Second AP* Edition
— with GridWorld

Java Methods

Object-Oriented Programming and Data Structures

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810

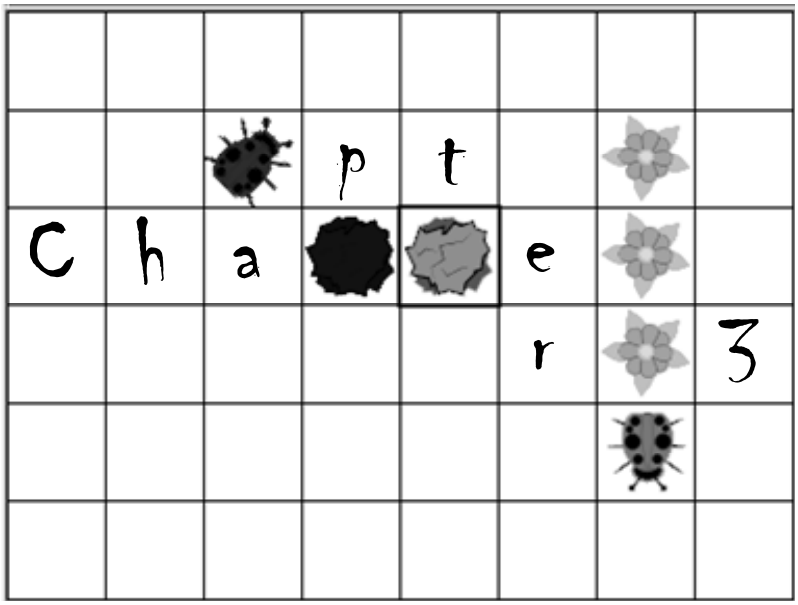
web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2010915303

ISBN 978-0-9824775-7-1

1 2 3 4 5 6 7 8 9 10 16 15 14 13 12 11

Printed in the United States of America



Objects and Classes

3.1 Prologue

Non-technical people sometimes envision a computer programmer's job as sitting at a computer and writing lines of code in a cryptic programming language. Perhaps this is how it might appear to a casual observer. This is not so. The work of a programmer (now called a software engineer) involves not only lines and pages of computer code, but also an orderly structure that matches the task. Even in the earliest days of the computer era, when programs were written directly in machine code, a programmer first developed a more or less abstract view of the task at hand. The overall task was split into meaningful subtasks; then a set of procedures was designed that accomplished specific subtasks; each procedure, in turn, was divided into meaningful smaller segments.

A software engineer has to be able to see the big picture or to zoom in on more intricate details as necessary. Over the years, different software development methodologies have evolved to facilitate this process and to help programmers better communicate with each other. The currently popular methodology is *Object-Oriented Programming* (*OOP*). OOP is considered more suitable than previous methodologies for:

- Team work
- Reuse of software components
- GUI development
- Program maintenance

In OOP, a programmer envisions a software application as a virtual world of interacting objects.

This world is highly structured. To think of objects in a program simply as fish in an ocean would be naive. If we take the ocean as a metaphor, consider that its objects include islands, boats, the sails on the boats, the ropes that control the sails, the people on board, the fish in the water, and even the horizon, an object that does not physically exist! There are objects within objects within objects, and the whole ocean is an object, too.

The following questions immediately come to mind:

- Who describes all the different types of objects in a program? When and how?
- How does an object represent and store information?
- When and how are objects created?
- How can an object communicate with other objects?
- How can objects accomplish useful tasks?

We'll start answering these questions in this chapter and continue through the rest of the book. Our objective in this chapter is to learn the following terms and concepts: object, class, CRC card, instance variable or field, constructor, method, public vs. private, encapsulation and information hiding, inheritance, IS-A and HAS-A relationships.

In this chapter we will refer to the GridWorld case study, developed by the College Board's AP Computer Science Development Committee for AP Computer Science courses and exams. GridWorld is a *framework*, a set of Java classes that can be used to create animations and games that involve "actors" in a rectangular grid. The GridWorld materials and code are available free of charge under the GNU license at the College Board's web site.

> **See `www.skylit.com/javamethods/faqs/` for instructions on how to download the GridWorld materials and configure and run GridWorld projects.**

In this chapter we will work only with Part 1 and Part 2 of GridWorld. We will return to GridWorld again in Chapters 11 and 13.

3.2 *Case Study:* GridWorld

The window in Figure 3-1 comes from the *BugRunner* program in the GridWorld case study. How does one write a program like this in OOP style? A good place to start is to decide what types of objects are needed.

> **An *object* in a running program is an entity that models an object or concept from the real world.**

Some of the objects in a program may model real-world objects, such as a rock, a flower, or a bug. There are also GUI objects that are visible or audible: buttons, sliders, menus, images, audio clips, and so on. Other objects may represent abstract concepts, such as a rectangular grid or a location.

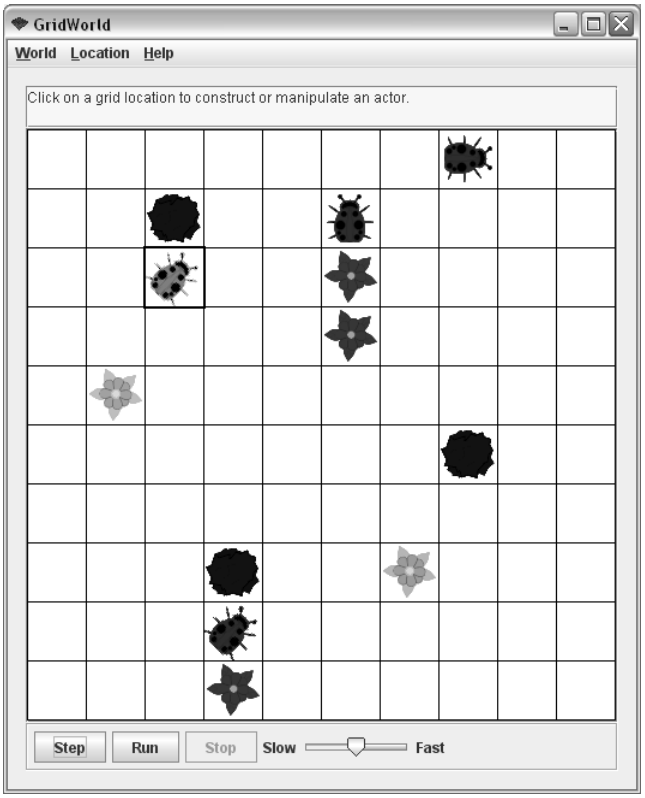


Figure 3-1. A window from the *BugRunner* program in GridWorld

Each object in a running program has a set of attributes and behaviors. The object's attributes hold specific values; some of these values can change while the program is running.

For example, in GridWorld, a "bug" object has such attributes as location, direction, and color, and such behaviors as moving and turning. The location or direction of a bug changes after each step.

A program often employs several objects of the same type. Such objects are said to belong to the same *class*. We also say that an object is an *instance* of its class.

