Table of contents

1 Synopsis

The purpose of this assignment is to: (0) Introduce you to the concept of "embedded systems", which can consist of microprocessors that are embedded in larger systems in order to control or sense physical devices; (1) Introduce you to "distributed programming", where several different computers (or microprocessors) are executing simultaneously, with some form of communication and synchronization between them; (3) Tell you about *execution threads* in Java, a feature that allows a Java program to consist of several program segments executing in parallel.

For this assignment, you will write code to simulate the cruise control system of a car. The job of our (simplified) cruise control mechanism is to attain and maintain a preset speed in spite of changes in road grade (i.e., in spite of whether the road is flat or uphill). One limitation of the cruise control mechanism that you will implement is that it will not attempt to maintain the preset speed if the car is headed downhill. In general, the cruise control mechanism that you will implement changes the speed of the car by turning the accelerator "on" (i.e., pressing the accelerator pedal down) to increase speed or turning the accelerator "off" (i.e., releasing the accelerator) to stop increasing the speed.

2 Preliminaries

Due: At the beginning of lecture on Tuesday, 24 November. You may turn it in before that date in the Carpenter consulting room. Do not turn it in at Carpenter on the due date.

Working with another person. You may do this assignment with one other person. However, you must share the work equally: write the code together, type it in together, and debug it together. Hand in only one assignment with both names on it.

Note. The first comment of your program must contain the name, Cornell ID, section day, section time, section number, and section instructor of each person who is to receive a grade for the assignment you hand in. Don't write it in by hand; type it in as the first comment of the file that contains class `CruiseControl`.

Grading. Section 10 describes three main tasks that you have to perform — each one requires writing code and testing it. Each part will be graded independently for both style (2 points) (i.e., clear logic, good commenting, good indentation) and for correctness (between 2 and 4 points).

What to hand in. Hand in a copy of class `CruiseControl`, which will contain all the code you will write. Also hand in a snapshot of one execution of your program that shows output that resulted from your program writing in the system output window.

3 Files to retrieve

Browse the CS100A home page and find the material for assignment 9. You will find:

(a) A link to a version of our completed CruiseControl program that you can execute from your browser. Execute this to become familiar with the simulation. Try different things —speeding up the clock by reducing the period, making the desired speed the same as the current speed, changing the road grade, and so forth.

NOTE. The screen that you see when you execute this program will look slightly different from the screen you get with your program. This program is a Java applet; your program is a Java application. On this screen, each of the little windows may have a message at the bottom, warning you that this applet may not be safe. Also, we have had to make up our own "System output" window, using a Frame with a TextArea, since one does not come automatically with an applet the way it does with an application.

(b) Eight files: SystemClock.java, SystemOutput.java, ClockedFrame.java, Accelerator.java, CurrentSpeed-.java, DesiredSpeed.java, RoadGrade.java, CruiseControl.java.

Option 1: You should start a new CodeWarrior project using the CUCS Java stationery (not the CUCS Graphics stationery). Then, replace file CUCSApplication.java in that project by CruiseControl.java. Also, move copies of the other other files mentioned in (b) into this project folder and then add them to your project. **Finally, change the preferences so that CodeWarrior knows that method `main` is in class `CruiseControl` instead of class `Application`.** This is VERY important to do; otherwise the code will not run. The first day handout on CodeWarrior has instructions for how to do this.

Option 2: Alternatively, we will give the consultants in Carpenter two floppies — one that contains the student version of the program (the whole project) for the Mac and another that contains the student version for the PC. You can then just copy the whole project onto your own floppy rather than downloading the files and setting up a project.

After following option 1 or option 2, you can compile and execute the program. It should run but it won't perform suitably until you fill in the missing sections in class `CruiseControl`.

4 Microprocessors

Microprocessors are small computers, usually embedded in other systems. Microprocessors are used to monitor and regulate the behavior of many common commercial products. For example, dedicated microprocessor systems are used in cars to regulate the fuel-air mixture for combustion and prevent skidding (using anti-lock brake systems). Around the home, they appear in appliances such as cellular phones, microwave ovens and thermostats for heating and cooling systems. They are used in modems, laser printers, graphics boards, and disk drives. More exotic applications include medical instrumentation, detection of collision and decay events in high energy particle accelerators, and regulation of nuclear reactors.

