Use enums instead of int constants

enumerate type이란?

- 일정 개수의 상수 값을 정의한 다음, 그 외의 값은 허용하지 않는 타입.
- An *enumerated type* is a type whose legal values consist of a fixed set of constants

기존 상수 취급 문제

1. 정수 열거 패턴(int enum pattern)

```
// The int enum pattern - severely deficient!
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH = 2;
public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

문제점

1. 타입 안전 보장 불가
    - APPLE넣을 자리에 ORANGE를 넣어도 어떤 경고를 발생하지 않습니다.
2. 좋지 않은 표현력
    - 네임스페이스를 지원하지 않기 때문에 접두어를 써서 구분지어야 합니다.
3. 프로그램이 깨지기 쉬움
    - 컴파일시 참조 값이 아니라 상수 값이 그대로 쓰여서 클라이언트에서도 반드시 컴파일해야 합니다.
4. 문자열로 출력하기 까다로움
    - 변수가 단순히 상수를 담는 형태이지요. 그렇기 때문에 디버거에서 출력할 때도 그리 도움되지 않는 메세지만 출력됩니다.

2. 문자열 열거 패턴(string enum pattern)

정수 열거 패턴의 변형으로서 정수 대신 문자를 담았다 보시면 됩니다. 정수 열거 패턴보다 더 좋지 못합니다.

상수의 의미는 출력할 수 있지만, 문자열 값 그대로 하드코딩하게 됩니다.

오타가 있어도 컴파일러가 확인할 길이 없어 런타임 버그가 생기고, 문자열 비교시 성능 저하 발생하죠

Enums Pattern 단점 해소 - enums

자바에서는 위 모든 문제를 해결해주고 장점도 주는 enum type을 만들었습니다.

[enums type의 기본 형태]

```
public enum Apple  { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

자바는 완전한 형태의 클래스라서 다른 언어의 열거타입보다 훨씬 강력합니다.

Idea

1. enum types는 class입니다.
2. 각 public static final field로 enumeration constant당 하나의 인스턴스를 나타냅니다.

   이로 인해 constructor 접근을 막습니다.

singleton

2번의 이유로, enum은 instance를 control한다고 표현할 수 있습니다. single ton과 똑같지요?

그러니 singleton을 일반화했다고 말할 수 있고, singleton은 Enum의 원소가 1개만 있는 타입이라고 말할 수도 있습니다.

1. compile time type safety

parameter를 Apple type을 선언한다면, Apple values가 올 것을 보장할 수 있습니다.

다른 타입을 전달하거나, 다른 enum type에 == operator를 쓰려고 한다면 compile time error가 발생합니다.

2. name sapce

동일한 constant name을 가지고 있어도 문제 없습니다. type이 그 constant의 name space가 되니까요.

3. recompile

그리고 Constant를 정렬하거나 새로 추가하여 재컴파일해도 client코드에선 문제가 없습니다. enum type이 중간 레이어가 되어 줍니다.

4. toString

상수 이름도 그대로 출력할 수 있습니다. toString method를 제대로 구현해주었거든요.

추가 기능 - Method와 Field

임의의 메서드나 필드를 추가할 수 있고, 임의의 인터페이스 또한 구현할 수 있도록 하였습니다.

Object 메서드 및 Comparable, Serializable을 구현해놨습니다.

그런데 메서드와 필드는 어떻게 사용하라고 만들어놓은 걸까요?

각 상수와 연관된 데이터를 해당 상수 자체에 내재하고 싶은 경우를 생각해보세여

연관된 값을 저장할 수 있는 필드를 선언하고 꺼낼 수 있는 메서드를 선언하는 방식이죠

예를 들어서 Apple, Orange가 있습니당

Apple과 Orange에 색깔을 상수로 표현해줄 수 있고, 이미지를 method 통해서 출력해줄 수도 있겠죠?

Enum의 특징을 잘 보여줄 수 있는 행성 예제를 알아보겠습니다.

Planet 예제

```
// Enum type with data and behavior
public enum Planet {
  MERCURY(3.302e+23, 2.439e6),
  VENUS  (4.869e+24, 6.052e6),
  EARTH  (5.975e+24, 6.378e6),
  MARS   (6.419e+23, 3.393e6),
  JUPITER(1.899e+27, 7.149e7),
  SATURN (5.685e+26, 6.027e7),
  URANUS (8.683e+25, 2.556e7),
  NEPTUNE(1.024e+26, 2.477e7);
  private final double mass;  // In kilograms
  private final double radius;  // In meters
  private final double surfaceGravity; // In m / s^2



  // Universal gravitational constant in m^3 / kg s^2
  private static final double G = 6.67300E-11;
  // Constructor
```

```
  Planet(double mass, double radius) {
    this.mass = mass;
    this.radius = radius;
    surfaceGravity = G * mass / (radius * radius);
  }
  public double mass()           { return mass; }
  public double radius()         { return radius; }
  public double surfaceGravity() { return surfaceGravity; }
  public double surfaceWeight(double mass) {
    return mass * surfaceGravity;  // F = ma
  } }
```

1. 열거 타입 상수와 데이터 연결

   생성자에서 데이터를 받아 인스턴스 필드에 저장하면 됩니다.
2. immutable 속성때문에 field는 반드시 final이어야 합니다.
3. public보다는 private 식별자를 쓰고 method로 값을 전달하세요.
4. 한편으로, surfaceGravity를 생성시 저장하는 이유는 최적화를 위해서입니다.

