Building custom components
IAT351

Week 1 Lecture 1

9.05.2012

Lyn Bartram

lyn@sfu.ca

SFU

SCHOOL OF INTERACTIVE ARTS + TECHNOLOGY

Today

- Review assignment issues
  - New submission method
- Object oriented design
- How to extend Java and how to scope
- Final project

Le HUGE sigh

- Submission approach did not work
- Moved to SAKAI for submission and feedback ONLY
- Log into sakai.sfu.ca

Assignment 1 showed …

- You can code Java! Most of you got it
  - Made use of existing Swing components

- Everything in one class
- Often most of the work in the `main` routine
- Lots of `static` declarations
- Inner and private classes

- …poor code architecture for getting bigger

Why do I need to make new (more) components?

- Re-use code
- Reduce overhead and complexity
- Specialise and localise behaviour

- "Outsource" programming:
  - find something that does almost what you want, or a piece of what you want, and design your application as a collection of these

This is a *design* issue

- Software engineering is about good *design* and not great *programming*
- Software architecture requires
  - Well-defined information flow
  - Appropriate allocation of function to objects according to requirements
  - Clear division of responsibility and function between functional components
- Design before you code!
- Your diagrams/sketches/stories are your best tools

Object-Oriented Design

Simplified methodology: tell a story (*use case*)

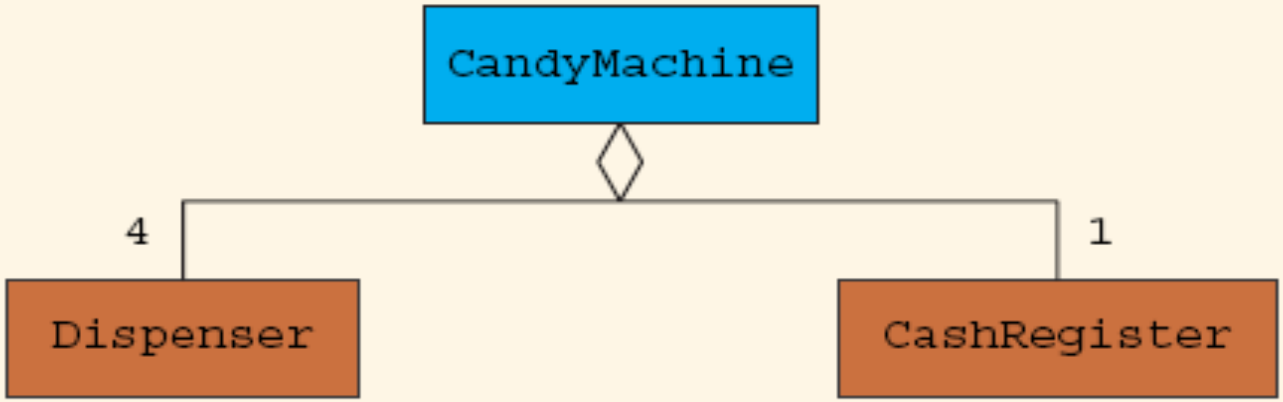
1. Write down detailed description of problem
2. Identify all (relevant) *nouns* and *verbs*
3. From list of nouns, select *objects*
4. Identify data components of each object (variables)
5. From list of *verbs*, select operations (methods)

Example

- An inoperable candy machine has a cash register and several dispensers to hold and release items sold by the machine
- Let's design the software outline for a machine that dispenses 4 items: candies, chips, gum, and cookies

Example cont.

- The program should do the following:
  - *Show* the **customer** the different **products** sold by the **candy machine**
  - Let the **customer** *make* the selection
  - *Show* the **customer** the **cost of the item** selected
  - *Accept* **money** from the **customer**
  - *Return* **change**
  - *Release* the **item**; that is, *make* the sale
- Notice the careful thought to nouns and verbs.
- The dispenser can keep track of the cost of the item, number in stock, etc



need a `Dispenser` class to record info about the items we are selling.
a `CashRegister` class to record how much money was inserted, and how much should be returned.
a `CandyMachine` class in which we will instantiate 4 `Dispenser` objects, and 1 `CashRegister` object.

•think of the fields & methods to use for these classes? This is the key step. Don't move on until you've spent some time thinking about this.

