# Programación Orientada a Objetos II - Abstracción y modularización

Algoritmos y Estructuras de Datos II

Año 2025

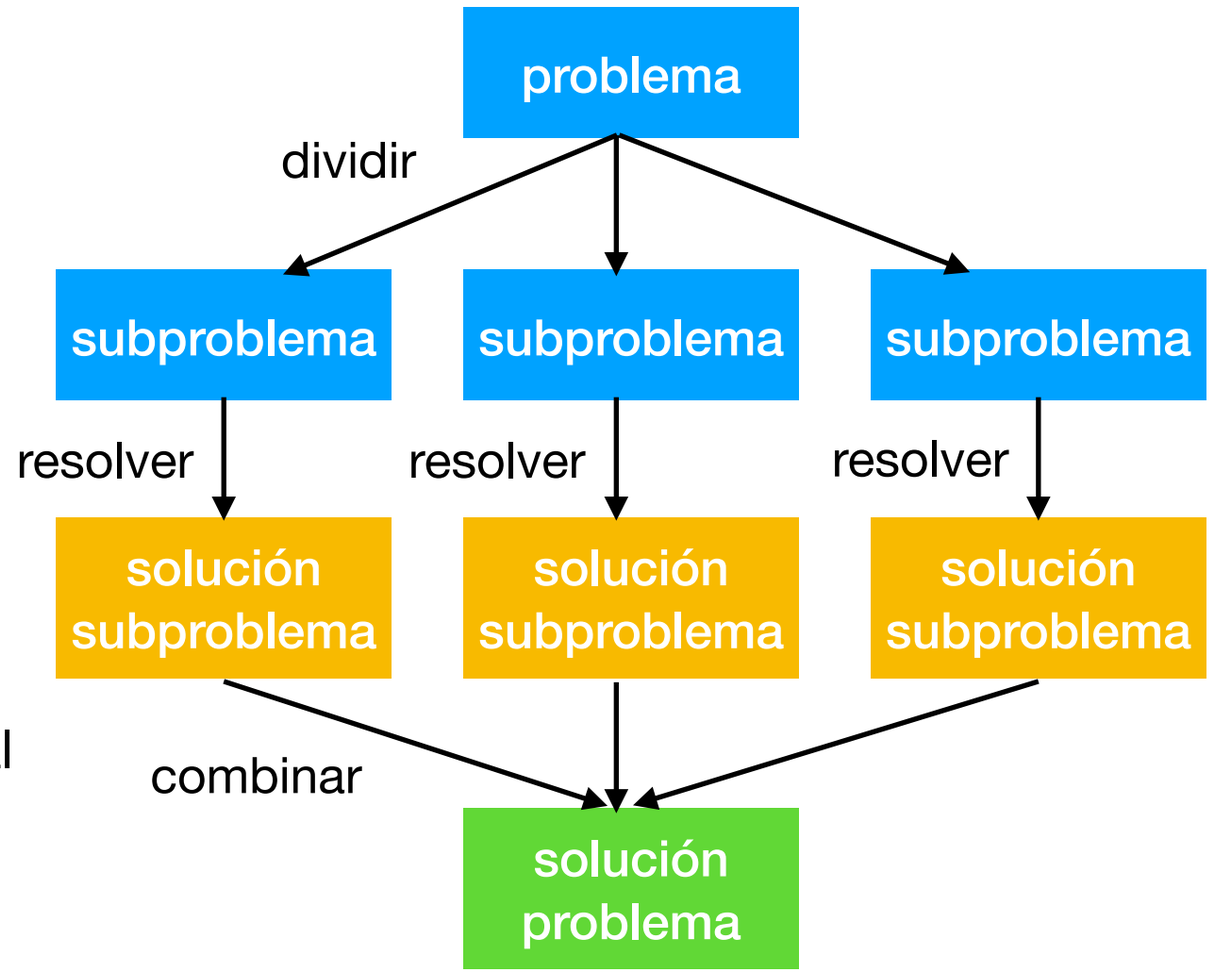
Dr. Pablo Ponzio

Universidad Nacional de Río Cuarto

CONICET

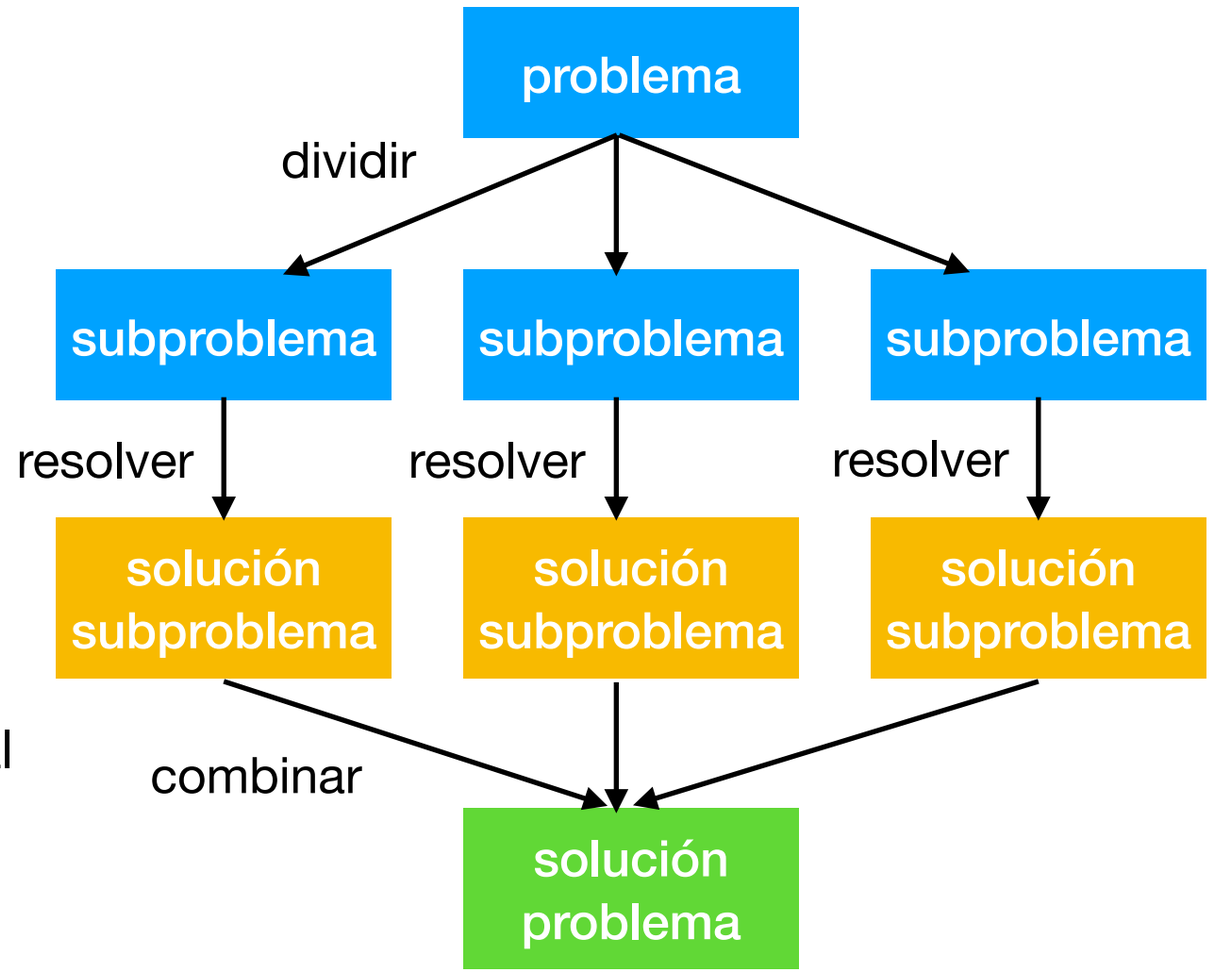

# Divide and conquer

- Cuando los problemas son muy complejos, es muy difícil tener en cuenta todos los detalles relevantes al problema todo el tiempo
- Para lidiar con la complejidad usamos la técnica conocida como divide-and-conquer (divide y reinarás)
- Dividimos el problema en subproblemas más fáciles de resolver que el problema original
- Resolvemos cada subproblema por separado, concentrándonos sólo en los detalles relevantes al subproblema en cuestión
- Finalmente, combinamos las soluciones de los subproblemas se combinan para dar solución al problema original
