

## Chapter 2

# Simple Java Applications

### *2.1 An Application and its Architecture*

### *2.2 How to Build and Execute an Application*

#### *2.2.1 Using an IDE*

#### *2.2.2 Using the JDK*

### *2.3 How the Application Works*

#### *2.3.1 An Execution Trace of the Application*

### *2.4 How One Object Constructs Another*

### *2.5 Repairing Compiler Error Messages*

### *2.6 Summary*

### *2.7 Programming Exercises*

### *2.8 Beyond the Basics*

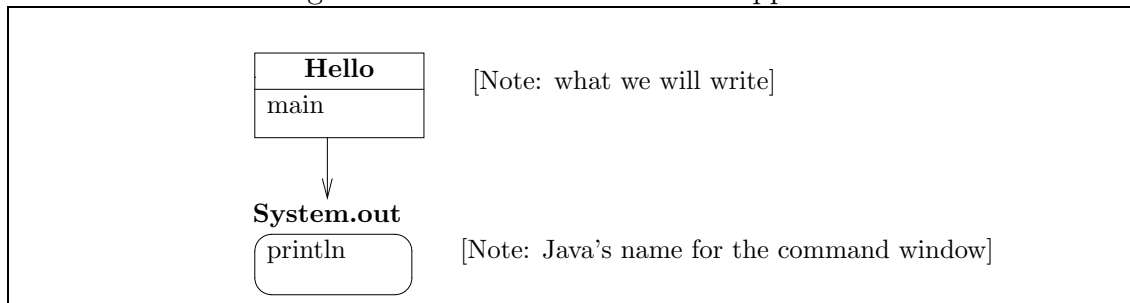
*This chapter applies concepts from Chapter 1 to building simple programs in the Java language. Its objectives are to*

- *present the standard format of a Java program and illustrate the instances of class, method, and object, as they appear in Java.*
- *explain the steps one takes to type a program, check its spelling and grammar, and execute it*
- *show how an object can send messages to other objects and even create other objects as it executes.*

## 2.1 An Application and its Architecture

If you were asked to build a robot, what would you do? First, you would draw a picture of the robot; next you would write the detailed instructions for its assembly. Finally, you would use the detailed instructions to build the physical robot. The final product, a robot “object,” has buttons on its chest, that when pushed, cause the robot to talk, walk, sit, and so forth. The buttons trigger the robot’s “methods”—the activities the robot can perform.

Figure 2.1: architecture of initial application



As we learned in the previous chapter, program construction follows the same methodology: We draw an initial design, the class diagram, and for each component (class) in the diagram, we write detailed instructions in Java, which are saved in a file in secondary storage. When we start the program, the class is copied into primary storage and becomes an executing *object*.

The Java designers use the term *application* to describe a program that is started by a human user. After an application's object is constructed in primary storage, a message is automatically sent to the application's `main` method, causing its instructions to execute. Therefore, every Java application must possess a method named `main`.

*Begin Footnote:* Not surprisingly, the precise explanation of how an application starts execution is more complex than what is stated here, but we nonetheless stick with our slogan that “classes are copied into primary storage and become executing objects.” *End Footnote*

To illustrate these ideas, we construct a small Java application that contains just the one method, `main`, which makes these two lines of text appear in the command window:

```

Hello to you!
49
  
```

To make the text appear, our application sends messages to a pre-existing object, named `System.out`, which is Java's name for the command window. The application we build is named `Hello`; it interacts with `System.out` in the pattern portrayed in the class diagram of Figure 1.

The class diagram shows that `Hello` has a `main` method; a message from the “outside world” to `main` starts the application. The other component, `System.out`, has its name underlined to indicate that it is a pre-existing object—a Java object that is already connected to the command window. The object has a method, `println` (read this as “printline”), which knows how to display a line of text in the command window. The Java designers have ensured that a `System.out` object is always ready and waiting to communicate with the applications you write.

The arrow in the diagram indicates that `Hello` sends messages to `System.out`. That is, it uses `System.out` by communicating with (or “connecting to”) its `println` method.

Using terminology from Chapter 1, we say that Figure 1 presents an architecture where `Hello` is the *controller* (it controls what the program does) and `System.out` is the *output view* (it displays the “view” of the program to the human who uses it).

We write and place the instructions for `class Hello` in a file named `Hello.java`. Java instructions look like “technical English,” and the contents of `Hello.java` will look something like this:

```
public class Hello
{ public static void main(String[] args)
  {
    ... to be supplied momentarily ...
  }
}
```

Java is a wordy programming language, and we must tolerate distracting words like `public`, `static`, `void`, which will be explained in due course. For now, keep in mind that we are writing a class that has the name `Hello` and contains a method named `main`. The set braces act as “punctuation,” showing exactly where a method and a class begin and end. The ellipses indicate where the instructions will be that send messages to `System.out`.

For our example, `main` must contain instructions that display two full lines of text. The algorithm for this task goes:

1. Send a message to `System.out` to print “Hello to you!” on a line of its own in the command window.
2. Send a message to `System.out` to print the number 49 on a line of its own in the command window.

We must convert the above algorithm into Java instructions and insert them where the ellipses appeared above. Step 1 of the algorithm is written like this in Java:

```
System.out.println("Hello to you!");
```

This instruction sends to `System.out` the message to print-line (`println`) the text, “Hello to you!”. In similar fashion, Step 2 is written

```
System.out.println(49);
```

The technical details behind these two instructions will be presented momentarily; for now, see Figure 2 for the completely assembled program.

Before we dissect Figure 2 word by word, we demonstrate first how one types the program into the computer, checks the program’s spelling and grammar, and starts it.

Figure 2.2: sample application

```
/** Hello prints two lines in the command window */
public class Hello
{ public static void main(String[] args)
  { System.out.println("Hello to you!");
    System.out.println(49);
  }
}
```

## 2.2 How to Build and Execute an Application

We take several steps to make the application in Figure 2 print its text in the command window:

1. `class Hello` must be typed and saved in the file, `Hello.java`.
2. The program's spelling and grammar must be checked by the Java compiler; that is, the program must be *compiled*.
3. The program must be started (*executed*).

To perform these steps, you use either your computer's *(i)* integrated development environment (IDE) for the Java language, or *(ii)* text editor and the Java Development Kit (JDK).

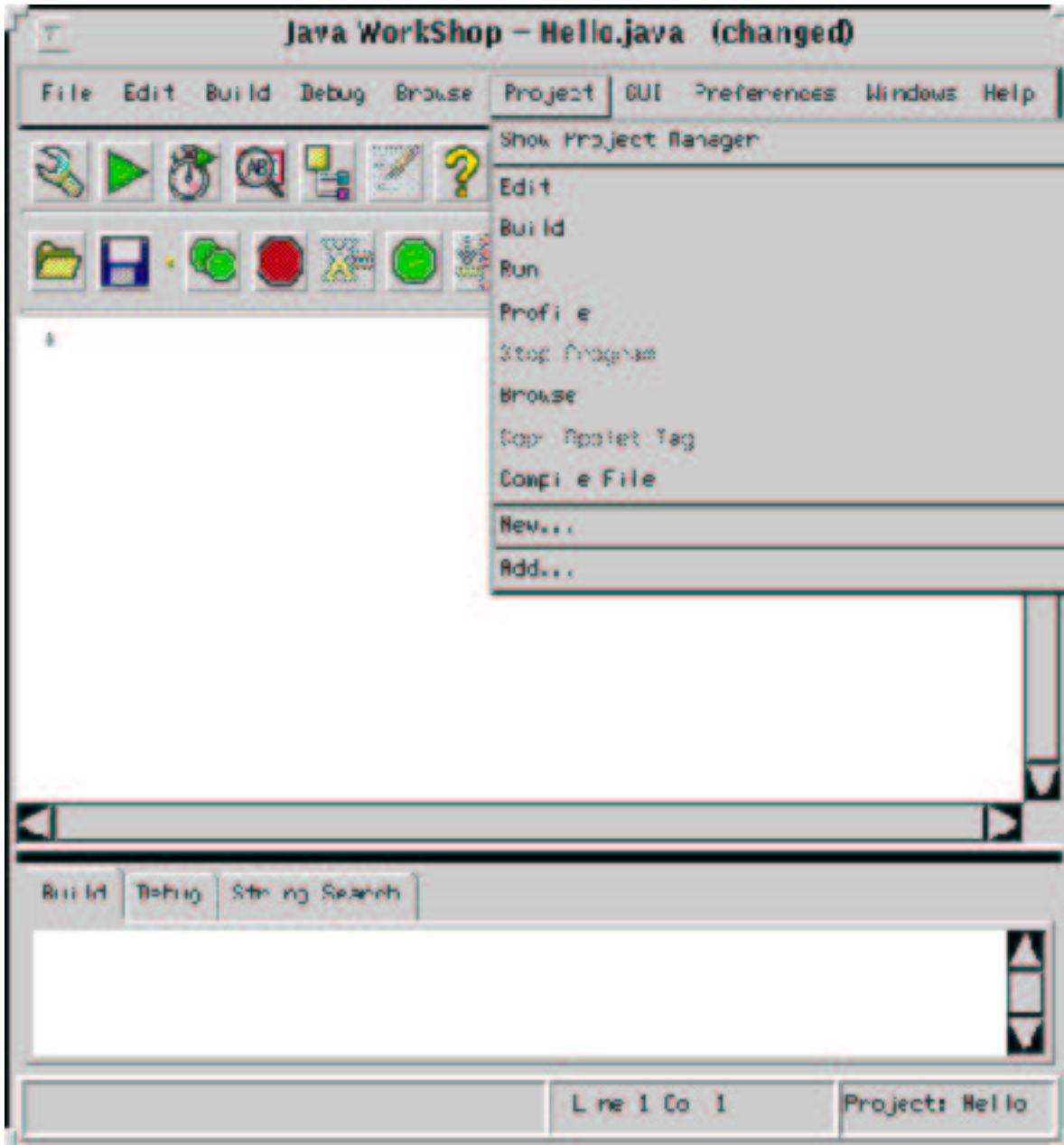
We briefly examine both options, and you should obtain help with selecting the one you will use. If you are not interested in this selection at this moment in time, you may skip either or both of the two subsections that follow.

