Enumerated Types

The atomic values in a programming language include numbers, boolean values, and characters.

However, to solve many problems we need to have atomic values that represent other concepts such as the days of the week, the colors of the spectrum, the kinds of employees, and so on.

These types of values can be represented by normal identifiers (Monday, Tuesday, red, blue, manager, and so on) and are known as *enumerated values*.

Prior to version 1.5, Java expected a programmer to use integer constants for enumerated values.

Example: Suit Values in a Card Class

```
static final int CLUBS = 1;
static final int DIAMONDS = 2;
static final int HEARTS = 3;
static final int SPADES = 4;
```

Now the program can use the identifiers CLUBS, DIAMONDS, HEARTS, and SPADES to represent the four suit values inside of the Card class.

Outside of the Card class, Card.CLUBS, Card.DIAMONDS, Card.HEARTS, and Card.SPADES must be used.

Java API

Enumerated values as integer constants are found thoughout the Java library of classes and interfaces.

Identifier	Value
Font.PLAIN	0
Font.BOLD	1
Font.ITALIC	2
Calendar.SUNDAY	1
Calendar.MONDAY	2
Calendar.TUESDAY	3
Calendar.WEDNESDAY	4
Calendar.THURSDAY	5
Calendar.FRIDAY	6
Calendar.SATURDAY	7
Calendar.JANUARY	0
Calendar.FEBRUARY	1
Calendar.MARCH	2
Calendar.APRIL	3
Calendar.MAY	4
:	:
Calendar.NOVEMBER	10
Calendar.DECEMBER	11
WindowConstants.DO_NOTHING_ON_CLOSE	0
WindowConstants.HIDE_ON_CLOSE	1
WindowConstants.DISPOSE_ON_CLOSE	2
WindowConstants.EXIT_ON_CLOSE	3

WindowConstants is an interface containing the fours constants; this interface is implemented by JFrame.

Problems with Integers as Enumerated Values

Using integer constants for enumerated values is an idea with many flaws.

- The compiler and the run time system think the identifiers are integers, not special values, and the type checking treats them the same as any other integers.

- Operations that use these enumerated values are not type safe.

  The following operations are legal at compile time and at run time even though they make no sense at all.

```
    int m = Calendar.MONDAY * Font.BOLD;

    int n = WindowConstants.EXIT_ON_CLOSE / m;

    double d = Math.sqrt(Calendar.DECEMBER + 100);

    Card Constructor:
        Card(int r, int s)
        {  rank = r;  suit = s;  }

        Card c1 = new Card(7, Card.SPADES);      // okay

        Card c2 = new Card(3, -77);              // nonsensical

    int s1 = Card.CLUBS;
    int s2 = Card.HEARTS;

    System.out.println(s1 * s2 - Font.ITALIC + Calendar.MAY);
```

  Sometimes, however, adding enumerated values makes sense.

```
    Font f1 = new Font("Serif", Font.BOLD+Font.Italic, 12);
```

  But what about this one?

```
    Font f2 = new Font("Serif", Font.PLAIN+Font.BOLD, 12);
```

Many of these operations are contrary to the intended meaning of the enumerated values, which define a new type in a sense, but neither the compiler nor the run time system will detect any type errors with these erroneous operations.

- We need to write code to display enumerated values since they will display as their integer values if we print them directly.

```
System.out.println(Card.SPADES)  =>  4
```

We need a conversion (toString) function.

```
static String mkString(int s)
{
        switch (s)
        {
                case CLUBS :        return "Clubs";
                case DIAMONDS :     return "Diamonds";
                case HEARTS :       return "Hearts";
                case SPADES :       return "Spades";
                default :           return "Oops";
        }
}
```

```
System.out.println(mkString(Card.SPADES))  =>  Spades
```

- Defining enumerated values as integer constants creates "brittle" code in the sense that changes in the set of values defined or even the class in which they are defined will require the code that uses the values to be recompiled.

