Abstraction and Modularization

Object Interaction

Produced by: Dr. Siobhán Drohan

(based on Chapter 3, Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling)

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

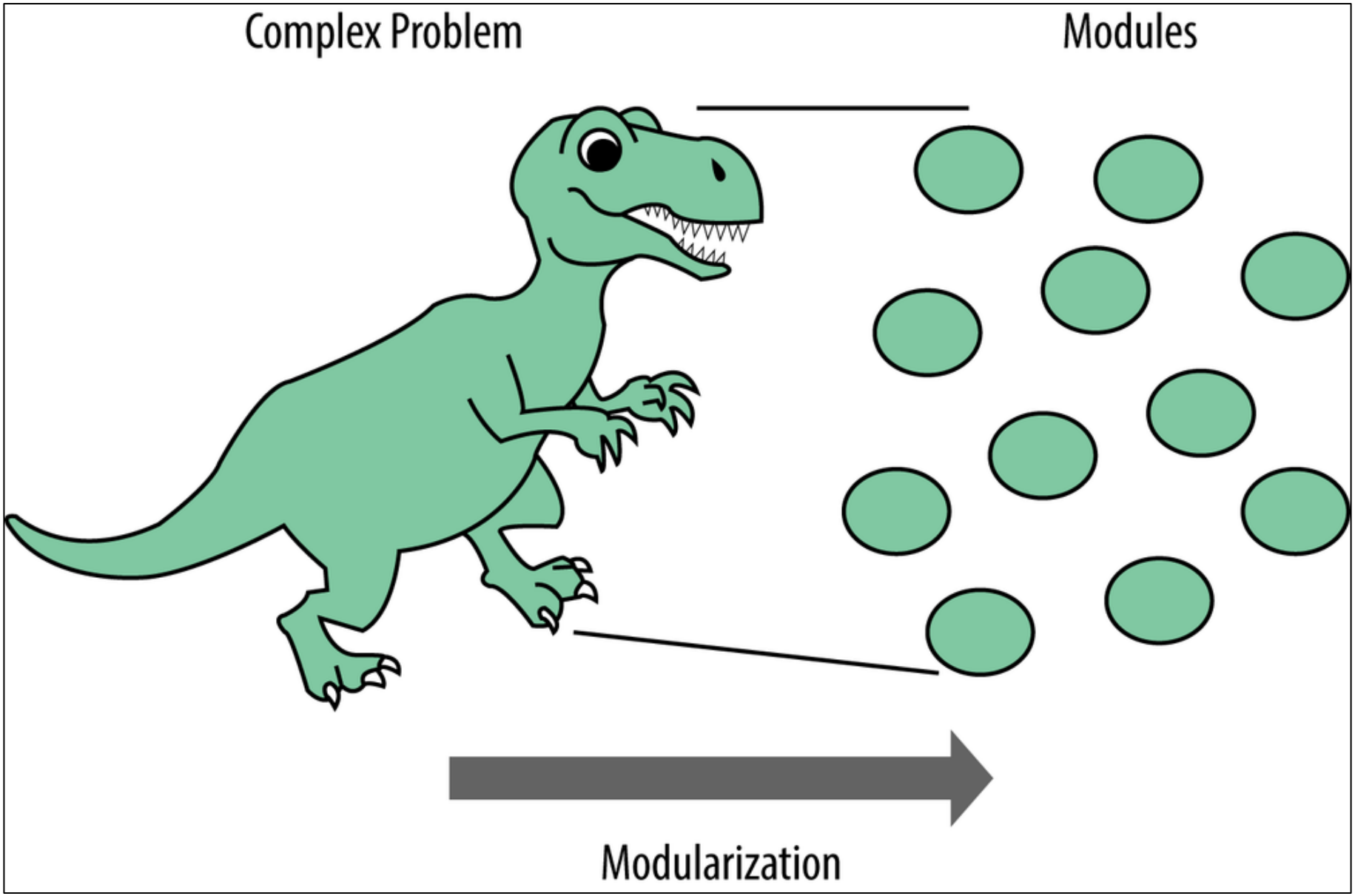
Topic List

- Divide and conquer: Abstraction and modularization.
- Demo of the digital clock-display project.
- Class and object diagrams.
- Implementing the clock display.
- Concepts covered in the clock-display project.
- Review of certain methods in the clock-display project.

Divide and Conquer Principle

- Applies to all problem solving.
  - Break down problem into parts small enough to solve.
  - Attack each sub-problem separately.
  - Combine the sub-problem solutions to solve the overall problem.

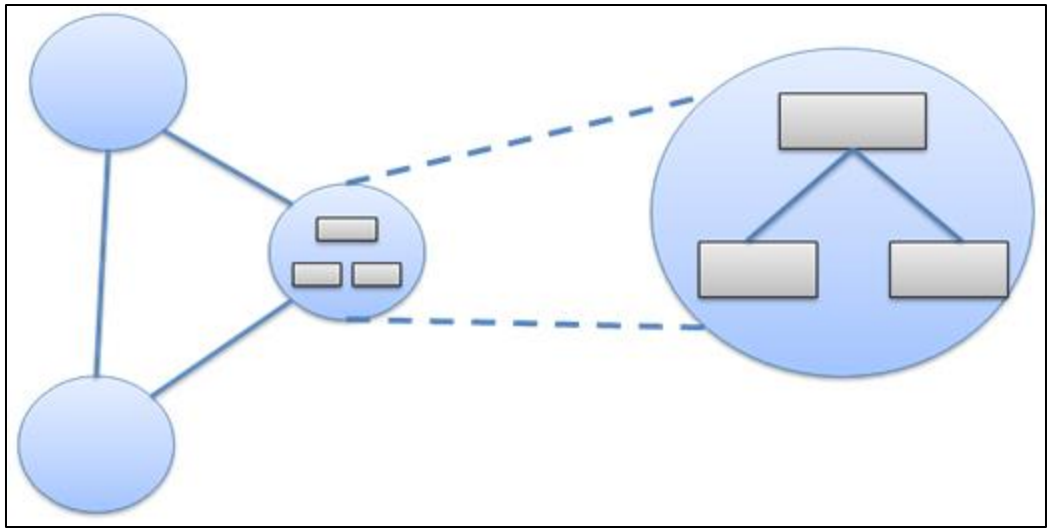
- In programming, this is where abstraction and modularization comes in!



