Programming - A summary for final

Applications Programming (University of Technology Sydney)

Process:

- A small program has one goal = one method.
- A large program has many sub-goals = many methods.
- Code reuse is the main benefit of splitting code into small methods.
- Put each goal in a separate method so that it can be reused

- **Goal**: what your program should achieve
- **Plan**: a series of steps to achieve the goal
- **Key**: the key line of code that achieves the goal
- Sometimes the goal is too difficult
- Start with a simplified goal and devise a plan for that
- Gradually add more, until the complete goal is achieved

Methods:

We consider two kinds of methods:

1. A **procedure** does something. It's name is a verb.
2. A **function** returns something. It's name is a noun.

Procedures

- A procedure is a method that does an action / has some "effect". e.g. prints a value, changes a value
- A procedure may take parameters, but should return nothing.
- The name of a procedure is a verb describing the goal.

```
public static void showCircleArea(double radius) {
double area = Math.PI * radius * radius;
System.out.println("The area of the circle is " + area);

}
```

- A procedure may use local variables. A local variable is temporary. It is deleted when the method exits.

Functions

- A function is a method that returns a value.
- A function should **not** have any side effects.

e.g. It should **not** print a value. It should **not** change a value.

- A function may take parameters.
- The name of a function is a noun describing what is returned.

```
public static double circleArea(double radius) { double
area = Math.PI * radius * radius; return area;

}
```

● A function may also use local variables.

Side effects

- Function design rule
  A function returns a value and changes nothing
- If a function changes something, this is called a “side effect”
- Side effects are bad:
  - the reader assumes the function changes nothing
  - the reader does not look inside the function
  - because a function changes nothing
- Avoid programming by side effect. Unless it is a known pattern.

Interaction between procedures and functions

- A procedure can call a function.
- A function can call a function.
- **But** a function should not call a procedure

  Functions should not have side effects
  Calling a procedure may introduce side effects

Strings

● In java, String is a class, providing a set of useful functions.

int length()	returns the length of the string
char charAt(int i)	returns the character at position i
String[] split(String separator)	returns an array of substrings split by the separator

The "string loop" pattern

Goal: Loop over the characters in a string.

```
for (int i = 0; i < <str>.length(); i++) {
   <use character str.charAt(i)>
}
```

The "for-each" loop

Create an array of values
String[] array = { "car", "truck", "bus", "van" };

These two code fragments **do the same thing**: for (int i = 0; i < array.length; i++)

System.out.println(array[i]);

for (String word **:** array) System.out.println(word);

Read: **For** each word **in** array, print that word.

A complete program still needs procedures!

- Functions don't have any "effect".
- To cause something to "happen" we need procedures.
- e.g. "show" the number of matching words in the terminal: `public static void` **`showMatchingWords`**`(String sentence) {`

  ```
  System.out.println("Matching words = " +
  matchingWords(sentence)); }
  ```

- Every program must have a main also have a main method:

  ```
  public static void main(String[] args){
  showMatchingWords(readSentence()); }
  ```

Interaction between procedures and functions

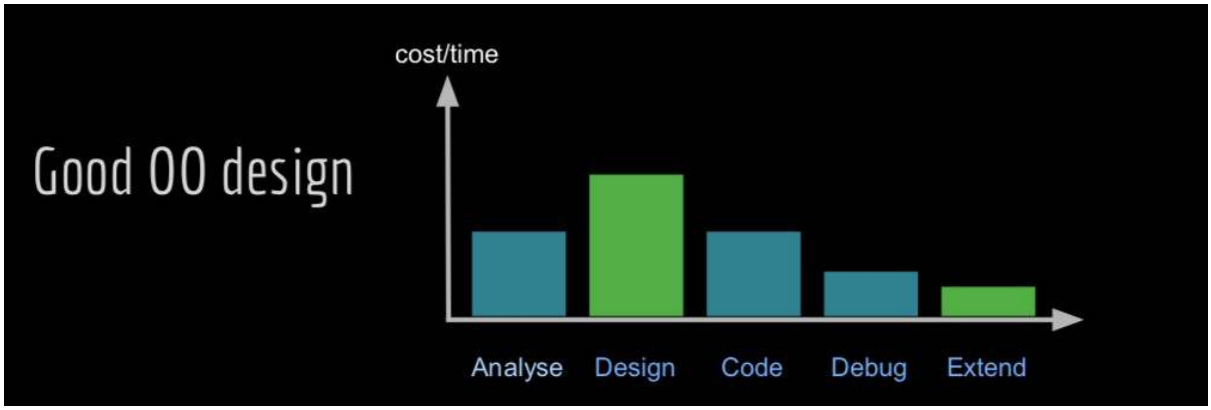
- The functions do all the grunt work
  - split a sentence into words
  - count the vowel words
  - test if a word contains a vowel
  - test if a character is a vowel
- The procedures just present the result of that hard work.

