Objects First

Problem Solving and Algorithms

Learn a basic process for developing a solution to a problem. Nothing in this chapter is unique to using a computer to solve a problem. This process can be used to solve a wide variety of problems, including ones that have nothing to do with computers.

Problems, Solutions, and Tools

I have a problem! I need to thank Aunt Kay for the birthday present she sent me. I could send a thank you note through the mail. I could call her on the telephone. I could send her an email message. I could drive to her house and thank her in person. In fact, there are many ways I could thank her, but that's not the point. The point is that I must decide how I want to solve the problem, and use the appropriate tool to implement (carry out) my plan. The postal service, the telephone, the internet, and my automobile are tools that I can use, but none of these actually solves my problem. In a similar way, a computer does not solve problems, it's just a tool that I can use to implement my plan for solving the problem.

Knowing that Aunt Kay appreciates creative and unusual things, I have decided to hire a singing messenger to deliver my thanks. In this context, the messenger is a tool, but one that needs instructions from me. I have to tell the messenger where Aunt Kay lives, what time I would like the message to be delivered, and what lyrics I want sung. A computer program is similar to my instructions to the messenger.

The story of Aunt Kay uses a familiar context to set the stage for a useful point of view concerning computers and computer programs. The following list summarizes the key aspects of this point of view.

- A computer is a tool that can be used to implement a plan for solving a problem.
- A computer program is a set of instructions for a computer. These instructions describe the steps that the computer must follow to implement a plan.
- An algorithm is a plan for solving a problem.
- A person must design an algorithm.
- A person must translate an algorithm into a computer program.

This point of view sets the stage for a process that we will use to develop solutions to Jeroo problems. The basic process is important because it can be used to solve a wide variety of problems, including ones where the solution will be written in some other programming language.

An Algorithm Development Process

Every problem solution starts with a plan. That plan is called an algorithm.

An **algorithm** is a plan for solving a problem.

There are many ways to write an algorithm. Some are very informal, some are quite formal and mathematical in nature, and some are quite graphical. The instructions for connecting a DVD player to a television are an algorithm. A mathematical formula such as πR^2 is a special case of an algorithm. The form is not particularly important as long as it provides a good way to describe and check the logic of the plan.

The development of an algorithm (a plan) is a key step in solving a problem. Once we have an algorithm, we can translate it into a computer program in some programming language. Our algorithm development process consists of five major steps.

Step 1: Obtain a description of the problem.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

Step 1: Obtain a description of the problem.

This step is much more difficult than it appears. In the following discussion, the word *client* refers to someone who wants to find a solution to a problem, and the word *developer* refers to someone who finds a way to solve the problem. The developer must create an algorithm that will solve the client's problem.

The client is responsible for creating a description of the problem, but this is often the weakest part of the process. It's quite common for a problem description to suffer from one or more of the following types of defects: (1) the description relies on unstated assumptions, (2) the description is ambiguous, (3) the description is incomplete, or (4) the description has internal contradictions. These defects are seldom due to carelessness by the client. Instead, they are due to the fact that natural languages (English, French, Korean, etc.) are rather imprecise. Part of the developer's responsibility is to identify defects in the description of a problem, and to work with the client to remedy those defects.

Step 2: Analyze the problem.

The purpose of this step is to determine both the starting and ending points for solving the problem. This process is analogous to a mathematician determining what is given and what must be proven. A good problem description makes it easier to perform this step.

When determining the starting point, we should start by seeking answers to the following questions:

- What data are available?
- Where is that data?
- What formulas pertain to the problem?
- What rules exist for working with the data?
- What relationships exist among the data values?

When determining the ending point, we need to describe the characteristics of a solution. In other words, how will we know when we're done? Asking the following questions often helps to determine the ending point.

- What new facts will we have?

- What items will have changed?
- What changes will have been made to those items?
- What things will no longer exist?

Step 3: Develop a high-level algorithm.

An algorithm is a plan for solving a problem, but plans come in several levels of detail. It's usually better to start with a high-level algorithm that includes the major part of a solution, but leaves the details until later. We can use an everyday example to demonstrate a high-level algorithm.

Problem: I need a send a birthday card to my brother, Mark.

Analysis: I don't have a card. I prefer to buy a card rather than make one myself.