https://www.safaribooksonline.com/library/view/learning-php-design/9781449344900/ch01.html

Abstraction

- **Abstraction**
  - the ability to ignore details of parts to focus attention on a higher level of a problem i.e. the bigger picture...the dinosaur!

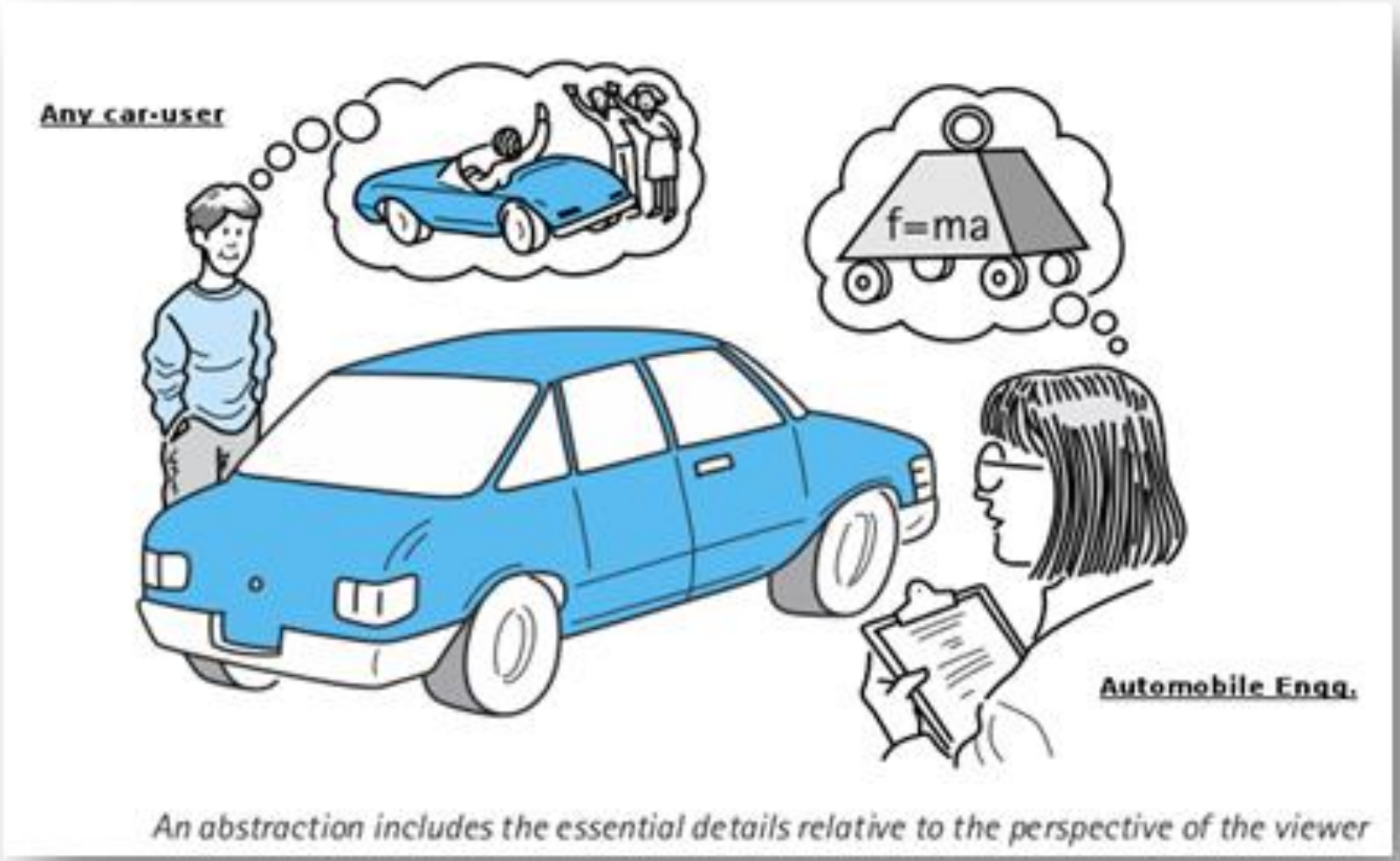


https://reachmnadeem.wordpress.com/2014/05/15/very-basic-object-oriented-concepts/

Why Abstraction?

- We don’t need to know the individual details of something that is already built for us; we just need to know how to use it.

