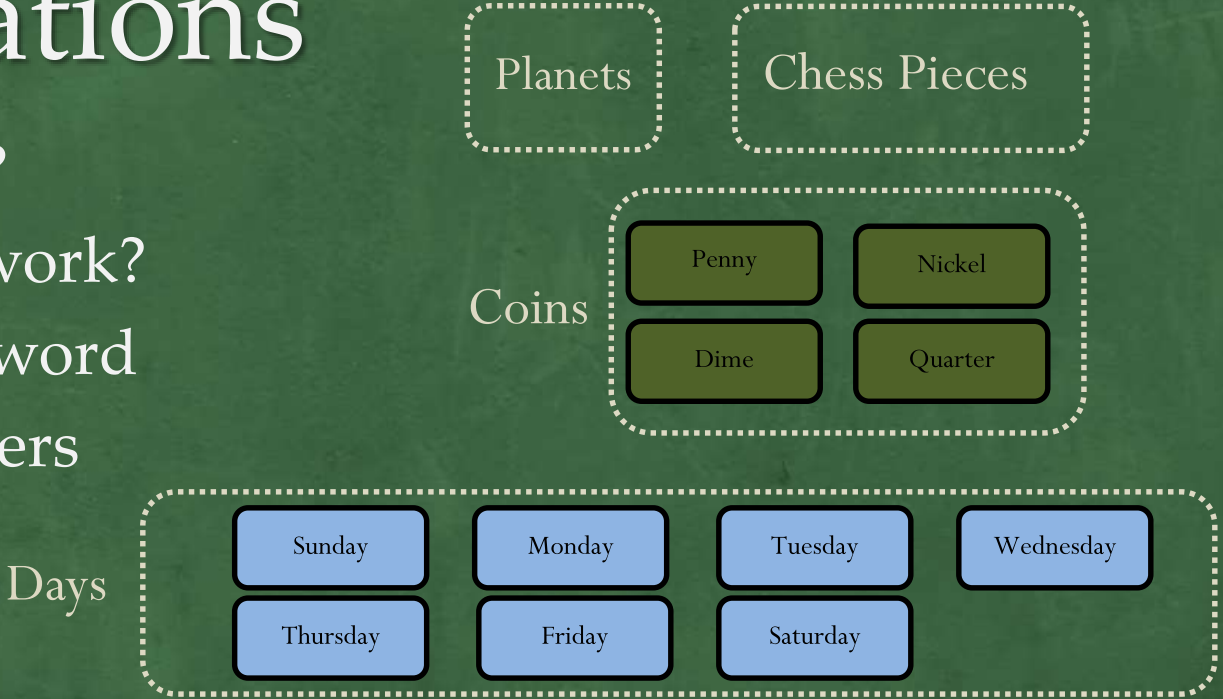
Enumerations

- What are they?
- How do they work?
- The enum keyword
- Adding members



Introduction to Java

See Also

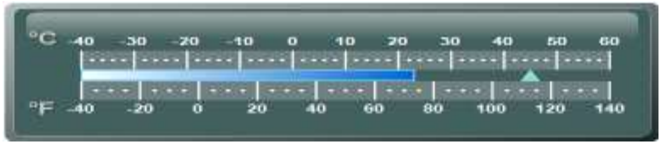
https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html

https://www.youtube.com/watch?v=A0GHaVRlYAQ

Ranges and Sets

- "Temperature" is a continuous range of values


°C -40 -30 -20 -10 0 10 20 30 40 50 60
°F -40 -20 0 20 40 60 80 100 120 140

- "Day of Week" is a limited set of values you can list (enumerate)

- You can map discrete values to numbers:

```
class Boolean {
    public static final int FALSE = 0;
    public static final int TRUE = 1;
}

class Square {
    public static final int EMPTY = 0;
    public static final int X = 1;
    public static final int O = 2;
}
```

```
int hasBeenRead = Boolean.FALSE; // 0=false, 1=true
int squareOne = Square.X;        // 0=empty, 1=X, 2=O

squareOne = -100;                // Oops! Hey, compiler?

hasBeenRead = Square.O * 5;      // Oops! Hey, compiler?
```

