- Enums are typesafe. Introduced in JDK 1.5.

- Enum name can be any java identifier. Following is valid enum declaration –

```
enum Boolean{
        …..
        …..
}
```

- The string representation of an enum constant is its name.

- We can declare an Enum type inside the class.

- Separate .class file is generated for enums.

- enum 'A' extends java.lang.Enum.

- java.lang.Enum is an abstract class.

```
public abstract class Enum extends Object implements Comparable, Serializable{
……..
}
```

- This clearly means that enums are **comparable and serializable implicitly**.

- Enums are singleton by default.

- For enums, equals() & '==' do the same thing and can be used interchangeably.

- Enum constructors can be private or default and public, protected constructors are not allowed. When constructor is declared, all enum constant declaration must match with a constructor.

- Enum constants are in fact final, static variables of the enum type, and they are implicitly initialized with objects of the enum type when the enum type is loaded at runtime.

- The list of enum constants must be terminated by a semi-colon (;). Each enum constant name can be followed by an argument list that is passed

to the constructor of the enum type having the matching parameter signature.

- Since the enum constants are static members, they can be accessed using the name of the enum type-analogous to accessing static members in a class.

- When the enum type is loaded at runtime, the constructor is run for each enum constant, passing the argument values specified for the enum constant. For the a enum type, If that enum type contains 'N' constants then total N objects are created that are initialized with the specified argument values, and are referenced by the those enum constants, respectively.

- Note that each enum constant is a final, static reference that stores the reference value of an object of the enum type, and methods of the enum type can be called on this object by using the enum constant name.

- It can declare constructors and other members as in an ordinary class, but the enum constants must be declared before any other declarations.

- An implicit standard constructor is created if no constructors are provided for the enum type. An enum type cannot be instantiated using the new operator. The constructors cannot be called explicitly. The only accessibility modifier allowed for a constructor is private.

- Constructor chaining can be done in enum types hence this() can be used inside enum type constructors.

- We can declare an interface inside enum as -

```
public class InterfaceInsideEnum {
        public enum Season{
            SPRING, SUMMER, FALL, WINTER;
            interface A2{
                    void print();
                    void print1();
            }
    }
}
```

And implement the methods in other class as –

```
class A3 implements InterfaceInsideEnum.Season.A2 {
    @Override
    public void print(){
            System.out.println("print");
    }
    @Override
    public void print1() {
            System.out.println("print1");
    }
}
```

- If we are using interface inside an Enum it is not possible to implement the methods within the same class.

- Inside Enum we cannot declare abstract class and inside abstract class we can declare Enum as –

```
abstract class A3{
        enum Days{
            SUNDAY, SATURDAY;
        }
}
public class A2 extends A3{
        public static void main(String[] args) {
                Days d1 = Days. SUNDAY;
                Days d2 = Days. SATURDAY;
                System.out.println(d1);
                System.out.println(d2.ordinal());
        }
}
```

- Inside Enum we can declare Init block, Static block, Static members and Instance members.

- Enums can not be declared inside method or any block.

- We can use Enum type in switch case also like int and char values.

Implicit Static Methods for Enum Types

- static *EnumTypeName*[] values()

Returns an array containing the enum constants of this enum type, in the order they are specified.

- static *EnumTypeName* valueOf(String name)

Returns the enum constant with the specified name. An IllegalArgumentException is thrown if the specified name does not match the name of an enum constant. The specified name is *not* qualified with the enum type name.

Inherited Methods from the Enum Class

- protected final Object clone()

An instance of an enum type can*not* be cloned. The method throws an CloneNotSupportedException.

- final int compareTo(E o)

The *natural order* of the enum constants in an enum type is according to their *ordinal values* (see the ordinal() method below). The compareTo() method in the Comparable interface.

- final boolean equals(Object other)

This method returns true if the specified object is equal to this enum constant.

- protected final void finalize()

An enum constant cannot be finalized, because this final method effectively prevents enum types from implementing their own finalize() method.

- final Class<E> getDeclaringClass()

This method returns the Class object corresponding to this enum constant's enum type.

- final int hashCode()

This method returns a hash code for this enum constant.

- final String name()

This method returns the name of this enum constant, exactly as declared in its enum declaration.

- final int ordinal()

This method returns the *ordinal value* of this enum constant (that is, its position in its enum type declaration). The first enum constant is assigned an ordinal value of zero. If the ordinal value of an enum constant is less than the ordinal value of another enum constant of the same enum type, the former occurs before the latter in the enum type declaration.

Extending Enum Types: Constant-Specific Class Bodies

- Constant-specific class bodies define anonymous classes inside an enum type, i.e., they implicitly extend the enclosing enum type.

```
BREAKFAST(7,30) {      // (1) Start of constant-specific class body
public double mealPrice(Day day) {     // (2) Overriding abstract method
...
}
public String toString() {
// (3) Overriding method from the Enum
class
...
}
}   // (4) End of constant-specific class body
```

- The constant-specific class body, as the name implies, is a class body that is specific to a particular enum constant.

- Constant-specific class body is an anonymous class, i.e., a class with no name. It inherits members of the enclosing enum supertype, that are not private, overridden, or hidden. When the enum type Meal is loaded at

runtime, this constant-specific class body is instantiated, and the reference value of the instance is assigned to the corresponding enum constant.

- Each enum constant has to override the abstract method (if defined any, In enum type) inside constant-specific class body. Otherwise it will report compile-time error.

- Constructors, abstract methods, and static methods cannot be declared in a constant-specific-class body.

- Instance methods declared in constant-specific class bodies are only accessible if they override methods in the enclosing enum supertype.

- When nested, it is implicitly static, and can be declared with the keyword static.

```
public class MealPrices {
        public enum Day { /* ... */ }                          // Static member
        public static enum Meal { /* ... */ }                   // Static member
        public static void main(String[] args) { /* ... */ }      // Static method
}
```

- An enum type cannot be explicitly extended using the extends clause. An enum type is implicitly final, unless it contains constant-specific class bodies. If it declares constant-specific class bodies, it is implicitly extended. No matter what, it cannot be explicitly declared final.

- An enum type cannot be declared abstract, regardless of whether each abstract method is overridden in the constant-specific class body of every enum constant.

- Enum can't extend class, but it can implement interfaces.

```
public interface ITimeInfo {
    public int getHour();
    public int getMins();
}
public enum Meal implements ITimeInfo {
```

```
    // ...
    public int getHour() { return this.hh; }
    public int getMins() { return this.mm; }
}
```

- Set and Map provide better performance while using with enum types.