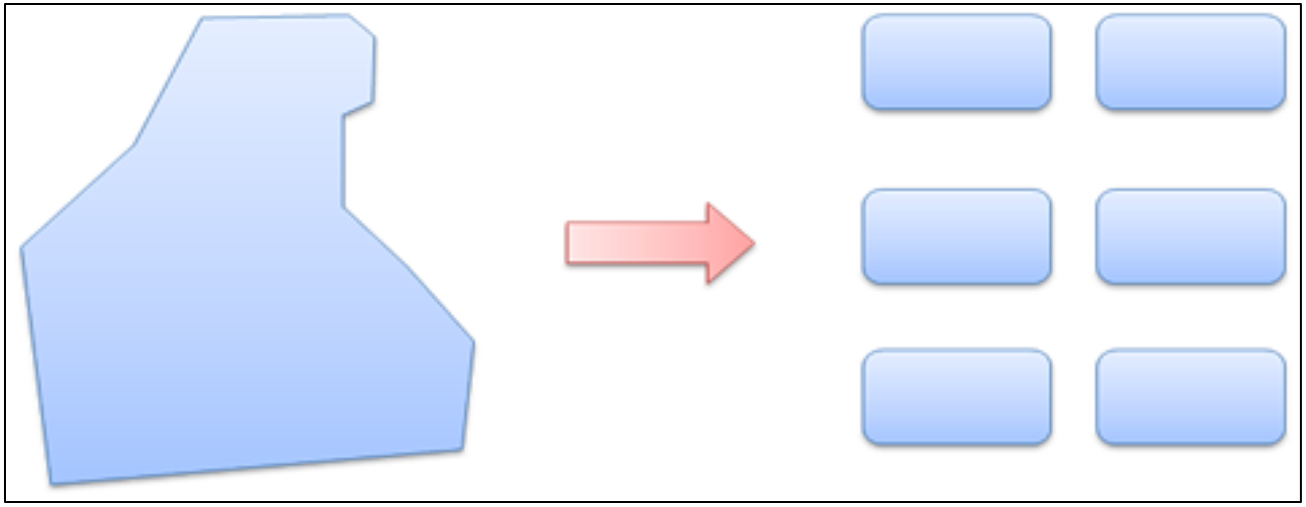
- For example:
  - we have used the ***Canvas*** class without needing to know how it was coded.



https://reachmnadeem.wordpress.com/2014/05/15/very-basic-object-oriented-concepts/

Modularization

- **Modularization**
  - Decompose the problem into smaller sub problems that can be solved separately.



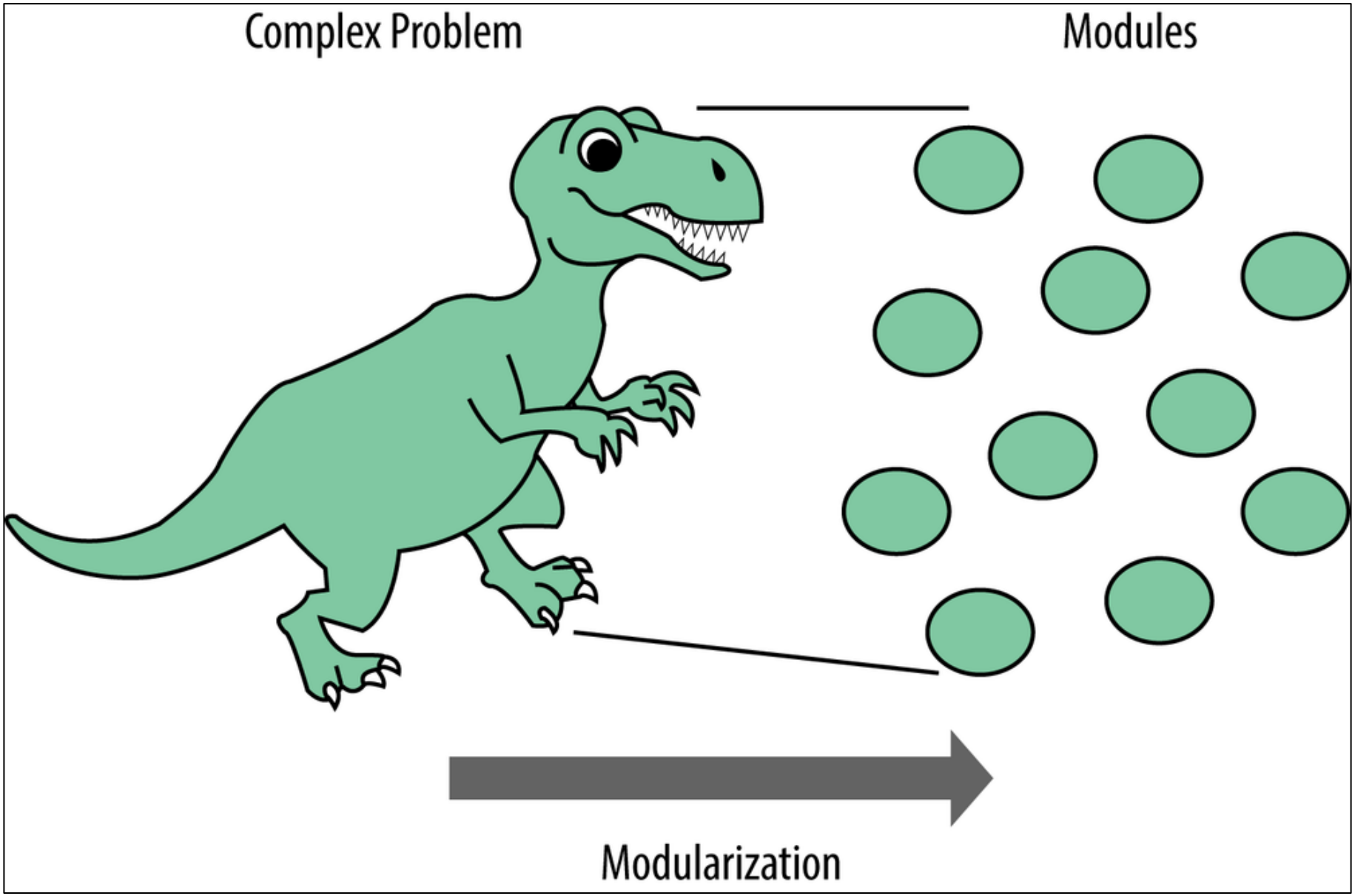
https://reachmnadeem.wordpress.com/2014/05/15/very-basic-object-oriented-concepts/

Why Modularization?

