Inf1B

Abstraction and Modularisation

Perdita Stevens
adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics

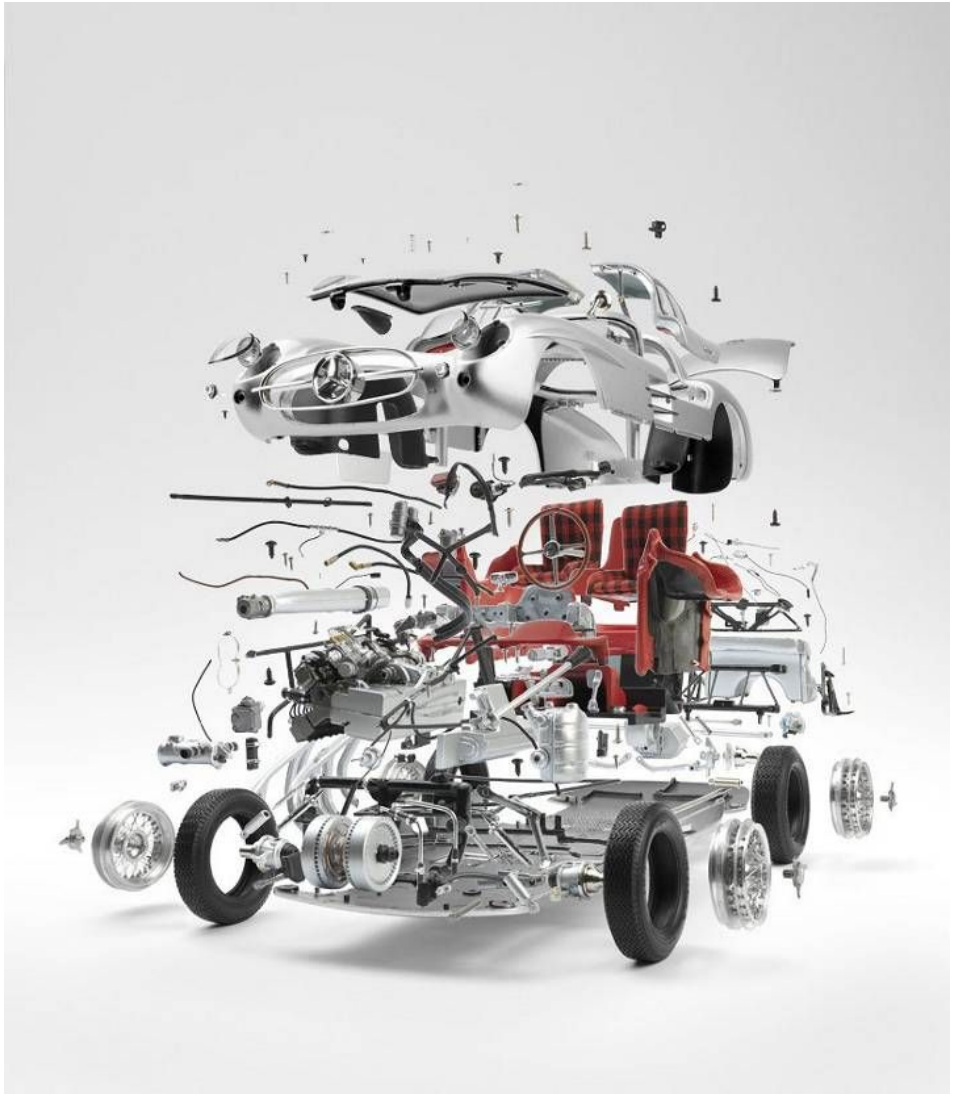
Divide and Conquer using Classes

Like functions, classes can be used to break a problem into several (often more trivial) sub problems which can be tackled one at a time.

Divide and Conquer using Classes

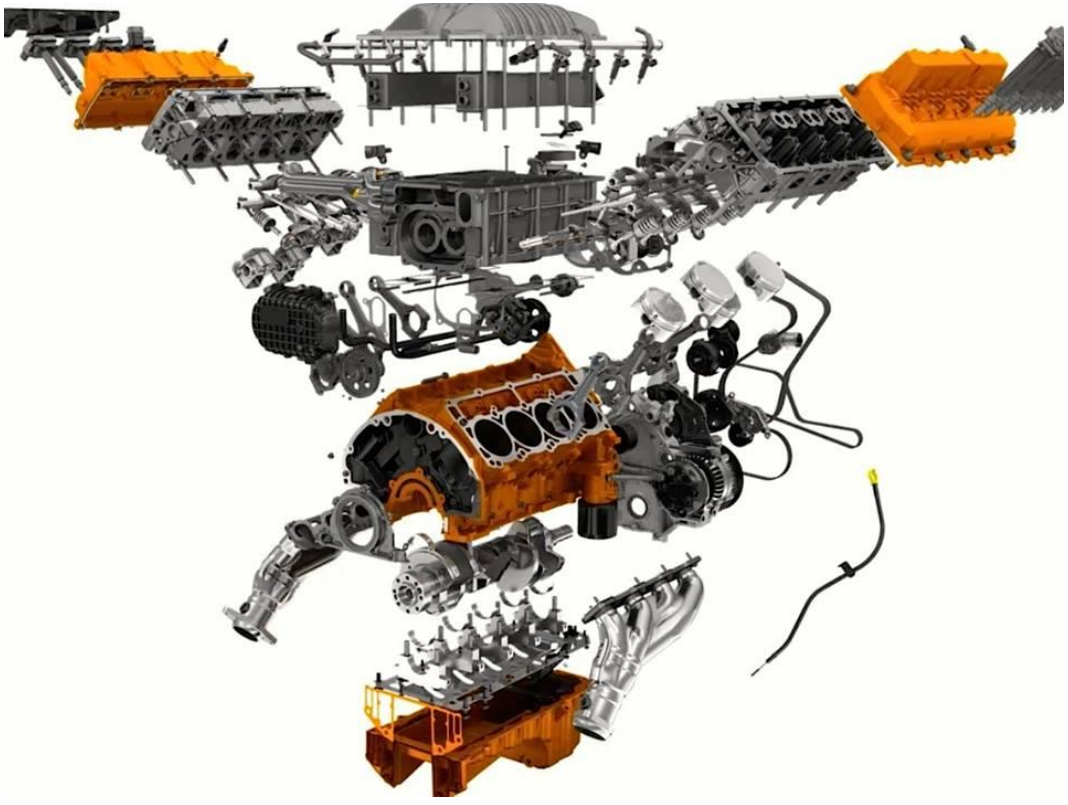
Source: `https://www.brabus.com/_Resources/Persistent/3/f/5/4/3f54c5290ebf3b440fb123df40dbb1754de77941/C4S_034C%20%288%29-604x400.jpg?bust=3f54c529`

