CS11 – Advanced Java

Winter 2011-2012
Lecture 3
Java Constants

- Frequently need to define constants in Java code
  ```java
  public class BoggleBoard {
      /** Default size for a Boggle board. */
      public static final int DEFAULT_SIZE = 4;
      ...
  }
  ```

- Standard conventions for Java constants:
  - Name usually follows **ALL_CAPS** naming convention
  - Declare `public static final`
  - (or, use `private` / `protected` if appropriate)
The **static** Keyword

- Members of a class can be declared **static**
  - They are associated with the class, not a particular object
  - For static fields, there is only one copy of the value

- Example:

  ```java
  public class CommandPrompt {
      public static final String PROMPT =
          "Type command:  ";
      ...
  }
  
  PROMPT is an object, but it isn’t associated with individual CommandPrompt instances
  
  Only one value, and all code can share that single value
  
  Much more efficient memory usage than an instance field, when other code can share a single value
When are static fields initialized?

```java
public class CommandPrompt {
    public static final String PROMPT =
        "Type command:  ";
    
    ...
}
```

The VM initializes a class the first time the type is actually used by other code.

- Class definition is found via the classpath, and then verified
  - e.g. all instructions are valid; jump instructions go to valid addresses; etc.
- Any references to other types may be verified and resolved
  - (may involve the loading of additional classes, of course)
- Finally, static fields in the class are initialized
Static Initialization (2)

- Static fields are initialized at the end of the class-load process
- Sometimes, can’t initialize a static field with a single line of code
  ```java
  public class NoiseGenerator {
      public static final Vector3f[] noiseVectors =
          new Vector3f[1024];
      ...
  }
  ```
  - Also need to initialize the noise-vector elements to random unit-vectors
  - Clearly can’t do it in a single line!
- How to implement this static initialization?
Classes can specify static initializers:

```java
public class NoiseGenerator {
    public static final Vector3f[] noiseVectors =
        new Vector3f[1024];

    static {
        for (int i = 0; i < noiseVectors.length; i++) {
            noiseVectors[i] = new Vector3f();
            ... // Initialize the vector
        }
    }