All in all, implementing enumerated value with integer constants is a bad idea, even though it is found in other languages such as C and C++.

Answer: Java 1.5 Enums

Java has several kinds of types that can be used to declare variables and method signatures.

Eight primitive types

Array types

Class types

Interface types

Java 1.5 has added a new keyword **enum** that can be used to define a finite set of enumerated values as a new type whose values are legal Java identifiers.

Example: Suit

```
enum Suit
{
      Clubs, Diamonds, Hearts, Spades;
}
```

Now Suit is a valid type in Java, and its values *are* the identifiers *Clubs*, *Diamonds*, *Hearts*, and *Spades*.

Inside of the Suit definition, these identifiers can be referred to directly, but from outside of the definition we need to use the qualified identifiers *Suit.Clubs*, *Suit.Diamonds*, *Suit.Hearts*, and *Suit.Spades*.

Variables can be declared of type Suit, and these variables can only take one of the four values defined for Suit.

```
Suit s = Clubs;
s = Spades;
if (s == Diamonds) ...
```

Observe that an **enum** definition is similar in structure to a class or interface definition.

Minimum Code

1. Keyword **enum**.
2. Name of the new type.
3. List of possible values, separated by commas, terminated by a semicolon.

Properties and Features of enums

- **enum** values are type safe—they can be used only as designed.
- Variables and values of an **enum** type are checked by the compiler (static type checking).
- **enum** values are *not* integers, even though they might be implemented as integers. Actually, they are implemented as references to constant objects.
- **enum** values are automatically **public**, **static**, and **final**.
- **enum** values can be compared using == or using *equals*, and the results will be the same.
- **enum** definitions implicitly extend the class java.lang.Enum and implement java.lang.Comparable.
- The instance method *toString* is supplied automatically, inherited from Enum.

  ```
  Suit.Spades.toString()  =>  "Spades"
  ```
- The class method *valueOf* is also supplied by Enum.

  ```
  Suit.valueOf("Hearts")  =>  Suit.Hearts
  ```
- The class method *Suit.values* with no parameters returns an array containing the values of this **enum** type in the order they were defined.

Card and CardDeck

To illustrate the use of enumerated values, we define these two classes with the integer values for the suit and rank of a card replace by **enum** types.

First we define the two **enum** types used to specify the Card objects.

File: Suit.java

```
enum Suit
{
     Clubs, Diamonds, Hearts, Spades;

}
```

File: Rank.java

```
enum Rank        // note the order of the values
{
     Two, Three, Four, Five, Six,
     Seven, Eight, Nine, Ten, Jack,
     Queen, King, Ace;
}
```

File: Card.java

```
class Card implements Comparable
{
     Card(Rank r, Suit s)
     {
          Rank = r;
          suit = s;
     }
```

```
    Rank get Rank()
    {
        return rank;
    }

    Suit getSuit()
    {
        return suit;
    }

    public int compareTo(Object ob)
    {
        Card other = (Card)ob;
        Rank rank = getRank();
        Rank otherRank = other.getRank();
        return rank.compareTo(otherRank);
    }

    public String toString()
    {
        String s = rank + " of " + suit;
        return s + "                 ".substring(0, 17 - s.length());
    }

    private Rank rank;
    private Suit suit;
}
```

Notes on Card

- Two Card objects are compared using the *compareTo* method defined on the Rank values, which are ordered according to the order of their declaration. The Rank type has Ace as the highest value.
- The *toString* method employs the *toString* methods for Rank and Suit to produce a string of the form "Five of Clubs".

File: CardDeck.java

In this class we show only those methods that have changed.

Note that the *fill* method uses the new version of the **for** command to iterate through the Rank values and the Suit values.

```
class CardDeck
{
    CardDeck()
    {
        deck = new Card [52];
        fill();
    }

    private void fill()          // only method changed
    {
        int index = 0;
        for (Rank r : Rank.values())
            for (Suit s : Suit.values())
            {
                deck[index] = new Card(r, s);
                index++;
            }
        numCards = 52;
    }

    :
    :

    private Card [] deck;
    private int numCards;
}
```

Game of War

To test the new versions of Card and CardDeck, we take a look at the classes in the Game of War.