High-level algorithm:

> Go to a store that sells greeting cards
> Select a card
> Purchase a card
> Mail the card

This algorithm is satisfactory for daily use, but it lacks details that would have to be added were a computer to carry out the solution. These details include answers to questions such as the following.

- "Which store will I visit?"
- "How will I get there: walk, drive, ride my bicycle, take the bus?"
- "What kind of card does Mark like: humorous, sentimental, risqué?"

These kinds of details are considered in the next step of our process.

Step 4: Refine the algorithm by adding more detail.

A high-level algorithm shows the major steps that need to be followed to solve a problem. Now we need to add details to these steps, but how much detail should we add? Unfortunately, the answer to this question depends on the situation. We have to consider who (or what) is going to implement the algorithm and how much that person (or thing) already knows how to do. If someone is going to purchase Mark's birthday card on my behalf, my instructions have to be adapted to whether or not that person is familiar with the stores in the community and how well the purchaser known my brother's taste in greeting cards.

When our goal is to develop algorithms that will lead to computer programs, we need to consider the capabilities of the computer and provide enough detail so that someone else could use our algorithm to write a computer program that follows the steps in our algorithm. As with the birthday card problem, we need to adjust the level of detail to match the ability of the programmer. When in doubt, or when you are learning, it is better to have too much detail than to have too little.

Most of our examples will move from a high-level to a detailed algorithm in a single step, but this is not always reasonable. For larger, more complex problems, it is common to go through this process several times, developing intermediate level algorithms as we go. Each time, we add more detail to the previous algorithm, stopping when we see no benefit to further refinement. This technique of gradually working from a high-level to a detailed algorithm is often called **stepwise refinement**.

Stepwise refinement is a process for developing a detailed algorithm by gradually adding detail to a high-level algorithm.

Step 5: Review the algorithm.

The final step is to review the algorithm. What are we looking for? First, we need to work through the algorithm step by step to determine whether or not it will solve the original problem. Once we are satisfied that the algorithm does provide a solution to the problem, we start to look for other things. The following questions are typical of ones that should be asked whenever we review an algorithm. Asking these questions and seeking their answers is a good way to develop skills that can be applied to the next problem.

- Does this algorithm solve a **very specific problem** or does it solve a **more general problem**? If it solves a very specific problem, should it be generalized?

  For example, an algorithm that computes the area of a circle having radius 5.2 meters (formula $\pi * 5.2^2$) solves a very specific problem, but an algorithm that computes the area of any circle (formula $\pi * R^2$) solves a more general problem.

- Can this algorithm be **simplified**?

  One formula for computing the perimeter of a rectangle is:

  > *length* + *width* + *length* + *width*

  A simpler formula would be:

  > 2.0 * (*length* + *width*)

- Is this solution **similar** to the solution to another problem? How are they alike? How are they different?

  For example, consider the following two formulae:

> Rectangle area = *length* * *width*
> Triangle area = 0.5 * *base* * *height*

Similarities: Each computes an area. Each multiplies two measurements.

Differences: Different measurements are used. The triangle formula contains 0.5.

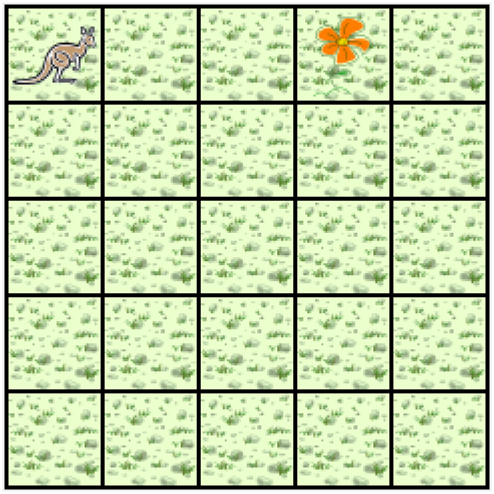
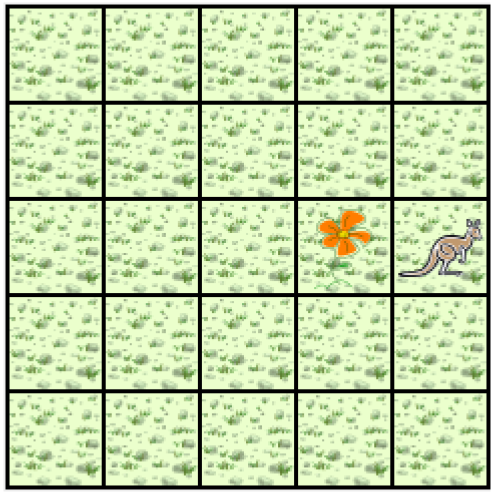
Hypothesis: Perhaps every area formula involves multiplying two measurements.