Divide and Conquer using Classes



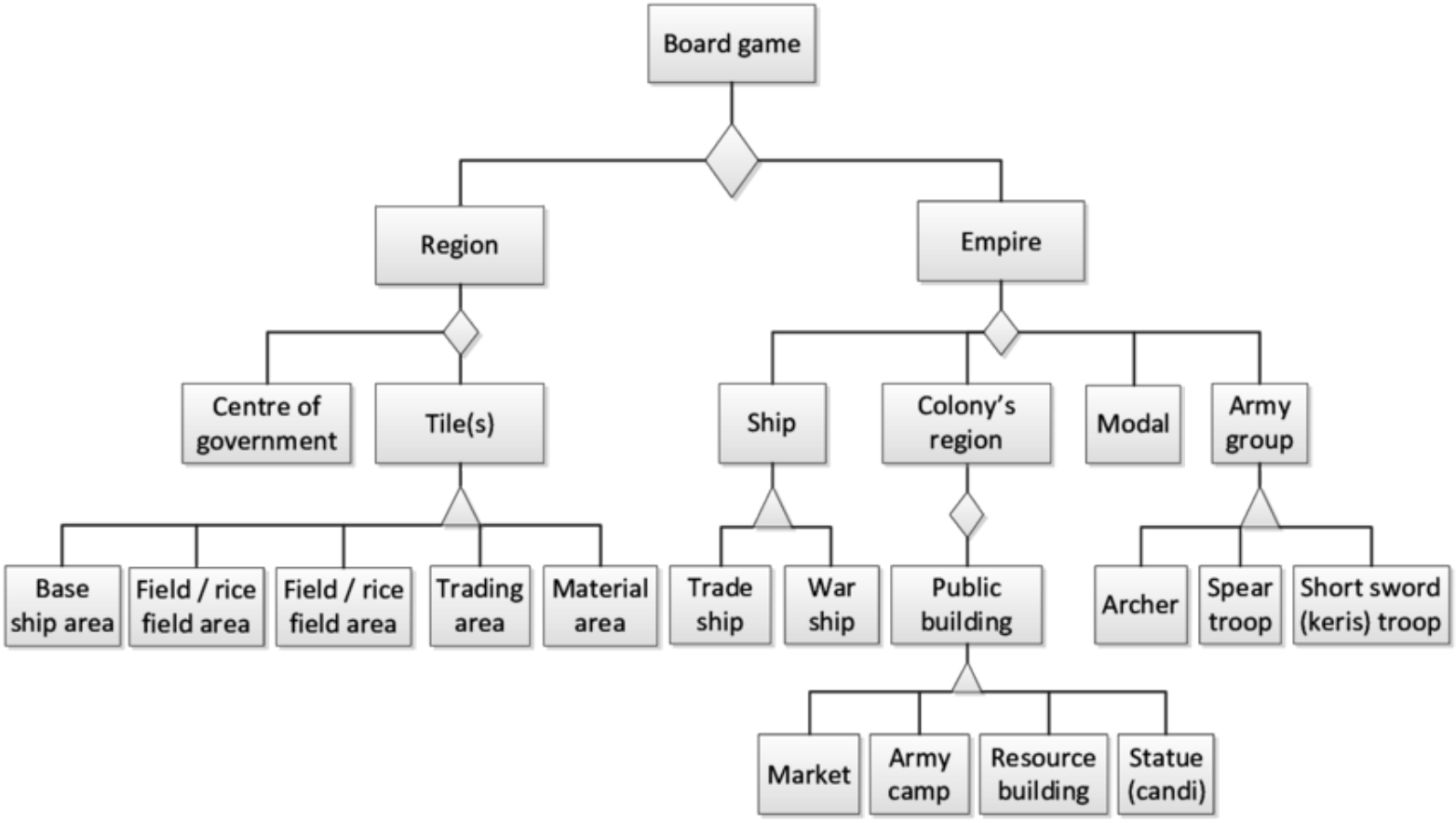
Source: `https://i.pinimg.com/originals/4e/77/6b/4e776b9ca336b8be7a1d46c9a5989eff.jpg`

Divide and Conquer using Classes



Source: `https://static.carthrottle.com/workspace/uploads/posts/2015/12/0dcc34fe3d640be61b251270d497ef49.jpg`

