Chapter 2 Basics of Scanning and Conventional Programming in Java

In this chapter, we will introduce you to an initial set of Java features, the equivalent of which you should have seen in your CS-1 class; the separation of problem, representation, algorithm and program – four concepts you have probably seen in your CS-1 class; style rules with which you are probably familiar, and scanning - a general class of problems we see in both computer science and other fields.

Each chapter is associated with an animating recorded PowerPoint presentation and a YouTube video created from the presentation. It is meant to be a transcript of the associated presentation that contains little graphics and thus can be read even on a small device. You should refer to the associated material if you feel the need for a different instruction medium. Also associated with each chapter is hyperlinked code examples presented here. References to previously presented code modules are links that can be traversed to remind you of the details.

The resources for this chapter are:

PowerPoint Presentation

YouTube Video

Code Examples

Algorithms and Representation

Four concepts we explicitly or implicitly encounter while programming are problems, representations, algorithms and programs. Programs, of course, are instructions executed by the computer. *Problems* are what we try to solve when we write programs. Usually we do not go directly from problems to programs. Two intermediate steps are creating algorithms and identifying representations. *Algorithms* are sequences of steps to solve problems. So are *programs*. Thus, all programs are algorithms but the reverse is not true. Algorithms, unlike programs, do not have to give all details of the solutions. Usually a program is a very uninteresting algorithm much as a shaggy dog story is a boring story. Like a gripping story, an algorithm is expected to skip details that are not relevant. What is relevant, of course, is very much in the eye of the beholder.

Even if we find a program an interesting algorithm, it is good practice to not jump to it immediately. It is better to write a series of algorithms with each successive algorithm adding more details to its predecessor - the program is the final element in this sequence in which all details are filled. This process is called *step-wise-refinement*, as we fill in details incrementally. It helps us systematically approach the solution, and lets others interested in our solution choose the algorithm with the details relevant to them. Algorithms are not always written explicitly – sometimes they stay in our head. For this reason, we will see algorithms for only a subset of the programs given here. It is not unusual for an implicit algorithm to be written down explicitly after the program has been coded as documentation for others.

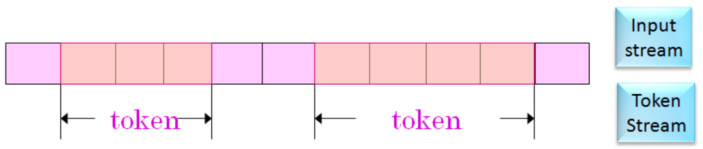
As algorithms are not required to fill in all details, they also do not have to be executable or written in programming languages. We will use a variety of languages to write algorithms. If the problem involves computing some value, then equations might be used in the algorithm descriptions. Some problems, such as making a telephone call, are not mathematical. Therefore, more general than an equation is pseudo code – a cross between natural language and real code. Both equations and pseudo code are textual. Animating graphics is another effective way of describing algorithms, which we will use in this chapter.

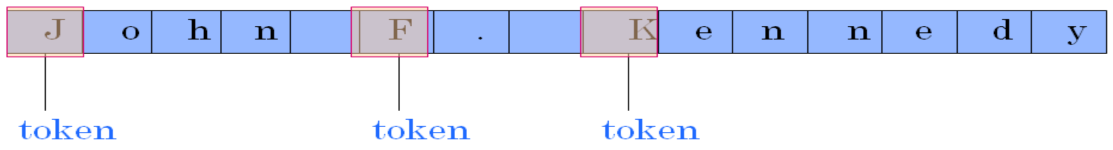
Most algorithms/programs operate on data stored in memory. For this reason, a famous book in computer science defines programs as algorithms + data structures. Usually there are alternative ways to store represent the same information in computer memory. The exact choice is called data representation or simply *representation*. It is a mapping of some abstract, often real-world information, to a computer-defined format called a *data structure*. For example, as we shall see later, a geometric point may be represented using Cartesian or Polar coordinates. Algorithms and programs are often dependent on the data representation choice, though in many cases, it is possible to isolate them from such details. As algorithms get more detailed, so do, often, the associated data structures, as we see in the scanning algorithm below.

Scanning and Scanners

We often talk of scanning physical objects, looking for interesting parts in it. For instance, we might scan the horizon to find hills, radio-wave frequencies looking for NPR stations, images to find the text in them, and faces to see if they are paying attention. Similarly, a computer program often scans computer data structures, looking for parts of it that have properties of interest. For instance, the program may scan a computer simulation of the horizon looking for hills, a computer representation of an image to find text in it, and a computer program looking for variable names and operators. An algorithm that scans data structures is called a *scanner*.

The example we see here is a very simple scanning problem. Given an ordered sequence or *stream* of characters, our task is to find and print the upper case letters in it. For instance, if the input is "John F. Kennedy" our task is to find an print the uppercase characters, 'J', 'F', and 'K'. In general, a scanner looks at an input stream, from one end to the other, to find subsequences, called tokens, that have properties of interest. In this problem, the tokens are character subsequences of length 1. Not each scanned element of the input stream is part of a token, just as not every scanned physical object in the hill-finding problem is a hill. Elements of the input stream that are not part of a token are called *whitespace*. In this example, all non upper case letters are whitespace. The order in which tokens are found defines another stream called the *output stream*. In this example, the output stream consists of the token sequence 'J', 'F', 'K'. Thus, a scanner converts an input stream to an output token stream. Scanners differ in the nature of the input data structures and the properties tokens must satisfy. For example, another scanner could find and print all lowercase letters. The output stream of one scanner can form the input stream to build powerful scanner combinations. For example, the output of the example scanner could be fed to a scanner that looks for vowels.





String Algorithm and Data Structures

In our example, the input stream is a string – a sequence of characters. The input stream, of course, is a data structure given to us. Once we output a token, we do not need to refer to the previous tokens. Thus, we do not need to store the entire stream of tokens; we can use the same data structure to store multiple tokens. As our output tokens are subsequences of length 1, we do not need to create even a data structure to even store a token as we can directly print each upper case letter in the input. So do we need any additional data structure?

Once we have scanned a character, and found it is a token or whitespace, we do not need to scan it again. Thus, we can simply scan the input string from left to right, storing the last position we scanned in a marker variable, and after examining the character at that position, incrementing the marker after each examination. If the examination shows that the character is an upper case letter, we output the character; otherwise we do nothing. This marker variable is then a data structure we need in addition to the input string. It is initialized to 0.

