THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

CompSci 230
Software Construction

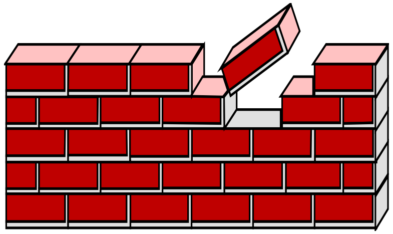
Lecture Slides #3: Introduction to OOD   S1 2015

Version 1.1 of 2015-03-12: added **return** to code on slides 10, 13

Agenda

- Topics:
  - Software Design (vs. hacking)
  - Object-Oriented Design (vs. other approaches to SW design)
  - Classes & Objects
  - Introduction to UML class diagrams
    - Object diagrams may be helpful for visualizing instantiations
  - Variables & Methods

Software Design



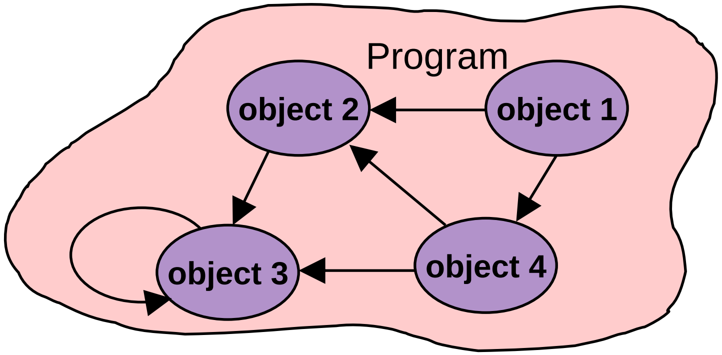
- **Communication**:
  - identify stakeholders, find out what they want and need.
- **Planning**:
  - list tasks, identify risks, obtain resources, define milestones, estimate schedule.
- **Modeling**
  - develop structure diagrams and use cases, maybe some other UML artifacts.
  - Different approaches: OO, procedural, data.
- **Construction**:
  - implement the software, with assured quality.
- **Deployment:**
  - deliver the software, then get feedback for possible revision.

To learn more:

R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th Ed., 2010, pp. 14-15.

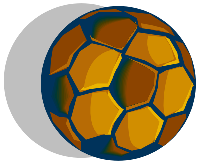
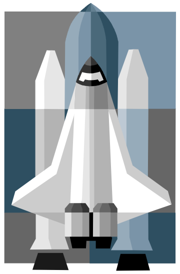
What is Object-Oriented Design?

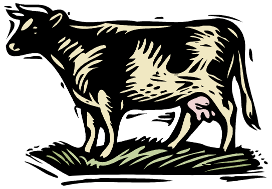
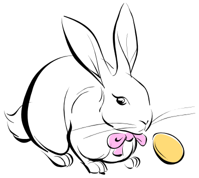
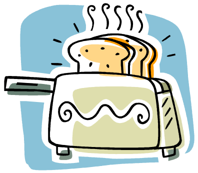
- In OO design, a system is a
  - collection of interacting objects.
  - Each object should have simple attributes and behaviours.
  - Each object should have simple relations to other objects.
- In procedural design, a system is a
  - collection of basic blocks.
  - Each basic block should have a simple effect on local and global variables.
  - Basic blocks are linked by control-flow arcs: if/then/else, call/return, while/loop, for/loop, case, goto, …
- In data architecture, a system is a
  - collection of data structures, with access and update methods.
  - Each data structure should have simple relations to other data structures.



What is an Object?

