Domain model: Manage a roster for a shop, that is, the assignment of staff for a shop to shifts for a working week. The shop is open for a single continuous period each day, the "working hours". Each day is divided into a number of shifts. A shift is a continuous period of time that has staff assigned to it for the entire period. Each shift has a manager, and zero or more workers. There is a minimum number of workers required for each shift (possibly zero). Shifts with fewer than this number are considered understaffed. Shifts with more than this number are considered overstaffed. Only registered staff can be assigned to a shift.

My design fulfills this domain model. The program has a function for adding working hours to each day, as well as adding shifts to each day. Each shift will track the amount of staff and the manager. The program can also display all the understaffed and overstaffed shifts by invoking the specific method. A staff member also has to be registered before they are assigned to a shift.

Objects creations expected with each method invocation:
newRoster: 1(2 if error), **setWorkingHours:** 1(2 if error), **addShift:** 3(2 if error), **registerStaff:** 1(2 if error), **assignStaff:** 0, **getRegisteredStaff:** 2, **getUnassignedStaff:** 3, **shiftsWithoutManagers:** 3, **understaffedShifts:** 3, **overstaffedShifts:** 3, **getRosterForDay:** 6, **getRosterForWorker:** 5, **getShiftsManagedBy:** 5, **reportRosterIssues:** 0, **displayRoster:** 0.

Errors:
addShift: checks for invalid day, start and end time overlap and if they are valid or not (all throw InvalidTimeException if error is found).
registerStaff: checks if either name is empty or null (throws InvalidNameException).
setWorkingHours: same as addShift.

The program throws the checked exceptions InvalidTimeException and InvalidNameException if an error is found. In my design the program throws the exception in the constructor. Every time a new object is potentially created, the program will check if the arguments passed into the constructor are valid or not. If yes, the object will be created and exception will not be thrown. However, if there are any errors, the program will halt and the exception will be thrown which will then be caught in the invoking method. The invoking method will then return the error message the exception object contains.

Explaining class choices:
I chose to have the Shift class extend the DateTime class as the Shift class needs the exact fields and methods that the DateTime class. It just needed a few more extra fields and methods that detail the staff and managers. And for both objects we needed a day and start and end time. So it made logical sense to create two classes and have one extend the other and include the extra members.

The StaffLastNameComparator and ShiftComparator was an easy solution to be able to sort the staff and shift lists.

There are two custom exceptions classes: InvalidTimeException and InvalidNameException. I created these as they are more expressive in telling the user what was wrong if they got an error.

The FormatStaff and FormatShift classes takes a parameter of List<Staff/Shift> type and processes it in various ways to format the list correctly for display. These classes help in increasing the readability of the program as when it was all in the roster class, it was too long and had too many methods to keep track of. These classes also makes it easier to change the program in the future if we wanted to change how the lists are displayed e.g. sorted by given names instead of family names or changing the format of start and end time display.