Enum type의 상수를 순회하며 무게 산출하기

```
public class WeightTable {
  public static void main(String[] args) {
    double earthWeight = Double.parseDouble(args[0]);
    double mass = earthWeight / Planet.EARTH.surfaceGravity();
    for (Planet p : Planet.values())
      System.out.printf("Weight on %s is %f%n",
                        p, p.surfaceWeight(mass));
  }
}
```

살펴볼 것

1. 정적 메서드인 values 제공

enum의 static method로, 선언된 순서대로 선언된 상수의 배열을 반환해줍니다.

2. toString

선언된 상수의 이름대로 toString메서드를 구현해서 printf 출력하기 좋습니다. 만약에 다른 이름으로 변경하고 싶다면 toString을 오버라이딩 해주면 되죠.

output

```
Weight on MERCURY is 69.912739
Weight on VENUS is 167.434436
Weight on EARTH is 185.000000
Weight on MARS is 70.226739
Weight on JUPITER is 467.990696
Weight on SATURN is 197.120111
Weight on URANUS is 167.398264
Weight on NEPTUNE is 210.208751
```

3. 열거타입 삭제시

명왕성 퇴출 되었잖아요? 그래서 삭제해도 클라이언트 재컴파일 필요없습니다. 만약에 삭제된 상수를 참조했다면 의미있는 message를 던져주죠.

4. scope

선언한 클래스 혹은 그패키지에서만 유효할 경우 - private or package private 로 선언하시고

널리 쓰이는 경우 - top level class로 선언하면 됩니다.

상수마다 동작을 다르게 할 경우

planet예제는 모든 method가 동일한 동작을 하였습니다. 그런데 각 상수마다 다른 동작을 하게 만들고 싶을 때도 있죠. 이때는 어떻게 해야할까요?

Operation 예제

1. bad-case

```
// Enum type that switches on its own value - questionable
public enum Operation {
  PLUS, MINUS, TIMES, DIVIDE;
  // Do the arithmetic operation represented by this constant
  public double apply(double x, double y) {
    switch(this) {
      case PLUS:   return x + y;
      case MINUS:  return x - y;
      case TIMES:  return x * y;
```

```
      case DIVIDE: return x / y;
    }
    throw new AssertionError("Unknown op: " + this);
  }
}
```

문제점

1. 가독성이 떨어집니다
2. throw는 기술적으로 도달할 수 있어 생략하면 컴파일조차 되지 않습니다.
3. OCP를 위배합니다.

2. good-case

열거 타입에 apply abstract method를 선언하여 이를 각 클래스 body에서 적절히 구현하도록 합니다. 상수에 선언된 메서드를 상수별 메서드(constant-specific method)라 부르죠.

```
// Enum type with constant-specific method implementations
public enum Operation {
  PLUS {public double apply(doublex,doubley){return x+y;}},
  MINUS {public double apply(double x, double y){return x - y;}},
  TIMES {public double apply(double x, double y){return x * y;}},
  DIVIDE{public double apply(double x, double y){return x / y;}};

  public abstract double apply(double x, double y);
}
```

apply method 구현 깜박하기 어렵고, 추상메서드로 선언되어 있어 반드시 구현하도록 하죠.

상수별 메서드를 상수별 데이터와 결합 예제

```
// Enum type with constant-specific class bodies and data
public enum Operation {
  PLUS("+") {public double apply(double x, double y) { return x + y; } },
  MINUS("-") {public double apply(double x, double y) { return x - y; } },
  TIMES("*") {public double apply(double x, double y) { return x * y; } },
  DIVIDE("/") {public double apply(double x, double y) { return x / y; } };

  private final String symbol;
  Operation(String symbol) { this.symbol = symbol; }

  @Override public String toString() { return symbol; }
  public abstract double apply(double x, double y);
}
```

```
public static void main(String[] args) {
  double x = Double.parseDouble(args[0]);
  double y = Double.parseDouble(args[1]);
  for (Operation op : Operation.values())
    System.out.printf("%f %s %f = %f%n",
                      x, op, y, op.apply(x, y));
}
```

이처럼 작성하면 상수의 데이터와 상수별 메서드를 결합하여 표현력을 크게 높힐 수 있습니다!

fromString

constant's의 이름을 constant로 변경시켜주는 valueOf(String)를 지원해줍니다.

