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 `if` statement in more detail
- Study the `switch` statement, multialternative selection
- Introduce conditional expressions
- Use event-driven program in GUIs
- 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:

Object	Type	Kind	Movement	Name
Univ Name	`String`	varying	received	`university`
Mascot	`String`	varying	returned	none

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

```
if (booleanExpression)
      statement1
else
      statement2
```

- Where `statement2` is simply another `if` statement
- Thus called a “nested” `if`

The Dangling-`else` Problem

- Consider

```
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 <u>nearest preceding unmatched</u> `if`

The Dangling-`else` Problem

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

```
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

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

- Thus `s1 == s2` is **`false`**.
  - they point to <u>different locations in memory</u>
- But `s3 == s2` is **`true`**.
  - they point to the <u>same</u> location in memory

Using Relational Operators with Reference Types

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

```
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
  - 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

Objects	Types	Kind	Name
Program			
Sceeen	`Screen`	varying	`theScreen`
Menu	`String`	constant	`MENU`
Prompt	`String`	constant	
Conversion	`char`	varying	`menuChoice`
Keyboard	`Keyboard`	varying	`theKeyboard`
temperature	`Temperature`	varying	`temp`
result	`Temperature`	varying	

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

Coding

- 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( … );
}
```

Expression evaluated

Value of expression searched for in case-list constants

If match found, statement(s) executed

If NOT found, default clause executed

The `switch` Statement

- Evaluated expression must be of type `char`, `byte`, `short`, or `int` (no `float` or `String`)
- Syntax in case list:
  `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 <u>usually</u> ends with a `break` statement
  - this transfers control to 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

Year-code Conversion Method

```
public static String academicYear
                     (int yearCode)
 {
 switch (yearCode) {
 case 1: return "Freshman";
 case 2: return "Sophomore";
   . . .
 default: System.err.println( … );
          return;
 }
}
```

Note source code for method and test driver, Figures 7.4, 7.5

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 <u>preferred</u> 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:

```
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 <u>three</u> 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:

```
public static largerOf(int v1, int v2 )
{
  return ( ( v1 > v2) ? v1 : v2);
}
```

Condition

Value returned if condition is true

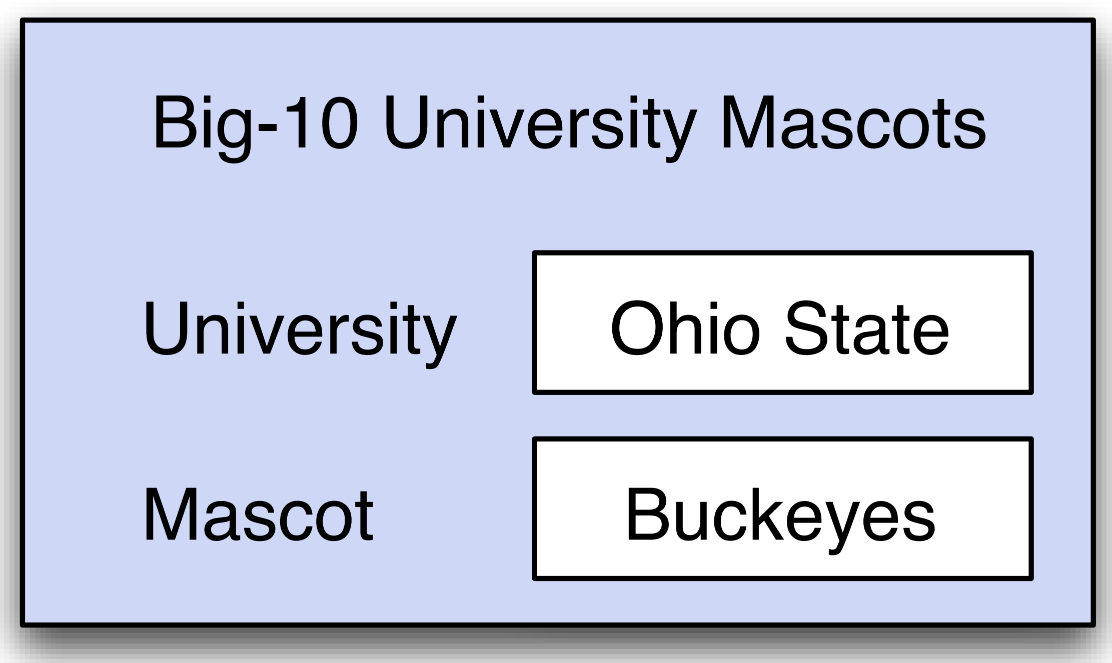
Value returned if condition is false

7.5 Graphical Internet Java: Event-Driven Programming

- 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

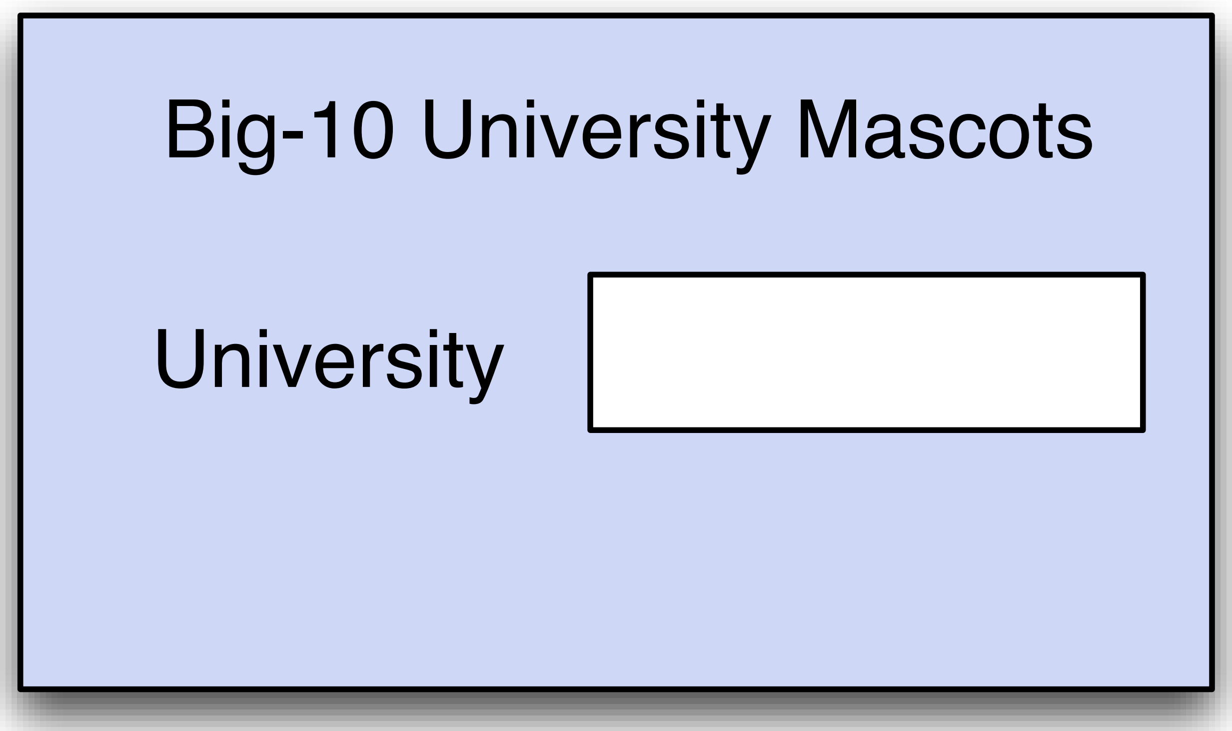
Example: A GUI Big-10-Mascot Program

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



GUI Design Principle

- Only show the user what he <u>needs</u> 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

Objects	Type	Kind	Name
The program			
A window		varying	`aGUI`
Prompt for univ	`JLabel`	constant	`mySchoolLabel`
First text field	`JTextField`	varying	`mySchoolField`
Big-10 name	`String`	varying	`school`
Mascot label	`JLabel`	constant	`myMascotLabel`
Second text field	`JTextField`	varying	`myMascotField`
A mascot	`String`	varying	`mascot`

Operations

1. Construct GUI to do following
   - Display window frame
   - Position `JLabel` (prompt, mascot label)
   - Position `JText` 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 <u>continuous behavior</u>
  - program does not terminate until user clicks on window close box
- Accomplished by using an <u>event-processing loop</u>
  - Get event
  - If event is terminate, terminate repetition
  - Process the event

Java's Event Model

- Building an event delegation model
- Define the event <u>source(s)</u>
- Define the event <u>listener(s)</u>
- <u>Register</u> 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`**
    `mySchoolField = new JTextField (14);`
  - a **`JTextfield`** "fires events" – it is an event source

Java's Interface Mechanism

- Note declaration:

