OBJECT INTERACTION

CITS1001
Overview

- Coupling and Cohesion
- Internal/external method calls
- null objects
- Chaining method calls
- Class constants
- Class variables
A digital clock

11:03
Abstraction and modularization

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

• **Modularization** 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.
Modularizing the clock display

One four-digit display?

Or two two-digit displays?
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.
}
Classes define types

- New concept: classes define types

private NumberDisplay hours;

- A class name can be used as the type for a variable. Variables that have a class as their type can store objects of that class.
Dynamic view at *runtime* (when the system is running)
Object diagram

- The **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 and presents the **dynamic** view of a program.
Class diagram

ClockDisplay \textit{depends on} NumberDisplay

ClockDisplay \textit{makes use of} NumberDisplay
Class diagram

• The **class diagram** shows the classes of an application and the relationships between them.

• It gives information about the source code and presents the **static** view of a program.
Objects creating objects

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

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        ...
    }
}
```
Objects creating objects

in class ClockDisplay:

    hours = new NumberDisplay(24);

    \textit{actual parameter}

in class NumberDisplay:

    public NumberDisplay(int rollOverLimit);

    \textit{formal parameter}
Method calling

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

Exerise: modify this method to include a day of week NumberDisplay (e.g. 0=Mon to 6=Sun). What test cases should we to use?
External method call

- external method calls
  
  `minutes.increment();`

  `object . methodName ( parameter-list )`
Internal method call

• internal method calls

    `updateDisplay() ;`

• No variable name is required.

• `this`
  • could be used as a reference to the invoking object, but not used for method calls.
Internal method

/**
 * Update the internal string that
 * represents the display.
 */

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

• NB: A method call on another object of the same type would be an external call.
• ‘Internal’ means ‘this object’.
• ‘External’ means ‘any other object’, regardless of its type.
null

• The Java reserved word null is used to mean “no object” when an object variable is not currently referring to a particular object.

• A field that has not explicitly been initialised will contain the value null by default.
null

- **null** is a special value in Java
- Object fields are initialized to **null** by default.
- You can test for and assign **null**:

```java
private NumberDisplay hours;

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

hours = null;
```
Anonymous objects

- Objects are often created and handed on elsewhere immediately:

  ```java
  Lot furtherLot = new Lot(...);
  lots.add(furtherLot);
  ```

- We don’t really need `furtherLot`:

  ```java
  lots.add(new Lot(...));
  ```
Chaining method calls

• Methods often return objects.
• We often immediately call a method on the returned object.
  
```java
Bid bid = lot.getHighestBid();
Person bidder = bid.getBidder();
```
• We can use the anonymous object concept and *chain* method calls:
  
```java
lot.getHighestBid().getBidder()
```
Chaining method calls

- Each method in the chain is called on the object returned from the previous method call in the chain.

```java
String name = lot.getHighestBid().getBidder().getName();
```

- Returns a **Bid** object from the **Lot**
- Returns a **Person** object from the **Bid**
- Returns a **String** object from the **Person**
CLASS AND CONSTANT VARIABLES
Class variables

- A class variable is shared between all instances of the class.
- In fact, it belongs to the class and exists independent of any instances.
- Designated by the `static` keyword.
- Public static variables are accessed via the class name; e.g.:
  - `Thermometer.boilingPoint`
Class variables

```
BouncingBall
  gravity: 3

ball1: BouncingBall
  xPosition: 10
  yPosition: 233

ball2: BouncingBall
  xPosition: 76
  yPosition: 155

ball3: BouncingBall
  xPosition: 782
  yPosition: 33
```
Constants

• A variable, once set, can have its value fixed.
• Designated by the `final` keyword.
  • `final int max = list.size();`
• Final `fields must` be set in their declaration or the constructor.
• Combing `static` and `final` is common.
Class constants

- **static**: class variable
- **final**: constant

```java
private static final int gravity = 3;
```

- Public visibility is less of an issue with **final** fields.
- Upper-case names often used for class constants:

```java
public static final int BOILING_POINT = 100;
public static final int WINDOW_WIDTH = 400;
```

Best Practice: Define constants instead of scattering “magic numbers” through your code.

Makes code easier to read and avoids update errors when changes are made.