

Part 2:

Strings and Enumerated Types

Lecture 01

Enums

(and a review of switch statements)

Enumerated values

- Sometimes you want a variable that can take on only a certain listed (enumerated) set of values
- Examples:
 - `dayOfWeek`: `SUNDAY`, `MONDAY`, `TUESDAY`, ...
 - `month`: `JAN`, `FEB`, `MAR`, `APR`, ...
 - `gender`: `MALE`, `FEMALE`
 - `title`: `MR`, `MRS`, `MS`, `DR`
 - `threadState`: `READY`, `RUNNING`, `BLOCKED`, `DEAD`
- The values are written in all caps because they are constants
- What is the actual type of these constants?

Enumerations

- In the past, enumerations were usually represented as integer values:
 - `public final int SPRING = 0;`
`public final int SUMMER = 1;`
`public final int FALL = 2;`
`public final int WINTER = 3;`
- This is a nuisance, and is error prone as well
 - `season = season + 1;`
 - `now = WINTER; ...; month = now;`
- Here's the new way of doing it:
 - `enum Season { WINTER, SPRING, SUMMER, FALL }`

Anti-pattern: int constants



```
public class Card {  
    public static final int CLUBS = 0;  
    public static final int DIAMONDS = 1;  
    public static final int HEARTS = 2;  
    public static final int SPADES = 3;  
  
    ...  
    private int suit;  
    ...  
    public void setSuit(int suit) {  
        this.suit = suit;  
    }  
}
```

- What's wrong with using `int` constants to represent card suits?
 - variation (also bad): using `Strings` for the same purpose.

Enumerated types

- **enum**: A type of objects with a fixed set of constant values.

```
public enum Name {  
    VALUE, VALUE, ..., VALUE  
}
```

- Usually placed into its own .java file.
- C has enums that are really `ints`; Java's are objects.

```
public enum Suit {  
    CLUBS, DIAMONDS, HEARTS, SPADES  
}
```

- **Effective Java Tip #30:** Use `enums` instead of `int` constants.

"The advantages of `enum` types over `int` constants are compelling. Enums are far more readable, safer, and more powerful."

What is an enum?

- The preceding enum is roughly equal to the following short class:

```
public final class Suit extends Enum<Suit> {  
    public static final Suit CLUBS      = new Suit();  
    public static final Suit DIAMONDS  = new Suit();  
    public static final Suit HEARTS    = new Suit();  
    public static final Suit SPADES     = new Suit();  
  
    private Suit() {}    // no more can be made  
}
```

What can you do with an enum?

- use it as the type of a variable, field, parameter, or return

```
public class Card {  
    private Suit suit;  
    ...  
}
```

- compare them with `==` (why don't we need to use `equals`?)

```
if (suit == Suit.CLUBS) { ...
```

- compare them with `compareTo` (by order of declaration)

```
public int compareTo(Card other) {  
    if (suit != other.suit) {  
        return suit.compareTo(other.suit);  
    } ...  
}
```


Enum methods

method	description
<code>int compareTo(E)</code>	all enum types are Comparable by order of declaration
<code>boolean equals(o)</code>	not needed; can just use <code>==</code>
<code>String name()</code>	equivalent to <code>toString</code>
<code>int ordinal()</code>	returns an enum's 0-based number by order of declaration (first is 0, then 1, then 2, ...)

method	description
<code>static E valueOf(s)</code>	converts a string into an enum value
<code>static E[] values()</code>	an array of all values of your enumeration

Enums are classes

- An **enum** is actually a new type of class
 - You can declare them as inner classes or outer classes
 - You can declare variables of an enum type and get type safety and compile time checking
 - Each declared value is an instance of the enum class
 - Enums are implicitly **public**, **static**, and **final**
 - You can compare enums with either **equals** or **==**
 - **enums** extend **java.lang.Enum** and implement **java.lang.Comparable**
 - Hence, enums can be sorted
 - Enums override **toString()** and provide **valueOf()**
 - Example:
 - **Season season = Season.WINTER;**
 - **System.out.println(season); // prints WINTER**
 - **season = Season.valueOf("SPRING"); // sets season to Season.SPRING**

Advantages of the new enum

- Enums provide compile-time type safety
 - `int` enums don't provide any type safety at all: `season = 43;`
- Enums provide a proper name space for the enumerated type
 - With `int` enums you have to prefix the constants (for example, `seasonWINTER` or `S_WINTER`) to get anything like a name space.
- Enums are robust
 - If you add, remove, or reorder constants, you must recompile, and then everything is OK again
- Enum printed values are informative
 - If you print an `int` enum you just see a number
- Because enums are objects, you can put them in collections
- Because enums are classes, you can add fields and methods