This algorithm can then be described using a series of textual animations of the marker along with the associated output produced. In these animations, the vertical bar, '|', will be used to denote the position of the marker.

The marker is initially at 0. The character at that position is indeed an upper case letter, so we output it:

|John F. Kennedy, marker = 0, output = J

Next, we increment the marker, making it 1. As the character is now not an uppercase character, there is no output.

J|ohn F. Kennedy, marker = 1, output = none

We continue incrementing, without output, until the marker is 5, when we output J.

John |F. Kennedy, marker = 5, output = F

Again the marker is incremented without output, until it reaches 8, at which point we output K.

John F. |Kennedy, marker = 8, output = K

Again we increment the marker.

John F. K|ennedy, marker = 9, output =

A visual scan of the string shows that there are no more upper case characters. The computer must similarly scan the string to make this determination. Thus, it keeps incrementing the marker, finding no upper case letters, until it reaches the end, at which point the process stops.

John F. Kenned|y, marker = 14, output = none

Scanning Java Program

Below, we see the data structures and algorithm converted to a Java program.

```
package lectures.scanning;
public class AnUpperCasePrinter {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Illegal number of arguments:" + args.length
+ ". Terminating program.");
      System.exit(-1);
    }
    String scannedString = args[0];
    System.out.println("Upper Case Letters:");
    int index = 0;
    while (index < scannedString.length()) {
      char nextLetter = scannedString.charAt(index);
      if (Character.isUpperCase(nextLetter))
        System.out.print(nextLetter);
      index++;
    }
    System.out.println();
  }
}
```

Figure 2-1 Java Program

As this is our first Java program and we have already seen the scanning algorithm, this code serves more as an introduction to Java – as a second language - than scanning.

Conventional vs. Object-Oriented Programming

As you probably know, Java is called an object-oriented programming language to distinguish it from a conventional programming language. These two types of languages support very different styles of programming. It is not possible to do object-oriented programming using a conventional programming language but it is possible to do conventional programming using an object-oriented language. The most simple-minded explanation of the difference between the two styles is that conventional programming creates the entire program as one class, while object-oriented programming looks for ways to decompose the program into multiple classes. The major goal of this course is to teach you object-oriented programming. However, to get you going quickly with a Java programming, in this chapter we will do conventional programming. What we see in Figure ??? is a Java class that defines the complete program.

Developing and Executing the Program Using a Bare-bone Environment

Before we understand the logic inside the single-class program above, let us see the external process of creating it and running it. Today, development and running of a program will be done through a sophisticated visual programming environment. There are a variety of such programming environments, each with a different user-interface. All of them provide a Java compiler and a Java interpreter, called, javac and java, respectively. Moreover, the operating system provides a text editor and a command interpreter. The combination of these three tools constitutes a bare-bone programming environment, which is sufficient, but not always suitable, to develop Java programs. Here we will illustrate how the program above can be developed using such an environment. It will show some of the steps all programming environments take.

The process of developing a Java program is more complicated than the process you might have seen for programs in other languages. This is because of Java's heavy emphasis on object-oriented programming which makes conventional programming more complicated than necessary. We must first choose a *project folder*. You have complete freedom in choosing its name. In this example, let us assume that the project folder is "course" The items in this folder and its subfolders are constrained by the text of the program above.

Consider the package header:

package lectures.scanning

and the class header:

public class AnUpperCasePrinter

Together, these declarations define a class called AnUpperCasePrinter, in package lectures.scanning. The class as two names, a short name, AnUpperCasePrinter, which can be used to refer to the class in its package and a full name, lectures.scanning.AnUpperCasePrinter, which is essentially the name of the package followed by the sort name.

The package declaration requires that we create a folder called lectures in the project folder (course); and in this folder we create a subfolder called scanning. In other words, we must create a package subfolder called lectures/scanning or lectures\scanning in the package folder, depending on the file name separator in the operating system we use. In the rest of the course, we will assume that '/' is the file name separator.

In general, if a class contains the package header:

package $a^1.a^2.a^3..a^n$

we must create a package subfolder $a^1/a^2/a^3../a^n$.

The class declaration requires us to create a file named AnUpperCasePrinter.java in the package folder. In general, each program class should be saved in a separate file (in the package folder) whose name is the short name of the class followed by the suffix, .java.

This is a text file - the suffix .java indicates it is a special kind of text file that contains Java code. As it is a text file we can use a text editor (such as Notepad or Notepad++) to enter the text in Figure 1.

This text represents the source code of the program. As you probably know from CS-1, source code is not directly executable and must be translated into object code, and the program that does the conversion is called a compiler. To compile our example file, we must ask the command interpreter to change its directory to the package folder of the class. Thus we are in the project folder, we must execute the command

```
cd lectures/scanning
```

and execute the command:

```
javac AnUpperCasePrinter.java
```

to compile it. The compiler creates a file called AHelloWorldGreeter.class in the same folder in which the source code exists, which contains the object code for this program.

Our next steps are to execute or interpret the program. To do so, we must cange the directory to the project directory, course by executing, for instance, the command:

```
cd ../..
```

We can now ask the Java interpreter to execute our compiled class, giving it as an argument or paramater the string to be scanned:

```
java lectures.scanning.AnUpperCasePrinter "John F. Kennedy"
```

The program creates the following output:

```
Upper Case Letters:
JFK
```

In general, to execute a program, we must type:

```
java <Full Name of Main Class> <A list of arguments>
```

in the directory containing the top-level package, that is, in the project folder.

Case matters in the names used in Java. In some operating systems/programming environments, it also matters in the names supplied to the tools. So if we typed:

```
java lectures.scanning.anuppercaseprinter "John F. Kennedy"
```

we may get an error message saying that the class ahelloworldAHelloWorldGreeter was not found.

