Instance and static methods

Method calls

Every method belongs to some object, unless specified otherwise. To emphasize this, we often call such a method an *instance* method. When we call an instance method, we must specify both the object and method:

```
object.method(<parameters>)
```

Internal method calls

To invoke a method within the same object, we can just call by the method name.

```
public class Dog {

  public void bark(String b) {
    ...
  }

  public void run() {
    bark("Woof!");
    ...
  }

}
```

Internal method calls

Or, to be more explicit, we may also put "`this`" with the method name.

```
public class Dog {

  public void bark(String b) {
    ...
  }

  public void run() {
    this.bark("Woof!");
    ...
  }

}
```

External method calls

To invoke a method of another object, we must always explicitly specify the object.

In general, from a class `Doo`, to call a `Coo` object `c`'s method `noo()`, we put:

```
public class Doo {

  public void poo() {
    Coo c = new Coo();
    c.noo(7);
  }

}
```

The `main` method

Some class has a special method, called `main`, that defines a stand-alone program. A class that has the `main` method is called a *Java application*.

```
public class Coo {

  public static void main(String[] args) {
    ...
  }

}
```

The `main` method

To execute such a stand-alone program defined in the `main` method, just invoke `java` with the name of its class:

```
$ java Coo
```

`static` method

In general, a method that does not belong to any object is called a `static` method.

To call such a method in another class, we must specify the class and method:

```
Class.method(<parameters>)
```

`static` method

```
public class Coo {

  public static void print() {
    ...
  }

}
```

To call the class `Coo`'s `static` method `print()` from outside, we put:

```
Coo.print();
```

Objects inside objects

Number objects

We wish to represent numbers in a certain range; for example, 0 through 59, 0 through 23, etc.

- We wish to support operations such as $59 + 1 = 0$ (instead of $59 + 1 = 60$).
- We wish to format single digit values with leading 0; for example, 01 (instead of 1).
- Such representations will be particularly useful in designing digital clocks.

Class specification

Class name: `Number`.

To represent two-digit numbers in the range 0 through $n - 1$.

Attributes:

- `limit : int` — the upper limit n of the range.
- `value : int` — the value of a number.

Behaviors:

- `increment()` — Increment `value` by 1, rolling over to 0 if necessary.
- `format()` — Return `value` with leading 0.

Class specification

Number
– limit : int – value : int
+ Number(l : int, v : int) + getValue() : int + increment() : void + format() : String

```
public class Number {

  private int limit;
  private int value;

  public Number(int l, int v) {
    limit = l;
    value = v;
  }

  public int getValue() { return value; }

  public void increment() {
    value = (value + 1) % limit;
  }

  public String format() {
    if (value < 10) return "0" + value;
    else return "" + value;
  }

}
```

Digital clock

Using this class `Number`, how can we implement a digital clock?

- For simplicity, we will consider a 24-hour clock (from 00:00 to 23:59).
- One `Number` object to represent minutes (00 to 59).
- Another `Number` object to represent hours (00 to 23).

Class specification

Class name: `DigitalClock`.

To represent a 24-hour clock (from 00:00 to 23:59).

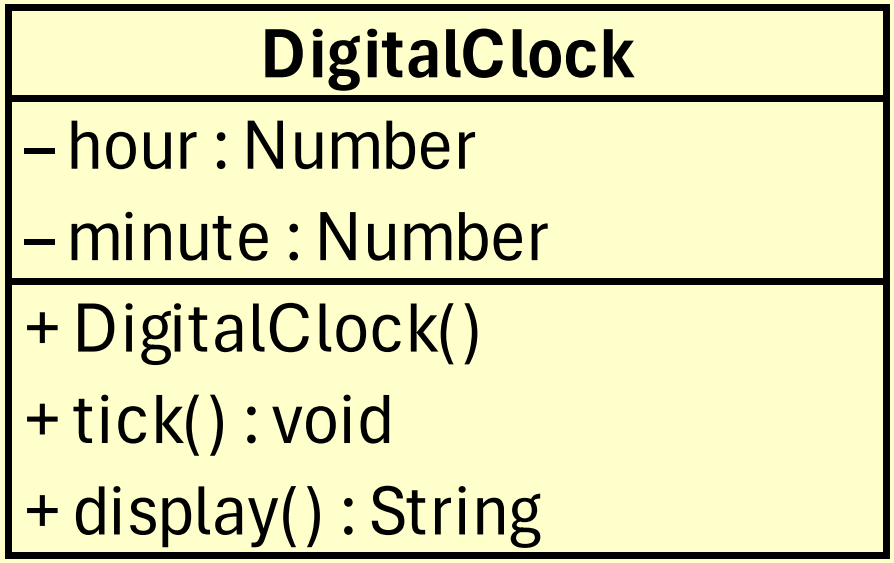
Attributes:

- `hour : Number` — the current hour.
- `minute : Number` — the current minute.

Behaviors:

- `tick()` — Increment the current time by one minute.
- `display()` — Return the current time as a `String`.

Class specification



```
public class DigitalClock {

  private Number hour;
  private Number minute;

  public DigitalClock() {
    hour = new Number(24, 0);
    minute = new Number(60, 0);
  }

  public void tick() {
    minute.increment();
    if (minute.getValue() == 0) hour.increment();
  }

  public String display() {
    return hour.format() + ":" + minute.format();
  }

}
```