CITS1001 week 3
Object interaction

Arran Stewart

March 12, 2018

Overview

- 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

Modularization

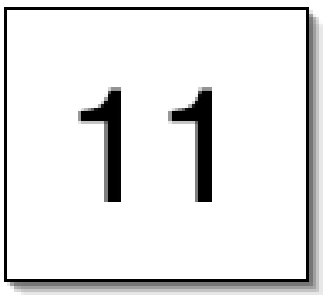
- Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways

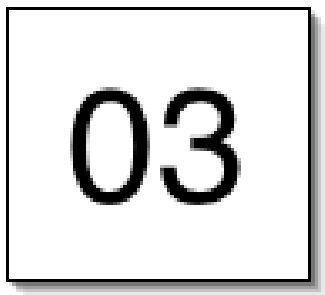
Abstraction

- 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?





Implementation - NumberDisplay

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

    // Constructor and methods omitted.
    // ...
}
```

(Full listing at http://teaching.csse.uwa.edu.au/units/CITS1001/code-listings/wk03-number.html)

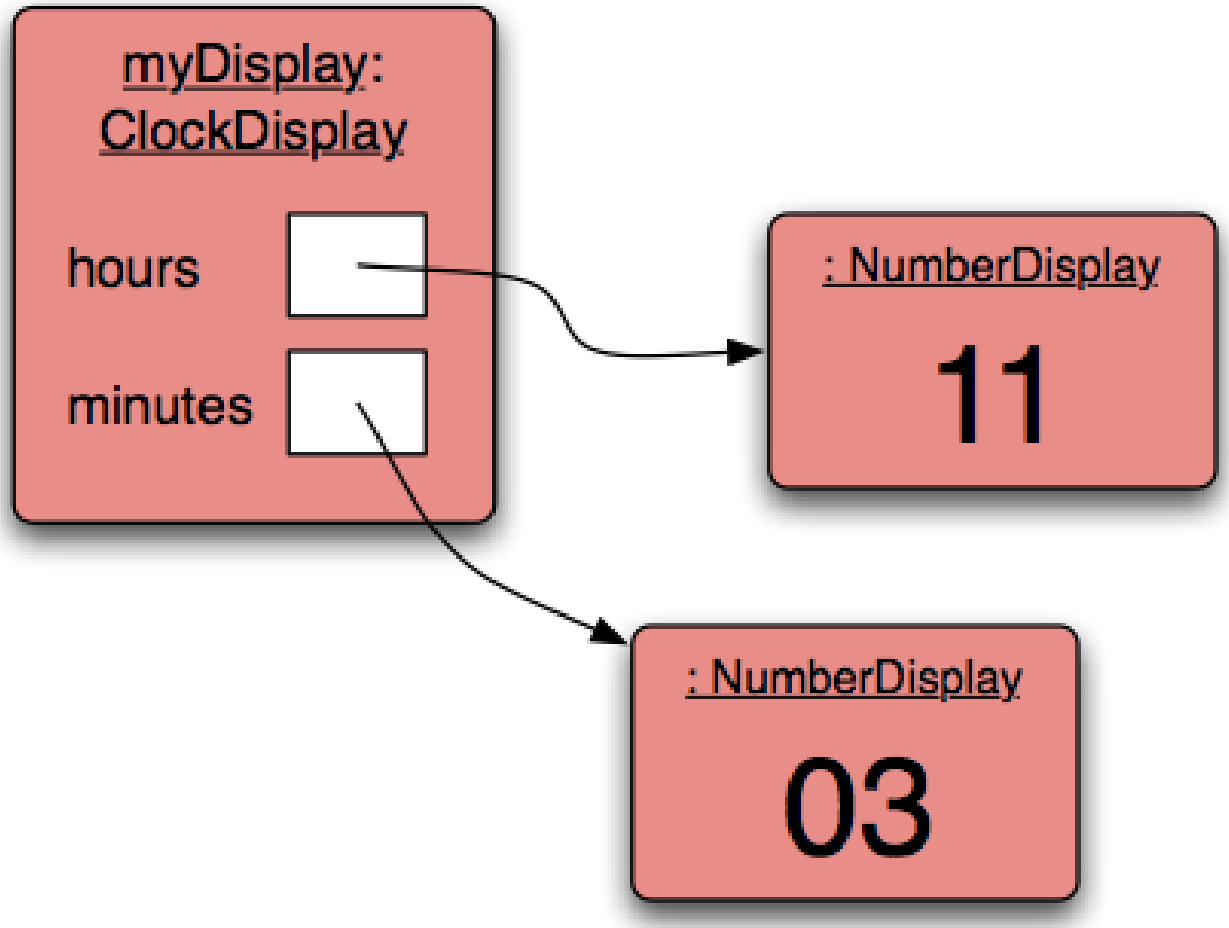
Implementation - ClockDisplay

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

    // Constructor and methods omitted.
    // ...
}
```

