A Little Book on Java

Fatima Ahmad and Prakash Panangaden

August 30, 2001

Contents

1 Introduction

These notes are a brief introduction to Java and to some basic object-oriented concepts. They should not be viewed as a comprehensive textbook but they should be more than adequate for learning Java in conjunction with the lectures and assignments. These notes were written specifically to cover those aspects of Java that arose in CS250 during the Fall of 1998. The emphasis in the course is algorithms and not programming languages. To properly appreciate the subtleties of the object-oriented paradigm, one needs to take a course on programming languages, such as CS302.

Java is a relatively new, but very popular object-oriented programming language. It builds on the syntax of C and is closely related to C++, but there are key differences. I assume that you know the elements of programming in some language such as Pascal or C (it does not really matter what) so the basic aspects of programming are not talked about at all; we will just give the basic Java syntax and semantics.

These notes will not tell you everything about Java. They will, however, tell you where to find out more. In particular, we will tell you how to find out information about Java from the official Java documentation available free on the internet.

There are three aspects of Java that one might cover:

- basic concepts,
- details of the libraries and utilities,
- advanced programming tips and secrets.

It is our goal to cover the first topic. You will not be very knowledgeable about the other two topics from this course alone. However, you will be able to learn those things easily in the context of a job or project – if you need to – later on. These are pricisely the things that you can teach yourself whereas learning the concepts is not easy without help.

2 Are you ready to program in Java?

We expect you to know nothing about Java before you read this book. We do however expect you to have a fair amount of programming experience in either Pascal or C. If you are not familiar with the following concepts, we strongly urge you to take an introductory programming course in Java (such as CS202) before tackling CS-250: *statements, conditionals and loops*, *expressions* and *types*, *variables*, *arrays*, *procedures*, *call by value* versus *call by reference* and *simple IO*.

We expect that you understand what it means to compile and run a program. If your only experience with computers has involved using preexisting packages then you need an introduction to programming first.

3 Installing Java

The official Java website can be found at `http://www.javasoft.com`. Among its many features, this site provides free downloads for Java and complete Java documentation. The latest version of Java is called "Java 2 SDK v1.3.1". The main homepage for this distribution is:

http://java.sun.com/j2se/1.3/

The page with all the documentation is:

http://java.sun.com/j2se/1.3/docs/index.html

In order to insure proper installation, do read the file that describes all steps that need to be taken for successful installation. In particular, pay attention to the section pertaining to setting classpaths. To be able to compile and run Java code from any directory, the classpaths on your system must be set properly.

4 Java Documentation

Complete documentation for Java SDK v1.3.1 can be found at

java.sun.com/j2se/1.3/docs/

This page lists and provides a description of all packages supported by Java SDK v1.3.1. These packages contain predefined Java classes (you will learn about classes later). The page also lists all Java classes in alphabetical order and provides a link to the definition for each class (i.e. the variables, constructors and methods provided by the class - do not worry if you do not know as yet what these words mean). An appropriate time to browse through the Java docs would be after you have learned about classes and objects.

You can look up all sorts of information including books at:

http://developer.java.sun.com/developer/infodocs/

For starters, get familiar with looking up information on Java on the web by finding the Java Language specification. You can also practice by looking up information on Java operators at

http://web2.java.sun.com/docs/books/tutorial/java/nutsandbolts/operators.html

5 Java Basics

In this section we review basic Java ideas. These are things you should know from a previous course in another language, so the discussion will be brief.

5.1 A Simple Java Program

A Java program is stored in a `.java` file. Let us create a simple Java program that you can compile and run. At this point, we do not expect you to understand the program's syntax. Type the following Java program using a text editor such as "NotePad"[1] or preferably "emacs" or "vi" - basically any editor that produces plain text output[2].

```
class First{
  public static void main(String args[]){
    System.out.println(``This is my first Java program!!!'');
  }
}
```

Make sure that you typed the code exactly as above. Note that Java is case sensitive and the code that you typed will contain errors if you did not preserve the case of the words. Save the program in a file called `First.java`. Note that this program can be saved in a file with any name as long as it has a `.java` extension.

Compiling and Running

Let us compile and run `First.java`. At this stage, we just want you to go through the motions of compiling and running a Java program. Later, we will go through the details of what happens when a Java program is compiled and run.

To compile `First.java`, type `javac First.java` at the command prompt and press enter. The program `javac` is the Java compiler and the command `javac First.java` creates a file called `First.class`.

If your classpaths are set properly, then you can compile your program from any directory. Otherwise, the system does not know where to find `javac`. `javac` is contained in the `bin` directory of the Java SDK. If your classpath has not been set, then you need to compile your program from the bin directory of the Java SDK[3]. Another alternative is to specify the location of `javac` (i.e. the full path to `javac`) each time you compile. This latter option is somewhat cumbersome, but it does allow you to compile from any directory. Again, read the README file to find out how to set classpaths.

To run `First.java`, type `java First` at the command prompt and press enter. The program `java` is the Java Interpreter, which executes Java programs by running the class files generated by compiling the program. In this case, the Interpreter runs `First.class`. Note that the command is `java First` and *not* `java First.class`. You must run `java` from the same directory in which the program was compiled. If you have successfully compiled and run your first Java program, your screen should display the following output: This is my first Java program!!!

[1] Notepad is not a good editor for use with programming languages; such editors designed for text.

[2] So Word and Wordperfect are right out!

[3] Needless to say, this is a very bad idea; you should set the paths.

Comments in Java

Comments can be written in two ways. Any text on the same line after two consecutive backslashes (//) is a comment. Any text enclosed within /* and */ is a comment. Comments are not nested, meaning that the first /* will end at the next */. The `//` is best used for short comments that come on the same line and the other notation can be used for longer comments.

This is a good place to say something about how to comment. Comments should be there to explain what the program is doing. Every program should have comments at the beginning that say briefly what the program does and what the general methods used are. It might be appropriate to say what kinds of assumptions are made - for example the nature or format of the expected input. Each method (procedure) should have comments specific to that procedure. Anything tricky should be explained but obvious things should not be belaboured. Here are examples of good comments and bad comments.

```
//Below is a good comment

/* The following code allows one to swap two values without using a
temporary variable. */
x = y - x;
y = y - x;
x = x + y;

//Comments below are bad

x = 0; //x is set to zero

int x; //x is declared

int x; //x is used to count the number of iterations of the loop

int x; //Nobody cares about me except my mother
```

One very good use of comments is to indicate the end of methods or classes.

5.2 Names in Java

Most things in a program have *names*. Data, procudures (methods), portions of memory, types and classes[4] are all named. The ability to give names was one of the innovations in high-level languages. When a name is used there has to be a *declaration* saying that a new name is being introduced and what *type* the name will stand for. Thus a name might be a

[4]It does not matter if you do not know as yet what a class is.

way of referring to a memory cell - this is what is usually called a *variable* - or be the name of a method (procedure).

In Java, declarations can be introduced in the middle of the text. In other languages, one has to make declarations at the beginning of the program or at the beginning of a block. We will see examples of names and declarations below.

5.3 Variables in Java

A variable is a name for a piece of memory. Java is a strongly typed language, which means that the datatype of a variable must be specified when a variable is declared. The datatype of a variable can be any Java datatype. We will cover some basic Java datatypes in the next section.

In a variable declaration, the datatype of the variable is written before the variable's name. One can initialize a variable at its declaration using the '=' operator. Multiple variables of the same datatype can be declared on the same line as long as they are separated by commas. In the following examples `float`, `int` and `double` are all names of Java datatypes.

```
/* Just declaring a variable of type integer */
int var;

/* Declaring more than one variable of the same datatype,
float in this case */
float var1, var2, var3;

/* Declaring a variable and initializing it */
double var4 = InitialValue;
```

5.4 Basic Java Datatypes

A datatype is used to organize data into collections that reflect structural and/or operational similarity. Java provides eight primitive data types for numbers, characters and booleans. These primitive datatypes are a part of the Java language. In this section, we will cover some of Java's primitive datatypes as well as datatypes for strings and arrays. A list of all of Java's primitive datatypes can be found at

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/vars.html

5.4.1 Numeric Datatypes

Java provides primitive datatypes for integers and floating point numbers. There are four such datatypes for integers, of which the most commonly used is `int`. Of the two datatypes (`float` and `double`) for floating point numbers, use `double` as it provides greater precision.