### 2.2.1 Using an IDE

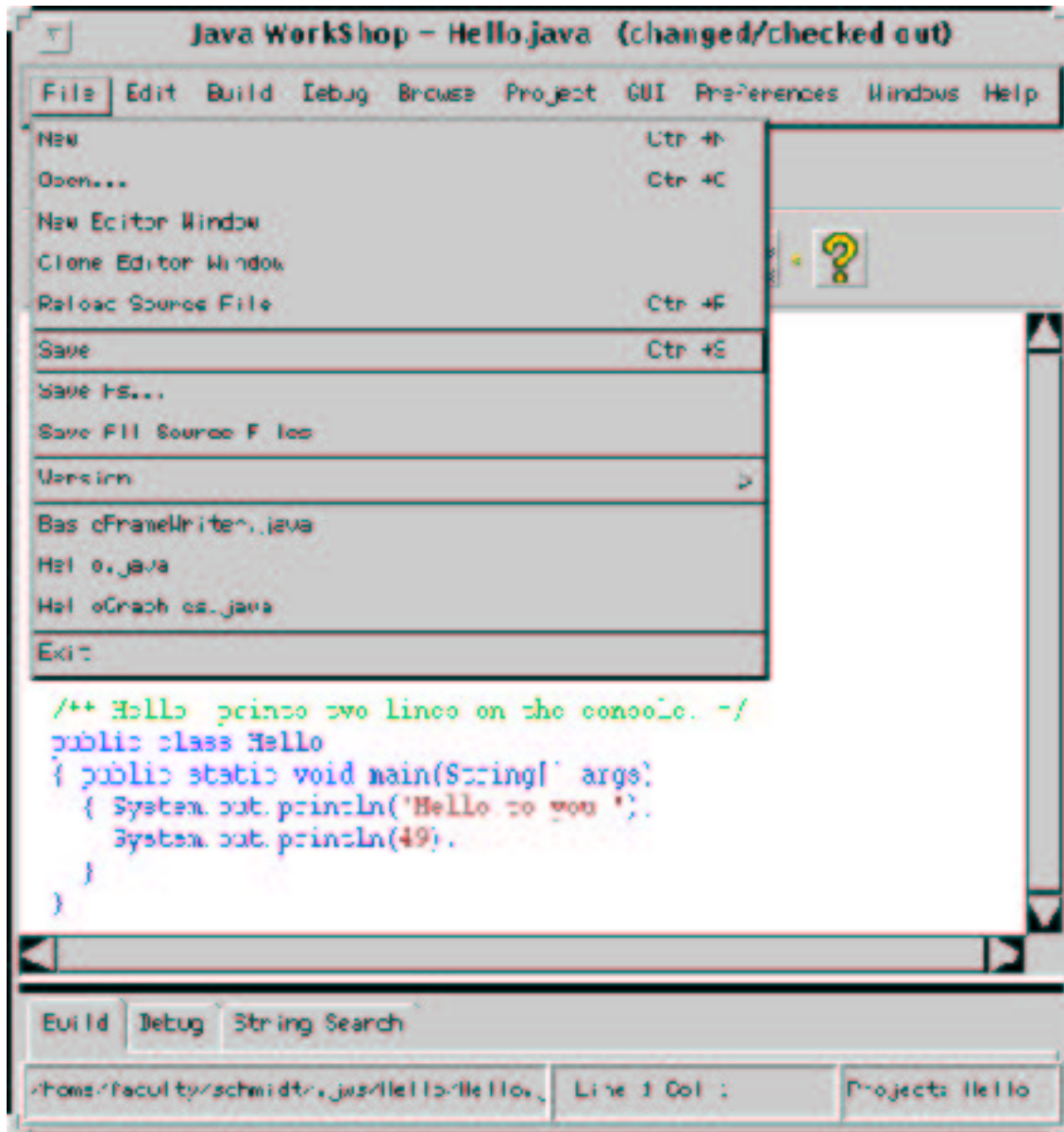
There are many IDEs available for Java, and we cannot consider them all. Fortunately, IDEs operate similarly, so we present a hypothetical example. *Be certain to read the manual for your IDE before you attempt the following experiment.*

When an IDE is started, it will present a window into which you can type a Java application (or "project," as the IDE might call it). Select the IDE's menu item or button named *New Project* to create a new project, and when the IDE asks, type the name of the class you wish to write. For the example in Figure 2, use the name,

Hello, for the project:



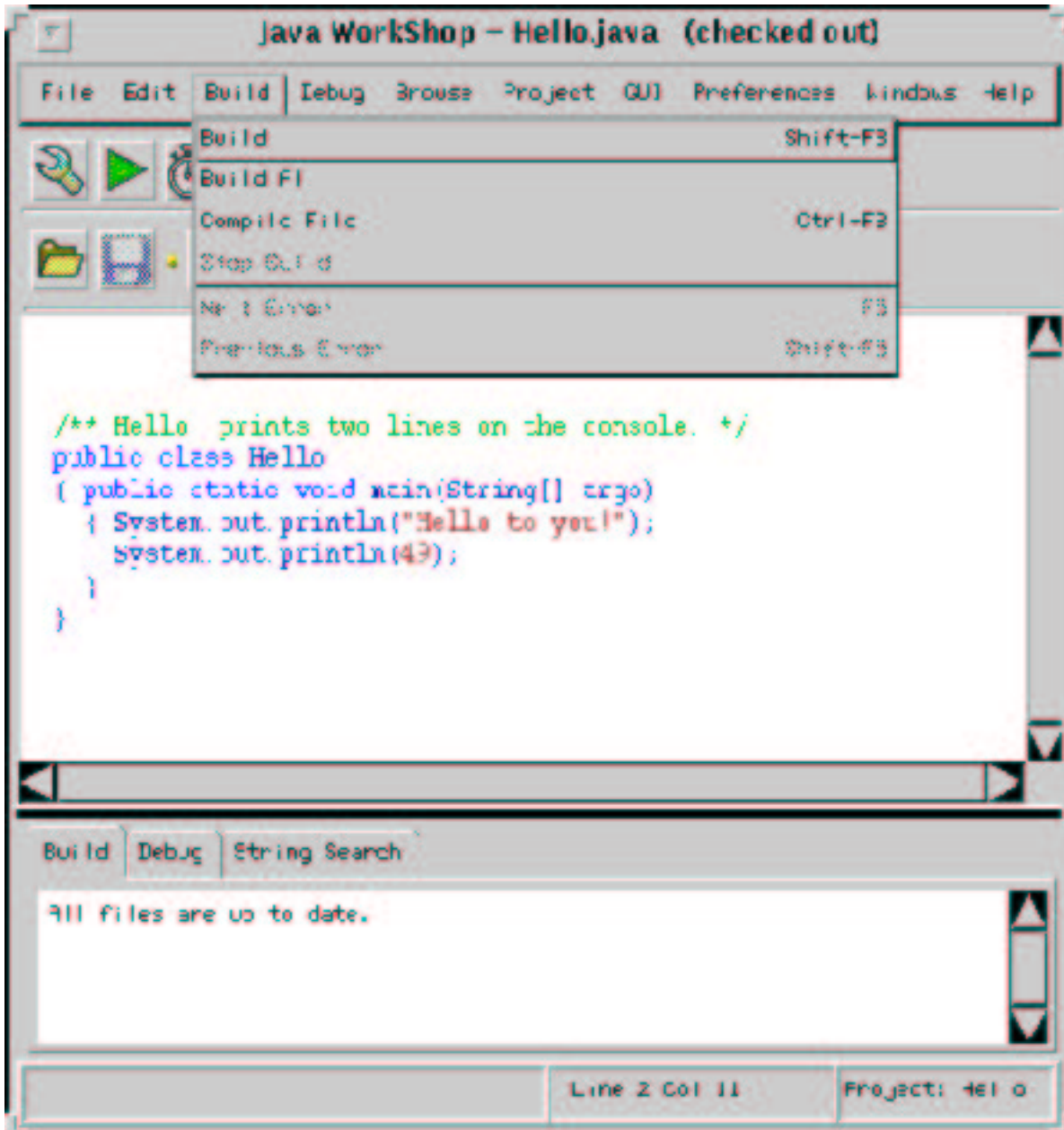
Next, type the class into the window and save it by using the IDE's *Save* button:



If done properly, class `Hello` will be saved in a file named `Hello.java`.

Next, the program must be compiled. Compile by selecting the IDE's *Compile* or

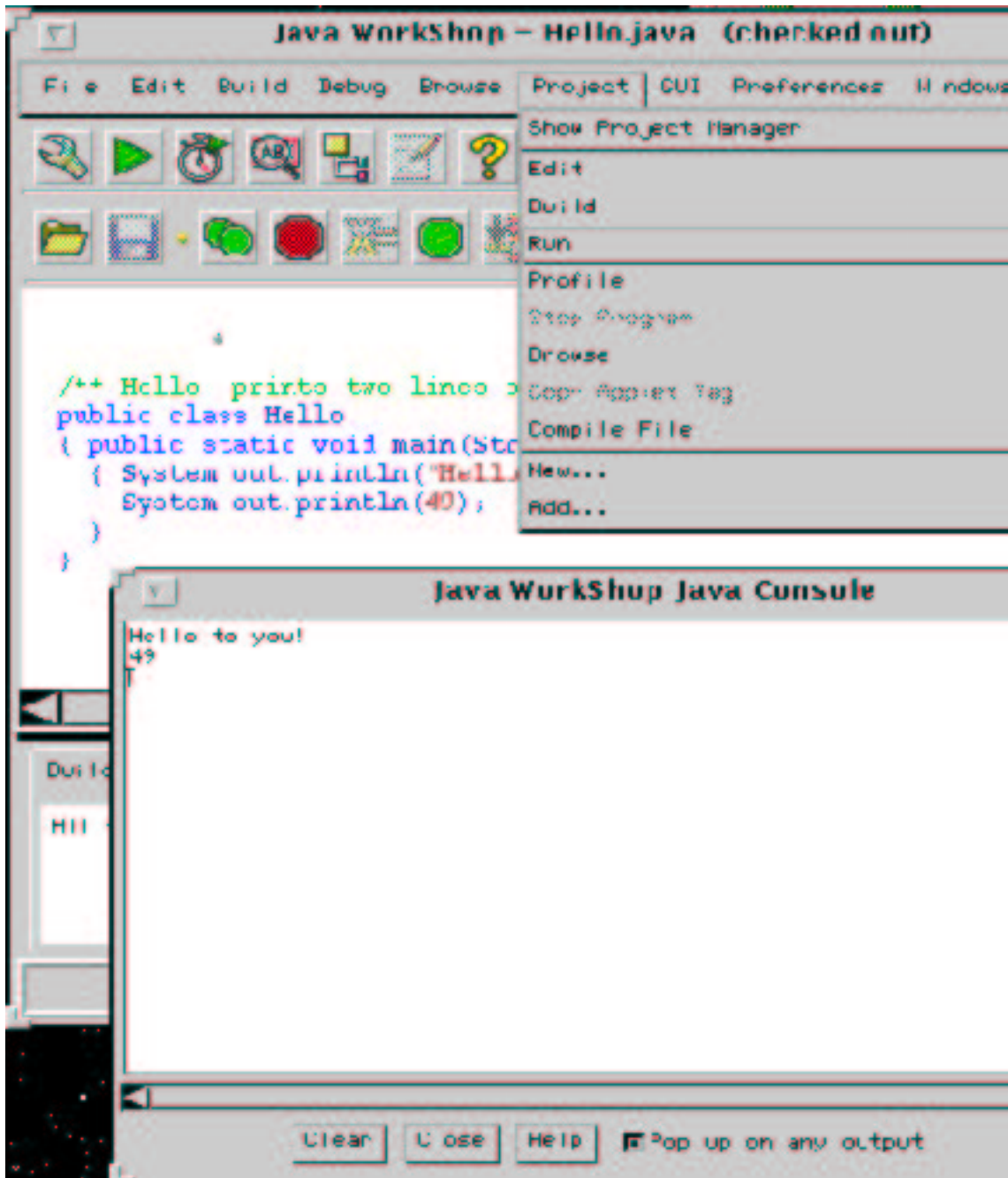
*Build* button:



If there are any spelling or grammar errors, they will be listed in a small window of their own; otherwise, a message will announce that the compile has completed successfully. In the former case, you repair the errors and try again; in the latter case, you will see that the compiler has created the translated version of the application and placed it in the file, `Hello.class`.

