Contents

Enumeration

Java provides what is often also provided by other language like C or C++, which is often called `enumeration` a way for a program to define a set of constants, which are logically related, however unlike most other languages where enumeration is basically an integer or ordinal number in Java the enumeration or enum is actually a type, more specifically a class. That provides a lot of benefits, since we can treat enums like classes with methods and even instance variables.

`Most importantly have to remember that Java enumerations are instances of the enum which itself is a simple class types, the run time does not treat them in any special ways, they inherit from the Enum class implicitly`

Declaration

To declare an enum in java, the most simplest way is to use the enum keyword and block inside of which the enums are listed. Note that those are separated by a comma, they can be on a single or multi line. Since enumerations are classes, the list of enums defined inside are actually instances of the enumeration class itself, in essence we create instances of `Apples`, which are immutable (kind of) and also defined at compile/source time. Also by default the enum instances defined in an enumeration are `public static final`, but that is implicitly used and deduced by the compiler, the enum class or type itself is by default implicitly also defined as `public static`

Usually it is recommended that the enumeration constants are either all upper case or follow the standard type naming convention which is PascalCase.

```
public enum Apples {
    Johnatan, GoldenDel, RedDel, Winesap, Cortland
}
```

Even though by default they are implicitly `public static final` we are NOT allowed to define any modifiers to the enumerated entries inside enum

```
public enum Apples {
    public static Johnatan, GoldenDel, RedDel, Winesap, Cortland //
       compile time error, no user specified modifiers are allowed
}
```

Usage

Since enumerations are instances of a class type / enumeration type, to access elements of it the type of enumeration has to be referenced first

```
    Apples ap = Apples.RedDel
```

Enumerations are compile time constants, and as such they can be used in a switch statement, inside a switch statement we can always use a constant, i.e an identifier that does not evaluate to expression

```
Apples ap = ....

switch(ap) {
    case GoldenDel: { // ap is of type Apples, no need to prefix it with
       Apples.[enumeration]
        break;
    }
    case RedDel: { // ap is of type Apples, no need to prefix it with
       Apples.[enumeration]
        break;
    }
    default { // default case
    }
}
```

Restrictions

There are some restrictions enforced on java annotations which if not met will produce a compile time error.

- they can not be generic or accept or be declared with generic type parameters
- annotations can not inherit other annotations (e.g. with extends keyword)
- all methods of an annotation must return a type and accept no arguments
- annotation method return types can be primitive, String or Class, enum, annotation or an array of the all fore mentioned

Comparison

As already mentioned the enumerations within an enum type are defined at compile time, meaning that each time we use the [enum].enumeration combination we refer to an instance of the enum class, however since we refer to the same instance it is possible to compare them with ==, yes, the == still does a reference compare however since they are immutable the [enum].enumeration always refers to the same enumeration from enum no matter at which point of the program it is used

```
Apples apOne = Apples.GoldenDel;
Apples apTwo = Apples.GoldenDel;
assert(apOne == apTwo);
```

This works since all references to an enumeration, refer to the very same object instance which was initially created, and it lives in the JVM, at run-time. Meaning that enums are not treated with any special kind of attention when it comes to the usual operations that can be performed on them, they are simply a convenient way and a convention to create an immutable final static instance of a class which can be globally referred to.

Methods

Every enumeration in an enum type, implicitly inherits from the Enum class in java.lang, this class has a few methods which are exposed to be used with a specific enumeration such as

- ordinal - extract the ordinal, in other words the position of enumeration in the list of enumerations in the enum type, as defined in source, starting from 0

  ```
  Apples.GoldeDel.ordinal() // this will produce an integer result of 1
     (second entry/enumeration in Apples enum type, counting from 0)
  ```

- compareTo - meant to compare one enumeration to another, be careful it is comparing the ordinal of the current instance with another enumeration's ordinal, then it returns this.ordinal - that.ordinal, meaning that it will be negative if current ordinal is less than other.ordinal, positive otherwise, equal if both have the same ordinal

  ```
  Apples apOne = Apples.GoldenDel;
  Apples apTwo = Apples.RedDel;
  assert(apOne.compareTo(apTwo) < 1); // in the example above GoldenDel
     comes right before the RedDel, therefore GoldenDel's ordinal is smaller
     than RedDel's
  ```

