Enum Types

• Built-in support for types of discrete values
• Advantages over C++’s enum:
  – Enum declaration defines a *class*
    • Type-safety*
    • Body can include methods and fields
  – Values may be objects
  – Support for iteration, naming

* C++11 introduced *enum class*, which are safer than the original Enum.
Representing a Set of Predefined Values –
The Old Way

```java
public class OldTrafficLight {
    public static final int RED = 1;
    public static final int YELLOW = 2;
    public static final int GREEN = 3;

    private int c = RED;
    public int getColor() {
        return c;
    }
    public void setColor(int newCol) {
        if (newCol >= RED && newCol <= GREEN)
            c = newCol;
    }
}
```
public class TrafficLight {
    public enum Color {
        RED, YELLOW, GREEN
    }

    private Color c = Color.RED;
    public Color getColor() {
        return c;
    }

    public void setColor(Color newCol) {
        c = newCol;
    }
}
The TrafficLight class - cont.

Using methods defined in the Enum class

```java
public class TrafficLight {
    ...
    public void setColor(int ordinal) {
        Color[] colors = Color.values();
        setColor(colors[ordinal]);
    }
    public boolean isCurColor(int ordinal) {
        return c.ordinal() == ordinal;
    }
    public void setColor(String colorName) {
        setColor(Color.valueOf(colorName));
    }
    public boolean isCurColor(String colorName) {
        return c.name().equals(colorName);
    }
}
```

Avoid declaring such methods
import static TrafficLight.Color.;

public void main() {
    TrafficLight tl = new TrafficLight();
    System.out.println(tl.getColor());
    tl.setColor(YELLOW); // ok thanks to static import
    System.out.println(tl.getColor()); // “YELLOW”
    System.out.println(tl.isCurColor("YELLOW")); // true
    System.out.println(tl.isCurColor(2)); // false
    tl.setColor(2);
    System.out.println(tl.getColor()); // “GREEN”
    tl.setColor("RED");
    System.out.println(tl.getColor()); // “RED”
}
public class TrafficLight {
    ...

document CODE

    public enum Color {
        RED, YELLOW, GREEN;

    public Color next() {
        switch (this) {
        case RED:
            return YELLOW;
        case YELLOW:
            return GREEN;
        default:
            return null;
        }
    }
}

}
public class TrafficLight {
    public enum Color {
        RED(22, "stop"),
        YELLOW(23),
        GREEN(24, "go");

        private int code;
        private string action;
        private Color(int c) { code = c; action = "careful" }
        private Color(int c, string s)
        {
            code = c;
            action = s;
        }
    }
}
An enum type with methods

```java
public class TrafficLight {
    public enum Color {
        RED(22) {
            @Override
            public Color next() { return YELLOW; }
        },
        YELLOW(23) {
            @Override
            public Color next() { return GREEN; }
        },
        GREEN(24);

        private int code;
        private Color(int c) { code = c; }
        public int getCode() { return code; }

        public Color next() { return null; }
    }

    private Color c = Color.RED;
    public int getColorCode() { return c.getCode(); }
    public void changeColor() { c = c.next(); }
}
```
Inheritance and Enums?

- Any enum type implicitly inherits from the `Enum` class.
- Therefore, due to no existence of multiple inheritance in Java, an enum type cannot inherit from any other class.
- However, implementation of interfaces is possible and sometimes can be useful. For example for adding behavior like `Comparable<T>` or `Iterable<T>`.
- Enum classes are also implicitly `final` and therefore cannot be extended.
  (for example, "`class X extends TrafficLight.Color {}`" won't compile)
Map of Enums

• EnumMap is a high performance map implementation for enums.
  – Implemented as an array.

```java
public class TrafficLight {
  ...
  EnumMap<Color, String> colorsMsgs =
      new EnumMap<Color, String>(Color.class);
  colorsMsgs.put(RED, “stop”);
  colorsMsgs.put(YELLOW, “get ready”);
  colorsMsgs.put(GREEN, “go”);
}
```
• EnumSet is a high performance set implementation for enums.
  – Implemented as a bit vector.
  – Replaces int-based “bit flags”.

```java
public class TrafficLight {
  ...
  EnumSet<Color> fullColorSet =
      EnumSet.<Color>.allOf(Color.class);
  EnumSet<Color> emptyColorSet =
      EnumSet.<Color>.noneOf(Color.class);
}
```
public class Singleton {
    private static Singleton instance;
    private static Boolean instantiated = false;

    public static Singleton getInstance() {
        if (!instantiated) {
            instantiated = true;
            instance = new Singleton();
        }
        return instance;
    }

    public void doSomething() {}

    private Singleton() { ... }
}
• “a single-element enum type is the best way to implement a singleton“ (Joshua Bloch, "Effective Java").
  – No drawbacks regarding serializable objects.
  – No Lazy initialization.

```java
public enum MySingleton {
    INSTANCE;

    public int x;
    public void doSomething(){...}
}
```
Enums - Summary

• Enums are classes
  – Extend java.lang.Enum
  – Might implement interfaces
• Enums have no public or protected constructor
  – removes the ability to create additional instances of the enum
    in addition to those defined at compile-time
• Enum values are public, static, and final
  – Values cannot be changed
  – The enum can’t be subclassed
• Enums override toString()
  – TrafficLight.Color.RED.toString() returns the String “RED”.
What’s in a source file?