Objects of the same class have the same set of attributes and behaviors; they differ only in the values of some of their attributes.

In Figure 3-1, for example, you can see four "bugs," three "rocks" and five "flowers," that is, four objects of the `Bug` class, three objects of the `Rock` class, and five objects of the `Flower` class. The flowers have different locations and colors; the bugs have different locations, directions, and colors.

In GridWorld, bugs, flowers, rocks, and other inhabitants of the grid are called *actors*.

It is important to distinguish objects from their visual representations in a GUI program. In a well-designed program, visual representations of objects are separate from abstract models of their behavior. For example, in GridWorld, a flower, a rock, or another "actor" is displayed as an image. The images are stored in separate image files, and it is possible to change them without rebuilding the program. In fact, a `Bug` object will remain a `Bug` even if a program does not display it all and just prints out some information about its location and direction. Similarly, GUI components (buttons, menus, etc.) are rather abstract entities. Their appearance can be modified to match the native *look and feel* of the operating system. The labels on menus and buttons, text messages, and program help text might be stored separately, too, so that they can be easily translated into different languages.

Read Part 1 of the GridWorld Student Manual. Set up a project with the `BugRunner` class (located in GridWorld's `firstProject` folder) and the GridWorld library, `gridworld.jar`. (See `www.skylit.com/javamethods/faqs/` for instructions on how to set up GridWorld projects.) Experiment with the program and notice how different types of "actors" behave in the program. A bug moves forward when it can; otherwise it turns 45 degrees clockwise. A bug leaves a flower in its wake when it moves forward. A flower stays in one place, but it gets darker as it gets older. A rock just sits there and does nothing. Each of these actors "knows" the grid to which it belongs and its own location and direction in the grid.

❖ ❖ ❖

Sometimes it is hard to decide whether two objects have serious structural differences and different behaviors and should belong to different classes or if they differ only in the values of some of their attributes. A color is just an attribute, so flowers of different colors are objects of the same class. But if we wanted to have carnivorous flowers that could catch and "eat" bugs, we would probably need a separate class to describe such objects. In OOP languages it is possible to compromise: an object can belong to a *subclass* of a given class and can "inherit" some of the code (data attributes and behaviors) from its parent class. For example, `CarnivorousFlower` can be a subclass of `Flower`. In GridWorld, a `Bug`, a `Rock`, a `Flower` are different types of "actors" and `Bug`, `Rock`, and `Flower` are all subclasses of the class `Actor`. `Actor` is their *superclass*. More on this later (Section 3.6).

The rest of the visible objects in Figure 3-1 are GUI components. There is a menu bar, a text area for messages, and the control panel at the bottom with three buttons and a slider.

Finally, we see an object that represents the whole window in which the program is running. This object holds the grid and the GUI components.

The *BugRunner* program also uses several abstract types of objects. For example, the location of an actor is represented by an object of the class `Location`, which has the attributes row and column. A color is represented by an object of the Java library class `Color`. Another invisible object is a timer that sets the pace when the user presses the `Run` button. The `Timer` class also comes from the Java library.

❖ ❖ ❖

In OOP, a lot of the emphasis shifts from software development to software design. The design methodology, object-oriented design (OOD), parallels the programming methodology (OOP). A good design makes implementation easier. But object-oriented design itself is not easy, and a bad design may derail a project.

The design phase starts with a preliminary discussion of the rough design. One of the informal techniques the designers might use is *CRC cards*. CRC stands for "Class, Responsibilities, Collaborators." A CRC card is simply an index card that describes a class of objects, including the class's name, the main "responsibilities" of this type of object in the program, and its "collaborators," that is, other classes that it depends on (Figure 3-2).

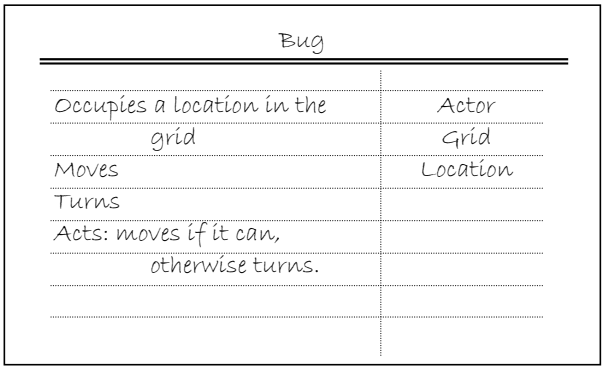


Figure 3-2. A CRC card for the class `Bug`

At this initial stage, software designers do not have to nail down all the details. The responsibilities of each type of object are described in general terms. The need for additional types of objects may become apparent later in the process.

3.3 Classes

In Java, a programmer must describe the different types of objects used in a program in the program's source code.

> **A *class* is a piece of the program's source code that describes a particular type of objects. A formal description of a class is called a *class definition* or a *class declaration*. Programmers write *class definitions*.**

Informally, we say that programmers write classes and that the source code of a Java program consists of one or several classes.

Figure 3-3 summarizes the concepts of a *class* and an *object* and the differences between them.

Class:	Object:
A piece of the program's source code	An entity in a running program
Written by a programmer	Created when the program is running
Specifies the structure (the number and types of attributes) for the objects of this class, the same for all of its objects	Holds specific values of attributes; some of these values can change while the program is running
Specifies the possible behaviors of its objects — the same for all of its objects	Behaves appropriately when called upon
A Java program's source code consists of several classes	A running program can create any number of objects (instances) of a class
Like a blueprint for building cars of a particular model	Like a car of a particular model that you can drive

Figure 3-3. ***Class* vs. *object***

A class is sometimes compared to a cookie cutter: all objects of the class have the same configuration but might have different flavors. When the program is running, different objects of the same class may have different values of attributes. A more apt comparison for a class, perhaps, would be a blueprint for making a specific model of a car. Like objects of the same class, cars of the same model have identically configured parts, but they may be of different colors, and when a car is running, the number of people in it or the amount of gas in the tank may be different from another car of the same model.

The source code for a class is usually stored in a separate file.

For example, `Actor`, `Rock`, `Bug`, `Flower`, `Location`, `ActorWorld`, and `BugRunner` are some of the GridWorld classes. Their source code can be found in `GridWorldCode\framework\info\gridworld` subfolders.

All in all, the GridWorld framework includes 24 classes and one *interface* (we will talk about interfaces in Chapter 11). In addition, `GridWorldCode\projects` subfolders hold the source code for eight project classes, including `BugRunner`, `BoxBug`, and `BoxBugRunner`.

In Java, the name of the source file must be the same as the name of the class, with the extension `.java`.

For example, a class `Flower` must be stored in a file named `Flower.java`.

In Java, all names, including the names of classes, are case-sensitive. By convention, the names of classes (and the names of their source files) always start with a capital letter.

A class describes three aspects that every instance (object) of this class has: (1) the data elements (attributes) of an object of this class, (2) the ways in which an object of this class can be created, and (3) what this type of object can do.