    - En la combinación, tratamos la solución de cada subproblema como una unidad; nos abstraemos de sus detalles



# Divide and conquer

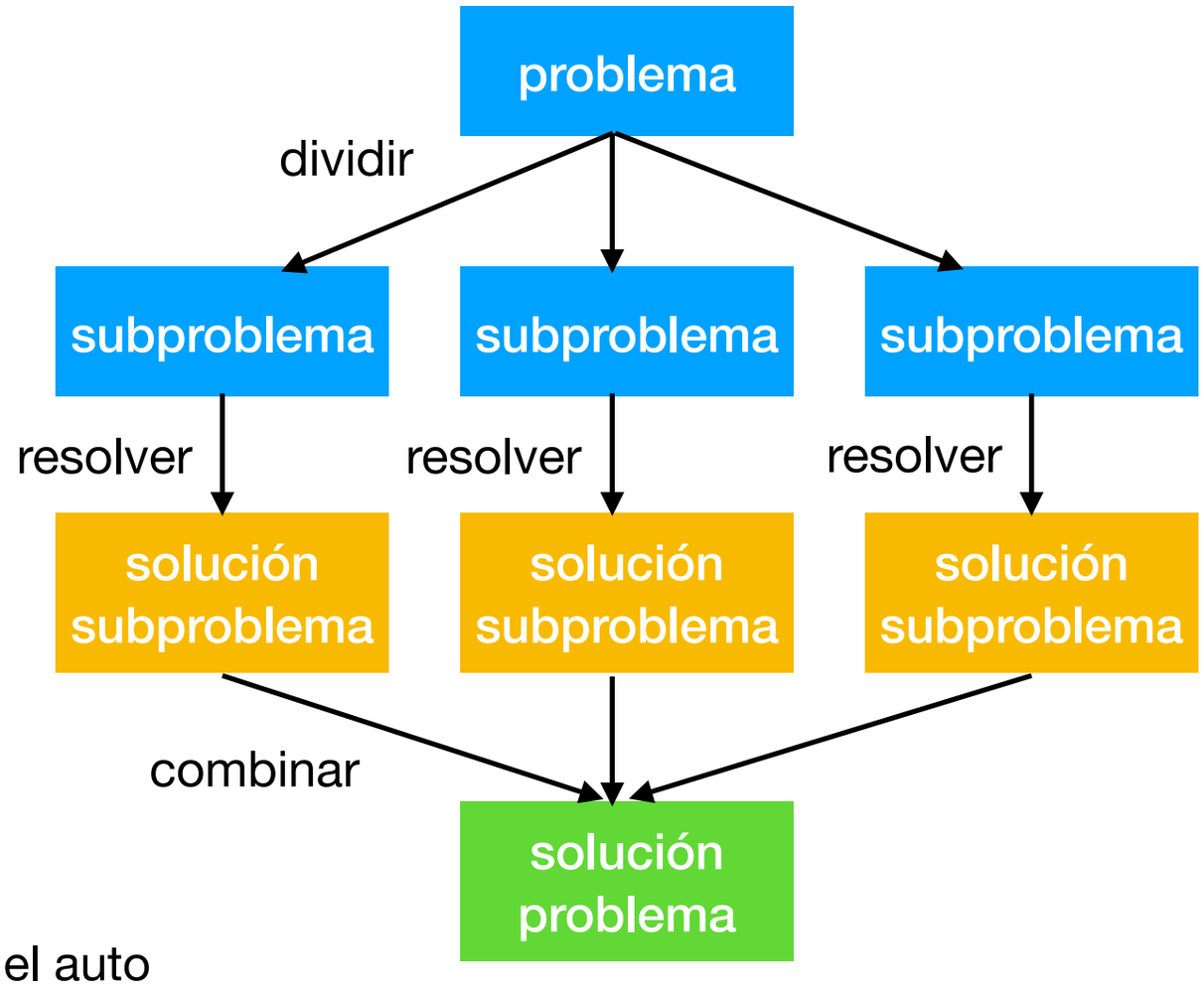
- Cuando los problemas son muy complejos, es muy difícil tener en cuenta todos los detalles relevantes al problema todo el tiempo
- Para lidiar con la complejidad usamos la técnica conocida como divide-and-conquer (divide y reinarás)
- Dividimos el problema en subproblemas más fáciles de resolver que el problema original
- Resolvemos cada subproblema por separado, concentrándonos sólo en los detalles relevantes al subproblema en cuestión
- Finalmente, combinamos las soluciones de los subproblemas se combinan para dar solución al problema original