5.4.2 The Character Datatype

In Java, the `char` datatype denotes character values. Single quotes are used to specify characters.

```
char a = 'a';   //Declaring and initializing a character variable
```

5.4.3 The Boolean Datatype

The `boolean` datatype contains only two values: `true` and `false`. This datatype can be used to denote the result of logical tests. Logical tests determine whether a certain logical condition has been met. The value of the test is `true` if the test's condition has been met and `false` otherwise. In the language C there is no boolean datatype. Having this datatype in Java reduces many common errors that occurred in C programs.

5.4.4 Strings

A string is a sequence of characters and is denoted by the `String`[5] datatype. In Java, double quotes are used to specify a string, thus distinguishing a string from a character.

```
String title = ``A Little Book on Java'';
String p = ``p''; //A string containing one symbol
String P = ``p'';
```

Note that `p` and `P` are different variables since Java is case-sensitive.

Variables of type `String` are Java objects. As we have yet to cover objects, we will not go into a further discussion of what can be done with objects of type `String`. Later, when you have learned about objects, you can look up class `String` in the Java documentation on the web to find out more about strings.

5.5 Arrays

It is often useful to cluster data of the same type in a collection. An array is a set of memory locations containing data of the same datatype. The most important aspect of arrays is that the elements of the array are accedded by numerical indices. Furthermore these indices can be computed.

An array can contain data of any Java datatype. Like strings, arrays are Java objects. In Java, an array of size n has indices from 0 to $n-1$. Recall that we expect you to be already familiar with arrays in Pascal or C (or Fortran or Basic).

When declaring an array, one must specify the size of the array and the datatype of its elements. The general declaration of an array is as follows:

[5]Note that the first letter is capitalized.

```
datatype[] ArrayName = new datatype[ArraySize];
```

This is how you declare an array of integers containing 250 elements:

```
int[] numbers = new int[250];
```

Note the use of the `new` keyword in the above declaration. We will discuss the importance of this keyword when we discuss objects. Briefly, the `new` actually creates space for the array while `int[] numbers` just says that the name "numbers" will be used for an array of integers (of unspecified size).

After an array has been created, its elements must be initialized. So far we have named the array and reserved space for it but we have not put any data in the array. Each element of an array has an index, with the first element having the index 0 and the last element having an index one less than the size of the array. Using an index, one can modify or obtain the array element at that index. This is the key aspect of arrays. By using expressions for the indices one can have names that are *computed* at run-time.

```
/* A small array of doubles */
double[] nums = new double[3];

/* Initializing the array */
nums[0] = 1.0;
nums[1] = 2.0;
nums[2] = nums[0] + nums[1];
```

An error occurs if one tries to access an array index that does not exist, i.e. an index that is less than zero, or greater than or equal to the size of the array. This error is a Java exception called `ArrayIndexOutOfBoundsException`. We will discuss exceptions in a little more detail in section 15.

5.6 Expressions and Statements in Java

An *expression* is anything that evaluates to a value. Thus in Java, `x + y` is an expression. When it occurs it is a signal that a calculation must be done. Not all expressions are arithmetical. The word is used whenever there is a *resulting* value.

By contrast a *statement* is an "order" to do something. There is in general no result from executing a statement. Here is a basic example.

```
x = 2 + 3;
```

Above is a statement, which contains the expression $2 + 3$. This statement is a command that can be rendered in English thus:

1. Compute the expression $2 + 3$.
2. Store the result of the expression evaluation in the memory cell named by x.

The statement of the previous paragraph is an example of the assignment statement. You must have seen it many times before. However it is worth being absolutely clear about several things now. A general assignment statement looks like this:

```
<name> = <expression>
```

The name on the left-hand side has to be the name of a memory cell. The expression on the right hand side has to produce a value. The type declared for the cell and the type of the expression must match. Later we will see that the "name" can be the result of a computation. Thus the most general form is:

```
<expression-which-produces-a-name> = <expression-which-produces-a-value>
```

We will see examples of this more general form later on. For the first few weeks you will not encounter it.

Things get more interesting when the same name occurs on both sides of the expression as in:

```
x = x + 2;
```

Or as C hackers like to write it:

```
x += 2;
```

Now the interpretation of the name is *different* on the two sides of the assignment operator (i.e. the `=` sign). In order to appreciate this properly, note that the name x is the name of a memory cell (here a cell for storing integers). This cell *contains* a value. The name x is used both to refer to the cell and to its contents. In the example at hand, on the left hand side the name x names the cell and on the right hand side it names the *value* currently stored in the cell.

Certain notations are both expressions and statements. A typical example is the construct $x++$. This means "increase the current value of x by 1"; but it is also a statement and it returns the old value of x. The construct $++x$ means - as a statement - "increase the current value of x by 1" (just the same as $x++$) but as an expression it returns the new value of x. Thus if x is 1729 and we write `y = x++` we will get 1729 stored in y but x will now be 1730. If we were to write `y = ++x` then both x and y would be 1730.

5.6.1 Blocks

In order to write interesting programs, we need mechanisms to put together complex statements from simple ones. One of the basic forms is the *block*. A block contains zero or more statements enclosed in parenthesis. A block is a compound statement and can always replace a single statement. We will see examples of blocks below.

5.7 Basic Java Control Constructs

Java provides looping mechanisms and conditional statements that allow us to alter the control flow of execution. In this section, we will discuss the syntax of some basic loops and conditional statements. Recall that we expect you to be familiar with the concept of loops and conditional statements. The notation may change from one language to another but the ideas are the same.

5.7.1 Looping Mechanisms in Java

A loop repeats the execution of some code a certain number of times. We will discuss three loops: the `while` loop, the `do-while` loop and the `for` loop.

The while Loop The syntax of a `while` loop is as follows:

```
while <boolean-expression>
  statement
```

As long as the boolean expression specified in the header of the `while` loop evaluates to `true`, the statement is executed. The while loop exits when the boolean expression is false. If the boolean expression is false the first time it is evaluated, then the statement is never executed. Note that the statement that the while loop repeatedly executes can also be a block.

The do-while Loop The syntax of a `do-while` loop is as follows:

```
do
  statement
while (boolean-expression);
```

The `do-while` loop is equivalent to the `while` loop except that the statement is executed at least once. The boolean-expression is evaluated after the statement is executed. The loop body is executed as long as the boolean expression remains true.

The `for` Loop The most general loop form is the `for` loop. The syntax of a `for` loop is as follows:

```
for (initial-expression, boolean-expression, increment-expression)
  statement
```

A `for` loop is almost the same as the following:

```
initial-expression;
while (boolean-expression){
  statement
  increment-expression;
}
```

A `for` loop iterates over a range of values given to a variable. The variable must have an initial value, which is specified by initial-expression. The value of the variable is modified in some way; usually it is incremented as specified by increment-expression. One can also decrement the value. The boolean-expression tests whether the value of the variable is within the range that specifies the number of iterations for the `for` loop.

5.7.2 Conditional Statements

We will cover the `if` statement. The syntax of the `if` statement is as follows:

```
if (boolean-expression)
  statement
```

If the boolean-expression evaluates to true, then the statement is executed. An `if` statement can be be used in conjuction with the `else if` and/or `else` clauses as follows:

```
if (boolean-expression)
  statement
else if (boolean-expression)
  statement
else
  statement
```

Here is a simple example of a loop and a conditional. In this fragment, A is an array containing some numbers and the loop searches through looking for the largest of the numbers.

```
/* Code fragment to find the largest number in an array by iterative
search.  The variable ‘‘size’’ stores the size of the array. */

//Start by assuming the first number is the largest
int max = A[0];

//Search until the end
for (int j = 0; j < size; j++){
  //Found a new possible max
  if (A[j] > max) max = A[j];
}
```

Note that the `if` statement did not have an `else` part in this example.