Microprocessors appear in settings where *real-time computing* is necessary: responses to external signals or changes —which can come at unpredictable times— must take place within a specific amount of time. Some microprocessors must also be able to generate electrical signals that can be used to control other devices.

The anti-lock brake system on a car is a good example of the challenging nature of real-time computing applications. Here, the microprocessor must detect changes in the rotational rates of the wheels characteristic of a skid. For example, a rapidly rotating wheel might abruptly stop rotating. If the angular velocity of the wheels increases too fast, the tire is slipping on the pavement and the car may be skidding; controlling the situation may be difficult for the driver. Detecting and correcting this situation involves monitoring the rate of change of the angular velocity (i.e. the angular acceleration), not just the angular velocity itself. Upon detecting this situation, the microprocessor must generate a sequence of output pulses that are used to control the hydraulic brake system, allowing the wheels to intermittently rotate to prevent a skid, thus making the car easier to control. Clearly, quick response[1] to inputs from external transducers is essential in this application. Not only must the response be quick, it must be appropriate for all imaginable driving conditions. For example, it would unacceptable for the microprocessor to interfere with normal braking operations at slow speeds.

Most real-time programming is done in assembly language or a programming language like C (or C++) that allows the programmer to explicitly manipulate the contents of specific memory locations. This is because communication with a microprocessor is often done through fixed memory locations. The real-time microprocessor itself is often programmed in its native assembly language. The machine instructions are

[1] On the order of tens of milliseconds.

placed in the microprocessors memory, which is physically distinct from the memory system of the host computer, so that the instructions can be executed autonomously by the real-time processor system.

5 Real-time simulation

A real-time application may consist of a host computer that communicates with several microprocessors, each of which senses some physical property (like the temperature in a room, or the speed at which a wheel is spinning) or controls some device. The host and an individual microprocessor communicate through a "register": a location in the microprocessor's memory. This location is "mapped" onto a fixed location in the host processor's memory, so that when the host references that memory location, it is really referencing the microprocessor's register. This is why many real-time applications need to read and write specific memory locations.

The microprocessor usually has a "program status word", or *psw*, whose bits are used to indicate the status of the microprocessor. For example, one of the bits, say `C`, might be used as follows: When the host computer is ready to read a value from the microprocessor's register, it (0) sets bit `C` to 1, (1) waits until `C` is set to 0 (by the microprocessor, to indicate that a new value is in its register), and then reads the register. This is how the host computer and the microprocessor communicate and synchronize.

Because of its built-in security measures, Java lacks the ability to manipulate the contents of specific memory addresses explicitly, so it is not well suited to real-time programming. However, Java does provide the tools needed to simulate a real-time computing environment. We have developed a real-time simulation program, which simulates a cruise control system, to introduce you to real-time computing. In it, we simulate this aspect of communication using a method `readValue()`.

6 Simulating a cruise control system

On the CS100A home page, you can find program `CruiseControl`, which you can execute from the web. Start it. Besides a system output window, five small windows will appear, each of which represents one component of the cruise control system. (To see the full title on a window, you may have to make the window wider.)

1. **A clock**. You can see the clock "ticking" away. The period 1000 means that the clock ticks every 1,000 milliseconds, or every 1 second. You can change this to a smaller or larger number of milliseconds by clicking in that text field, typing a new number, and pressing button `Read period`. The program won't allow a period of less than 100 milliseconds).

2. **The road grade** is set initially at 3 degrees — slightly uphill — since this is Ithaca. Typically, a microprocessor would be attached to a sensor of some sort that would detect the road grade. In our simulation, you, the user, can change the road grade by typing a new integer and pressing button `Read grade`. The program won't allow a grade steeper than +/- 10 degrees.

3. **The accelerator** will be switched on or off by the program as needed. This window just displays its status. There is also a button that you can press to turn output on or off. Press it; thereafter, at each clock tick, a single line of output is printed in the system output window, telling you the status of the accelerator, the change in the current speed due to the accelerator being on, the change in speed due to the road grade, the total speed change for this tick, and finally the new current speed.

4. **The desired speed** is set initially at 65. You can change the desired speed, just as you can change the road grade. Go ahead; raise it or lower it.

5. **The current speed** is initially 50 because we just merged into faster traffic and want to let the cruise control bring us back up to speed.