Divide and Conquer using Classes



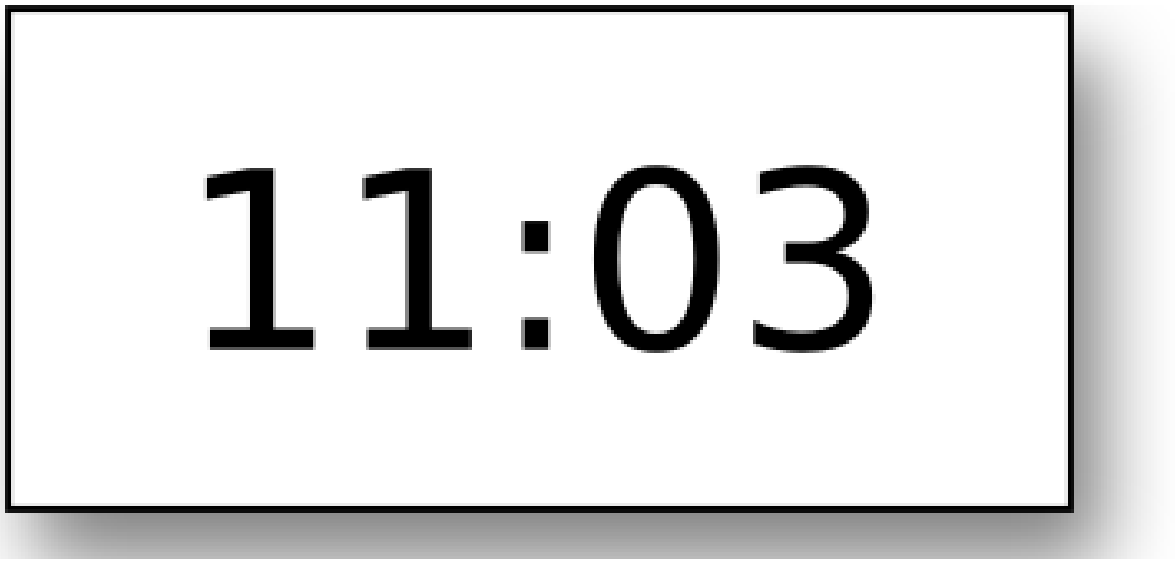
Source: Liliana, Liliana et al. Interactive game design for learning of united nusantara in the Majapahit era. ARPN Journal of Engineering and Applied Sciences

Abstraction and Modularisation

Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.

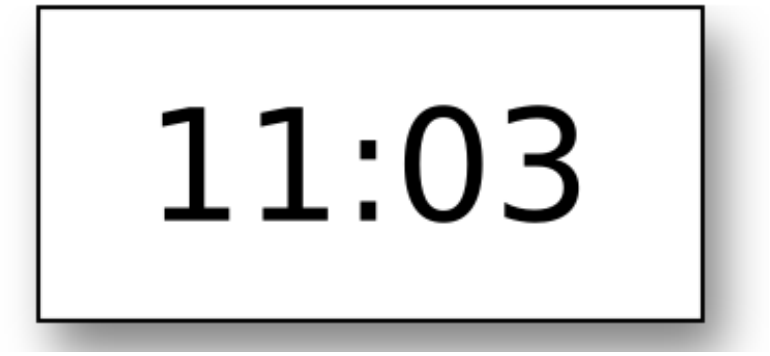
Modularisation is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

Clock Display



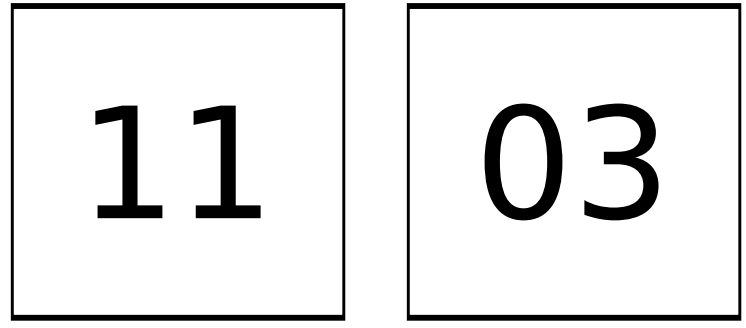
Software to display a clock with two numbers, one for hours and one for minutes.

Modularising the Clock Display

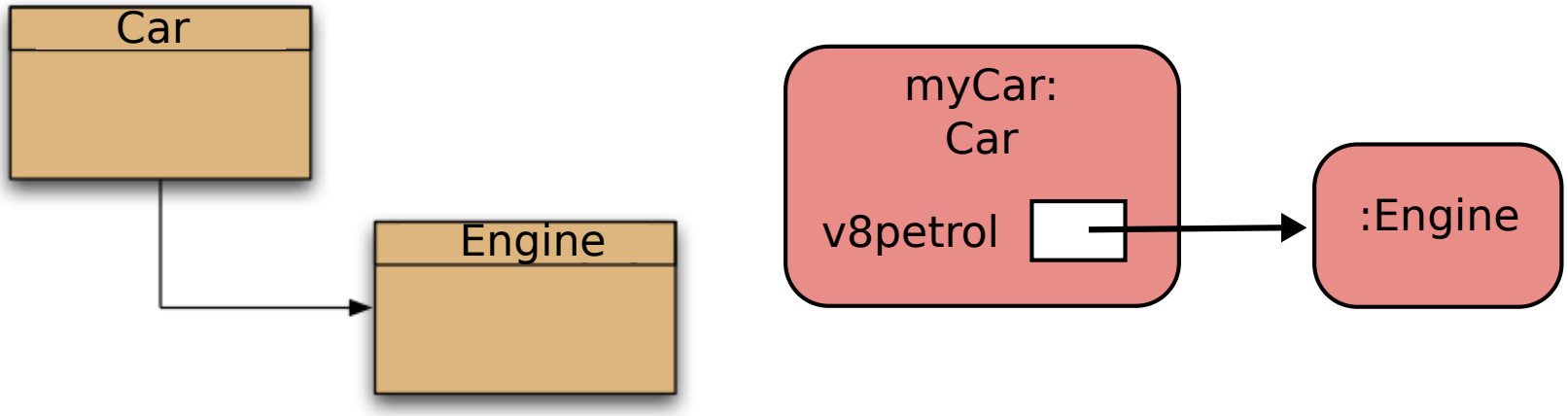


One four-digit display?

Or two two-digit displays?