Roll Your Own

- Use class instances instead of int

```
public class Square {

    public static Square EMPTY = new Square();
    public static Square X = new Square();
    public static Square O = new Square();

}

 Square squareOne = Square.X;
 Square squareTwo = Square.EMPTY;
```

- Improvements to the “pattern”:
  - Private constructor
  - Use “final” on class and instances
  - Pass “code name” and “code ordinal” to instances
  - Add utility functions for common needs

The "enum"

- The compiler writes the "class" for you
- You write this:
- The compiler generates this:

```
public enum Square {
    EMPTY, X, O
}
```

```
public final class Square extends Enum<Square> {

    public static final Square EMPTY = new Square("EMPTY",0);
    public static final Square X     = new Square("X",1);
    public static final Square O     = new Square("O",2);

    public static Square[] values() {...}
    public static Square valueOf(String name) {...}

    private Square(String name, int ordinal) {
        super(name,ordinal);
    }

}
```

```
// Available from Enum<Square>:
//    public String name()
//    public int ordinal()
```

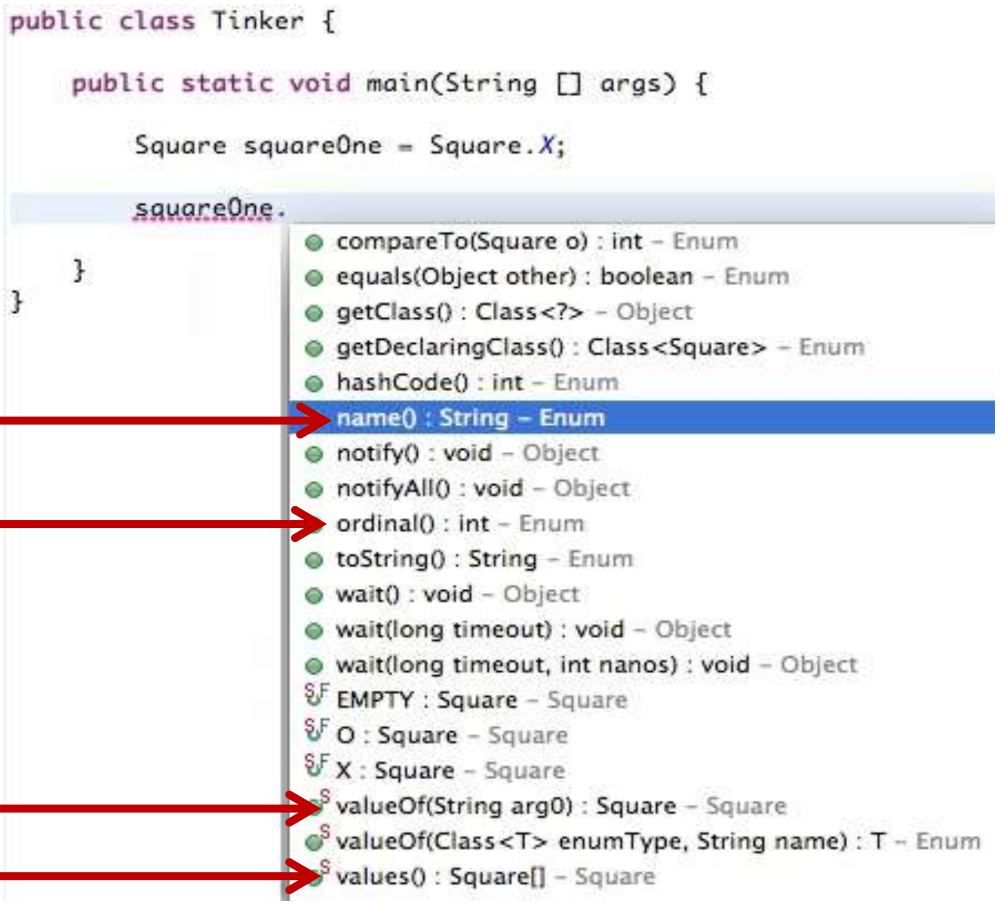
Using the "enum"