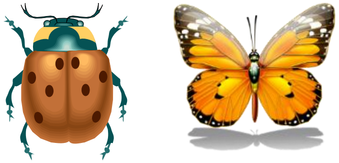
- A building block for OO development
  - Like objects in the world around us
  - Objects have state and behaviour
- Examples:
  - Dog
    - State/field/attribute: name, colour, isHungry, …
    - Behaviour: bark(), fetch(), eat(), …
  - Bicycle
    - State: gear, cadence, colour, …
    - Behaviour: brake(), turn(), changeGear(), …
  - VCR
    - State: brand, colour, isOn …
    - Behaviour: play(), stop(), rewind(), turnOn(), …

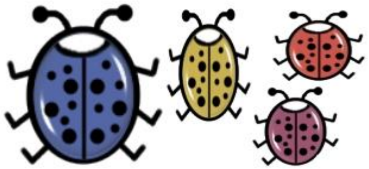




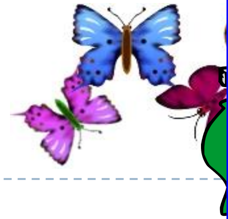
Classes & Objects

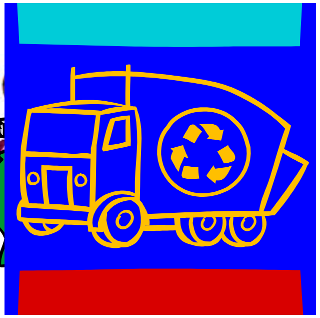
- Class
  - A set of objects with shared behaviour and individual state
  - Individual state:
    - Data is stored with each instance, as an instance variable.
  - Shared behaviour:
    - Code is stored with the class object, as a method.
  - Shared state may be stored with the class object, as a class variable.



- Object
  - Objects are created from classes at runtime by instantiation
    - usually with **`New`**.
  - There may be zero, one, or many objects (instances) of a class.
  - Instantiated objects are garbage-collected if no other user-defined object can reference them.







Imagine a world of communicating objects

- Object
  - An object remembers things (i.e. it has a memory): its state.
  - An object responds to messages it gets from other objects.
    - It performs the method with the given parameters, then sends a response.
    - An object that receives a strange message may throw an exception. Be careful!
  - An object's method may "ask for help" from other objects.
    - It sends a message to an object, and waits for a response.
    - A method may send a message to itself! This is called recursion. Be careful.
- Messages between objects
  - Usually: method calls and method returns, sometimes exceptions.

Information Hiding

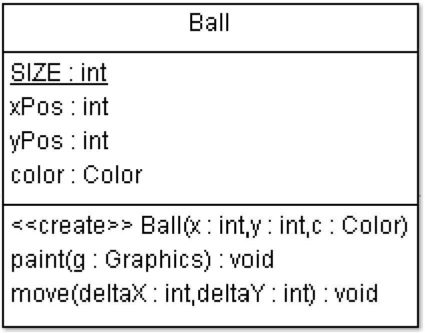
- The implementation details of a method should be of no concern to the sender of the message.
  - If a **`JavaKid`** tells a **`JavaDog`** to **`fetch()`**, the **`dog`** might run across a busy street during its **`fetch()`**.
  - Parameterised methods allow the senders to have more control over object behaviour. For example, a **`JavaDog`** might have a parameterised **`fetch()`** method:

```
ball = dog.fetch(SAFELY);
```

- Note: in these lecture slides, the word “should” indicates an element of style.
  - You should write Java code that is understandable to other Java programmers.

Example 1: Ball

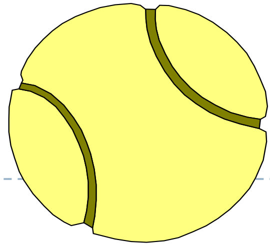
- Attributes
  - Represent the internal state of an instance of this class.
- Constructor
  - Creates the object
- Methods
  - Implement the processing performed by or to an object, often updating its state.
  - If there are read and write methods for an attribute **`x`**, these should be called **`getX()`** and **`setX()`**.
    - You should learn Java's conventions for capitalisation and naming.