Cash Register Example cont.

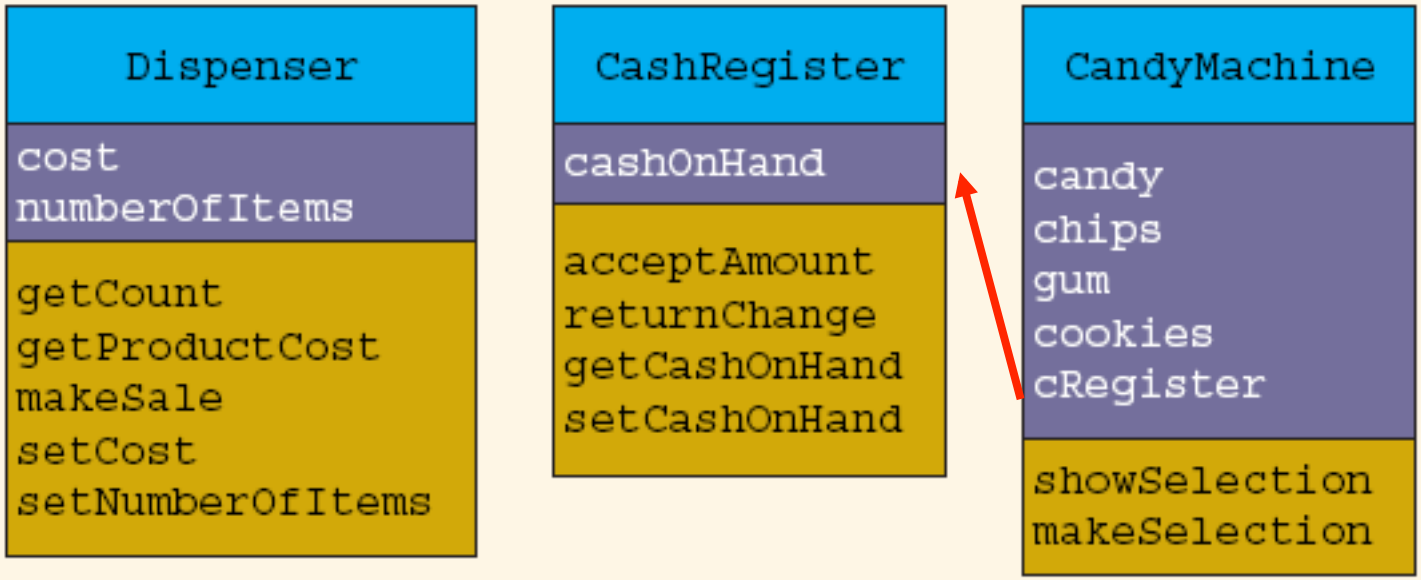


FIGURE 6-15 Classes `Dispenser`, `CashRegister`, `CandyMachine`, and their members

- Now what about putting this into a GUI?
- Can use layout managers and image support of Swing components

`DispenserWindow` **`extends`** `Jpanel`

- Contains a `Dispenser`

`Dispenser` **`extends`** `Jpanel`

- Adds `Dispenser` functions to `JPanel`

JPanels

JPanel

Design patterns for customising classes

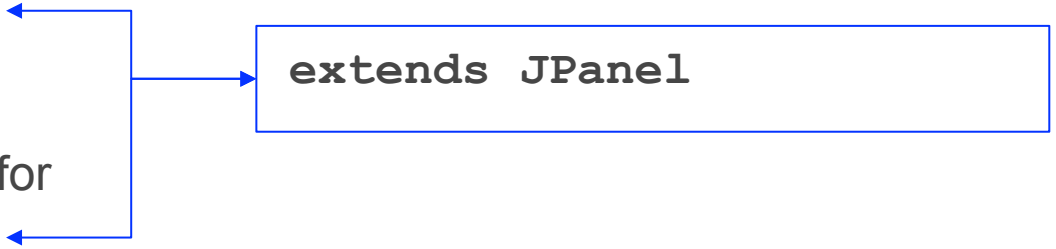
- *Inheritance*:
  - Derive a new subclass from a parent class
  - Use all the functionality and add new behaviour
  - Have to be careful of trashing previously defined methods (*over-riding*)
- *Object composition*
  - Collect and organise existing classes into new

