Laborator 7

1. Closure and callback

**interface** Incrementable {

**void** increment();

}

// Very simple to just implement the interface:

**class** Callee1 **implements** Incrementable {

**private** **int** i = 0;

**public** **void** increment() {

i++;

System.*out*.println(i);

}

}

**class** MyIncrement {

**public** **void** incr() { System.*out*.println("Other operation"); }

**static** **void** f(MyIncrement mi) { mi.incr(); }

}

// If your class must implement increment() in

// some other way, you must use an inner class:

**class** Callee2 **extends** MyIncrement {

**private** **int** i = 0;

**public** **void** incr() {

**super**.incr();

i++;

System.*out*.println(i);

}

**private** **class** Closure **implements** Incrementable {

**public** **void** increment() {

// Specify outer-class method, otherwise

// you’d get an infinite recursion:

Callee2.**this**.incr();

}

}

Incrementable getCallbackReference() {

**return** **new** Closure();

}

}

**class** Caller {

**private** Incrementable callbackReference;

Caller(Incrementable cbh) { callbackReference = cbh; }

**void** go() { callbackReference.increment(); }

}

**public** **class** Callbacks {

**public** **static** **void** main(String[] args) {

Callee1 c1 = **new** Callee1();

Callee2 c2 = **new** Callee2();

MyIncrement.*f*(c2);

Caller caller1 = **new** Caller(c1);

Caller caller2 = **new** Caller(c2.getCallbackReference());

caller1.go();

caller1.go();

caller2.go();

caller2.go();

}

}

1. Application framework – control framework

**public** **abstract** **class** Event {

**private** **long** eventTime;

**protected** **final** **long** delayTime;

**public** Event(**long** delayTime) {

// constructorul captureaza timpul masurat ca si timp trecut de la momentul creerii obiectului

**this**.delayTime = delayTime;

start();

}

**public** **void** start() { // Allows restarting

eventTime = System.*nanoTime*() + delayTime;

}

**public** **boolean** ready() {

**return** System.*nanoTime*() >= eventTime;

}

**public** **abstract** **void** action();

}

**import** java.util.\*;

**public** **class** Controller {

// A class from java.util to hold Event objects:

**private** List<Event> eventList = **new** ArrayList<Event>();

**public** **void** addEvent(Event c) { eventList.add(c); }

**public** **void** run() {

**while**(eventList.size() > 0)

// Make a copy so you’re not modifying the list

// while you’re selecting the elements in it:

**for**(Event e : **new** ArrayList<Event>(eventList))

**if**(e.ready()) {

System.*out*.println(e);

e.action();

eventList.remove(e);

}

}

}

//This produces a specific application of the

//control system, all in a single class. Inner

//classes allow you to encapsulate different

//functionality for each type of event.

**public** **class** GreenhouseControls **extends** Controller {

**private** **boolean** light = **false**;

**public** **class** LightOn **extends** Event {

**public** LightOn(**long** delayTime) { **super**(delayTime); }

**public** **void** action() {

//Put hardware control code here to

//physically turn on the light.

light = **true**;

}

**public** String toString() { **return** "Light is on"; }

}

**public** **class** LightOff **extends** Event {

**public** LightOff(**long** delayTime) { **super**(delayTime); }

**public** **void** action() {

//Put hardware control code here to

//physically turn off the light.

light = **false**;

}

**public** String toString() { **return** "Light is off"; }

}

**private** **boolean** water = **false**;

**public** **class** WaterOn **extends** Event {

**public** WaterOn(**long** delayTime) { **super**(delayTime); }

**public** **void** action() {

//Put hardware control code here.

water = **true**;

}

**public** String toString() {

**return** "Greenhouse water is on";

}

}

**public** **class** WaterOff **extends** Event {

**public** WaterOff(**long** delayTime) { **super**(delayTime); }

**public** **void** action() {

//Put hardware control code here.

water = **false**;

}

**public** String toString() {

**return** "Greenhouse water is off";

}

}

**private** String thermostat = "Day";

**public** **class** ThermostatNight **extends** Event {

**public** ThermostatNight(**long** delayTime) {

**super**(delayTime);

}

**public** **void** action() {

//Put hardware control code here.

thermostat = "Night";

}

**public** String toString() {

**return** "Thermostat on night setting";

}

}

**public** **class** ThermostatDay **extends** Event {

**public** ThermostatDay(**long** delayTime) {

**super**(delayTime);

}

**public** **void** action() {

// Put hardware control code here.

thermostat = "Day";

}

**public** String toString() {

**return** "Thermostat on day setting";

}

}

// An example of an action() that inserts a

// new one of itself into the event list:

**public** **class** Bell **extends** Event {

**public** Bell(**long** delayTime) { **super**(delayTime); }

**public** **void** action() {

addEvent(**new** Bell(delayTime));

}

**public** String toString() { **return** "Bing!"; }

}

**public** **class** Restart **extends** Event {

**private** Event[] eventList;

**public** Restart(**long** delayTime, Event[] eventList) {

**super**(delayTime);

**this**.eventList = eventList;

**for**(Event e : eventList)

addEvent(e);

}

**public** **void** action() {

**for**(Event e : eventList) {

e.start(); // Rerun each event

addEvent(e);

}

start(); // Rerun this Event

addEvent(**this**);

}

**public** String toString() {

**return** "Restarting system";

}

}

**public** **static** **class** Terminate **extends** Event {

**public** Terminate(**long** delayTime) { **super**(delayTime); }

**public** **void** action() { System.*exit*(0); }

**public** String toString() { **return** "Terminating"; }

}

}

**public** **class** GreenhouseController {

**public** **static** **void** main(String[] args) {

GreenhouseControls gc = **new** GreenhouseControls();

// Instead of hard-wiring, you could parse

// configuration information from a text file here:

gc.addEvent(gc.**new** Bell(900));

Event[] eventList = {

gc.**new** ThermostatNight(0),

gc.**new** LightOn(200),

gc.**new** LightOff(400),

gc.**new** WaterOn(600),

gc.**new** WaterOff(800),

gc.**new** ThermostatDay(1400)

};

gc.addEvent(gc.**new** Restart(2000, eventList));

**if**(args.length == 1)

gc.addEvent(**new** GreenhouseControls.Terminate(**new** Integer(args[0])));

gc.run();

}

}

1. In GreenhouseControls adaugati un eveniment de tip Event care sa realizeze operatia fan – on si off pentru o variabila fan. Configurati clasa GreehouseControler sa utilizeze noul tip de eveniment
2. Mosteniti din clasa GreenhouseControls pentru a adauga noi evenimente care sa faca tun on si turn off pe water\_mist . scrieti o noua versiune a GreenhouseController care sa utilizeze si acest nou tip de eveniment creat.
3. Mostenire din clasa inner

**class** WithInner {

**class** Inner {}

}

**public** **class** InheritInner **extends** WithInner.Inner {

//! InheritInner() {} // Won’t compile

InheritInner(WithInner wi) {

wi.**super**(); // aici outerclassreference.super()

}

**public** **static** **void** main(String[] args) {

WithInner wi = **new** WithInner();

InheritInner ii = **new** InheritInner(wi);

}

}