Example 4.1: Pick and Plant

This section contains an extended example that demonstrates the algorithm development process. To complete the algorithm, we need to know that every Jeroo can hop forward, turn left and right, pick a flower from its current location, and plant a flower at its current location.

Problem Statement (Step 1)

A Jeroo starts at (0, 0) facing East with no flowers in its pouch. There is a flower at location (3, 0). Write a program that directs the Jeroo to pick the flower and plant it at location (3, 2). After planting the flower, the Jeroo should hop one space East and stop. There are no other nets, flowers, or Jeroos on the island.

Start	**Finish**
	

Analysis of the Problem (Step 2)

1. The flower is exactly three spaces ahead of the jeroo.
2. The flower is to be planted exactly two spaces South of its current location.
3. The Jeroo is to finish facing East one space East of the planted flower.
4. There are no nets to worry about.

High-level Algorithm (Step 3)

Let's name the Jeroo Bobby. Bobby should do the following:

```
Get the flower
Put the flower
Hop East
```

Detailed Algorithm (Step 4)

Let's name the Jeroo Bobby. Bobby should do the following:

```
Get the flower
    Hop 3 times
    Pick the flower
Put the flower
    Turn right Hop 2 times Plant a flower
Hop East
    Turn left Hop once
```

Review the Algorithm (Step 5)

1. The high-level algorithm partitioned the problem into three rather easy subproblems. This seems like a good technique.
2. This algorithm solves a very specific problem because the Jeroo and the flower are in very specific locations.
3. This algorithm is actually a solution to a slightly more general problem in which the Jeroo starts anywhere, and the flower is 3 spaces directly ahead of the Jeroo.

Java Code for "Pick and Plant"

A good programmer doesn't write a program all at once. Instead, the programmer will write and test the program in a series of builds. Each build adds to the previous one. The high-level algorithm will guide us in this process.

A good programmer works **incrementally**, add small pieces one at a time and constantly re-checking the work so far.

FIRST BUILD

To see this solution in action, create a new Greenfoot4Sofia scenario and use the **Edit** ➜ **Palettes** ➜ **Jeroo** menu command to make the Jeroo classes visible. Right-click on the `Island` class and create a new subclass with the name of your choice. This subclass will hold your new code.

The recommended first build contains three things:

1. The main method (here `myProgram()` in your island subclass).
2. Declaration and instantiation of every Jeroo that will be used.
3. The high-level algorithm in the form of comments.

```java
public void myProgram()
{
    Jeroo bobby = new Jeroo();
    this.add(bobby);

    // --- Get the flower ---

    // --- Put the flower ---

    // --- Hop East ---

}   // ===== end of method myProgram() =====
```

The instantiation at the beginning of `myProgram()` places `bobby` at (0, 0), facing East, with no flowers.

Once the first build is working correctly, we can proceed to the others. In this case, each build will correspond to one step in the high-level algorithm. It may seem like a lot of work to use four builds for such a simple program, but doing so helps establish habits that will become invaluable as the programs become more complex.

SECOND BUILD

This build adds the logic to "get the flower", which in the detailed algorithm (step 4 above) consists of hopping 3 times and then picking the flower. The new code is indicated by comments that wouldn't appear in the original (they are just here to call attention to the additions). The blank lines help show the organization of the logic.

```java
public void myProgram()
{
    Jeroo bobby = new Jeroo();
    this.add(bobby);

    // --- Get the flower ---
    bobby.hop(3);                  // <-- new code to hop 3 times
    bobby.pick();                  // <-- new code to pick the flower

    // --- Put the flower ---

    // --- Hop East ---

}   // ===== end of method myProgram() =====
```

By taking a moment to run the work so far, you can confirm whether or not this step in the planned algorithm works as expected.