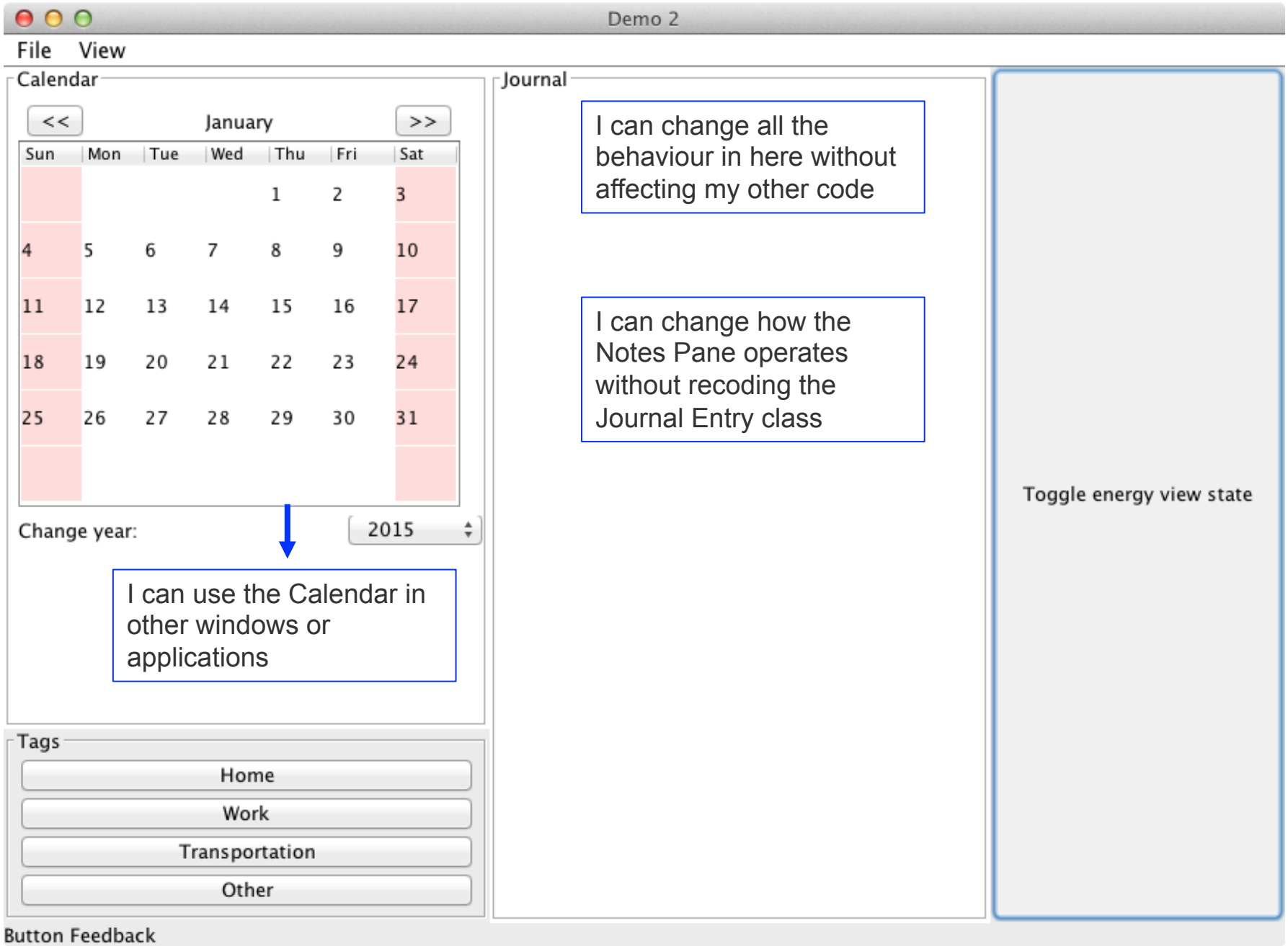
- Good code architecture is a combination of these

Example

Journal has:

- Calendar
- JournalEntry
  - Notes Pane: for taking notes, text entries, discussion
  - Graphics/media Pane: for inserting media objects that can include short annotations
  - Date, author, etc