Enums have weird constructors

- Except for constructors, an **Enum** is an ordinary class
- Each name listed within an **Enum** is actually a call to a constructor
- Example:
 - `enum Season { WINTER, SPRING, SUMMER, FALL }`
 - This constructs the four named objects, using the default constructor
- Example 2:
 - ```
public enum Coin {
 private final int value;
 Coin(int value) { this.value = value; }
 PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
}
```
- Enum constructors are only available within the Enum itself
  - An enumeration is supposed to be complete and unchangeable

# Enums extend and inherit from Enum

- **String toString()** returns the name of this enum constant, as contained in the declaration
- **boolean equals(Object other)** returns true if the specified object is equal to this enum constant
- **int compareTo(E o)** compares this enum with the specified object for order; returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object
- **static enum-type valueOf(String s)** returns the enumerated object whose name is **s**
- **static enum-type [] values()** returns an array of the enumeration objects

# EnumSet

- class `EnumSet` from `java.util` represents a set of enum values and has useful methods for manipulating enums:

|                                                         |                                                               |
|---------------------------------------------------------|---------------------------------------------------------------|
| <code>static EnumSet&lt;E&gt; allOf (Type)</code>       | a set of all values of the type                               |
| <code>static EnumSet&lt;E&gt; complementOf (set)</code> | a set of all enum values other than the ones in the given set |
| <code>static EnumSet&lt;E&gt; noneOf (Type)</code>      | an empty set of the given type                                |
| <code>static EnumSet&lt;E&gt; of (...)</code>           | a set holding the given values                                |
| <code>static EnumSet&lt;E&gt; range (from, to)</code>   | set of all enum values declared between from and to           |

```
Set<Coin> coins = EnumSet.range(Coin.NICKEL, Coin.QUARTER);
for (coin c : coins) {
 System.out.println(c);
} // see also: EnumMap
```

- **Effective Java Tip #32:** Use `EnumSet` instead of bit fields.
- **Effective Java Tip #33:** Use `EnumMap` instead of ordinal indexing.

# The switch statement

```
switch (boolean test) {
 case value:
 code;
 break;
 case value:
 code;
 break;
 ...
 default: // if it isn't one of the above values
 code;
 break;
}
```

- an alternative to the `if/else` statement
  - must be used on integral types (e.g. `int`, `char`, `long`, **`enum`**)
  - instead of a `break`, a case can end with a `return`, or if neither is present, it will "fall through" into the code for the next case

# Syntax of the **switch** statement

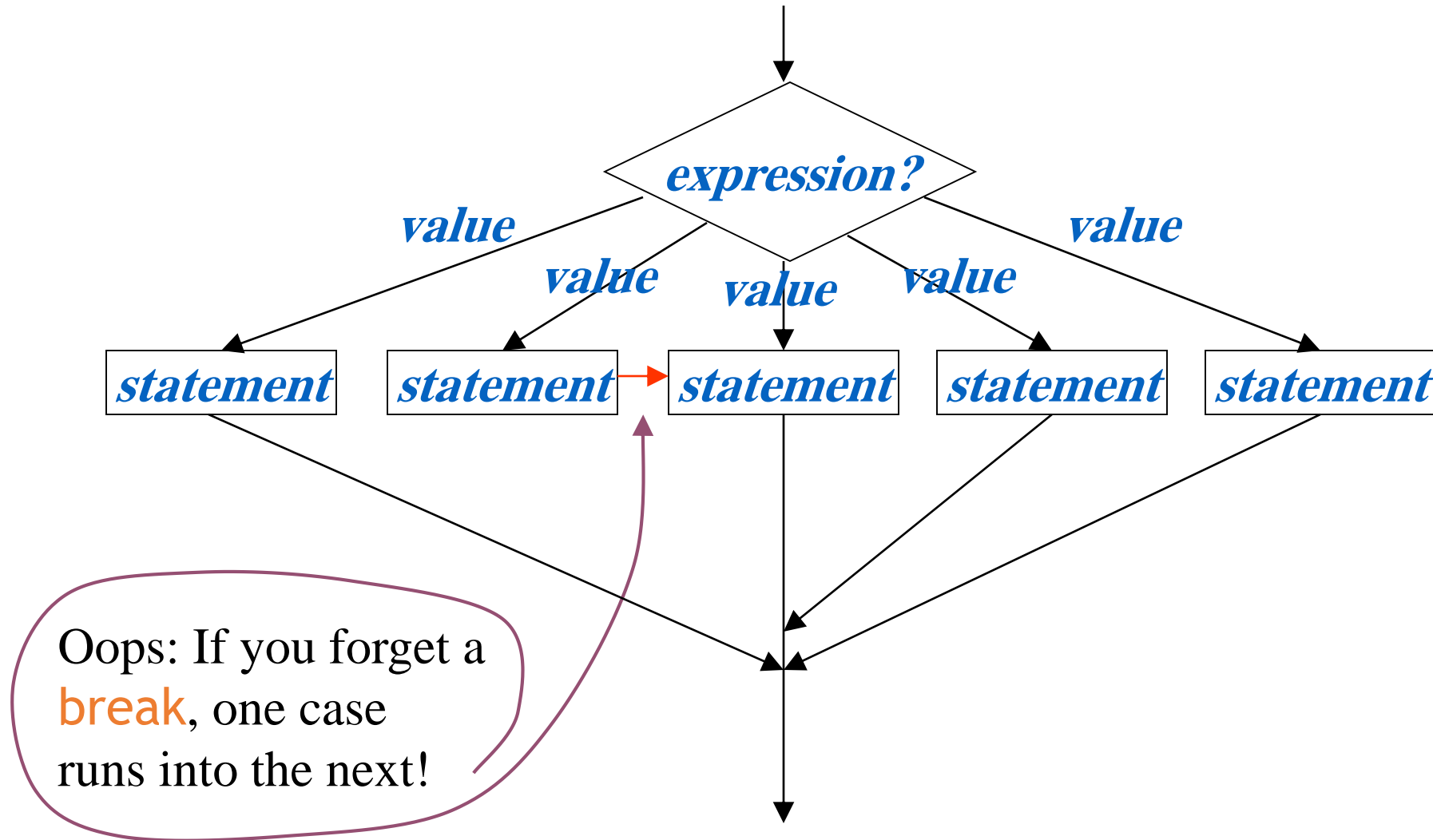
- The syntax is:

```
switch (expression) {
 case value1 :
 statements ;
 break ;
 case value2 :
 statements ;
 break ;
 ...(more cases)...
 default :
 statements ;
 break ;
}
```