> The current speed is simulated by the program; how much it changes at each clock tick depends on whether the accelerator is on, the difference between the current and desired speeds, and the road grade.

Spend some time experimenting with this cruise control simulation. Become familiar with the five components on the screen.

7 Threads in Java

Typically, several programs may be executing on a computer at one time. Suppose you are working on your laptop and it is connected to a printer and a modem. One program on your computer could be printing something, another one could be faxing a file, a third could be computing something in the background (perhaps you are on the team to crack an RSA code), and a fourth could be some game that you are playing. Thus, (at least) four programs are executing at the same time on your laptop. Each is called an *execution thread*.

There is only one CPU (central processing unit) on your computer, so the four programs are not really executing simultaneously. Instead, your computer allocates a bit of execution time to each of these threads, giving each a turn. Your operating system uses some sort of allocation scheme to decide which thread should be the next to execute and how much time it should get. This switching is so fast and frequent that it gives the illusion of simultaneous execution.

These threads (of execution) can be independent (e.g. the printer program and your game don't interact), in which case allocation is fairly easy.

But sometimes, threads have to communicate with each other and perhaps synchronize in some fashion. For example, one thread may be responsible for maintaining windows on the screen, so when your Java program — which has at least one thread of execution — draws something in a window, behind the scenes it communicates with the thread that maintains the window.

In Java, synchronization is provided using three features. Suppose c is a class instance that is a thread. Then, execution of `c.wait()` in another thread b (say) tells the system to stop executing thread b until thread c executes a statement `notifyAll`. In other words, execution of `c.notifyAll` tells all threads that are waiting on c that they can now continue.

There is also a "synchronize" property, which can be attached to methods or to individual statements; it says that all other processes must be deterred from executing code in this class until this method or statement has finished — other processes are locked out.

Java provides facilities for declaring and using different threads of execution. We make use of this facility in our cruise control simulation. Each of the five components mentioned in Sect. 6 is a separate thread. And there is a sixth thread, the main program itself, which controls the five components.

So, in our Java simulation, we really have six programs —or six threads of execution— executing simultaneously.

[Note: You are not responsible for knowing about threads and how to use them. You will not have to write any code that deals directly with statements to manage and synchronize threads. Threads are discussed here because our program needs them and because you should be familiar with the concept of parallel execution — of having several threads executing simultaneously. Nevertheless, this background will give you some understanding of the code in this program and also should enlighten you a bit on an important topic in computer science, distributed computing.]

8 The SystemClock

File SystemClock.java simulates a clock. At the end of the line `public class SystemClock...` appears the phrase `implements Runnable`. This is Java's way of indicating that this class has a method `run()`, which should be called by the system to execute a class instance as a thread. We'll look at `run` later.

So, instances of this class are separate threads of execution.

The class has two variables that deal with the clock itself: `counter` contains the number of clock ticks that have happened thus far; `period` is the number of milliseconds to wait between clock ticks.

Four fields are components that go in the Frame (which you see on the screen): two Labels, a Textfield, and a Button.

Look at the constructors for `SystemClock` (including method `initialize`). In the game Checkers, we used layout manager `GridBagLayout` to place components in the Frame. Here, because our Frame layout is so much simpler, we can use the default layout manager for Frames, `BorderLayout`. This layout manager allows us to place components in five places: North, West, Center, East, and South:

<table>
<tr><td colspan="3" align="center">North</td></tr>
<tr><td>West</td><td>Center</td><td>East</td></tr>
<tr><td colspan="3" align="center">South</td></tr>
</table>

In method `initialize`, you can see the four calls on method `add`, which place the four components in the Frame.

Two methods in `SystemClock` are fairly obvious: `reset` sets the clock back to zero, and `ticks` is used by other processors to get the clock's value.

Method `run` is of special significance. The main program (method `main` of `CruiseControl`), calls `cThread.start()`, where `cThread` is the variable that contains the instance of `SystemClock`. In turn, method `start` will call `run` to start execution of this thread.

Method `run` is basically a loop in which each iteration does the following:

- Make the thread sleep for `period` milliseconds. Then wake up.
- Notify all the other threads that may be waiting on this one (for example, to read the tick) that the thread is now awake.
- Increase the time counter (and display it properly in the Frame).