- Esta técnica puede aplicarse repetidamente: cada subproblema a su vez puede dividirse en subproblemas más pequeños, y estos se resuelven por separado

# Abstracción y Modularización

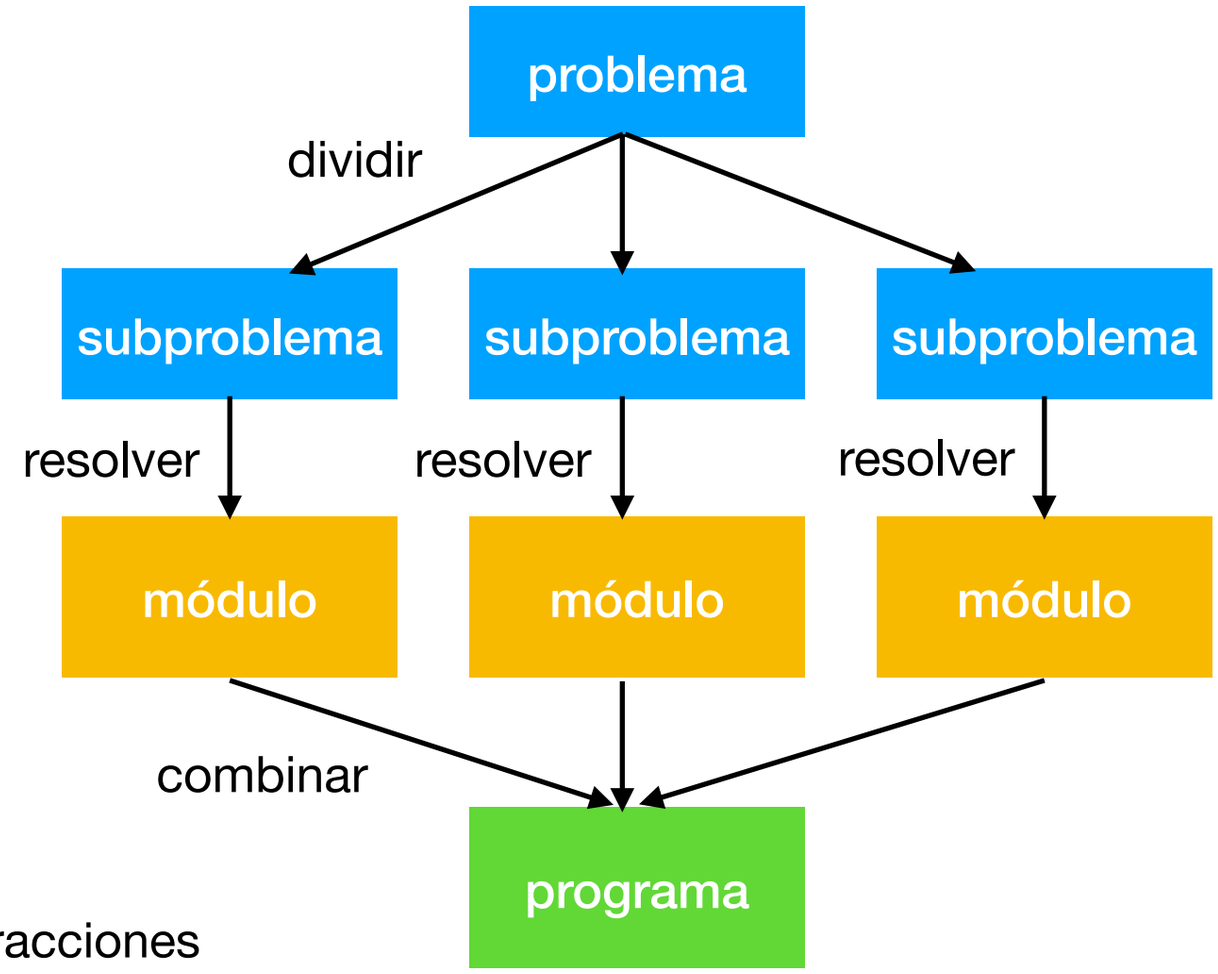
- Ejemplo: Para construir un auto es imposible para una sola persona conocer cada detalle de cada parte del auto
- Los autos se construyen exitosamente porque los ingenieros usan la estrategia de divide-and-conquer, y los principios de modularización y abstracción
- El auto se divide en módulos independientes entre sí: la carrocería, el motor, las ruedas, los asientos, etc.
    - Distintos ingenieros trabajan independientemente en los distintos módulos




- Finalmente, los módulos se combinan para construir el auto
    - En este punto, se usa abstracción: se ve a cada módulo como una unidad, ignorando sus detalles
    - Ej: Cuando ensamblamos el auto nos abstraemos de los detalles de los materiales que se usaron para hacer los neumáticos, el proceso usado para su construcción, etc.