Java provides a special statement called `break` which can be used to exit a block of code. The syntax of the `break` statement is as follows:

```
break;
```

One can use the `break` statement to exit loops early. Here is an example of a loop that searches an array for a given number and stops the search as soon as it finds the number.

```
/* Code fragment that searches an array for a given element. The
variable ‘‘elt’’ stores the element we are looking for, while the
variable ‘‘size’’ stores the size of the array. */

int i = 0;

//Set to true if elt is found
boolean found = false;

while (i < size){
  //If the element is found, stop looking
  if (A[i] == elt){
    found = true;
    break;
  }

  else i++;
}
```

5.8 Conclusion to the Basic Part

This completes our review of the basic aspects of Java. Except perhaps for our emphasis on the role of names, all of this should have been familiar to you, possibly with different notations. There are all sorts of details that we have not told you. You should look up the Java documentation to find out about these as needed. The next step will be the study of larger units of code.

6 Abstraction Mechanisms

A vital part of an advanced programming language like Java is *abstraction*. By abstraction we mean *packaging* code and data into organized units in such a way that the details are *hidden*. There is a tendency for beginning programmers (and writers of beginning textbooks) to emphasize control constructs and data manipulation and spend less time on overall program

organization. This might be reasonable for small programs, but as programs get larger it is imperative to think about the overall program structure. In this class we will be moving towards medium sized programs and the principles of program organization will be more important than mastery of all the control constructs.

Programs are the most complex mechanisms built by humans. There is no hope for any one person to keep the details of a large program in mind. The only viable strategy is for the program to be broken into small manageable pieces. These pieces will *interact* with each other and this interaction has to be controlled in a structured way. Thus, when we decompose a program into pieces we have to think at two levels. We have to understand a piece of code *ignoring all the interactions with the rest of the program* and then, we have to understand the interactions between the pieces *without thinking about the details of the code inside each piece*. I emphasize this because this is the part where many beginners have trouble. They tend to be obsessed with understanding each piece of code at *the lowest level of detail* and cannot ignore details even when it is appropriate to do so.

The basic abstraction mechanisms are *procedural abstraction* and *data abstraction*. In procedural abstraction, a piece of code - called a *procedure* - is packaged together and *given a name*. Thereafter, this code can be run or invoked simply by using this name. This is done not to save typing (as many students seem to think) but to enforce abstraction levels. When the procedure is defined you do not worry about all the different places in the program where it might be used, when the procedure is used, you do not think about how it was coded.

In data abstraction, some collection of data is organized into a structured form (a data structure) and some code is provided to manipulate the data while maintaining the integrity of the structure. The abstraction consists of hiding the details of the data structure. The point of this is to force you to use the code provided rather than to mess with the structure directly. This way you are less likely to violate the integrity of the structure. Just as in procedural abstraction, hiding is the key. We will see examples of data abstraction and procedural abstraction in Java in the next two sections.

The correct way to think of abstractions is as a contract. When you write a procedure you have in mind that a solution to a certain well-defined task is to be implemented. You do not think about what a user might do with the procedure. Abstractions like this occur in everyday life. Perhaps one of the best examples of abstraction is in a car. The driver of a car is usually not an expert engineer. In fact, relatively uneducated people can drive cars well, despite the fact that there is a significant amount of sophisticated engineering involved. Furthermore when one operates a car by using the controls, one has to be aware that some basic contracts are fulfilled. Thus turning the wheel causes the car to turn and pressing the brake slows down the car. One need not have the faintest idea of how this is accomplished. *In fact in an emergency, it is important to* ***not*** *waste your time thinking about auto mechanics, you have to spend your mental effort dealing with the emergency.*

The brake and the other controls are like procedures. They fulfill a certain contract but you need now know how this is done. The designer of the car certainly has to know how this is done. But she does not have to think about what the driver might do with the car. She does not think about the fact that you are going to use the car to commute to work, or go on

a date or go to the store. The separation of concerns is complete, as the only link between the two levels is the control panel of the car. This is the *interface* between the low-level engineering and the higher-level human concerns of the person using the car.

We will be using all these concepts in programming. We will define procedures - called methods in object-oriented programming - and data abstractions - using classes. We will define interfaces and we will use these concepts to structure a program into manageable pieces. A modern object-oriented programming language like Java supports these ideas directly.

6.1 Procedures, Parameters and Types

A procedure needs to interact with the rest of the program. This happens in two phases. The program may want to *run* the procedure so that it does whatever it is supposed to do. When the procedure is done, it needs to *return* control back to the part of the program that called it. Thus the basic control flow is the following: a procedure is *called* (by using its name), the rest of the program stops and waits (suspends) while the procedure runs (executes) and, finally, the procedure completes and returns control to the suspended program.

At each stage the procedure and the program may need to communicate. When the procedure is called it *may* need some data to work with and when it finishes, it *may* want to return the result of some computation that it has performed. The data passed to a procedure is named by special variables called *parameters*.

In order to use a procedure we need to know three things:

- its name,
- what kinds of parameters it expects (if any),
- what kind of result it might return.

The information in the latter two items is called the *type* of the procedure. In a typed language like Java, these items have to be explicitly specified.

A simple example of these ideas is a procedure for performing multiplication. To use it you have to know its name, say `times`. You have to know that it expects *two* integers and that the result is an integer. Thus we would say that the type of `times` is

$$integer * integer \rightarrow integer.$$

In programming languages we do not use this explicit notation with the arrow but we essentially say the same thing. Notice that to use the procedure `times`, we need not have any idea how this was coded. In fact, we often use mathematical libraries knowing the names *and types* of procedures without knowing how they were coded.

You might think that we have left out something obvious, namely we need to know *what* the procedure does even if we do not know *how* it does it. This is very true *but the language provides no way of ensuring that a procedure does what you think it does.* Thus if you want

to perversely to call the multiplication procedure `add` or `foobar`, nothing in the language stops you. This part of the contract is informal. This is one reason why we need comments that document what a procedure does.

7 Classes and Static Methods

This section has two parts: classes are concerned with data abstraction and methods are concerned with procedural abstraction. We will emphasize procedural abstraction more for now.

7.1 Overview of Classes

A Java program is built from classes. In Java, all code is written inside a class. Classes form the basic organizational unit of all Java programs. For now, think of a class as a module that organizes a cluster of code. It is essentially a data abstraction mechanism. In other words, it provides mechanisms for hiding things.

At first we will see simple classes that will only contain constants, variables and methods (Java's version of procedures or functions). Later we will study objects and see how they fit with classes. In doing so, we will introduce a new type of abstraction mechanism, i.e. *object abstraction*.

7.1.1 Syntax of Class Declarations

A class is declared using the `class` keyword, followed by the name of the class. The body of a class is enclosed within curly brackets. Here is a class declaration:

```
class Hello{
/* The body of this class contains nothing right now */
}
```

The name of a class must begin with a letter, but can otherwise contain both letters and digits. The Java compiler raises an error at compile-time if the name of a class is one of Java's reserved words.

7.1.2 Class Variables

A class is a package for some data and some code designed to interact with the data. The data packaged in the class - like any other data - is referred to through the use of variables. Thus, a class can have variables that belong to it. These variables are called class variables or *static variables*. Do not worry about the strange use of the word "static"; it just means that the data referred to by the variable is associated with the class.

A variable is declared to be static using the *static* keyword, followed by the datatype and the name of the variable. The class mechanism hides its data from the rest of the

program according to some precise rules spelled out below in section 8. As far as the internal structure of the class is concerned, static variables can be accessed and modified by all methods contained in a class. Let us revisit class `Hello` and add a static variable to it.

```
class Hello{

  //Static variable
  static String s = "Hello?";

}
```

This is still not a very exciting class. It just contains a datum and provides nothing one can do with it. A class needs to have code that can manipulate the data. This is what we now turn to with our discussion of methods.