Demo 2
File
View
Calendar
<<
January
>>
Sun
Mon
Tue
Wed
Thu
Fri
Sat
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
Change year:
2015
I can use the Calendar in other windows or applications
Tags
Home
Work
Transportation
Other
Journal
I can change all the behaviour in here without affecting my other code
I can change how the Notes Pane operates without recoding the Journal Entry class
Toggle energy view state
Button Feedback

Inheritance vs. composition

Inheritance

- Is-a relationship exists
  - `Bicycle` is-a `Vehicle`
- Parent does a lot of what child needs to do
- Add only specialisation behaviour and variables
- **`extends`** keyword
- Be **careful** when over-riding methods
- Add rather than replace!

Composition

- `has-a` relation exists
  - `Vehicle` has-a `Wheel`
- Object manages its constituent members
- Creates as necessary
- **`new`**
- Can't share behaviour
  - No over-ride danger
  - More code to write

(Inheritance vs. composition) + interface

Inheritance

- Is-a relationship exists
  - `Bicycle` is-a `Vehicle`
- Add only specialisation behaviour and variables
- **`extends`** keyword

Composition

- `has-a` relation exists
  - `Vehicle` has-a `Wheel`
- Object manages its constituent members
- **`New`** keyword

Functional support (interface)

- `Does-support-a` relation exists
  - `Vehicle` supports `Steering`
- **`implements`**
- Requirements for behaviour

Some rules of thumb

- If you are duplicating code in several objects that are similar, make a parent object and subclass
- If you need to encapsulate behaviour and hide it so that other objects don't need to know about it, make a new class
- If your code is getting very complex and unwieldy, consider restructuring and re-classing

The complexities of class

- Access to information
- SCOPE
- Some support defined at declaration
- Some defined by structure

- Parent-child
- Inner-outer

Parent-child

Child class has access to all of parent's behaviour

- Implicitly: by creation
- Explicitly: by accessing exposed methods and variables
  - `myPanel.setLayout(..)`

Child can alter and add to behaviour

- Over-riding methods
- Can cause damage! Be careful. With great power comes great responsibility
- Use `super()` to invoke the parent's method

Class Hierarchy

- Good class design puts all common features as high in the hierarchy as reasonable
- inheritance is transitive
  - An instance of class `Parrot` is also an instance of `Bird`, an instance of `Animal`, …, and an instance of class `Object`
- The class hierarchy determines how methods are executed:
  - when variable v is an instance of class C, then a procedure call v.proc1() invokes the method proc1() defined in class C
  - However, if C is a child of some superclass C', methods of class C can *override* the methods of class C' (next two slides).

Defining Methods in the Child Class: Overriding by Replacement

- A child class can *override* the definition of an inherited method in favor of its own
  - that is, a child can redefine a method that it inherits from its parent
  - the new method must have the same signature as the parent's method, but can have different code in the body

- all methods except of constructors override the methods of their ancestor class by replacement. E.g.:
  - the Animal class has method eat()
  - the Bird class has method eat() and Bird extends Animal
  - variable *b* is of class Bird, i.e. Bird b = …
  - b.eat() simply invokes the eat() method of the Bird class
- If a method is declared with the `final` modifier, it cannot be overridden

Defining Methods in the Child Class: Overriding by Refinement

- Constructors in a subclass *override* the definition of an inherited constructor method by *refining* them (instead of replacing them)
- Assume class Animal has constructors Let's say we create a Bird object, e.g. Bird b = Bird(5)
- This will invoke ***first*** the constructor of the Animal (the superclass of Bird) and ***then*** the constructor of the Bird

- This is called *constructor chaining*

Overloading vs. Overriding

- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overloading lets you define a similar operation in different ways for different data

- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overriding lets you define a similar operation in different ways for different object types

Overloading

- two or more methods have same name within the same class
- Java distinguishes the methods by number and types of parameters
  - it attempts to do type conversions (bad idea)
- A method's name and number and type of parameters is called the *signature*
- Can call one overloaded method from another

```
public void addJournalEntry(Date d, int u, String t)
{
  addJournalEntry(d, t,);
  setUserId(u);
}
```

