Object interactions
Lecture 6

Waterford Institute of Technology

January 26, 2016

John Fitzgerald

Presentation outline

Estimated duration presentation

Questions at end presentation

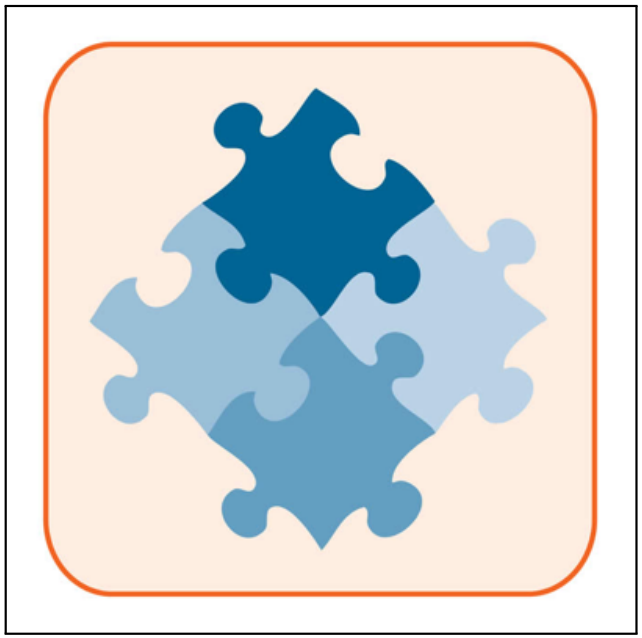
Topics discussed:

- Revisit abstraction & modularization
  - BlueJ Clock
- **this** keyword
- Logical operations
- Modular (clock) arithmetic
- The **this** keyword
- Strings
- Primitive type conversion

Abstraction revisited

BlueJ Clock

- demonstrates **abstraction**
- Application decomposed into modules
- Implementation details hidden
- Public interface only exposed
- Development separable by both
  - Location
  - Time

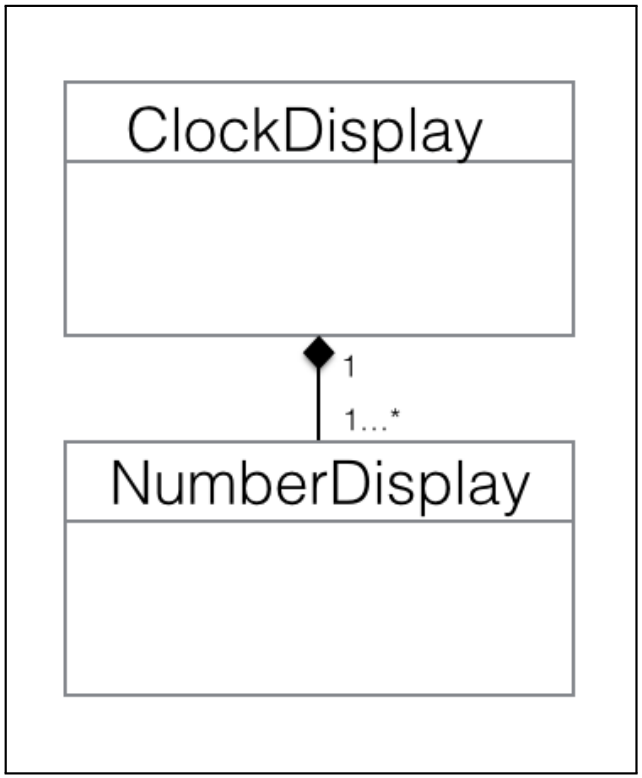


Class diagram

BlueJ Clock Example

Class diagram

- *Static* view
- Represents class design
- Arrow means *ClockDisplay* **has a** *NumberDisplay*
- One ClockDisplay has one or many NumberDisplay fields

