Topic 5:
Enumerated Types
and Switch Statements

Reading: JBD Sections 6.1, 6.2, 3.9
What's wrong with this code?

```c
if (pressure > 85.0)
    excessPressure = pressure - 85.0;
else
    safetyMargin = 85.0 - pressure;
```

- "Magic Numbers" make code hard to read and maintain.

- Better:

```c
double MAXPRESSURE = 85.0;
if (pressure > MAXPRESSURE)
    excessPressure = pressure - MAXPRESSURE;
else
    safetyMargin = MAXPRESSURE - pressure;
```
Constants

- "Constants" are a good way to avoid magic numbers
- A constant is like a variable that does not vary
- Declare a constant whenever:
  - A number is used more than once in a program
  - It represents something that does not change
- Declare constants inside a class but not inside a method.
- Convention: name of constants are all-upper-case
- Use the keywords `static` `final`

```java
static final double MAXPRESSURE = 85.0;
static final int SIZE = 100;
static final String US = "United States";
```
Sets of related constants

- Suppose that the state of a machine could be running, waiting, or stopped.
- It's inefficient to represent these states as Strings.
- You could represent them as constant integers:
  ```java
  static final int RUNNING = 1;
  static final int WAITING = 2;
  static final int STOPPED = 3;
  ```
- But consider a method that wants a machine state for its parameter:
  ```java
  hourlyCost(int x)
  ```
- There's nothing to prevent this call: `hourlyCost(5)`
A Safer Approach

- Define a type with only three values:
  ```cpp
  enum MachineState { RUNNING, WAITING, STOPPED };
  ```

- Define methods that take parameters of this type:
  ```cpp
  static double hourlyCost(MachineState s)
  ```

- Call the method using symbolic names for arguments:
  ```cpp
  hourlyCost(MachineState.RUNNING)
  ```

- Now the compiler can ensure that the method will get a valid argument:
  ```cpp
  hourlyCost(5)
  ```

  Error because 5 is not a MachineState
Enumerated Types

- Enumerated types are:
  - A new feature in Java 5.0
  - A simple kind of user-defined type
  - Used to restrict a type to a finite set of values

- Examples:
  ```java
  enum MachineState { RUNNING, WAITING, STOPPED };
  enum Size { SMALL, MEDIUM, LARGE };
  enum Season { WINTER, SPRING, SUMMER, FALL };
  enum Suit { SPADES, HEARTS, DIAMONDS, CLUBS };
  enum Answer { YES, NO, MAYBE };
  enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };
  ```
Enumerated Types

- Declaring an enumerated type
  ```java
enum Season { WINTER, SPRING, SUMMER, FALL };
```

- Declaration may be either inside a class or outside
  (But a few people have reported problems with DrJava when declaring an enum type that is not inside a class.)

- Using an enumerated type
  ```java
  Season s;
  s = Season.SUMMER;
  System.out.println(s); // prints SUMMER
  ``

- Advantages of enumerated types
  - Type safety
  - Self-documenting code
  - Prints a readable name (unlike a static final int)
Operations on Enumerated Types

- **OK**: assignment, equality-comparison
  
  \[
  = \quad == \quad !=
  \]

- **Examples:**
  
  
  - `myColor = Color.RED`
  
  - `today == Day.TUESDAY`

- **Not OK**: arithmetic, inequality-comparison
  
  \[
  + \quad - \quad * \quad / \quad > \quad >= \quad < \quad <=
  \]

- **Examples:**
  
  - `Day.MONDAY + Day.TUESDAY`
  
  - `Season.WINTER > Season.SPRING`
Enumerated types are passed by value

```java
static double hourlyCost(MachineState s) {
    s = MachineState.STOPPED;
    return 47.00;
}

public static void main(String[] args) { 
    MachineState x = MachineState.RUNNING;
    double cost = hourlyCost(x);
    System.out.println(x);    // Still RUNNING
}
```
The `values()` method

- Every enumerated type has a `values()` method that returns an array of all the enumerated values

```java
enum Season { WINTER, SPRING, SUMMER, FALL }
```

- `Season.values()` returns an array containing all the values of the `Season` type

```java
Season[] s = Season.values();
// s.length is 4
// s[0] is Season.WINTER
// s[1] is Season.SPRING
// s[2] is Season.SUMMER
// s[3] is Season.FALL
```
Iterating over an Enumerated Type

- Use `values()` to get an array of values
- Use a "for-in" loop to iterate over this array.