```
class GUIBig10Mascots extends
CloseableFrame
             implements ActionListener
{   .   .   .   }
```

- Note the **`extends ClosableFrame`**
  - inherits all its instance fields & methods
- Note the **`implements ActionListener`**
  - this is not a class, it is an <u>interface</u>
  - contains only <u>method headings</u>, prototypes

Java's Interface Mechanism

- A class that implements an interface
  - must provide a <u>definition</u> for each method whose heading is in the interface
- Interface objects <u>cannot</u> 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

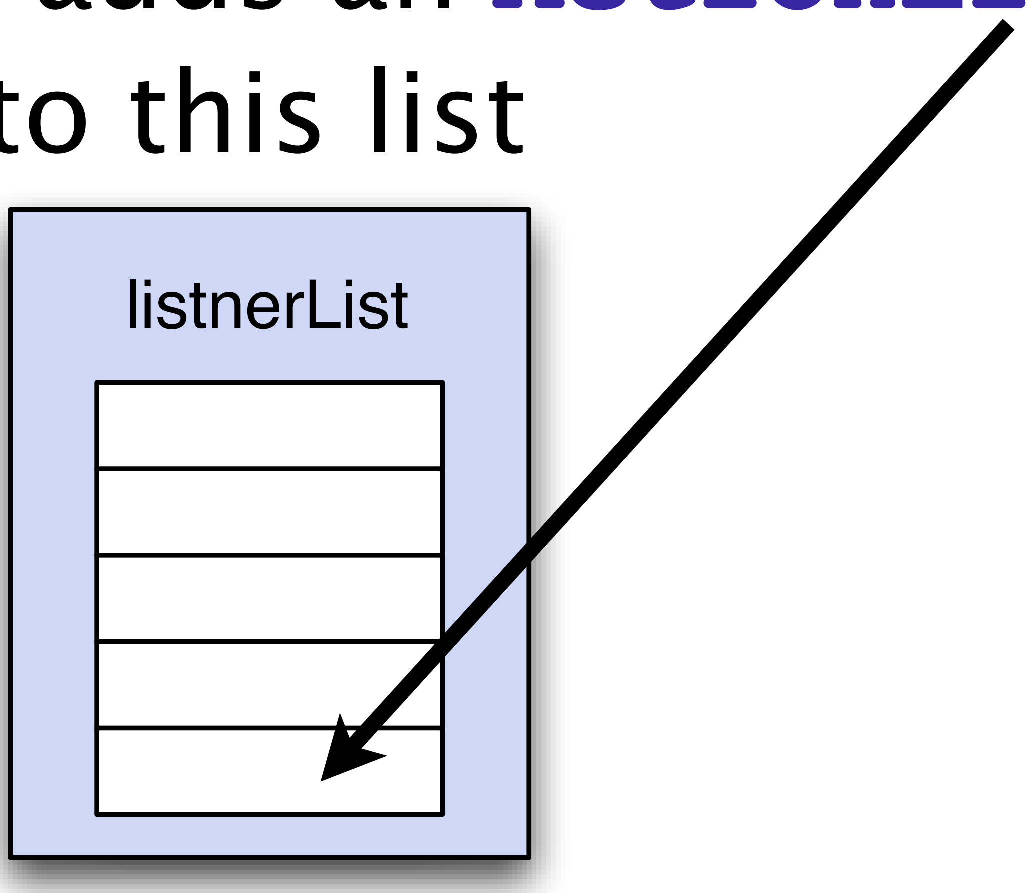
- To have a GUI respond to events
  - Create a listener for that event source
  - Register the listener with that event source
- In our example, when the main method creates a `GUIBig10Mascots` object, it <u>also creates</u>