Finally, start the program by selecting the button named *Run* or *Launch* (or *Start*, etc.) This starts the Java interpreter, which helps the processor read and execute

the byte-code instructions in `Hello.class`. In the case of the `Hello` application, its execution causes two full lines of text to appear in the command window. The IDE shows the command window when needed:



(Footnote: Unfortunately, your IDE might show and remove the command window before you can read the text that appeared! If the command window disappears too quickly, insert these lines into your Java program immediately after the last



System.out.println:

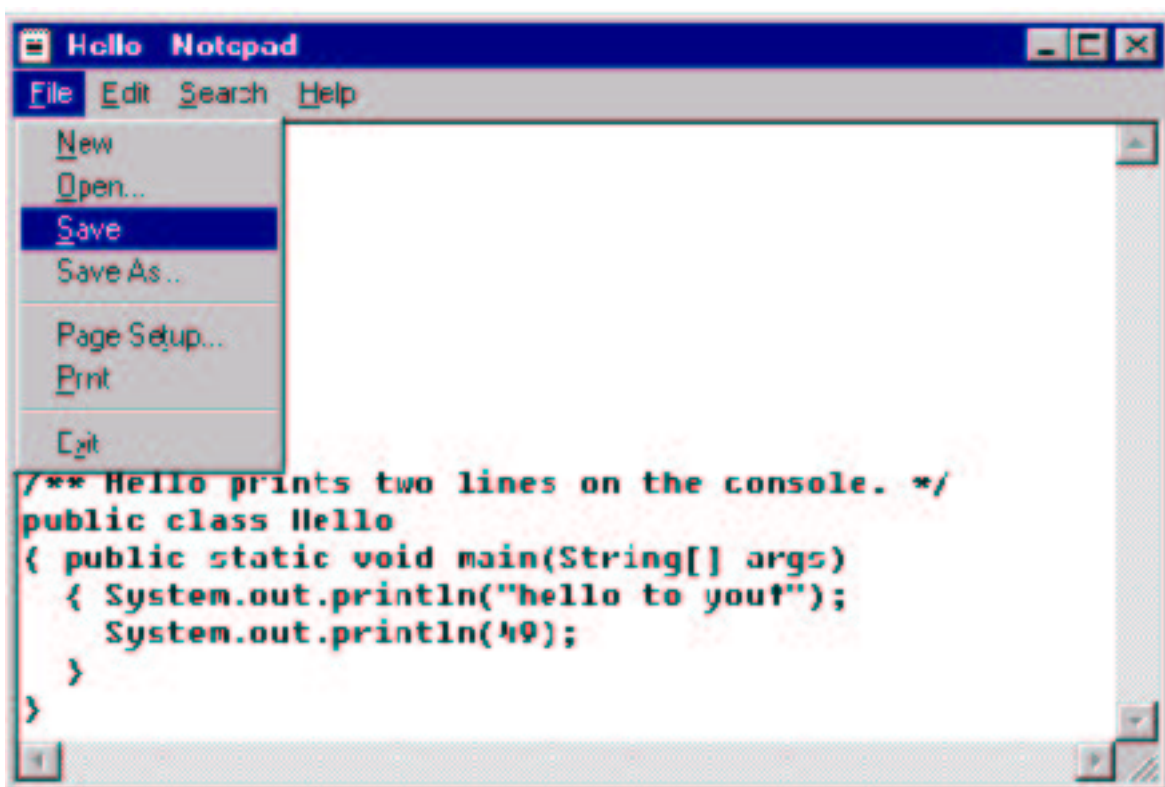
```
try { Thread.sleep(5000); }  
catch (InterruptedException e) { }
```

These cryptic extra lines delay the program by 5 seconds, giving you time to read the contents of the command window. *EndFootnote.*)

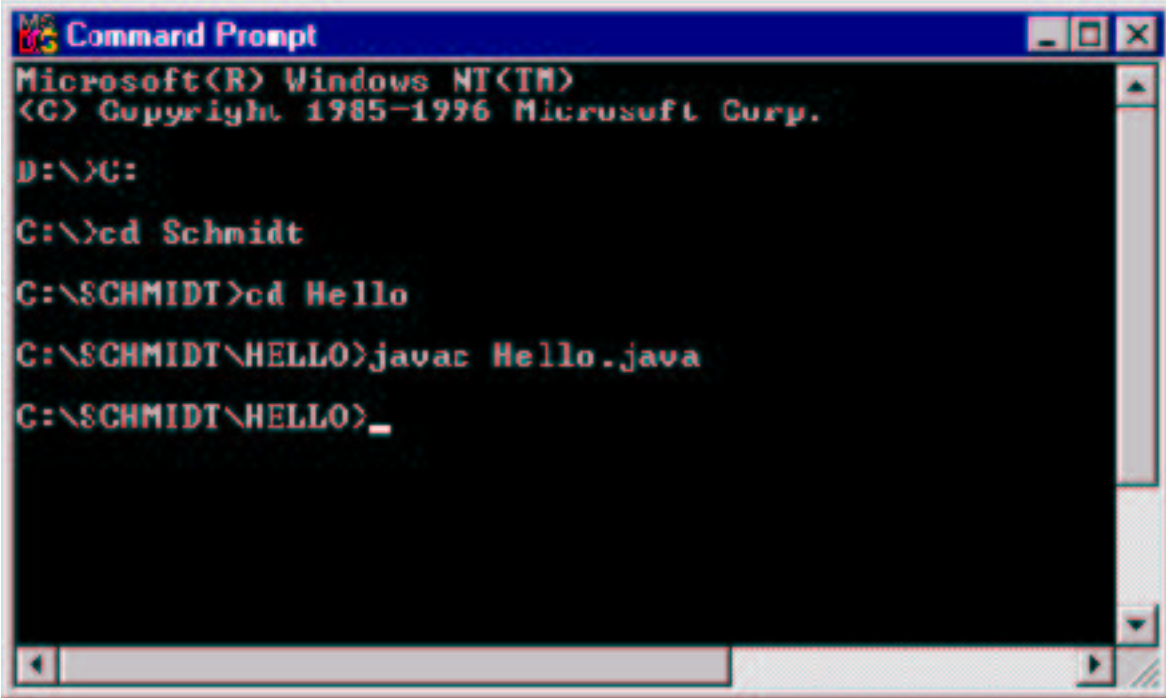
## 2.2.2 Using the JDK

An alternative to an IDE is a text editor and the Java Development Kit (JDK). Your computer should already have a text editor (e.g., Notepad or emacs or vi) that you can use to type and save files. The JDK can be obtained from Sun Microsystems's Web Site at <http://java.sun.com>. You must download the JDK and properly install it on your computer; installation is not covered here, but instructions come with the JDK.

Once you have the JDK installed, the first step is to use a text editor to create the file, Hello.java, with the contents of Figure 2:



Next, compile the application by typing in a command window, `javac Hello.java`:



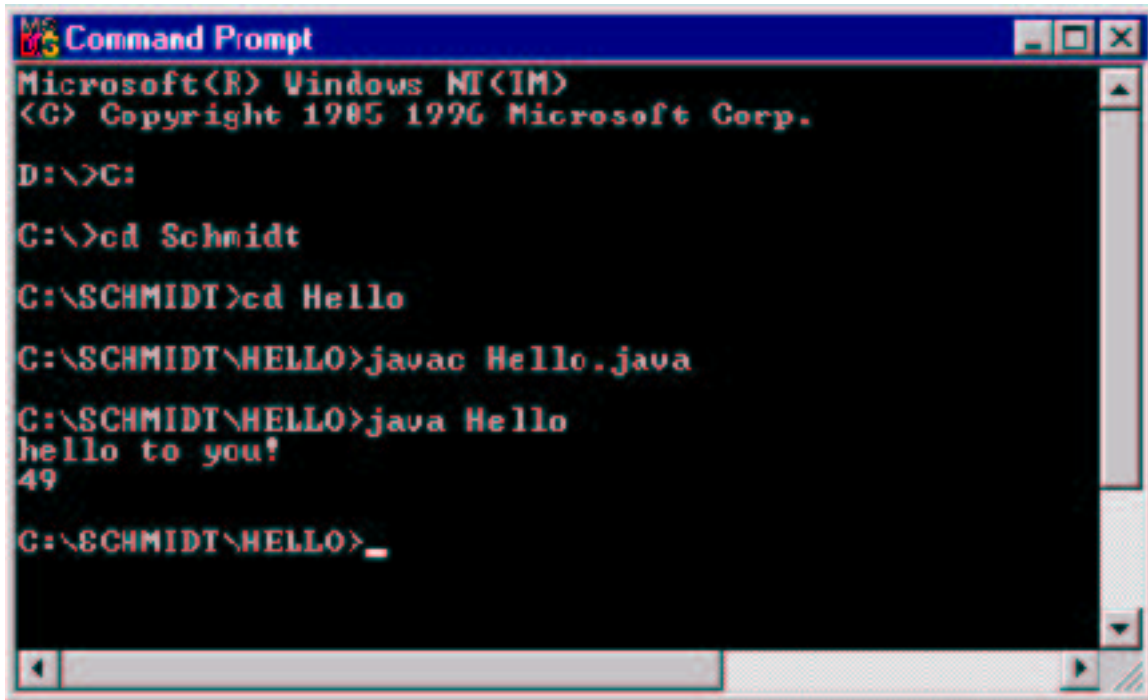
```
Microsoft Windows [Version 5.00.4779.0131] Copyright (c) 2009 Microsoft Corporation
All rights reserved.

C:\> cd \
C:\> cd Schmidt
C:\Schmidt> cd Hello
C:\Schmidt\Hello> javac Hello.java
C:\Schmidt\Hello> _
```

This starts the Java compiler, which examines the application, line by line, attempting to translate it into byte-code language. If the program is badly formed—there are spelling or grammar or punctuation errors—then the compiler will list these errors. If there are errors, correct them with the text editor and compile again; if not, then you will see that the Java compiler created the translated version of the application, a byte-code file named `Hello.class`.