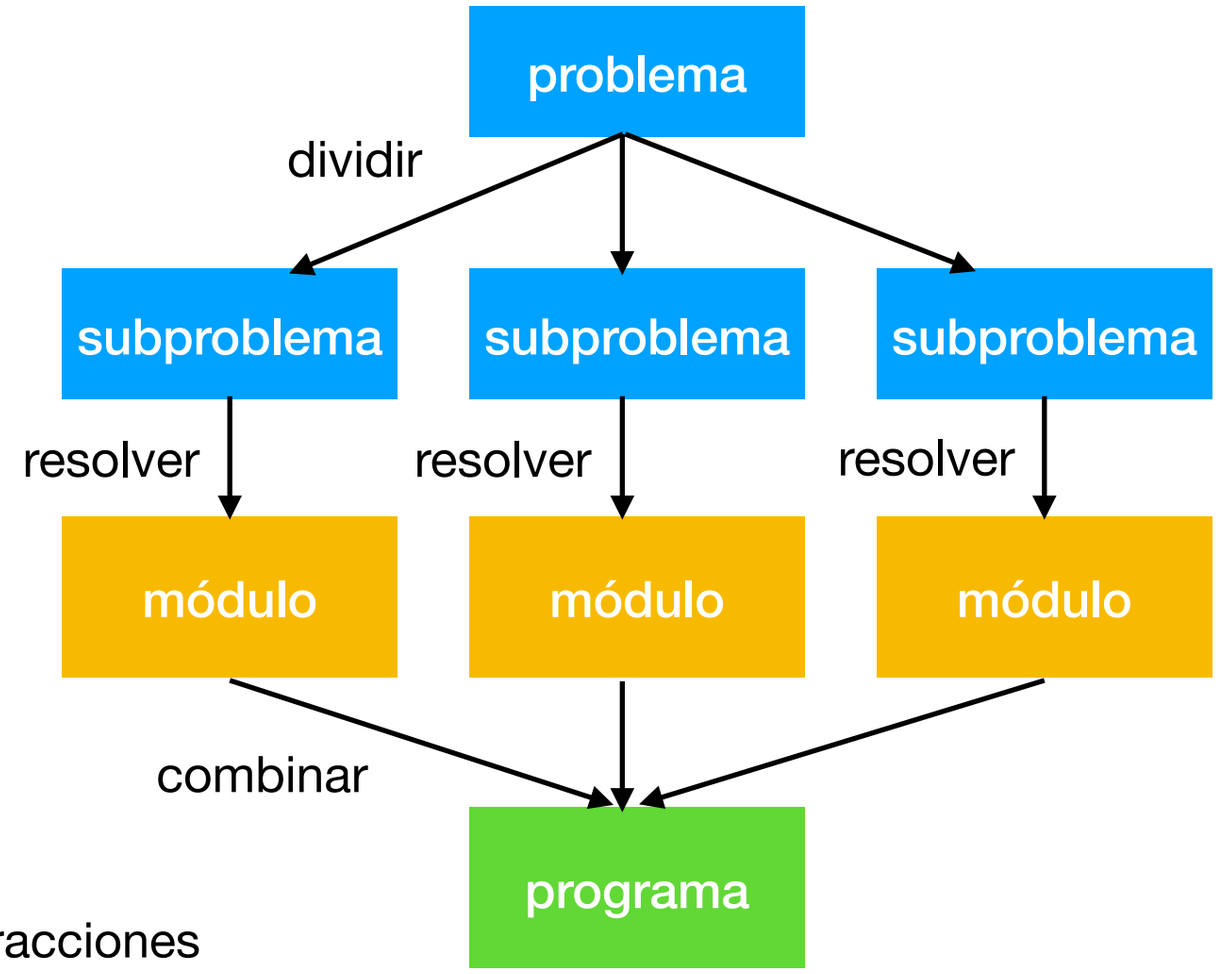
# Abstracción y Modularización

- Modularización: Proceso de dividir un todo en partes bien definidas, que pueden ser construidas y examinadas por separado, y que interactúan de manera bien definida
- Abstracción: es la capacidad de ignorar los detalles de las partes, para centrar la atención en un nivel superior de un problema
- Los principios de modularización y abstracción son cruciales en el desarrollo de software
- Para desarrollar un programa complejo identificamos subcomponentes (módulos) que podemos desarrollar independientemente
- El programa completo estará conformado por las interacciones entre los distintos módulos
  - Usamos los servicios que proveen los módulos, ignorando su complejidad interna (abstracciones de datos)
- En programación orientada a objetos, los módulos están conformados por conjuntos de objetos