Ball

SIZE : int
xPos : int
yPos : int
color : Color

<<create>> Ball(x : int,y : int,c : Color)
paint(g : Graphics) : void
move(deltaX : int,deltaY : int) : void



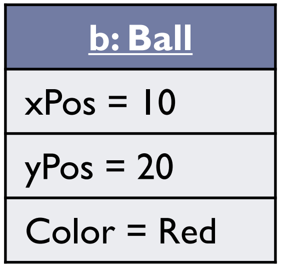
Example: Ball.java

```
public class Ball
{
  public final static int SIZE = 20;
  private int xPos;
  private int yPos;
  private Color color;
  public Ball(int x, int y, Color c) {
    xPos = x;
    yPos = y;
    color = c;
  }
  public void move(int deltaX, int deltaY) {
    xPos += deltaX;
    yPos += deltaY;
  }
  public void paint(Graphics g) {
    g.setColor(color);
    g.fillOval(xPos,yPos,SIZE,SIZE);
  }
}
```

Object Instantiation

- When a constructor method is called, a new instance is created.

```
Ball b = new Ball( 10, 20, Color.Red  );
Ball c = new Ball(  0, 10, Color.Blue );
```

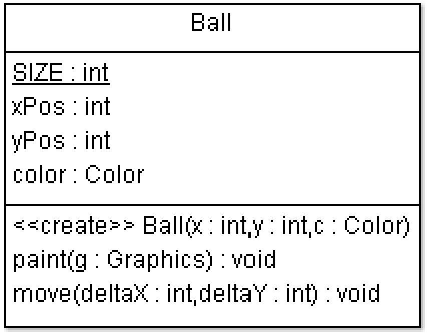


b: Ball

xPos = 10

yPos = 20

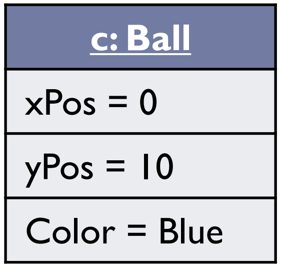
Color = Red



Ball

SIZE : int
xPos : int
yPos : int
color : Color

<<create>> Ball(x : int,y : int,c : Color)
paint(g : Graphics) : void
move(deltaX : int,deltaY : int) : void



c: Ball

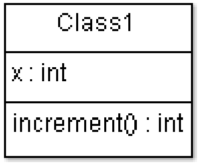
xPos = 0

yPos = 10

Color = Blue

- If a class definition doesn't include a constructor method, the Java compiler inserts a default constructor with default initialisations.

```
public class Class1 {
  private int x;
  // Note no explicit constructor
  public int increment() {
    return ++x;
  }
}      // is this good code?
```

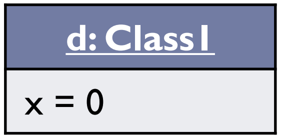


Class1

x : int

increment() : int

```
Class1 d = new Class1();
```



d: Class1

x = 0

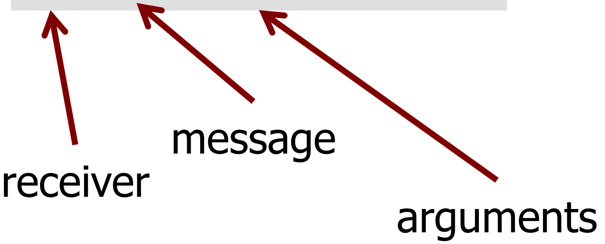
Blecch!

Message Passing

- In a method call, a message is passed to a receiver object.
- The receiver's response to the message is determined by its class.

```
Ball b = new Ball(10, 20, Color.Red);
```

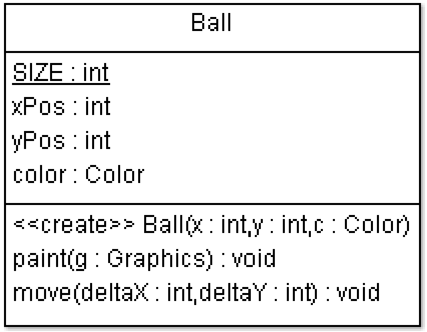
```
b.move(50, 100);
```



receiver

message

arguments



Ball

<u>SIZE : int</u>
xPos : int
yPos : int
color : Color

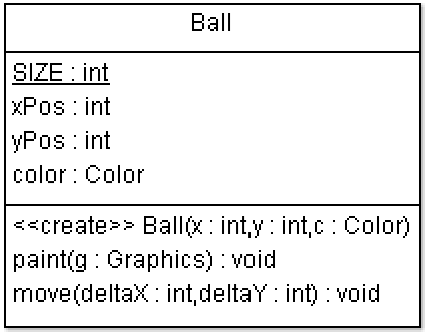
<<create>> Ball(x : int,y : int,c : Color)
paint(g : Graphics) : void
move(deltaX : int,deltaY : int) : void

