Java Programming Tutorial
Enum (Enumeration)

TABLE OF CONTENTS (HIDE)

1. Introductin to Enumeration (JDK 1.5)

1.1 Example: Scissor-Paper-Stone

Suppose that we are writing a Scissor-Paper-Stone game. We could use three arbitrary integers (e.g., 0, 1, 2; or 88, 128, 168), three inefficient strings ("Scissor", "Paper", "Stone"), or three characters ('s', 'p', 't') to represent the three hand-signs. The main drawback is we need to check the other infeasible values (e.g. 4, "Rock", 'q', etc.) in our program to ensure correctness.

A better approach is to define our own list of permissible items in a construct called enumeration (or `enum`), introduced in JDK 1.5. The syntax is as follows:

```
enum {
   ITEM1, ITEM2, ...
}
```

For example,

```
enum HandSign {
   SCISSOR, PAPER, STONE
}
```

An enumeration is a *special class*, which provides a *type-safe* implementation of constant data in your program. In other words, we can declare a variable of the type `HandSign`, which takes values of either `HandSign.SCISSOR`, `HandSign.PAPER`, or `HandSign.STONE`, but NOTHING ELSE. For example,

```
HandSign playerMove;             // Declare variables of the enum type HandSign
HandSign computerMove;
playerMove = HandSign.SCISSOR;   // Assign values into enum variables
computerMove = HandSign.PAPER;
// playerMove = 0;               // Compilation error
```

Example: Below is a Scissor-Paper-Stone game using an enumeration.

```
import java.util.Random;
import java.util.Scanner;

/*
 * Define an enumeration called Sign, with 3 elements, referred to as:
 * HandSign.SCISSOR, HandSign.PAPER, HandSign.STONE.
 */
enum HandSign {
   SCISSOR, PAPER, STONE
}

/*
 * A game of scissor-paper-stone.
 */
public class ScissorPaperStone {
   public static void main(String[] args) {
      Random random = new Random();   // Create a random number generator
      boolean gameOver = false;
      HandSign playerMove = HandSign.SCISSOR;
      HandSign computerMove;
      int numTrials = 0;
      int numComputerWon = 0;
      int numPlayerWon = 0;
      int numTie = 0;
```

```
      Scanner in = new Scanner(System.in);
      System.out.println("Let us begin...");

      while (!gameOver) {
         System.out.printf("%nScissor-Paper-Stone");

         // Player move
         // Use a do-while loop to handle invalid input
         boolean validInput;
         do {
            System.out.print("   Your turn (Enter s for scissor, p for paper, t for stone, q to quit): ");
            char inChar = in.next().toLowerCase().charAt(0); // Convert to lowercase and extract first char
            validInput = true;
            if (inChar == 'q') {
               gameOver = true;
            } else if (inChar == 's') {
               playerMove = HandSign.SCISSOR;
            } else if (inChar == 'p') {
               playerMove = HandSign.PAPER;
            } else if (inChar == 't') {
               playerMove = HandSign.STONE;
            } else {
               System.out.println("   Invalid input, try again...");
               validInput = false;
            }
         } while (!validInput);

         if (!gameOver) {
            // Computer Move
            int aRandomNumber = random.nextInt(3); // random int between 0 (inclusive) and 3 (exclusive)
            if (aRandomNumber == 0) {
               computerMove = HandSign.SCISSOR;
               System.out.println("   My turn: SCISSOR");
            } else if (aRandomNumber == 0) {
               computerMove = HandSign.PAPER;
               System.out.println("   My turn: PLAYER");
            } else {
               computerMove = HandSign.STONE;
               System.out.println("   My turn: STONE");
            }

            // Check result
            if (computerMove == playerMove) {
               System.out.println("   Tie!");
               numTie++;
            } else if (computerMove == HandSign.SCISSOR && playerMove == HandSign.PAPER) {
               System.out.println("   Scissor cuts paper, I won!");
               numComputerWon++;
            } else if (computerMove == HandSign.PAPER && playerMove == HandSign.STONE) {
               System.out.println("   Paper wraps stone, I won!");
               numComputerWon++;
            } else if (computerMove == HandSign.STONE && playerMove == HandSign.SCISSOR) {
               System.out.println("   Stone breaks scissor, I won!");
               numComputerWon++;
            } else {
               System.out.println("   You won!");
               numPlayerWon++;
            }
            numTrials++;
         }
      }

      // Print statistics
      System.out.printf("%nNumber of trials: " + numTrials);
      System.out.printf("I won %d(%.2f%%). You won %d(%.2f%%).%n", numComputerWon,
            100.0*numComputerWon/numTrials, numPlayerWon, 100.0*numPlayerWon/numTrials);
      System.out.println("Bye! ");
   }
}
```

```
Let us begin...
Scissor-Paper-Stone
   Your turn (Enter s for scissor, p for paper, t for stone, q to quit): s
   My turn: SCISSOR
   Tie!

Scissor-Paper-Stone
   Your turn (Enter s for scissor, p for paper, t for stone, q to quit): s
   My turn: STONE
   Stone breaks scissor, I won!
```

```
Scissor-Paper-Stone
   Your turn (Enter s for scissor, p for paper, t for stone, q to quit): p
   My turn: STONE
   You won!

Scissor-Paper-Stone
   Your turn (Enter s for scissor, p for paper, t for stone, q to quit): t
   My turn: SCISSOR
   Scissor cuts paper, I won!

Scissor-Paper-Stone
   Your turn (Enter s for scissor, p for paper, t for stone, q to quit): a
   Invalid input, try again...
   Your turn (Enter s for scissor, p for paper, t for stone, q to quit): p
   My turn: STONE
   You won!

Scissor-Paper-Stone
   Your turn (Enter s for scissor, p for paper, t for stone, q to quit): q

Number of trials: 5
I won 2(40.00%). You won 2(40.00%).
Bye!
```