- Trivial problems (like T***icketMachine***) can be solved in a single class.

- As systems become more complex, one class is just not enough.
  - → In these cases, identify subcomponents in the problem that can be turned into separate classes.
  - → For example:

    our Shapes project had many classes i.e. Canvas, Square, Circle, etc.

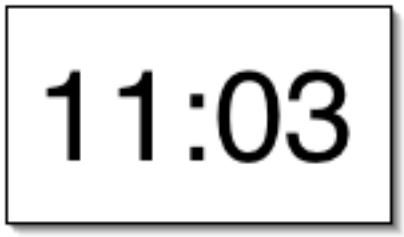


https://www.safaribooksonline.com/library/view/learning-php-design/9781449344900/ch01.html

demo

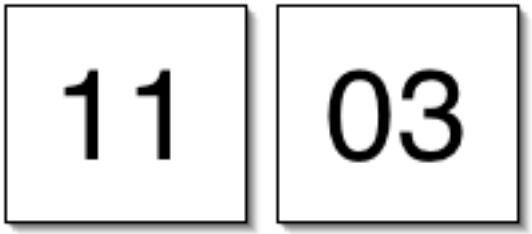
The digital ***clock-display*** project

Modularizing the clock display

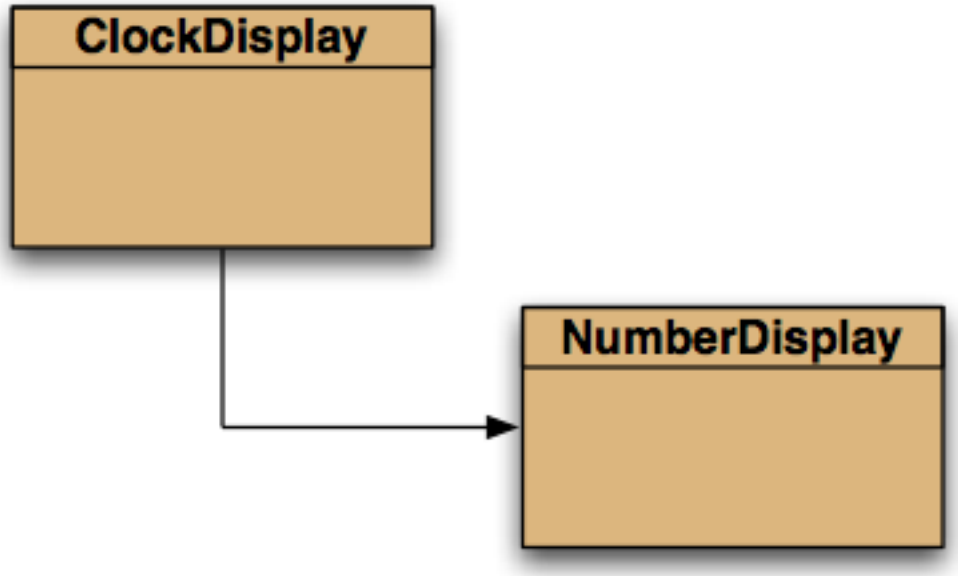


One four-digit display?

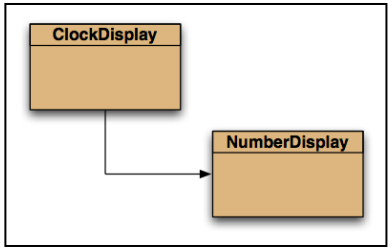
Or two two-digit displays?