  - a CloseableFrame object is specified by the constructor
  - An ActionListener 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 <u>bound</u> to the event source

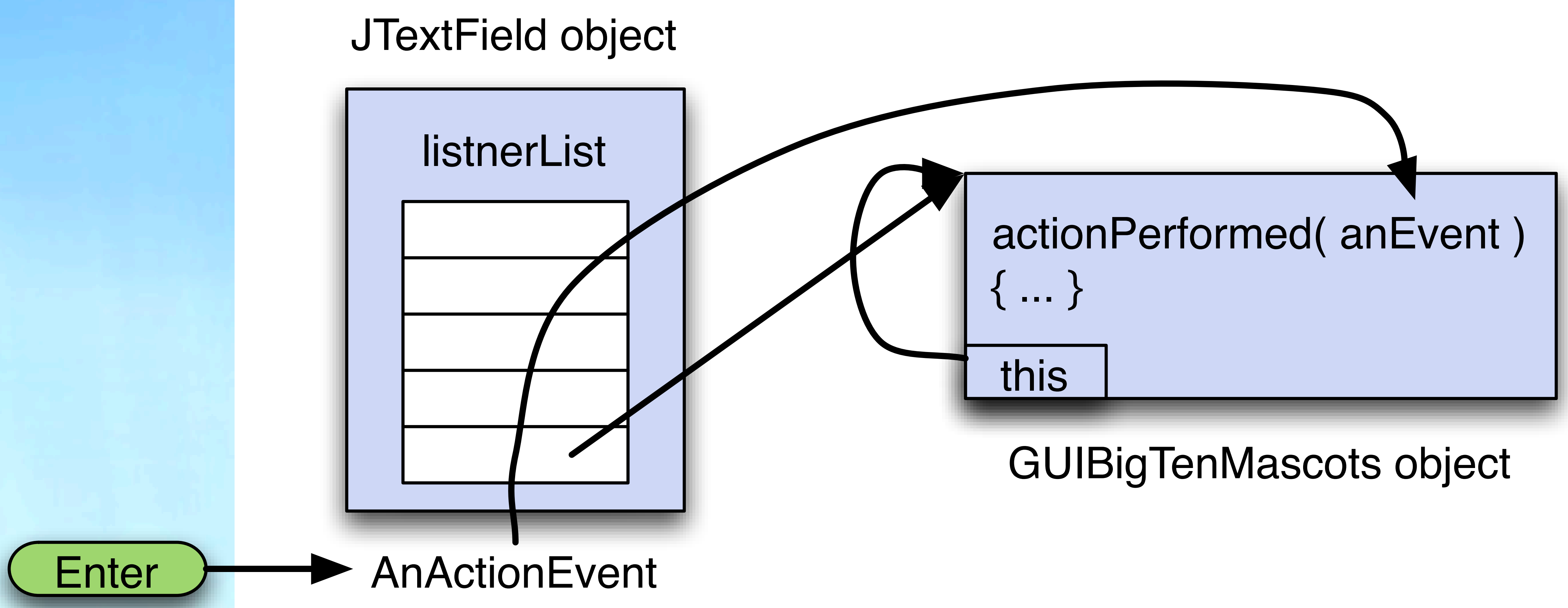
Usefulness of Interfaces

- 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



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

Layout Managers

- 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 invoked when `ActionEvent` source fires **`ActionEvent`**
  - class must have been specified as the listener
- Method must specify what to do when the event occurs
- `Big10Mascot` example:
  - evaluate string in `myMascotField`
  - could be empty, valid, or invalid
  - respond accordingly

Big10 Mascot
An Applet Version

- Make the class extend `JApplet` instead of `CloseableFrame`