Note that I used the utility `Random` to generate a random integer between 0 and 2, as follows:

```
import java.util.Random;      // Needed to use Random

// In main()
Random random = new Random(); // Create a random number generator
rand.nextInt(3);              // Each call returns a random int between 0 (inclusive) and 3 (exclusive)
```

1.2 Examples: Card Suit

A card's suit can only be spade, diamond, club or heart. In other words, it has a limited set of values. Before the introduction of `enum` type in JDK 1.5, we usually have to use an `int` variable to hold these values. For example,

```
class CardSuit {
   public static final int SPADE   0;
   public static final int DIAMOND 1;
   public static final int CLUB    2;
   public static final int HEART   3;
   ......
}
class Card {
   int suit;   // CardSuit.SPADE, CardSuit.DIAMOND, CardSuit.CLUB, CardSuit.HEART
}
```

The drawbacks are:

- It is not *type-safe*. You can assign any `int` value (e.g., 88) into the `int` variable `suit`.
- No namespace: You must prefix the constants by the class name `CardSuit`.
- Brittleness: new constants will break the existing codes.
- Printed values are uninformative: printed value of 0, 1, 2 and 3 are not very meaningful.

JDK 1.5 introduces a new `enum` type (in addition to the exisitng top-level constructs `class` and `interface`) along with a new keyword `enum`. For example, we could define:

```
enum Suit { SPADE, DIAMOND, CLUB, HEART }
```

An `enum` can be used to define a set of `enum` constants. The constants are implicitly `static final`, which cannot be modified. You could refer to these constants just like any `static` constants, e.g., `Suit.SPADE`, `Suit.HEART`, etc. `enum` is *type-safe*. It has its own *namespace*. `enum` works with switch-case statement (just like the exisitng `int` and `char`).

For example,

```
1  import java.util.*;
2
3  enum Suit { SPADE, DIAMOND, CLUB, HEART }
4  enum Rank { ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING }
5
6  class Card {  // A card
7     private Suit suit;
8     private Rank rank;
9
10    Card(Suit suit, Rank rank) {   // constructor
```

```
      this.suit = suit;
      this.rank = rank;
   }

   Rank getRank() { return rank; }
   Suit getSuit() { return suit; }
   public String toString() { return "This card is " + rank + " of " + suit; }
}

class CardDeck {  // A deck of card
   List<Card> deck;
   CardDeck() {   // constructor
      deck = new ArrayList<Card>();
      for (Suit suit : Suit.values()) {
         for (Rank rank : Rank.values()) {
            deck.add(new Card(suit, rank));
         }
      }
   }
   public void print() {
      for (Card card : deck) System.out.println(card);   // print all cards
   }
   public void shuffle() {
      Collections.shuffle(deck);  // use java.util.Collections' static method to shuffle the List
   }
}

public class CardTest {
   public static void main(String[] args) {
      CardDeck deck = new CardDeck();
      deck.print();
      deck.shuffle();
      deck.print();
   }
}
```

For each `enum`, the Java compiler automatically generates a `static` method called `values()` that returns an array of all the `enum` constants, in the order they were defined.