An object's data elements are called *instance variables* or *fields*. Procedures for creating an object are called *constructors*. Behaviors of an object of a class are called *methods*.

The class describes all these features in a very formal and precise manner.

Not every class has fields, constructors, and methods explicitly defined: some of these features might be implicit or absent.

Also some classes don't have objects of their type ever created in the program. For example, the `HelloWorld` program (Chapter 2, page 23) is described by the class `HelloWorld`, which has only one method, `main`. This program does not create any objects of this class. The `main` method is declared `static`, which means it belongs to the class as a whole, not to particular objects. It is called automatically when the program is started.

❖ ❖ ❖

Figure 3-4 shows a schematic view of a class's source code (adapted from GridWorld's `Actor` class).

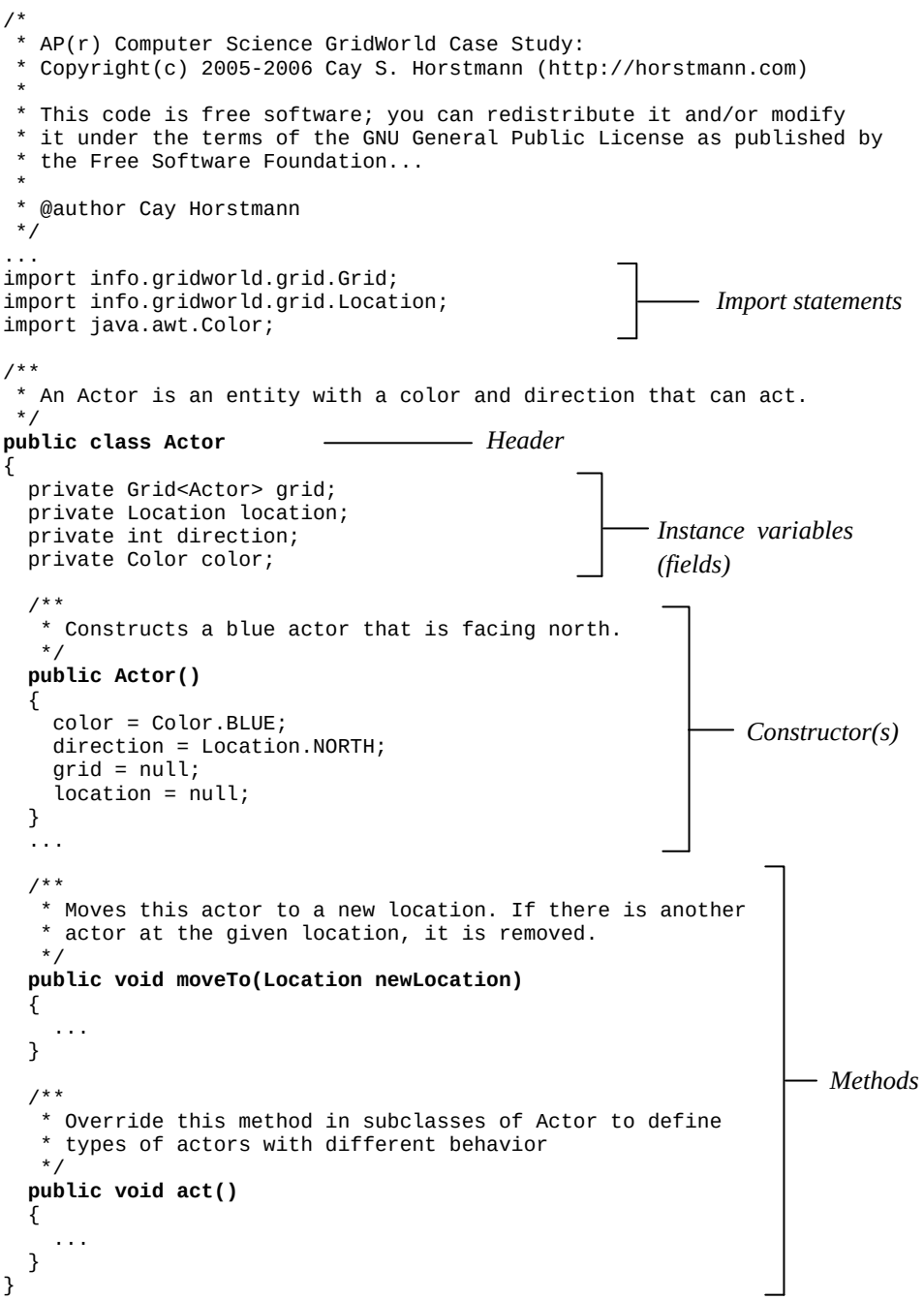


Figure 3-4. A schematic view of a class's source code

You can see the following elements:

1. An comment at the top

   It is a good idea to start the source code of each Java class with a comment that briefly describes the purpose of the class, its author, perhaps the copyright arrangement, history of revisions, and so on. Usually, the header of a class and each important feature of the class (each field, constructor, and method) is preceded by a comment, which describes the purpose of that feature. The compiler ignores all comments.

2 "import" statements, if necessary

   `import` statements tell the compiler where to look for other classes used by this class. They may refer to

   - other classes created by you or another programmer specifically for the same project;
   - more general reusable classes and packages created for different projects;
   - Java library classes.

   The compiler automatically finds the definitions of the needed classes, as long as they are located in the same folder as this class. But you need to tell the compiler where to find Java library classes and classes from different packages that reside in other folders or in "jar" files. A *Java archive* (`.jar`) file is a file that contains several pre-compiled classes (`.class` files) in compressed form. For example, all of the GridWorld framework classes are collected in one jar file, `gridworld.jar`. A jar file maintains the same structure of folders as in folders that are not compressed. Java library classes are collected in `.jar` files, too.

   If you examine GridWorld's source code, you will see that the `Actor.java` file resides in the `framework/info/gridworld/actor` folder. The `Grid` and `Location` classes used by the `Actor` class reside in a different folder, `framework/info/gridworld/grid`. So the `Actor` class needs `import` statements for these two classes:

```
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
```

   `Actor` also uses the Java library class `Color`, so the code for the `Actor` class has an `import` statement for it, too:

```
import java.awt.Color;
```

This statement tells the compiler that it can find the `Color` class in the `java.awt` package[awt] of the Java's standard library. Or we could put

```
import java.awt.*;
```

Then the compiler would search the entire `java.awt` package for the classes that it couldn't find elsewhere.

Without `import` statements, you would have to use the *fully-qualified* names of library classes everywhere in your code, as in

```
private info.gridworld.grid.Location location;
private java.awt.Color color;
```

instead of

```
private Location location;
private Color color;
```

This would clutter your code.

3. The class header

The import statements are followed by the class header. The header —

```
public class Actor
```

— states that the name of this class is `Actor` and that this is a "public" class, which means it is visible to other classes.

4. The class definition body

The class header is followed by the class definition body within braces.

The order of fields, constructors, and methods in a class definition does not matter for the compiler, but it is customary to group all the fields together, usually at the top, followed by all the constructors, and then all the methods.