<u>**b: Ball**</u>
xPos = ~~10~~ 60
yPos = ~~20~~120
Color = Red

```
public class Ball {
...
  public void move(int deltaX, int deltaY) {
    xPos += deltaX;
    yPos += deltaY;
  }
}
```

Instance & Class Variables

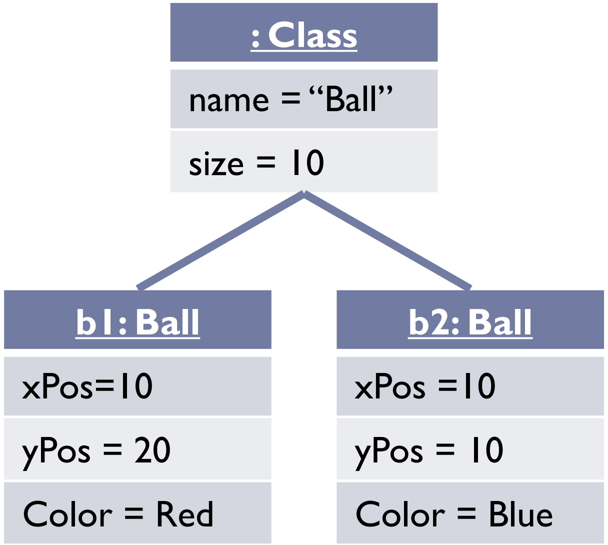
- Class variables are statically allocated, so they
  - are shared by an entire Class of objects.
  - The runtime system allocates class variables once per class, regardless of the number of instances created of that class.
  - Static storage is allocated when the class is loaded.
  - All instances share the same copy of the class variables.
- Instance variables are dynamically allocated, so they
  - may have different values in each instance of an object.
  - When an object is instantiated, the runtime system allocates some memory to this instance – so that it can "remember" the values it stores in instance variables.
- Test your understanding:
  - List the names of all class variables in Ball.
  - List the names of all instance variables in Ball.



Ball

SIZE : int
xPos : int
yPos : int
color : Color

<<create>> Ball(x : int,y : int,c : Color)
paint(g : Graphics) : void
move(deltaX : int,deltaY : int) : void



: Class
name = "Ball"
size = 10

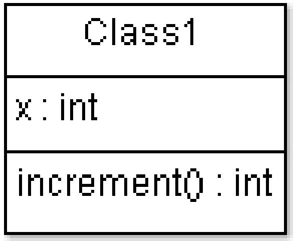
b1: Ball
xPos=10
yPos = 20
Color = Red

b2: Ball
xPos =10
yPos = 10
Color = Blue

Instance & Class Methods

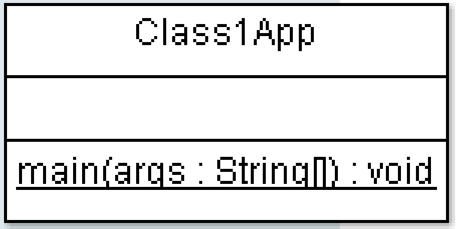
- Instance methods operate on **`this`** object's instance variables.
  - They also have read & write access to class variables.
  - E.g. ________________
- Class methods are **`static`.**
  - Class methods cannot access instance variables.
  - Class methods are handled by the "class object" – they can be called even if there are no instances of this class.
  - (Example on the next slide.)

```
public class Class1 {
  private int x;
  public int increment() {
    return ++x; // or x++ ?
  }
}
```



Class1App

```
public class Class1App {
  public static void main( String[] args ) {
    Class1 x = new Class1();
    System.out.println(
      "Without initialisation, ++x = "
      + x.increment()
    );
    System.out.println(
      "After another incrementation, ++x = "
      + x.increment()
    );
  }
}
```