2. More on Enumeration

2.1 Constructor, Member Variables and Methods

An `enum` is a *reference type* (just like a class, interface and array), which holds a reference to memory in the heap. It is implicitly `final`, because the constants should not be changed. It can include other component of a traditional class, such as constructors, member variables and methods. (This is where Java's `enum` is more powerful than C/C++'s counterpart). Each `enum` constant can be declared with parameters to be passed to the constructor when it is created. For example,

```
enum TrafficLight {
   RED(30), AMBER(10), GREEN(30);  // Named constants

   private final int seconds;      // Private variable

   TrafficLight(int seconds) {     // Constructor
      this.seconds = seconds;
   }

   int getSeconds() {              // Getter
      return seconds;
   }
}

public class TrafficLightTest {
   public static void main(String[] args) {
      for (TrafficLight light : TrafficLight.values()) {
         System.out.printf("%s: %d seconds\n", light, light.getSeconds());
      }
   }
}
```

Three instances of `enum` type `TrafficLight` were generated via `values()`. The instances are created by calling the constructor with the actual argument, when they are first referenced. You are not allowed to construct a new instance of `enum` using `new` operator, because `enum` keeps a fixed list of constants. `enum`'s instances could have its own instance variable (`int seconds`) and method (`getSeconds()`).

2.2 Enum Constructor and Abstract Method

```
enum TLight {
   // Each instance provides its implementation to abstract method
   RED(30) {
      public TLight next() {
         return GREEN;
      }
   },
   AMBER(10) {
      public TLight next() {
         return RED;
      }
   },
   GREEN(30) {
      public TLight next() {
         return AMBER;
      }
   };

   public abstract TLight next(); // An abstract method

   private final int seconds;     // Private variable

   TLight(int seconds) {          // Constructor
      this.seconds = seconds;
   }

   int getSeconds() {             // Getter
      return seconds;
   }
}

public class TLightTest {
   public static void main(String[] args) {
      for (TLight light : TLight.values()) {
         System.out.printf("%s: %d seconds, next is %s\n", light,
               light.getSeconds(), light.next());
      }
   }
}
```

Each of the instances of `enum` could have its own behaviors. To do this, you can define an `abstract` method in the `enum`, where each of its instances provides its own implementation.

Another Example

```
enum Day {
   MONDAY(1) {
      public Day next() { return TUESDAY; }   // each instance provides its implementation to abstract method
   },
   TUESDAY(2) {
      public Day next() { return WEDNESDAY; }
   },
   WEDNESDAY(3) {
      public Day next() { return THURSDAY; }
   },
   THURSDAY(4) {
      public Day next() { return FRIDAY; }
   },
   FRIDAY(5) {
      public Day next() { return SATURDAY; }
   },
   SATURDAY(6) {
      public Day next() { return SUNDAY; }
   },
   SUNDAY(7) {
      public Day next() { return MONDAY; }
   };

   public abstract Day next();

   private final int dayNumber;

   Day(int dayNumber) {   // constructor
      this.dayNumber = dayNumber;
   }

   int getDayNumber() {
      return dayNumber;
```

```
   }
}

public class DayTest {
   public static void main(String[] args) {
      for (Day day : Day.values()) {
         System.out.printf("%s (%d), next is %s\n", day, day.getDayNumber(), day.next());
      }
   }
}
```

2.3 java.util.EnumSet, java.util.EnumMap

Two classes have been added to `java.util` to support `enum`: `EnumSet` and `EnumMap`. They are high performance implementation of the `Set` and `Map` interfaces respectively.

[TODO]

3. Summary

So when should you use `enum`s? Any time you need a fixed set of constants, whose values are known at compile-time. That includes natural enumerated types (like the days of the week and suits in a card deck) as well as other sets where you know all possible values at compile time, such as choices on a menu, command line flags, and so on. It is not necessary that the set of constants in an enum type stays fixed for all time. In most of the situations, you can add new constants to an enum without breaking the existing codes.

Properties:

1. Enums are type-safe!
2. Enums proivde their namespace.
3. Whenever an `enum` is defined, a class that extends `java.lang.Enum` is created. Hence, `enum` cannot extend another `class` or `enum`. The compiler also create an instance of the class for each constants defined inside the `enum`. The `java.lang.Enum` has these methods:

```
public final String name();  // Returns the name of this enum constant, exactly as declared in its enum declaration.
                             // You could also override the toString() to provide a more user-friendly description.
public String toString();    // Returns the name of this enum constant, as contained in the declaration.
                             // This method may be overridden.
public final int ordinal();  // Returns the ordinal of this enumeration constant.
```

4. All constants defined in an enum are `public static final`. Since they are `static`, they can be accessed via *`EnumName.instanceName`*.
5. You do not instantiate an `enum`, but rely the constants defined.
6. Enums can be used in a switch-case statement, just like an `int`.

LINK TO JAVA REFERENCES & RESOURCES

Latest version tested: JDK 1.7.0_03
Last modified: May, 2012