
</tr>
</tbody>
</table>
GUI Design Principle

- Only show the user what he **needs** to see.
- Note that the label “Mascot” and the text field with the mascot do not appear until the name of the university is entered.
- Otherwise the user might think they can enter the mascot and get the univ.
### Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>The program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A window</td>
<td>varying</td>
<td></td>
<td>aGUI</td>
</tr>
<tr>
<td>Prompt for univ</td>
<td>JLabel</td>
<td>constant</td>
<td>mySchoolLabel</td>
</tr>
<tr>
<td>First text field</td>
<td>JTextField</td>
<td>varying</td>
<td>mySchoolField</td>
</tr>
<tr>
<td>Big-10 name</td>
<td>String</td>
<td>varying</td>
<td>school</td>
</tr>
<tr>
<td>Mascot label</td>
<td>JLabel</td>
<td>constant</td>
<td>myMascotLabel</td>
</tr>
<tr>
<td>Second text field</td>
<td>JTextField</td>
<td>varying</td>
<td>myMascotField</td>
</tr>
<tr>
<td>A mascot</td>
<td>String</td>
<td>varying</td>
<td>mascot</td>
</tr>
</tbody>
</table>
Operations

1. Construct GUI to do following
   - Display window frame
   - Position `JLabel` (prompt, mascot label)
   - Position `JTextField` fields (univ, mascot)
   - Set title of window frame

2. When user enters something in univ. text field
   - Get text from `JTextField` (university name)
   - Set text in `JTextField` (mascot name)
   - Make `JLabel` (mascot-label) disappear
   - Make `JTextField` (univ name) disappear
   - Select between 2b, 2c, and 2d, based on result of 2a
Coding and Testing

- Note source code in Figure 7.7 in text
- Note testing
- Application provides continuous behavior
  - program does not terminate until user clicks on window close box
- Accomplished by using an event-processing loop
  - Get event
  - If event is terminate, terminate repetition
  - Process the event
Java's Event Model

- Building an event delegation model
- Define the event source(s)
- Define the event listener(s)
- Register a listener with each source that listener handles the events generated by that source
Event Sources

Define an event-generating component in the GUI usually in its constructor.

Example is a `JTextfield`:

```java
mySchoolField = new JTextField (14);
```

A `JTextfield` “fires events” – it is an event source.
Java's Interface Mechanism

Note declaration:
```java
class GUIBig10Mascots extends CloseableFrame
    implements ActionListener
{
    // ...
}
```

Note the `extends CloseableFrame`
- inherits all its instance fields & methods

Note the `implements ActionListener`
- this is not a class, it is an interface
- contains only method headings, prototypes
Java's Interface Mechanism

- A class that implements an interface must provide a definition for each method whose heading is in the interface.
- Interface objects cannot be created with `new`.
- When an interface is implemented, we can:
  - create interface handles
  - send an interface message to an object referred to by the handle.
Event Listeners

To have a GUI respond to events
- Create a listener for that event source
- Register the listener with that event source

In our example, when the main method creates a `GUIBig10Mascots` object, it also creates
- a `CloseableFrame` object is specified by the constructor
- An ActionEventListener object
Registering Event Listeners with Event Sources

- Action event sources provide an addActionListener() method

- In GUIBig10Mascots constructor we have
  `mySchoolField.addActionListener(this);`

  - this refers to the object being constructed
  - the object registers itself as an ActionListener

- Now the listener has been bound to the event source
Usefulness of Interfaces

- A `JTextField` object has a `listenerList` field.

- The `addActionListener()` method adds an `ActionListener` handle to this list.
Handling an Event

Enter key pressed in the JTextField
an ActionEvent is built
sent to listener via actionPerformed() message

Enter actionPerformed( anEvent )
{ ... }
this
AnActionEvent
GUIBigTenMascots object
JTextField object
listenerList
Constructor for GUI Application

1. Create components & listeners, register listeners with those that fire events
2. Create `JPanel` for components
3. Tell `JPanel` which layout manager to use
4. Mount components on `JPanel` usually using the `add()` method
5. Make `JPanel` the content panel of window frame
Sample layout managers:

- `BorderLayout()` – components added at compass positions
- `BoxLayout()` – components added in horizontal or vertical box
- `FlowLayout()` – components added L→R, Top→Bottom
- `GridLayout(m,n)` – components added L→R, Top→Bottom in a grid of m by n equal sized cells
Inside the `actionPerformed()` Method

- This method is invoked when an `ActionEvent` source fires an `ActionEvent`.
- The `ActionEvent` class must have been specified as the listener.
- The method must specify what to do when the event occurs.

**Big10Mascot** example:

- Evaluate string in `myMascotField`.
- The string could be empty, valid, or invalid.
- Respond accordingly.
Big10 Mascot
An Applet Version

1. Make the class extend `JApplet` instead of `CloseableFrame`.
   ```java
   public class GUIBig10Mascots2 extends JApplet
   implements ActionListener
   ```

2. Change the `main()` method to a non-static `init()` method.
   ```java
   public void init (String [] args)
   {
       ... 
   }
   ```
Example 2: GUI Temperature Converter Application

- GUIBig10Mascots had single source of ActionEvents
- GUITemperatureConverter lets user enter any one of three types of temperatures
- Note source code, Figure 7.8
GUI Temperature Converter

- Constructor method builds the GUI
- `get-source()` message takes `ActionEvent` as argument
  - returns the event source (as an object) that fired the event
- `action-performed()` casts object into `JTextField`
  - `JTextField` messages can be sent to it
  - also checks for the object's type with `instanceof` operator
GUI Temperature Converter