(Full listing at http://teaching.csse.uwa.edu.au/units/CITS1001/code-listings/wk03-clock.html)

Objects in the running program

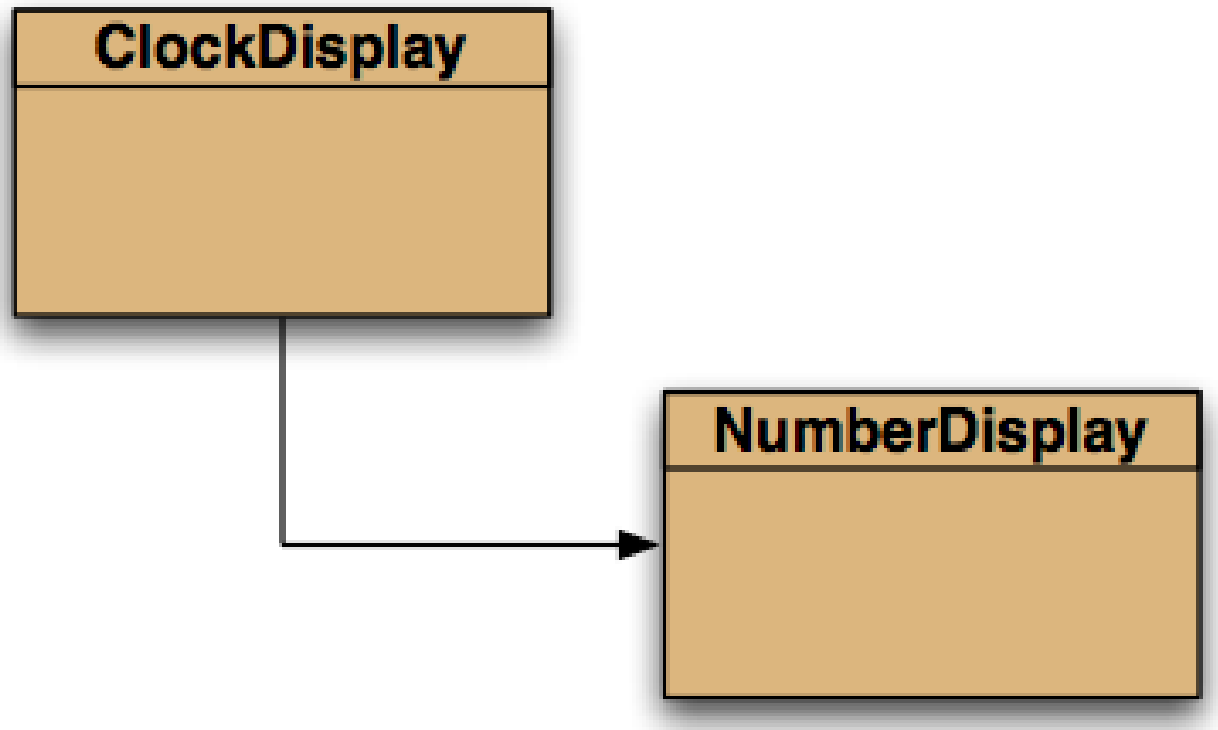


- Dynamic view at runtime (when the system is running)

Objects in the running program (2)