The following transcript of our commands, executed by the Bash command interpreter, illustrates a successful compilation and execution of the program. Here the lines starting with $ are the commands we entered, the italicized ones staring with Dewan@DEWAN-T431S are prompts from the command interpreter telling us the current directory or folder, and the remaining lines are program input and output. In this example, there is no program input – the scanned string is supplied as an argument. Later we will see how to get this string as an input.

```
Dewan@DEWAN-T431S /d/dewan_backup/java/course
$ cd lectures/scanning
Dewan@DEWAN-T431S /d/dewan_backup/java/course/lectures/scanning
$ ls
AnUpperCasePrinter.java
Dewan@DEWAN-T431S /d/dewan_backup/java/course/lectures/scanning
$ javac AnUpperCasePrinter.java
Dewan@DEWAN-T431S /d/dewan_backup/java/course/lectures/scanning
$ ls
AnUpperCasePrinter.class  AnUpperCasePrinter.java
Dewan@DEWAN-T431S /d/dewan_backup/java/course/lectures/scanning
$ cd ../..
Dewan@DEWAN-T431S /d/dewan_backup/java/course
$ java lectures.scanning.AnUpperCasePrinter "John F. Kennedy"
Upper Case Letters:
JFK
Dewan@DEWAN-T431S /d/dewan_backup/java/course
```

An equivalent set of commands can be executed in the Windows command interpreter, with the main difference being that the "/" file name separator is replaced by "\" and the "ls" command is replaced by "dir". For more on the nature of command interpreters, see: ????

A window that shows the Input and Output (I/O) of a program is called the *console* or *transcript window*. Here the command interpreter window serves as the console window. An interactive programming environment provides a special console window for program I/O, as program compilation and execution occurs trough a GUI rather than a command window that could double as a console window

Program from the inside

We saw above our example program from the outside – as a person compiling and executing the program. Let us now look at it from the inside by studying its contents. As mentioned before, the code in Figure 1 defines a single class, in a single package, and the package header:

package lectures.scanning

names the package of the class and the class header:

public class AnUpperCasePrinter

defines the short name of the class. The rest of the code defines the code *body*. The class and package header contains information about the class that is of interest to the users of the class. It consists of two *keywords* or *reserved words.* The first is the keyword, **class**, which is defined by Java, while the second is the class name, which is defined by the programmer. As you probably know from your CS-1 course, a keyword is reserved by the language in that it cannot be overridden by the programmer. We will use **boldface** to identify keywords**.** Not all predefined Java words are keywords. For instance, *String* and *println* are not keywords, and thus can be overridden by a program. Though, in this course, we will not be overriding predefined Java words, it is important to distinguish keywords from overriddable (predefined and programmer-defined) words.

Method Declaration

The class body contains the implementation of the class, which consists of the definition of the *methods* –*procedures* and *functions* - of the class. These are named parameterized sequences of instructions, sometimes also called *subroutines*. This material assumes you are familiar with defining and calling or invoking methods. Here we focus mainly on giving the Java syntax for representing them.

In our example, the class body consists of the definition or declaration of a single *method,* `main`, enclosed within the outermost curly braces:

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

   ….

}
```

Though this program contains a single method declaration, in general, a Java program can contain many method declarations.

Like a class declaration, a method declaration has two parts: a *header* and a *body.* The method header is the first line of the declaration before the first curly brace:

public static void main (String args[])

The method body is the rest of the declaration:

```
{

  if (args.length != 1) {
  ….

}
```

Method Header

The method header contains information that is of interest to its users. The user of a method needs to know how to invoke the method. Therefore, the header includes the name of the method and the nature of the arguments or parameters accepted by it. In this example, it consists of three Java *keywords,* **public**, **static**, and **void**, followed by the method name, *main*, followed by an arguments specification.

The arguments specification tells us that the method takes a single argument, which an array of *strings* named, *args.* This material assumes you are familiar with the concept of an array - is a list of elements of a certain type whose size is fixed. A string is a special kind of array whose elements are characters – letters, numbers and other symbols we can enter on a keyboard.

The three keywords indicate various properties of the method. The keyword **public** says that the method is visible outside the class to other software modules –in particular the Java interpreter. The keyword **static** says that it is a "class method" – a method that does not require the dynamic "instantiation" of a class - rather than an "instance method," which does require dynamic instantiation. For now, since we are not doing object-based programming, all methods will be class methods. When we instantiate classes dynamically, we will omit this keyword to define instance methods. The keyword **void** says that it is a procedure rather than a function, that is, it does not return a value.

This method header indicates this is the *main* method of the class. A main method is a special method of a class in that it is automatically called by the Java interpreter when we ask the interpreter to execute the class. You should type the header of a main method exactly as shown above to indicate that it is the main method. Check that you have done so in case you get an error when you execute the class saying that the main method was not found.

Variables and Expressions

The integer variable index is the marker in our algorithm. It is declared to be of type *int* (for integer) and initialized to 0. In general, a variable named, <Variable>, of type <Type >, initialized to expression, <Expression> is declared as follows:

```
<Type> <Variable> = <Expression>;
```

An expression is any program fragment that yields a value. The program elements within angle brackets are *place holders*, which are replaced with matching values in program code:

```
int index = 0;
```

Here the variable is initialized to the integer *literal* 0, so called because we specify its value literally. In general a variable can be assigned or initialized an arbitrary expression.

A declared variable can be reassigned a new value:

```
Index = index + 1;
```

We do not have to initialize a variable when it is declared:

```
int index;
```

A code fragment that (re) assigns a variable is a called an *assignment statement*. A statement is a code unit that takes some action. Later we will see other kinds of statements such as if statements and while statements.

Final, Named Constants, Literals, Magic Numbers

If we do not want a variable to be reassigned, we can use the prefix final in its declaration. If such a variable is assigned a literal, it becomes a named constant **–** a name associated with a constant or unchanging value:

final int MAX_UPPER_CASE_LETTERS = 3;

As it has probably drilled into you in CS-1:

> *Number in expressions can be unintelligible to readers trying to understand their meaning. Use named constants instead.*

Thus:

int upperCaseCharacters[MAX_UPPER_CASE_LETTERS];

Is preferable to:

int upperCaseCharacters[3];

as 3 can be a magic number for the reader. In general, any literal, including a String or character, can be magic unless it is assigned to a named constant– therefore people do not talk of magic characters or strings, though some of them could indeed use a description through a named constant.

Perhaps an even more compelling reason for a named constant, seen in the example above, is that if we decide to increase the maximum number of upper case letters, we need to only change the definition of the named constant rather than all the places where the upper limit is used. This motivation applies to all values, not just numbers. This leads to the following related rule:

> Replace multiple instances of the same (possibly not magic) literal with a named constant.

It is possible for a final variable to not be a named constant as it can be being assigned the value of a variable at declaration time or later:

final int savedIndex;

savedIndex = index;

=, ==, !=

As we see above, Java uses = as the assignment operator. Thus, the above code says that the integer variable index is assigned the value 0. = is not to be confused with the relational == operator which returns true if its left operand has the same value as its right operand. Thus:

index == 0 →true

index == 1 → false

!= is the inverse of ==, returning the opposite of ==:

Index != 0 → false

Index != 1 → true

++ and --

Often an assignment statement increments or decrements the value of a variable. Java provides the following shorthand:

<Variable> ++;

which is the same as:

<variable> = <variable > + 1;

Similarly, the shorthand:

<Variable> --;

Is the same as:

<Variable > = <variable > - 1;

Thus, in our example program:

index--;

is te same as:

index = index – 1;

char

While you should all be familiar with integer values, it is possible many of you have not seen characters. The computer must often process characters. Most programs accept character input and produce character output. In fact, some programs, such as a spelling checker, perform most of their computations in terms of characters. Characters are the building blocks for the strings used in our example program.

This type defines a variety of characters including the English letters (both lowercase, a..z, and uppercase, A..Z), the decimal digits 0..9; "whitespace" characters such as blank and tab; separators such as comma, semicolon, and newline; and other characters on our keyboard. A character can be represented in a program by enclosing its print representation in single quotes:

'A' 'Z' '1' '0' ' '

Two consecutive single-quotes denote a special character called the *null* character:

''

The null character is used to mark the end of a string. It is not useful to print it since Java prints nothing for it.

How do we represent the single-quote character itself? We could enclose it in single-quotes:

‘’’

However, Java would match the first two single-quotes as the null character, and think you have an extra single-quote character. So, instead, it defines the following representation for a single-quote:

‘\’’

Here, instead of enclosing one character within quotes, we have enclosed a two-character escape *sequence*. The first of these characters, \, or *backslash*, is an *escape* character here, telling Java to escape from its normal rules and process the next character in a special way.

Java defines escape sequences to represent either those characters that cannot use the normal representation or those for which the normal representation may not be readable. A character value or *literal* cannot have a new line character in it, so \n denotes the new line character. A backslash after the first quote denotes special processing, not the backlash character itself, so \\ denotes the backlash character. Typing a backspace after a single-quote would erase the single-quote, so \b denotes the backspace character. We can represent the tab character by entering a tab between quotes:

‘   ‘

but this representation can be mistaken as the space character. So \t denotes a tab character. Similarly, we can represent the double quote character as:

‘”’

but it may be mistaken for two null characters. So \” denotes the double quote character. The following table summarizes our discussion:

Escape Sequence	Character Denoted
\’	‘
\n	new line
\b	back space
\\	\
\t	tab
\”	“

Table 1 Some Useful Java Escape Sequences

As programmers, we do not have to concern ourselves with the exact integer assigned to each character. However, we sometimes need to know something about the relative order in certain subsets of the character set such as the set of lower case letters and the set of digits.

The following code uses the order among uppercase letters to determine if a character is upper case:

```
   if (nextLetter >= 'A' && nextLetter <= 'Z')

     System.out.print(nextLetter);
```

This check is automatically done by the predefined Boolean Character.isUpperCase() function. We see here that letter ordering allows this function to be implemented efficiently- otherwise the test character would have to be compared against each upper case letter.

&& and ||

In the Boolean expression above, && is the **short-circuit and** operator **–** it evaluates its second argument only if the first argument is true. Similarly, || is the short circuit or operator – it evaluates its second argument only if the first argument is false.

String Literals, Indexing, Length

A string can be considered essentially an array of characters. A sequence of characters within double quotes denotes a Java String literal, which can be assigned to a variable declared to be of type String.

String s = “hello world”

This problem requires a way to decompose a string into its constituent characters. We identify string components through their positions or *indices* in the string. For Strings, the charAt() method is provided, which takes the component index as an argument and returns a character. In Java, String (and array) indices start from 0 rather than 1

Thus, given string s above:

s.charAt(0) == ‘h’;

s.charAt(1) ==   ‘e’

In general, we access the ith character of string, s, as:

s.charAt(i-1);

Not all string indices are legal. An index that is smaller (greater) than the index of its first (last) character is illegal. Thus, both of the following accesses will result in a StringIndexBounds error or exception :

s.charAt(11)

s.charAt(-1)

Given a String, s, s. length(), returns the number of characters in a string. Thus:

"helloworld".length()   →   11

"".length()   →   0

We can use this function to define the range of legal indices of an arbitrary string s:

0 .. (s.length() - 1)

Sub-Strings

Besides individual characters, we may also wish to retrieve *sub-strings* of a string, that is, sequences of consecutive characters that appear in the string. Given a string, s, we can invoke:

    s .substring(beginIndex, endIndex)

to return a new string that consists of the character sequence starting at beginIndex and ending at endIndex – 1, that is:

s.charAt(beginIndex) .. s.charAt(endIndex - 1)

Invoking this operation results in the StringIndexOutOfBounds error or exception if beginIndex is greater than endIndex. If they are both equal, it returns the empty string. Thus:

"hello world".substring(4, 7) → "o w"

"hello world".substring(4, 4) → ""

"hello world".substring(7, 4) throws StringIndexOutOfBounds

Java strings are readonly or *immutable*, that is, they cannot change. A separate type, StringBuffer, which we will not study in this course, defines mutable strings. Java does provide the + operation on strings to create *new* strings from existing strings. Thus:

“hello” + “world” == “hello world”

Here, we do not change either string, but instead, create a new string that stores the result of appending the second string to the first one.

If we add to a string a non String value such as an integer, the value is converted to its String representation. Thus,

"null string length is " + “”.length == null string length is 0

println and print

We can not only create new strings but also, of course, print them using the methods System.out.println() or System.out.print(). Both the String representation of an arbitrary value on the console window. The difference between them is that the former prints a new line character after printing the String, thereby ensuring that the next output appears on the next line. Thus, the statement:

System.out.println(*nextLetter)*

prints each letter on the same line and

System.out.println()

Prints the null string (nothing) on the current line and then output a new line character. The two methods can

Overloading

Consider the following two uses of the operator, +, in Java:

5 + 3

“hello” + “world”

The operation has different meanings in the two examples above. In the first example, it means integer addition, while the second case it means string concatenation. An operation such as + that has more than one meaning is called an overloaded operation. It is analogous to an English word such as “fly” that has more than one meaning. Just as we use context to determine the meaning of an overloaded English word, Java also uses context to determine the actual action performed when we use an overloaded symbol.

Like symbols, method names can also be overloaded in Java. Consider the following two uses of the method println:

```
System.out.println("Hello World");
```

```
System.out.println(8);
```

The second println is different from the one we used to print strings, since in the first case, it simply prints its string argument while in the second case it converts its double argument to a string before printing it. As in the case of +, Java uses the context of the actual parameter to determine which version of println to call.

Array Indexing, Length, System.exit()

The arguments specified by a user can be accessed by the body of the main method as args[0], args[1], args[2] and so on, where args is the name given to the argument array in the header of the main method. In general, the Nth argument is accessed in the main method as args[N-1].

It is an error to access an argument that has not been entered by the user. To prevent such an error, our program has the following code fragment:

```
  if (args.length != 1) {
    System.out.println("Illegal number of arguments:" + args.length + ". Terminating program.");
    System.exit(-1);
   }
```

This program fragment illustrates several concepts in Java. It is possible to determine the length or size of some array, a, using the syntax, a.length. Here we are essentially accessing the value of a final variable variable a.length. In the case of a String s, we find the size by calling the method s.length(), hence the need for parentheses.

Usually a program terminates when it finishes execution of the main method. However, we can terminate it at any point, by executing the predefined method, System.exit(). The argument to this method is an error code. By convention the code 0 indicates no error and a negative value indicates some kind of error associated with the value.

Entering and Parsing Main Arguments

The programming environment parses or processes the arguments we supply to determine the actual parameters passed to the main method of the class we execute. For example, when we execute the command:

```
java lectures.scanning.AnUpperCasePrinter "John F. Kennedy"
```

the programming environment converts characters between the quotes to an argument array consisting of a single String, John F. Kennedy". It is not necessary to enclose each element of the argument array within quotes. Thus the following is legal:

```
$ java lectures.scanning.AnUpperCasePrinter John F. Kennedy
```

However, in this case, the output is:

```
Illegal number of arguments:3. Terminating program.
```

By default the programming environment uses whitespace (space and tab characters) to delineate the elements of the argument array. Thus, in the example above, it assumes that the argument array consists of three elements:

John
F.
Kennedy

As we saw above, our main method gives the error message when the actual number of arguments is not the same as the expected number, and terminates. By putting quotes around the scanned string, we ask the programming environment to ignore the whitespace in it and consider it a single argument element.

If-Else and If Statement

It is possible to execute one statement, <Then Statement>, if a Boolean condition, <Boolean Expression>, is true, and another statement, <Else Statement>, if it is false, using the if-else statement:

```
If (<Boolean Expression>)

        <Then Statement>

else

        <Else Statement>
```

Note that parentheses are required around the Boolean expression. <Then Statement > is called the *then-part* or *then-branch* of the if statement and <Else Statement> the *else-part* or *else-branch*.

Sometimes we wish to take no action in an else branch. In this case, the *if statement* can be used instead of the if-else statement. It has the syntax:

```
        if (<Boolean Expression>)

                <Then Statement>;
```

This is the form of the if statement we see in the code below.

```
   if (Character.isUpperCase(nextLetter))
    System.out.print(nextLetter);
```

Compound Statement

A series of statements, <S1>, <S2> , …., <SN> can be converted into a single *compound statement* by enclosing them in curly braces and separating them with semicolons:

```
{
        <S1>;
        <S2>;
        …
        <SN>
}
```

Thus

```
{
   char nextLetter = scannedString.charAt(index);
   if (Character.isUpperCase(nextLetter))
    System.out.print(nextLetter);
   index++;
  }
```

Is a compound statement consisting of three statements.

Boolean Constants and Simplifying Boolean and Other Expressions

If statements, of course, require a good understanding and use of Boolean expressions, which in Java compute to the predefined values, **true** and **false**, rather than 1 and 0, as in some other languages such as C. True and false are also words in the English language. Sometimes there is a tendency to overcomplicate Boolean expressions because by translating them to English, as in the following example:

```
   if (Character.isUpperCase(nextLetter) == true) {
        System.out.print(nextLetter);
   }
```

Character.isUppercase() is already a Boolean expression and comparing it with **true** does not change the result just as adding 0 to an integer values does not change the result. This brings us to the following rule:

Simplify expressions, especially Boolean expressions, by removing redundant clauses in them.

For the same reason:

In a function, we should simply return a Boolean value rather write an if statement that returns the value.

Thus, the following if statement :

```
        If (Character.isUpperCase() ) {
                return true;
        } else {
                return false;
        }
```
should be replaced by:

```
        return Character.isUpperCase();
```

While Statement

It is possible to repeatedly execute a statement, <While Body>, until some Boolean condition, <Boolean Expression> is met by executing the while statement with the syntax:

while (<Boolean Expression>)

  <while Body>

The repeated statement is called the while body and the Boolean expression is called the whole condition

Console Input

Instead of scanning a string provided as a program argument, a scanner could also process a string input while a program is executing. The following class is like the one we saw above, except that it processes a String input by the user:

```
package lectures.scanning;
import java.util.Scanner;
public class AConsoleReadingUpperCasePrinter {
	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);
		String scannedString = scanner.nextLine();
		System.out.println("Upper Case Letters:");
		int index = 0;
		while (index < scannedString.length()) {
			char nextLetter = scannedString.charAt(index);
			if (nextLetter >= 'A' && nextLetter <= 'Z')
				System.out.print(nextLetter);
			index++;
		}
		System.out.println();
	}

}
```

The following command-window transcript shows how a user interacts with this program:

```
$ java lectures.scanning.AConsoleReadingUpperCasePrinter
John F. Kennedy
Upper Case Letters:
JFK