To execute the program, type the command, `java Hello`. This starts the program, more specifically, it starts the Java interpreter that helps the processor execute the

byte-code instructions, and the two lines of text appear in the command window:



```

Microsoft Windows [Version 5.00.4779.0131] Copyright (c) 2009 Microsoft Corporation
C:\> cd C:\SCHMIDT
C:\SCHMIDT> cd HELLO
C:\SCHMIDT\HELLO> javac Hello.java
C:\SCHMIDT\HELLO> java Hello
hello to you!
49
C:\SCHMIDT\HELLO>

```

Details about error messages and error correction appear in a section later in this Chapter.

### Exercise

Install either an IDE or the JDK on your computer and type, compile, and execute the program in Figure 2.

## 2.3 How the Application Works

We now scrutinize Figure 2 line by line. The class's first line,

```
/** Hello prints two lines in the command window */
```

is a *comment*. A comment is not a Java instruction for the computer to execute—it is a sentence inserted, as an aside, for a human to read, in the case that the human wants to inspect the application before it is compiled and executed. The comment is meant to explain the purpose of `class Hello`. A comment can extend over multiple lines, as we will see in later examples.

*You should begin every class you write with a comment that explains the class's purpose.* Java programs are not all that easy for humans to read, and a few lines of explanation can prove very helpful!

*Begin Footnote:* By the way, it confuses the Java compiler if you place one comment inside another, e.g.

```
/** A badly formed /** comment */ looks like this. */
```

so do not do this! Also, the Java compiler will accept comments that begin with `/*` as well as `/**`, but we use the latter for reasons explained in Chapter 5. *End Footnote.*

Following the comment is the line that gives the class's name, and then there are two matching brackets:

```
public class Hello
{
    ...
}
```

The line with the program's name is the program's *header line*. `Hello` is of course the name, but the words `public` and `class` have special, specific meanings in Java.

Words with special, specific meanings are called *keywords*. The keyword, `public`, indicates that the class can be used by the "general public," which includes other objects and human users, to build objects on demand. The keyword, `class`, indicates of course that `Hello` is a Java class, the basic unit of program construction.

Following the program's header line is the program's *body*, which is enclosed by the matching brackets, `{` and `}`. Within the body lie the program's methods (as well as other items that we encounter in later chapters).

In the examples in this text, we will align the brackets vertically so that they are easy to see and match. The only exception to this rule will be when the brackets enclose a mere one-line body; then, the closing bracket will be placed at the end of the same line, to save space on the printed page. (See Figure 2 again.) When you type your own Java programs, you might prefer to use this style of bracket placement:

```
public class Hello {
    ...
}
```

because it is easier to use with a text editor. (But this style makes it easy to forget a bracket, so be careful.) Do as your instructor indicates; we will not discuss this issue again.

Back to the example—`Hello`'s body has but one method, `main`, which has its own header line and body:

```
public static void main(String[] args)
{
    ...
}
```

Method `main` is surrounded by a slew of keywords! The word, `public`, means the same as before—the `main` method may receive messages from the general “public.” (This includes the user who starts `Hello` and wants `main` to execute!) The keyword, `static`, refers to a subtle, technical point, and its explanation, along with that of `void` and `(String[] args)`, must be postponed.

The body of the `main` method contains the instructions (*statements*) that will be executed when a `Hello` object is sent the message to execute its `main` method. These instructions are

```
System.out.println("Hello to you!");
System.out.println(49);
```

Each of the statements has the form, `System.out.println(SOMETHING_TO_DISPLAY);` this statement form sends a message to the `System.out` object, and the message is `println(SOMETHING_TO_DISPLAY)`. The `SOMETHING_TO_DISPLAY`-part is the *argument* part of the message. The statement must be terminated by a semicolon. (Semicolons in Java are used much like periods in English are used.)

This message asks `System.out` to execute its `println` method. The instructions inside the `println` method display the argument, `SOMETHING_TO_DISPLAY`, at the position of the cursor in the command window. And, `println` inserts a *newline* character immediately after the displayed argument—this makes the cursor in the command window move to the start of a fresh line.

As we learn in the next chapter, the Java language requires that textual phrases must be enclosed within double quotes, e.g., `"Hello to you!"`. Textual phrases are called *strings*. The double quotes that surround strings prevent confusing strings with keywords. For example, `"public class"` is clearly a string and *not* two keywords side by side. In contrast, numbers like `49` are not strings and are not enclosed by double quotes.

*Begin Footnote:* But note that `"49"` is a string, because of the double quotes. The distinction between `49` and `"49"` becomes clear in the next chapter. *End Footnote*

Method `println` serves as `System.out`’s “communication” or “connection” point, and the `Hello` object makes two such communications with `System.out` by using `println`, not unlike contacting a friend twice on the telephone.

Here is a final point, about spelling: *The Java language distinguishes between upper-case and lower-case letters.* Therefore, `public` is spelled differently from `PUBLIC` and `Public`, and only the first is accepted as the correct Java keyword. It is traditional to spell the names of classes beginning with a single upper-case letter, e.g., `Hello`, and method names traditionally begin with a lower-case letter, e.g., `println`. Additional spelling guidelines are introduced in the next chapter.

## Exercises

1. Write a Java program that displays your name on one line.

2. Write a Java program that prints your name, where your first (given) name is on a line by itself, and you last (family) name is on a line by itself.
3. For both of the previous exercises, explain why your program has the architecture presented in Figure 1. (This makes clear that different programs can be built in the same architectural style.)

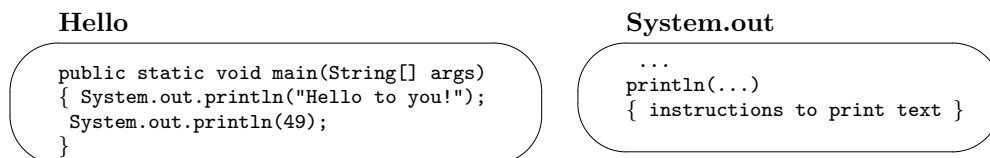
### 2.3.1 An Execution Trace of the Application

When we demonstrated the `Hello` application, we saw the lines, `Hello to you!` and `49` appear in the command window, but we did not see how the computer executed the application's instructions to produce these results. To be a competent programmer, you must develop mental images of how the computer executes programs. To help develop this skill, we study a step-by-step pictorial re-creation—an *execution trace*—of the computer's actions.

When the `Hello` application is started by a user, the file, `Hello.class`, is copied into primary storage and becomes a `Hello` object.

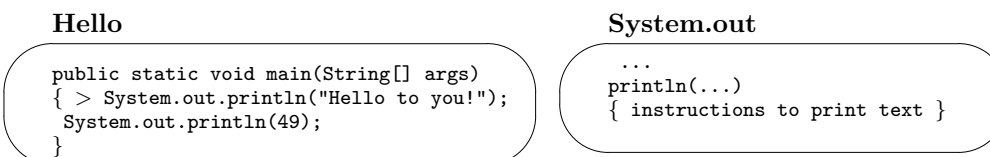
*Begin Footnote:* The preceding explanation is imprecise, because it fails to explain that the “object” created from `Hello.class` is not the usual form of object described in Chapter 1. But it is not worth the pain at this point to state the distinction between a class that possesses an invoked `static main` method and one that possesses an invoked non-`static` constructor method. Our situation is similar to the one in an introductory physics course, where Newtonian physics is learned initially because it is simple and beautiful although slightly incorrect. In subsequent chapters, these technical issues are resolved. *End Footnote*

Here is a depiction of the `Hello` object (and `System.out`, which already exists) in primary storage:



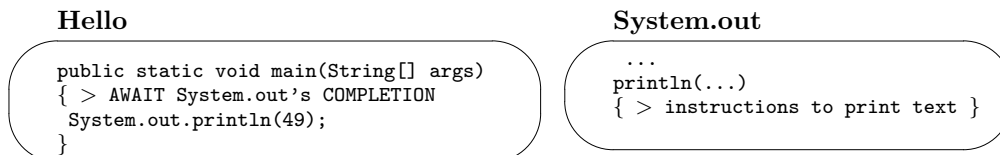
Objects rest in primary storage and wait for messages. So, to start the computation, the computer (more precisely, the JVM) sends `Hello` a message to start its `main` method.

When `Hello` receives the message, it indeed starts `main`, whose statements are executed, one by one. We use a marker, `>`, to indicate which statement is executed first:

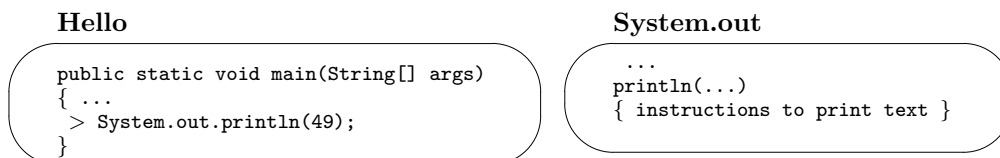


The statement, `System.out.println("Hello to you!")`, sends the message, `println("Hello to you!")`, to the `System.out` object. The argument, "Hello to you!", is what the method will print. (The enclosing double quotes are not printed, but you must include them, nonetheless.) This is an example of how one object sends a message to another object, awakening it from its rest.

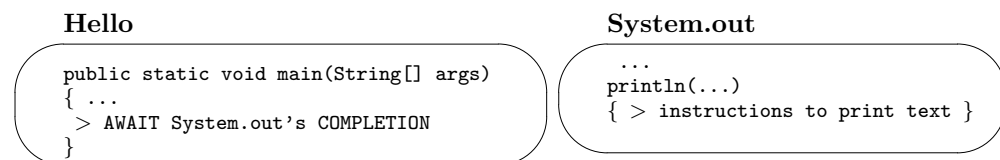
The message is delivered to `System.out`, which puts its `println` method to work. In the interim, the `Hello` object waits:



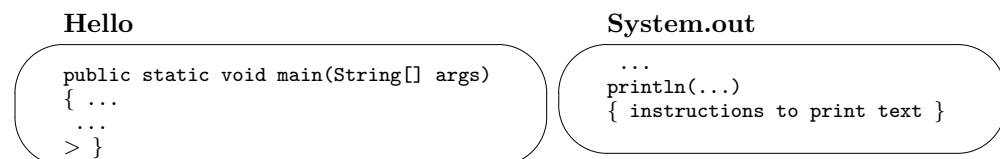
Eventually, `System.out` fulfills its request and the text, `Hello to you!` appears in the command window. Once this happens, `System.out` signals `Hello` that it can proceed to its next statement, and `System.out` returns to a “resting” state:



Next, another message is sent to `System.out`, requesting that its `println` method display the number 49. (Notice that double-quote marks are not used around numbers.) Again, `Hello` waits until `System.out` executes its message and signals completion:



Once `println` finishes, there is nothing more to execute in `main`'s body,

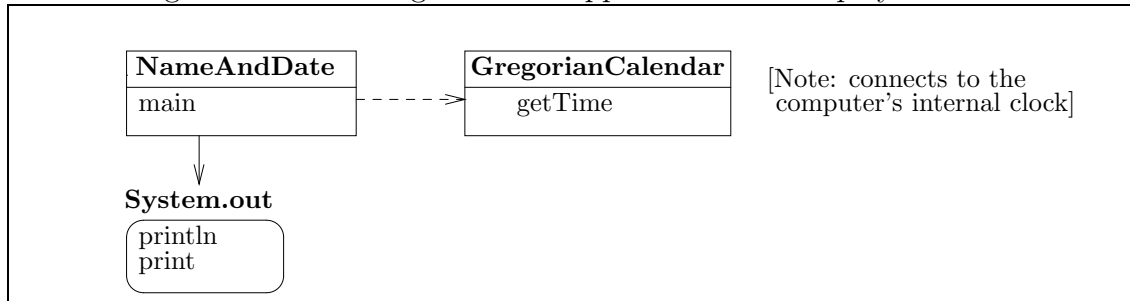


and `Hello` signals that it is finished.