- 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

```
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

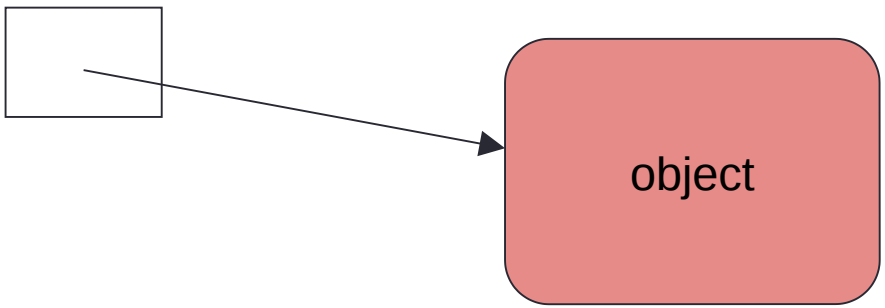
Class diagram (2)

- 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;
```

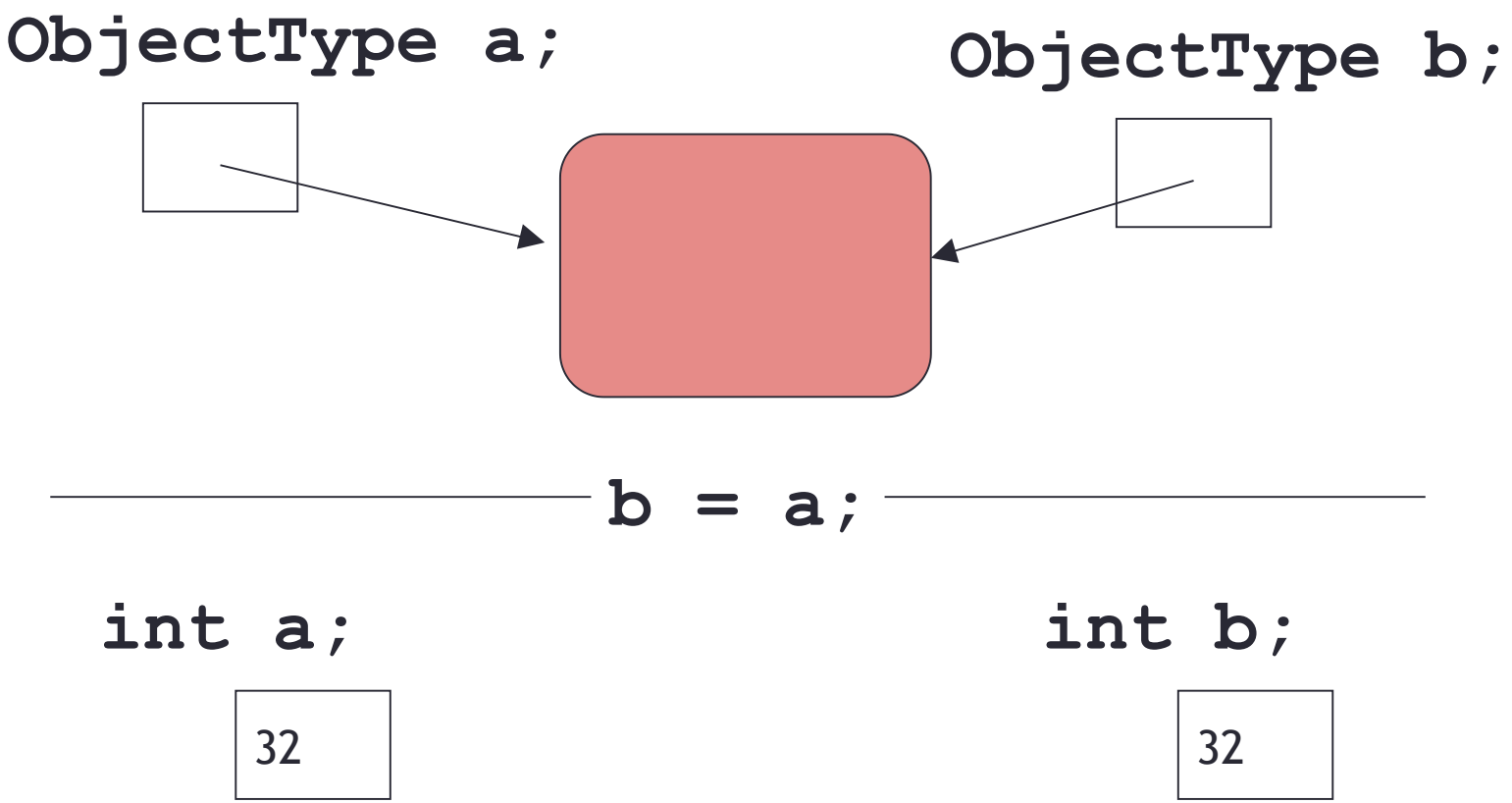


object type

```
int i;
```

primitive type

Primitive types vs. object types



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());
```

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 / 5` gives result 3, remainder 2

  In Java:

  ```
  17 / 5 == 3
  17 % 5 == 2