- values - extract all values of the given enum type, note that this method also exists on the enum type as well as on the enumeration and produces the same result

  ```
  Apples[] apples = Apples.RedDel.values(); // valid exists on the
     enumeration itself
  Apples[] applesAgain = Apples.values(); // valid, exists on the type
     itself too
  ```

- valueOf - can either take a name (of the enumeration target) or can take a class and enumeration target name, also exists on both an enumeration and it's type

  ```
  Apples apTarget = Apples.RedDel.valueOf("RedDel"); // valid exists on the
     enumeration itself
  Apples apTarget = Apples.valueOf("RedDel"); // valid, exists on the type
     itself

  Apples apTarget = Apples.valueOf(Apples.class, "RedDel"); // also valid,
     but not really useful in general
  Apples apTarget = Enum.valueOf(Apples.class, "RedDel"); // valid exists on
     the base enum class, obviously
  ```

-

Members

Since already established that enumerations are instances of the enum, which is a normal java class type, it is also good to note that instance member variables and methods can be defined for them.

```
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8); //
       do not omit this when having enumeration members defined !
```

```
    int price; // by default the access modifier is also private if none
       is specified

    Apple(int p) { // this constructor is implicitly private, done to
       encapsulate the creation only in the Apple enum type itself
        price = p;
    }

    int getPrice() { // by default package-protected, just as regular
       class method members
        return price;
    }
}
```

`Note that if any members are defined for an enum, the list of enumerations has to be terminated with semicolon ; - otherwise it is compile time error, further more the member variables are by default PRIVATE, if no modifier is specified, while the member methods are implicitly package-private, the constructor is PRIVATE as the member variables`

Just like any other java class, in enum class types we can also define `static` and `final` members which work the same way as they do in any other regular class type in the language

Wrappers

The java standard language provides the basic primitives which are common in other languages as already disussed, those are `byte, short, int, long, float, double, boolean, char`, for each of them the language provides what are called Wrapper classes, which represent the same primitive however they are not actually primitives but actual fully formed java class types which are `Byte, Short, Integer, Long, Float, Double, Boolean and Character`.

These wrappers are often very useful because the language has a special feature, only valid for the primitive types where the wrapper class and primitive can be used intercangeably, that is called `auto-boxing`. This boxing is what the compiler does implicitly when it sees that a given variable or method accept an argument of a given type but another is passed - e.g. it accepts primitive but wrapper is passed, or the other way around

```
int method(Integer i) {
    return 0;
}

int method(Double d) {
    return 0;
}

method(5); // java will box the primitive into Integer
method(5.5d); // java will box the primitive into Double
```

Extraction

Each of the wrappers has a method which can be used to extract the unboxed raw primitive value inside the wrapped class. Those methods follow the naming pattern of `{primitiveType}Value()`

- Boolean.`booleanValue()`

- Character.`charValue()`
- Byte.`byteValue()`
- Short.`shortValue()`
- Integer.`integerValue()`
- Long.`longValue()`
- Float.`floatValue()`
- Double.`doubleValue()`

`Note that the integral wrappers - Byte, Short, Integer, Long, Float and Double inherit by default from Number class which exposes those methods, meaning that one can invoke Byte.doubleValue or Integer.floatValue, which will do exactly as expected return the actual double or float representation value of the boxed type`

```
Integer boxed = 850; // auto boxing the primitive value of 5 into new
   Integer(5);
boxed.byteValue(); // this will overflow the byte, and return 82
boxed.doubleValue(); // this will return the double representation of 850,
   which is 850.0d
boxed.floatValue(); // this will return the double representation of 850,
   which is 850.0f
```

Boxing

To expand on the auto boxing and unboxing of variables one needs to understand that this is done during runtime, and is not in any way efficient, since the runtime has to implicitly create objects of the wrapper type when a primitive is used but a wrapper expected, and vice versa, the compiler will use the methods mentioned above to obtain the raw value where wrapper is used, but primitive required. Therefore one should not rely too much on boxing and unboxing, and use raw primitives whenever possible, which is usually easy.

```
Integer k = 1;
Integer f = 2;
Integer r = k * f; // the runtime will first unbox into primitives,
   evaluate the expression `k*f` and then box back the result into the
   variable `r`
```

The auto boxing and unboxing can happen in expressions too, from the example above one can observe that a lot of `new` wrapper objects might be temporarily created to auto box/unbox which can lead to loss in performance

