Java Programming Unit 9
Characters and Decisions

Orange Coast College
Computer Science 170
Spring 2004

The Plan For Unit 9

- **Characters**: Unicode and Java's `char` type
- **Strings**: processing text with Java's `String` class
- **Scope**: how to use access control, `static` and `final`
- **Boolean**: how to use Java's `boolean` data type
- **The if Statement**: applet decision-making power
- **Lab**: meet the `SodaPop` machine
- **Homework**: let's try a `SuperSodaPop`
- **Reading**: finish Chapter 5, strings in Chapter 8

Storing Characters

- Computer cannot process anything but binary numbers
  - This means that human-readable text must be encoded
  - Have everyone agree that 'A' is stored as 65 in computer
  - Similar to way that Morse Code (telegraph) worked
- One of the first codes was **EBCDIC**
  - IBM mainframe standard
  - Extended Binary Coded Decimal Interchange Code
- Considered "proprietary" by IBM; others used ASCII
  - American Standard Code for Information Interchange
  - Most characters stored in 7-bits

Computer cannot process anything but binary numbers
This means that human-readable text must be encoded
Have everyone agree that ‘A’ is stored as 65 in computer
Similar to way that Morse Code (telegraph) worked
One of the first codes was **EBCDIC**
IBM mainframe standard
Extended Binary Coded Decimal Interchange Code
Considered “proprietary” by IBM; others used ASCII
American Standard Code for Information Interchange
Most characters stored in 7-bits
Unicode
- ASCII was designed to be efficient in use
  - Small size meant quick transmission
  - Extra bit used by terminals to separate characters
- Modifying it for non-English languages was difficult
  - The "extra bit" was used to provide alternate characters
  - "Code-pages" allowed DOS programs to switch
- Better solution was to use more memory than a byte
  - Most of the world has standardized on Unicode
  - Go visit at www.unicode.org

The char Data Type
- The char data type is used to store single characters
  - Uses two bytes of storage per character
  - Each storage location is interpreted using Unicode
- Creating char variables: just like other primitives

<table>
<thead>
<tr>
<th>char initial</th>
<th>= &lt;char literal&gt;;</th>
</tr>
</thead>
<tbody>
<tr>
<td>char controlA</td>
<td>= &lt;char literal&gt;</td>
</tr>
<tr>
<td>char hTab</td>
<td>= &lt;char literal&gt;</td>
</tr>
<tr>
<td>char backlash</td>
<td>= &lt;char literal&gt;</td>
</tr>
<tr>
<td>char copyright</td>
<td>= &lt;char literal&gt;</td>
</tr>
</tbody>
</table>

Writing char Literals
- How do you write a literal D?

```
char initial = 'D';
```
- Can't just use the character; looks like a variable
- Must use some punctuation
- Character used in Java is a single quote (')

```
char initial = "D";
```
- Single/double quotes are not interchangeable, like HTML
Other char Literals

- Some characters are not "printable"
  - ASCII 0-31 traditionally used to control hardware
    - Thus, named "control characters"
- How do you include such characters in your code?
  - Generated by holding down CTRL+ Character
  - Still used to "control" software
  - Can write these by simply using the integer value

```
    char ctrlA = '1';
```

"Special" Characters

- Java has a second way to enter "special" characters
  - These include some control characters and punctuation
  - The method is called an escape sequence
- Parts of an escape sequence
  - The escape character: the backslash (\)
    - Tells the compiler "Look out - special instructions!"
  - Followed by a translation value

Common, Predefined Escape Sequences

- Here are some common escape sequences
- These have been predefined to make them easy to use

```
\b  Backspace (numeric value 8)
\t  Tab (numeric value 9)
\n  Newline (numeric value 10)
\r  Return (ENTER on PCs) (numeric value 13)
\f  Form-feed character (numeric value 12)
```
Unicode Escape Sequences