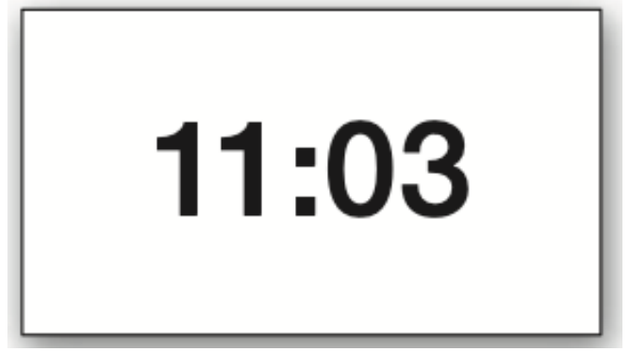
# Abstracción y Modularización

- Modularización: Proceso de dividir un todo en partes bien definidas, que pueden ser construidas y examinadas por separado, y que interactúan de manera bien definida
- Abstracción: es la capacidad de ignorar los detalles de las partes, para centrar la atención en un nivel superior de un problema
- Los principios de modularización y abstracción son cruciales en el desarrollo de software
- Para desarrollar un programa complejo identificamos subcomponentes (módulos) que podemos desarrollar independientemente
- El programa completo estará conformado por las interacciones entre los distintos módulos
- En




**Identificar las clases y objetos que componen (los módulos de) un sistema de software es una tarea crucial para desarrollar software de calidad**

# Ejemplo: ClockDisplay




- Para nuestro primer ejemplo de modularización, vamos a implementar un reloj de 24 horas
  - El display muestra números entre 00:00 y 23:59
- Una forma de implementar el reloj sería representar los cuatro dígitos por separado
- Sin embargo, observemos que los dígitos de las horas y los de los minutos se comportan de manera similar
  - Las horas comienzan en 0, se van incrementando en 1, y cuando llegan a su límite (24) se resetean a 0
  - Los minutos comienzan en 0, se van incrementando en 1, y cuando llegan a su límite (60) se resetean a 0
  - Este comportamiento similar nos indica que podemos ver tanto a los minutos como a las horas como un display de dos dígitos que va desde 0 a un límite dado
    - Notar que esta abstracción también nos sirve si quisiéramos agregar los segundos al reloj

# Ejemplo: ClockDisplay

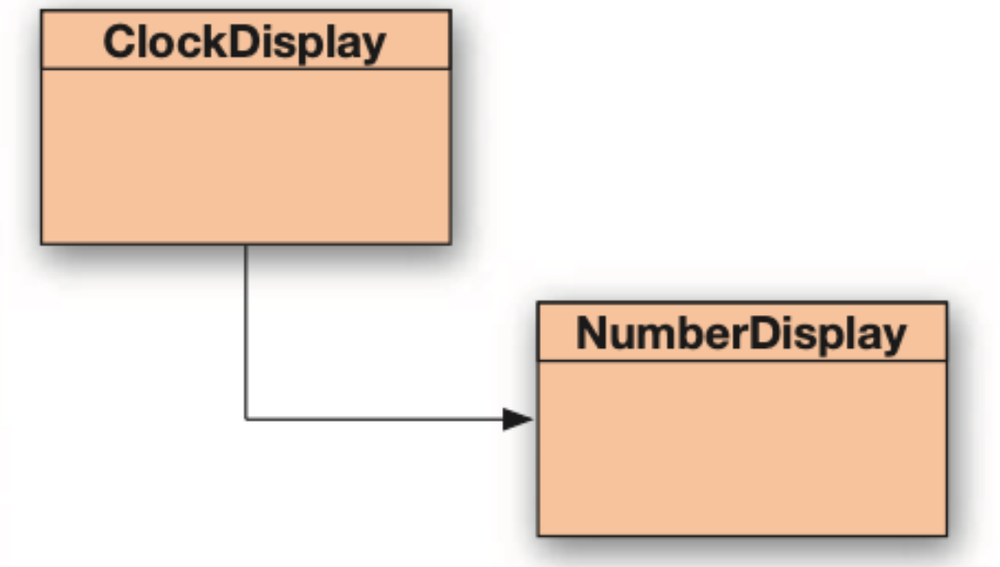
- Primero, vamos a definir nuestro módulo `NumberDisplay`, encargado de modelar displays de dos dígitos
  - En este caso el módulo consiste de una única clase `NumberDisplay`
  - Recordar que:
    - Las clases definen tipos
    - Los elementos del tipo son objetos: las instancias de la clase
    - Podemos definir variables para almacenar referencias a objetos del tipo
- Luego, definimos `ClockDisplay`
  - `ClockDisplay` usa las funcionalidades del módulo `NumberDisplay`
  - Tiene dos atributos de tipo `NumberDisplay`, uno para representar las horas y otro para los minutos
- Inicialmente, las variables que almacenan objetos toman el valor `null` (no hay memoria reservada para un objeto)

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

    Constructor and methods omitted.
}
```

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

    Constructor and methods omitted.
}
```

