Pattern Book

**Sum Pattern:**

Java

<type> sum = 0

<for each item> {

sum += <item>;

}

Python

Sum1 = sum(list1)

**Output Pattern:**

Java

System.out.println(“<label>” + <value>);

Python

print(“label “ + <value>);

**Read Pattern:**

Java

System.out.print(“<prompt>”);

<type><variable> = <read operation>;

Python

<variable> = <type>(input(‘<prompt>))

**Read Loop Pattern**

Java

<read pattern>

While (<value> != <end value>) {

<use the value>

<read pattern>

}

Python

<read pattern>

While (<value> != <end value>):

<use the value>

<read pattern003E

**Array Loop Pattern**

Java

For (int i = 0; i < <array>.length; i++) {

<use the item in the array [i]>;

}

Python

For index in range(0, len(array)):

<use the item array[index]>

**Count Pattern**

Java

int count = 0;

<for each item>

If (<guard>)

count++

Python

count = 0

<for each item>:

If <guard>:

count = count + 1

**Max Pattern**

<type> max = <smallest number>;

<for each item> {

If (<item> > max) {

max = <item>;

}

}

**Min Pattern**

<type> min = <largest number>;

<for each item> {

If (<item> < min) {

min = <item>;

}

}

**String Loop Pattern**

for (int i = 0; I < <str>.length(); i++) {

<use character str.charAt(i)>

}

**For each loop Pattern**

// For each word in array, print that word

for (String word: array) {

System.out.println(word);

}

**Read Function**

// The read pattern returns a value, so it is a function. It has the form read<x>

int readAge() {

System.out.println(“Age: ”);

return scanner.nextInt();

}

String readName() {

System.out.println(“Name: ”);

return scanner.nextLine();

}

**Merged read loop pattern (read loop using methods)**

double age;

while ((age = readAge()) != -1) {

<use age>

}

// Example: reading characters

char c

while ((c = readChar()) != ‘.’) {

<use c>

}

// Example: reading strings

String s

While (! (s = readString()).equals(“end”)) {

<use s>

}

**The “any” pattern**

// Determine if any item in a collection passes <test>

<for each item>

If (<item passes test>)

return true;

return false;

// Example if any number in an array is negative

boolean anyNegative (int[] array) {

for (int item: array) {

if (item < 0) {

return true;

}

return false;

}

}

**The “every” pattern**

// Determine if all items in a collection pass <test>>

<for each item>

if (! <item passes test>)

return false;

return true;

**The “none” pattern**

// Determine if no items in a collection pass <test>

<for each item>

if (<item passes test>)

return false;

return true;

**Methods: Functions vs Procedures**

1. A procedure does something. It’s name is a verb

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

Example:

public static void showCircleArea(double radius) {

double area = Math.PI \* radius \* radius;

System.out.println(“The area of the circle is ” + area);

2. A function returns something. It’s name is a noun

* A function is a method that returns a value
* A function should not have any side “effect”. 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.
* A function may also use local variables

Example:

public static double circleArea(double radius) {

double area = Math.PI \* radius \* radius;

return area;

}

**Relationship between procedures and functions**

* A procedure can call a function
* A procedure can call a procedure
* A function can call a function
* A function should not call a procedure

**Constructors**

// Initialise a new object

1. Initialise from literals

public class Account {

…

public Account() {

name = “Default name”;

type = “Savings”;

balance = “0.0”;

}

}

2. Initialise a new object with values read from the user (using a read pattern)

public class Account {

…

public Account() {

name = readName();

type = readType();

balance = readBalance();

}

}

3. Initialise from parameters

public class Account {

private String name;

private String type;

private double balance;

public Account(String name, String type, double Balance) {

// (this) refers to the field.

this.name = name;

this.type = type;

this.balance = balance;

}

}

**toString method (return a string representation of the object)**

// override the default toString() method

@Override

public String toString() {

return “The account has $” + balance;

}

**Formatted pattern (format to 2 decimal places)**

// Show to two decimal places

Import java.text.\*

@Override

public String toString() {

return “The account has $” + formatted(balance);

}

Private String formatted (double value) {

// 0 means always show a digit. # means show a digit if needed  
 DecimalFormat f = new DecimalFormat(“###,##0.00”);

Return f.format(value);

}

// using a toString method

1. Using another object’s toString method

**Explicitly**: System.out.println(janesAccount.toString());

**Implicitly**: System.out.println(janesAccount);

1. Using this object’s toString method

**Explicitly without this**: System.out.println(toString());

**Explicitly with this**: System.out.println(this.toString());

**Implicitly with this**: System.out.println(this);

**Menu Pattern**

public void use() {

// read choice until exit  
 char choice;

while((choice = readChoice()) != ‘x’) {

// execute an action

switch (choice) {

// one procedure for each action

case <first choice>: <first choice>();

break;

case <second choice>: <second choice>();

break;

default: <default method>();

break;

}

}

}

private <Datatype> readChoice() {

System.out.println(“Choice <d/w/s/x): ”);

return In.nextChar();

}

**Match pattern**

Public Boolean hasType (String type) {  
 return type.equals(this.type);

}

**Looping over Array/Linked list**

LindedList<String> list = new LinkedList<String>();

for (String word: list) {

System.out.println(word);

}

**Copying list**

1. LinkedList < datatype> original = new LinkedList <datatype>();

LinkedList <datatype> copy = new LinkedList <datatype>();

for (datatype element: original) {

copy.add(element);