- Use escape sequences to represent any Unicode character
- Starts with `\u` or `\U`, followed by 4-digit hex Unicode value
- Can use escapes to create `char` variables or in strings

```java
char controlA = '\u0001';
char tradeMark = '\u2122';
char copyright = '\u00A9';
String message = '\u00BC is \u00BD of \u00BD\n';
```

- Can also use as part of regular Java code
- Difference between characters and glyphs

Escaping Punctuation

- Store an apostrophe inside a `char`
  ```java
  char apostrophe = '\';
  ```
- Store a quote inside a `String`
  ```java
  String msg = "and then he said, \"Hello\"";
  ```
- Store a backslash inside a `String`
  ```java
  String iniFile = "C:\\Windows\\my.ini";
  ```
- Introducing Exercise 9A

Meet the Strings

- What are `Strings` in Java?
  - **Immutable** sequences of 0…n characters
  - **Not** null terminated arrays [as in C++]
  - Some characteristics of primitives, some of objects
- `Strings` are the only object that permits use of literals
  - Formed by enclosing text inside double quotes
  - May also include escape characters
- Always use literals instead of the `String` constructor
Strings, Primitives, Objects

- Strings are like primitives in several ways
  - Initialize using a literal rather than using `new`
  - Java overloads the `+` to perform concatenation
- Strings really are objects, though, not primitives
  - Strings are references not values

How Programs Store Data I

- 3 Types of storage space (at program run time)
  - Static storage area
    - Stores all static variables and constants
  - Stack-based storage area
    - Local variables are stored here during method call
    - Create when variable defined, free when out of scope
    - Created contiguously at runtime in LIFO order
  - Heap or "free-store"-based storage area
    - Area of memory reserved for reference types

How Programs Store Data II

- Reference types are divided into two pieces
  - The `variable` or `handle` used in our programs
    - May be stored on the stack or in the static storage area
    - This is the object variable
    - The unnamed, actual object stored on the heap
  - The actual object is created by using the `new` operator
    - This allocates memory and may initialize the variable
    - The variable stores the address of the object or `null`
    - This memory must be managed (garbage collector)
How Programs Store Data III

- The stack and the heap

Reference and Value Types I

- Primitives are "value types"
  - Means that a variable is just a "bucket" holding actual value
- Copying value types
  - Assignment makes duplicate, independent copy

```java
int a = 32, b;  // Two variables
b = a;          // Two independent values
```

Reference and Value Types II

- This is called a deep copy or value semantics
Reference and Value Types III
- Copying reference types
  - Variable or handle holds address of actual value
  - Only the address is copied

```java
Button a = new Button("The One and Only");
Button b = a;
```

Reference and Value Types IV
- Called a shallow copy or reference semantics

```
The Stack

The Heap
```

A Tale of Two Strings
- The null String is an uninitialized String
  - Its reference contains the value null
  - The only thing you can do to it is initialize it
- The empty String is a String of length 0
  - Can send messages to the empty String, but not the null

```java
String emptyString = "", nullString;
int len = emptyString.length(); // OK
int len2 = nullString.length(); // Error
```
String Operations

- Java "overloads" two operators to work with strings
- The plus (+) concatenates or "pastes" strings together
  
  ```java
  String s1 = "How", s2 = "now";
  String s3 = s1 + " " + s2;
  ```
- You can also use +=, but only on a string variable
  
  ```java
  s3 += " brown cow";
  ```

String Methods

- The `String` class has many methods for manipulating and finding values inside Strings
  - `s.compareTo("OK")`
    - Compares String `s` to the String "OK".
    - Returns 0 if equal, 1 if `s` is "greater" than OK, -1 if less
  - `s.toUpperCase(), s.toLowerCase()`
    - Returns a String translated to either upper or lower case
  - `s.replace('a','b')`
    - Returns a String with all occurrences of a replaced by b
    - Notice that `s` cannot be modified