7.2 Static Methods

In any language, there is some mechanism for *packaging* a piece of code together in such a way as to view it as a single entity. This is what we have called *prodedural abstraction* in the discussion of the last section. Typically, one uses the word "procedure" (Pascal) or "function" (C, C++) for such constructs. In Java they are called methods. We will start with a very simple kind of method called *static* methods. Just as with static variables, the word "static" means that the methods are associated with the class. Later - when we study objects - we will look at non-static methods (which are just called "methods").

Like all code in Java, static methods are contained in classes. A static method is declared using the `static` keyword, followed by the return-type and the name of the method. The parameter list of a static method directly follows the method's name and is enclosed in parenthesis. Each parameter is declared with its datatype followed by its name. Commas separate multiple parameters. If there are no parameters, the parenthesis for the parameters are present but empty. The general syntax of the header of a static method is as follows:

```
static <return-type> name(type 1st-Parameter, ... ,type nth-Parameter)
```

The return-type of a static method can be any Java datatype. A static method that returns nothing has its return-type given as `void`. The name of a static method must begin with a letter, but can otherwise contain letters and digits. The Java compiler raises an error if the name of a method is one of Java's reserved words.

The body of a static method is enclosed within curly brackets. Variables can be declared within a method's body: these variables are local to the method. Local variables are undefined until they are initialized. Accessing the value of an undefined variable leads to a compiler error. A method with a return-type other than `void` must contain a `return` statement. The `return` statement must be placed such that all paths through the method end with it.

Let us add a simple static method to class `Hello`.

```
class Hello{

  //Static variable
  static String s = "Hello?";

  //Static methods

  //This method modifies the class variable s
  static void helloWorld(){
    s = "Hello, world!";
  }//helloWorld

}//class Hello
```

Note the use of comments in the above code to indicate the end of a method and a class.

7.2.1 The main Method

A class cannot be executed unless it contains a special method called `main`, which has the following syntax:

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

    /* Body of main */
}
```

A class that is run must contain a method called `main`, as when a class is run, the code in `main` is executed first. From the `main` method of a class, one calls other methods of the class.

The header of the `main` method must be written as above. Do not worry about the `public` keyword. We will discuss what this keyword means in section 8. The `main` method is static. It takes as its parameter an array of type `String` called *args*. The size of this array is not specified. The contents of this array can be specified at the command prompt when a class is run. Suppose we want to provide parameters to the `main` method of a class named `foo`. Then the parameters to the `main` method of class `foo` are specified after the command `java foo` as follows:

```
java foo a b c
```

The parameters *a*, *b* and *c* are read as strings into array *args*. After these parameters have been read into *args*, its contents are as follows:

```
args[0] = "a"
args[1] = "b"
args[2] = "c"
```

Let us add a main method to class `Hello` so that we can run it.

```
class Hello{

  //Static variable
  static String s = "Hello?";

  //Static methods

  //This method modifies the class variable s
  static void helloWorld(){
    s = "Hello, world!";
  }//helloWorld

  public static void main(String args[]){
    System.out.println(s);  //Prints the value of s
    helloWorld();
    System.out.println(s);
  }//main

}//class Hello
```

Save class `Hello` in a file called `Hello.java`. Compile the source code in `Hello.java` through the command `javac Hello.java`. To run, type `java Hello`.

7.2.2 Method Execution

It is time to look at exactly what happens when a method is used. We will use a simple example from arithmetic.

```
class Arith {

  //Static variables
  static int a,b;

  //Declaration of method add
  static int add(int n, int m){

    int r = 0; //Local variable for add method, initializaed to 0
    r = n + m;
    return(r);
  }//add
```

```
  public static void main(String[] args) {
    int p = 0; //Declares and initializes variable local to main

    //Assign values to the class variables
    a = 1729;
    b = 4104;

    p = add(a,b); //Method add is called, result is stored in p.

    System.out.print(p); //Print p on the screen
  }//main
}//class Arith
```

This class has two static variables a and b, which are declared to be integers. It also has a static method `add` which expects two integers and will return an integer as a result. We see that from the type information given with the declaration for `add`.

There are two other kinds of names appearing in the program. First, there are *local variables* - variables that are part of a method - such as r (local to `add`) and p (local to `main`). Second, there are *parameters* namely the n and m which are parameters of the method `add`.

Local variables are *temporary* storage, used when a method is active and *removed* when the method finishes. Parameters are names used to refer to data that is passed to a method when it is called. Let us see what happens when a method is called. A special area of memory is reserved for the method. This is called the *activation record*. The following space is reserved in the activation record. There is space for each local variable, space for each parameter and space to remember where the control has to return when the call is complete (the return address). The local variables get values like any other variables when an assignment is made. The parameters get values when the call is made.

The precise rule for passing values to methods (procedures) varies from language to language, and many languages support different mechanisms for passing parameters. In Java, all parameters are *passed by value*. This means the following. When the call is made, the arguments to the method are evaluated first. The resulting values are stored. In the example above, the actual arguments are a and b. These are evaluated to yield 1729 and 4104 respectively. What is stored in the activation record are the values 1729 in the memory cell named by n and 4104 in the memory cell named by m. Thus we get *copies* of the values. We cannot affect the values of a and b from inside the `add` method, we can only affect the copies. The copies of parameters are destroyed when a method exits. In fact, the whole activation record is destroyed. The next time a method is called there is no memory of what it had done before.

7.3 The Dot Operator

To access static variables and static methods inside the class to which they belong, one can simply refer to the said variables and methods by their names. This is not the case when one wants to access static variables and static methods belonging to other classes. One can access static variables and static methods belonging to other classes using the dot(.) operator. Suppose one wants to access method `showFoo` belonging to class `foo` from class Bar. This is done as follows:

```
Foo.showFoo();
```

The dot(.) operator is preceded by the name of the class containing the method that needs to be called. The method name follows the dot(.) operator. Thus in general, one calls a static method belonging to another class as follows:

```
classname.methodname(parameters);
```

Similarly, one calls a static variable belonging to another class as follow:

```
classname.variablename;
```

Note that one can only successfully access those methods and variables belonging to another class that are not declared `private`. We will discuss what this keyword means in greater detail in sectionr̃efvisibililty.

Compiling and Running a Java program

So far, you have only seen Java programs with one class. A Java program can however contain more than one class.

Suppose `Foobar.java` contains two classes named `Foo` and `Bar` respectively. One compiles the source code in `Foobar.java` by typing `javac Foobar.java` at the command prompt. When a Java program is compiled, a `classname.class` file (where classname is the name of a class) is generated for each class contained in the program. The Java compiler converts the source code of each class into bytecode which is stored in the class files. Compiling `Foobar.java` results in the creation of two class files: `Foo.class` and `Bar.class`.

One runs a Java program by running one of its classes. To run a class, type `java classname` at the command prompt. The bytecode stored in the class file of the class whose name is typed at the command prompt is read by the Java Virtual Machine and executed.

Not all classes contained in a program can be run. A class that can be run must contain a method called `main`, as when a class is run, the code in `main` is executed first.

It is perfectly possible for a program to contain several classes with the method `main` defined. This means that one can compile a program once and test its different parts by invoking Java on different class names.

8 Visibility Issues

The variables and methods contained in a class can be accessed by all code in that class. In order for classes to be an effective data abstraction mechanism, we will need to *hide* the code and data in a class from other classes. One can control whether code in *other* classes can access the variables and methods in a class through the use of the keywords: `public`, `private` and `protected`. We will refer to these keywords as *access modifiers*. In this section, we will only discuss the `public` and `private` keywords. For the purposes of this course, you do not need to know what the `protected` keyword means.

Public data and code is visible to all classes, while private data and code is visible only inside the class that contains it. To control the visibility of a method, place `public` or `private` as the first word in the method header i.e. before the `static` keyword for static methods and before the return-type for instance methods (you will learn about instance methods in the next section). To control the visibility of a variable, place `public` or `private` as the first word in the variable declaration: before the `static` keyword for static variables and before the datatype for instance variables (again, you will learn what these are in the next section).

One can also declare a class as `public` or `private`. We will not be going into a discussion on the visibility of classes except that a public class must be contained in a `.java` file that has the same name as the public class. One file can thus contain only one public class. To declare a class `public` or `private`, place `public` or `private` before the `class` keyword in the class declaration.

9 Classes and Objects

Recall our discussion of classes in section 7. We defined a class as a module that contains a cluster of code. In this section, you will discover that a class is much more than just a module containing code. We will discuss two new definitions of classes and how objects relate to classes.

9.1 Why Static is not Enough

We have introduced classes as a data abstraction mechanism and methods as a procedural abstraction mechanism. We have so far seen static variables and static methods. Is this enough? Let us look at the class `Arith`. It has two integers a and b packaged inside. This is fine if we never ever need more than two integers but one is very unlikely to only work with some fixed number of integers. Well we can repeat the pattern of the class definition and make new classes. But this is silly! Repeating all those class definitions again and again and pretending that they are different. Clearly we want to have different "things" but all of the same class. This is exactly what objects are.

10 The Object Concept

One of the revolutionary ideas of computer science is that code is just another form of data and can be treated as such. This idea was the basis of the fundamental conceptual breakthroughs by Turing and von Neumann in the 30s and 40s which made computer science possible.

In our discussion of abstraction mechanisms, we talked about procedural abstraction and data abstraction. We saw how methods captured the idea of procedural abstraction and classes captured the idea of data abstraction. This separation of abstractions makes an artificial distinction. In fact, it makes sense to cluster code and data together into a single entity called an *object*. In order for this to be possible, the distinction between code and data has to be blurred. An object is treated like data; it is better to say an object *is* data. It can be created and passed as an argument to a method. It can be the result of a method and it has a type just like other data. However, part of an object is code as an object carries methods in addition to other forms of data.

What is the type of an object? This is where classes come in. Classes have two roles with respect to objects. They constitute types for objects and they are also generators of objects. Let us ignore the second role for the moment. Objects of the same "family" are grouped together by classes. Now, a class declaration becomes a description of an object. *There may be several objects of the same class present at the same time.* Furthermore, individual objects have their own *names*. We discuss the role of names below - be warned that names for objects are a source of great confusion for beginners.

In summary, an object is a package of data and code. An object belongs to a class. Each object is an *instance* of the class to which it belongs. There are two roles that a class plays for its objects: it provides a type for its objects and it is a generator of its objects. The name of a class is the type that a class gives to its objects. A class contains *constructors* that allow one to generate or create objects of that class. Both roles will be discussed below.

10.1 The State of an Object

An object contains data. The data stored in an object specifies useful information about the object: it is thus the *state* of the object. The non-static or *instance* variables of a class determine the data stored in the objects of that class. These variables form the *fields* of the object.

Instance variables are declared the same way as static variables, except for the omission of the `static` keyword. Each instance variable defines a separate field of an object. Each object has its own copy of the instance variables defined in its class. Instance variables belong to the objects of a class and not the class itself. It is thus possible for each object to preserve a distinct state.

Here is a simple class that contains instance variables.

```
class PointIn3D{
```

```
  //Instance Variables
  private double x;
  private double y;
  private double z;

}//class PointIn3D
```

Each object of class `PointIn3D` represents a point in three-dimensional space. Each instance of class `PointIn3D` contains its *own copy* of the x, y and z fields. This is the distinction between static variables and instance variables. We do not have a special word to designate instance variables because they are actually much more common than static variables.

Instance variables can be given initial values upon declaration. The fields of an object can also be explicitly initialized when that object is being constructed (see below).

10.2 Constructors

We need a way of manufacturing new instances of a class, i.e. new objects. A *constructor* is a special method that creates an object of the class that contains it. A constructor has the following syntax:

```
access-modifier NameOfConstructor(parameter-list){
  /* body of constructor */
}
```

A constructor may be declared `public`, `private` or `protected`. If a constructor is `private`, then it cannot be called from a class other than the one that contains it, i.e. other classes cannot create objects of a certain class using a private constructor. The name of a constructor is the same as the name of the class that contains it. A constructor contained in class `PointIn3D` thus has `PointIn3D` as its name. The parameter list of a constructor is written the same way as the parameter list of any other method. We do not declare a return type for a constructor because it returns a new object of the class. Notice that we are using classes as types for objects.

A class may contain more than one constructor. The constructors of a class are distinguished by their types; i.e. by the number and types of their parameters. No two constructors belonging to the same class are allowed to have the same type. The parameters of a constructor are used to initialize the fields of the object that the constructor creates. If the instance variables of a class are not initialized at their declaration, then these fields should be initialized when an object is being created. Let us add constructors to class `PointIn3D`.

```
class PointIn3D{

  //Instance Variables
```

```
  private double x;
  private double y;
  private double z;

  //Constructors

  //This constructor does not take parameters
  public PointIn3D(){

    /* Initializing the fields of this object to the origin,
       a default point */
    x = 0;
    y = 0;
    z = 0;
  }

  //This constructor takes parameters
  public PointIn3D(double X, double Y, double Z){

    /* Initializing fields of this object to values specified by
       the parameters */
    x = X;
    y = Y;
    z = Z;
  }

}//class PointIn3D
```

Any of the two constructors above can be used to create a `PointIn3D` object. The second constructor differs from the first one as its parameters specify values for the fields of a newly constructed object. Of course, we do not really need the first constructor, but perhaps it is useful to have since it might be used so often.

10.3 Creating an Object

An object is created using the `new` keyword followed by a call to a constructor of the appropriate class. Let us create objects belonging to class `PointIn3D`.

```
//Creates a PointIn3D object with coordinates (0, 0, 0)
new PointIn3D();

//Creates a PointIn3D object with coordinates (10.2, 78, 1)
new PointIn3D(10.2, 78, 1);
```

Figure 1: A Reference as a Pointer

Figure 2: Two references to the same object

10.4 Object References

How do we use and manipulate objects once they are created? We need a way to refer to them. They are referred to by *references*. One may think of a reference to an object as a special value that tells you where an object is stored. In many languages, references are called *pointers* but I prefer not to use this word. What about *names*? Do objects have names? One often says that various objects have names but the situation is subtle. We will explain once we see some examples.

An object's *reference* is used to manipulate the object. A reference is declared as follows:

```
ReferenceType ReferenceName;
```

A reference declaration is similar to a variable declaration. The ReferenceType is the type of the object for which the reference is intended. Recall that a class provides a type for its objects; this type is the name of the class. The ReferenceName is any name for the reference. Let us declare a reference for an object of type `PointIn3D`.

```
PointIn3D p;
```

p is now a reference to an object of type `PointIn3D`. Declaring an object reference does not create the object for which the reference is intended. An object reference must be explicitly assigned to an object. Since `p` has not yet been assigned to an object, it is a `null` reference, that contains `null` as its value. Let us assign p to an object of type `PointIn3D`.

```
p = new PointIn3D(1, 1, 1);
```

Now that we have assigned p to an object, its value is no longer `null`. Instead, p contains the *address* of the object to which it was assigned. p does not contain the actual object that it was assigned, but the address of that object. This distinction is illustrated in figure 1.

A single object can have more than one reference. Let us create another reference for the object referred to by `p`.

```
PointIn3D q = p;
```

q now contains the same value (i.e. the same address) as p as shown in figure 2.

10.5 Accessing the Fields of an Object

We can access a field of an object using the dot(·) operator on a reference to that object as follows:

```
ReferenceName.FieldName;
```

The above statement returns the appropriate field of the object whose reference we used. Attempting to access a `private` field from outside the object's class with the above statement will result in a compiler error. Note that all fields of a `PointIn3D` object are `private`. This measure of privacy limits the access of these fields by code in other classes. It also limits the ability of other classes to corrupt the fields of a `PointIn3D` object.

10.6 The Behavior of an Object

An object can contain code. The code stored in the object defines the operations that can be performed on the object: it is thus the *behavior* of the object. The behavior of an object is determined by the non-static or instance methods of its class. Instance methods can be used to manipulate and access the state of an object.

Instance methods are declared the same way as static methods except for the omission of the `static` keyword. Like, instance variables, each object contains its own copy of its instance methods.

An instance method is *invoked* on an object using the dot(·) operator on a reference to that object as follows:

```
ObjectReference.InstanceMethodName(Parameter-List)
```

If a `private` method is invoked on an object from outside the object's class, a compiler error occurs. Let us add some instance methods to class `PointIn3D`.

