In this lecture, we look at more advanced concepts relating to objects, classes, and the way objects interact.
Fundamental concepts

- Coupling and cohesion
- Internal/external method calls
- null objects
- Chaining method calls
- Class constants
- Class variables

Reading: Chapter 3 of *Objects First with Java – A Practical Introduction using BlueJ*, © David J. Barnes, Michael Kölling
Modelling a clock
A digital clock

11:03
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.
Abstraction

- Abstraction is the ability to ignore details of the parts of a problem, to focus attention on its higher levels
Modularizing the clock display

One 4-digit display?
Or two 2-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.
    // ...
}

Objects in the running program

- Dynamic view at runtime (when the system is running)
Objects exist at run-time

An *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 depends on NumberDisplay
ClockDisplay makes use of NumberDisplay
Classes define types

```java
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 belonging to that class
Class diagram (2)

- Classes exist at compile time
- 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
Classes as types
Primitive types vs. object types

SomeObject obj;

object type

int i;

primitive type
Primitive types vs. object types

```
ObjectType a;
int a;
ObjectType b;
int b;
b = a;
```

Quiz: What is the output?

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

```java
Person a;
Person b;
a = new Person("Everett");
b = a;
a.changeName("Delmar");
System.out.println(b.getName());
```
Interlude – some useful operators for building our clock
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.

- E.g., generally:
  \[17 \div 5 \text{ gives result } 3, \text{ remainder } 2\]

  In Java:
  \[17 \div 5 == 3\]
  \[17 \mod 5 == 2\]
Quiz

- What is the result of the expression
  \[8 \mod 3\]
- For integer \(n \geq 0\), what are all possible results of:
  \(n \mod 5\)
- Can \(n\) be negative?
Modelling a clock

Classes as types

Interlude – some useful operators for building our clock

Back to the clock
public NumberDisplay(int rollOverLimit) {
    limit = rollOverLimit;
    value = 0;
}

public void increment() {
    value = (value + 1) % limit;
}
Consider the constructor for the ClockDisplay class:

```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:
  ```java
  hours = new NumberDisplay(24);
  ```
  (actual parameter)

- In class NumberDisplay:
  ```java
  public NumberDisplay(int rollOverLimit);
  ```
  (formal parameter)
ClockDisplay object diagram

myDisplay: ClockDisplay

hours

minutes

: NumberDisplay

limit 24

value 15

: NumberDisplay

limit 60

value 23
public void timeTick() {
    minutes.increment();
    if (minutes.getValue() == 0) {
        // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}
External method calls

- For calling a method on another object
- external method call example:
  
  ```java
  minutes.increment();
  
  where signature of increment is:
  
  ```java
  public void increment()
  ```

- general form is:

  ```java
  object . methodName ( parameter-list )
  ```

- If increment() had been a private method we would not have been able to invoke it.
Internal method calls

- For calling a method on our own object.
- Why would we want to do that?
Internal method calls (2)

- internal method call example:
  ```java
  updateDisplay();
  ```
- No variable name is required.
The updateDisplay method of ClockDisplay:

```java
/**
 * 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’, ‘ourselves’.
- ‘External’ means ‘any other object’, regardless of its type.
null

- null is a special Object in Java
- All Object variables (of any class) are initially null
- Variables can be tested for whether they are null
  
  ```java
  private NumberDisplay hours;
  if (hours != null) {
    //... nothing to show
  } else {
    //... display the hours
  }
  
  ```

- Variables can be given the value null - losing the reference to anything they were previously holding.
  
  ```java
  public void forgetHours() {
    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.
  
  Bid bid = lot.getHighestBid();
  Person bidder = bid.getBidder();

- We can use the anonymous object concept and *chain* method calls:

  lot.getHighestBid().getBidder()
Chaining method calls (2)

- 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`
Concept summary

- object creation
- overloading
- internal/external method calls
- debugger
Review (1)

- Abstraction
  - ignore some details to focus attention on a higher level of a problem
- Modularisation
  - Divide a whole into well defined parts that can be built separately and that interact in well-defined ways
- Classes define types
  - 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.
Review (2)

- **Object diagram**
  - Shows the objects and their relationships at one moment during the execution of an application

- **Object references**
  - Variables of object types store references to objects

- **Primitive type**
  - The primitive types of Java are non-object types. The most common primitive types are int, boolean, char, double and long.

- **Object creation**
  - Objects can create other objects using the new operator
Review (3)

- **Internal method call**
  - Methods can call other methods of the same class.

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