  ```

Quiz

- What is the result of the expression
  `8 % 3`
- For integer `n >= 0`, what are all possible results of:
  `n % 5`
- Can n be negative?

Back to the clock

Source code: NumberDisplay

```
public NumberDisplay(int rollOverLimit) {
    limit = rollOverLimit;
    value = 0;
}

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

Objects creating objects

Consider the constructor for the ClockDisplay class:

```
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);
```

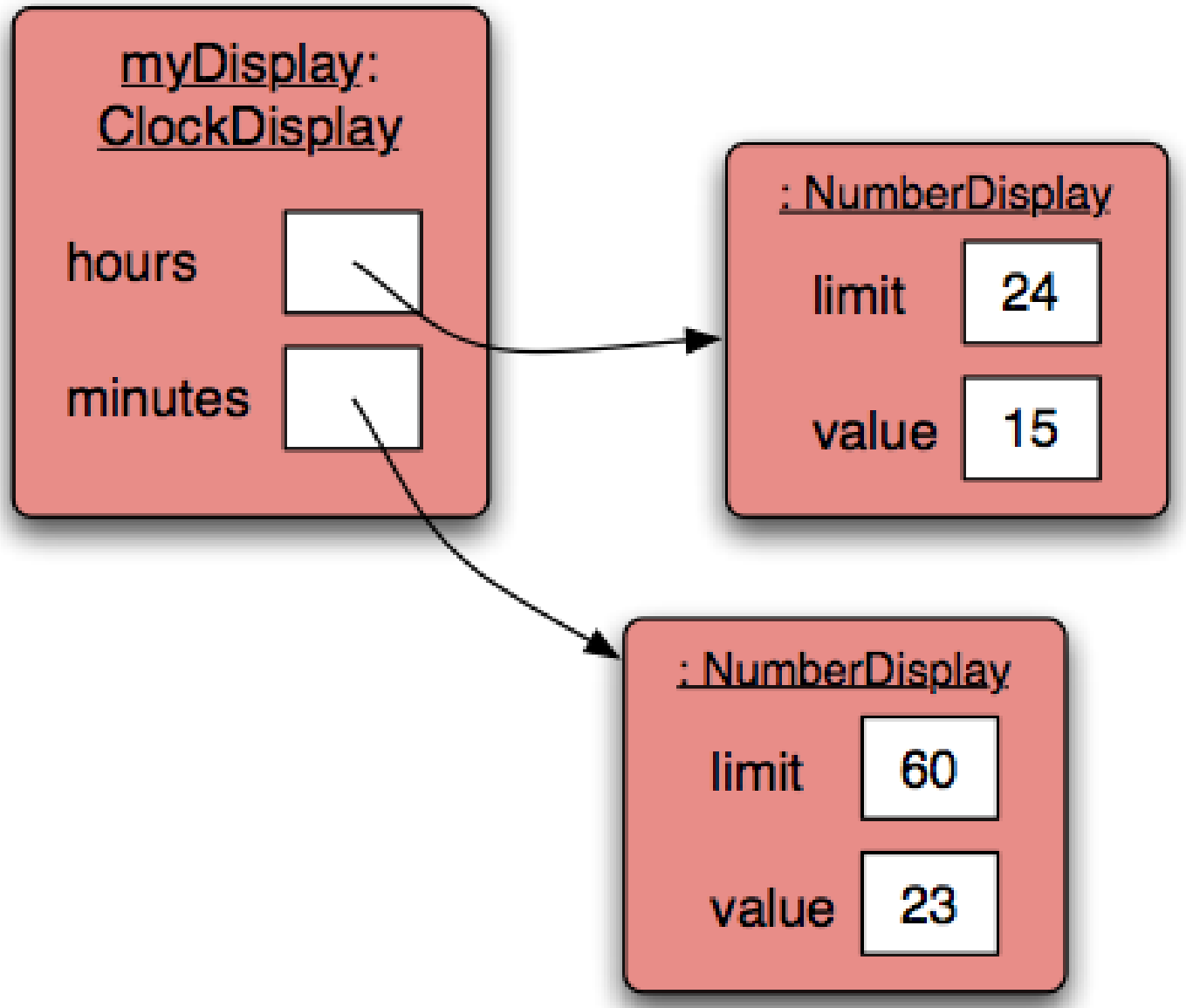
(actual parameter)