Classes

- We break down a larger program into classes of objects.
- This is **object oriented programming**.

Advantages of Object-Oriented Programming

- Each kind of object has separate concerns. Different programmers can code different kinds of object without stepping on each other's toes.
- Dependencies between objects are few and easy to manage. Most dependencies are isolated within an object.
- Objects export an interface and hide the implementation details. The programmer of one object can change its internal details without bothering the programmers of other objects.
- Object structures simplify naming. e.g. if `accountBalance` is inside an account object, just name it `balance`.
- Objects better map onto the way the real world works. The real world has objects.



What is a class?

A class is a template for creating objects.
The members of a class are **fields** and **methods**.

What is an object?

An object is an instance of a class. Each object gets its own copy of the members.

Instance vs Static

- Static members:

```
private static int x;
     public static void foo() { ... }
```

- Instance members: `private int x;`
  `public void foo() { ... }`
- Only instance members are copied into each object

Class diagrams

● An arrow indicates one class uses another class.
● * indicates multiplicity.
● Class diagrams help us to sketch and evaluate OO program designs.
● A class is depicted as a box with class name / fields / methods

Design rule #1: Encapsulation

Encapsulation:

- Fields are **hidden** behind methods
  - fields are always private
  - methods may be public
- An **object** encapsulates related fields+methods.
- **Rule:** If a method uses a field, it is defined in the same class.

Design rule #2: Push it right

Design rule #3: Spread plans across classes

● Convention: Use the same method name across classes for the same goal.

Design rule #4: Hide by default

- Make everything private unless there is a reason to make it public.
- Make all fields private.
- Make methods private if no other class needs to use them.
- Make methods public only if other classes need to use them.

Access modifiers

Class members may be declared with an access modifier.

- **private**: can be accessed only within the class. `private double readBalance()`
- **no modifier**: can also be accessed within the package. `String getPassword()`
- **protected**: can also be accessed by subclasses. `protected int width;`
- **public**: can also be accessed by other classes. `public void deposit()`

Format to 2 decimal places - pattern

Goal: Show to two decimal places.

```
@Override
public String toString() {
return "The account has $" + formatted(balance);
}
private String formatted(double value) {
DecimalFormat f = new DecimalFormat("###,##0.00");
        return f.format(value);
    }
```

0 means always show a digit. # means show a digit if needed. **Package to import**: java.text.*

Process steps

● Analysis/Design

- Read the specification
- Identify the classes and fields (analyse the **nouns**)
- Identify the constructors (look for these words: **initial**, **create**, **add**)
- Identify the goals (analyse the **verbs**)
- Write these down on a class diagram, following design rules

● Coding

- Code the classes and fields
- Code the constructors
- Write a plan for each goal (patterns and key code)
- Code the goals as methods
- Add the main method

Lists

Lists are data structures that:

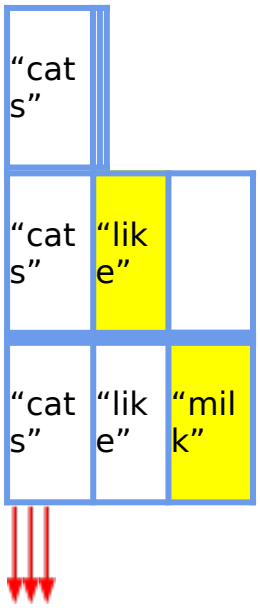
- Store a sequence of elements - like an array.
- Allow new elements to be added - unlike an array!
- Allow elements to be removed - unlike an array!