- Lines of Code
  - For the Compiler
- Comments
  - For the Programmer
- Annotations
  - For the editor / IDE / Production & analyzing Tools.
The goal: Allow the programmer to provide additional information (Not-stringly form) about the program
  – This information can be used by software engineering tools

An annotation is a type
  – defined using an interface-like syntax

An annotation can be specified whenever a modifier is allowed
  – Convention: before the public/static modifiers

Annotations do not directly affect semantics of the class
  – But may affect semantics of things using the class (tools, code generation, runtime options, etc.)
Predefined Annotations

• @Override
  – Assert intention to override a method in a superclass
  – Compiler fails if not actually overriding
    • Checks spelling, override vs. overload

• @FunctionalInterface
  – Makes a single-method interface replaceable with a
    \[ \lambda \rightarrow Expr. \]

• @Deprecated
  – Indicates that an element should not be used

• @SuppressWarning
  – Tells the compiler to suppress specific warnings
package my.prg;

public class Rational extends Number {
    private long a, b;
    public Rational(long a, long b) {
        this.a = a;
        this.b = b;
    }

    private Double asDouble() {
        return new Double(a * 1.0 / b);
    }

    @Override
    public short shortValue() {
        return asDouble().shortValue();
    }
}
// Author.java:
public  @interface Author {
    String value() default "Unknown";
}

// MethodKind.java:
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
public  @interface MethodKind {
    boolean composite() default false;
    boolean mutator() default false;
}
public class Rational extends Number {
...

    @Author(value="Pazit")
    @MethodKind(composite=false, mutator=true)
    public void assign(long n) {
        a = n;
        b = 1;
    }

    @Override
    @Author("Pazit")
    @MethodKind(composite=true, mutator=false)
    public byte byteValue() {
        return asDouble().byteValue();
    }
}
Kinds of Annotations

• Marker annotations
  – Have no attributes
  – @Override, @Deprecated

• Single value annotations
  – Provide a single piece of data.
  – Attribute type can be primitive, string, Enum, or array of the previous.
    • @Author(“Pazit”)
    • @SuppressWarnings({“unchecked”, “deprecation”})

• Multi valued annotations
  – @MethodKind(composite=true, mutator=false)
What Can Be Annotated?

• Any program element
  – Package
  – Types
    • Class, Interface, Enum definition, Annotation type
  – Method, Constructor, Field, Enum constant, Method parameter
  – Local variable declaration
Meta-Annotations

- Annotations that annotate annotations
- Specify how the annotation should be used
  - @Documented
    - Javadoc should be generated when this annotation is applied to an element
  - @Inherited
    - Does the annotation get applied to subclasses
  - @Target
    - Where the annotation can be used (source elements)
    - Default is all
  - @Retention
    - Where is the annotation retained (where it “lives”)
    - Possible values: runtime, class, source
The scope in which annotations are available is determined by the `@Retention` meta-annotation:

- `RetentionPolicy.SOURCE` – Compile time only, e.g. `@Override`, `@SuppressWarnings`
- `RetentionPolicy.CLASS` – Class load time.
- `RetentionPolicy.RUNTIME` – Run time. The most commonly used type. e.g. `@Test` (JUnit)

Surprisingly, the default...
@Target({TYPE, CONSTRUCTOR, FIELD})
@Retention(RetentionPolicy.RUNTIME)
@interface Marker {
}

@Marker
class FooClass {
    //OK
    @Marker public FooClass(){} //OK
    @Marker int x; //OK
    @Marker public void m() { //Error
        @Marker int y; //Error (@Target(LOCAL_VARIABLE) needed)
    }
}

Reflection and Annotations

• Annotations marked with `@Retention(RUNTIME)` are available via reflection.

• *Class, Constructor, Field, Method, Package* have methods to handle annotations:
  – `isAnnotationPresent`
  – `getAnnotations`
  – `getDeclaredAnnotations`
  – `getAnnotation((annotationClass)`
Inspecting Annotations using Reflection

public class AuthorPrinter {

    static void printMethodsAuthor(Class c) {

        Arrays.stream(c.getMethods).filter(m ->
            m.isAnnotationPresent(Author.class))
            .map(m -> String.format("%s by %s",
                             m.getName(),
                             m.getAnnotation(Author.class)))
            .forEach(System.out::println);
    }

• Author annotation will be accessible only if its declaration was annotated with @Retention(RUNTIME)