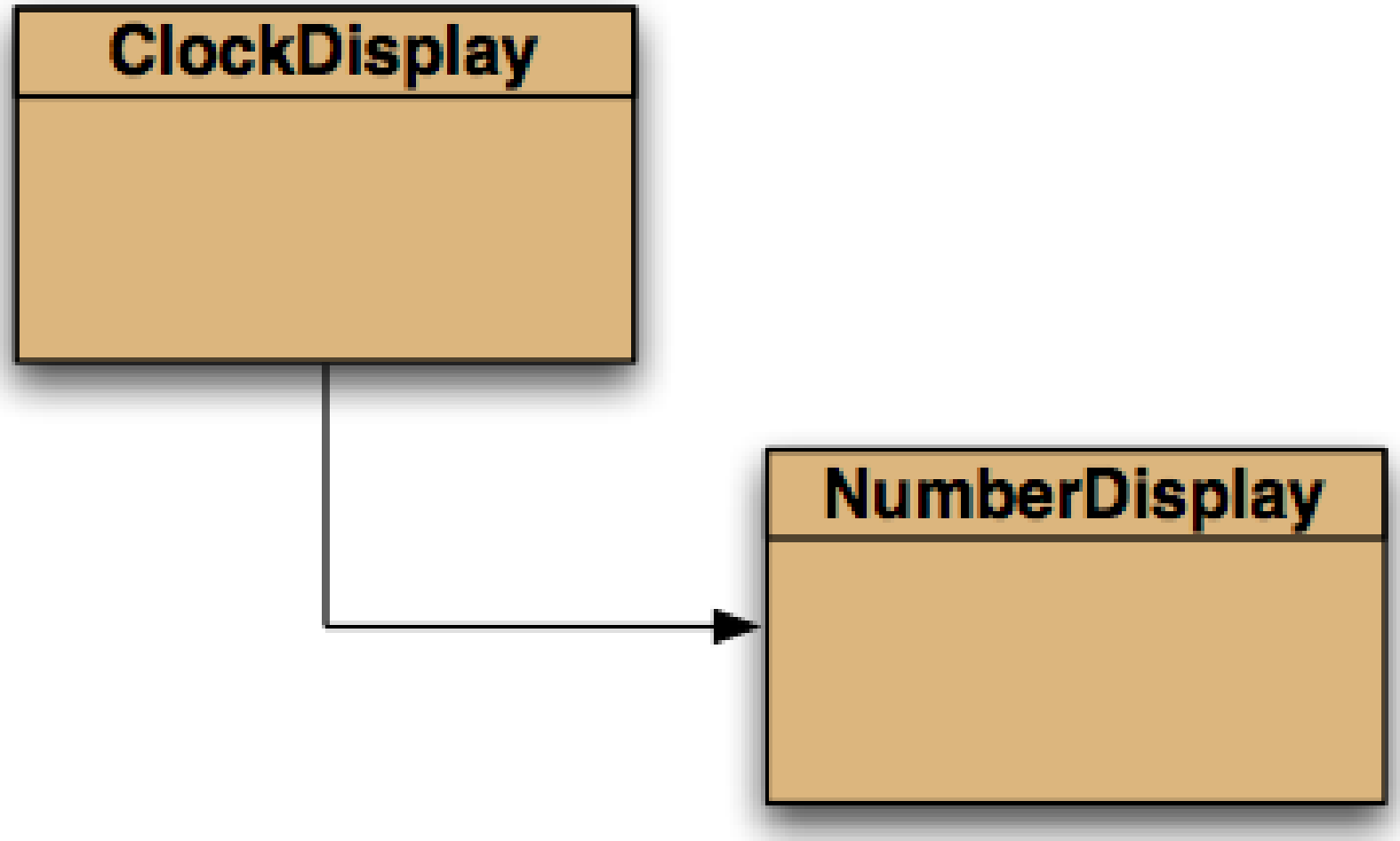
Object Diagram vs. Class Diagram



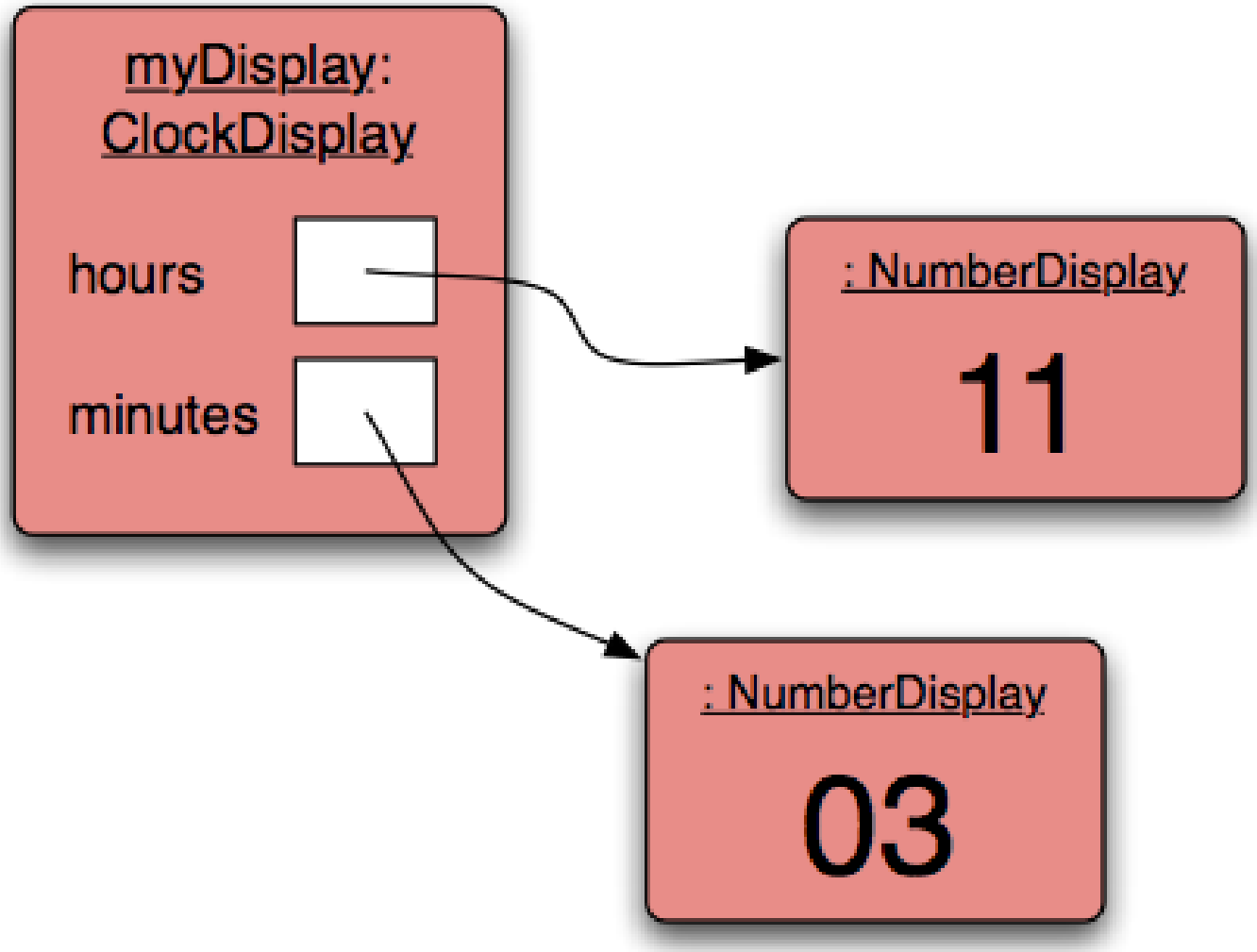
Class Diagram shows classes and relationship between them on a source code level (static view)

Object Diagram shows objects and relationships between them at one moment during execution (dynamic view)

Clock Display - Class Diagram



Clock Display - Object Diagram

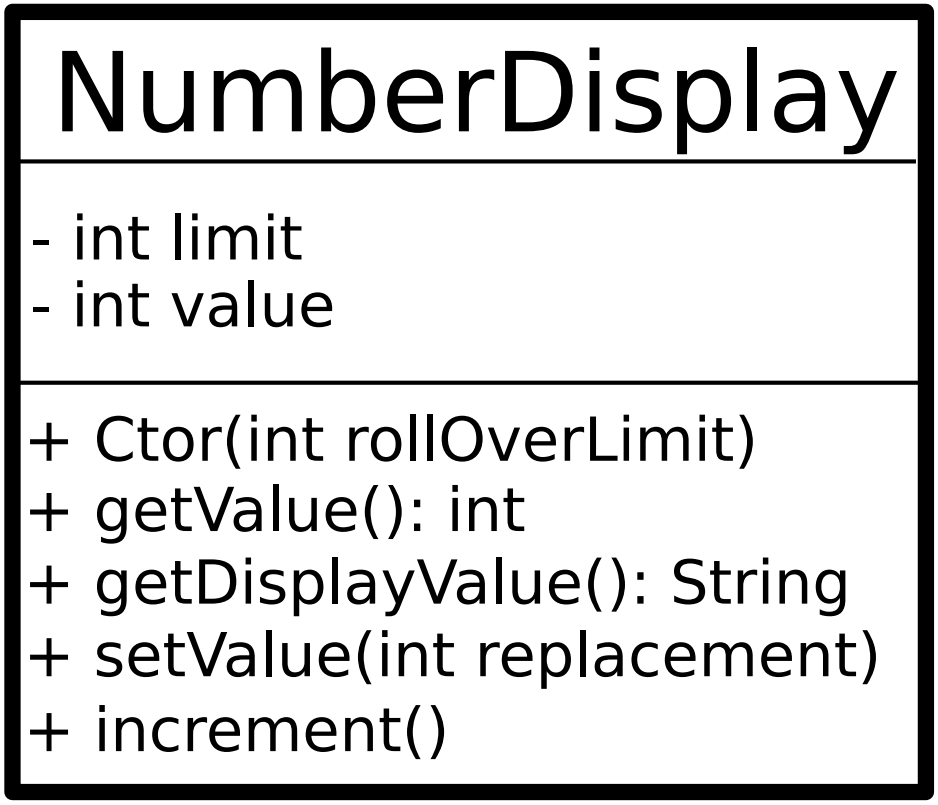


Clock Display

```
public class NumberDisplay {
    private int limit;
    private int value;

    // ctor and methods omitted
}

public class ClockDisplay {
    private NumberDisplay hours;
    private NumberDisplay minutes;

    // Ctor and methods omitted
}
```

NumberDisplay is used like static type in ClockDisplay.

Number Display API



Number Display Constructor

```
public NumberDisplay(int rollOverLimit) {
      limit = rollOverLimit;
      value = 0;
}
```

Reminder, Constructors ...

- have same name as class
- have no return type
- initialise the state of the object instance

Interlude: Encapsulation

...or, why do instance variables have to be `private`?

Dalek Encapsulation: Unprotected Dalek

```
public class Dalek {
    public double batteryCharge = 5;
    public void batteryReCharge(double c) {...}
    public void move(int distance) {...}
}
```

Disabling the Dalek:

```
Dalek d = new Dalek(); // start off with a
                       // well-charged battery
d.batteryCharge = Double.NEGATIVE_INFINITY;
d.batteryReCharge(1000); // battery charge still -Infinity!
```

Dalek Encapsulation: Protected Dalek!

```
public class Dalek {
    private double batteryCharge = 5;
    public void batteryReCharge(double c) {...}
    public void move(int distance) {...}
}
```

Disabling the Dalek:

```
Dalek d = new Dalek(); // start off with a
                       // well-charged battery
d.batteryCharge = Double.NEGATIVE_INFINITY;
//Triggers compile-time Error
```

```
Exception ...: Unresolved compilation problem:
  The field Dalek.batteryCharge is not visible
```

Changing Internal Representation

Encapsulation:

- Keep data representation hidden with `private` access modifier.
- Expose API to clients using `public` access modifier.