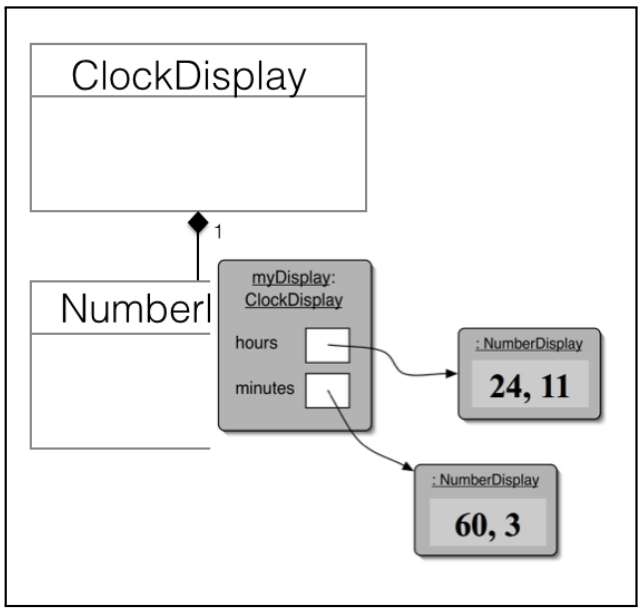


Object composition

BlueJ ClockDisplay

Both Object & Class diagrams reveal

- ClockDisplay has 2 NumberDisplay fields
- ClockDisplay exclusive owner fields
- *Object Composition* example



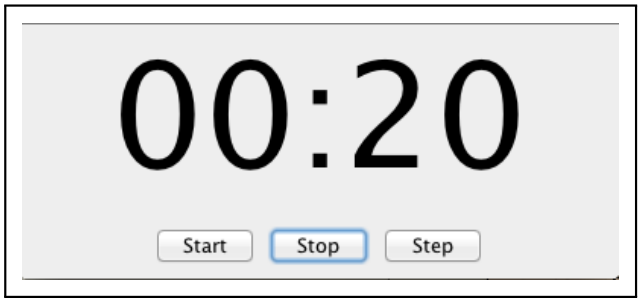
NumberDisplay class

NumberDisplay

- Represents digital number display
- Fields
  - int `limit`
  - int `value`
- *value* range 0 to *limit-1*
- *value* resumes at 0 when limit reached
- *modular* or *clock* arithmetic

```
public class NumberDisplay
{
    private int limit;
    private int value;
    public NumberDisplay(int limit)
    {
        this.limit = limit;
        value = 0;
    }
}
```

NumberDisplay class



NumberDisplay

- Updating *value* attribute
- Accepts parameter only if in valid range
- Otherwise no action

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

Object reference

this keyword

this: a reference to the current object.

```
public class Circle {
  public Circle() {
    System.out.println("this : " + this);
  }
}
```

```
public class TestCircle {
  public TestCircle() {
    Circle circle = new Circle();
    System.out.println("circle : " + circle);
  }
}
```

```
this : Circle@2f67d81
circle : Circle@2f67d81
```

Conditional and Unary Operators

Conditional operators

- Logical **AND** &&
- Logical **OR** ||

Unary operator

- Logical complement !
- Also called **NOT** operator
- Inverts value of boolean

```
boolean a = false;
boolean b = true;
/* !a is true
 * !b is false
 * a && b is false
 * a || b is true
*/
```