They are implemented as classes which you import:

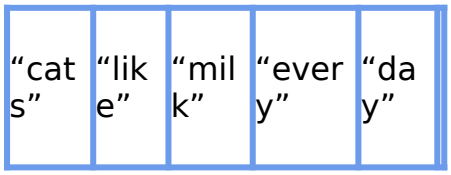
```
import java.util.*;
```

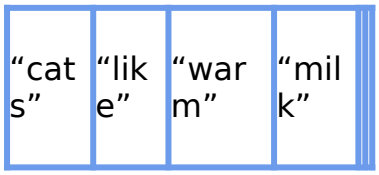
Array lists

- An array list uses an array internally, with extra space at the end...
- ... so that you have room to add more elements

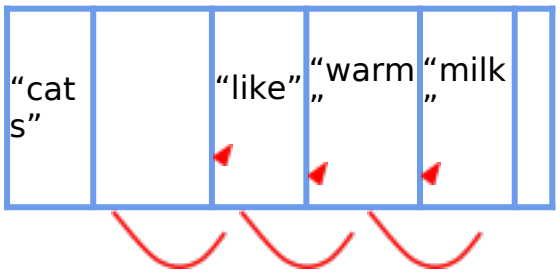


● When you run out of space, a bigger array is created and the elements copied across:

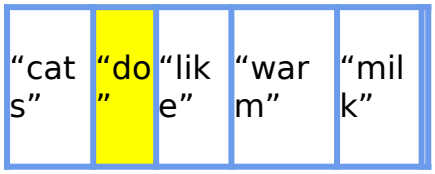




● To insert an element, you must shift elements to the right



● Then insert



Should I use an array list?

- Array lists provide instant access to any element. They are FAST.
- Adding elements to the end of an array list is reasonably fast.
- Inserting elements near the beginning of a list is slow.

  Use an array list if you need random access to elements.
  Don’t use an array list if you often need to insert elements near the beginning.

Linked lists

- Elements are stored in objects that are linked together
- To insert an element, just change two arrows

Type parameters vs Method parameters

- Method parameters go after a method and use round brackets:
  ```
  System.out.println(“zoo”);
  repeat(5, “* “);
  ```
- Type parameters go after a type and use angled brackets:
  ```
  LinkedList<Customer> customers; ArrayList<Card> cards;
  TreeSet<String> symbols;
  ```
- Type parameters must be classes. For primitives, use class wrappers:
  ```
  LinkedList<Integer> ages;
  ArrayList<Double> rainfall;
  ```

LinkedList‹X› and ArrayList‹X› methods

Method	**Description**
add(X element)	Add an element of type X to the end
add(int i, X element)	Add an element of type X at position i

remove(X element)	Remove this element
remove(int i)	Remove the element at position i
set(int i, X element)	Replace the element at position i
X get(int i)	Return the element at position i
int size()	Return the size of the list
clear()	Remove all elements

Copying a list

```
LinkedList<String> original = new LinkedList<String>();
-- add elements to original --
LinkedList<String> copy = new LinkedList<String>();
for (String word : original)
   copy.add(word);
```

You now have two lists that contain the same elements.

Copying a list with addAll

Method	**Description**
addAll(Collection<X> elements)	Add a collection of elements to this list

```
LinkedList<String> original = new LinkedList<String>();
-- add elements to original --
LinkedList<String> copy = new LinkedList<String>();
copy.addAll(original);
```

The “lookup” pattern

Goal: Find and return an element in a list. Return null if not found.

```
<for each item in the list>
    if (<this is the item I want>)
         return <item>;
return null;
```

Example: Find a particular kind of account. e.g. account(“Savings”)

```
private Account account(String type) { for (Account account :
accounts)

         if (type.equals(account.getType()))
             return account;
    return null;
}
```

Find all matches

Specification: Find all words in a list that contain “z”. **Solution**: Create a new list and add the matching words.

```
private LinkedList<String> zWords(LinkedList<String> words) {

    LinkedList<String> matches = new LinkedList<String>();
    for (String word : words)
         if (word.contains(“z”))
             matches.add(word);
    return matches;
}
```

Remove all matches - two correct solutions

- **Solution #1**: Make a list of z words, then remove them all at once:
  `LinkedList<String> zWords = zWords(list);`
  `list.removeAll(zWords);`
- **Solution #2**: Use an iterator:
  `for (Iterator<String> it = list.iterator();`
  `it.hasNext();)`

```
        if (it.next().contains(“z”))
              it.remove();
```

The first solution is simpler but slower (loops over the list twice).
The second solution is more complex but more efficient (loops once).