- In class NumberDisplay:

```
public NumberDisplay(int rollOverLimit);
```

(formal parameter)

ClockDisplay object diagram



Method calling

```
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:

  ```
  minutes.increment();
  ```

  where signature of increment is:

  ```
  public void increment()
  ```
- general form is:

  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:

```
updateDisplay();
```

- No variable name is required.

Internal method (helpers)

The updateDisplay method of ClockDisplay:

```
/**
 * 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

```
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.

```
public void forgetHours() {
  hours = null;
}
```

Anonymous objects

- Objects are often created and handed on elsewhere immediately:

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

- We don't really need `furtherLot`:

```
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();
```

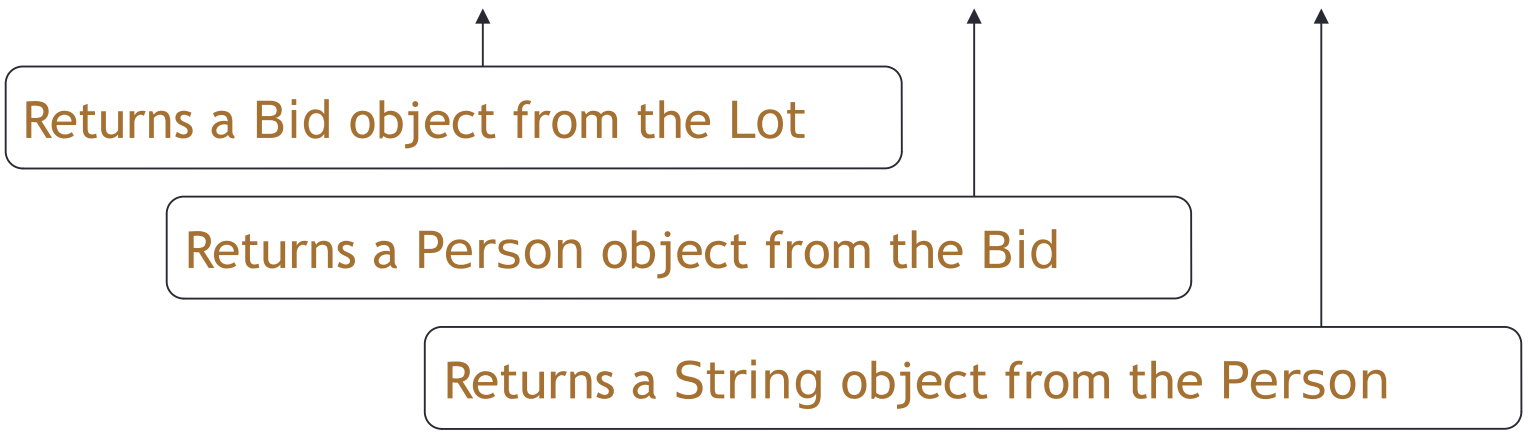
- We can use the anonymous object concept and *chain* method calls:

```
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.

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



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