```
public class GUIBig10Mascots2 extends JApplet
                    implements ActionListener
```

- Change the `main()` method to a non-static `init()` method

```
public void init (String [] args)
{ … }
```

Example 2: GUI Temperature Converter Application

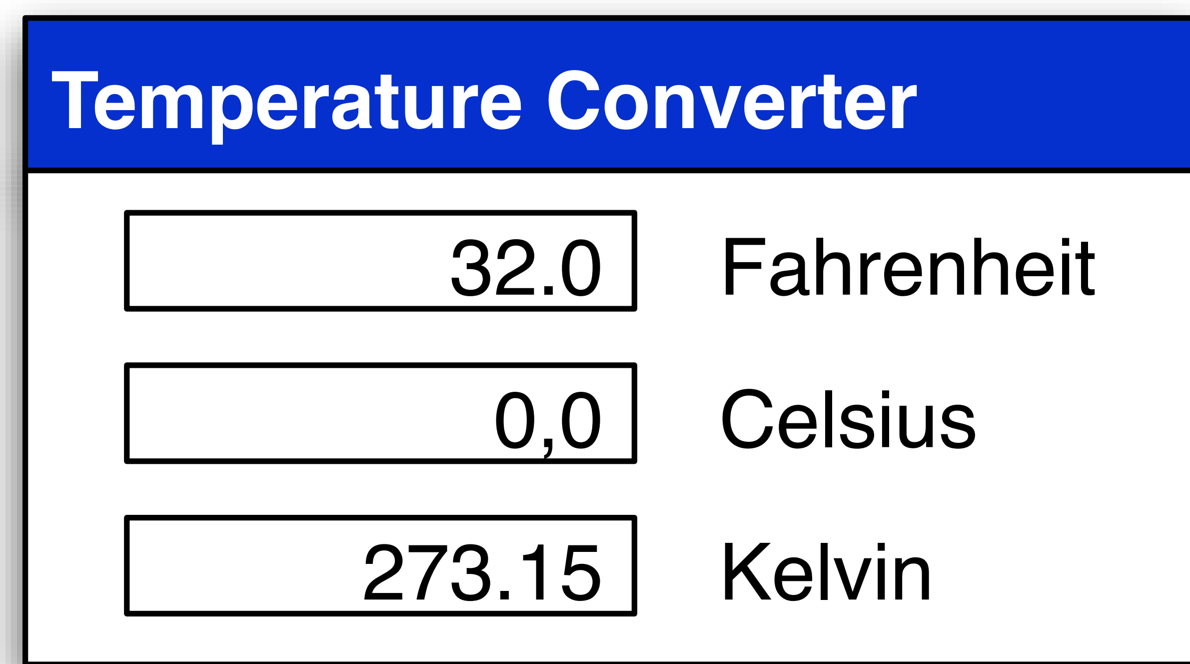
- `GUIBig10Mascots` had single source of ActionEvents
- `GUITemperatureConverter` lets user enter any one of three types of temperatures
- Note source code, Figure 7.8

GUI Temperature Converter