Some More String Methods

- `s.equalsIgnoreCase("OK")`
  - Performs a case-insensitive compare
- `s.startsWith("OK"), s.endsWith("OK")`
  - Returns true if OK found at beginning or end of `s`
- `s.charAt(i)`
  - Returns the character at position `i` (starting at 0)
- `s.indexOf('O'), s.indexOf("OK")`
  - Returns position of char or String inside `s`

Introducing Exercise 9B
**Introducing Scope**

- Scope is the **visibility** of variables and methods.
- The first kind of scope is **local scope** or **block scope**.
  - Variables created in a method and formal arguments.
  - Variable visible from declaration to end of block.

```java
public void aMethod()
{
    // x is not visible here
    int x = 5;
    // x is visible here
}
// x is not visible here
```

**Local Scope**

- Identifier names must be unique in a method.

```java
public void aMethod()
{
    int x = 3;           // OK, only x in scope
double x = 2.5;       // ERROR, x already used
}

Can use same variable name in different scopes
```

```java
public void aMethod() { int x = 3;           // OK }
public void bMethod() { double x = 2.5; // OK }
```

**Class Scope**

- The second type of scope is **class scope**.
  - Fields (instance variables) and methods have class scope.
  - Visible inside any method (even before declaration).
  - Not visible to fields defined previous to declaration.

```java
public class AClass
{
    public void aMethod() { x = 5; // OK }
    public void bMethod() { x = 7; // OK }
    private int x = 3;
    public int PUBLIC_FIELD = 5;
}
```
Unique Names

- Each field in **same class** must have a unique name

```java
public class A {
    int x;
    double x; // illegal
}
```

- Fields in **different classes may have** the same name

```java
public class A { int x; // OK }
public class B { double x; // OK }
```

Inherited Names

- A field may have **the same name** as a superclass field

```java
public class A { int x = 3; }
public class B extends A { double x = 3.3; }
```

  - The field `x` in class `B` "hides" or "shadows" `A.x`

  - Use the keyword `super` to access `A.x` from `B`

```java
public void aMethod() {
    x = 5.2; // double B.x
    super.x = 7; // A.x
}
```

Instance and Local Variables

- A local variable may **have the same name** as a field

  - The local variable hides the field inside the local scope
  
  - You can access the field by using the keyword `this`

```java
public class A {
    double x = 2.2;
    public void aMethod() {
        int x;
        x = 7; // illegal
        this.x = 5.5; // local variable x
        x = 7; // field x
        this.x = 5.5; // field x
    }
}
```
**External Scope**

- Java has **no global variables** or external scope.
- Class scope variables and methods can be accessed outside the class, if they are fully qualified.

```java
public class BClass {
    private AClass aObj = new AClass();
    public void aMethod() {
        int noGo = PUBLIC_FIELD; // Not in scope
        int OK = aObj.PUBLIC_FIELD; // OK
    }
}
```

**Static fields**:
- Are shared among all objects in a class.
- Act like “class wide” global variables.
- Called “class variables” instead of “instance variables.”

**Static methods**:
- Do not require an object to access them.
- Are used by sending the message to the **class**.
- The **Math** class is composed of **static** methods.
- “Factory” methods are often **static** methods.

---

**What Are “statics”**

- **Static fields**: are shared among all objects in a class.
- Act like “class wide” global variables.
- Called “class variables” instead of “instance variables.”

- **Static methods**: do not require an object to access them.
- Are used by sending the message to the **class**.
- The **Math** class is composed of **static** methods.
- “Factory” methods are often **static** methods.

---

**Defining a Class Variable**

- The only difference syntactically is to add the word **static**.
- Doesn’t matter which modifier comes first.

```java
public class XFighter extends SpaceCraft {
    private static int numShips = 0; // class variable
    public XFighter(int id) {
        numShips++;
        serialNo = id;
    }
    public void dieDieDIE() { numShips--; }
}
```
**Static (or Class) Methods**

- Users can call a static method by using the class name:
  ```java
  int ships = XFigher.getNumShips();
  ```
- They don't need to create an XFigher object first.
- To define a static method, use the keyword static:
  ```java
  public static int getNumShips()
  {
      // cannot access serialNo from this method
      return numShips;
  }
  ```
- You can only access static fields, no instance variables.