Auto promoting types is also something that works with auto boxing and unboxing, the premise is similar since auto promoting is already in the language, and as an example multiplying an integer and double - `Double r = new Double(k.doubleValue()* f.integerValue()); - this is how the byte code might look like - simplified`

```
Double k = 1.25d;
Integer f = 2;
Double r = k * f; // will be automatically promoted to double, since the
   expression will evaluate to double, and assign to the resulting
   variable just fine
```

```
Integer q = k * f; // this is a compile time error, since auto-promotion
   will upgrade the expression to double, can not down-cast double to
   integer automatically
```

```
Auto boxing and unboxing also works on Boolean wrappers, which is quite cool because
that means that one can use a Boolean wrapper inside an if statement however be careful
because auto unboxing is not safe, what if the Boolean wrapper instance was null, well
the auto unboxing will fail, there is no way to call booleanValue method on a null wrapper
object
```

```
Boolean b1 = null;
Boolean b2 = true;
assert(b1 && b2); // b1 will be auto-unboxed and fail with null pointer
   since the compiler simply tries to do b1.booleanValue() &&
   b2.booleanValue()
```

Annotations

Annotations in java language are a way for the program to define a data about itself, the annotations can be used in reflection to obtain information and specifics about the program and the underlying code implementation.

To declare an annotation the special `@interface` keyword is used, each annotation contains only methods, but no these methods have no bodies, the implementation itself is provided by java, in that regard the annotation is very much like a regular interface, hence the keyword used above.

```
@interface Annotation {
    int val();
}
```

Annotation is a super-interface of all annotations. It is declared within the java.lang.annotation package. It overrides hashCode(), equals(), and toString(), which are defined by Object. It also specifies annotationType(), which returns a Class object that represents the invoking annotation.

Retention

Each annotation can be specified what is called retention policy which is telling the compiler at which stage of the program's lifecycle this annotation is going to be kept. The retention itself is an annotation, provided by the java library, it is named `@Retention` and located inside `java.lang` package

- SOURCE - this is the annotation is only kept in source code, meaning it is discarded during byte code generation
- CLASS - the annotation information is retained after the byte code generation and stored in the `.class` file
- RUNTIME - the information is kept throughout the runtime of the program and can be obtained during runtime

```
@Retention(RetentionPolicy.RUNTIME)
@interface Annotation {
    int val();
}
```

Obtaining

To extract information about an annotation, one can use the built in java reflection mechanisms to do so, by fetching the annotation types defined on given identifiers - classes, methods etc.

```
public static class Meta {

    @Annotation(val = 1234)
    public void method() {
    }

    @Annotation(val = 1234)
    public void method(String arg1, int arg2) {
    }
}
public static void main(String args[]) {
    Meta ob = new Meta();
    try {
        Class<?> c = ob.getClass();
        Method m = c.getMethod("method");
        Method m2 =  c.getMethod("method", String.class, int.class);
        Annotation anno = m.getAnnotation(Annotation.class);
        Annotation anno2 = m2.getAnnotation(Annotation.class);
        System.out.println(" " + anno.val());
        System.out.println(" " + anno2.val());
    } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
    }
}
```

From the example above, the annotation information for two different methods is obtained, observe that the information about the method without any arguments and a method with arguments, as specified in the methods declaration. There is a way to obtain all annotations for a given target - by using getAnnotations(), which simply returns a list of Annotations instances of the Annotation type class

The Annotation API also provides a way to check if a give annotation exists instead of locating it one can simply obtain a boolean flag indicating if a given annotation is present on a given target - isAnnotationPresent(annotationClass) -

Types

- Annotation - this is the base class for all annotations, from which they extend
- AnnotatedElement - this is the base interface to annotate elements - Method, Package, Class, and so on.
- AnnotatedType - this type is reserved for annotations which are put on `args` or `return` types in general

Inheriting

By default all methods which obtain information about a given single or list of annotations, will try to obtain all annotations, including inherited ones, by default annotations are not inherited, but they can be annotated with a special annotation to specify that they are. How that works is simple, when one invokes the `getAnnotations` method the reflection implementation will start walking the entire class hierarchy, upwards, checking which annotations have the inherited annotations, adding them to the list of collected annotations, those that do not have one (by default) will simply not be present in the final result list. To obtain only the current level annotations without traversing the entire tree, one must use the `getDeclaredAnnotations` or in general the `*Declared*` variant of the methods related to obtaining annotation information, for a given target.

```
@Inherited
@interface Annotation {
    int val();
}
```

Target

The target annotation specified what target the given annotation is applicable for, by default when an annotation is created without a target, it is applicable everywhere, which is usually not what we want, some specific annotations must only be targeting specific language objects - methods, packages, classes and so on, below in the table are listed all types that an annotation can be specified for. Note the TYPE_USE - which is used to annotate both the return type and arguments of a given method, this is quite powerful, especially being able to target specific arguments on a method.