# Diagrama de clases

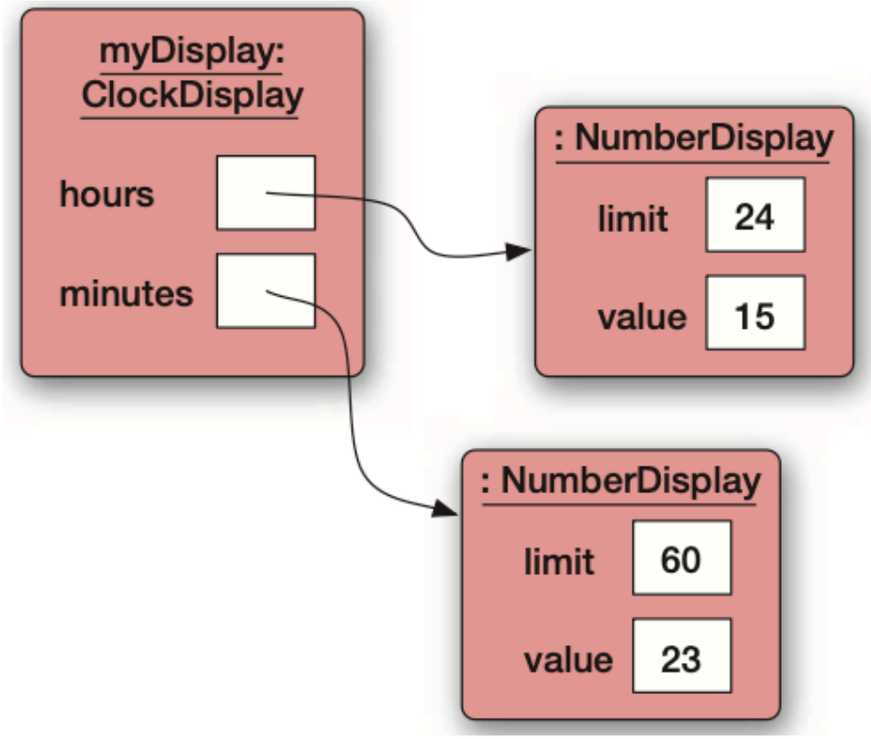
- El diagrama de clases muestra las clases de la aplicación y la relación entre ellas
  - Muestra como está organizado el código de nuestro programa
  - Es decir, provee una vista estática del sistema




- En el diagrama, las clases se grafican con rectángulos
- Las flechas indican relaciones de uso
  - En el ejemplo, el código de la clase `ClockDisplay` usa funcionalidades de `NumberDisplay`
- Para entender un programa orientado a objetos es muy importante conocer el diagrama de clases, y las responsabilidades de las clases (qué función cumple cada clase)

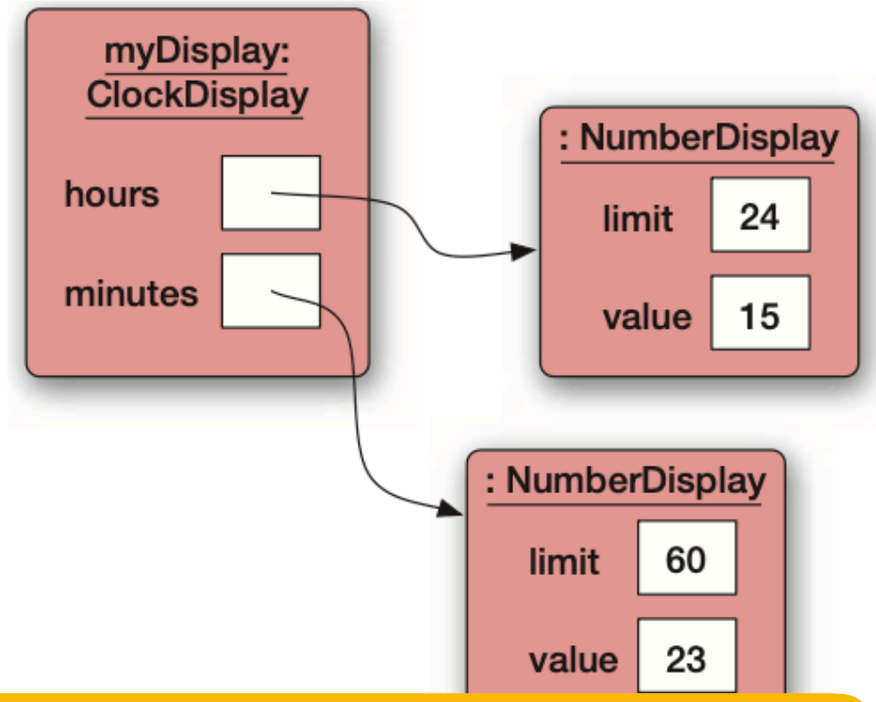
# Diagrama de objetos

- El diagrama de objetos muestra los objetos y las relaciones entre ellos en un momento dado de la ejecución del programa
  - Es decir, el diagrama de objetos muestra una vista dinámica del sistema
- Los diagramas de objetos son una herramienta útil para entender la ejecución de un programa OO
- Los rectángulos del diagrama representan objetos
  - Tienen espacio reservado para cada uno de sus atributos
- Una flecha en el diagrama indica que una variable almacena una referencia a un objeto
  - Ej: `hours` tiene una referencia a un objeto de tipo `NumberDisplay` con el valor `15`



# Diagrama de objetos

- El diagrama de objetos muestra los objetos y las relaciones entre ellos en un momento dado de la ejecución del programa
  - Es decir, el diagrama de objetos muestra una vista dinámica del sistema
- Los diagramas de objetos son una herramienta útil para entender la ejecución de un programa OO
- Los rectángulos del diagrama representan objetos
  - Tienen espacio reservado para cada uno de sus atributos
- Una flecha en el diagrama indica que una variable almacena una referencia a un objeto




**Se recomienda dibujar los diagramas de objetos para entender la ejecución de sus programas**

# NumberDisplay: Funcionalidades

```
/**
 * @pre {@literal 'rollOverLimit' > 0}
 * @post Constructs a NumberDisplay that starts at 0
 * and resets at the given 'rollOverLimit'.
 */
public NumberDisplay(int rollOverLimit)

/**
 * @post Increment the display value by one, rolling
 * over to zero if the limit is reached.
 */
public void increment()

/**
 * @post Return the display value, that is, the current value
 * as a two-digit String.
 */
public String getDisplayValue()

/**
 * @pre {@literal 'replacementValue' >= 0 and 'replacementValue' < 'limit'.}
 * @post Set the value of the display to the new specified value.
 */
public void setValue(int replacementValue)
```

# NumberDisplay: Implementación

```
/**
 * @pre {@literal 'rollOverLimit' > 0}
 * @post Constructs a NumberDisplay that starts at 0
 * and resets at the given 'rollOverLimit'.
 */
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;
    value = 0;
}
```