만약에 출력될 이름 변경하기 위해 toString을 overriding했다면 fromString method도 고려하시길 바랍니다.

```
// Implementing a fromString method on an enum type
private static final Map<String, Operation> stringToEnum =
  Stream.of(values()).collect(
  toMap(Object::toString, e -> e));

// Returns Operation for string, if any
public static Optional<Operation> fromString(String symbol) {
  return Optional.ofNullable(stringToEnum.get(symbol));
}
```

1. static이 초기화되는 시점에 stringToEnum이 생성됩니다.
2. constructor에 map을 넣지 않았습니다. 만약 가능했다면 생성시점에 method가 없으므로 NullPointException이 Runtime중에 발생했을 겁니다.
3. Optional<String> valid operation을 하지 않으면 string에 맞는 연산이 없음을 알려서 클라이언트가 대처하도록 합니다.

Strategy enum pattern

constant-specific method의 한계점

enum constants간 code를 공유하기 어렵습니다.

Payroll 예제

급여명세서에서 작성할 요일을 표현하는 열거 타입을 예로 들겠습니다.

1. 직원의 시간당 기본 임금과 그날 분단위로 일한 시간이 주어지면 일당을 계산하는 메서드를 가지고 있습니다.
2. 주중에 초과근로에 대한 잔업수당을 주고 주말에는 반드시 잔업수당을 주도록합니다.

만약 case문을 작성하면 간단히 작성할 수 있죠

```
// Enum that switches on its value to share code - questionable
enum PayrollDay {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
  SATURDAY, SUNDAY;

  private static final int MINS_PER_SHIFT = 8 * 60;

  int pay(int minutesWorked, int payRate) {
    int basePay = minutesWorked * payRate;

    int overtimePay;
    switch(this) {
      case SATURDAY: case SUNDAY: // Weekend
        overtimePay = basePay / 2;
        break;
      default: // Weekday
        overtimePay = minutesWorked <= MINS_PER_SHIFT ?
          0 : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
    }

    return basePay + overtimePay;
  }
}
```

간결하나 OCP관점에서 문제가 있죠.

constant-specific method로 구현해도 문제가 있습니다. 구현은 다음처럼 할 수 있겠죠

Constant-specific method 구현

구현방법1

모든 상수에 중복하여 pay method 구현

구현방법2

helper method를 주말과 평일로 두어 각 constant별로 적절한 helper method 호출

두 방법 모두 코드가 장황해지고 error가 발생하기 쉬워집니다.

그렇다면 어떻게 해야할까요? 바로! 요일이 전략을 선택하도록 합니다.

Payroll 예제

```
// The strategy enum pattern
enum PayrollDay {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
  SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

  private final PayType payType;

  PayrollDay(PayType payType) { this.payType = payType; }
  PayrollDay() { this(PayType.WEEKDAY); }  // Default

  int pay(int minutesWorked, int payRate) {
    return payType.pay(minutesWorked, payRate);
  }

  // The strategy enum type
  private enum PayType {
    WEEKDAY {
      int overtimePay(int minsWorked, int payRate) {
        return minsWorked <= MINS_PER_SHIFT ? 0 :
          (minsWorked - MINS_PER_SHIFT) * payRate / 2;
        }
    },
    WEEKEND {
      int overtimePay(int minsWorked, int payRate) {
        return minsWorked * payRate / 2;
      }
    };

    abstract int overtimePay(int mins, int payRate);
```

```
    private static final int MINS_PER_SHIFT = 8 * 60;

    int pay(int minsWorked, int payRate) {
      int basePay = minsWorked * payRate;
      return basePay + overtimePay(minsWorked, payRate);
    }
  }
}
```

swtich statements 좋은 예

앞서보듯, 보통 switch는 constant-specific method와 어울리지 않습니다. 그렇다면 언제 좋을까요?

enum types를 constant-specific method와 매칭할 때 좋습니다.

inverse 예제

서드파티에서 가져온 Operation 열거 타입의 각 연산의 반대 연산을 반환하는 메서드를 고려합시다.

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
  switch(op) {
    case PLUS:   return Operation.MINUS;
    case MINUS:  return Operation.PLUS;
    case TIMES:  return Operation.DIVIDE;
    case DIVIDE: return Operation.TIMES;

    default: thrownewAssertionError("Unknownop:"+op); }
}
```

해당 method가 enum type에 속할 것이 아니라면 이 방식을 적용하는게 좋습니다.

한편, 열거타입 성능은 정수타입과 별반차이가 없죠.

열거 타입 사용할 때

결국 열거타입은

필요한 원소를 컴파일 타임에 다 알 수 있는 상수 집합이라면 항상 열거타입을 쓰는게 좋습니다.

또한, 상수를 나중에 추가하더라도 바이너리 호환이 되므로 문제가 없습니다!