❖ ❖ ❖

When a Java application starts, control is passed to the `main` method. In the *BugRunner* application, the `main` method resides in the `BugRunner` class. `BugRunner` is different from the classes we described above (`Actor`, `Bug`, `Flower`, `Rock`): it is not used to define objects, and it has no instance variables or constructors; `main` is its only method (Figure 3-5).

```
... (comment)

import info.gridworld.actor.ActorWorld;
import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;

... (comment)
public class BugRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    world.add(new Bug());
    world.add(new Rock());
    world.show();
  }
}
```

Figure 3-5. Fragments from `BugRunner.java`

The word `static` in `public static void main(...)` indicates that the method `main` is not called for any particular object: it belongs to the class as a whole. In fact, `main` is called automatically when a Java application is started. A Java application can have only one `main` method.

We could place `main` in any class, but it is cleaner to put it in a separate class. The name of the class, `BugRunner`, is chosen by its author; it could be called `BugTest` or `FirstProject` instead. `BugRunner`'s `main` method creates a `world` (an object of the type `ActorWorld`) and adds one `Bug` object and one `Rock` object to `world` (at random locations) by calling `world`'s `add` method for each of the added actors. It then displays `world` by calling its method `show` and waits for a command from the user.

3.4 *Lab:* Interacting with Actors

The author of GridWorld implemented in it the *direct manipulation interface* feature, which allows you to invoke constructors and call methods of different actors interactively, by clicking on occupied and empty cells of the grid.

This feature is described in GridWorld's help, accessible from the `Help` menu. If you click on an empty square in the grid, a menu will pop up that lists all the constructors for different types of actors that are currently in the grid. You can choose one of the constructors, and the corresponding type of actor will be added to the grid at that location. If you click on a square that already contains an actor, a menu pops up that lists all the methods for that type of actor. If you choose one, that method will be executed. You can also choose (click on) an actor and press `Delete` to remove that actor from the grid.

This interactive feature is provided for study purposes only — in general it is not required (or easy to implement) in a typical OOP program.

Experiment with the *BugRunner* program. Add a few bugs and flowers of different colors interactively, when the program is already running. Also add an `Actor` object. (`Actor`'s constructor is listed among other constructors because `Actor` is the superclass for more specific types of actors, such as `Bug`, `Flower`, etc.) Make one of the bugs move, turn, and "act" by invoking the respective methods from the pop-up menu. Change the color of the `Actor` by invoking its `setColor` method. Observe how the `Actor` "acts" by invoking its `act` method. Then delete some of the actors from the grid.

3.5 Fields, Constructors, and Methods

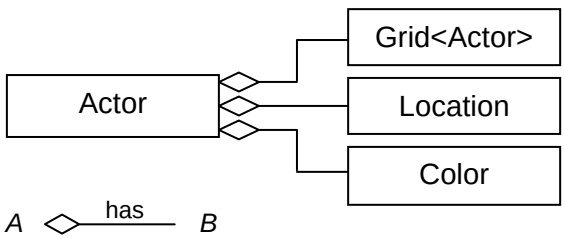
As we said earlier, the definition of a class describes all the *instance variables* of objects of this class. Instance variables are also called *data fields* or simply *fields*, from the analogy with fields in a form that can be filled in with different values.

Each field has a name, given by a programmer, and a type. An `Actor` object, for example, has four fields:

```
private Grid<Actor> grid;
private Location location;
private int direction;
private Color color;
```

Think of an object's instance variables as its private "memory." (This is only a metaphor, of course: in reality, when a program is running, its objects are represented by chunks of RAM.) An object's "memory" may include other objects, and also numbers and text characters. (In Java, numbers and text characters are usually not represented by objects; for them Java provides special data types, `int`, `char`, etc., called *primitive data types*.)

The `grid` field in `Actor` refers to the grid to which this actor belongs. The type of this field is `Grid<Actor>`. `location` is another field; its type is `Location`: that is, it is an object of the `Location` class. `color` is another field; its type is `Color`. An OOP designer would say that an `Actor` *HAS-A* (has a) `Grid<Actor>`, HAS-A `Location`, and HAS-A `Color`. We can show these relationships between classes in a UML (Unified Modeling Language) diagram:



❖ ❖ ❖

An object is created with the `new` operator, which invokes (calls) one of the constructors defined in the object's class. For example:

```
Actor alice = new Actor();
```

A constructor is a procedure, usually quite short, that is used primarily to initialize the values of the instance variables for the object being created.

For example:

```
public Actor()
{
  color = Color.BLUE;
  direction = Location.NORTH;
  grid = null;          // will be set later when this actor is
  location = null;      //    added to a grid
}
```

A constructor must always have the same name as its class.

A constructor can accept one or more parameters or no parameters at all. The latter is called a "no-args" constructor (parameters are often called "arguments," as in math, or "args" for short).

A class may have several constructors that differ in the number and/or types of parameters that they accept.

For example, the `Bug` class defines two constructors:

```
public Bug()
{
  setColor(Color.RED);
}
```

and

```
public Bug(Color bugColor)
{
  setColor(bugColor);
}
```

The first one is a no-args constructor; it creates a red bug; the second takes one parameter, color, and creates a bug of that color. (Both constructors call `Actor`'s `setColor` method to set the color of the actor.)

The number, types, and order of parameters passed to the `new` operator when an object is created must match the number, types, and order of parameters accepted by one of the class's constructors.

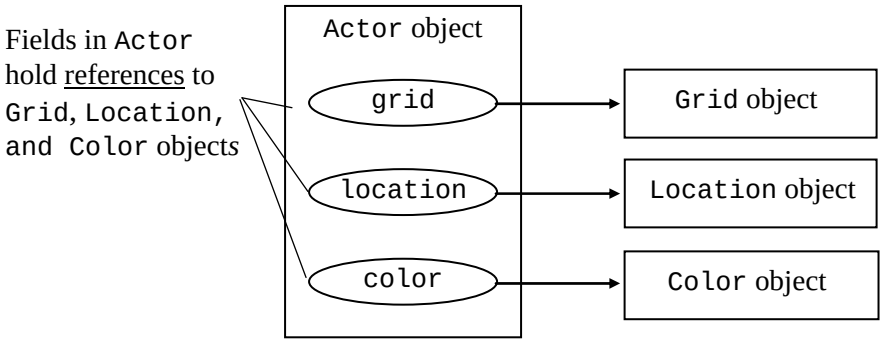
For example, if we wanted to add a green bug to `world` at location (2, 3) we could write:

```
Location loc = new Location(2, 3);
Bug bob = new Bug(Color.GREEN);
world.add(loc, bob);
```

or simply

```
world.add(new Location(2, 3), new Bug(Color.GREEN));
```

When an object is created, a chunk of RAM is allocated to hold it, and `new` returns a *reference* to that location, which is basically the object's address in RAM. The reference may be stored in a variable:



Eventually several variables may hold references to the same object. If you compare an object to a web page, a reference is like the page's URL (web address). Many users may have that URL saved somewhere in their "favorites" or in their own web pages. A Java interpreter is equipped with a mechanism that keeps track of all the references to a particular object that are currently in existence in a running program. Gradually the references to an object may all cease to exist. Then the object is unreachable, because no other object in the program can find it or knows it exists. The Java interpreter finds and destroys such useless objects and frees the memory they occupied. This mechanism is called *garbage collection*.