# NumberDisplay: Implementación

```
/**
 * @pre {@literal 'rollOverLimit' > 0}
 * @post Constructs a NumberDisplay that starts at 0
 * and resets at the given 'rollOverLimit'.
 /**
p  * @post Increment the display value by one, rolling over to zero if the
{  * limit is reached.
   */
  public void increment()
} {
      value = (value + 1) % limit;
  }
```

# NumberDisplay: Implementación

```
/**
 * @pre {@literal 'rollOverLimit' > 0}
 * @post Constructs a NumberDisplay that starts at 0
 * and resets at the given 'rollOverLimit'.
 */
p
{
  p
} {
  }
```

```
 * @post Increment the display value by one, rolling over to zero if the
 * limit is reached.
 */
```

```
 * @post Return the display value, that is, the current value
 * as a two-digit String.
 */
public String getDisplayValue()
{
    if(value < 10)
        return "0" + value;
    else
        return "" + value;
}
```

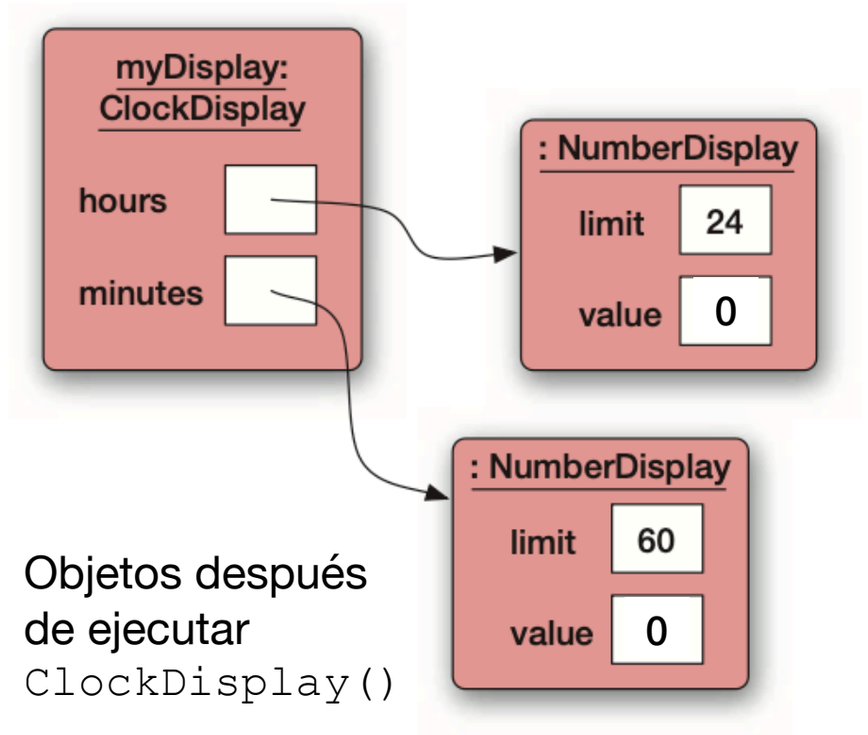
# NumberDisplay: Implementación

```
/**
 * @pre {@literal 'rollOverLimit' > 0}
 * @post Constructs a NumberDisplay that starts at 0
 * and resets at the given 'rollOverLimit'.
 */
p
{
p
} {
}
p
{
public void setValue(int replacementValue)
{
}
```

```
 * @post Increment the display value by one, rolling over to zero if the
 * limit is reached.
 */
/**
 * @post Return the display value, that is, the current value
 * as a two-digit String.
 */
/**
 * @pre {@literal 'replacementValue' >= 0 and 'replacementValue' < 'limit'.}
 * @post Set the value of the display to the new specified value.
 */
    if((replacementValue >= 0) && (replacementValue < limit)) {
        value = replacementValue;
    }
```

# Objetos que crean objetos

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

    /**
     * @post Creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
```




Objetos después de ejecutar `ClockDisplay()`

- Estamos listos para definir nuestro `ClockDisplay`, usando funcionalidades de `NumberDisplay`
- `ClockDisplay` tiene dos atributos de tipo `NumberDisplay`
  - Inicialmente, estos atributos son `null`
  - Para que estos atributos hagan referencia a objetos tenemos que crearlos primero
- Creamos los objetos directamente en el constructor de `ClockDisplay`
  - Notar que los objetos de tipo ClockDisplay usan objetos de tipo `NumberDisplay`

# Invocación a métodos internos

- Para mantener el ejemplo simple, vamos a representar el valor del display con un atributo de tipo `String` (`displayString`)
- La última línea del constructor invoca a un método privado de la misma clase: `updateDisplay()`
  - Esto se llama una invocación a un método interno
- La invocación a un método interno no requiere que especifiquemos el objeto al cual se aplica
  - El método se ejecuta sobre el objeto actual
- Los métodos internos son por lo general privados (`private`)
  - Sólo pueden ser invocados desde la clase que los define
  - Definen funcionalidades útiles sólo para la clase

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

    /**
     * @post Creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

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

# Invocación a métodos externos

- El operador `.` de Java puede usarse para invocar a métodos de otros objetos
- Por ejemplo, el código de `timeTick()` invoca a:
  - `minutes.increment()`
  - `minutes.getValue()`
  - `hours.increment()`