Advantage: can switch internal representations without changing client.

Encapsulated data types:

- Don't touch data to do whatever you want.
- Instead, ask object to manipulate its data.

Access Modifiers Summary

Modifier	Class	Package	Global
Public	Yes	Yes	Yes
Default	Yes	Yes	No
Private	Yes	No	No

Access Modifiers Summary

Modifier	Class	Package	Global
Public	Yes	Yes	Yes
Default	Yes	Yes	No
Private	Yes	No	No

Don't use the default modifier - bad style (see INF1B Coding Conventions)

Access Modifiers Summary

Modifier	Class	Package	Global
Public	Yes	Yes	Yes
Default	Yes	Yes	No
Private	Yes	No	No

Don't use the default modifier - bad style (see INF1B Coding Conventions)

There is a fourth modifier which you will get to know later.

Number Display Instance Members

```
public class NumberDisplay {

    private int limit;
    private int value;

    public NumberDisplay(int rollOverLimit) {
          limit = rollOverLimit;
          value = 0;
    }
}
```

Reminder: Object state has default values!

Number Display Immutable Limit

```
public class NumberDisplay {

    private final int limit;
    private int value;

    public NumberDisplay(int rollOverLimit) {
          limit = rollOverLimit;
          value = 0;
    }
}
```

Immutable state improves type safety!

Immutability

Immutable data type: object's internal state cannot change once constructed.

mutable	*immutable*
`Picture`	
`Dalek`	`String`
Java arrays	primitive types

Immutability: Advantages and Disadvantages

Immutable data type: object's value cannot change once constructed.

Advantages:

- Makes programs easier to debug (sometimes)
- Limits scope of code that can change values
- Pass objects around without worrying about modification
- Better for concurrent programming.

Disadvantages: New object must be created for every value.

The `final` Modifier

Final: declaring a variable to by `final` means that you can assign it a value only once, in initializer or constructor. E.g., Daleks come in three versions, Mark I, Mark II and Mark III.

```
public class Dalek {
    private final int mark;
    private double batteryCharge;
    ...
 }
```

this value doesn't change once the object is constructed

this value can be change by invoking the instance method `batteryReCharge()`

Advantages:

- Helps enforce immutability.
- Prevents accidental changes.
- Makes program easier to debug.
- Documents the fact that value cannot change.

Number Display Accessors

```
private final int limit;
private int value;

// ctor omitted

public int getValue() {
    return value;
}
```

Sometimes a way of accessing the object's state is needed.

Number Display Accessors

```
private int limit;
private int value;

// methods omitted

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

Here is a more sophisticated accessor. Before returning data, the class has a chance to modify it.

Number Display Accessors

```
private int limit;
private int value;

// methods omitted

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

Here is a more sophisticated accessor. Before returning data, the class has a chance to modify it.

NB note the conversion trick for String and number

Number Display Mutator

```
private int limit;
private int value;

// methods omitted

public void setValue(int replacementValue) {
   if (   (replacementValue >= 0)
       && (replacementValue < limit)) {
      value = replacementValue;
   }
}
```

Sometimes, you also want to change the state of an object directly.

Number Display Mutator

```
private int limit;
private int value;

// methods omitted

public void setValue(int replacementValue) {
   if (   (replacementValue >= 0)
       && (replacementValue < limit)) {
      value = replacementValue;
   }
}
```

Sometimes, you also want to change the state of an object directly.

This is only interesting for mutable objects!

Number Display Mutator

```
private int limit;
private int value;

// methods omitted

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

Here is a more sophisticated mutator.

Getters and Setters

Encapsulation: instance variables should be private

```
public class Student {

    private String firstName;
    private String lastName;
    private String matric;

    public Student(String fn, String ln, String m) {
        firstName = fn;
        lastName = ln;
        matric = m;
    }
 }
```

Getters and Setters

Encapsulation: instance variables should be private