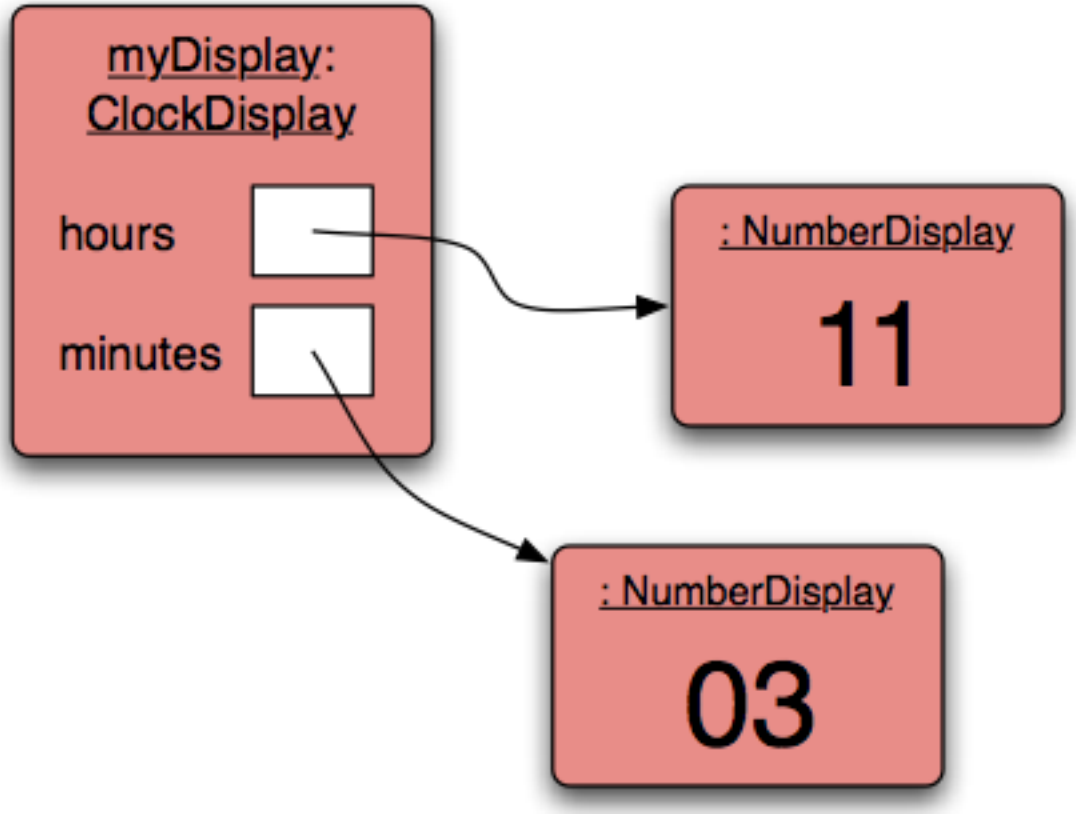


Class diagram

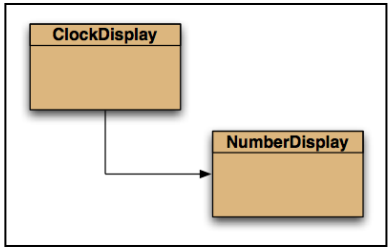


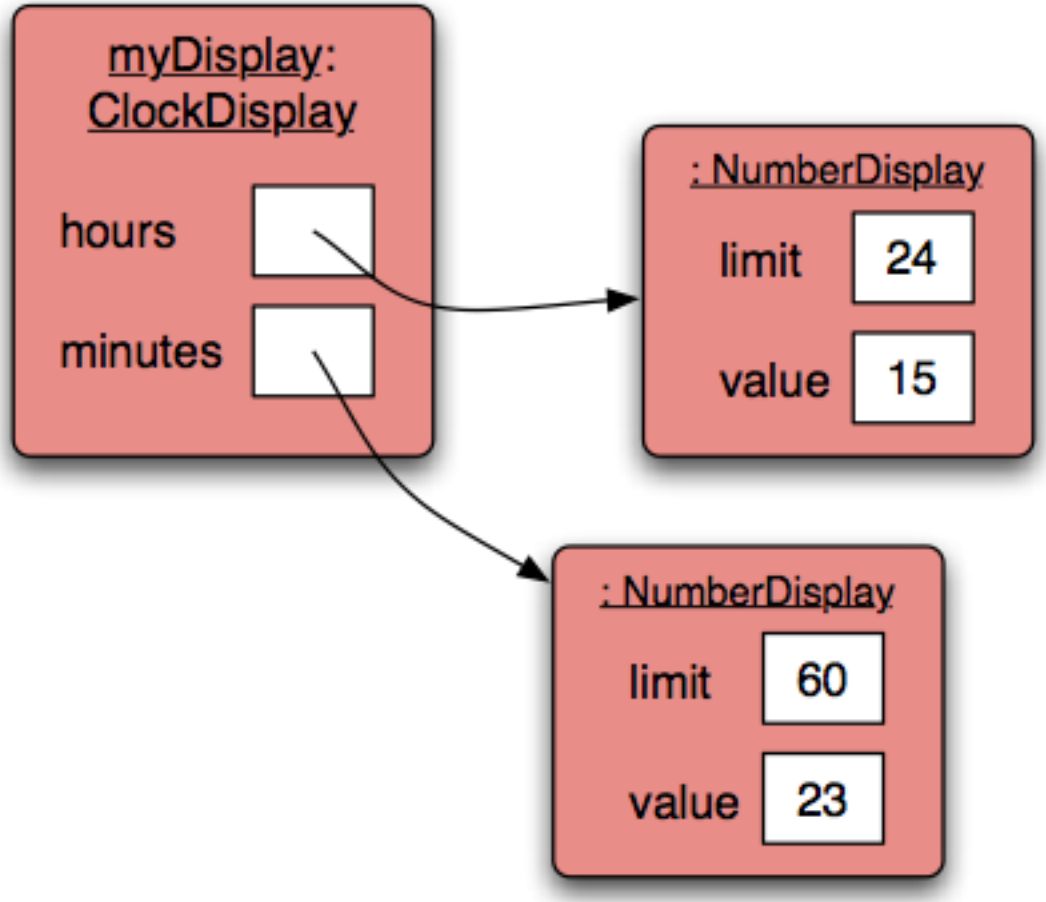
ClockDisplay: object diagram





ClockDisplay: object diagram





Implementation - NumberDisplay

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

    //Constructors and
    //methods omitted.
}
```

Implementation - ClockDisplay

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

    //Constructors and
    //methods omitted.
}
```

Some concepts in the…

- NumberDisplay source code:
  - Modulo operator
  - Logical operators
- ClockDisplay source code:
  - Objects creating objects
  - null
  - Multiple constructors
  - Internal method calls
  - External method calls
  - Dot notation