```
class PointIn3D{

  //Instance Variables
  private double x;
  private double y;
  private double z;

  //Constructors

  //This constructor does not take any parameters
  public PointIn3D(){

    /* Initializing the fields of this object to the origin,
       a default point */
```

```
  x = 0;
  y = 0;
  z = 0;
}

//This constructor takes parameters
public PointIn3D(double X, double Y, double Z){

  /* Initializing fields of this object to values specified by
     the parameters */
  x = X;
  y = Y;
  z = Z;
}

/* Returns the x field of this object i.e the object
   on which this method is invoked */
public double getX(){
 return x;
}

//Returns the y field of this object
public double getY(){
  return y;
}

//Returns the z field of this object
public double getZ(){
  return z;
}

//Sets the x field of this object to X
public void setX(double X){
  x = X;
}

//Sets the y field of this object to Y
public void setY(double Y){
  y = Y;
}

//Sets the z field of this object to Z
```

```
  public void setZ(double Z){
    z = Z;
  }

  //Prints the x, y, z fields of this object
  public void print(){
    System.out.println("x = " + x + " y = " + y + " z = " + z);
  }
}//class PointIn3D
```

The instance methods defined in this class allow us to access and modify the fields (i.e. the state) of a `PointIn3D` object. These methods are `public` and can be used by code in any class.

10.7 Passing Object References as Parameters

An actual object can never be passed as parameter to a method or a constructor: what can be passed is a reference to the object. Recall that parameters are passed by value in Java. When a reference is passed as a parameter, a copy of the address stored in the reference is passed. Both the original reference and its copy refer to the same object. Thus, using the copy of the original reference, one can manipulate and modify the actual object. Since the original reference itself is passed by value, its value cannot be changed.

Let us add a few static methods to class `PointIn3D`.

```
class PointIn3D{

  //Instance Variables
  private double x;
  private double y;
  private double z;

  //Constructors

  //This constructor does not take any parameters
  public PointIn3D(){

    /* Initializing the fields of this object to the origin,
       a default point */
    x = 0;
    y = 0;
    z = 0;
  }
```

```
//This constructor takes parameters
public PointIn3D(double X, double Y, double Z){

  /* Initializing fields of this object to values specified by
     the parameters */
  x = X;
  y = Y;
  z = Z;
}

/* Returns the x field of this object i.e the object
   on which this method is invoked */
public double getX(){
 return x;
}

//Returns the y field of this object
public double getY(){
  return y;
}

//Returns the z field of this object
public double getZ(){
  return z;
}

//Sets the x field of this object to X
public void setX(double X){
  x = X;
}

//Sets the y field of this object to Y
public void setY(double Y){
  y = Y;
}

//Sets the z field of this object to Z
public void setZ(double Z){
  z = Z;
}

//Prints the x, y, z fields of this object
```

```
public void print(){
  System.out.println("x = " + x + " y = " + y + " z = " + z);
}

/* Static Methods */

//Sets the coordinates of PointIn3D p1 to zero
public static void makeZero(PointIn3D p1){

  // The fields get permanently changed
  p1.x = 0;
  p1.y = 0;
  p1.z = 0;

  /* The reference p1 is set to null, but the original reference
     whose copy is passed to this method is not changed */
  p1 = null;
}

/* Returns true if the coordinates of PointIn3D p1 are equal to the
coordinates of PointIn3D p2 */
public static boolean isEqual(PointIn3D p1, PointIn3D p2){
  return (p1.x == p2.x) && (p1.y == p2.y) && (p1.z == p2.z);
}

public static void main(String args[]){

  PointIn3D p, q, r, s;   //Declaring references

  //Assigning references to objects
  p = new PointIn3D(3, 4, 5);
  q = new PointIn3D(9, 8, 6);
  r = p;  //r and p are two references to the same object
  s = new PointIn3D(3, 4, 5);

  //Display the coordinates of all 4 PointIn3Ds
  System.out.println("Here are the coordinates of p, q, r, s: ");
  System.out.print("p: "); p.print();
  System.out.print("q: "); q.print();
  System.out.print("r: "); r.print();
  System.out.print("s: "); s.print();
  System.out.println(); //Skip a line
```

```
    /* Do p and r refer to the same object? Check if both references
    are equal */
    if (p == r)
      System.out.println("p and r are references to the same object!");

    //Do p and s refer to the same object? Compare both references
    if (p != s)
      System.out.println("p and s don't refer to the same object");
    System.out.println(); //Skip a line

    System.out.println("Let us set the coordinates of p to zero: ");
    makeZero(p);
    //Does the reference p point to null?
    if (p != null)
     System.out.println("p is not null");

    System.out.println("Here again are the coordinates of p, q, r, s: ");
    System.out.print("p: "); p.print();
    System.out.print("q: "); q.print();
    System.out.print("r: "); r.print();
    System.out.print("s: "); s.print();
    System.out.println();
  }

}//class PointIn3D
```

In the `makeZero` methods, we are passing references to `PointIn3D` objects. Using these references, we can access and modify the fields of the actual objects. Note that because any reference that is passed to a method is a copy of the original reference, we cannot modify the original reference in the method. In the `makeZero` method, the reference `p1` is assigned to `null`. The original reference `p` is not modified.

Copy class `PointIn3D` into a `.java` file. Compile the program and run class `PointIn3D`.

10.8 The `this` reference

Java provides a special reference called `this`, which can be used inside instance methods or constructors. Inside an instance method, `this` is a reference to the object on which the instance method is invoked. Inside a constructor, `this` refers to the object that the constructor just created.

Within instance methods and constructors, any instance variable that is accessed or any instance method that is invoked without a reference has a `this` reference implicitly attached

to it. Let us rewrite a constructor and an instance method of class `PointIn3D` using the `this` reference:

```
public PointIn3D(){
  this.x = 0;
  this.y = 0;
  this.z = 0;
}

public double getX(){
  return this.x;
}
```

The above constructor and instance method are equivalent to those previously written in class `PointIn3D`.

11 Linked Lists: An Extended Example

In this section we give a discussion of linked lists in Java. The array construct allows one to have collections of a fixed size but one often needs data structures that are *dynamically growing*. In other words, they should grow as needed during the program. For example, most databases have to have the ability to expand as they are used and new data is inserted. The linked list is a paradigmatic example of a dynamically growing structure.

The basic ingredient that we need is the `new` construct, which allows one to create a new object dynamically (the jargon is "on the fly"). Let us imagine having just a list of numbers. Thus a list will consist of a sequence of objects. Each object will contain a number and whatever else we need to form the list. The question is: "How do we glue all these objects together to form a single structure?" The answer is simple: each object should contain the number and a reference (pointer) to the next cell. Think of a herd of elephants with each one holding onto the tail of the next so as to stay together.

The basic unit will be called a `cell`, we will have a class definition for cells. Here it is in all its glory!

```
class Cell{

  //Instance variables
  private int item;
  private Cell next;

  //Constructors
  public Cell(int n){item = n; next = null;};
```

```
  public Cell(int n, Cell c){ item = n; next = c;};

  //Instance methods
  public int first(){ return item;};
  public Cell rest(){ return next;};
  public void SetItem(int n){ item = n;};
  public void SetNext(Cell c){ next = c;};


  //Static (Class) Methods

  public static void display(Cell c){
    System.out.print("The list is: [ ");
    while (c != null) {
      System.out.print(c.item + " ");
      c = c.next;
    }
    //Println flushes the line of buffer
    System.out.println( "]");
  };

};
```

Do not worry about all the methods just yet. Note that the basic data is an integer and a reference to another cell. Note that the class definition is recursive! There are two constructors one for making a brand new cell and one for making a new cell and linking it with an existing chain of cells. The basic data is declared private but the instance methods allow one to see and to modify the value and the next link. We have provided a class method to display a sequence of cells. To recapitulate: each cell object has data representing a number and a link to another cell. The code for the instance methods is copied in each object (at least conceptually) and there is a class method for displaying the sequence of cells. Why did we not make this an instance method? It was a design choice, it could very well have been an instance method.

In order to work with linked lists and *hide all the manipulation that goes on*, we will package up the sequences of cells inside another class called `List` with its own class definition. The class `list` will contain the sequence of linked cells as private data. As before, we have access methods that allow us to see the private data. The point is that the private data can be manipulated but only through the methods that we have provided.