Each Java class has at least one constructor. If you don't define any, the compiler supplies a default no-args constructor that initializes all the instance variables to default values (zeroes for numbers, `null` for objects, `false` for `boolean` fields).

Since `Bug` is a subclass of `Actor`, `Bug`'s constructors first call `Actor`'s no-args constructor to initialize `Actor`'s instance variables. These calls are implicit — they are not shown in `Bug`'s code.

But it is possible to call a superclass's constructor explicitly and pass parameters to it using the keyword `super`.

❖ ❖ ❖

Metaphorically speaking, an object can "send messages" to other objects. In Java, to "send a message" means to call another object's *method*. A method is a function or a procedure that performs a certain task or computation. All of an object's methods are described in its class definition, and all the objects of a given class have exactly the same set of methods. The methods define what an object can do, what kind of "messages" it "understands" and can respond to.

Does a method belong to an object or to a class? The terminology here is not very precise. When we focus on a running program, we say that an object has a method, meaning that we can call that method for that particular object. When we focus on the program's source code, we say that a class has a method, meaning that a programmer included code for the method in the class definition.

Each method has a name, given by the programmer. Like a constructor, a method may accept one or more parameters. For example, once we have created an object `alice` of the `Actor` class and added it to a grid, we can call its `moveTo` method:

```
alice.moveTo(loc);
```

This statement moves `alice` to the `Location loc`. The compiler understands this statement because we have defined the `moveTo` method in the `Actor` class.

Parameters passed to a method must match the number, types, and order of parameters that the method expects.

Empty parentheses in a method's header indicate that this method does not take any parameters. Such a method is called with empty parentheses. For example, if `bob` is a `Bug`, we can call its `turn` method:

```
bob.turn();
```

A method may return a value to the caller. The method's header specifies whether the method returns a value or not, and if it does, of what type. For example, `Actor`'s `getLocation` method returns this actor's location:

```
public Location getLocation()
{
  return location;
}
```

The keyword `void` in a method's header indicates that the method does not return any value.

For example, `Bug`'s `move` and `turn` methods are declared `void`. The `main` method is `void`, too.

A method can call other methods of the same object or of a different object. For example, `Bug`'s `act` method calls its `canMove` method and then its `move` or `turn` methods:

```
public void act()
{
  if (canMove())
    move();
  else
    turn();
}
```

`Flower`'s `act` method calls `Flower`'s `getColor` and `setColor` methods (inherited from `Actor`) and `Color`'s `getRed`, `getGreen`, and `getBlue` methods (which return the red, green, and blue components of the color):

```
public void act()
{
  Color c = getColor();

  int red = (int) (c.getRed() * (1 - DARKENING_FACTOR));
  int green = (int) (c.getGreen() * (1 - DARKENING_FACTOR));
  int blue = (int) (c.getBlue() * (1 - DARKENING_FACTOR));

  setColor(new Color(red, green, blue));
}
```

❖ ❖ ❖

The `Actor` class provides a well-defined functionality through its constructors and public methods. The user of the `Actor` class (possibly a different programmer) does not need to know all the details of how the class `Actor` works, only how to construct its objects and what they can do. In fact, all the instance variables in `Actor` are declared `private` so that programmers writing classes that use `Actor` cannot refer to them directly. Some of class's methods can be declared private, too. This technique is called *encapsulation* and *information hiding*.

There are two advantages to such an arrangement:

- `Actor`'s programmer can change the structure of the fields in the `Actor` class, and the rest of the project won't be affected, as long as `Actor`'s constructors and public methods have the same specifications and work as before;
- `Actor`'s programmer can document the `Actor` class for the other team members (and other programmers who want to use this class) by describing all its constructors and public methods; there is no need to document the implementation details.

It is easier to maintain, document, and reuse an encapsulated class.

After a class is written, it is a good idea to test it in isolation from other classes. The `Actor` class is too complicated to be tested in its entirety outside GridWorld, but we can create a small program that tests some of `Actor`'s features, for example, its `getDirection` and `setDirection` methods:

```
import info.gridworld.actor.Actor;

public class TestActor
{
  public static void main(String[] args)
  {
    _______________________________  // create an Actor called alice

    System.out.println(alice.getDirection());

    _______________________________ // call alice's setDirection method
                                    //    with the parameter 90

    System.out.println(alice.getDirection());
  }
}
```

Type in the above code, filling in the blanks, and save it in a file `TestActor.java`. Create a project that includes the `TestActor` class and the GridWorld library, and test your program. Explain the output. Now try to call `setDirection` with the parameter 500. Explain the output.

3.6 Inheritance

A `Bug` does not have a method to reverse direction. It would be easy to add a method `turnAround` to the `Bug` class. But there are several reasons why adding methods to an existing class may be not feasible or desirable.

First, you may not have access to the source code of the class. It may be a library class or a class that came to you from someone else without its source. For example, if you only had `gridworld.jar` and `BugRunner.java`, but not the source code for other classes, you could still run GridWorld projects, but you couldn't change `Bug.java`. Second, your boss may not allow you to change a working and tested class. You may have access to its source, but it may be "read-only." A large organization cannot allow every programmer to change every class at will. Once a class is written and tested, it may be off-limits until the next release. Third, your class may already be in use in other projects. If you change it now, you will have different versions floating around and it may become confusing. Fourth, not all projects need additional methods. If you keep adding methods for every contingency, your class will eventually become too large and inconvenient to use.

The proper solution to this dilemma is to *derive* a new class from an existing class.

> **In OOP, a programmer can create a new class by extending an existing class. The new class can add new methods or redefine some of the existing ones. New fields can be added, too. This concept is called *inheritance*.**

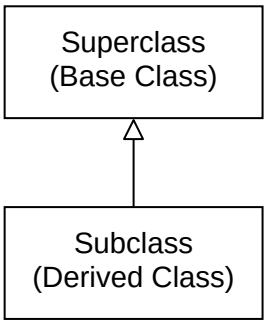
Inheritance is one of the fundamental OOP concepts, and all OOP languages support it.

> **Java uses the keyword `extends` to indicate that a class extends another class.**

For example,

```
public class BoxBug extends Bug
{
  ...
}
```

If class *D* extends class *B*, then *B* is called a *superclass* (or a *base class*) and *D* is called a *subclass (or a derived class*). The relationship of inheritance is usually indicated in UML diagrams by an arrow with a triangular head from a subclass to its superclass:

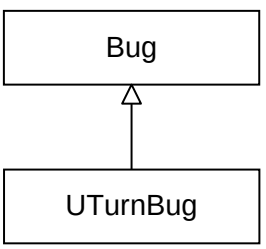


In Java you can extend a class without having its source code.