**Using the final Keyword**

- A final field, method or class cannot be changed:
  - A final field is a constant.
  - A final method cannot be overridden in a subclass.
  - A final class cannot be extended.
- To create constants for your class (like Font.BOLD):
  - Make the field public, static, and final:
    ```java
    public static final int MODEL_MB2000 = 3;
    ```
- Introducing Exercise 9C

**The boolean Primitive Type**

- The last of the four primitive types is boolean:
  - The boolean type has two possible values.
  - boolean literals are written as true and false.
    - Remember, Java is case sensitive.
  - Such literals are often used as method arguments:
    ```java
    theLabel.setVisible(false);
    theButton.setEnabled(true);
    ```
### boolean Variables and Methods

- You create `boolean` variables like other variables
  ```java
  boolean visible = false;
  ```
- Methods can also produce `boolean` values
  - Normally store result in a `boolean` variable
  ```java
  visible = aLabel.isVisible();
  enabled = aButton.isEnabled();
  showing = aWindow.isShowing();
  ```
- Example: [Switcher.java][Switcher.java] [Run the applet]

### The Relational Operators

- **Most** boolean values are produced by comparing two values using the **relational** operators
- Six operators: `<`, `<=`, `>`, `>=`, `==`, `!=`
  - Compares relationship between two values
  - Produces a true/false value
  - Only works with comparable primitive types
  - Examples: `(sales > 1000.0)`, `(numFrogs != 100)`
- Used primarily in "Flow of Control" statements

### A Few Fine Points

- Many languages use the `=` symbol for equality
  - If `( x = 10 )` won’t compile in Java
  - If `( ok = true )` does compile, but is incorrect
- **Never** compare against the literals `true` or `false`
  - If `( ok == true )` is very very poor style
- Many languages use `<>` to mean not-equal
  - If `( x <> 10 )` won’t compile in Java
- Use `>=` rather than `=>` which doesn’t compile
### Primitive Relations
- You can compare most primitive numeric types, including `char`, but not `boolean`, without problem.
- You don't have to worry about precision or size.

#### Floating-point Relations
- Generally, you should not use `==` or `!=` with floating-point.
- With mixed type, 2.34 and 2.34f are NOT equal.
- They have different bit patterns.
- Example: `FloatingRelations.java`.
- Even within the same type binary calculations may differ.
- Multiply `0.1 * 10.0`, and then add `0.1` ten times.
- Mathematically equivalent, but differ in Java.
- Simple calculation compares difference to 10E-14.
- Example: `FloatingRelations2.java`.

### Object Relations
- When comparing objects and `Strings` you can only use the `==` and `!=` operators, not `>`, `<`, `>=`, or `<=`.
- You should not mix different object types.
- Don't compare a `Button` with a `Label`, for instance.
- You can compare an `Object` with any object type.
Simple Equality I

- With objects, == measures identity or simple equality
- Two variables refer to the same object or String
- Here's an example with three Button variables

```java
Button a = new Button("Hi Mom");
Button b = new Button("Hi Mom");
Button c = b;
```

Simple Equality II

- In memory, the Buttons and variables look like this
- Here's what happens when we compare these Buttons

```java
a == b; // false - different button objects
b == c; // true - same button
a == c; // false - different buttons
```

String Equality I

- Strings are objects, and comparisons work the same way
- If you look at this code:

```java
String a = "Hi Mom";
String b = "Hi Mom";
String c = b;
```