```
public class StudentTester {

   public static void main(String[] args) {
        Student student = new Student("Fiona", "McCleod", "s01234567");
        System.out.println(student.firstName);
        student.matric = "s13141516";
    }
}
```

we cannot assign to this variable!

we cannot read this variable!

Getters and Setters

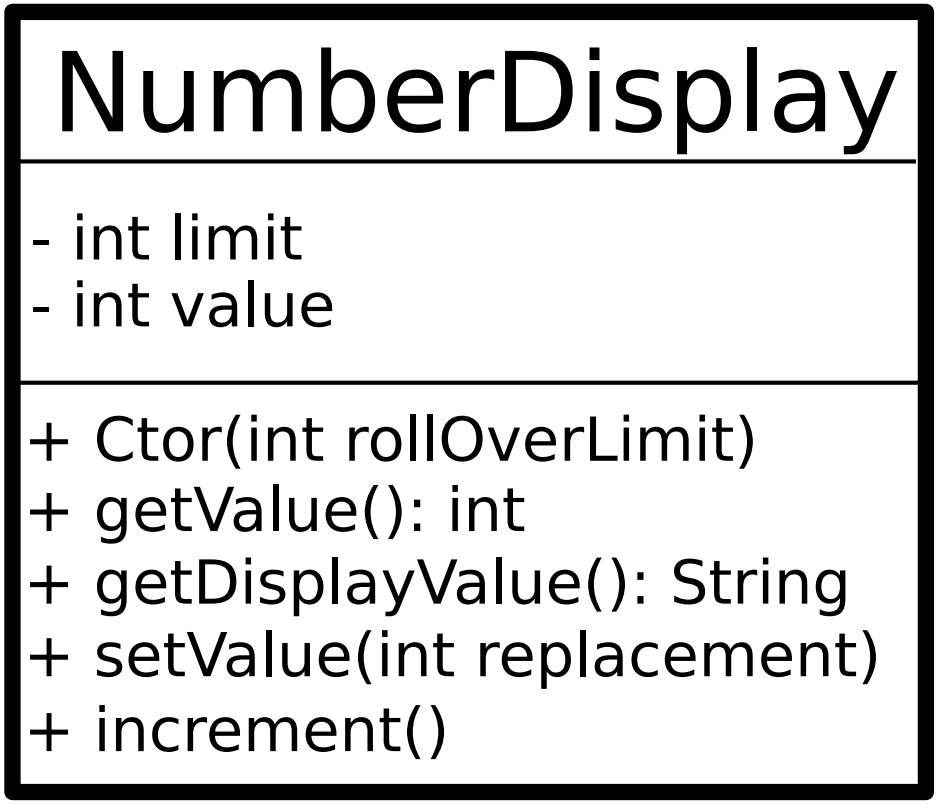
Encapsulation: instance variables should be private

- We use instance methods to mediate access to the data in private instance variables, as needed.
- Accessor methods: just read the data
- Mutator methods: modify the data
- Java convention: given an instance variable `myData`, use
  - `getMyData()` method to read the data, and
  - `setMyData()` method to write to the data.
- Often called 'getters' and 'setters' respectively.

Getters and Setters

```
public class Student {

    private String firstName, lastName, matric, tutGroup;

    public Student(String fn, String ln, String m) {
        ...
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getMatric() {
        return matric;
    }
 }
```

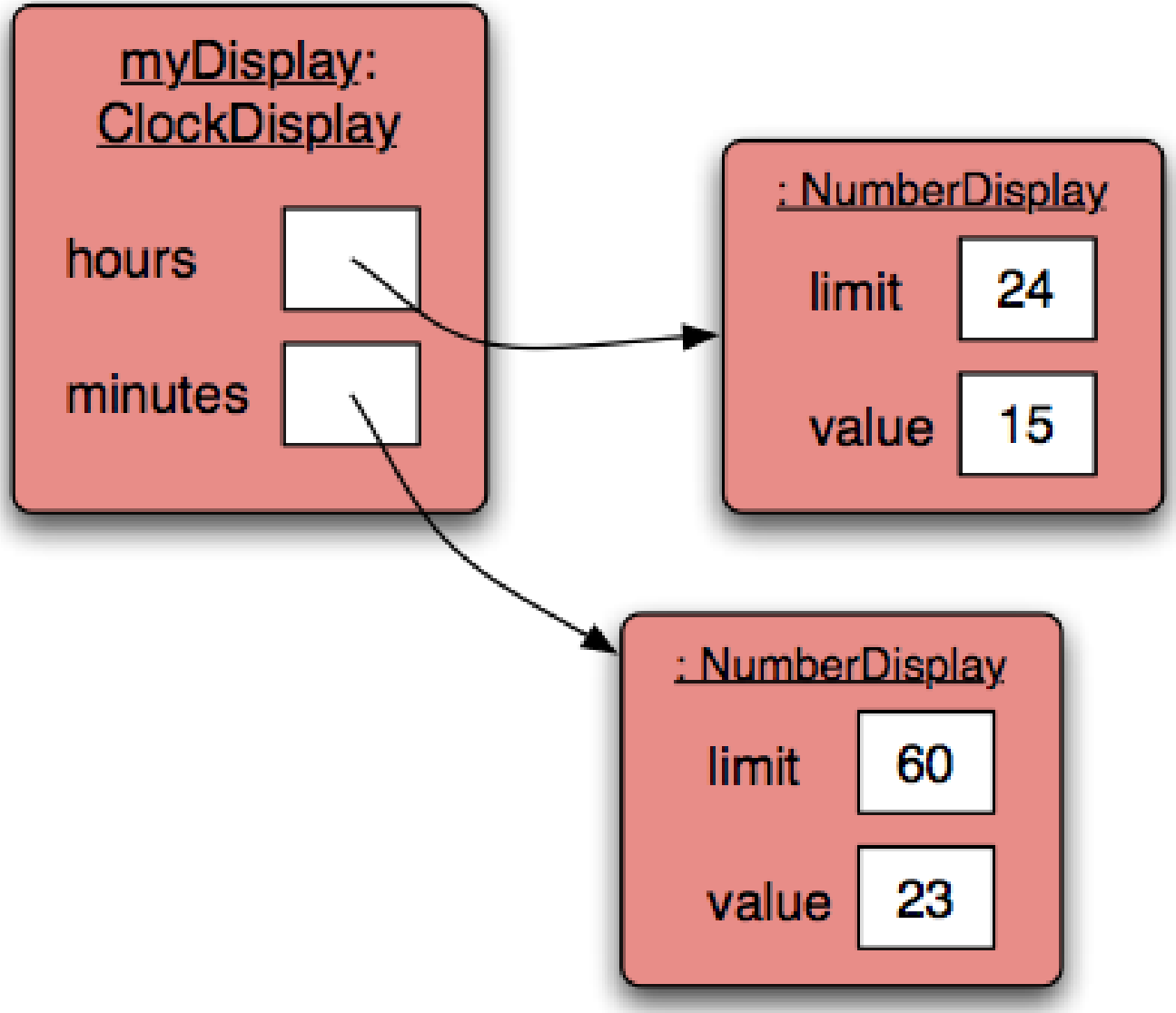
Number Display API



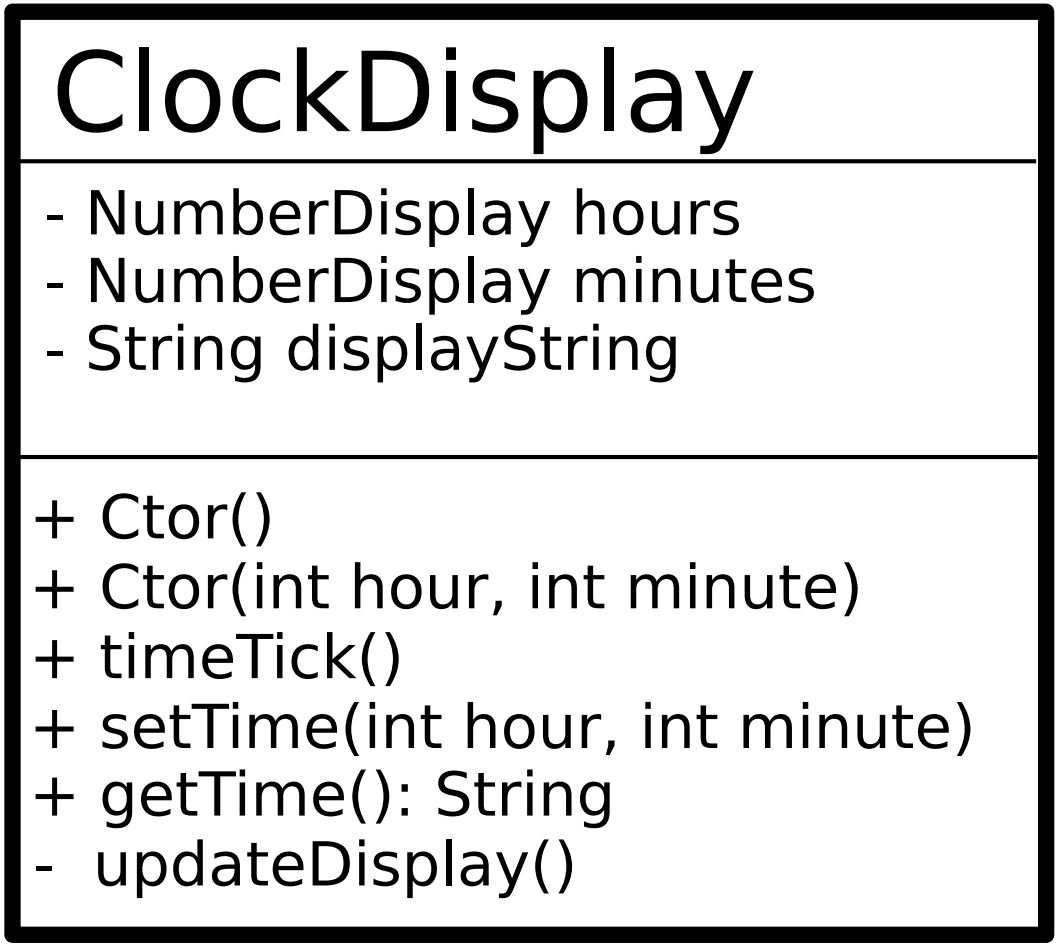
Number Display Source