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

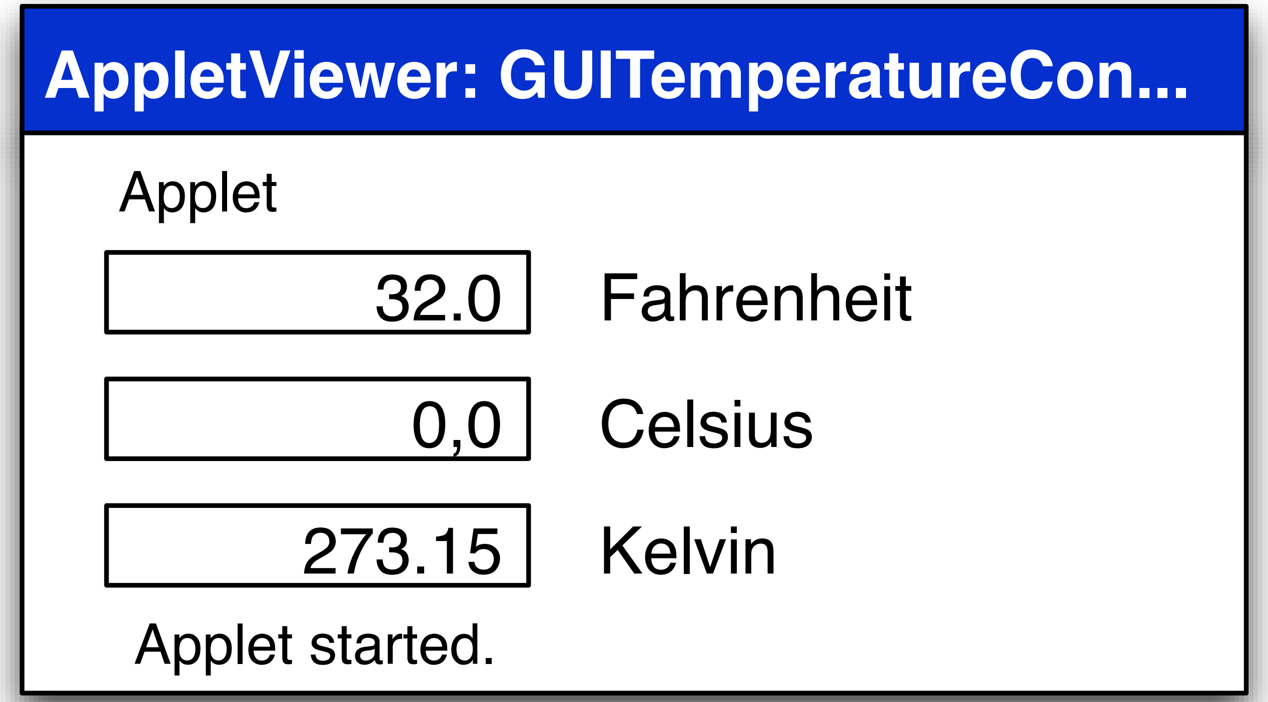
GUI Temperature Converter

- 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



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:



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

Part of the Picture: Boolean Logic & Digital Design

- Arithmetic operations performed by the CPU carried out by <u>logic circuits</u>
- 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

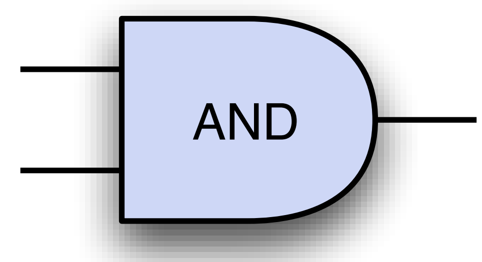
```