THIRD BUILD

This build adds the logic to "put the flower". New code is indicated by the comments that are provided here to mark the additions.

```java
public void myProgram()
{
    Jeroo bobby = new Jeroo();
    this.add(bobby);

    // --- Get the flower ---
    bobby.hop(3);
    bobby.pick();

    // --- Put the flower ---
    bobby.turn(RIGHT);             // <-- new code to turn right
    bobby.hop(2);                  // <-- new code to hop 2 times
    bobby.plant();                 // <-- new code to plant a flower

    // --- Hop East ---

}   // ===== end of method myProgram() =====
```

FOURTH BUILD (final)

This build adds the logic to "hop East".

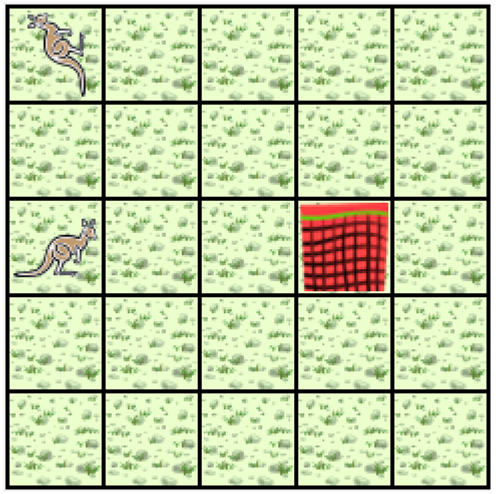
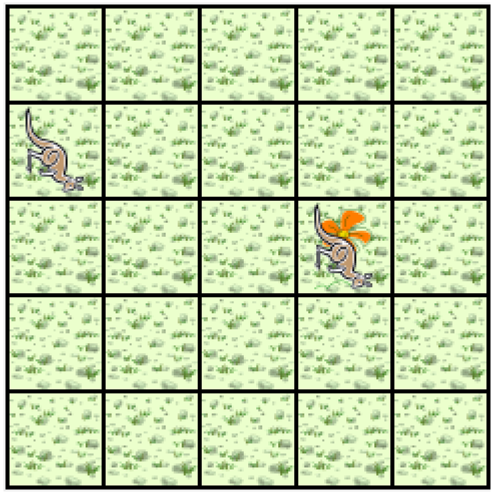
```
public void myProgram()
{
    Jeroo bobby = new Jeroo();
    this.add(bobby);

    // --- Get the flower ---
    bobby.hop(3);
    bobby.pick();

    // --- Put the flower ---
    bobby.turn(RIGHT);
    bobby.hop(2);
    bobby.plant();

    // --- Hop East ---
    bobby.turn(LEFT);              // <-- new code to turn left
    bobby.hop();                   // <-- new code to hop 1 time

}   // ===== end of method myProgram() =====
```

Example 4.2: Replace Net with Flower

This section contains a second example that demonstrates the algorithm development process.

Problem Statement (Step 1)

There are two Jeroos. One Jeroo starts at (0, 0) facing North with one flower in its pouch. The second starts at (0, 2) facing East with one flower in its pouch. There is a net at location (3, 2). Write a program that directs the first Jeroo to give its flower to the second one. After receiving the flower, the second Jeroo must disable the net, and plant a flower in its place. After planting the flower, the Jeroo must turn and face South. There are no other nets, flowers, or Jeroos on the island.

Start	Finish
	

Analysis of the Problem (Step 2)

1. Jeroo_2 is exactly two spaces behind Jeroo_1.
2. The only net is exactly three spaces ahead of Jeroo_2.
3. Each Jeroo has exactly one flower.
4. Jeroo_2 will have two flowers after receiving one from Jeroo_1.
   One flower must be used to disable the net.
   The other flower must be planted at the location of the net, i.e. (3, 2).
5. Jeroo_1 will finish at (0, 1) facing South.
6. Jeroo_2 is to finish at (3, 2) facing South.
7. Each Jeroo will finish with 0 flowers in its pouch. One flower was used to disable the net, and the other was planted.

High-level Algorithm (Step 3)

Let's name the first Jeroo Ann and the second one Andy.