A subclass *inherits* all the methods and fields of its superclass. Constructors are not inherited; a subclass has to provide its own.

In Java every class extends the library class `Object` by default. So all the classes in a program belong to one large hierarchy of classes with `Object` at the top. `Object` supplies a few common methods, including `toString` and `getClass`.

In our example, we create a new class `UTurnBug`, which extends `Bug`:



`UTurnBug` is a short class (Figure 3-6). It has two constructors that parallel `Bug`'s constructors and adds one method, `turnAround`. It also overrides (redefines) the `Bug`'s `act` method. We should not duplicate `Bug`'s or `Actor`'s other methods and fields in the `UTurnBug` class, as they are inherited from `Bug` and `Actor`.

```
/*
 * A subclass of Bug that adds the turnAround method and
 * redefines Bug's act method so that this bug
 * makes a U-turn when it can't move
 */

import info.gridworld.actor.Bug;
import java.awt.Color;

public class UTurnBug extends Bug
{
  public UTurnBug()
  {
    setColor(Color.YELLOW);
  }

  public UTurnBug(Color bugColor)
  {
    setColor(bugColor);
  }

  public void turnAround()
  {
    turn(); turn(); turn(); turn();
            // Or: setDirection(getDirection() + 180);
  }

  // Overrides Bug's act method
  public void act()
  {
    if (canMove())
        move();
    else
        turnAround();
  }
}
```

Figure 3-6. JM\Ch03\GridWorld\UTurnBug.java

Set up a GridWorld project with the `BugRunner` class and the `UTurnBug` class (from `JM\Ch03\GridWorld\UTurnBug.java`). Add a statement to `BugRunner` to add a `UTurnBug` to `world`. Run the program and test `UTurnBug`'s `turnAround` method, first interactively, by invoking it from the pop-up menu, then by running the program through several steps.

❖ ❖ ❖

In the `UTurnBug` example, `UTurnBug`'s constructors take the same number and types of parameters as `Bug`'s constructors. In general, this does not have to be the case. `BoxBug`, for example, has only one constructor, and it takes one parameter, an integer (which specifies the number of steps before a turn).

Also, notice a paradox. Our `UTurnBug` bug inherits all the fields from `Bug`, which in turn has inherited all the fields from `Actor`. So a `UTurnBug` has a `direction` field. However, this and other fields are declared `private` in `Actor`. This means that the programmer who wrote the `UTurnBug` class did not have direct access to them (even if he is the same programmer who wrote `Actor`!). Recall that the `Actor` class is fully encapsulated and all its fields are private. So the statement

```
direction = direction + 180;
```

won't work in `UTurnBug`. What do we do? `Actor`'s subclasses and, in fact, any other classes that use `Actor` might need access to the values stored in the private fields.

To resolve the issue, the `Actor` class provides public methods that simply return the values of its private fields: `getDirection`, `getLocation`, `getColor`, `getGrid`.

> **Such methods are called *accessor* methods (or simply *accessors*) or *getters* because they give outsiders access to the values of private fields of an object.**

It is a very common practice to provide accessor methods for those private fields of a class that may be of interest to other classes. In our example, we could use

```
setDirection(getDirection() + 180);
```

Methods like `setDirection` are called *setters* or *modifiers*.

❖ ❖ ❖

Inheritance represents the *IS-A relationship* between objects. A `Bug` IS-A (is an) `Actor`. A `UTurnBug` IS-A `Bug`. In addition to the fields and methods, an object of a subclass inherits a less tangible but also very valuable asset from its superclass: its type. It is like inheriting the family name or title. The superclass's type becomes a secondary, more generic type of an object of the subclass. Whenever a statement or a method call expects an `Actor`-type object, you can plug in a `Bug`-type or a `UTurnBug`-type object instead, because a `Bug` is an `Actor`, and a `UTurnBug` is an `Actor`, too (by virtue of being a `Bug`).

For example, the class `ActorWorld` has a method `add`:

```
public void add(Actor a)
{
  ...
}
```

Since `Bug`, `Flower`, `Rock`, and `UTurnBug` are all subclasses of `Actor`, you can pass a `Bug`, a `Flower`, a `Rock`, or a `UTurnBug` object to `ActorWorld`'s `add` method:

```
world.add(new UTurnBug());
```

In Java, a collection of objects can only hold objects of a specified type. For example, the `grid` field in `Actor` is defined as a `Grid<Actor>` object — `grid` holds `Actors`. We wouldn't be able to place bugs, flowers, and rocks into the same grid if `Bug`, `Flower`, and `Rock` were not subclasses of `Actor`.

For the same reason, the following statements will compile with no problems:

```
Actor daisy = new Flower();
Bug boxy = new BoxBug();
Actor pacer = new UTurnBug();
```

Since every class extends the class `Object` by default, every object IS-A(n) `Object`.

❖ ❖ ❖

Figure 3-7 shows a UML diagram of some of the GridWorld classes and their relationships.

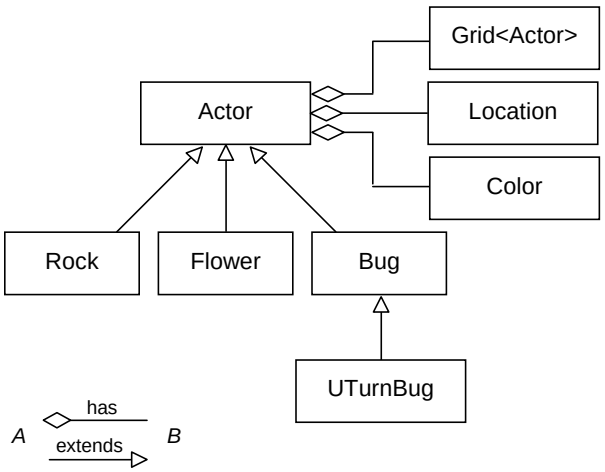


Figure 3-7. `Actor` has a `Grid<Actor>`, `Location`, and `Color`; `Rock`, `Flower`, and `Bug` extend `Actor`; `UTurnBug` extends `Bug`

❖ ❖ ❖

We have made our first steps in OOP. But a lot remains to be learned. We will discuss more advanced concepts and continue with GridWorld in Chapter 11.

3.7 *Lab:* Random Bugs

Read Part 2 of the GridWorld *Student Manual*. Set up and run the *BoxBug* project with the `BoxBug` and `BoxBugRunner` classes (provided in the GridWorld's `projects/boxBug` folder) and `gridworld.jar`. Review `BoxBug`'s source code.

In this lab you will create a new variation of `Bug`, a `RandomBug`. A `RandomBug` is similar to a regular `Bug`: if it can move, it does. Then, regardless of whether it has moved or not, it changes direction randomly (by a multiple of 45 degrees).