- The "using" code is the same as before:

```
public static void main(String [] args) {

    Square squareOne = Square.EMPTY;
    Square squareTwo = Square.X;

    if(squareOne == squareTwo) {
        System.out.println("SAME");
    }
}
```

Using the "enum"

- The "using" code is the same as before.
- The helper functions are written for you:



```
public class Tinker {

    public static void main(String [] args) {

        Square squareOne = Square.X;

        squareOne.

    }
}
```

```
compareTo(Square o) : int - Enum
equals(Object other) : boolean - Enum
getClass() : Class<?> - Object
getDeclaringClass() : Class<Square> - Enum
hashCode() : int - Enum
name() : String - Enum
notify() : void - Object
notifyAll() : void - Object
ordinal() : int - Enum
toString() : String - Enum
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
EMPTY : Square - Square
O : Square - Square
X : Square - Square
valueOf(String arg0) : Square - Square
valueOf(Class<T> enumType, String name) : T - Enum
values() : Square[] - Square
```

Using the "enum"

- Useful helper functions:

```
Square [] sqs = Square.values(); // Array of 3 instances

Square squareOne = Square.valueOf("EMPTY");

String s = squareOne.name(); // "EMPTY"

squareOne = Square.O;
int i = squareOne.ordinal(); // 2
```

Switches

- The "ordinal" allows you to make a switch

```
squareOne = Square.O;

switch(squareOne.ordinal()) {

    case 0:
        System.out.println("Case empty");
        break;
    case 1:
        System.out.println("Case X");
        break;
    case 2:
        System.out.println("Case O");
        break;
}
```

Switches

- The “ordinal” allows you to make a switch
- The compiler calls ordinal for you

```
squareOne = Square.O;

switch(squareOne) {

    case EMPTY:
        System.out.println("Case empty");
        break;
    case X:
        System.out.println("Case X");
        break;
    case O:
        System.out.println("Case O");
        break;
}
```

Extending the "enum"

- You can add to the generated code:

```
public enum Coin {
    PENNY, NICKEL, DIME, QUARTER;
    int cents;
}
```

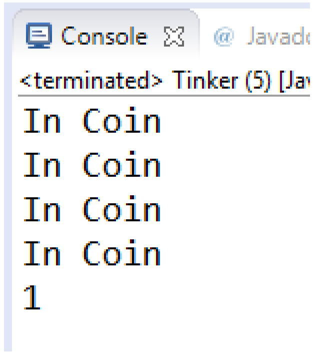
- The instance names must be first
- If you add anything you must add the semicolon

Extending the "enum"

- You can add to the generated code:

```
public enum Coin {

    PENNY(), NICKEL(), DIME, QUARTER;

    int cents;

    private Coin() {
        cents = 1;
        System.out.println("In Coin");
    }

}
```

```
public static void main(String[] args) {

    Coin c = Coin.PENNY;

    System.out.println(c.cents);

}
```



```
<terminated> Tinker (5) [Ja
In Coin
In Coin
In Coin
In Coin
1
```

- The instance names must be first
- You can add your own constructor

The "Coin" Example

- Good encapsulation:

```
public enum Coin {

    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private int cents;

    private Coin(int c) {
        cents = c;
    }

    public int getCents() {
        return cents;
    }

}


Coin c = Coin.QUARTER;
System.out.println(c.getCents()); // "25"
```

Tinkering

- The Java tutorial creates the “Planet” enum.
- https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
- Before you look at the tutorial, create your own code.
  - Add attributes for “mass” and “radius”. What numeric type should these be? Look up the values for each planet.
  - Add a method to calculate the surface weight on a planet for the given mass “m”:
    - Weight = m * 6.67300E-11 * mass / (radius * radius)
  - The tutorial puts “main” inside the enum. Do you like that? Or do you want it in a separate “Tinker” class? Why or why not?