- The *expression* must yield an integer or a character
- Each *value* must be a literal integer or character
- Notice that colons ( : ) are used as well as semicolons
- The last statement in every case should be a *break*;
  - I even like to do this in the *last* case
- The *default*: case handles every value not otherwise handled
  - The default case is usually last, but doesn't have to be



# Flowchart for **switch** statement



# Example **switch** statement

```
switch (cardValue) {
 case 1:
 System.out.print("Ace");
 break;
 case 11:
 System.out.print("Jack");
 break;
 case 12:
 System.out.print("Queen");
 break;
 case 13:
 System.out.print("King");
 break;
 default:
 System.out.print(cardValue);
 break;
}
```

# Enums and the switch statement

- **switch** statements can now work with enums
  - The switch variable evaluates to some enum value
  - The values for each case must (as always) be constants
- **switch** (*variable*) { **case** *constant*: ...; }
- In the switch constants, do not give the class name—that is, you *must* say **case SUMMER:**, *not* **case Season.SUMMER:**
- It's still a very good idea to include a default case

# Example enum and switch

```
public void tellItLikeItIs(DayOfWeek day) {
 switch (day) {
 case MONDAY:
 System.out.println("Mondays are bad.");
 break;
 case FRIDAY:
 System.out.println("Fridays are better.");
 break;
 case SATURDAY:
 case SUNDAY:
 System.out.println("Weekends are best.");
 break;
 default:
 System.out.println("Midweek days are so-so.");
 break;
 }
}
```

Source: <http://java.sun.com/docs/books/tutorial/java/javaOO/enum.html>

# More complex enums

- An enumerated type can have fields, methods, and constructors:

```
public enum Coin {
 PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

 private int cents;

 private Coin(int cents) {
 this.cents = cents;
 }

 public int getCents() { return cents; }
 public int perDollar() { return 100 / cents; }
 public String toString() {
 // "NICKEL (5c)"
 return super.toString() + " (" + cents + "c)";
 }
}
```

# Value-specific enum methods

```
// These are the the opcodes that our stack machine can execute.
abstract static enum Opcode {
 PUSH(1),
 ADD(0),
 BEZ(1); // Remember the required semicolon after last enum value
 int numOperands;
 Opcode(int numOperands) { this.numOperands = numOperands; }

 public void perform(StackMachine machine, int[] operands) {
 switch(this) {
 case PUSH: machine.push(operands[0]); break;
 case ADD: machine.push(machine.pop() + machine.pop()); break;
 case BEZ: if (machine.pop() == 0) machine.setPC(operands[0]); break;
 default: throw new AssertionError();
 }
 }
}
```

From: <http://snipplr.com/view/433/valuespecific-class-bodies-in-an-enum/>