- You'd expect to see this in memory:
String Equality II

- When we write the same boolean expressions, though:

  ```java
  a == b; // true?? - Why?
  b == c; // true - same String
  a == c; // true??
  ```

- The reason the Strings are identical is because Strings are immutable, so memory really looks like this:

Value Equality

- Value equality is when two objects have the same contents
  - This varies on a type-by-type basis
  - Types that are comparable in this way will overload the `Object` method called `equals()`

```java
String nameString = tf.getText();
namestring == "Fred"; // false
nameString.equals("Fred"); // true
```

- Use `equals()` to compare Strings [TestStringEquals.java]
- Use `==` to compare if a String is null

Introducing Selection

- One type of "Flow of Control" statement in computer programs
- Selection acts as "highway divider" in your code
  - Causes conditional execution of alternative code
  - Uses boolean expressions from relational operators
    - If your salary is less than $25,000 then
      - Calculate tax based on 15% marginal rate
    otherwise
      - Calculate tax based on 15% base and a 28% marginal rate
- Introducing Exercise 9D
The if Statement

- Like classes and methods, the if has two parts
  - The header contains a boolean condition
  - The body contains code to be executed
- If the condition is true, then the code in the body is executed
- If the condition is false, the code in the body is skipped

Syntax Points

- The keyword if must be in lower-case
  - If or IF won’t work
- Place a boolean expression in the parentheses
  - The parentheses are required (unlike VB or Pascal)
  - You can’t use a numeric value like C
- Make sure you don’t put a semicolon after the condition
  - This error will not be caught by the compiler
  - Your “real” body will always be executed when this happens
- The body consists of a single statement

A Simple Example I

- Let’s create an applet that contains two Buttons
  - Paint the applet red or green depending on the button
  - We’ll use the if to tell the buttons apart
- Step-by-step with JCreator
  - Create an empty project named SimpleIf
  - Add an empty Java file named SimpleIf.java
  - Add an HTML file named SimpleIf.html
  - Add the applet tag to your HTML file
A Simple Example II

- Write the body of the SimpleIf applet
- Import java.awt, java.applet, and java.awt.event
- Add two Button instance variables: redBtn, greenBtn
- Add your buttons and hook them up in the init() method
- Write the actionPerformed() method
  - Use e.getSource() to find out "who" triggered the event
  - Save in an Object variable named pressed
  - Compare pressed to redBtn using an if statement
  - If true, set the background red
  - Repeat for the greenBtn variable

A Small Problem

- The SimpleIf program is not very efficient
  - If you press redBtn, your code still checks greenBtn
  - Not necessary because choices are mutually exclusive
- Java’s if has an else portion that can be efficiently used for such either/or conditions
- SimpleIf02.java

Blocks and Indentation

- Much of the time, you need to carry out several actions inside the body of an if statement, not just one
- To do this, surround the body with braces (a "block")

```java
if ( amountSold <= 35000 ) {
    bonusPct = .035;
    bonusAmt = amountSold * bonusPct;
}
```
- Indent the code inside the block by one tab stop
**Why Use Braces?**

- Braces are not required for a single-line if statement
- Still a good idea, even when not required
- Helps avoid mistakes when modifying code

```java
if (amountSold <= 35000)
    bonusPct = .035;
else
    bonusPct = .075;
bonusAmount += 100;
bonusAmount += amtSold * BonusPct;
```

**The Phantom Semicolon**

- A common mistake is to put a semicolon after the condition

```java
if (filesAreAllBackedUp);
{
    formatTheHardDrive();
}
```

- Because of the semicolon, this fragment of code says:
  1. Check to see if files are backed up
  2. If they are, then don’t do anything (the null statement)
  3. Format the hard-drive regardless of the backup
- Introducing Exercise 9E

**Finish Up**

- Assignments due on Tuesday April 20
- Week 9 Lab: Soda Pop
  - Run the applet, examine and modify the code
- Quiz 9
- Homework 9 [ A SuperSodaPop ] due April 27