Remove one match - two solutions

- **Solution #1**: Stop loop after removing to avoid an exception: `for (String word : list)`

```
        if (word.contains("z")) {

            list.remove(word);
            break;

}
```

- **Solution #2**: Use an iterator:

```
for (Iterator<String> it = list.iterator(); it.hasNext();
)

        if (it.next().contains("z")) {
       it.remove();

break;

}
```

System Design

Interfaces

```
public interface Polygon {
   double area();
   int numberOfSides();

}
```

- An interface declares a set of methods common to multiple classes. E.g. All polygons have area() and numberOfSides() methods.
- Each class provides its own "implementation" of these methods.

Implementing an interface

- Implement an interface with the **`implements`** keyword.
- Override an interface method with the **`@Overrides`** annotation.
- Methods from an interface must be **`public`**.

The Payoff: Polymorphism

- Polymorphism allows for a single object to have many types. `new Square(10)`
- This object has type `Square` and type `Polygon`. i.e. It can be used as a Square or a Polygon.

Implementing multiple interfaces

- A class can implement multiple interfaces.
- `public class ArrayList<X>`
- `  implements List<X>, Iterable<X>`

Superclasses

Like interfaces:
● Define methods common to multiple classes.

Unlike interfaces:

- Provide implementations for those common methods[1].
- Define common fields.
- Define non-public members.

Superclass / Subclass

- A superclass defines common methods and fields.
- Each subclass inherits those common methods and fields.
- Methods which must be implemented in the subclasses are declared “abstract”.
- A class containing abstract methods must also be declared abstract.

Multiple inheritance not supported

The problem:

- Two superclasses define two different implementations of `move()`.
- Which one gets inherited into `Square`?

Java’s solution:

● A subclass cannot extend more than one superclass.

Inheritance

- Although Square did not define a `move()` method, Polygon's `move()` method was inherited:

```
Square square = new Square(10);
square.move(2, 3);
```

- Inheritance is a form of **code reuse**.
- Don't repeat code across classes. Put it in a superclass and inherit it.

Method overriding

- Non-abstract methods can also be overridden.
- The superclass's version of the method can be called with `super`.

```
public class Square extends Polygon {
    ...

@Override

    public void move(double dx, double dy) {
        super.move(dx, dy);
        System.out.println("I'm a square and I'm
moving!");

} }
```

Constructors

● The subclass constructor must call the superclass constructor first.

```
public abstract class Polygon { protected double x, y;
public Polygon(double x, double y) {

          this.x = x; this.y = y;
    }
}
public class Square extends Polygon {
    private double size;
    public Square(double x, double y, double size) {

super(x, y);

          this.size = size;
    }

}
```

Graphical User Interfaces (GUIs)

JavaFX Concepts

● A **node** is a graphical object (e.g. a Button, TextField, Label, GridPane).

● A **scene** is a tree of nodes.

● A **stage** is a place to display a scene (typically a window).

● An **application** has a main method. It sets up and shows the primary stage.

The scene graph

- A scene is a tree of nodes.
- Each node is either a branch or a leaf.
  - A branch node can have children e.g. GridPane, HBox, VBox
  - A leaf node cannot have children e.g. Button, Label, TextField

Packages to import

- **Nodes**:
  ```
  import javafx.scene.control.*; import
  javafx.scene.layout.*; import javafx.scene.text.*; import
  javafx.scene.image.*;
  ```
- **Scene**:
  ```
  import javafx.scene.*;
  ```
- **Stage**:
  ```
  import javafx.stage.*;
  ```
- **Application**:
  ```
  import javafx.application.*;
  ```