Truth table for logical operations

Table 1 : Using **booleans**

a	b	a&&b	a||b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

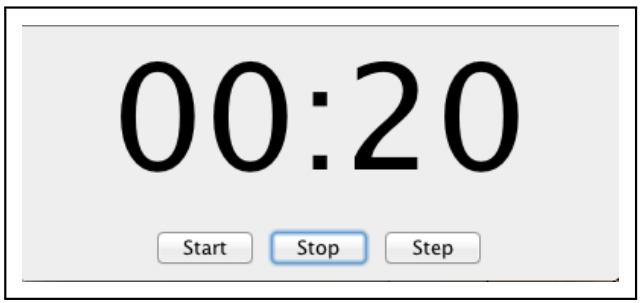
Table 2 : Alternative representation

a	b	a&&b	a||b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

NumberDisplay class

NumberDisplay

- Return time to display
- String concatenation used
- Leading zero inserted

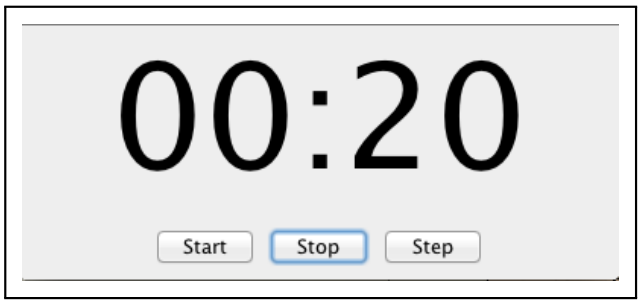



```
public String getDisplayValue()
    {
    if(value < 10) {
        return "0" + value;
    }
    else {
        return "" + value;
    }
}
```

NumberDisplay class

NumberDisplay

- Increment time
- Use modulus operator %
- Forces a roll-over to zero at limit

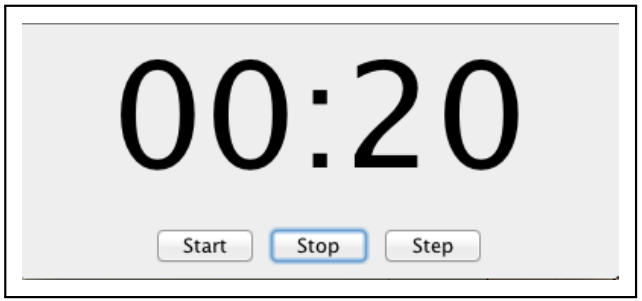



```
public void increment()
{
    value = (value + 1) % limit;
}
```

NumberDisplay class

NumberDisplay

- Increment time
- Use modulus operator %
- Forces a roll-over to zero at limit



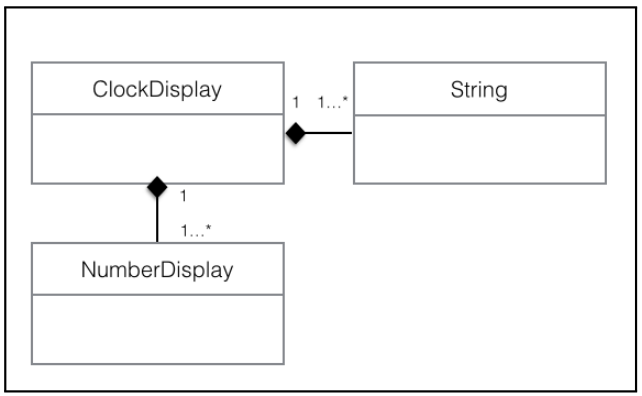
Time Start	Time +1	Check limit	End Time
57	58	58%60	58
58	59	59%60	59
59	60	60%60	00
00	01	01%60	01

Table 3 : Increment clock minutes display

ClockDisplay class

ClockDisplay composition
Has fields

- *NumberDisplay*
- *String*




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

ClockDisplay class

ClockDisplay instantiation

Default constructor

- Two new *NumberDisplay* objects
  - Minute display
  - Hour display
- Initializes *limit* attributes
- Sets clock display 00.00

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

ClockDisplay class

ClockDisplay instantiation

Overloaded constructor

- Two new *NumberDisplay* objects
  - Minute display
  - Hour display
- Initializes *limit* attributes
- Updates clock display

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

ClockDisplay class

Simulate ticking clock

- Advance one minute
- Check minute elapsed
- Then increment hour

```
public void timeTick()
{
minutes.increment();
//if minute display zero
//time to advance an hour
if(minutes.getValue() == 0) {
    hours.increment();
}
updateDisplay(); }
```

ClockDisplay class

Update clock display

- Get hour value
- Get minute value
- Concatenate values
- Example display 00:20

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

Strings

String class

```
String greeting = "Hello ICTSkills group: the future is bright";
```

- Java String class widely used
- Java *String*s are objects
  - Comprise series of characters
- Direct creation
  - `String s = "this is a string"`
  - "this is a string" is a String literal
- Creation using *new* operator
  - `String s = new String("this is a string")`