The modulo operator

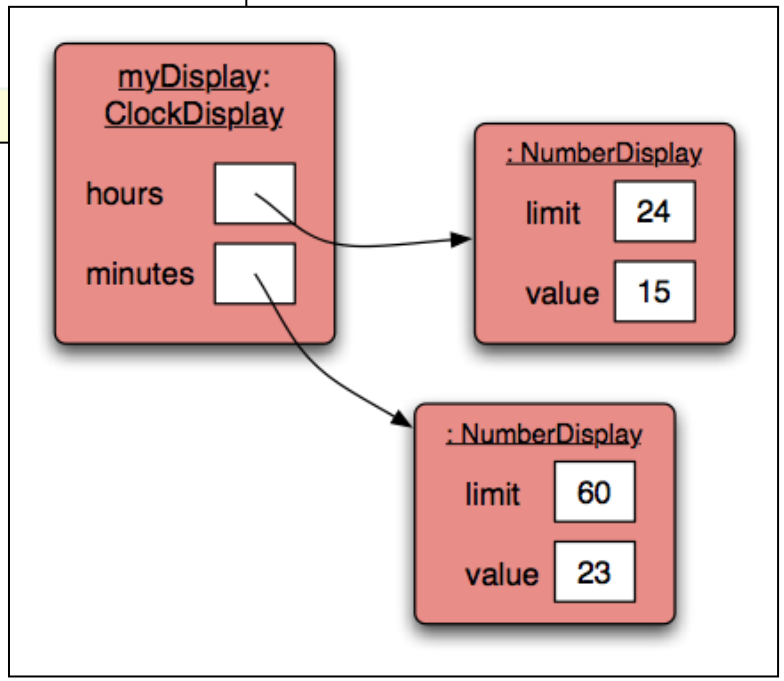
- The 'division' operator (/), when applied to int operands, returns the **result** of an integer division.

- The 'modulo' operator (%) returns the remainder of an integer division.

  - In Maths:
    17 / 5 = result 3, remainder 2
  - In Java:
    17 / 5 = 3
    17 % 5 = 2

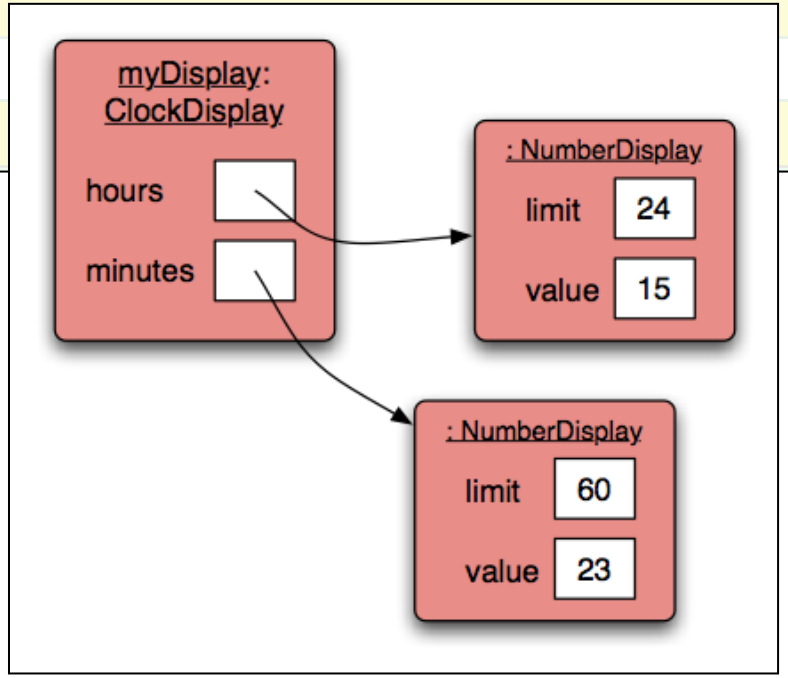
Modulo in the NumberDisplay class

```
/**
 * Constructor for objects of class NumberDisplay.
 * Set the limit at which the display rolls over.
 */
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;
    value = 0;
}
```



Modulo in the NumberDisplay class

```
/**
 * Increment the display value by one, rolling over to zero if the
 * limit is reached.
 */
public void increment()
{
    value = (value + 1) % limit;
}
```



Arithmetic operators

- So far, we have seen how to add (+) and subtract (-) e.g.:

```
balance = balance + amount;
balance = balance – price;
value = (value + 1) % limit;
```

- Examples of multiply (*) and divide (/):

```
totalCost = unitCost * numberOfItems;
average = sum / numberOfItems;
```

Order of evaluation

- Brackets ()
- Multiplication (*)
- Division (/)
- Addition (+)
- Subtraction (-)

*Note: The **modulo** operator is just between Division and Addition.*

BoMDAS

Beware My Dear Aunt Sally

Order of evaluation

- Brackets ()
- Multiplication (*)
- Division (/)
- Addition (+)
- Subtraction (-)

Quick Quiz:

7 * (4 - 3) + 1
(6 / 2) + (4 - 2) * (2 * 2)

BoMDAS

Beware My Dear Aunt Sally

Order of evaluation

- Brackets ()
- Multiplication (*)
- Division (/)
- Addition (+)
- Subtraction (-)

Quick Quiz:

7 * (4 - 3) + 1 **= 7**

(6 / 2) + (4 - 2) * (2 * 2) **= 11**

BoMDAS

Beware My Dear Aunt Sally

Recap: Logical operators

- Logic operators operate on boolean values.
- They produce a new boolean value as a result.
- The most important ones are:

&&	(and)
||	(or)
!	(not)

Recap: Logical operators

a && b *(and)*

- This evaluates to true if both ***a*** and ***b*** are true.
- It is false in all other cases.

a || b *(or)*

- This evaluates to true if either ***a*** or ***b*** or both are true, and false if they are both false.

!a *(not)*

- This evaluates to true of ***a*** is false, and false if ***a*** is true.