Dewan@DEWAN-T431S /d/dewan_backup/java/course
```

Now the scanned string is entered in the console window after the program is run rather than supplied as an argument before the program is run. It does not have to be enclosed in quotes as our program is the only code involved in processing this input – the programming environment does not intercept it.

There are many ways to read input in a Java program, all of which are more complicated than writing output. Perhaps the simplest of them involves creating and initializing a variable of type Scanner:

```
Scanner scanner = new Scanner(System.in);
```

For now, we do not quite understand the expression assigned to the variable but we can see that it involves System.in, which denotes the stream from which we read values, just as System.out denotes the stream to which we write values.

Now we can use the variable as follows to repeatedly read input lines:

```
String scannedString = scanner.nextLine();
```

Thus, we first use the predefined Scanner type to convert the input entered by the user into a line of characters and then we use our own scanner to convert it into a series of upper case characters.

Manual and Automatic Imports

The Scanner type is in fact a class, whose full name is java.util.Scanner, which means it is created in the package java.util. We were able to use its short name because of the import declaration:

import java.util.Scanner;

An import declaration must be placed between the package and class headers of a class and tells Java that the class will use the short name of the imported class rather than the full or long name. If we did not have this import declaration, we could have initialized our scanner variable using the full name:

```
java.util.Scanner scanner = new java.util.Scanner(System.in);
```

It is illegal to refer to a class by its short name if we have not imported its full name. It is not legal to import two classes with the same short name as otherwise Java does not know to which class that name refers.

Often there is only one class with a given short name. Some programming environments such as Eclipse provide commands that automatically import such classes if we have used their short names but not imported them. For example, if we use the short name, Scanner, in our code, and type CTRL-SHIFT_O in Eclipse, the import statement will be automatically generated. We will often omit the full names of classes with unique short names, as your programming environment can find their full names.

Import All and Why to Not Use It

Another way to reduce import overhead applies to situations in which we want to import multiple classes from the same package. We can use "*" instead of a class name in an import declarations:

import java.util.*;

Those who have seen regular expressions would know that '*" (invented by Kleene of UW-Madison) matches arbitrary character sequences. In the import declaration context, it matches all classes in the specified package. Thus, the above import declaration matches all classes in the package java.util, and thus would import all of them.

The *import all* or *star import* feature has been presented here only to forbid it, for two reasons. First, persons using star import all import all declaration may accidentally import and a use a class whose short name is the same as the name of the class they want to use. Second, readers of the code cannot use the import declarations in a class to understand which classes from other packages are being used. Given that programming environments today provide features to automatically import classes, such an import makes even less sense. Therefore:

> *Do not use import all (star import).*

Unique Short Names and No Long Names

Even with the import of a single package it is possible to use a class whose name conflicts with that of a class in the package of the importing class. Moreover, we may repackage classes just as we move files into different folders. This problem gives rise to the following rule:

> *Give a class a short name that is different from the short names of other classes in your project and the packages your project uses or is later likely to use.*

In Eclipse, it is easy to "refactor" code by renaming classes, variables and other identifiers (names), and repackaging classes – so it is easy to correct violations of this rule.

In the class material, this rule is constantly violated to show incremental development of a class. The various increments are given the same short name but put in different packages. You will face a similar problem as you incrementally develop your project. However, you do not have to show all increments at the same time and should be creating a new project for each assignment. Thus, you should not face this problem..

As mentioned before, we can use the long name of a class instead of its short name:`java.util.Scanner scanner = new java.util.Scanner(System.in);`

This is useful when we have two classes with the same name – the long name disambiguates. When this is not the case, using a long name is simply verbose. Moreover, it does not require an import at the start of the class declaration, which, as mentioned above, is useful documentation for someone trying to get an overview of a class declaration without looking at the details of the class body. As we are disallowing classes with the same short name, we will also require:

> *Do not use the full name of a class except in an import declaration,*

Multi-Method Classes and Calling a Method in a Different Class

The only difference between the two classes we have defined above, `AnUpperCasePrinter` and `AConsoleReadingUpperCasePrinte` is in the code that initializes the variable scannedString, that is, the code between the main method header and the first println(). In fact, the second class was created by copying and editing the code in the first class – always a bad sign, as it means that we are duplicating code. Duplicated code has the disadvantage that keeping its copies consistent is tedious and error prone. For instance, if we decide that we want to print each upper case letter on a different line, we must change both of the classes above.

Much of this material addresses techniques for reducing code duplication. The simplest of these, which you should have seen in your CS-1 class, is to divide code into multiple methods. This technique serves us well in our examples, as the code to initialize and process the scanned string can be put in different methods. Java allows methods of a class to be called from other classes. Our second class can call the scanning code without executing the initializing code.

We can use this technique to rewrite the class AConsoleReadingUpperCasePrinter:.

```
package lectures.scanning;
import java.util.Scanner;
public class AModularConsoleReadingUpperCasePrinter {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    String scannedString = scanner.nextLine();
    scanAndPrint(scannedString);
  }
  public static void scanAndPrint(String scannedString) {
    System.out.println("Upper Case Letters:");
    int index = 0;
    while (index < scannedString.length()) {
      char nextLetter = scannedString.charAt(index);
      if (nextLetter >= 'A' && nextLetter <= 'Z')
        System.out.print(nextLetter);
      index++;
    }
    System.out.println();
  }
}
```

The code to process the scanned string is now in a separate method, scanAndPrint(), which takes this String as an argument. The main method sets this String to be the next input line and calls scanAndPrint().

We can now rewrite the class, AnUppercCasePrinter, to use rather than duplicate the scanAndPrint in our new input-reading class, AModularConsoleReadingUpperCasePrinter.

```
package lectures.scanning;
public class AModularUpperCasePrinter {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Illegal number of arguments:" + args.length
      + ". Terminating program.");
      System.exit(-1);
    }
    AModularConsoleReadingUpperCasePrinter.scanAndPrint(args[0]);
  }
}
```

We see here that the general syntax for calling a static or class method is:

<Class>.<Method> (<Arguments>)

When the method is defined in the calling class, the class name can be omitted from the call. This is the reason that the call to scanAndPrint() in AModularConsoleReadingUpperCasePrinter did not need the class prefix.