## Exercise

Write an execution trace for a program you wrote from the earlier Exercise set.

Figure 2.3: class diagram of an application that displays the time



## 2.4 How One Object Constructs Another

The `Hello` example showed how an object can send messages to the preexisting object, `System.out`. But it is also possible for an object to construct its own, “helper,” objects as needed and send them messages. We see this in an example.

Say that you want an application that prints your name and the exact date and time, all on one line, in the command window. Of course, the date and time are dependent on when the program starts, and you cannot guess this in advance. Fortunately, the Java designers wrote a class, named `class GregorianCalendar`, that knows how to read the computer’s internal clock. So, we construct an object that itself constructs a `GregorianCalendar` object and sends the `GregorianCalendar` object a message for the exact date and time. Figure 3 shows the architecture of the application we plan to build. The class we write is called `NameAndDate`. It will send a `getTime` message to the `GregorianCalendar` and then it will send messages to `System.out` to print the time. (The new method, `print`, of `System.out`, is explained below.)

When this program is executed, we will see in the command window something like this:

```
Fred Mertz --- Fri Aug 13 19:07:42 CDT 2010
```

```
Finished
```

That is, the programmer’s name, Fred Mertz, is printed followed by the exact moment that the program executes. Then an empty line and a line consisting of just `Finished` appears.

Figure 3 has an interesting architecture: `NameAndDate` is the controller, because it controls what will happen; `System.out` is the output view, because it presents the “view” of the program to its user; and `GregorianCalendar` is the *model*, because it “models” the computer clock—this is our first, simplistic example of a *model-view-controller* (MVC) architecture—see Chapter 1.

We must write the controller, `class NameAndDate`. The controller’s `main` method asks `System.out` to print the text in the command window, but `main` must also ask



a `GregorianCalendar` object for the exact time. To do this, `main` must *construct* the object first before it sends it a message. Here are the crucial steps `main` must do:

- Construct a new `GregorianCalendar` object, by saying

```
new GregorianCalendar()
```

The keyword, `new`, constructs an object from the class named `GregorianCalendar`. The matching parentheses, ( and ), can hold extra arguments if necessary; here, extra arguments are unneeded to construct the object.

- Give the newly created object a *variable name* so that we can send messages to the object:

```
GregorianCalendar c = new GregorianCalendar();
```

Here, the name is `c`. (The extra word, `GregorianCalendar`, is explained later.) We can use the name `c` like we used the name, `System.out`—we can send messages to the named object.

- Send object `c` a `getTime` message that asks it to consult the clock and reply with the date and time:

```
c.getTime()
```

- Tell `System.out` to display this date and time:

```
System.out.println(c.getTime());
```

Recall that the statement, `System.out.println(ARGUMENT)`, prints the `ARGUMENT`. Here, we insert, `c.getTime()`, which asks `c` to reply with the time as a response—*it is the time received in response that is printed*.

Figure 4 shows the program that uses the above steps. In addition to what we have just studied, several other new techniques are illustrated in the example. We explain them one by one.

To understand the first line,

```
import java.util.*;
```

we must provide some background: The Java language comes with many prewritten classes for your use. The classes are grouped into *packages*, and the package where one finds class `GregorianCalendar` is `java.util`. The statement, `import java.util.*`, tells the Java interpreter to search in the `java.util` package, where it will locate class `GregorianCalendar`. Once the class is located, an object may be constructed

Figure 2.4: application that prints the date and time

```

import java.util.*;
/** NameAndDate prints my name and the exact date and time. */
public class NameAndDate
{ public static void main(String[] args)
  { System.out.print("Fred Mertz --- ");
    // The next statement creates an object:
    GregorianCalendar c = new GregorianCalendar();
    System.out.println(c.getTime()); // ask c the time and print its reply
    System.out.println();
    System.out.println("Finished");
  }
}

```

from it. It will always be clear in this text's examples when a special statement like `import java.util.*` is necessary.

*Begin Footnote:* Stated more precisely, when a class, `C`, is mentioned in a statement, the Java interpreter must locate the file, `C.class`. Normally, the interpreter searches in two places: (1) the folder in which the application was started, and (2) `java.lang`, a package of general-purpose classes. If `C.class` does not reside in either of these two places, an `import` statement must indicate where else the interpreter should search. *End Footnote*

The first statement within `main` says

```
System.out.print("Fred Mertz --- ");
```

Method `print` is another method that belongs to `System.out`. When executed, `print(ARGUMENT)` displays the `ARGUMENT` at the position of the cursor in the command window, *but no newline character is appended immediately after the displayed argument*—this retains the command window's cursor on the same line as the displayed argument so that additional information can be printed on the line.

The line,

```
// The next statement creates an object:
```

is an *internal comment*. An internal comment is a comment inserted inside a class, providing technical information to the person who is studying the program. An internal comment extends only from the double slashes until the end of the line on which it appears. In this text, we use internal comments to explain the workings of important or subtle statements.

The statement,

```
GregorianCalendar c = new GregorianCalendar();
```

was explained above: a new object is constructed from class `GregorianCalendar` and is named `c`. the reason for the apparently redundant leading word, `GregorianCalendar`, is best explained in the next chapter; for the moment, we note that the leading word states that `c` is a “type” of name only for `GregorianCalendar` objects.

Variable names are developed fully in the next chapter, and our use of one to name for the `GregorianCalendar` object is merely a “preview.”

The statement,

```
System.out.println(c.getTime());
```

has been explained; the date that is computed and returned by the `GregorianCalendar` object becomes the argument to the `println` message, and therefore it is the date (and *not* the message itself) that is displayed in the command window. We see this behavior explained in the execution trace that follows.

Finally,

```
System.out.println()
```

prints an argument that is nothing, to which is appended a *newline* character. The net effect is that the cursor in the command window is moved to the beginning of a fresh line. (The parentheses, `()`, hold no information to print.)

## Execution Trace

Because this example is an important one, we study its execution trace. The instant after the `NameAndDate` application is started, primary storage looks like this:

### NameAndDate

```
main
{ > System.out.print("Fred Mertz --- ");
  GregorianCalendar c = new GregorianCalendar();
  System.out.println(c.getTime());
  System.out.println();
  System.out.println("Finished");
}
```

### System.out

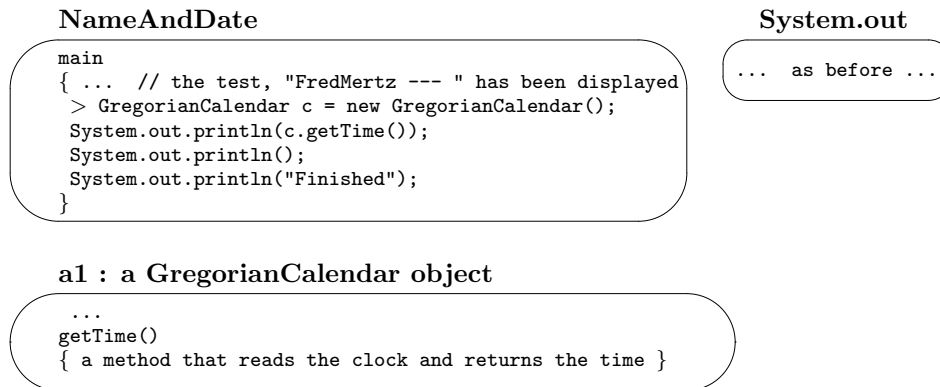
```
print(...)
{ instructions to print text }

println(...)
{ instructions to terminate text }
```

Once the text, `Fred Mertz ---`, is printed, the interpreter begins the next statement, which creates a new `GregorianCalendar` object—a segment of primary storage is allocated to hold the contents of a `GregorianCalendar` object, as described by class `GregorianCalendar`.

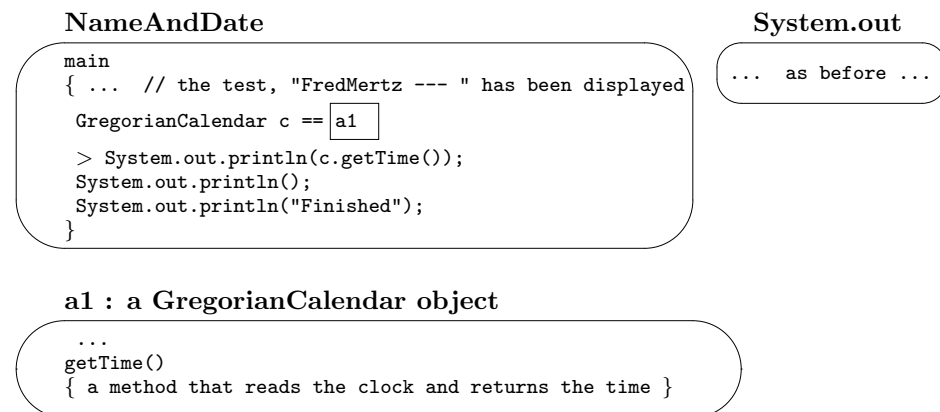
*Begin Footnote:* More specifically, the file, `GregorianCalendar.class` is used to construct the object in storage. *End Footnote*

We see the following:



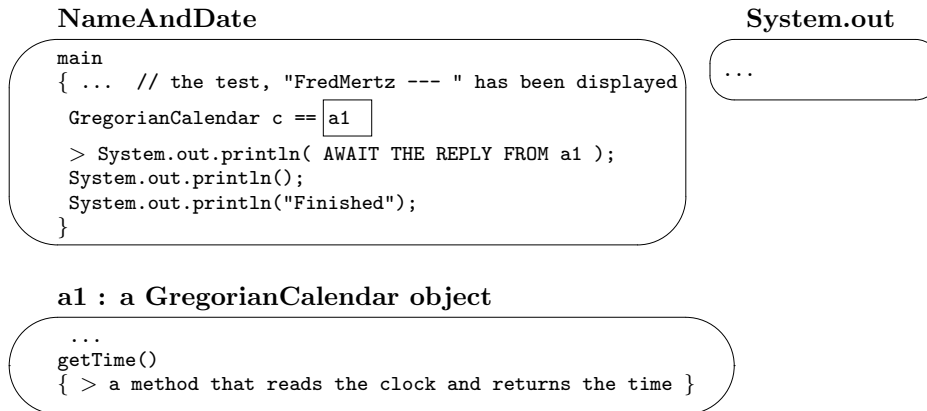
The new object has an internal *address* where it is found. Here, the address of the newly constructed `GregorianCalendar` object is `a1`. The object at address `a1` has its own internal structure, including a method named `getTime`.

