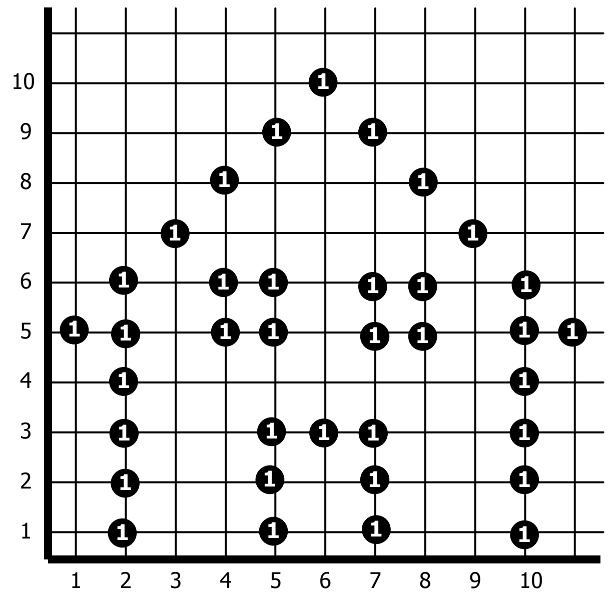
AP Computer Science Name: ________________________________
Chapter 4 Notes (3)

OBJECT ORIENTED DESIGN
(CLIENTS AND SERVERS)

We have learned techniques for designing a single class. We learned to ask what services (methods) are needed from that class and designing complex tasks as a decomposition into simpler tasks. Now we are going to look at design from a broader perspective. Here we will learn to recognize that we may need several classes of robots to carry out some task, and these robots may need to cooperate in some way. We will discuss only the design issues, leaving implementation until we have seen some more powerful ideas.

Suppose that we want to build a robot house as shown below. Houses will be built of beepers. In the real world, house building is a moderately complex task that is usually done by a team of builders, each with his or her own specialty. Perhaps it should be the same in the robot world.



If you look back at some of our earlier examples, you will find that there are really two kinds of instructions. The first kind of instruction, such as `pickTwoRows()` in the `Harvester` class is meant to be used in the main task block. The other kind, like `turnAroundRight()` and `turnAroundLeft()`, is meant primarily to be used internally, as part of problem decomposition. For example, the instruction `turnAroundRight()` is unlikely to be used except from within other instructions. The first kind of instruction is meant to be public and defines in some way what the robot is intended to do. If we think of a robot as a *server*, then its *client* (the main task block, or perhaps another robot) will send it a message with one of its public instructions. The client requests that service. The place to begin design is with the public services and the servers that provide them. The server robot itself will then, perhaps execute other instructions to provide the service. We can use successive refinement to help design the other instructions that help the server carry out its service.

The easiest way to get a house built is to call on the services of a `Contractor`, who will assemble some appropriate team to build our house. We tell the contractor robot `buildHouse()` and somehow the job gets done. The client doesn't especially care how the contractor carries out the task as long as the result (including the price) are acceptable. Notice that we have just given the preliminary design for a class, the `Contractor` class, with one public method: `buildHouse()`. We may discover the need for more methods and for more classes as well. We also know that we will probably need only a single robot of the `Contractor` class.

What does our `Contractor` need to do to build a house? One possibility is to place all of the beepers itself. But another way is to use specialist robots to handle well-defined parts of the house: for example, walls, the roof, and the doors and windows. The contractor will `delegate()` the work to other robots. Well, since we want the walls to be made of bricks (beeper-bricks), we should call on the services of one or more `Mason` robots. The door and windows could be built by a `Carpenter` robot, and the roof build by a `Roofer` robot.

The contractor needs to be able to gather this team, so we need another method for this. While it could be called by the client, it probably should be called internally by the contractor itself when it is ready to begin construction. The contractor also needs to be able to get the team to the construction site. We will want a new `Contractor` method `gatherTeam()`.

Focusing, then, on the smaller jobs, the `Mason` robot should be able to respond to a `buildWall()` message. The contractor can show the mason where the walls are to be built. Similarly, the `Roofer` should be able to respond to `makeRoof()`, and the `Carpenter` robots should know the messages `makeDoor()` and `makeWindow()`. We might go a step farther with the roofer and decide that it would be helpful to make the two gables of the roof separately. So would also want a `Roofer` to be able to `makeLeftGable()` and `makeRightGable()`.