> Ann should do the following:

```
        Find Andy (but don't collide with him)
        Give a flower to Andy (he will be straight ahead)
    After receiving the flower, Andy should do the following:
        Find the net (but don't hop onto it)
        Disable the net
        Plant a flower at the location of the net
        Face South
```

Detailed Algorithm (Step 4)

Let's name the first Jeroo Ann and the second one Andy.

```
Ann should do the following:
    Find Andy
        Turn around (either left or right twice)
        Hop (to location (0, 1))
    Give a flower to Andy
        Give ahead
Now Andy should do the following:
    Find the net
        Hop twice (to location (2, 2))
    Disable the net
        Toss
    Plant a flower at the location of the net
        Hop (to location (3, 2))
        Plant a flower
    Face South
        Turn right
```

Review the Algorithm (Step 5)

1. The high-level algorithm helps manage the details.
2. This algorithm solves a very specific problem, but the specific locations are not important. The only thing that is important is the starting location of the Jeroos relative to one another and the location of the net relative to the second Jeroo's location and direction.

Java Code for "Replace Net with Flower"

As before, the code should be written **incrementally** as a series of builds. Four builds will be suitable for this problem. As usual, the first build will contain the main method, the declaration and instantiation of the Jeroo objects, and the high-level algorithm in the form of comments. The second build will have Ann give her flower to Andy. The third build will have Andy locate and disable the net. In the final build, Andy will place the flower and turn East.

FIRST BUILD

This build creates the main method, instantiates the Jeroos, and outlines the high-level algorithm. In this example, the main method would be `myProgram()` contained within a subclass of `Island`.

```java
public void myProgram()
{
    Jeroo ann  = new Jeroo(0, 0, NORTH, 1);
    this.add(ann);
    Jeroo andy = new Jeroo(0, 2 , 1);  // default EAST
    this.add(andy);

    // --- Ann, find Andy ---

    // --- Ann, give Andy a flower ---

    // --- Andy, find and disable the net ---

    // --- Andy, place a flower at (3, 2) ---

    // --- Andy, face South ---

}   // ===== end of method myProgram() =====
```

SECOND BUILD

This build adds the logic for Ann to locate Andy and give him a flower.

```java
public void myProgram()
{
    Jeroo ann  = new Jeroo(0, 0, NORTH, 1);
    this.add(ann);
    Jeroo andy = new Jeroo(0, 2 , 1);  // default EAST
    this.add(andy);

    // --- Ann, find Andy ---
    ann.turn(LEFT);
    ann.turn(LEFT);
    ann.hop();
    // Now, Ann is at (0, 1) facing South, and Andy is directly ahead

    // --- Ann, give Andy a flower ---
    ann.give(AHEAD);                   // Ann now has 0 flowers, Andy has 2

    // --- Andy, find and disable the net ---

    // --- Andy, place a flower at (3, 2) ---

    // --- Andy, face South ---

}   // ===== end of method myProgram() =====
```

THIRD BUILD

This build adds the logic for Andy to locate and disable the net.

```
public void myProgram()
{
    Jeroo ann  = new Jeroo(0, 0, NORTH, 1);
    this.add(ann);
    Jeroo andy = new Jeroo(0, 2 , 1);  // default EAST
    this.add(andy);

    // --- Ann, find Andy ---
    ann.turn(LEFT);
    ann.turn(LEFT);
    ann.hop();
    // Now, Ann is at (0, 1) facing South, and Andy is directly ahead

    // --- Ann, give Andy a flower ---
    ann.give(AHEAD);                   // Ann now has 0 flowers, Andy has 2

    // --- Andy, find and disable the net ---
    andy.hop(2);                       // Andy is at (2, 2) facing the net
    andy.toss();

    // --- Andy, place a flower at (3, 2) ---

    // --- Andy, face South ---

}   // ===== end of method myProgram() =====
```

FOURTH BUILD (final)

This build adds the logic for Andy to place a flower at (3, 2) and turn South.

```
public void myProgram()
{
    Jeroo ann  = new Jeroo(0, 0, NORTH, 1);
    this.add(ann);
    Jeroo andy = new Jeroo(0, 2 , 1);  // default EAST
    this.add(andy);

    // --- Ann, find Andy ---
    ann.turn(LEFT);
    ann.turn(LEFT);
    ann.hop();
    // Now, Ann is at (0, 1) facing South, and Andy is directly ahead

    // --- Ann, give Andy a flower ---
    ann.give(AHEAD);                   // Ann now has 0 flowers, Andy has 2

    // --- Andy, find and disable the net ---
    andy.hop(2);                       // Andy is at (2, 2) facing the net
    andy.toss();

    // --- Andy, place a flower at (3, 2) ---
    andy.hop();
    andy.plant();

    // --- Andy, face South ---
    andy.turn(RIGHT);

}   // ===== end of method myProgram() =====
```