Next, a storage space (called a *cell*) is created, and the address is placed into the cell. In this way, `c` is made to name the object at address `a1`:



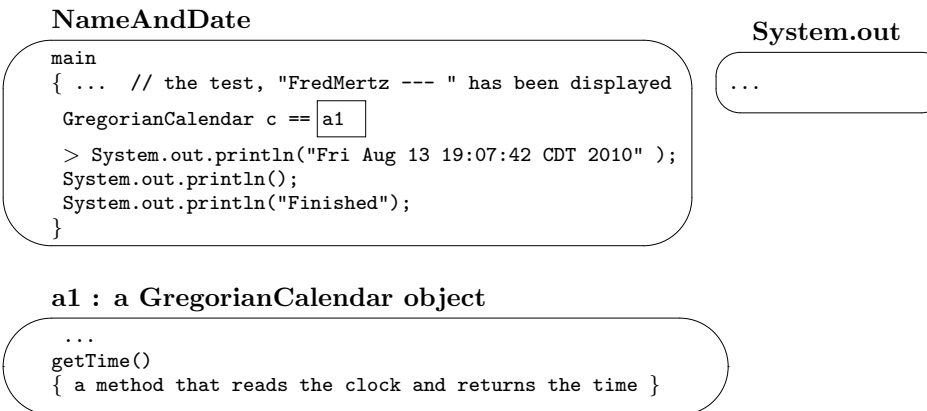
Now, a `println` message to `System.out` must be sent. *But the message's argument is not yet determined.* For this reason, the interpreter sends a `getTime()` message to `c`, and since `c` names a cell that holds address `a1`, the object at address `a1` is sent the

message to execute the instructions in its `getTime` method:



The main method waits while `a1`'s `getTime` method executes.

Once `getTime` gets the date and time from the computer's clock, *it returns the date and time to the exact position from which the message was sent*:



After the time is placed in the position where it was requested, it becomes the argument part of the `println` message that is sent to `System.out`. This is a pattern that appears often in Java statements: *When part of a statement is written within parentheses, the parenthesized part executes first, and the result, if any, is deposited in the parentheses.*

`System.out` displays the date and time, and execution proceeds to the last two statements in `main`.

Finally, we note that only one message is sent to the `GregorianCalendar` object that is constructed by the application. In the case that an object is constructed and just one message is sent to it immediately thereafter, we need not give the object a name. Here is the application rewritten so that the object's name no longer appears:

```
import java.util.*;
/** NameAndDate prints my name and the exact date and time. */
```

```

public class NameAndDate
{ public static void main(String[] args)
  { System.out.print("Fred Mertz --- ");
    // The next statement constructs an object and sends it a getTime message:
    System.out.println(new GregorianCalendar().getTime());
    System.out.println();
    System.out.println("Finished");
  }
}

```

The key statement,

```
System.out.println(new GregorianCalendar().getTime());
```

embeds within it the steps of *(i)* constructing a new `GregorianCalendar` object *(ii)* immediately sending the object a `getTime()` message, and *(iii)* using the message's reply as the argument to `System.out.println`. This complex arrangement executes correctly because the Java interpreter executes phrases within parentheses first.

### Exercise

Revise Figure 3 so that it prints your own name with the date and time. Compile and execute it. Execute it again. Compare the outputs from the two executions.

## 2.5 Repairing Compiler Error Messages

Humans can tolerate errors in spelling and grammar, but computers cannot, and for this reason, the Java compiler complains if a program contains any spelling or grammar errors. The compiler will identify the line on which an error appears and give a short explanation.

For example, here is a program that contains several errors. (Take a moment to locate them.)

```

public class test
{ public static main(String[] args)
  { System.out.println(Hello!)
}

```

Perhaps we save this program in `Test.java` and compile it. The compiler replies with several error messages, the first of which is

```

Test.java:2: Invalid method declaration; return type required.
{ public static main(String[] args)
      ^

```

This message states there is an error in Line 2 of the program, at approximately the position marked by the caret. The explanation of the error is not so helpful, but its position is important because we can compare the above to a correctly written program and notice that we forgot the keyword, `void`.

The compiler reports an error on the next line, also:

```
Test.java:3: ')' expected.
  { System.out.println>Hello!)
                        ^
```

Again, the compiler's message is not so helpful, but the position of the error suggests there is something wrong with the word, `Hello!`—we forgot to enclose the string within double quotes. Again, we can spot this error quickly by comparing our `println` statement to one in the text that was correctly written.

There is also an error in the following line:

```
Test.java:4: '}' expected.
}
^
```

This time, the explanation is on target—we are indeed lacking a closing bracket.

Finally, the compiler reports an error at Line 1:

```
Test.java:1: Public class test must be defined in a file called "test.java".
public class test
      ^
```

The compiler's message is correct—the name of the `class` must be spelled exactly the same as the name of the file in which the class is saved. Therefore, we should change the first line to read `public class Test`.

Once we repair the errors, we have this program:

```
public class Test
{ public static void main(String[] args)
  { System.out.println("Hello!") }
}
```

When we compile the revised program, the compiler identifies one error we missed:

```
Test.java:3: ';' expected.
  { System.out.println("Hello!") }
                        ^
```

Indeed, we have forgotten to end the statement with a semicolon. Once we repair this last error, the compiler will successfully compile our program. (Unfortunately, a compiler is not perfect at detecting all grammar errors on a first reading, and it is

common that an highly erroneous program must be compiled several times before all its grammar errors are exposed.)

Although the Java compiler will methodically locate all the grammar errors in an application, a programmer should *not* rely on it as a kind of “oracle” that announces when an application is ready to be executed. Indeed, a programmer should be familiar with the appearance (*syntax*) and meanings (*semantics*) of the statements in the Java language. An introduction to these notions appears in the sections at the end of this chapter, and Appendix I provides a thorough description of the syntax and semantics for the subset of the Java language used in this text.

Finally, remember that the compiler’s role is to enforce spelling and grammar and *not* to comment on the suitability of the statements in the program. For example, the English sentence, “Turn left into the Atlantic Ocean, and drive on the ocean floor until you reach the east coast of France”, is a grammatically correct but procedurally and geographically dubious instruction for someone who wishes to travel from New York City to Paris! In a similar way, a program can contain grammatically correct but dubious statements.

For this reason, *the Java compiler can not guarantee that a program will perform the actions that its author intends*. What the author intends lives in her brain; how she converts these intentions into written instructions cannot be checked by the compiler! For this reason, we must study design, testing, and validation techniques that help a programmer correctly convert her intentions into programs. We encounter these techniques in subsequent chapters.

## 2.6 Summary

We conclude the chapter with a summary of the new constructions, terms, and concepts.

### New Constructions

- *Java class* (from Figure 2):

```
public class Hello
{
    ...
}
```

- *main method* (from Figure 2):

```
public static void main(String[] args)
{
    ...
}
```



- *comment* (from Figure 2):

```
/** Hello prints two lines in the command window */
```

- *internal comment* (from Figure 4):

```
// The next statement creates an object:
```

- *message-sending statement* (from Figure 2):

```
System.out.println("Hello to you!");
```

- *constructing a new object* (from Figure 4):

```
new GregorianCalendar()
```

- *variable name* (from Figure 4):

```
GregorianCalendar c = new GregorianCalendar();
```

## New Terminology

- *application*: a Java program that is started by a human; an object is constructed in primary storage and a message is sent to the object to execute its `main` method.
- *statement*: a single instruction within a Java program; typically terminated by a semicolon.
- *main method*: the “start up” method that executes first when an application is started.
- *argument*: extra information that is appended to a message sent to an object; enclosed within parentheses following the method name within the message.
- *comment*: an explanation inserted into a program for a human (and not the computer) to read.
- *string*: a sequence of characters, enclosed by double quotes, e.g., `"hello"`.
- *Java Development Kit (JDK)*: a collection of programs, available from Sun Microsystems, that let a user compile and start a Java application from the command window.

- *integrated development environment (IDE)*: a single program from which a user can edit, compile, start, and monitor a Java application
- *execution trace*: a sequence of pictures of primary storage, showing the changes to the objects as the computer executes statements one by one.
- *address (of an object)*: an object's "name" within primary storage; used to send messages to it.
- *variable name*: a name of a cell where a value, such as the address of an object, is saved. (See Chapter 3 for a fuller development.)

### Additional Points to Remember

- A Java application must be compiled before it can be started. When a file, `C.java`, is compiled, another file, `C.class` is created, and objects are constructed from `C.class` when the user starts `C`. A message is sent to object `C`'s `main` method.
- When an application is created in primary storage, there are already other preexisting objects, such as `System.out`, to which messages can be sent.
- An object can create other objects by stating the keyword, `new`, followed by the name of the class from which the object is to be constructed.
- When an object is created from a class that is located in a Java package, the package's name should be explicitly *imported* so that the Java interpreter can locate the class. For example, class `GregorianCalendar` is located in the package, `java.util`, so we import the package to use the class:

```
import java.util.*;
public class ExampleOfImportation
{
    ... new GregorianCalendar() ...
}
```

### New Constructions for Later Use

- `System.out`: the object that communicates with the command window. Methods:
  - `print(ARGUMENT)`: displays `ARGUMENT` in the command window at the position of the cursor
  - `println(ARGUMENT)`: displays `ARGUMENT`, appended to a *newline* character, in the command window at the position of the cursor.

- `class GregorianCalendar` (a class from which one can construct objects that read the computer's clock). Found in the package, `java.util`. Method:
  - `getTime()`: returns the current time

## 2.7 Programming Exercises

1. Write a Java program that displays your name and postal address on three or more lines. Next, use a `GregorianCalendar` object to print the date and time before your name.
2. Write a Java program that appears to do nothing at all.
3. Create *two* `GregorianCalendar` objects by inserting these two statements,

```
System.out.println( new GregorianCalendar().getTime() );
System.out.println( new GregorianCalendar().getTime() );
```

into a `main` method. What happens when you execute the program?

4. Modify the program from the previous exercise as follows: Insert between the two statements this Java instruction:

```
try { Thread.sleep(5000); }
catch (InterruptedException e) { }
```

We will not dissect this complex statement—simply stated, it causes the `main` method to delay for 5000 milliseconds (that is, 5 seconds). Compile and execute the program. What happens?

5. Write the smallest Java program (fewest number of characters) that you can. Is there a largest possible (most number of characters) program?
6. Write a Java program that draws this picture on the display:

```
  /\
 /  \
 ----
 |  - |
 |  |  ||
```

## 2.8 Beyond the Basics

### 2.8.1 Syntax

### 2.8.2 Semantics

### 2.8.3 Java Packages

### 2.8.4 Java API

*Here is optional, supplemental material that will bolster your understanding of the concepts in this chapter.*

### 2.8.1 Syntax

When we discuss the appearance of a program's statements—spelling, use of blanks, placement of punctuation—we are discussing the program's *syntax*. The Java compiler strictly enforces correct syntax, and we have no choice but to learn the syntax rules it enforces.