However, we can also make the following assumption. Each specialized worker robot knows how to do its own task and also knows the names of the tasks and subtasks that it must do. If the contractor simply tells it to `getToWork()`, then it already knows what is should do. Therefore, we can factor out this common behavior into a Java interface named `Worker`.

```
public interface Worker
{
     public void getToWork();
}
```

(See last page for explanation on interfaces)

The classes then implement this interface and provide an implementation of the new method. Here we can see outlines of the new methods. Each class implements `getToWork` in a different way. We also mark each method telling whether it is `public` or `private`. A `public` method may be called from anywhere that the robot is known. A `private` method may only be called within the class definition of the robot itself. Private methods are "helper" methods to carry out the public services.

```
public class Mason extends UrRobot implements Worker
{
      // Constructor omitted

      public void getToWork()
      {
            buildWall();
            ...
            buildWall();
      }

      private void buildWall()
      {...}
}

public class Carpenter extends UrRobot implements Worker
{
      // Constructor omitted

      public void getToWork()
      {
            makeWindow();
            ...
            makeWindow();
            ...
            makeDoor();
      }

      private void makeWindow()
      {...}

      private void makeDoor()
      {...}
}

public class Roofer extends UrRobot implements Worker
{
      // Constructor omitted

      public void getToWork()
      {
            makeRoof();
      }

      private void makeRoof()
      {
            makeLeftGable();
            makeRightGable();
      }

      private void makeLeftGable()
      {...}

      private void makeRightGable()
      {...}
}
```

This gives us an outline for the helper classes. Let's look at the `Contractor` class. Since the team of builders is assembled by the contractor it must know their names. Therefore, it would be useful if the names of the helpers were declare as private names in the `Contractor` class itself, rather than global names. This will also effectively prevent the client of the contractor from telling the workers directly what to do.

```
class Contractor extends UrRobot
{
     private Mason kenMason = new Mason(...);
     private Roofer sueRoofer = new Roofer(...);
     private Carpenter lindaCarpenter = new Carpenter(...);

     private void gatherTeam()
     {
          ...// messages here to initial positioning of the team.
     }

     public void buildHouse()
     {
          gatherTeam();
          kenMason.getToWork();
          sueRoofer.getToWork();
          lindaCarpenter.getToWork();
     }
}

public static void main(String[] args)
{
     Contractor kristin = new Contractor(1, 1, East, 0);
     kristin.buildHouse();
     kristin.turnOff();
}
```

Note that the `Contractor kristin` is a server. Its client is the main task block which delivers message to it. But note also that `kristin` is a client of the three helpers since they provide services (wall services...) to `kristin`. In fact, it is relatively common in the real world for clients and servers to be mutually bound. For example, doctors provide medical services to grocers who provide food marketing services to doctors.

In object-oriented programming, this idea of one object delegating part of its task to other objects is very important. When one bit of code asks an object to perform some action or retrieve some information, the receiver of the message will often delegate the work to another object. In this world, that object could be another robot, or something else entirely as we shall see. Delegation in this way helps us break up a large program into small, understandable, parts. And, of course, delegation models what happens in the real world. It is nearly the same in the object-oriented world. Delegation is a big idea. One object carries out part of its task by asking another to do something for it.

Interfaces:

A Java interface is used to introduce a protocol that some objects must follow without any restrictions on how the protocol should be implemented. It defines a set of public method names (and parameters when necessary) that we would like to use elsewhere for some objects. A class can implement (not extend) one or more interfaces. Doing so gives it the requirement of implementing all of the methods named in all the interfaces. Also, an interface can define many methods, not just one as we have here.

Unlike abstract classes, interfaces never contain any implementations of methods (and no constructors). They may define constant values, however, though we haven't been introduced to that concept yet. On the other hand, a class can implement several interfaces, but can only extend one class (even if it is an abstract class). In effect, an interface is "all holes" while an abstract class has some holes and usually some additional structure. An interface is purely a way for objects to agree on how they will communicate.

For our robots, if we implement an interface, such as `Worker`, we will also need to extend one of the robot classes, such as `UrRobot`. Like classes, interfaces are usually put into their own files with the name of the interface and `.java` appended to the end: `Worker.java` here. In this case, the interface should be marked public and should be in the same package as the one you are working in.

There is an interface named `Directions` already defined in the Karel J Robot system. It is where the constants `North`, `East`, `South`, `West` (and `infinity`) are defined. The class of `North`, etc., is `Direction`, however, without an s on the end. `UrRobot` implements this interface, so our robots know about the various directions. If you write non-robot classes that need to use these words, you will need to "implement Directions" as we have done before.

By way of analogy, a class if like a blueprint for an object (robot, for example), containing information about how to build it. An interface is like a user's manual, showing how it can be used, but not built.