```java
enum Season { WINTER, SPRING, SUMMER, FALL }; 
for (Season s: Season.values()) {
    System.out.println("I love Paris in the " + s);
}
```

- Prints:
  
  I love Paris in the WINTER
  I love Paris in the SPRING
  I love Paris in the SUMMER
  I love Paris in the FALL
Branching on an Enumerated Type

- One way to handle several enumerated cases:

```java
enum Season { WINTER, SPRING, SUMMER, FALL };  
Season s = someSeasonExpression;  
if (s == Season.WINTER) {
    heating = true;  
    cooling = false;  
    thermostat = 68.0;
}
else if (s == Season.SUMMER) {
    heating = false;  
    cooling = true;  
    thermostat = 78.0;
}
else {
    heating = false;  
    cooling = false;
}
```
The switch statement: a new way of branching

```java
enum Season { WINTER, SPRING, SUMMER, FALL }
Season s = someSeasonExpression;
switch(s) {
    case WINTER:
        heating = true; cooling = false;
        thermostat = 68.0;
        break;
    case SUMMER:
        heating = false; cooling = true;
        thermostat = 78.0;
        break;
    default:
        heating = false; cooling = false;
}
```
A switch statement is like a stack of if's

```java
if (s == T.CASE1) {
    handleCase1();
}
else if (s == T.CASE2) {
    handleCase2();
}
else {
    handleOthers();
}
```

```java
switch(s) {
    case CASE1:
        handleCase1();
        break;
    case CASE2:
        handleCase2();
        break;
    default:
        handleOthers();
}
```
More about the \texttt{switch} statement

- Valid types for the expression in the \texttt{switch}-clause:
  - an enum type (\texttt{Day}, \texttt{Season}, etc.)
  - \texttt{int}
  - a primitive type that can be converted to \texttt{int} by a \textit{widening conversion} (\texttt{short}, \texttt{char}, or \texttt{byte})

- Don't forget a \texttt{break} after each case

- Values in the case-clauses must be constants (evaluated at compile-time)
  - OK: 47, \texttt{SUMMER}
  - Not OK: \texttt{myAge}, \texttt{thisSeason}, \texttt{x + y}

- Default clause is optional
  - If no matching case or default, \texttt{switch} takes no action
Inside a switch, you **must not** prefix the enum values with their typename. (Why not?)

```java
enum Suit { SPADES, HEARTS, DIAMONDS, CLUBS };
switch(trump) {
  case SPADES:
    handleSpades();
    break;
  case HEARTS:
    handleHearts();
    break;
  case DIAMONDS:
    handleDiamonds();
    break;
  case CLUBS:
    handleClubs();
    break;
  default:
    System.out.println("Invalid trump suit");
}
```

`Suit.HEARTS` is not OK here.
Why does a switch require breaks?

- Sometimes you may want to "double up" cases:

  ```java
  static boolean weekend(Day today) {
      boolean answer;
      switch(today) {
          case SATURDAY:
          case SUNDAY:
              answer = true;
              break;
          default:
              answer = false;
      }
      return answer;
  }
  ```

- Note that if there is no `break`, execution falls through from one case to the next.
Why does this method fail to compile?

```java
enum Temp {HOT, COLD};

static String drink(Temp t) {
    String drink;
    switch(t) {
        case HOT:
            drink = "Coffee";
            break;
        case COLD:
            drink = "Beer";
            break;
    }
    return drink;
}
```

Error: Variable drink might not have been initialized.
This version compiles with no errors

    enum Temp {HOT, COLD};

    static String drink(Temp t) {
        String drink;
        switch(t) {
            case HOT:
                drink = "Coffee";
                break;
            default:
                drink = "Beer";
                break;
        }
        return drink;
    }

    Moral: default clauses are a good thing.