Branch nodes - VBox

- A VBox lays out its children in a vertical box.
- Create a VBox with 10 pixel spacing:

  ```
  VBox box = new VBox(10);
  ```

- Add the the children one by one:
  ```
  box.getChildren().add(usernameLbl);
  box.getChildren().add(usernameTf);
  box.getChildren().add(passwordPf);
  ```
- Or add many children at once:
  ```
  box.getChildren().addAll(loginBtn, flowerIv);
  ```

- Or Create a VBox with children:

```
VBox box = new VBox(10, usernameLbl, usernameTf, passwordPf, loginBtn,
flowerIv);
```

Branch nodes - HBox

- An HBox lays out its children in a horizontal box.
- `HBox box = new HBox(10);`

```
    box.getChildren().addAll(usernameLbl, usernameTf,
loginBtn, flowerIv);
```

- Align with setAlignment: `box.setAlignment(Pos.CENTER);`

Branch nodes - Alignment

- `import javafx.geometry.*; box.setAlignment(position);`
- Valid positions: ○ `Pos.CENTER`

  ○ `Pos.CENTER_LEFT` ○ `Pos.CENTER_RIGHT` ○ `Pos.TOP_CENTER`
  ○ `Pos.BOTTOM_CENTER` ○ `Pos.TOP_LEFT`

  ○ `Pos.TOP_RIGHT`
  ○ `Pos.BOTTOM_LEFT` ○ `Pos.BOTTOM_RIGHT`

Branch nodes - GridPane

- A GridPane lays out its children in a grid of rows and columns.
- Create a GridPane:

```
GridPane grid = new GridPane();
```

column row

● Add children to the grid: `grid.add(usernameLbl, 0, 0);`
`grid.add(passwordLbl, 0, 1); grid.add(usernameTf, 1, 0);`
`grid.add(passwordPf, 1, 1); grid.add(loginBtn, 1, 2);`

Application class

● The main class extends Application.

- It defines a main method.
- It overrides the start method.

```
public class BankApplication extends Application {
```

```
    public static void main(String[] args) { launch(args);
}
    @Override
    public void start(Stage stage) throws Exception {
        ... code to set up and show the stage ...
    }

}
```

The Observer Pattern

● **Phase 1 (registration)**: Each observer registers to be notified. **handle()**

Subject code:
public void addObserver(Observer o) {
observers.add(o);
}
Observer code:
subject.addObserver(this);

● **Phase 2 (notification)**: When something happens to the subject, notify the observers. Observer code:
public void handle() {
    do something in response
}
Subject code:

```
for (Observer o : observers)
    o.handle();
```

The Observer Interface

- An observer is any object that can handle the notification.
- Define an interface:

```
public interface Observer {
    void handle();
}
```

- An observer is any object that implements this interface.
- Each observer implements the `handle()` method to achieve its own goal.

Inner Classes

- An **inner class** is a class defined inside another class.
- An inner class can access all members of the outer class.
- An inner class offers **better encapsulation**:
    - x and foo can be hidden from the outside but shared with the inner class.
    - The inner class can also be hidden from the outside.

```
public class OuterClass {
    private int x;
    private void foo() { x++; };
    private class InnerClass {
        public void bar() {
            foo();
            System.out.println(x);
        }
} }
```

Anonymous inner classes

● An interface cannot be instantiated since it has no implementation: ✖`new ProductObserver()`

● However, you can provide the implementation while instantiating it: ✔`new ProductObserver() {`

```
        @Override public void handleSale(double money) {
            System.out.println("You paid $" + money);

} }
```

● Same as defining a class that implements the interface, then creating a new instance of that class.
Except the class has no name. Hence, it is "anonymous".

Lambda Expressions (Java 8)

● Anonymous inner classes with one method are very common.

```
    new ProductObserver() {
        @Override public void handleSale(double money) {
            System.out.println("You paid $" + money);
        }

}
```

● This is a LOT of syntax for just one method!
● A lambda expression is a shorter way to write such a method:

```
    money -> System.out.println("You paid $" + money)
```

● A body with one statement has no braces or semicolon: `money -> System.out.println("Sale: $" + money)`
● Curly braces enclose a block of code. Each statement has a semicolon: `money -> {`

```
            String moneyStr = formatted(money);

            System.out.println("Sale: $" + moneyStr);
        }
```

● Multiple parameters are enclosed in parentheses: `(param1, param2, param3) -> body`

Example

```
public class Store {
    private Product product;
    private CashRegister cashRegister;
    public Store() {

} }

product = new Product(); cashRegister = new CashRegister();
product.addObserver(cashRegister); product.addObserver(
money -> System.out.println("You paid $" + money)
); }}
```

Which one should I use?