Appendix I gives a precise description of the syntax for the subset of the Java language we use in this text. The description is precise, detailed, and a bit tedious. Nonetheless, it is important that we learn to read such definitions, because they tell us precisely what grammatically correct Java programs look like.

For exercise, we present here the part of the syntax definition that matches the examples seen in this chapter. (Note: because they are developed more fully in the next chapter, we omit the presentation of variable names here.)

#### Class

A Java class has this format:

```
CLASS ::= public class IDENTIFIER { METHOD* }
```

The above is an equation, called a *syntax rule* or a *BNF rule*. Read the equation, `CLASS ::= ...` as stating, “a well-formed `CLASS` has the format ...”.

We see that a well-formed `CLASS` begins with the keywords, `public` and `class`, followed by an entity called an `IDENTIFIER`, which we describe later. (The `IDENTIFIER` is the class's name, e.g., `NameAndDate` in Figure 4. For now, think of an `IDENTIFIER` as a single word.)

Following the class's name is a left bracket, `{`, then zero or more entities called `METHODs`, defined momentarily. (The `*` should be read as “zero or more.”) A class is concluded with the right bracket, `}`.

The classes in Figures 2 and 4 fit this format—both classes hold one method, `main`.

## Method

A method, like the `main` method, can have this format:

```
METHOD ::= public static void METHOD_HEADER
           METHOD_BODY
```

That is, following the words, `public static void`, a `METHOD_HEADER` (the method's name) and `METHOD_BODY` (its body) appear.

```
METHOD_HEADER ::= IDENTIFIER ( FORMALPARAM_LIST? )
```

The `METHOD_HEADER` has a format consisting of its name (e.g., `main`), followed by a left bracket, followed by an optional `FORMALPARAM_LIST`. (The `?` means that the `FORMALPARAM_LIST` can be absent.) Then there is a right bracket.

In this chapter, the `FORMALPARAM_LIST` was always `String[] args`, but we study other forms in a later chapter.

```
METHOD_BODY ::= { STATEMENT* }
```

The body of a method is a sequence of zero or more `STATEMENTS` enclosed by brackets.

## Statement

There are several statement forms in Java; the form we used in Chapter 2 is the message-sending statement, called an `INVOCATION`. The format looks like this:

```
STATEMENT ::= INVOCATION ;
INVOCATION ::= RECEIVER . IDENTIFIER ( ARGUMENT_LIST? )
```

That is, an `INVOCATION` has a format that first lists the `RECEIVER`, which is the name of an object (e.g., `System.out`). A period follows, then comes an `IDENTIFIER`, which is the method name (e.g., `print`). Finally, there are matching brackets around an optional `ARGUMENT_LIST`, defined later.

The syntax rule for well-formed `RECEIVERS` is instructive:

```
RECEIVER ::= IDENTIFIER
           | RECEIVER . IDENTIFIER
           | OBJECT_CONSTRUCTION
```

We read the vertical bar, `|`, as “or.” That is, there are three possible ways of writing a `RECEIVER`: The first is just a single `IDENTIFIER`, e.g., `System` is a `RECEIVER`; the second way is an existing receiver followed by a period and an identifier, e.g., `System.out`. The third way is by writing an `OBJECT_CONSTRUCTION`, e.g., `new GregorianCalendar`; see below.

Notice how the recursive (self-referential) definition of `RECEIVER` gives a terse and elegant way to state precisely that a `RECEIVER` can be a sequence of `IDENTIFIERS` separated by periods.

## Object Construction

```
OBJECT_CONSTRUCTION ::= new IDENTIFIER ( ARGUMENT_LIST? )
```

An `OBJECT_CONSTRUCTION` begins with the keyword, `new`, followed by an `IDENTIFIER` that is the name of a class. Brackets enclose an optional `ARGUMENT_LIST`.

Messages to objects and newly constructed objects can use `ARGUMENT_LISTS`:

```
ARGUMENT_LIST ::= EXPRESSION [[ , EXPRESSION ]]*
EXPRESSION ::= LITERAL | INVOCATION
```

An `ARGUMENT_LIST` is a sequence of one or more `EXPRESSIONS`, separated by commas. (The double brackets, `[[` and `]]`, indicate that the `*` groups *both* the comma and the `EXPRESSION`.

For example, say that `"hello"` and `49` are both well-formed `LITERALS`. Then, both are well-formed `EXPRESSIONS`, and this means `"hello", 49` is a well-formed `ARGUMENT_LIST`.

All the examples in this chapter used `ARGUMENT_LISTS` that consisted of zero (e.g., `System.out.println()`) or just one (e.g., `System.out.println(new GregorianCalendar().getTime())`) `EXPRESSIONS`.

Within Chapter 2, an `EXPRESSION` was either a `LITERAL`, like `"hello"`, or an `INVOCATION` (`new GregorianCalendar().getTime()`).

### Literal and Identifier

These constructions will be developed more carefully in the next chapter; for the moment, we state that a `LITERAL` consists of numbers, like `12` and `49`, and strings, which are letters, numerals, and punctuation enclosed by double quotes. The `IDENTIFIERS` used in Chapter 2 were sequences of upper- and lower-case letters.

## 2.8.2 Semantics

The syntax rules in the previous section tell us nothing about what a Java application *means* (that is, what the application does). When we discuss a program's meaning, we are discussing its *semantics*. From the examples in this chapter, we learned the informal semantics of a number of Java constructions. As an exercise, we repeat this knowledge here. The subsections that follow are organized to match the similarly named subsections of syntax rules in the previous section, and by reading both in parallel, you can gain a systematic understanding of the syntax and semantics of the Java constructions used in this Chapter.

## Class

Given a class, `public class IDENTIFIER { METHOD* }`, a user starts the class as an *application* by typing a class's name, that is, the `IDENTIFIER` part. The file with the name `IDENTIFIER.class` is located, and an object is constructed by copying the file into primary storage.

*Begin Footnote:* As noted earlier in the Chapter, this explanation deliberately avoids technicalities regarding invocation of so-called `static` methods of classes. The issue will be handled later. *End Footnote*

A message is sent to the newly constructed object's main `METHOD`.

## Method

When an object receives a message, it must identify which method is requested by the message. The object extracts from the message the method name, `IDENTIFIER`, and it locates the `METHOD` whose `METHOD_HEADER` mentions the same `IDENTIFIER`:

```
public static void METHOD_HEADER METHOD_BODY
```

(At this time, we cannot discuss the use of the `FORMALPARAM_LIST` and we skip this. In a later chapter, we learn that the `ARGUMENT_LIST` information that is attached to a message “connects” or “binds” to the `FORMALPARAM_LIST`.)

The statements in the `METHOD_BODY` are executed, one by one, in order. It is possible for the last statement in the `METHOD_BODY` to “reply” with an “answer” to the message. We study this behavior in a later chapter.

## Statement

An invocation, `RECEIVER . IDENTIFIER ( ARGUMENT_LIST? )`, sends a message to the object named `RECEIVER`, telling it to execute its method named `IDENTIFIER`. The optional `ARGUMENT_LIST` states additional details that help the method do its job. If the `RECEIVER`'s method, `IDENTIFIER`, is equipped to “reply” with an “answer,” then the answer is inserted at the very position in the program where the message was sent.

## Object Construction

The phrase, `new IDENTIFIER ( ARGUMENT_LIST? )`, constructs an object in primary storage from the file, `IDENTIFIER.class`. The optional `ARGUMENT_LIST` lists information that aids in the construction. The construction step generates an internal “address” that names the newly constructed object, and this address is treated as a “reply” similar to that described in the previous subsection.

### 2.8.3 Java Packages

It is painful to write any computer program completely “from scratch” and it is better to use already-written objects and classes to simplify the task. We did this in the examples in this chapter when we used the `System.out` object and the `GregorianCalendar` class to help us display text in the command window and ask the computer’s clock the time.

Objects and classes like these two are organized into folders called *packages*. `System.out` lives in a package named `java.lang`, and `GregorianCalendar` appears in `java.util`. The former package, `java.lang`, contains many objects and classes that are essential for basic programming. Indeed, `java.lang` contains those Java-components that represent the “computing environment” portrayed in Figure 2 in Chapter 1. For this reason, every Java application automatically gets use of the objects and classes in `java.lang`.

But there are many other Java packages—for graphics, networking, disk-file manipulation, general utilities—and if an application wishes to use a component from one of these packages, then that package must be explicitly *imported* for the use of the application. We saw this in Figure 4, where we stated `import java.util.*` at the beginning of `class NameAndDate` so that `class GregorianCalendar` could be found. (`java.util` is the general utility package; it has classes that can create clocks, calendars, and other structures. In Chapter 4, we will import the `java.awt` and `javax.swing` packages, whose classes know how to draw graphics on the display.)

### 2.8.4 Java API

There are literally hundreds of objects and classes organized into Java’s two dozen packages. How do we learn the names of these components and how to use them? It is a bit early to undertake this ambitious task, but we can at least learn where to look for basic information—we read Java’s *Application Programming Interface (API)* documentation.

The Java API documentation is a summary of the contents of the packages; it is a bit like a dictionary, where each word is listed with its proper spelling and a one-sentence description of its use. The Java API documentation lists the components of its packages and gives short explanations about how to use the components.

Fortunately, the Java API documentation is organized as a collection of Web pages. It is best to download into your computer a complete set of the API’s web pages; see <http://java.sun.com> for details about “Documentation.” But you are welcome to survey the API documentation at Sun’s Web site at the URL just mentioned, as well.