```
class List{

  //Instance variables
  private Cell list; //This is the actual list
```

```
//Constructors
public List(){
  //Used to construct an empty list
  list = null;
}

public List(Cell c){
  list = c;
}

public List(int n, Cell c){
  list = new Cell(n,c);
}

public List(int n){
  list = new Cell(n);
}

public List(int n, List l){
  list = new Cell(n,l.list);
}

//Instance methods
public int car(){
  //Accessing the first cell
  return list.first();
}

public List cdr(){
  //Accessing all but the first cell, note the type
  return (new List(list.rest()));
}

public boolean IsNull(){
  return (list == null);
}

public List copy(){
  if (IsNull()) return new List();
  else return new List(car(), cdr().copy());
}
```

```
//Concatenates a list to this list, nothing is destroyed
public List append(List l){
  if (this.IsNull()) return l.copy();
  else if (l.IsNull()) return this.copy();
  else return new List(this.car(),this.cdr().append(l));
}

//Returns a reversed copy of the list
public List reverse(){
  if (IsNull()) return new List();
  else return cdr().reverse().append(new List(car()));
}

//Class methods

public static void display(List l){
  Cell.display(l.list);
}

public static void main(String[] args){
  int i;
  List l,l1,l2, l3;

  //Small test
  l1 = new List(3,new List(4));
  display(l1);

  //Larger test
  l = new List();
  for (i=10;i > 0; i--) l = new List(i,l);
  display(l);

  //Examples of using the methods
  Cell foo = new Cell(1);
  foo.SetNext(new Cell(6));
  Cell bar = new Cell(28);
  bar.SetNext(new Cell(496));
  foo.rest().SetNext(bar);
  l3 = new List(foo);
  display(l3);
}//main
```

```
}//class List
```

A natural question to ask is "why did we have this two-level structure?". After all, the linked structure already existed when we defined class `Cell`. Class `List` was only there as packaging. One response is that packaging is important. A more precise answer appropriate for the present situation can be seen by considering the following question. What would happen if we wanted to insert a cell into a sequence? Suppose that we had no class `List` and just worked with the class `Cell`. Suppose that we had a sequence of cells named s and a number n and we wished to insert n (first making a new cell for it) into the sequence. Perhaps s is sorted and we want to insert n in such a way as to keep it sorted. Now suppose that n fits somewhere in the middle of s. This is not a problem at all. The following code does this. We shall view it as a new static method added to the class `Cell`. Make sure that you understand the piece of code below before reading further.

```
/* The following method makes a new cell for n and inserts it into
   a sorted sequence of cells starting with c in such a way as
   to preserve the sorting. */

  public static void cellInsert(int n, Cell c){

    Cell temp = new Cell(n); //Make a new cell for insertion
    trailer = null; //Keeps track of the cell just behind the current

    /* We use temp.next to search through the list starting at the
       beginning and advancing whenever the value of n is bigger than
       the current value.  */

    temp.next = c; //Initializing temp.next to the start

    while ((temp.next != null) && ((temp.item) > ((temp.next).item))){
      trailer = temp.next;
      temp.next = (temp.next).next;
    }//while

    if (trailer == null) //We never did the while loop
        c = temp; //So the new item goes in front
    else trailer.next = temp; //New item is linked into the middle
  }
```

In order to insert a number into s, we would call `cellInsert` with s as the second argument. This code has a serious problem. It only works if the new value being inserted is not right in front. Suppose that n is smaller than the smallest item in s (this is handled

by the case `trailer == null`) and we perform the assignment `c = temp`. What actually happens? Well, remember that parameter passing is by value. So in the activation record for `cellInsert`, there is a cell named `c` which has a reference to the actual cell s. When we perform the assignment `c = temp`, we change this copy but the original cell `s` still points to whatever it used to point to, it will not point to `temp`. We need a handle on the first cell.

One solution is to keep this handle in a special reference. But this is exactly what the list class does. It has a private reference (called `list`) which is a reference to the first cell in its list of data. By packaging this inside an object we keep it secure and accessible only through our methods. Now we can keep the cell insertion code above, but in the class `List` we have an insert method that first ensures that the new item should not be inserted in front before calling `cellInsert`. If it has to be inserted in front the List class deals with it directly. This is done in the method (of class `List`) shown below.

```
public void insert(int n){
  if (list == null)
    list = new Cell(n);
  else if (n < list.first())
    list = new Cell(n,list);
   else
    Cell.CellInsert(n,list);
}
```

The basic design is that class `Cell` is an auxiliary class which manages the lowest level of data manipulation. Class `List` is a higher-level package which is what a user of this class might actually work with.

12 Inheritance

One of the fundamental advances in the object-oriented paradigm is the ability to *reuse* code. It often happens that you find yourself coding a small variation of something that you had coded before. If your code is organized into classes, you might observe the following patterns. The new code that you want to write is a new class, but it looks just like an old class that you wrote a while ago except for a couple of methods. It is to handle situations like this that we have the notion of inheritance.

The following code is a basic example.

```
class myInt {

  //Instance variable
  private int n;

  //Constructor
  public myInt(int n){
```

```
    this.n = n;
  }

  //Instance methods
  public int getval(){
    return n;
  }

  public void increment(int n){
    this.n += n;
  }

  public myInt add(myInt N){
    return new myInt(this.n + N.getval());
  }

  public void show(){
    System.out.println(n);
  }
}
```

This class just has an integer in each object together with some basic methods. It is not a class we would really write. It is here for illustrative purposes only.

Now imagine that we decide to have "integers" made out of complex numbers. These numbers are called gaussian integers used by the great mathematician Gauss for his work in number theory. He discovered that as an algebraic system, these numbers behaved very much like ordinary integers. We might want to extend our ordinary integers to deal with these gaussian integers. Here is the code that does it. The keyword `extends` in the class declaration tells the system that you want all the code that worked for class `myInt` to be present in class `gaussInt`. We say that `gaussInt` *inherits* from `myInt`. We also say that `myInt` is the *superclass* and that `gaussInt` is the *subclass*.

```
class gaussInt extends myInt {

  //Instance variable
  private int m; //Represents the imaginary part

  /* We do not need the real part that is already present because we
     have inherited all the data and methods of myInt.  Thus the
     private int n is also present in every instance of a gaussInt. */

  //Constructor
```

```
  public gaussInt(int x, int y){
    super(x); //Special keyword
    this.m = y;
  }

  //Instance methods

  //This method is overridden from the superclass
  public void show(){
    System.out.println(
    "realpart is: " + this.getval() +" imagpart is: " + m);
  }

  public int realpart(){
    return getval();
  }

  /*The method getval is defined in the superclass. It is not defined
    here but it is inherited by this class so we can use it. */

  public int imagpart(){
    return m;
  }

  //This is an overloaded method
  public gaussInt add(gaussInt z){
    return new gaussInt(z.realpart() + realpart(),
                        z.imagpart() + imagpart());
  }