```
package lectures.scanning;
import java.util.Scanner;
public class Console {
	static Scanner scanner = new Scanner(System.in);
	public static int readInt() {
		return Integer.parseInt(scanner.nextLine());
	}
	public static String readString() {
		return scanner.nextLine();
	}
}
```

Console Library Class, Global Variables, Boolean and double

The ability to call methods in a different class allows us to create our own libraries – classes without main methods whose only function is to provide methods for other classes. To illustrate, let us create a library class that provides a slightly more convenient way to read input:

```
package lectures.scanning;
import java.util.Scanner;
public class Console {
	static Scanner scanner = new Scanner(System.in);
	public static int readInt() {
		return scanner.nextInt();
	}
	public static boolean readBoolean() {
		return scanner.nextBoolean();
	}
	public static double readDouble() {
		return scanner.nextDouble();
	}
	public static String readString() {
		return scanner.nextLine();
	}
}
```

This class creates the scanner variable as a **global variable** of the class – that is a variable declared outside all methods in the class that does not belong to a specific method. A global variable declared as static, it is accessible to all methods in the class. This variable is initialized to a value of type Scanner as before. In contrast, a variable declared inside a method is a **local variable** and accessible only within that method. Thus, AModularConsoleReadingUpperCasePrinter , the variable index is accessible only in scanAndPrint() and not main(). Local variables can also be declared inside compound and for statements are only accessible within these statements.

The user of Console now simply calls the methods in this class, and is oblivious to the fact that a value of type Scanner, involving System.in, has to be created. Thus, to read an input line from some other class, all we have to do is import Console, and execute:

```
Console.readString();
```

The types boolean and double refer to Boolean and decimal values respectively.

Java Program Structure

The examples presented so far, illustrate the general structure of a Java program. In general, a program consists of a main class and can have several other classes. The main class must have the main method and can have several other methods

For Loops and Array Declarations,

The following method introduces the Java syntax for two additional concepts with which you are probably familiar:

```
public static void readAndPrintStrings() {
            // read the number of strings to be input
            System.out.println("Number of Strings:");
            int numElements = Console.readInt();
            // fill array with num elements numbers input by the user
            System.out.println("Please enter " + numElements + " strings");
            String[] strings = new String[numElements]; // dynamic array
            for (int elementNum = 0; elementNum < numElements; elementNum++)
                  strings[elementNum] = Console.readString();
            /*
             * Print the array input* in the previous loop
             */
            for (int elementNum = 0; elementNum < strings.length; elementNum++)
                  System.out.println(strings[elementNum]);
            // print 0th array element
            String s = strings[0]; // unsafe
            for (int i = 0; i < s.length(); i++)
                  System.out.println(s.charAt(i));
      }
```

A for loop has the syntax:

```
for (<Initializing Statement>;<Boolean Expression>;<Updating Statement>)

      <Body Statement>
```

It is equivalent to:

```
<Initializing Statement>;

while (<Boolean Expression>) {

  <Body Statement>;

   <Updating Statement>;

}
```

The <Body Statement> is called the for body.

As we saw before in the main header, an array variable whose elements are of type <Element Type> is declared as:

<Element Type> <Array Variable>

We see here that it can be initialized to an actual arrauy using by assigning it the expression:

new <Element Type> [<Array Size>]

The array size can be an arbitrary expression in Java, which, as we see above, can involve variables. An array whose size can involve variables is called a dynamic array.

Single line comments begin with "//" and extend to the end of the line. Mult-line comments begin and end with /* and */ respectively and there is a standard way to format them, shown above. There are also standard ways to format comments that describe methods, into which we will not delve in this course.

Comments, Comments vs. Modularity

Comments, of course, document code so others, and even you, can understand it later. However, more comments is not always better, and the fact that some of your code requires comments can be bad sign.

Here is a rewrite of the code above that requires fewer comments:

```
public static void modularReadAndPrintStrings() () {
 String[] strings = readStrings(readNumStrings());
 printStrings(strings);
 printString(strings[0]);// unsafe
}
public static int readNumStrings() {
 System.out.println("Number of Strings:");
 return Console.readInt();
}
public static String[] readStrings(int numElements) {
 System.out.println("Please enter " + numElements + " strings");
 String[] strings = new String[numElements]; // dynamic array
 for (int elementNum = 0; elementNum < numElements; elementNum++)
  strings[elementNum] = Console.readString();
 return strings;
}
public static void printStrings(String[] strings) {
 for (int elementNum = 0; elementNum < strings.length;
      elementNum++)
 System.out.println(strings[elementNum]);
}
public static void printString(String s) {
 for (int i = 0; i < s.length(); i++)
  System.out.println(s.charAt(i));
}
```

Many of the comments have been replaced with reusable methods with descriptive names. Some of these names are long. Many programming environments provide commands to automatically complete long names. In Eclipse, CTRL-SPACE completes a prefix for some name you have entered.

As this example shows, we should not comment for the sake of commenting. Be clear who your audience is (yourself and the instructors of this course) and write comments that you think would be useful to at least one of these parties. When you cannot estimate what the instructors would need, just comment for yourself.

Comments vs. Mnemonic Role Names

Comments describing the role of a variable as in the following:

double w; // weight

can be replaced by mnemonic names that describe the role:

double weight;

Again, do not be resistant to using long names, given name completion techniques in programming environments.

Two corollaries of this discussion are that:

> *An identifier (name) such as a variable or method should have a single logically defined role.*
>
> *An identifier should be a mnemonic name describing its name.*

When memory was expensive, programmers used to use the same memory location for multiple purposes, overlaying multiple logical entities on top of a single physical memory location. Today, programming style trumps over efficiency.

Case Conventions

Comments and modularity are two techniques for making code understandable to others, ourselves, and even tools, as we will see later. Case conventions are another, much simpler technique. Three important conventions that have been established for Java programs:

- Start the name of a variable, method and package with a lower case letter.
- Start the name of a class with an upper case letter.
- To increase readability, we often need to create multi word identifiers. Since spaces cannot be part of an identifier, separate these words by starting each new word (that is, second and subsequent words) with an upper case letter. We followed this rule in the identifiers `AModularConsoleReadingUpperCasePrinter`, `Console`, and `readString`. The second rule above, of course, determines the case of the first letter of the first word. This is called the *camel case* convention.
- Name constants such as MAX_UPPER_CASE_LETTERS should consist of upper case letters and underscores.

As we study other kinds of namable entities such as interfaces we will extend these conventions suitably.

Formatting

The readability of programs can be greatly reduced by using formatting that shows the structure of the program. There are standard rules for formatting code, including comments. If comments are formatted using standard rules (called JavaDoc in Java), they can be used to create automatic documentation. Standard rules can be and are supported directly by interactive programming environments. Therefore, we will not delve into them, and expect you to use your own or standard rules for showing the structure of the program. If the structure is good enough for you to understand your code, it is good enough for us.

Annotations and Array Constants

Comments are detected and then ignored by the compiler. This means there is no trace of them at in the executable code produced by the compiler. Sometimes we do want the programmer to associate parts of the program with descriptions that are not ignored by the compiler and even used by it. Such associations are called annotations. Like comments, you can put them before the code unit you want to tag. They have the syntax:

```
@<Annotation Type> (<Annotation Parameters>)

< Annotatable Code Unit>
```

Annotatable code units include variable, class, package and method definitions. The following is an example of an annotation:

```
@Deprecated

public static void bubbleSort (String[] aList) {

…

}
```

It tells the reader and, since it is available at runtime, even a potential caller of this method, that method bubbleSort() is available for those who still rely on it but it is no longer recommended that it be called (as it might have, for instance, replaced with a better alternative). This annotation takes no parameters, so nothing goes in between the parentheses. An annotation does not end with a semicolon and multiple annotations can be associated with the same code unit. An annotations are typed and available a runtime, they are a better alternative than comments when applicable.

Deprecated is an example of a predefined Java annotation. It is also possible for programmers to defined their own annotations. One such annotation, provided by the ObjectEditor library, is the Tag annotation, which takes an array of tag names as an argument.

```
@Tags({“Modularity”, “Scanning”, “ConsoleInput”})

public class AModularConsoleReadingUpperCasePrinter {
        …

}
```

This annotation allows a Facebook-like tag to be associated with an annotatable item such as a method or a class. It takes as an argument an array of Strings constants to be used as tags. Here we have tagged our example class with Strings describing the concepts illustrated by the class AModularConsoleReadingUpperCasePrinter. The example shows that we can create an array constant by enclosing a list of comma separated element values within curly braces.

As you will see later, the Tag annotation will be a crucial mechanism for communicating to us the functionality implemented by your classes and methods.

Formal vs. Actual Parameters

We have used the word “parameter” or “argument” above for two distinct but related concepts. We have used it both for the identifiers such as numElements, declared in the header of methods such as readStrings() , and also expressions such as *readNumStrings(),* supplied before executing the method. The former, are in fact, called the *formal parameters* of the method definition, while the later are called the *actual parameters* of the method execution.

Empty and Compound If and Loop Bodies

Consider the following code:

```
   if (Character.isUpperCase(nextLetter)) ;
    System.out.print(nextLetter);
```

Some but not all of you have probably noticed the semicolon after the Boolean condition of the if. It is an indication to Java that the then-part of theif is empty. As a result, the print statement is considered the statement following the if rather than the then-part of the if. An empty then part make sense only if the else part is not empty – in which case we can invert the Boolean condition and make the original else part the then part. A similar problem can occur with while and for statements as empty while and for bodies are allowed in Java.

To increase readability and reduce errors, we can derive the following rule:
A statement nested in another should never be empty.

When we detect an upper case letter, we might want to, in addition to printing it, also record the fact that we found an uppercase. We may add an extra assignment statement, as shown below:

```
   if (Character.isUpperCase(nextLetter))
    System.out.print(nextLetter);
    foundUpperCase = true;
```

The code, however, does not, however, achieve its purpose, as regardless of its indentation, the assignment is considered the statement after the if rather than a part of then statement. This problem is less likely if we had made the original then statement a compound statement containing exactly one statement:

```
if (Character.isUpperCase(nextLetter)) {
    System.out.print(nextLetter);
}
```

Which would have been transformed to

```
if (Character.isUpperCase(nextLetter)) {
    System.out.print(nextLetter);
    foundUpperCase = true;
}
```

It is for this reason that Python uses indentation rather than curly braces to group statements.

In Java, we can avoid such errors by always insisting that:
The then and else parts of an if and the body of a loop statement should always be a compound statement unless the if or loop statement is a single line.

Thus, according to this rule, the following is acceptable, as it does not cause indentation-based confusion:

```
if (Character.isUpperCase(nextLetter)) foundUpperCase = true;
```

To save space, in the class material, this rule will be broken frequently.

Summary

Java objects, methods, and classes correspond to real-world objects, the operations that can be performed on them, and the factories that manufacture the objects, respectively.

A class definition consists of a class header identifying its name and accessibility; and a class body declaring a list of methods.

A method definition consists of a method header identifying its name, accessibility, formal parameters, and return type; and a method body specifying the statements to be executed when the method is called.

Formal parameters of a method are variables to which actual parameters are assigned when the method is called.

The set of values that can be assigned to a formal parameter/returned by a method is determined by the type of the parameter/method.

The process of writing and executing code we have seen so far consists of declaring one or more methods in a class, compiling the class, asking the interpreter to start ObjectEditor, asking ObjectEditor to instantiate the class, and finally, invoking methods on the new instance.

By following style principles in writing code, we make the code more understandable to us, others, and tools such as ObjectEditor.

Other Resources

Code package: lectures.objects

Slides: PowerPoint PDF

Exercises

Define class, method, object, variable, formal parameter, actual parameter, type, statement, return statement, and expression.

What are the case conventions for naming classes, methods, and variables?

Define syntax error, semantics error, logic error, accessibility error, and user error.

Why is it useful to follow style principles?

Suppose we need a function that converts an amount of money in dollars to the equivalent amount in cents and can be invoked by ObjectEditor. The following class attempts to provide this function Identify all errors and violation of case conventions in the class.

```
class aMoneyconverter {

 int Cents (dollars) {

return dollars*100;

}

}
```

Write and test a function, `fahrenheit()`, that takes a parameter an integer centigrade temperature and converts it into the equivalent integer Fahrenheit temperature, as shown in the figures below. Assuming F and C are equivalent Fahrenheit and centigrade temperatures, respectively, the conversion formula is:

F = C * 9/5 + 32

Implement the function in the class `ATemperatureConverter.` Use a bare-bones programming environment to develop and execute the class.