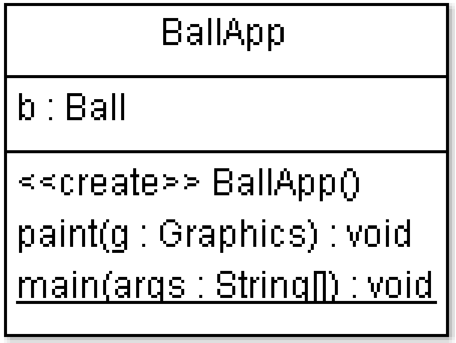
BallApp

```
import java.awt.*;
import java.awt.event.*;

public class BallApp extends Frame{
  Ball b = new Ball( 20, 30,  Color.blue );

  public BallApp() {
    addWindowListener(
      new  WindowAdapter() {
        public void windowClosing(
          WindowEvent e
        ) {
          System.exit( 0 );
        }
      }
    );
    setSize( 300,  200 );
    setVisible( true );
  }
  public void paint(Graphics g) {
    b.paint( g );
  }

  public static void main(
    String[] args
  ) {
    new BallApp();
  }
}
```



BallApp

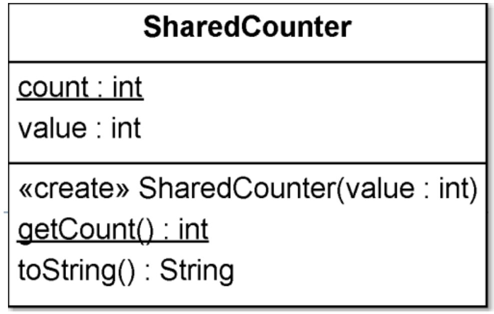
b : Ball

<<create>> BallApp()

paint(g : Graphics) : void

main(args : String[]) : void

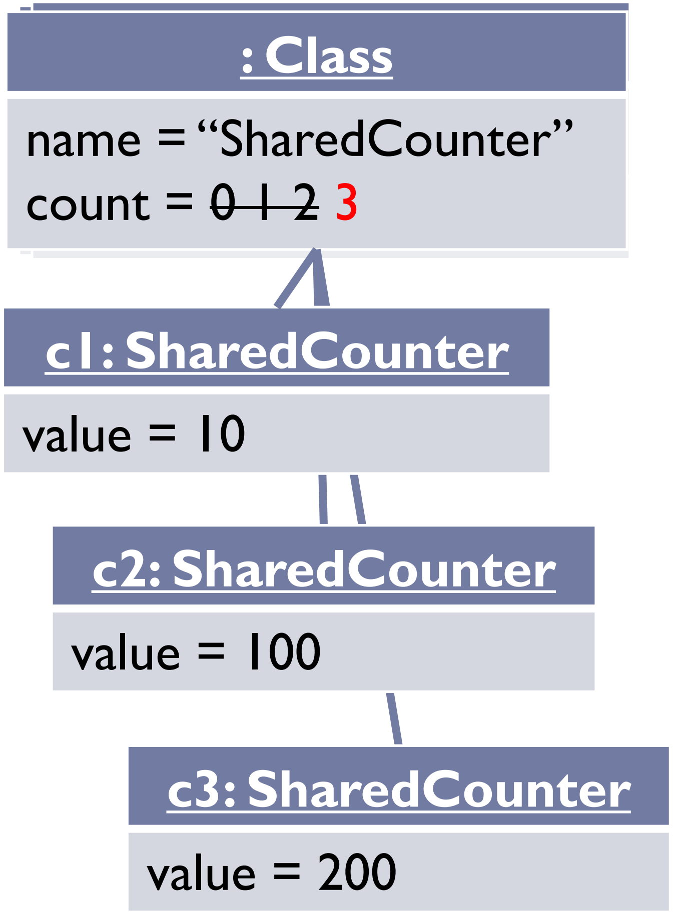
```
public class SharedCounter {
  private static int count;
  private int value;
  public SharedCounter(int value) {
    this.value = value;
    count++;
  }
  public int getValue() {
    return value;
  }
  public static int getCount() {
    return count;
  }
  public String toString() {
    return "value=" + value + " count=" + count;
  }
}
```