Follow these steps:

1. Copy `UTurnBug.java` into `RandomBug.java` and update the class's header and constructor names in the code. `RandomBug` should extend `Bug`.

2. Replace the `turnAround` method with a `turn(angle)` method:

```
public void turn(int angle)
{
  ...
}
```

   Use `UTurnBug`'s code (Figure 3-6) as a prototype for setting the direction of the bug.

   `RandomBug` already has a `turn` method, inherited from `Bug`. That method takes no parameters. It is acceptable and sometimes desirable to give the same name to two or several methods in a class, as long as these methods take different numbers and/or types of parameters. This is called *method overloading*.

3. Modify the `act` method to make the bug move if it can. Then the bug should turn by a random angle, which is a multiple of 45 degrees (or — the same thing — simply turn in a random direction). The statement

```
int angle = 45 * (int)(Math.random() * 8);
```

   sets `angle` to a random multiple of 45, from 0 to 315. (We will explain how it works later, in Chapter 6; for now just use it as written above.)

4. Copy `BugRunner.java` into `RandomBugRunner.java`, change the class's name in the class's header, and edit the `main` method to add a couple of `RandomBug` objects to `world`. Create a GridWorld project with `RandomBugRunner` and `RandomBug` and test your program.

Also complete Exercises 1 and 2 on page 13 of the GridWorld Student Manual.

3.8 Summary

An OOP program is best visualized as a virtual world of interacting objects. A program's source code describes different types of objects used in the program. Objects of the same type are said to belong to the same *class*. An object is called an *instance* of its class. The source code of a Java program consists of *definitions of classes*.

The source code for each class is stored in a separate file with the same name as the class and the extension `.java`. A class name always starts with a capital letter. It is customary to place all your classes for a small project into the same folder. Several compiled Java classes may be collected in one `.jar` file.

A *CRC card* gives a preliminary, informal description of a class, listing its name, the key "responsibilities" of its objects, and the other classes this class depends on ("collaborators").

The `import` statements at the top of a class's source code tell the compiler where it can find the library classes and packages used in that class.

A class's source code begins with an optional brief comment that describes the purpose of the class, followed by `import` statements, if necessary, then the class's header, and the class's body within braces. A class defines the data elements of an object of that class, called *instance variables* or *fields*. Each instance variable has a name, given by a programmer, and a type. The set of fields serves as the "personal memory" of an object. Their values may be different for different objects of the class, and these values can change while the program is running. A class also defines *constructors*, which are short procedures for creating objects of that class, and *methods*, which describe what an object can do.

A constructor always has the same name as its class. A constructor is used primarily to set the initial values of the object's fields. It can accept one or several parameters that are used to initialize the fields. A constructor that does not take any parameters is called a *no-args constructor*. A class may have several constructors that differ in the number or types of parameters that they accept. If no constructors are defined, the compiler automatically supplies one default no-args constructor that sets all the instance variables to default values (zeroes for numbers, `null` for objects, `false` for `boolean` variables).

You create a new object in the program using the `new` operator. The parameters passed to `new` must match the number, types, and order of parameters of one of the constructors in the object's class, and `new` invokes that constructor. `new` allocates memory to store the newly constructed object.

The functionality of a class — what its objects can do — is defined by its *methods*. A method accomplishes a certain task. It can be called from constructors and other methods of the same class and, if it is declared `public`, from constructors and methods of other classes. A method can take parameters as its "inputs." Parameters passed to a method must match the number, types, and order of parameters that the method expects. A method can return a value of a specified type to the caller. A method declared `void` does not return any value.

In OOP, all the instance variables of a class are usually declared `private`, so only objects of the same class have direct access to them. Some of the methods may be private, too. Users of a class do not need to know how the class is implemented and what its private fields and methods are. This practice is called *information hiding*. A class interacts with other classes only through a well-defined set of constructors and public methods. This concept is called *encapsulation*. Encapsulation facilitates program maintenance, code reuse, and documentation. A class often provides public methods that return the values of an object's private fields, so that an object of a different class can access those values. Such methods are called *accessor methods* or *accessors*. Methods that set the values of private fields are called *setters* or *modifiers*.

A class definition does not have to start from scratch: it can *extend* the definition of another class, adding fields and/or methods or overriding (redefining) some of the methods. This concept is called *inheritance*. It is said that a *subclass* (or *derived class*) extends a *superclass* (or *base class*). Constructors are not inherited.

An object of a subclass also inherits the type of its superclass as a secondary, more generic type. This formalizes the *IS-A relationship* between objects: an object of a subclass IS-A(n) object of its superclass.

Exercises

Sections 3.1-3.5

1. Mark true or false and explain:

(a) The name of a class in Java must be the same as the name of its source file (excluding the extension `.java`). _____
(b) The names of classes are case-sensitive. _____
(c) The `import` statement tells the compiler which other classes use this class. _____ ✓

2. Mark true or false and explain:

(a) The *BugRunner* program consists of one class. _____ ✓
(b) A Java program can have as many classes as necessary. _____
(c) A Java program is allowed to create only one object of each class. _____
(d) Every class has a method called `main`. _____ ✓

3. Navigate your browser to Oracle's Java API (Application Programming Interface) documentation web site (for example, `http://download.oracle.com/javase/6/docs/api/index.html`), or, if you have the JDK documentation installed on your computer, open the file `<JDK base folder>/docs/api/index.html` (for example, `C:/Program Files/Java/jdk1.6.0_21/docs/api/index.html`).

(a) Approximately how many different packages are listed in the API spec?

(b) Find `JFrame` in the list of classes in the left column and click on it. Scroll down the main window to the "Method Summary" section. Approximately how many methods does the `JFrame` class have, including methods inherited from other classes? 3? 12? 25? 300-400? ✓

4. Mark true or false and explain:

(a) Fields of a class are usually declared `private`. _____
(b) An object has to be created before it can be used. _____ ✓
(c) A class may have more than one constructor. _____
(d) The programmer gives names to objects in his program. _____
(e) When an object is created, the program always calls its `init` method. _____ ✓

5. What are the benefits of encapsulation? Name three.

6. Make a copy of `gridworld.jar` and rename it into `gridworld.zip`. Examine its contents. As you can see, a `.jar` file is nothing more than a compressed folder. How many compiled Java classes does it hold in its `info/gridworld/actor` subfolder?

7. The constructor of GridWorld's `Location` class takes two integer parameters. Which of them is the row and which is the column of the location? Are rows and columns counted from 0 or from 1? Find out by running *BugRunner* and positioning the cursor over the grid.