```
public class NumberDisplay {
  private final int limit;
  private int value;

  public NumberDisplay(int rollOverLimit) {
    limit = rollOverLimit;
    value = 0;
  }

  public int getValue() { return value; }

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

  public void setValue(int replacementValue) {
     if (   (replacementValue >= 0)
         && (replacementValue < limit)) {
        value = replacementValue;
     }
  }

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

Clock Display - Object Diagram Extended



Clock Display API



Clock Display Accessors

```
/**
* Return representation of the actual clock.
* @return current time on the clock in text format
*/
public String getTime() {
    return displayString;
}
```

Clock Display Mutators

```
public void setTime(int hour, int minute) {
    hours.setValue(hour);
    minutes.setValue(minute);
    updateDisplay();
}

public void timerTick() {
    minutes.increment();
    if (minutes.getValue() == 0) { // it just rolled over
        hours.increment();
    }
    updateDisplay();
}

private void updateDisplay() {
    displayString = hours.getDisplayValue()
        + ":" + minutes.getDisplayValue();
}
```

Note that updateDisplay is private and used internally.

Clock Display Constructors

```
private NumberDisplay hours;
private NumberDisplay minutes;
private String displayString;

public ClockDisplay() {
  hours = new NumberDisplay(24);
  minutes = new NumberDisplay(60);
  updateDisplay();
}

public ClockDisplay(int hour, int minute) {
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    setTime(hour, minute);
}
```

Clock Display Constructors

```
private NumberDisplay hours;
private NumberDisplay minutes;
private String displayString;

public ClockDisplay() {
  hours = new NumberDisplay(24);
  minutes = new NumberDisplay(60);
  updateDisplay();
}

public ClockDisplay(int hour, int minute) {
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    setTime(hour, minute);
}
```

The ClockDisplay class is overloading its constructor!

Method Overloading

Overloading: two methods with same name but different parameter lists.

Overloaded `add`

```
public int add(int a, int b) { ... }
public int add(float a, float b) { ... }
```

Overloaded `println`

```
System.out.println(3); // int
System.out.println(3.0); // double
System.out.println((float) 3.0); // cast to float
System.out.println("3.0"); // String
```

Summary: Abstraction and Modularisation

Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.

Modularisation is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

Summary: Access Control

Encapsulation and visibility: All the instance variables and methods (i.e., members) of a class are visible within the body of the class.

Access modifiers: control the visibility of your code to other programs.

`public`:	member is accessible whenever the class is accessible.
`private`:	member is only accessible within the class.
default:	member is accessible by every class in the same package.

Benefits of encapsulation:

- Loose coupling
- Protected variation
- Exporting an API:
  - the classes, members etc, by which some program is accessed
  - any client program can use the API
  - the author is committed to supporting the API

Reading

Objects First

Chapter 3 *Object Interaction*