- Use a lambda expression if the class has one method and is used once.
- Use an anonymous inner class if the class has multiple fields/methods.
- Use an inner class if you also need to create more than one instance.
- Use a normal class if you also need to access it from other classes (or if you anticipate needing to)

Event-driven programming

- An "event" is something that "happens" in a GUI application.
    - A button is clicked
    - The mouse is dragged
    - A menu item is selected
- GUI programs are entirely driven by events using the observer pattern.
    - Notify me when a button is clicked
    - Notify me when the mouse is dragged
    - Notify me when this menu item is selected
- The observers respond to events to achieve the program's goals.

Registering an observer

- Package: `import javafx.event.*;`
- Observer interface: `public interface EventHandler< X> { void handle(X event); }`

- `X` is the event type. e.g.:

○ `ActionEvent` - when a button is clicked or a menu item is selected
○ `KeyEvent` - when a key is pressed, released or typed

● Registering an observer: `loginBtn.setOnAction(observer);`
`usernameTf.setOnKeyTyped(observer);`

Registering an observer as an inner class

```
public class MyApplication extends Application {

private TextField usernameTf;
private PasswordField passwordTf;
@Override public void start(Stage stage) {

Button loginBtn = new Button("Login");
loginBtn.setOnAction(new LoginButtonHandler()); ...

}

private class LoginButtonHandler implements
EventHandler<ActionEvent> { @Override public void
handle(ActionEvent event) {

    if (checkPassword(usernameTf.getText(),
passwordPf.getText())
      ...

} } }
```

Registering as an anonymous inner class

```
public class MyApplication extends Application { private
TextField usernameTf;
private PasswordField passwordTf;
@Override public void start(Stage stage) {
Button loginBtn = new Button("Login");

loginBtn.setOnAction(new EventHandler<ActionEvent>() {
@Override public void handle(ActionEvent event) {

    if (checkPassword(usernameTf.getText(),
passwordPf.getText())
      ...} });

... } }
```

Registering as a lambda expression

```
public class MyApplication extends Application {
  private TextField usernameTf;
  private PasswordField passwordTf;
```

```
@Override public void start(Stage stage) { Button loginBtn =
new Button("Login");

loginBtn.setOnAction(event -> {
if (checkPassword(usernameTf.getText(), passwordPf.getText())

... }); ... } }
```

TextField getter/setter pattern

- A `TextField` has a getter that converts from a String.
    - Use `Integer.parseInt(s)` to convert the `String s` to an `int`.
    - Use `Double.parseDouble(s)` to convert the `String s` to a `double`.

- A `TextField` has a setter that converts to a String.

```
public class IncrementorApplication extends Application
{    private TextField valueTf;
     private int getValue() {
return Integer.parseInt(valueTf.getText() );
}
private void setValue(int value) {
     valueTf.setText("" + value); } }
```

Set the event handler (observer)

```
public class IncrementorApplication extends Application {
private TextField valueTf;
private int getValue() { return
Integer.parseInt(valueTf.getText()); } private void
setValue(int value) { valueTf.setText("" + value); }

@Override
 public void start(Stage stage) {
         ...
          incrementBtn = new Button("+1");
incrementBtn.setOnAction(event -> setValue(getValue() + 1)); }

}
```

- The event handler can access getValue/setValue from the outer class.

Model View Controller (MVC)

FXML

- Consensus: Programming languages are not good for laying out GUIs.
- Current trend: use a markup language.

- **FXML** is the JavaFX Markup language based on XML.
- Replace this Java code:

```
Label usernameLbl = new Label("Username:");
```

with this FXML code:

```
<Label text="Username:"/>
```

GridPane attributes

Attributes for GridPane:

- hgap sets the horizontal gap between child nodes.
- vgap sets the vertical gap between child nodes.