Recap: Logical operators - quiz

```
int a = 5;
int b = 10;
int c = 7;

(a > b) && (a < c)
(a < b) || (c < a)
!(b < a) && (c > b)
```

Note: Try these yourself in the code pad in BlueJ

&& in the NumberDisplay class

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

```
/**
 * Set the value of the display to the new specified value. If the new
 * value is less than zero or over the limit, do nothing.
 */
public void setValue(int replacementValue)
{
    if((replacementValue >= 0) && (replacementValue < limit)) {
        value = replacementValue;
    }
}
```

Mutator method for value field.

Objects creating objects

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
```

Objects creating objects

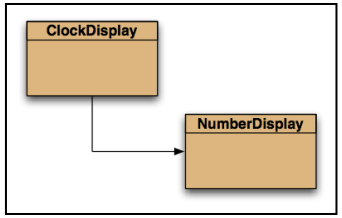
```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
```

Declaring two object-type instance fields

Instantiating the hours and minutes objects.

Objects creating objects



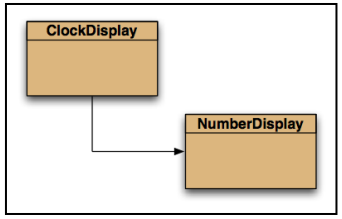

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
```

NumberDisplay is the other class in our clock-display project…it is the **type** of our variables, **hours** and **minutes**.

hours and **minutes** are the names of the instance fields of type NumberDisplay.
Each holds a reference to the object of type NumberDisplay.

Objects creating objects



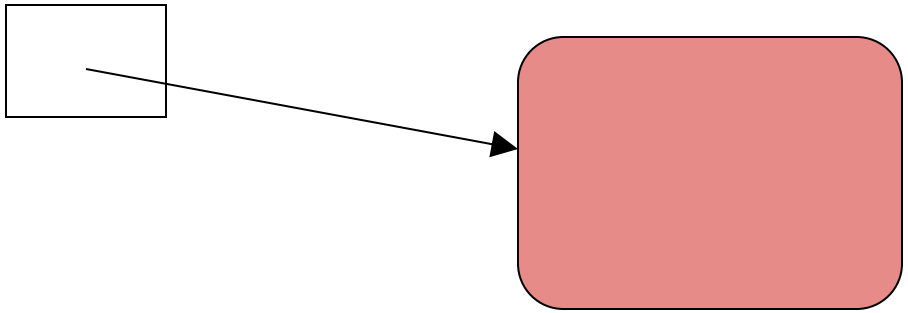

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
```

NumberDisplay(24) is a call to the constructor in the NumberDisplay class, passing an actual parameter of 24 into the constructor. The call to this constructor creates an object of type NumberDisplay.

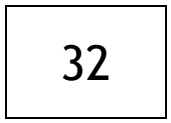
Primitive types vs. object types

```
private NumberDisplay hours;
hours = new NumberDisplay(24);
```



object type

```
int i;
```



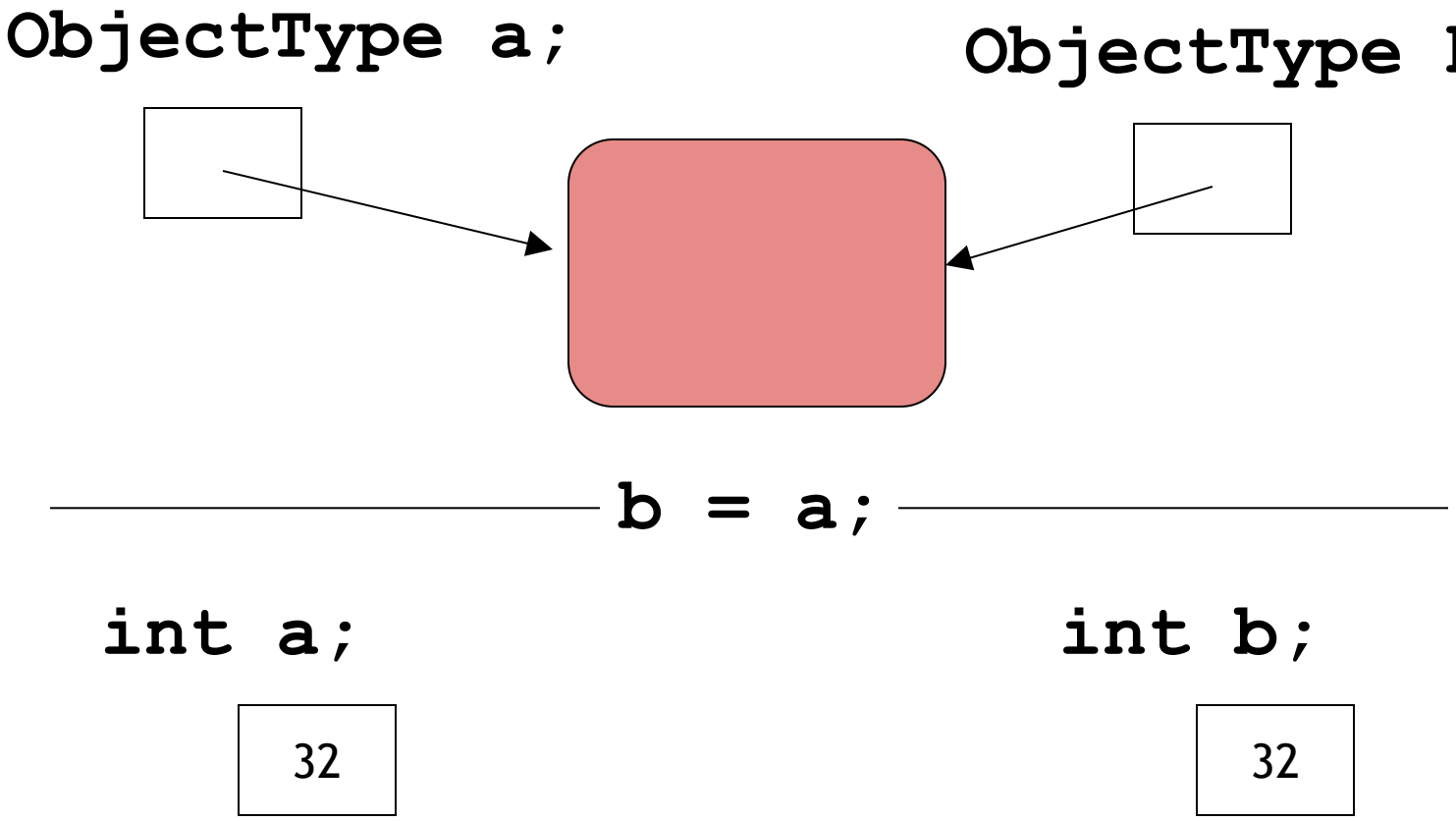
primitive type

Quiz: What is the output?