SharedCounter

count : int
value : int

«create» SharedCounter(value : int)
getCount() : int
toString() : String



: Class

name = “SharedCounter”
count = ~~0 1 2~~ 3

c1: SharedCounter

value = 10

c2: SharedCounter

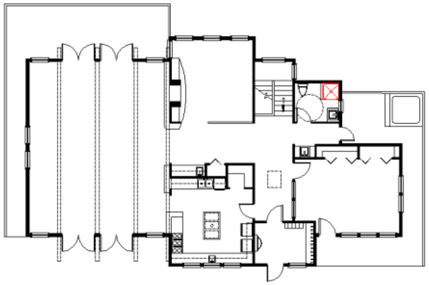
value = 100

c3: SharedCounter

value = 200

```
public static void main(String[] args) {
  SharedCounter c1 = new SharedCounter(10);
  SharedCounter c2 = new SharedCounter(100);
  SharedCounter c3 = new SharedCounter(200);
  System.out.println(c1 + " " + c2 + " " + c3);
}
```

UML



- Unified Modeling Language (UML)
  - When creating complex OO systems, where do we start?
  - When building complex systems, it might be worthwhile to plan things out before you start coding!
    - When building a house, we usually have a set of plans.
- UML is a language which allows us to graphically model an OO system in a standardised format.
  - This helps us (and others!) understand the system.
- There are many different UML diagrams, allowing us to model designs from many different viewpoints. Roughly, there are
  - Structure diagrams (documenting the architecture), e.g. class diagrams
  - Behaviour diagrams (documenting the functionality), e.g. use-case diagrams

Object Diagrams in UML

- In this lecture, I have drawn some object diagrams of instance models (using coloured boxes).
  - An object diagram is a graphic representation of an instance model, showing the state of a system after some objects have been instantiated, and after some variables of these objects have been updated.
  - Object diagrams are very helpful in tuition, but are *not commonly used outside the classroom*.
- Please focus on the basics.
  - Understand the distinction between static variables and instance variables.
  - Develop a working understanding of instantiation – this is a crucial concept!
  - Learn how to draw UML-standard class diagrams.
  - Honours-level students *might* want to learn more about object diagrams. I recommend "Modelling instances of classifiers using UML object diagrams", online Help resource for the IBM Rational Software Modeler, available 4 March 2014.

Tool Support: Eclipse & ArgoUML?

- You will need a Java development environment. I strongly recommend Eclipse.
    - The de-facto industry standard for Java developers. It's FOSS: free and open-source software. Its codebase is robust and is under active development. Your tutors will help you learn Eclipse.
  - Alternatively, you may use `javac` and a text editor (e.g. emacs) with Java support
    - I reckon every Java developer should know how to run `javac` from a console, but I won't attempt to teach this!
- You will draw some class diagrams and use-case diagrams. Options:
  - ArgoUML
    - Supports forward- and reverse-engineering.
      - □ Class diagrams → Java skeletons. Java classes → class diagrams.
    - FOSS, works ok but missing some features such as an "undo" button – save your versions carefully!
    - No longer under active development: v0.34 is dated 15 December 2011.
    - Not on lab image – you'll have to download and unzip the binary distribution in your echome directory (or on your USB pendrive) then double-click on `argouml.jar` (this is an "executable jarfile"). See http://argouml-stats.tigris.org/documentation/quickguide-0.32/ch02s02.html.
  - Any general-purpose drawing package (e.g. Visio)
    - Warning: you'll have trouble with the fancy arrowheads in UML! Maybe Softwarestencils.com/uml/visio?
  - By hand:
    - This is your only option during exams and tests
    - You'll have to scan your drawings into your assignments (which are submitted online)

Review

- The OO approach is based on modeling the real world using interacting objects.
  - OO design is a process of determining what the stakeholders require, designing a set of classes with objects which will meet these requirements, implementing, and delivering.
- The statements in a class define what its objects remember and what they can do (the messages they can understand), that is, they define
  - Instance variables, class variables, instance methods, and class methods
- The hardest concept in this set of lecture slides: instantiation.
  - Very important!
- A UML class diagram shows the "bare bones" of an OO system design.
  - It need not show all classes! (A diagram should not have irrelevant information.)