Let’s make a quick search for information about the `System.out` object. We begin

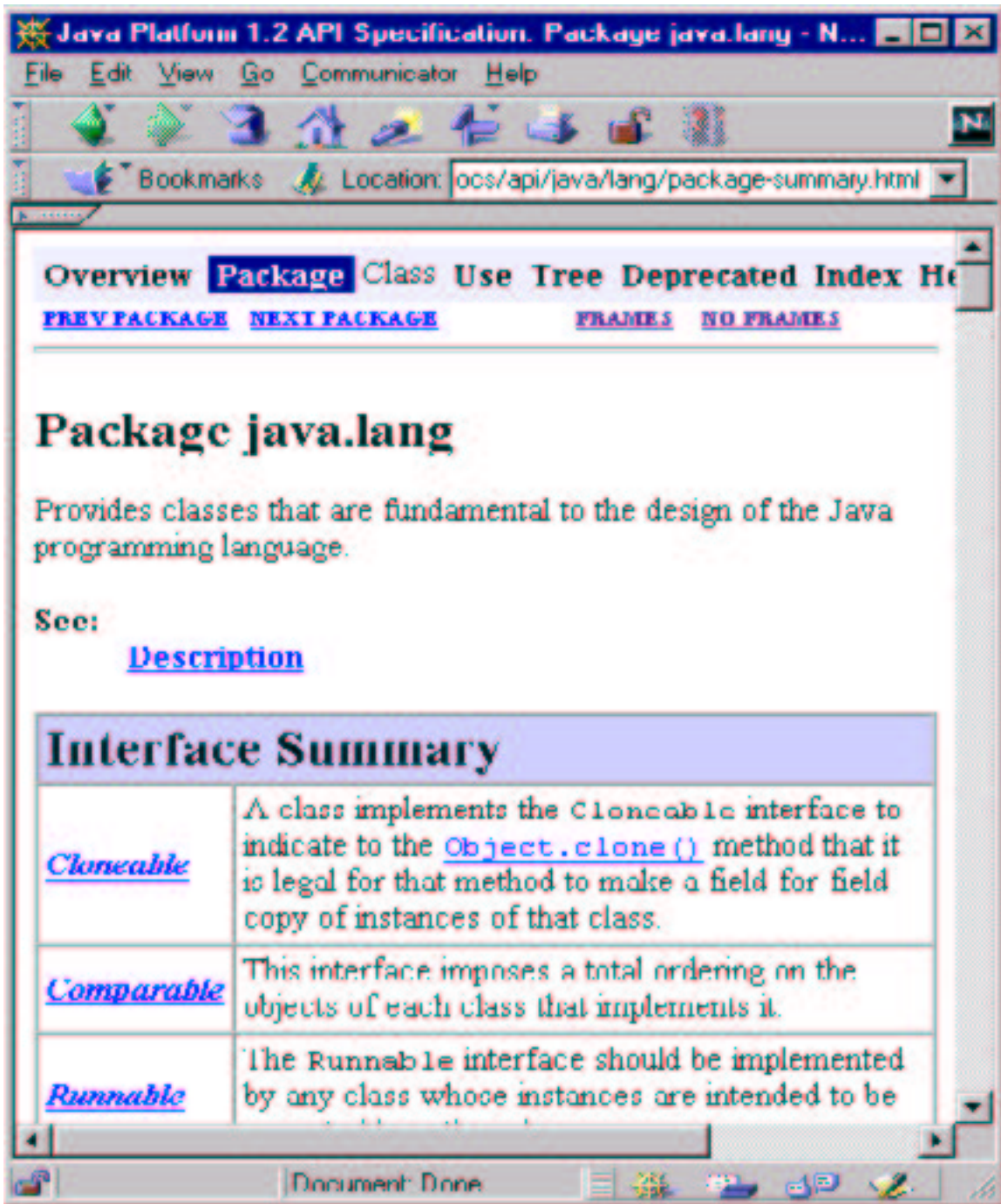


at the API's starting Web page, which looks something like the following:



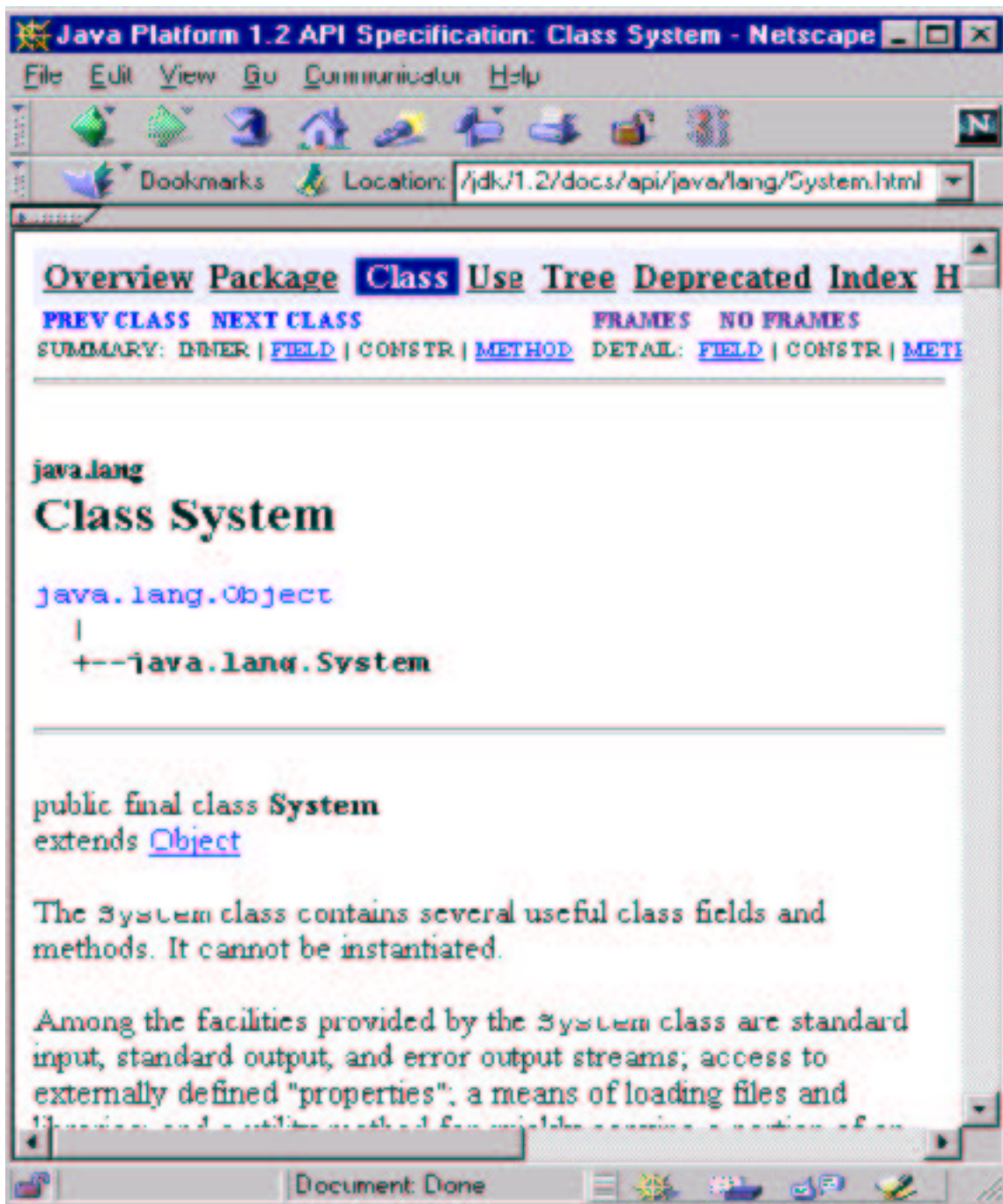
We recall that `System.out` lives in the `java.lang` package, so we search down the list of packages to find and select the link named `java.lang`. (If you do not know the package where a component lives, you can always use the alphabetized index.)

When we reach the Web page for `java.lang`, we see a description of the package:



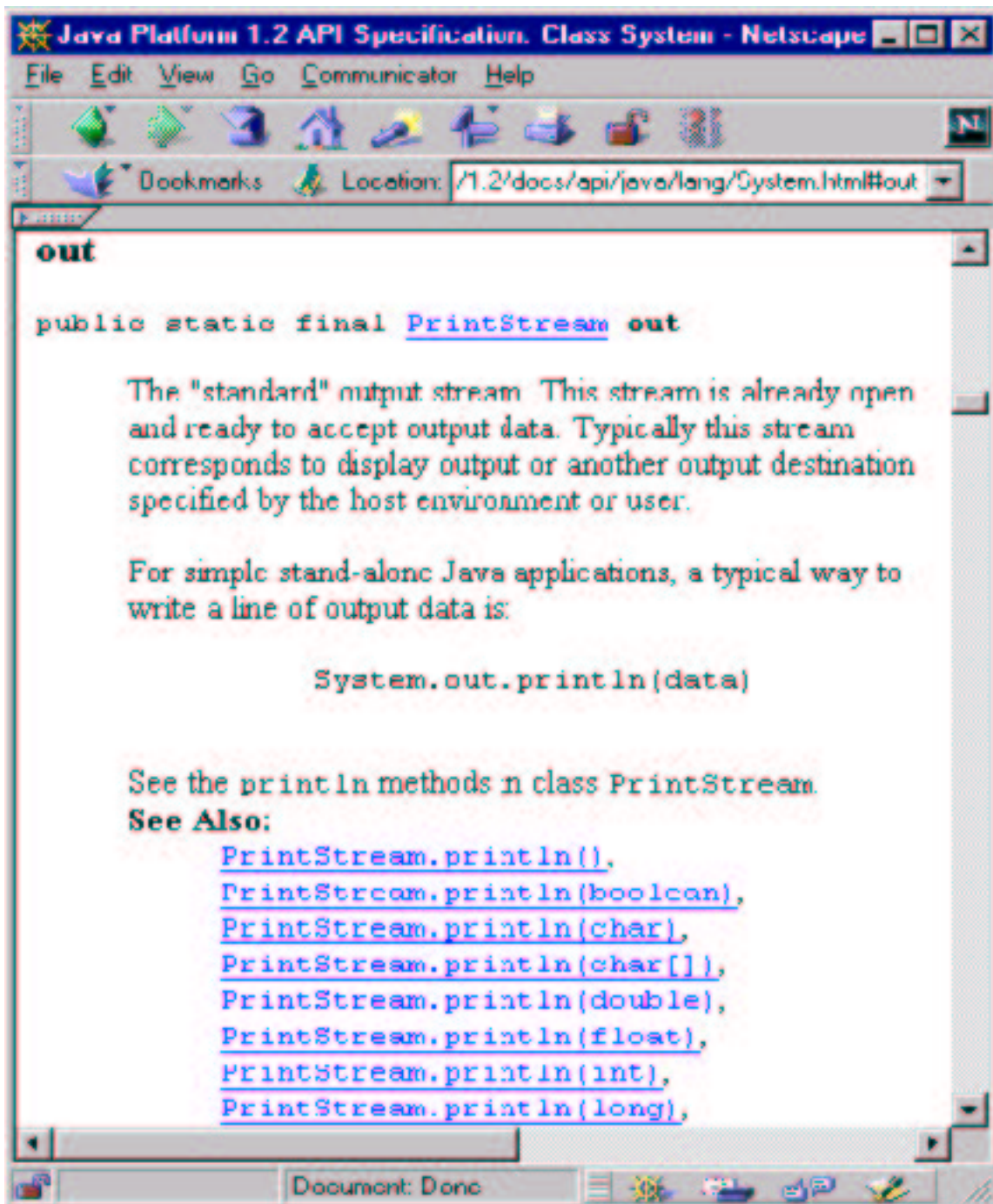
The components of the package are listed, and one them is named `System`. By selecting

its link, we see



Among the System's components, we find and select the one named out, and we end

our investigation by viewing the details for `out` (that is, `System.out`):



In this way, you can browse through the Java packages and learn their contents. (As an exercise, you should consult the Web page for `java.util` and locate class `GregorianCalendar`; you will see that it has many more methods than just the `getTime` method we used so far.)