    ...}
```

- Static initializers **cannot** throw checked exceptions!
- Initialization of static fields, and execution of static initializers, occurs in order of appearance in the source file
- Static initialization is also specified to be thread-safe in Java
The **final** Keyword

- Java variables can be declared as **final**
  - The variable can only be assigned to once.
- Frequently used for constant class and instance fields
  ```java
class CommandPrompt {
  public static final String PROMPT =
    "Type command: ";
  ...
}
```
  - **PROMPT** can only be written to once, and then it is fixed
- **final** fields are usually assigned where they are declared, but this is not strictly required by Java!
  - **final** instance fields must be assigned to, by the end of every constructor
  - **final** class fields must be assigned to, by some static initializer
The **final** Keyword (2)

- **final** sometimes uses on local variables or method-arguments
  - Prevents reassignment to variables that shouldn’t change
  - Used to reduce correctness issues
  - Technique does have some *limited* usefulness...

- Example:
  ```java
  int findWord(String w, final ArrayList<String> words) {
    int i = 0;
    for (String s : words) {
      if (s.equals(w)) return i;
      i++;
    }
    return -1;
  }
  ```

- What can’t we do with *words*?
  - We can’t set *words* to refer to something else
  - Increases the correctness of our own method (slightly)
The **final** Keyword (3)

- Example:
  ```java
  int findWord(String w, final ArrayList<String> words) {
    int i = 0;
    for (String s : words) {
      if (s.equals(w)) return i;
      i++;
    }
    return -1;
  }
  ```

- What can we do with **words**?
  - We can call any of the methods on **words**…
  - We can call mutators on **words**!
    - **words.add("yo' mama!");**
    - **words.clear();**

- **final** only prohibits reassignment to the variable
- Declaring **words** as **final** doesn’t really get us much…
Java final keyword is nothing like C++ const
  (and Java has no equivalent to C++ const)
You will probably run into projects that use final for method-args and local variables...
  Just be aware of the significant limitations of this technique
If you really need immutable state:
  Create a class without mutators!
    (and if necessary, a subclass that provides mutators)
    Java String, Integer, etc. classes are all immutable
  Or, see Collections.unmodifiableList(List), etc.
    Provides an immutable view of another collection
    Original collection is still mutable, but can pass the immutable view to other methods to work with
Back to Java Constants…

- Covered the standard modifiers used for constants
  
  ```java
  public class BoggleBoard {
      /** Default size for a Boggle board. **/
      public static final int DEFAULT_SIZE = 4;
      ...
  }
  ```

- For simple constants, this is the recommended way
  - When constant is an object, improves memory efficiency

- Two other common ways constants are often used
  - Both are not so good. 😊
Interfaces and Constants

- Interfaces can contain two kinds of members
  - Public methods, and constants!
    - Constants are declared as `static final`, since all interface members are automatically `public`

- When a package uses a lot of constants, commonly put into a “constant interface”
  - The interface contains only constants, no methods

- Lots of examples of this in the Java API
  - `javax.swing.SwingConstants` interface
    - e.g. defines alignment constants `LEFT`, `CENTER`, `RIGHT`
  - Many Swing classes “implement” `SwingConstants`, so they can easily use the constants in their implementations
    - No methods need to be added; `SwingConstants` has none!
Joshua Bloch and Constant Interfaces

- **Interfaces define a type in Java**
  - They specify a set of behaviors that implementing objects provide

- **When a class implements an interface:**
  - It should say something about what clients of the class can do with objects of that type!
  - Other code can refer to an object by its interface types

- **Constant interfaces violate this principle**
  - e.g. `SwingConstants` doesn’t specify any behavior at all!
  - But, we can write strange code like this:
    ```java
    SwingConstants c = new JButton("this is weird");
    ```
  - Can’t call any methods on `c` because it declares none!
A Better Solution: Constant Utility Classes

If you have a lot of constants to group together:

- Put them into a utility class that can’t be instantiated
  - Implement a private default constructor
- Provide the set of public static final fields

Moral:

- Just because the Java API uses certain design patterns, doesn’t mean that you should. 😊
Simple Enumerations

- Constants are also frequently used for enumerations
  ```java
  /** Represents the suits of cards in a card deck. */
  public class Card {
    public static final int SPADES   = 1;
    public static final int HEARTS   = 2;
    public static final int CLUBS    = 3;
    public static final int DIAMONDS = 4;
    ...
  }
  ```

- Problems?
  - No type-safety:
    ```java
    public class Card {
      ...
      void setSuit(int suit);
    }
    ```
  - Could accidentally mix different enums, or specify invalid values!
Typesafe Enumerations

- Implementing enumerations this way is very error-prone
- A better approach: “typesafe enumerations”
  - Create a specific class for each enumeration
  - Create a unique object for each enum value
    ```java
    public class Suit {
        /** Only Suit can call its own constructor. */
        private Suit() { }

        public static final Suit SPADES   = new Suit();
        public static final Suit HEARTS   = new Suit();
        public static final Suit CLUBS    = new Suit();
        public static final Suit DIAMONDS = new Suit();
    }
    ```
  - Can add other fields to represent details of each enum value, such as `name`, `id`, etc.
The “typesafe enumerations” pattern is very useful, but also needs a lot of infrastructure code.
- Primarily to ensure that each enum-value is actually unique within the JVM.

Also, can’t write switch-statements that test objects:
```java
Card c = ... ;
switch (c.getSuit()) {
    case Suit.SPADES:
        ...
}
```
- This code won’t compile with the typesafe enum approach!
- Will compile if suits are represented as integers, but that approach has bigger issues.
Java 1.5 **enum** Types

- Java 1.5 introduced support for typesafe enums
  - The pattern is tremendously useful…
  - The implementation can be tricky to get right…
  - And we would also like language support (e.g. `switch`)

- Updating our **Suit** to be an enumeration:
  ```java
  public enum Suit {
    SPADES,
    HEARTS,
    CLUBS,
    DIAMONDS
  }
  ```
  - Can put Javadoc comments on the enumeration, and on each value
Can write `switch` statements against `enum` values:

```java
Card c = ... ;
switch (c.getSuit()) {
    case SPADES:
        ...
}
```

Java enums also provide support for `toString()` and other `Object` methods automatically

```java
System.out.println(c.getSuit());
→ SPADES
```

 Enums also have a `values` array-member, containing all specified enum-values

```java
for (int val = 1; val <= 13; val++)
    for (Suit s : Suit.values)
        deck.add(new Card(val, s));
```
Extending Enumerations

- Java `enum` types are implemented as classes
  - Can add fields and methods to your enum types
- Example:
  ```java
  public enum ChessPiece {
      KING  (200), // Arbitrary value for king
      QUEEN (9),
      ROOK  (5),
      BISHOP(3),
      KNIGHT(3),
      PAWN  (1); // Note the semicolon!
  
  private final int value;   // Point-value of piece
  
  ChessPiece(int value) { this.value = value; }
  
  public int value() { return value; }
  }
  ```
Can also put `enum` declarations within other classes