!(x && y) == (!x || !y)
!(x || y) == (!x && !y)
```

  - helps simplify complicated Boolean expressions

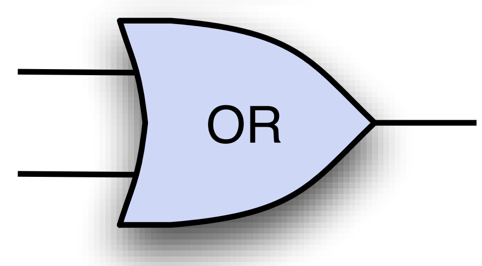
Digital Circuits

- Use three basic electronic components which mimic logical operators

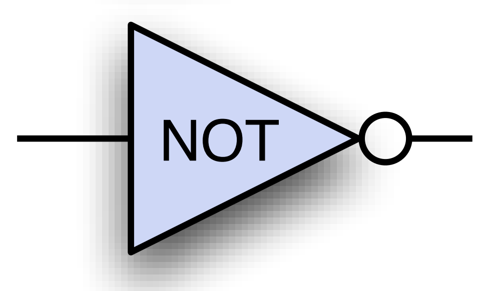
AND gate



OR gate



NOT gate
(inverter)



Circuit Design: A Binary Half-Adder

- Truth table

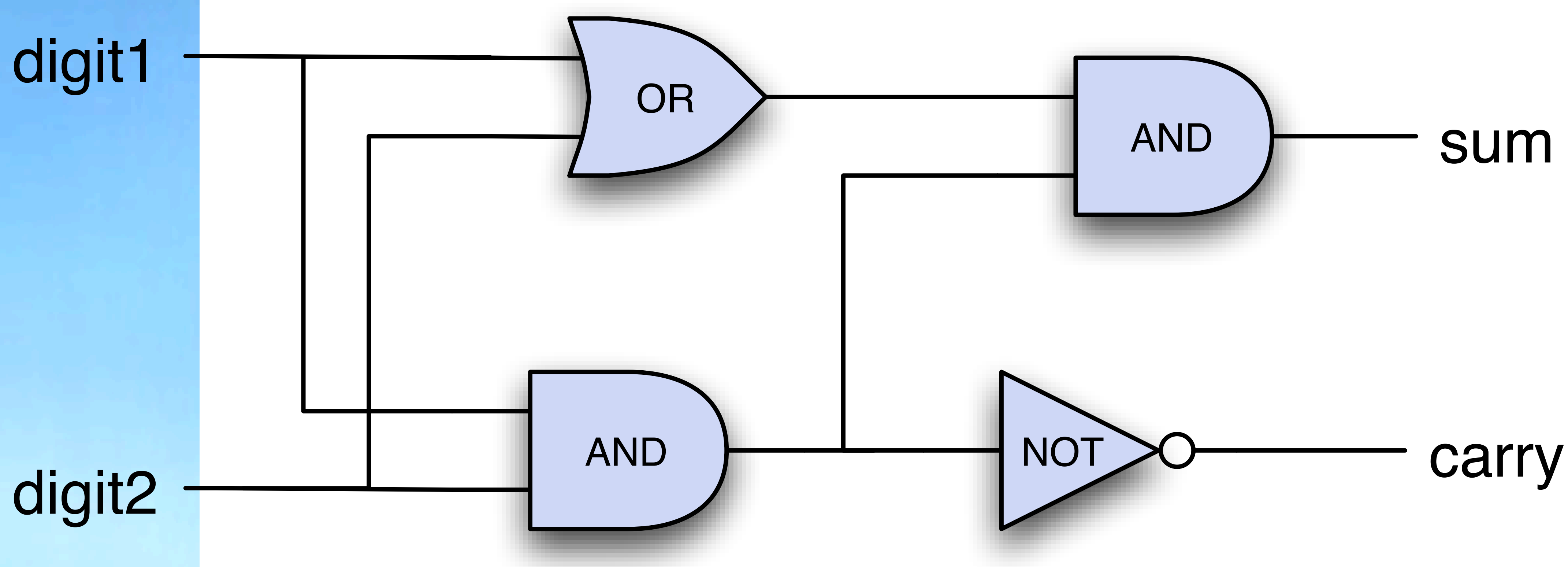
digit1	digit2	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- Boolean expression equivalent:

```
Boolean carry = digit1 && digit2,
          sum = (digit1 || digit2) &&
                       !(digit1 && digit2);
```

Circuit Design: A Binary Half-Adder

- Digital circuit equivalent:



Note binary half-adder class, source code, Figure 7.9, test driver Figure 7.10

Part of the Picture: Computer Architecture

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

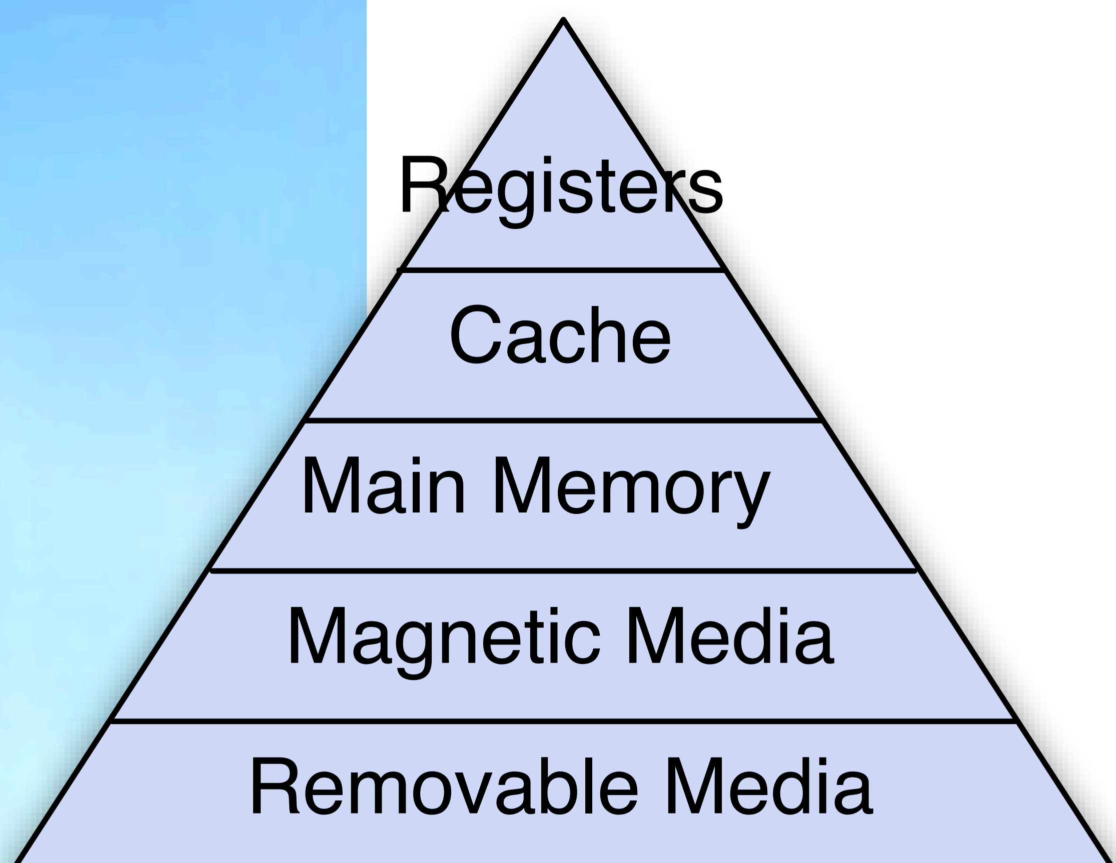
- I/O modules can exchange data directly with processor
  - disk controllers have memory locations to be accessed
- I/O modules may be granted authority to read/write directly from/to memory
  - this frees up processor to do other things

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

Memory Hierarchy

- 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

Control and Timing

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