The only change necessary is in the *play* method of the Game class.

File: Game.java

```
class Game
{
    void play()
    {
        CardDeck cd = new CardDeck();
        cd.shuffle();
         :       :
loop:   for (int t=1; t<=100; t++)
        {
                :        :
            else                   // War
            {   down.clear();
                down.addCard(c1);     down.addCard(c2);
                boolean done = false;
                do
                {
                    int num = c1.getRank();   // error here
                        :        :
                }
                while (!done);
            }
            System.out.println(p1.numCards() + " to "
                                      + p2.numCards());
        }
    }

    :

    private Player p1, p2;
}
```

Now the *getRank* method in Card returns a Rank object instead of an **int** value.

Corrections

Change the incorrect line in the method *play* from

```
        int num = c1.getRank();
```

to

```
        int num = c1.getNum();
```

and add an instance method to Card.

```
    int getNum()
    {
        switch (value)
        {
            case Two      : return 2;
            case Three    : return 3;
            case Four     : return 4;
            case Five     : return 5;
            case Six      : return 6;
            case Seven    : return 7;
            case Eight    : return 8;
            case Nine     : return 9;
            case Ten      : return 10;
            case Jack     : return 11;
            case Queen    : return 12;
            case King     : return 13;
            case Ace      : return 1;
        }
        return 0;   // should never happen
    }
```

The behavior of this version of the War game will be identical to the first version, but now the compiler will type-check the use of the Suit and Rank values, so that many foolish errors will be caught early.

Another Version of War

The definition of an **enum** type allows a number of other features that make it much like a class definition.

Instance variables

Constructors

Method definitions

We use these features to associate an integer value with each of the **enum** values in the definition of Rank.

```
enum Rank
{
    Two(2), Three(3), Four(4), Five(5), Six(6),
    Seven(7), Eight(8), Nine(9), Ten(10), Jack(11),
    Queen(12), King(13), Ace(1);

    private int num;        // instance variable

    Rank(int n)             // constructor
    {
        num = n;
    }

    int getNum()            // instance method
    {
        return num;
    }
}
```

Now each of the values in the enumerated type Rank carries along a number that can be produced by calling the instance method *getNum*.

```
Rank.Jack.getNum()  =>  11
Rank.Ace.getNum()  =>  1
```

No changes need to be made in the Card and CardDeck classes.

But the *play* method in Game needs one line altered.

Change the incorrect line in the original *play* method from

```
        int num = c1.getRank();
to
        int num = c1.getRank().getNum();
```

In addition, we may remove the *getNum* method from the Card class because it is no longer needed.

Again, the behavior of the Game of War will be the same as with the previous versions.

Overloading

Java **enum** types allow the overloading of the values in the types.

```
    enum Weapons
    {
        Knives, Clubs, Guns;
    }
```

Static Imports

The **import** statement in Java programs is used to make classes and interfaces directly accessible (visible) without writing their fully qualified names, such as ArrayList instead of java.util.ArrayList.

Java 1.5 has added another kind of **import** statement, called a **static import**, that can be used to make class variables and class methods directly accessible.

Example

To import the **enum** definitions, we need to put them in a package (and a subdirectory), say *enums*.

```
package enums;
public enum Suit { Clubs, Diamonds, Hearts, Spades; }

package enums;
public enum Weapons { Knives, Clubs, Guns; }
```

Test Program

```
import static enums.Suit.*
import static enums.Weapons.*
import static java.lang.System.out;
import static java.lang.Math.*;

public class StImport
{
    public static void main(String [] args)
    {
        out.println("abs(-45.6) = " + abs(-45.6));
        out.println("Spades = " + Spades);
        out.println("Knives = " + Knives);
        out.println("Clubs = " + enums.Suit.Clubs);
    }
}
```