- ```
  int a;
  int b;
  a = 32;
  b = a;
  a = a + 1;
  System.out.println(b);
  ```

- ```
  Person a;
  Person b;
  a = new Person("Everett");
  b = a;
  a.changeName("Delmar");
  System.out.println( b.getName());
  ```

Primitive types vs. object types



null

- null is a special value in Java.
- All object variables are initialised to null.

- null means that the object variable does not have a reference e.g. str2 below.

```
String str1;
```

```
String str2;
```

null

- null is a special value in Java.
- All object variables are initialised to null.

- You can assign and test for null:

```
private NumberDisplay hours;

if(hours == null) { ... }

hours = null;
```

Multiple constructors

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

Initialises the starting time to 00:00

Initialises the starting time to the user input

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

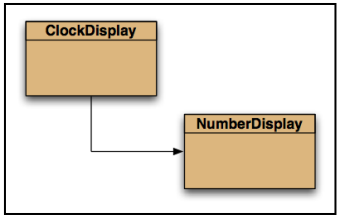
Multiple constructors

- In the ClockDisplay class, we have two constructors.
- Each constructor initialises a clock display in a different way.
- We can have as many constructors as our design requires, ONCE they have unique parameter lists.
- We are overloading our constructor.

Multiple constructors - Overloading

Overloading happens when you have
more than one method of the same name
as long as each has
a distinctive set of parameter types

Internal method calls




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

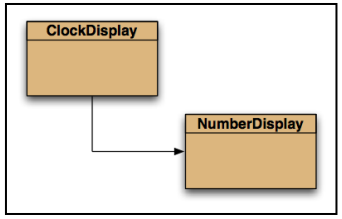
This is an internal method call…

…to this method that exists in the same class, ClockDisplay.

```
/**
 * Update the internal string that represents the display.
 */
private void updateDisplay()
{
    displayString = hours.getDisplayValue() + ":" +
                    minutes.getDisplayValue();
}
```

Internal method calls have the syntax: ***methodname (parameter-list)***

External method calls

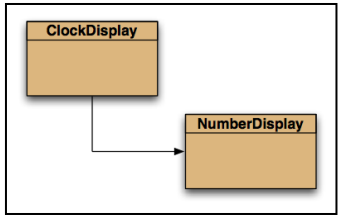



```
public void timeTick()
  {
     minutes.increment();
     if(minutes.getValue() == 0) {  // it just rolled over!
        hours.increment();
     }
     updateDisplay();
  }
```

This timeTick() method is written in the ClockDisplay class.

Each method call is highlighted in red above is an external method call.

External method calls




```
public void timeTick()
  {
     minutes.increment();
     if(minutes.getValue() == 0) {
        hours.increment();
     }
     updateDisplay();
  }
```

ClockDisplay class:

- minutes.increment() is a method call.
- minutes is a NumberDisplay object.
- increment() method is written in the NumberDisplay class.
- minutes.increment(), invokes the increment() method over the minutes object (which is of type NumberDisplay).

Each method call is highlighted in red above is an external method call.

External method calls

- As the increment() method is written in a different class to the call of the method, we call it an external method call.

- A method call to a method of another object is called an external method call.

- External method calls have the syntax:

  object.methodname (parameter-list)

Dot Notation

- Methods can call methods of other objects using dot notation.

- This syntax is known as dot notation:

  object.methodname (parameter-list)

- It consists of:
  - An object
  - A dot
  - A method name
  - The parameters for the method

Questions?

Review

- Divide and conquer; break down problem into parts small enough to solve.
- Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem i.e. the bigger picture.
- Modularization is the process of dividing a large problem into smaller parts.
- Class diagram - shows classes of an application and the relationships between them. It represents a static view of the program.
- Object diagram - shows the objects and their relationships at one moment in time during the execution of an application. It gives information about objects at runtime.

Review

- The 'modulo' operator (%) returns the remainder of an integer division.
- Logic operators operate on boolean values. They produce a new boolean value as a result. The most important ones are: && (and), || (or), ! (not).
- Variables of object types store references to objects.
- null is a special value in Java. All object variables are initialised to null.

Review

- We can have as many constructors as our design requires, ONCE they have unique parameter lists. In this case, we are overloading our constructor.
- Overloading happens when you have more than one method of the same name as long as each has a distinctive set of parameter types.

Review

- When the method is in the same class as the call of the method, we call it an internal method call. Internal method calls have the syntax:

  methodname (parameter-list)

- When a method call is to a method of another object is called an external method call. External method calls have the syntax:

  object.methodname (parameter-list)

- Methods can call methods of other objects using dot notation. This syntax is known as dot notation:

  object.methodname (parameter-list)

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/