Attributes for children of GridPane:

- GridPane.columnIndex sets the column position of a child.
- GridPane.rowIndex sets the row position of a child.
- GridPane.columnSpan sets how many columns the child occupies.

Label vs Text

```
import javafx.scene.text.*;
```

- A <Label> is used to label a form input (e.g. a TextField, PasswordField, RadioButton, ...)
- A <Text> is to display free-standing text (e.g. a heading, informative text, error messages, ...)
- The text of a <Label> never changes.
- The text of a <Text> can be changed programmatically via its `setText()` method.

Model-View-Controller (MVC)

The MVC pattern splits a GUI program into 3 layers

- The models are Java objects that represent the data of your application and the operations on that data.
- The views are the components that represent the graphical user interface of your application. Views "observe" data in the models.
- The controllers are the components that handle user interaction. Controllers "observe" events that occur in the views.

Pattern #1: Immutable Property

- A property that never changes.
- Final getter. No setter.

```
public class SomeClass {
    private final int value;
    public SomeClass(int value) {
        this.value = value;
    }
    public final int getValue() { return value; }
}
```

Pattern #2: Read Write Property

- A property that is readable, writable and observable. ● Encapsulate the value in a property object.
- Final getter and setter.
- Property method called xProperty (where x is the name of the property).

```
public class SomeClass {
    private IntegerProperty value = new
SimpleIntegerProperty();

public SomeClass(int value) { this.value.set(value) ;

}
public final int getValue() { return value.get(); }
public final void setValue(int value) {
this.value.set(value); } public IntegerProperty
valueProperty() { return value; }
```

Pattern #3: Read Only Property

- A property that is readable and observable. Can be written by the class.
- Encapsulate the value in a property object.
- Final getter and optional **private** setter.
- Property method returns a read only property.

```
public class SomeClass {
    private IntegerProperty value = new
SimpleIntegerProperty();
    public SomeClass(int value) {
        this.value.set(value);
    }

public final int getValue() { return value.get(); }
private final void setValue(int value)
{ this.value.set(value); } public ReadOnlyIntegerProperty
valueProperty() {
```

```
return value; }
} }
```

Pattern #4: Immutable Property, Mutable State

- A property that is a reference to an object.
- The reference doesn't change, but the properties of the object can.
- Final getter. No setter.

```
public class Customer {
private Account account;

    public Customer() {
        account = new Account("Mr Smith");
    }
    public final Account getAccount() { return account; }
}
```

- Not possible: `customer.setAccount(new Account("Dr Smith"));`
- Still possible: `customer.getAccount().setName("Dr Smith");`

GUI Lists

Packages

- A package is a collection of related classes.
- Each application or library should be placed in its own package.
- To avoid two programmers using the same package name for their application, we follow a convention:
  - Companies use the reverse of their domain name.

  e.g. For domain name **`mycompany.com`**, use the package name **`com.mycompany`**
  - Different applications made by the same company are in sub-packages.

  e.g. **`com.mycompany.calculatorapp`** and **`com.mycompany.studyapp`**

ListView<X>

- A ListVew<X> displays a list of items of type X.
- Items can be either:
  - Strings
  - Objects that have a toString() function
- Create a ListView in FXML: `<ListView fx:id="accountsLv"/>`
- Create a ListView in Java:

```
ListView<Account> accountsLv = new ListView<Account>();
```

Setting preferred dimensions

- In FXML:

```
<ListView prefWidth="300" prefHeight="200"/>
```

- In Java:

```
accountsLv.setPrefWidth( 300);
accountsLv.setPrefHeight( 200);
```

Selecting a ListView item

Goal: The user selects an item from a ListView then clicks a button to perform an action on the selected item.

Solution: Set the onAction handler for the button to perform the following two steps:

1. Get the selected item (pattern)
2. Perform an action on that item

Linking a ListView to the model

● **Solution**: Use an ObservableList

```
public class Customer {
private ObservableList<Account> accounts = FXCollections.

observableArrayList() ;

  public void addAccount(String type) {
    accounts.add(new Account(type));

} }
```

● Observers are notified whenever the list contents changes.