```java
public class Card {
    public enum Suit {
        SPADES, HEARTS, CLUBS, DIAMONDS
    }

    public Card(int value, Suit suit) {
        ...
    }
}
```

- Card can refer to `enum` values as `Suit.SPADES`, etc.
- External code must specify `Card.Suit.SPADES`, etc.
This Week’s Lab

- Begin building the UI for Boggle game
  - Start with classes to display a Boggle board, and to let users enter words they find
- Give the user some visual cues
  - Use a font that is large enough to read easily
  - Update button “enabled” status and border-color to indicate what letters are available to select next
- UI code also needs to provide a method to return the currently selected word
Example User Interface

- A grid of buttons displays the current Boggle board
- Button borders indicate what letters can be chosen
- When user selects a letter, it shows a red border
- Only the letters adjacent to last selection are available
Example User Interface (2)

- As letters are selected, word is shown in red
  - The word itself is the concatenation of each button’s text-value

- “Available letters” are always based on last selected letter
  - Exclude already-selected letters!
General Approach

- Don’t reinvent the wheel!
- Swing already provides buttons and panels
  - Let’s just customize their behavior!
- Create a subclass of JButton that handles Boggle-specific details of displaying a cell
  - Manage button-state, appearance, cell value, etc.
- Create a subclass of JPanel that displays an entire BoggleBoard
  - Methods to set the board to use, and to get current word
  - Handles action-events from buttons and updates their appearance
Swing Component Appearance

- All Swing components derive from \texttt{javax.swing.JComponent}
  - Provides common functionality across all components
  - Custom components that paint their own contents are also derived from \texttt{JComponent}
- Many ways to change a \texttt{JComponent}'s appearance
  - Set a tooltip, add one or more borders, change foreground / background colors, change the cursor, change the font, etc.
- Can also enable/disable components
  - Disabled components do not receive user input
  - Indicated in UI by graying out the component
  - Use \texttt{setEnabled(boolean)} and \texttt{isEnabled()}
Swing Component Naming

- Another naming convention for Swing components
- All Swing components derive from `JComponent`
  - The Swing analogue to Java AWT’s `Component` type
- All Swing component names start with a “J”
- Unless it *really* doesn’t make sense for your code, you should also follow this convention
  - e.g. `JBoggleButton`, `JBoggleBoard`
Swing Components and Fonts

- Can change the font on Swing components
  - `setFont(Font)` and `getFont()` methods
- The `java.awt.Font` class represents fonts in Java
- Java fonts fall into two categories:
  - Physical fonts correspond to actual fonts installed on your computer (e.g. Arial or Helvetica)
  - Logical fonts are “generic” fonts that all Java VMs must provide
    - Typically provided by mapping each logical font-name to a physical font, based on what OS provides by default
    - Serif, SansSerif, `Monospaced`, Dialog, and DialogInput
Swing Components and Fonts (2)

- Easiest way to get fonts is via **Font** constructor
  - `Font(String name, int style, int size)`
  - **Font** has constants for all logical font names, and all styles
    // Get a bold, 20-point font without serifs
    Font f = new Font(Font.SANS_SERIF, Font.BOLD, 20);
  - Can also specify other font names, but no guarantee they will be available!
    // Get an italicized, 12-point Times New Roman font
    f = new Font("Times New Roman", Font.ITALIC, 12);
  - If a font name is unrecognized then Java will switch to the “Dialog” logical font
  - Suggestion: only use **logical** font names with constructor
Swing Components and Fonts (3)

- To get all fonts on a particular system, use:
  ```java
  Font[] java.awt.GraphicsEnvironment.getAllFonts()
  ```
  - Returns an array of Font objects that includes all available fonts
  - Returned fonts are only 1-point in size
  - Looks like this: 
  - Application must derive fonts from these “base fonts”

- To make your application most portable, use this mechanism to find system fonts
  - Or, just stick with the logical fonts
Swing components can be given a border
- Effectively shrinks the Swing component itself

Set and get a component’s border via `setBorder(Border)` and `getBorder()` methods

`Border` is an `interface` defined in `javax.swing.border` package
- See Java APIs for implementations!

Two ways to get simple borders:
- Create it yourself:
  ```java
  Border b = new LineBorder(Color.RED, 3);
  ```
- Use the `javax.swing.BorderFactory` class
  ```java
  Border b = BorderFactory.createLineBorder(Color.RED, 3);
  ```
References

- **Effective Java** by Joshua Bloch
  - Item 17: Use interfaces only to define types
  - Item 21: Replace `enum` constructs with classes