Target	Description
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
PACKAGE	Package
PARAMETER	Parameter
TYPE	Class, interface, or enumeration
TYPE_PARAMETER	Type parameter (Added by JDK 8.)
TYPE_USE	Type use (Added by JDK 8.)

```
@Target({ ElementType.FIELD, ElementType.LOCAL_VARIABLE })
@interface Annotation {
    int val();
}
```

Functional

The functional interface annotation specifies that an interface is functional - what that means is that an interface has only one single method (like `Runnable`) that needs to be implemented, this can be used by the compiler to then optimize and simplify the byte code generation, it also allows one to define a lambda function where a class or type is expected

```
@FunctionalInterface
interface Functional {
    void work();
}
class Concrete {
    void method(Functional functional) {
    }
}
Concrete c = new Concrete();
c.method(() => { /* method body implements the functional interface as
   anonymous class but represented as lambda function */ });
```

Note that by default all interfaces that have just one single abstract, non-implemented method are by default implicitly treated as FunctionalInterface type, meaning that one does not need to manually specify the annotation for this to be true

Type

As already mentioned with java 8, the new type annotation was added which allows annotations to be placed on method return and argument types, even in the `throws` declaration of a method in front of an exception type. One can also annotate the `this` implicit method parameter passed to object method invocations

```
class Concrete {
    void methodOne() throws @TypeAnno NullPointerException {
    }
    @TypeAnno Integer methodTwo(@TypeAnno Concrete this, int i, int j) {
       // note that we have exposed the implicit `this` argument of type
       Concrete
    }
}
```

In the example above one can see how not only the return type can be annotated, but the declaration in the `throws` block, as well as the implicit `this` which is passed on every instance member method invocation.

```
// An annotation that can be applied to a type function return type or
   argument.
@Target(ElementType.TYPE_USE)
@interface TypeUse { }

// An annotation that can be applied to a type parameter i.e generics
@Target(ElementType.TYPE_PARAMETER)
@interface TypeParam { }

// An annotation that can be applied to a field declaration - class member
   field.
@Target(ElementType.FIELD)
@interface FieldAnno { }

// An annotation that can be applied to a method declaration. - class
   member method
@Target(ElementType.METHOD)
@interface MethodAnno { }

// An annotation that can be applied to a type - class, enum, interface
   and so on
@Target(ElementType.TYPE)
@interface Type { }

// the TypeParam here targets the type parameter itself, as part of the
   class declaration
public @Type class SomeClass<@TypeParam T extends Integer> {

    // the FieldAnno here targets the declared member field, not the
       return type of the field
    public @FieldAnno Integer variable;

    public @MethodAnno @TypeUse Integer method(@TypeUse Integer input) {
        // one can see that MethodAnno, and TypeUse both target different
           things
        // - MethodAnno - targets the method declaration in this case
        // - TypeUse - targets the return type as well as the method
           argument and locals
        @TypeUse TypeAnnoDemo<Integer> target = new @TypeUse
           Integer(input.byteValue());
    }
}
```

An annotation can be put anywhere around the identifier, and only based on the target of the annotation can we actually deduce the exact target that is going to be annotated. Be careful, even if an annotation is put on another line above the identifier that does not mean it somehow changes the scope or target of the annotation

Repeating

Since the annotations strictly specify that by default only one instance of the given annotation can exist on a target, one can not simply put as many of the same annotation as one wants on the same target, to do this first the annotation has to be marked as repeatable, and a special container annotation has to be created for this repeatable annotation.

```
// the annotation that needs to be repeated
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}
// container which will hold the annotations, compiler handles that
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
    MyAnno[] value();
}
// declare multiple annotations of the same type on the same target
@MyAnno(str = "First annotation", val = -1)
@MyAnno(str = "Second annotation", val = 100)
public static void method(String str, int i)

// one can then obtain the container itself which will by default hold
   only the repeated annotations of the specified type
Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);

// this is also possible if we know the type of the repeated annotation as
   well.
Annotation[] annos = m.getAnnotationsByType(MyAnno.class);
```