Strings

String methods

Extensive set of methods available

- String length
  - `String s = "this is a string";`
  - `int length = s.length;`
    - length is 16
- String concatenation method 1
  - `String s1 = "Hello ";`
  - `String s2 = "ICTSkills group";`
  - `String s3 = s1.concat(s2);`
    - s3 : *Hello ICTSkills group*
- String concatenation method 2
  - `String s1 = "Hello ";`
  - `String s2 = "ICTSkills group";`
  - `String s3 = s1 + s2;`
    - Result: *Hello ICTSkills group*

Strings

Converting String to number

- String s = "100.45";
  - double d = Double.parseDouble(s);
  - System.out.println("d is "+ d);
    - Output: d is 100.45
- String s = "100";
  - int number = Integer.parseInt(s);
  - System.out.println("number is "+ number);
    - Output: number is 100

Two members of the **Number** family:

- **Integer** class contains a single field whose type is *int*.
- **Double** class contains a single field whose type is *double*.

Strings

Converting number to String

Method 1 : Concatenate with empty string

- ```
  int num = 100;
  String s1 = "" + num;
  ```

Method 2: Invoke String method *valueOf*

- ```
  int num = 100;
  String s1 = String.valueOf(num);
  ```

Method 3: Use one of *Number* family

- ```
  int num = 100;
  double dnum = 100.35;
  String s1 = Integer.toString(num);
  String s2 = Double.toString(dnum);
  ```

Other *Number* family members are Byte, Float, Long, Short
  ```

Strings

Comparing String objects

Using *String equals* and *equalsIgnoreCase* methods

- ```
  String s0 = "ICTSkills Group";
  String s1 = "ICTSkills group";
  boolean b1 = s0.equals(s1);
  boolean b2 = s0.equalsIgnoreCase(s1);
  ```

- b1 is *false*
- b2 is *true*

Some other methods are

```
boolean endsWidth(String suffix); //true if string ends with suffix
boolean startsWith(String prefix) //true if string begins with prefix
```
```

Primitive type conversion

Explicit conversion & explicit cast

Table 4 : **Explicit conversion**

Expression	Type	Value
Math.round(3.14)	long	3
Math.round(2.71)	long	3

Table 5 : **Explicit cast**

Expression	Type	Value
(int)Math.round(3.14)	int	3
(int)3.14	int	3
(int)2.71	int	2

Primitive type conversion

Automatic promotion - no loss of data

Java automatically converts data to type with larger range

- No need for explicit cast

```
double val = 0.3*11; //11 transparently promoted to 11.0
```

```
int b = 10;
int c = 10;
int d = b*b - 4*c;
double e = b*b - 4.0*c;
```

Primitive type conversion

Explicit cast required where loss of data

```
public class Circle {
  double radius;
  ...
  //this will not compile
  public int getRadius() {return radius;}
}
```

```
public class Circle {
  double radius;
  ...
  //this will compile
  public int getRadius() {return (int)radius;}
}
```

Summary

- Abstraction & modularization
  - Hide details
  - Logical partitioning
- **this** keyword with examples.
- Logical operations
  - Boolean operations
- Modular arithmetic
  - Fundamental branch of mathematics
  - Important applications in IT
- The **this** keyword
- Strings
  - Facilitates manipulation of character sets (text).
- Primitive type conversion
  - Implicit conversion
  - Casting

Referenced material

1. Sedgewick R. Wayne K. Introduction fo Programming in Java Precedence and associativity of Java operators `http://introcs.cs.princeton.edu/java/11precedence/` [Accessed 2014-02-12]
2. Java 7 String API `http://docs.oracle.com/javase/7/docs/api/java/lang/String.html?is-external=true` [Accessed 2014-31-4]
3. Class Double `http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html` [Accessed 2014-04-01]
4. Integer Class `http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html?is-external=true` [Accessed 2014-04-01]
5. Comparing Strings `http://docs.oracle.com/javase/tutorial/java/data/comparestrings.html` [Accessed 2014-04-01]