- Note use of *if-else-if* statement using the *equals()* method
- determines which *JTextField* is source of event
- Then the equivalent values in the other two fields are displayed

<table>
<thead>
<tr>
<th>Temperature Converter</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.0</td>
</tr>
<tr>
<td>0.0</td>
</tr>
<tr>
<td>273.15</td>
</tr>
</tbody>
</table>
Applet Version of Temperature Converter Program

- Class extends `JApplet` instead of `CloseableFrame`
- Replace `main()` with non-static `init()`
- Remove the call to `setTitle()`
- Set dimensions of the applet frame in the HTML file:

```
32.0 Fahrenheit
273.15 Kelvin
0.0 Celsius
```

AppletViewer: GUITemperatureCon...

Applet
Applet started.

Table:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fahrenheit</td>
<td>32.0</td>
</tr>
<tr>
<td>Celsius</td>
<td>0.0</td>
</tr>
<tr>
<td>Kelvin</td>
<td>273.15</td>
</tr>
</tbody>
</table>
Conclusions

Compare and contrast the textual application versions and GUI versions

Design principle: Objects and their user interfaces should be kept separate

Note that the Temperature class was used for both versions
Arithmetic operations performed by the CPU carried out by logic circuits

Logic circuits implement Boolean (digital) logic in hardware
Early Work

- Foundations for circuit design
- English mathematician, George Boole
- Early 1900s
- Basic axioms of Boolean algebra seen in computer language Boolean expressions

One of more useful axioms is DeMorgan's law:

\[
! (x \&\& y) = (\neg x \mid\mid \neg y) \\
! (x \mid\mid y) = (\neg x \&\& \neg y)
\]

helps simplify complicated Boolean expressions
Use three basic electronic components which mimic logical operators:

- AND gate
- OR gate
- NOT gate (inverter)
Circuit Design: A Binary Half-Adder

Truth table

<table>
<thead>
<tr>
<th>digit1</th>
<th>digit2</th>
<th>sum</th>
<th>carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean expression equivalent:

Boolean carry = digit1 && digit2,
sum = (digit1 || digit2) && !(digit1 && &digit2);
A Binary Half-Adder

Digital circuit equivalent:

Note binary half-adder class, source code, Figure 7.9, test driver Figure 7.10
Four main structural elements of a computer:
- Processor: controls operation, performs data processing
- Main Memory: stores data and program, it is volatile
- I/O Modules: move data between computer and external environment
- System Interconnection: provides communication among processors, memory, I/O devices
Processor Registers

- Provide memory that is faster and smaller

Functions:
- enable assembly-language programmer to minimize main memory references
- provide the processor with capability to control and monitor status of the system
User–Visible Registers

- Data registers
  - some general purpose
  - some may be dedicated to floating-point operations

- Address registers
  - index register
  - segment pointer
  - stack pointer
Control and Status Registers

- Program Status Word (PSW) contains:
  - sign of the last arithmetic operation
  - zero – set when result of an arithmetic operation is zero
  - carry – set when operation results in carry or borrow
  - equal – set if logical compare result is equality
  - overflow
  - interrupt enable/disable
  - supervisor – indicates whether processor is in supervisor or user mode
Instruction Execution

- Processor reads instructions from memory
- Program counter keeps track of which instruction is to be read
- Instruction loaded into instruction register

Categories of instructions
- Move data to/from memory and processor
- Move data to/from I/O devices and processor
- Perform data processing (arithmetic, logic)
- Alter sequence of execution (loop, branch, jump)
I/O Function

- I/O modules can exchange data directly with processor
- Disk controllers have memory locations to be accessed
- I/O modules may be granted authority to read/write directly from/to memory
- This frees up processor to do other things
Memory Hierarchy

- Design constraints
  - how much?
  - how fast?
  - how expensive?
- Relationships:
  - faster access time, greater cost per bit
  - greater capacity, smaller cost per bit
  - greater capacity, greater (slower) access time
Solution:
- do not rely on a single memory component or technology
- employ memory hierarchy

As we go down the hierarchy:
- Decrease cost/bit
- Increase capacity
- Increase access time
- Decreasing frequency of access by processor
I/O Organization

- I/O modules interface to system bus
- More than just a mechanical connection
- Contains "intelligence" or logic

Major functions

- Interface to processor and memory via system bus
- Interface to one or more external devices
I/O Module Function

Categories of I/O module functions:
- Control and timing
- Communication with processor
- Communication with external device
- Data buffering
- Error detection
Typical sequence of steps when processor wants to read an I/O device:
1. Processor interrogates module for status of a peripheral
2. I/O module returns status
3. Processor requests transfer of data
4. Module gets byte (or word) of data from external device
5. Module transfers data to processor
I/O Module Communication with Processor

- Receive and decode commands
- Exchange data between processor and module via data bus
- Report status – I/O devices are slow, module lets processor know when it is ready
- Address recognition – recognizes addresses of peripherals it controls
Data Buffering

- Contrast transfer rate of data to/from main memory is high
- to/from peripheral devices low
- Data buffered in I/O module
- data moved to/from processor much faster
Error Detection

- Detect and report errors in I/O process
- Mechanical, electrical malfunctions in the device
  - Floppy disk not fully inserted
  - Paper jam, out of paper in printer
- Invalid data transmission (found by parity check, etc.)
  - 8th bit of a byte used as a check bit for the other 7 bits