Enumerations

Enumerations (1)

An **enumeration** (keyword **enum**) is a type consisting of named constants. For example:

```
public enum Day {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

Enumerations (2)

Since an **enumeration** is a type, you can use it as you would any other type:

```
Day today = Day.MONDAY;
today = Day.TUESDAY;
Day tomorrow = Day.WEDNESDAY;
```

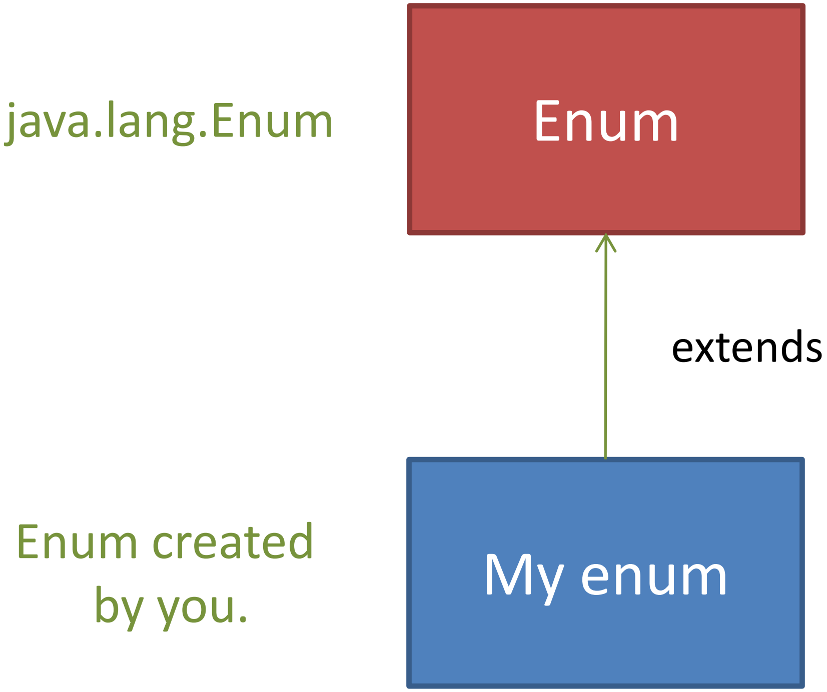
Enumerations (3)

The value of the enum can be used in the same way as other variables, such as using them to make decisions regarding program flow:

```
switch (day) {
    case MONDAY: {
        System.out.println("I hate Mondays!");
        break;
    }

    case FRIDAY: {
        System.out.println("I love Fridays!");
        break;
    }

    case SUNDAY: {
        System.out.println("Back to work tomorrow.");
        break;
    }

    default: {
        System.out.println("Just another day of the week");
    }
}
```

Enumerations (4)

Enumerations are extensions of the Enum class in Java.



java.lang.Enum

Enum

extends

Enum created by you.

My enum

Enumerations (5)

Enums are reference types, so you can define methods that take enums as parameters or that return enums.

Enums can be compared using ”==”

Enums in Java are much more powerful than in many other languages.

Enumerations (6)

Enums can be given values, but doing this requires that we create a constructor and a field.

```java
public enum Day {
    MONDAY ("I hate Mondays!"),
    TUESDAY ("Just another day."),
    WEDNESDAY ("Middle of the week."),
    THURSDAY ("Just another day."),
    FRIDAY ("Weekend tomorrow!"),
    SATURDAY ("Time to relax!"),
    SUNDAY ("Back to work tomorrow.");

    // The variable represents the value of the enum
    private final String opinion;

    // The Constructor MUST be private.
    private Day(String opinion) {
        this.opinion = opinion;
    }
}
```

Enumerations (7)

You can also override toString().

```java
public enum Day {
    MONDAY ("I hate Mondays!"),
    TUESDAY ("Just another day."),
    WEDNESDAY ("Middle of the week."),
    THURSDAY ("Just another day."),
    FRIDAY ("Weekend tomorrow!"),
    SATURDAY ("Time to relax!"),
    SUNDAY ("Back to work tomorrow.");

    // Code omitted

    @Override
    public String toString() {
        return this.opinion;
    }
}
```

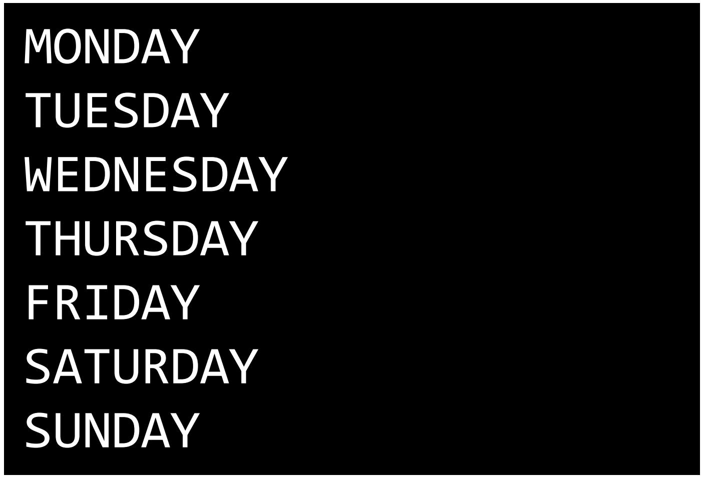
Enumerations (8)

You can iterate over enums using the **values()** method:

```java
for (Day day : Day.values()) {
    System.out.println(day);
}
```

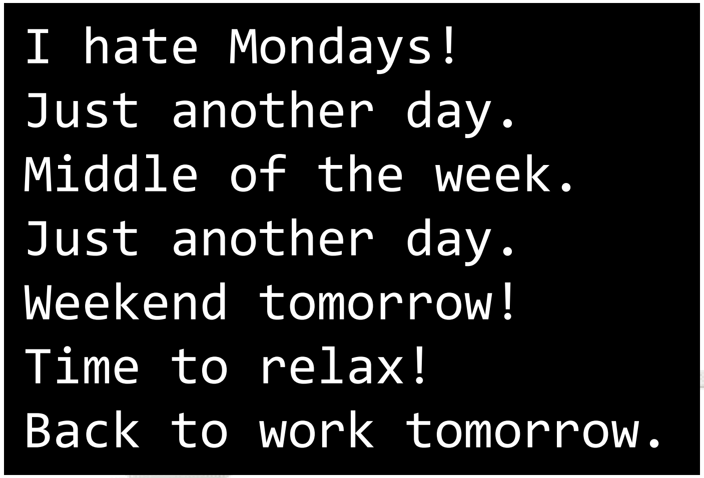
Output:

Without overriding toString():



```
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
SUNDAY
```

With overriding toString():



```
I hate Mondays!
Just another day.
Middle of the week.
Just another day.
Weekend tomorrow!
Time to relax!
Back to work tomorrow.
```

Enumerations (9)

You can add methods to enums:

```java
public enum Day {
    MONDAY ("I hate Mondays!"),
    TUESDAY ("Just another day."),
    WEDNESDAY ("Middle of the week."),
    THURSDAY ("Just another day."),
    FRIDAY ("Weekend tomorrow!"),
    SATURDAY ("Time to relax!"),
    SUNDAY ("Back to work tomorrow.");

    // Code omitted

    public String getLowerCase() {
        return this.name().toLowerCase();
    }
}
```

Enumerations (10)

Interesting methods available for enums:

Method	Description	Example
ordinal()	Return ordinal number for enum.	Day.MONDAY has ordinal 0 (zero), Day.TUESDAY has 1, etc.
compareTo(o)	Compares enum to another of same type.	-1 if ordinal is smaller, 0 if equal, 1 if bigger.
name()	Returns name of enum constant	Day.MONDAY returns MONDAY Day.TUESDAY returns TUESDAY
values()	Static method returning a list of enum constants	{ MONDAY, TUESDAY, WEDNESDAY, ..., SUNDAY }

Why use enumerations?

Enums can help reduce coupling and adds readability to your code. Consider the following code, which represents a **process**. Each process can have a priority between 1 and 3.

```java
public class Process {
    private int priority;

    public void StartProcess() {
        // Code to start the process
    }

    public void PauseProcess() {
        // Code to pause the process
    }

    public int getPriority() {
        return priority;
    }

    public void setPriority(int priority) {
        this.priority = priority;
    }
}
```

Why use enumerations?

We have to make sure that the provided priority is a valid value, i.e. between 1 and 3.

```java
public void setPriority(int priority) {
    if (priority < 1 || priority > 3) {
        throw new IllegalArgumentException("priority out of range.");
    } else {
        this.priority = priority;
    }
}
```

Readability

We would then go on and use the class in the following way:

```
Process aProcess = new Process();

aProcess.setPriority(1);
aProcess.StartProcess();
```

Notice that it’s not very clear from the context what a priority of 1 means.
Is 1 high or low priority?

Code coupling

Now, if later we would have to add one more priority level, how much work would be involved?

1. We would have to change the **Process** class to accept 4 as well as 1-3.
2. We might have to update the priority usage on every single **Process** instance in our code (will 2 still mean 2?)

Code coupling

Our code would probably be cluttered with statements such as:

```
for (Process proc : processList) {
    if (proc.getPriority() == 3) {
        // ...
    } else {
        // ...
    }
}
```

If we change the meaning of the priority levels, every single place where we have added such a number will have to be changed as well

Introducing an enum (1)

We’ll try to loosen the **coupling** and add some **readability** by using an **enum** instead:

```
public enum Priority {
    LOW,
    MEDIUM,
    HIGH
}
```

Introducing an enum (2)

In the **Process** class, we'll just change the **Setters/Getters** and the **priority field**:

```
public class Process {
    private Priority priority;

    // Code omitted

    public Priority getPriority() {
        return priority;
    }

    public void setPriority(Priority priority) {
        this.priority = priority;
    }
}
```

Improving readability

The intent of the priority is now clearer. We don’t have a mysterious integer anymore:

```
Process aProcess = new Process();

aProcess.setPriority(Priority.LOW);
aProcess.StartProcess();
```

Lowered coupling (1)

Neither will our code be full of mysterious **int**-usages which need to be updated:

```
for (Process proc : processList) {
    if (proc.getPriority() == Priority.LOW) {
        // ...
    } else {
        // ...
    }
}
```

Lowered coupling (2)

We could even add another priority level, without having to change anything:

```
public enum Priority {
    LOW,
    BELOWMEDIUM,
    MEDIUM,
    HIGH
}
```

Lowered coupling (3)

We could add as many as we want, in fact.

```
public enum Priority {
    LOWEST,
    LOW,
    BELOWMEDIUM,
    MEDIUM,
    ABOVEMEDIUM,
    HIGH,
    HIGHEST
}
```

We get a semantic value from doing this, as well as a code base which is easier to maintain.

Exercise 24.x

Let's do exercises 24.x