The code to notify the other processes should be performed only when no other process is referencing variables associated with this thread. This "lock out" is indicated by the phrase **synchronize**. We don't go into any more detail here on this notion of synchronization.

Finally, take a look at method `action`, which is called when a button is pressed. There is only one button in the Frame, so there is no need for code to determine what the action was. The pressing of the button is processed by (0) reading `TextField periodField`, (1) trimming whitespace from both ends of it, (2) converting this `String` value to an integer, and (3) finally storing the integer in `period` (if the value is at least 100).

9 CurrentSpeed

We discuss just two other classes in this project, because the others are similar.

First, an instance of class `ClockedFrame` is a thread of execution that synchronizes with the clock. Thus, within this class, you see a method `run`, which is a loop that at each iteration synchronizes with the clock. That is all.

Also, this class is a Frame on the monitor screen. This class is a superclass of `CurrentSpeed` as well as of several other classes, so an instance of `CurrentSpeed` is an executable thread.

Now turn to subclass `CurrentSpeed`. Besides some components for the Frame (two labels), there is only one variable, `speed`, which contains the current car speed in the simulation. The constructor just constructs the Frame, shows it, and returns; there is nothing special about it.

Methods `readValue` and `setValue` can be called to reference and set the current speed. Classes `DesiredSpeed` and `RoadGrade` are nearly identical. Class `Accelerator` is only slightly more complicated, with two additional methods for turning the accelerator "on" and "off".

10 Your job in this assignment

We have given you five processes, which simulate a clock, the road grade, the accelerator, a desired speed, and a current speed. Your job is to write the program that simulates the entire cruise control process using these classes.

All your changes will be made to class `CruiseControl`. Your code will communicate with the other processes using their methods `ticks()`, `readValue()`, `setValue(double t)`, `turnOn()`, and `turnOff()`.

Controlling the accelerator

One of your tasks is to write the bodies of methods `checkTurnOff` and `checkTurnOn`, which should turn the accelerator off and on depending on the desired speed and the current speed.

There is one problem here. If you program the accelerator to turn on and off based simply on whether the desired speed is less than or greater than the current speed, then the accelerator may switch on or off at almost every single clock tick when the desired and current speeds are almost equal. You don't want your accelerator doing that. So, you must program in something that doesn't allow this to happen —how you do it is up to you.

[You could save this little problem for last; first program in the obvious; then, when the program is all checked out except for this problem, fix this problem.]

Changing the current speed

A second task is to write the bodies of the two methods `ChangeDueToGrade` and `ChangeDueToAccel`. Make sure you implement code that does precisely what the specification says to do.

The first task is easy: Method `ChangeDueToGrade` should update the current speed by some amount proportional to the road grade. Going uphill decreases the speed; going downhill increases the speed.

The second task is harder, and we leave it to you to do something reasonable. Method `ChangeDueToAccel` should update the current speed based on the state of the accelerator. Here are some constraints. Add 1 to the current speed if (and only if) the accelerator is on. But you should also add some amount that depends on the difference between the desired and current speeds. The additional speed change should be proportional to that difference. You can decide on the constant of proportionality. Make it reasonable. For example, if the accelerator is on, it should be able to hold to the current speed when the road grade is 10. If the current speed keeps dropping, perhaps the accelerator isn't powerful enough. If the accelerator is off, just subtract some small amount from the speed, say 0.3 mph.

Note that in this simulation, the road grade and accelerator affect the speed of the car independent of one another.

Processing after a clock tick

You must also write the code that changes the state of the accelerator at each clock tick. This is the body of method `CruiseControl.main`.

Here, you should write an infinite loop that (0) waits for a clock tick to occur — call method `CruiseControl.sleep` for this — and then (1) does whatever processing is necessary to update the cruise control system. Note that this should include printing data to the output window using System.out.print if the state of the system calls for it. It will be best if printing data to the output window is done via a separate method (and method main just makes a call to it).

The design of this portion of the program is up to you. Naturally, you can/should make use of the methods that you wrote for the previous tasks.

11 Extra credit

If you find this assignment too simple, try the following addition to it, for 3 extra points. Add a new thread (and its associated window) for a brake. If the user puts the brake on, the car should come to a halt in a reasonable number of seconds, changing all of the affected cruise control components as it does so.