```
/**
 * @post Makes the clock display go one minute forward
 * (should get called once every minute).
 */
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {
        // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}
```

- Las llamadas a métodos de otros objetos se denominan invocaciones a métodos externos
- Notar que la responsabilidad de llevar el tiempo está dividida entre las clases `NumberDisplay` y `ClockDisplay`
  - Esto es un ejemplo de divide and conquer
- `timeTick()` incrementa en 1 los minutos (`minutes.increment()`), y cada vez que vuelven a 0 incrementa e 1 la hora (`hours.increment()`)

# Múltiples constructores

- Notar que una clase puede tener más de un constructor
- En este ejemplo, tiene uno que inicializa el display por defecto (en 00:00), y otro que lo inicializa en un tiempo dado como parámetro
- Java soporta la sobrecarga de operadores: puede haber más de un constructor, o más de un método con el mismo nombre, siempre que tengan distintos  parámetros

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

    /**
     * @post Creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    /**
     * @pre {@literal 0 <= 'hour' < 24 and 0 <= 'minu
     * @post Creates a new clock set at the time spec
     * parameters.
     */
    public ClockDisplay(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }
```

# ClockDisplay: implementación

- Finalmente, proveemos métodos de `ClockDisplay` para cambiar el tiempo del reloj, y para retornar el valor del display

```
/**
 * @pre {@literal 0 <= 'hour' < 24 and 0 <= 'minute' < 60.}
 * @post Set the time of the display to the specified hour and
 * minute.
 */
public void setTime(int hour, int minute)
{
    hours.setValue(hour);
    minutes.setValue(minute);
    updateDisplay();
}

/**
 * @post Returns the current time of this display in the format HH:MM.
 */
public String getTime()
{
    return displayString;
}
```

# ClockDisplay: Resumen

- En este ejemplo usamos modularización para dividir el problema en partes más simples: definimos un módulo `NumberDisplay`, y luego lo utilizamos para definir `ClockDisplay`
- Para usar `NumberDisplay` empleamos la técnica de abstracción, e ignoramos sus detalles de implementación
- Sólo nos basamos en los métodos y sus especificaciones
- Invocamos a `increment()` y asumimos que va a incrementar el display correctamente (y lo va a resetear cuando llegue a su límite)
- Invocamos a `getValue()` para obtener el valor del display

```
/**
 * @post Makes the clock display go one minute forward
 * (should get called once every minute).
 */
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {
        // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}
```

- Notar como trabajamos a un nivel más abstracto
- Recordar que definimos a `NumberDisplay` como una abstracción de datos

# ClockDisplay: Resumen

- En este ejemplo usamos modularización para dividir el problema en partes más simples: definimos un módulo `NumberDisplay`, y luego lo utilizamos para definir `ClockDisplay`
- Para usar `NumberDisplay` empleamos la técnica de abstracción, e ignoramos sus detalles de implementación
  - Sólo nos basamos en los métodos y sus especificaciones
  - Invocamos a `increment()` y asumimos que va a incrementar el display correctamente (y lo va a resetear cuando lleg
  - Invo
    obte
- Notar como trabajamos a un nivel más abstracto
- Recordar que definimos a `NumberDisplay` como una abstracción de datos

```
/**
 * @post Makes the clock display go one minute forward
 * (should get called once every minute).
 */
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {
        // it just rolled over!
        hours.increment();
```

**Estas abstracciones simplifican la implementación, y hace que el código sea más fácil de comprender**

# ClockDisplay: Resumen

- En este ejemplo usamos modularización para dividir el problema en partes más simples: definimos un módulo `NumberDisplay`, y luego lo utilizamos para definir `ClockDisplay`
- Para usar `NumberDisplay` empleamos la técnica de abstracción, e ignoramos sus detalles de implementación
  - Sólo nos basamos en los métodos y sus especificaciones
  - Invocamos a `increment()` y asumimos que va a incrementar el display correctamente (y lo va a resetear cuando lleg
  - Invo
    obte
  - Not
  - Rec

```
/**
 * @post Makes the clock display go one minute forward
 * (should get called once every minute).
 */
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {
        // it just rolled over!
        hours.increment();
```

**Estas abstracciones simplifican la implementación, y hace que el código sea más fácil de comprender**

**También pueden facilitar la extensión del código, por ejemplo, es muy simple agregar otro display que lleve los segundos**

# Interfaz de una clase o módulo

- En proyectos del mundo real distintas personas suelen escribir el código de distintos módulos
- Para que esto sea posible, los desarrolladores tienen que ponerse de acuerdo en las funcionalidades que proveerá cada módulo
- De esta manera, un grupo de desarrolladores puede concentrarse en implementar un módulo, que los restantes desarrolladores pueden luego usar para implementar otros módulos
- La interfaz de una clase/módulo define las funcionalidades que provee la clase/módulo
  - La interfaz de la clase/módulo está conformada por sus métodos públicos
  - Por ejemplo, la interfaz de `ClockDisplay` es:

**Constructor Summary**

| Constructors |
| --- |
| **Constructor** |
| `ClockDisplay()` |
| `ClockDisplay(int hour, int minute)` |

**Method Summary**

All Methods | Instance Methods | Concrete Metho

| Modifier and Type | Method |
| --- | --- |
| `String` | `getTime()` |
| `void` | `setTime(int hour, int minute)` |
| `void` | `timeTick()` |

- Hablaremos mucho más sobre las interfaces más adelante en el curso...

# Actividades

- Leer el capítulo 3 del libro "*Objects First with Java A Practical Introduction using BlueJ*". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016

- Leer el capítulo 5 del libro "*Program Development in Java - Abstraction, Specification, and Object-Oriented Design*". B. Liskov & J. Guttag. Addison-Wesley. 2001

# Bibliografía


- "*Objects First with Java A Practical Introduction using BlueJ*". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016

- "*Program Development in Java - Abstraction, Specification, and Object-Oriented Design*". B. Liskov & J. Guttag. Addison-Wesley. 2001