}

1. LinkedList < datatype> original = new LinkedList <datatype>();

LinkedList <datatype> copy = new LinkedList <datatype>();

copy.addAll(original);

**Lookup pattern**

private Account account (String type) {

for (Account account: accounts) {

if (type.equals(account.getType())) {

return account;

}

}

return null;

}

**Find all matches pattern**

// find all words in a list that contains “z”

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 pattern**

// the zWords(list) on the right side is a function that removes all matches out of the list

1. LinkedList<String> zWords = zWords(list);

list.removeAll(zWords);

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

if (it.next().contains(“z”)) {

it.remove(); break;

}

}

// 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 in a list pattern**

// Stop loop after removing to avoid an exception

1. for (Sting word: list) {

if (word.contains(‘z’)) {

list.remove(word);

}

}

// Use an iterator

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

if (it.next().contains(“z”)) {

it.remove();

break;

}

}

**The observer patterns**

// observer are notified whenever a subject changes.

Examples:

* A button notifies you when its clicked
* A file notifies you when it is modified
* A product notifies you when its sold

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

Observer code:

subject.addObserver(this)

Subject code:

public void addObserver(Observer o) {

observers.add(o);

}

**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();

}

**Inner class**

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 outside but sharded with the inner class
* The inner class can also be hidden from the outside

Example:

public class OuterClass {

private int x;

private void foo() {

x++;

}

private class InnerClass {

public void bar() {

foo();

System.out.println(x);

}

}

}

Example 2:

public class Store {

private Product product;

private CashRegister cashRegister;

public Store() {

product = new product();

cashRegister = new CashRegister():

product.addObserver(cashRegister);

product.addObserver(new SalePrinter());

}

private class SalePrinter implements ProductObserver {

@Override

public void handleSale(double money) {

System.out.println(“You paid $” + money):

}

}

}

**Anonymous Inner class**

Provide the implementation while instantiating it

new ProductObserver() {

@Override

public void handleSale(double money) {

System.out.println(“You paid $” + money);

}

}

Example

public class Store {

private Product product;

private CashRegister cashRegister;

public Store() {

product = new product();

cashRegister = new CashRegister():

product.addObserver(cashRegister);

product.addObserver(new ProductObserver() {

@Override

public void handleSale(double money) {

System.out.println(“You paid $” + money):

}

});

}

}

**Lambda Expressions**

Anonymous inner classes with one method are very common

new ProductObserver() {

@Override

public void handleSale(double money) {

System.out.println(“You paid $” + money);

}

}

A lambda expression is a shorter way to write such a method:

* A body with one statement has no braces or semicolon:

money ->System,out.println(“You paid $” + money)

(method parameter) (method body)

* 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)

}

}

}

**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 event 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

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:

loginButton.setOnAction(observer);

usernameTf.setOnKeyTyped(observer);

Example – Registering an observer as an inner class

Import javafx.event.\*;

public class MyApplication extends Application {

private TextField usernameTf;

private PasswordField passwordTf;

@Override

public void start(Stage stage) {

Button loginButton = new Button(“Login”);

loginButton.setOnAction(new LoginButtonHandler());

…

}

private class LoginButtonHandler implements EventHandler<ActionEvent> {

@Override

public void handle(ActionEvent event) {

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

…

}

}

}

}

Example – Registering as an anonymous inner class

Import javafx.event.\*;

public class MyApplication extends Application {

private TextField usernameTf;

private PasswordField passwordTf;

@Override

public void start(Stage stage) {

Button loginButton = new Button(“Login”);

loginButton.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

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

…

}

}

});

…

}

}

Example – Registering as a lambda expression

Import javafx.event.\*;

public class MyApplication extends Application {

private TextField usernameTf;

private PasswordField passwordTf;

@Override

public void start(Stage stage) {

Button loginButton = new Button(“Login”);

loginButton.setOnAction(event -> {

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

…

}

});

}

}

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 usernameLb1 = new Label(“Username: ”);

With this FXML code

<Label text=“Username: ”/>

* Can use CSS to style with

<stylesheets>

<URL value=”@style.css”/>

</stylesheets>

* Assign a style class to a node with styleClass=“xyz ”
* Select nodes with style class “xyz” using the selector .xyz { … }
* Allows you to invent categories for selecting nodes from CSS.
* Use class FXMLLoader to load an FXML file

@import javafx.fxml.\*;

FXMLLoader loader = new FXMLLoader(getClass().getResource(“login.fxml”));

Parent root = loader.load();

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 view 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.

Immutable Property pattern

* 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 this.value;

}

}

Read Write Property pattern

* 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 SimpleIntergerProperty();

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;

}

}

Read Only Property pattern

* A property that is readable, writable and observable.
* 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 SimpleIntergerProperty();

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;

}

}

Immutable property, mutable state pattern

* 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”));
* Possible: customer.getAccount().setName(“Dr Smith”);

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 accounts.Lv.getSelectionModel().getSelectedItem();

}

}

Opening a window with ViewLoader

Import au.uts.edu.ap.javafx.\*

ViewLoader.showStage(<model>, <fxml>, <title>, <stage>);

ChangeListener Pattern

productsTv.getSelectionModel().selectedItemProperty().addListener ((observable, oldProduct, newProduct) -> viewBtn.setDisable(newProduct == null));