Understanding access

- Because a class encapsulates its data (variables) and behaviour (methods) access to them is SCOPED
  - **`private`**: only objects of this class
  - **`public`**: any objects (world)
  - **`protected`**: only objects of this class and any subclsses
  - **`private`**: only objects of this class
- Good design is to privatise any data/method that is strictly internal
- *Expose* by public methods and fields: the API

Understanding access (2): `static`

- Static variables are shared by all objects of a class
  - Variables declared **`static final`** are considered constants – value cannot be changed
- Variables declared **`static`** (without **`final`**) can be changed
  - Only one instance of the variable exists
  - It can be accessed by all instances of the class

```
public class SavingsAccount {

private double balance;  // only this object knows this
public static double interestRate = 0;
public static int numberOfAccounts = 0;

public static final int bankID = 4072;
//only one instance of this anywhere in the application

public SavingsAccount ()
{
    balance = 0;
    numberOfAccounts++;  only one value
}
```

Static Variables

- Static variables also called *class variables*
  - Contrast with *instance variables*
- Do not confuse class variables with variables of a class type
- Both static variables and instance variables are sometimes called *fields* or *data members*

- ?? Problems ??

Static Methods

- Some methods may have no relation to any type of object
- Example
  - Compute max of two integers
  - Convert character from upper- to lower case
- Static method declared in a class
  - Can be invoked without using an object
  - Instead use the class name
- Good way to define *utilities* your application needs

```
/**
Class of static methods to perform dimension conversions.
*/
public class DimensionConverter
{
    public static final int INCHES_PER_FOOT = 12;

    public static double convertFeetToInches (double feet)
    {
        return feet * INCHES_PER_FOOT;
    }


    public static double convertInchesToFeet (double inches)
    {
        return inches / INCHES_PER_FOOT;
    }
}
```

```
import java.util.Scanner;
/**
Demonstration of using the class DimensionConverter.
*/
public class DimensionConverterDemo
{
    public static void main (String [] args)
    {
        Scanner keyboard = new Scanner (System.in);
        System.out.println ("Enter a measurement in inches: ");
        double inches = keyboard.nextDouble ();
        double feet =
            DimensionConverter.convertInchesToFeet (inches);
        System.out.println (inches + " inches = " +
                feet + " feet.");
        System.out.print ("Enter a measurement in feet: ");
        feet = keyboard.nextDouble ();
        inches = DimensionConverter.convertFeetToInches (feet);
        System.out.println (feet + " feet = " +
                inches + " inches.");
    }
}
```

Types of nested classes

- Inner classes
  - local
    - anonymous or named
  - non-local
    - named only
- Static nested classes
  - non-local named only

Non-local inner classes

- Simply a nested class that does not have the **`static`** attribute and is not defined within a class method.
- Can be private, public, package, protected, abstract, etc. just like any class member.
- Think of outer class as owning inner class – inner class can only be instantiated via outer class reference (including **`this`**)
- Inner class has access to all outer class iv's, private or otherwise!

Simple non-local inner class example

```
class Outer{
  private int x1;
  Outer(int x1){
     this.x1 = x1;
  }
  public void foo(){ System.out.println("fooing");}
  public class Inner{
     private int x1 = 0;
     void foo(){
       System.out.println("Outer value of x1: " + Outer.this.x1)
       System.out.println("Inner value of x1: " + this.x1);
     }
  }
```

When to use non-local inner classes

- Most typically used when inner class is instantiated from outer class.
- If classes naturally "belong together", it is cumbersome to pass a this pointer to a separate outer class just so second class can access first class's properties/ methods.
- Note that inner class can access outer class's private data, making them even more powerful than mechanism implied above

- Best example: extending listeners

Local inner classes

- Inner classes may also be defined within class methods.
  - These are called *local inner classes*.
- Principle advantage is scoping: such classes are completely inaccessible anywhere but the method itself where they are defined.
  - Also, can NOT access local variables other than those declared with `final` attribute.
- Very hard to debug, so handle with care

Summary

- Making GUIs from software components depends on good architecture design as much as good user interface look and feel
- Object oriented principles good guidelines
  - Inheritance where logical
  - Composition as organiser
- Pay attention to scoping and levels of access