8. Modify GridWorld's `BugRunner` class to place into the grid a red bug at location (1, 2) and a green bug at the upper left corner of the grid. Add a gray rock at a random location. ≤ Hint: see `BoxBugRunner.java` for an example of how to place an actor at a specified location. ≥ ✓

9.■ Modify `BugRunner` to place into the grid three bugs in a row, next to each other, all facing east. ≤ Hint: turn each bug a couple of times before adding it to the grid. ≥

10.■ Write a simple console application that creates a `Bug` and "prints it out":

```
Bug bob = new Bug();
System.out.println(bob);
```

What is displayed? How does the program know how to "print" a bug? Examine `Bug.java` and `Actor.java` to find an explanation. ✓

11.◆ Create a class `Book` with two private `int` fields, `numPages` and `currentPage`. Supply a constructor that takes one parameter and sets `numPages` to that value and `currentPage` to 1. Provide accessor methods for both fields. Also provide a method `nextPage` that increments `currentPage` by 1, but only if `currentPage` is less than `numPages`.

Hint:

```
if (currentPage < numPages)
  currentPage++;
```

Create a `BookTest` class with a `main` method. Let `main` create a `Book` object with 3 pages, then call its `nextPage` method three times, printing out the value of `currentPage` after each call.

12.◆ The class `Circle` (`Circle.java` in `JM\Ch03\Exercises`) describes a circle with a given radius. The radius has the type `double`, which is a primitive data type used for representing real numbers. The `CircleTest.java` class in `JM\Ch03\Exercises` is a tiny console application that prompts the user to enter a number for the radius, creates a `Circle` object of that radius, and displays its area by calling the `Circle`'s `getArea` method.

Create a class `Cylinder` with two private fields: `Circle base` and `double height`. Is it fair to say that a `Cylinder` HAS-A `Circle`? Provide a constructor that takes two `double` parameters, `r` and `h`, initializes `base` to a new `Circle` with radius `r`, and initializes `height` to `h`. Provide a method `getVolume` that returns the volume of the cylinder (which is equal to the base area multiplied by height). Create a simple test program `CylinderTest`, that prompts the user to enter the radius and height of a cylinder, creates a new cylinder with these dimensions, and displays its volume.

13.◆ `JM\Ch03\Exercises\CoinTest` is part of a program that shows a picture of a coin in the middle of a window and "flips" the coin every two seconds. Your task is to supply the second class for this program, `Coin`.

The `Coin` class should have one constructor that takes two parameters of the type `Image`: the heads and tails pictures of the coin. The constructor saves these images in the coin's private fields (of the type `Image`). `Coin` should also have a field that indicates which side of the coin is displayed. The `Coin` class should have two methods:

```
/**
 * Flips this coin
 */
public void flip()
{
  ...
}
```

and

```
/**
 * Draws the appropriate side of the coin
 * centered at (x, y)
 */
public void draw(Graphics g, int x, int y)
{
  ...
}
```

Hints:

1. `import java.awt.Image;`
   `import java.awt.Graphics;`
2. The class `Graphics` has a method that draws an image at a given location. Call it like this:

   ```
   g.drawImage(pic, xUL, yUL, null);
   ```

   where `pic` is the image and `xUL` and `yUL` are the coordinates of its upper left corner. You need to calculate `xUL` and `yUL` from the `x` and `y` passed to `Coin`'s `draw`. Explore the documentation for the library class `Image` to find methods that return the width and height of an image.
3. Find copyright-free image files for the two sides of a coin on the Internet or scan or take a picture of a coin and create your own image files.

Sections 3.6-3.8

14. Mark true or false:

(a) A subclass inherits all the fields and public methods of its superclass. _____ ✓

(b) A subclass inherits all those constructors of its superclass that are not defined explicitly in the subclass. _____ ✓

15. Question 12 above asks you to write a class `Cylinder` with two fields: `Circle base` and `double height`. Instead of making `base` a field we could simply derive `Cylinder` from `Circle`, adding only the `height` field. Discuss the merits of this design option. ✓

16. Which of the following assignment statements will compile without errors?

(a) `Actor a1ice1 = new Actor();` _____

(b) `Actor alice2 = new Bug();` _____

(c) `Actor alice3 = new Flower();` _____

(d) `Bug bob1 = new Actor();` _____

(e) `Flower rose1 = new Actor();` _____

(f) `Flower rose2 = new Flower(Color.RED);` _____

(g) `BoxBug boxy1 = new BoxBug();` _____

(h) `BoxBug boxy2 = new BoxBug(5);` _____

(i) `Bug boxy3 = new BoxBug(5);` _____

Explain your answers and write a program to test them.

17.■ Write a class `WiltingFlower` as a subclass of `Flower`. Provide one constructor that takes `WiltingFlower`'s life span as a parameter and saves it in a private field. Another field, `age`, should be initialized to 0. On each step (each call to the `act` method) the age of the `WiltingFlower` should increase by 1. Once the age exceeds the life span, the flower should "die" (call its `removeSelfFromGrid` method). ≤ Hint: do not start from scratch — adapt `WiltingFlower` from `BoxBug`. ≥

18.■ The BoxBug class has one constructor, which takes one integer parameter:

```
/**
 * Constructs a box bug that traces a square of a given
 * side length
 * @param length the side length
 */
public BoxBug(int length)
{
  steps = 0;
  sideLength = length;
}
```

(The side of the square traced by the bug is measured between the centers of the grid cells where the bug turns. When `sideLength` is set to `0`, the bug keeps turning in one cell and never moves.)

Add three more constructors to this class: (1) a no-args constructor, which sets bug's color to `Color.red` and `sideLength` to 0; (2) a constructor that takes one parameter, `bugColor`, and sets the bug's color to `bugColor` and `sideLength` to 0; and (3) a constructor that takes two parameters, `bugColor` and `length`, and sets bug's color to `bugColor` and `sideLength` to `length`. Run the BoxBugRunner program and test the added constructors interactively, using GridWorld's direct manipulation interface feature.

19.■ Make rocks "roll." Derive a class `RollingRock` from `Rock`. Provide two constructors similar to the `Rock` constructors, but make them set `RollingRock`'s direction to southeast (135 degrees). Add a `roll` method, identical to `Bug`'s `move` method (only do not add a flower to the grid when the rock rolls). Provide an `act` method: make it simply always call `roll`. Supply a class `RockRunner` that places a few `RollingRock` objects into the grid and test your program.

20.◆ Derive a class `CarefulBug` from `Bug`. A `CarefulBug` acts as follows. Like a `BoxBug`, it counts the steps (calls to its `act` method). On count zero, it turns 45 degrees counterclockwise and increments the steps count by 1. On count one, it returns back to its original direction and increments the steps count by 1. On count two it turns 45 degrees clockwise and increments the steps count by 1. On count three it returns to its original direction and increments the steps count by 1. On count four it acts like a regular bug (if it can move, it moves, otherwise it turns) and resets the steps count to zero.

Provide two constructors similar to `Bug`'s constructors. They should also set the steps count to 0. Add a method `turnLeft` that turns the bug 45 degrees counterclockwise. Override `Bug`'s `act` method, calling `turnLeft`, `turn`, `canMove`, and `move` as necessary.

Add an orange `CarefulBug` to `world` in the `BoxRunner` program to test your `CarefulBug` class.

Hint:

```
if (steps == 0)  // If steps is equal to 0
{
  ...            //   do this
}
else if (steps == 1)
{
  ...
}
...
```