ListView getter pattern

- A `ListView` has a getter that gets the currently selected item.
- It uses the `getSelectedItem()` method of the selection model.

```
public class CustomerController {
    @FXML private ListView<Account> accountsLv;
    private Account getSelectedAccount() {
        return
accountsLv.getSelectionModel().getSelectedItem();
} }
```

Getter/Setter patterns for controls

● It is good practice to define getters and setters to wrap the contolled value.
E.g.

```
    private String getGender() {
        return
genderTg.getSelectedToggle().getUserData().toString();
    }
    private boolean isAgree() {
```

```
            return agreeCb.isSelected();
        }
        private Account getAccount() {
            return
    accountsCmb.getSelectionModel().getSelectedItem();
    }
```

GUI Tables

FXML and Java code

- Creating a TableView in FXML:

```
<TableView fx:id="accountsTv" prefWidth="300" prefWidth="200">

            <placeholder><Label text="No
  accounts"/><placeholder>
              <columns>
                  <TableColumn text="Type"/>
                  <TableColumn text="Balance"/>
              </columns>
        </TableView>
```

- Declaring the TableView in your controller:

```
@FXML private TableView<Account> accountsTv;
```

Linking the TableView to the model

- Two ways to link the view and model
  - In FXML:

```
<TableView fx:id="accountsTv" items="  ${controller.customer.accounts} ">
```

  - In Java: `accountsTv.setItems( getCustomer().getAccounts() );`
- You must:
  - Expose a "`customer`" property in the controller
  - Expose an "`accounts`" property in the customer model

Linking each TableColumn to a model property

- Use a `PropertyValueFactory` to link the column to a property value:

```
    <?import javafx.scene.control.cell.*?>
...
                          <columns>
          <TableColumn text="Type">
            <cellValueFactory><PropertyValueFactory
  property="type"/></cellValueFactory>
          </TableColumn>
          <TableColumn text="Balance">
            <cellValueFactory><PropertyValueFactory
  property="balance"/></cellValueFactory>
          </TableColumn>
```

```
        </columns>
```

- You must expose the following properties in the account model:
  - `type`
  - `balance`

Setting a custom cell value factory

- Assign an id to the column:

```
<TableColumn fx:id="balanceClm" text="Balance"/>
```

- In your controller:

```
@FXML private TableColumn<Account, String> balanceClm; @FXML
private void initialize() {

            balanceClm.setCellValueFactory(cellData ->

    cellData.getValue().balanceProperty().asString("$%.2f"));

    }
```

- `TableColumn<Account,String>` means the item for this row is an Account, and the cell contents to be displayed is a String.

Any property can be observed for changes

- Print the account balance whenever it changes:

```
account.balanceProperty().addListener((obs, oldBal, newBal) ->
System.out.println("Balance changed from "+oldBal+" to
"+newBal));
```

- Print the text of a TextField whenever it changes:

```
nameTf.textProperty().addListener((obj, oldText, newText) ->
System.out.println("Text updated to " + newText));
```

- Print the selected toggle whenever it changes:

```
genderTg.selectedToggleProperty().addListener((o, old, now) ->
System.out.println("Selected gender: " + now));
```

Exceptions

- Sometimes a method can fail to do its job. In such situations, that method *throws* an "exception".
- To handle this error, the caller *catches* the exception.
- To know what types of exception a method might throw, refer to the Java

Try-with-resource (Java 7)

- A try-with-resource statement declares a resource that is auto-closed:

```
try (Scanner scanner = new Scanner(new File("data.txt")))
{ int a = Integer.parseInt(scanner.nextLine());
int b = Integer.parseInt(scanner.nextLine());
```

```
int c = a / b;
System.out.println(a + " / " + b + " = " + c); } catch
(Exception e) {
System.out.println("An error occurred: " + e.getMessage());
}
```

Throwing an exception

- If you write a method that can fail, consider declaring that method to throw an exception.
- Examples:
  - `public void withdraw(double amount) throws InsufficientFundsException`
  - `public void addAccount(String type) throws DuplicateAccountException`
  - `public void removeAccount(String type) throws NoSuchAccountException`
- You may define your own exception classes or use a generic exception.