  public static void main(String[] args){
    gaussInt kreimhilde = new gaussInt(3,4);
    kreimhilde.show();
    kreimhilde.increment(2);
    kreimhilde.show();
  }
}//class gaussInt
```

There are a couple of things to note. In the constructor, we first want to use the constructor for the superclass; this is done with the keyword `super`. The `super` keyword invokes the superclass constructor and then continues with whatever is written next. The picture that you should have in mind is that an instance of `gaussInt` *contains* an instance of `myInt`.

Now you really want to inherit some of the methods unchanged, but some methods need

to be modified. Obviously the `show` method of `myInt` is no use for `gaussInt`. In the subclass, you can give a new definition for an old method name. Thus in the above class, we have inherited `getval` and `increment` unchanged but we have modified `show`. This is called *overriding*. From class `gaussInt` you cannot use the `show` method of the superclass as it is well hidden.

There is also a more subtle phenomenon called *overloading*. Look at the `add` method in the two classes. At first sight, this looks like overriding. However the types of the arguments expected by the two definitions of the `add` method are different. The types of the arguements are called the *signature* of the method. Now when we use the same name for a method *but with a different signature* we get two different methods *with the same name*. Both mwthods are available from the subclass. In this case, from the class `gaussInt,` we can use both `add` methods. How does the system know which one to use? It looks at the types of the actual arguments and decides which one to use. Thus if you want to add an ordinary `myInt` to a `gaussInt`, then the `add` method of the superclass is used.

There are many more subtle aspects to inheritance that we will go into in more detail in a later class. For now, it suffices to grasp the basic concept of inheritance and to recognize when to use it.

13 Subtyping and Interfaces

One of the features of object-oriented languages is a sophisticated type system that offers a possibility called "subtype polymorphism." In this section we will explain what this is and how it works in Java. One very important caveat when reading the extant literature: there is terrible confusion about the words "subtyping" and "inheritance". **They are not the same thing**! Yet one sees in books phrases like "inhertitance (i.e. subtyping) is very easy ..."! My only explanation is that lots of people really are clueless. Do not learn from them.

Recall that a type system classifies values into collections that reflect some structural or computational similarity. There is no reason that this classification should be into disjoint collections. Thus, for example, a value like 2 can be both an integer and a floating-point number. When a value can be in more than one type we say that we have a system that is *polymorphic* - from the Greek, meaning "having many shapes." A type system that allows procedures to gain in generality by exploiting the possibility that a value may be in many types is called a polymorphic type system.

The kind of polymorphism that one sees in Java is called "subtype polymorphism"[6] which is based on the idea that there may be a relation between types called *the subtyping relation*. Do not confuse subtyping with the notion of subset. We will say that a type A is a subtype of a type B if *whenever* the context requires an element of type B it can accept an element of type A. We write $A \triangleleft B$ to indicate this.

Here is the basic example of subtyping in many languages. Consider the types `int` and `float`. It is easy to see that `int` is a subtype of `float`, in symbols $int \triangleleft float$. Whenever

[6]There are other kinds, most notably *parametric* polymorphism. We will not consider this further.

you need a floating-point value an integer can be used but definitely not the other way. For example, we may have a method for computing the prime factors of an integer, obviously such a method would not even make sense of a floating-point number.

How do we set up the subtyping relation? There are some built in instances of subtyping – such as $int \lhd float$ – but, clearly, this is not worth making such a fuss about. Where subtyping really comes into its own is with user-defined types. In Java, subtyping occurs automatically when you have inheritance; *this does not mean that subtyping and inheritance are the same thing*. You can also have instances of subtyping without any inheritance as we shall see.

Thus, if we declare a class B to be an extension of class A, we will have - in addition to all the inheritance - that $B \lhd A$. In other words, if at some point you have a method `foo` which expects an argument of type A; `foo` will always accept an object of type B. If we extend class B with class C then we have

$$C \lhd B \lhd A.$$

If we extend A with another class D then we will have $D \lhd A$ but there will be no subtyping relation between B and D or C and D. A method expecting an object of type A will accept objects of type B, C or D. This gives increased generality to the code.

It is often the case that the inheritance hierarchy is not very "wide". In other words it is unlikely that a class A can be sensibly extended in many incompatible ways. This is because the code has to be inherited and usually only a few methods are modified. If we are modifying all or almost all the methods then it is clear that we are not really using inheritance. We are really trying to get the generality offered by subtype polymorphism.

Java supports subtype polymorphism independently of inheritance. This is done by **interfaces**. An interface is a declaration of a collection of method names - *without any method bodies* - and perhaps some constants. They describe a (sub)set of methods that a class might have. There is no code in an interface, hence there is nothing to inherit. A concrete class[7] may be declared to `implement` an interface. This is really a subtyping declaration. An interface names a new type (not a class) and when we say that a class P implements interface I we have set up the subtyping relation $P \lhd I$. Because there was no code in the interface, P does not inherit any code; but *it must have a method of the same name and type as every method name in the interface* I.

A class can implement many interfaces and thus one can have a complex type hierarchy with multiple subtyping. One often hears the phrase "interfaces allow Java to fake multiple inheritance". This is a source of confusion. What interfaces allow is multiple subtyping. Java definitely does not have multiple inheritance (C++ does have true multiple inheritance); wat it has is multiple subtyping.

Here is an example of the use of inheritance. Imagine that you have written code to draw the graph of a mathematical function. You want this to be abstracted on the function. You

[7]We use the adjective "concrete" to mean that the class is completely defined, i.e. it has all the actual code for the methods.

do not want to write code to plot a graph of the *sine* function and another different - but almost identical - piece of code to plot a graph of the *exp* function. You want the function - a piece of code - to be a *parameter*. We can do this with objects because objects can be passed around like any other piece of data but yet they carry code. What kind of object is a mathematical function? It expects a `double` argument and returns a `double` result. There might be other methods associated with function objects but the `plot` method does not care. It only cares that there is a method - called, say, `y` - such that `f.y(3.14159)` returns a double. So instead of having to know all about the details of the class of mathematical functions we just define an interface `plottable` with the one method `y` in it with the appropriate type. When we define our mathematical function objects we can make them as complicated as we please as long as we declare that they implement `plottable` and ensure that they really do have the method `y`. If we have another type of object - say `fin-data` - for financial data we would expect to define a totally different class with no relation to mathematical function objects. However, `fin-data` could also have a method `y` and be declared to implement `plottable`. Then our `plot` method works on totallly unrelated classes. We have achieves generality for our plotting method through subtype polymorphism in a situation far more general than could have been achieved by inheritance.

In between interfaces and classes are *abstract classes* that have some methods defined and some that are left blank as in interfaces. You can extend them as you would any class.

There are many subtle issues about method lookup and how it interacts with inheritance and type-checking. However, those issues are best discussed in a programming languages class. A future edition of these notes will deal with these issues.

13.1 Subtypes and Typechecking in Java

In this section we revisit the gaussian integers example and formalize the rules for type checking a little more carefully.

The following is a brief summary of the

> Rules for Method Lookup and Type Checking.

In particular we discuss the Guassian integers example from class.

First the rules. Remember that there are two phases: compile time, which is when type checking is done and run time, which is when method lookup happens. Compile time is before run time.

- The type checker has to say that a method call is OK at compile time.
- All type checking is done based on what the declared type of a reference to an object is.
- Subtyping is an integral part of type checking. This means if B is a subtype of A and there is a context that gets a B where A was expected there will not be a type error.

- Method lookup is based on actual type of the object and not the declared type of the reference.
- When there is overloading (as opposed to overriding) this is resolved by type-checking.

14 Analysis of the `gaussInt` Example

We recapitulate the full example:

```
class myInt {
  private int n;
  public myInt(int n){ this.n = n;}

  public int getval(){return n;}

  public void increment(int n){ this.n += n;}

  public myInt add(myInt N){ return new myInt(this.n + N.getval());}

  public void show(){
    System.out.println(n);}

}


class gaussInt extends myInt {
  private int m;  //represents the imaginary part

  public gaussInt(int x, int y){
    super(x);
    this.m = y;}

  public void show(){
    System.out.println(
    "realpart is: " + this.getval() +" imagpart is: " + m);}

  public int realpart() {return getval();}
  public int imagpart() {return m;}

  public gaussInt add(gaussInt z){
    return new gaussInt(z.realpart() + realpart(),
                        z.imagpart() + imagpart());
```

```
  }

  public static void main(String[] args){
    gaussInt kreimhilde = new gaussInt(3,4);
    kreimhilde.show();
    kreimhilde.increment(2);
    kreimhilde.show();

    System.out.println("Now we watch the subtleties of overloading.");
    myInt a = new myInt(3);
    gaussInt z = new gaussInt(3,4);
    gaussInt w;   //no object has been created
    myInt b = z;  //b and z are names for the same object
    //even though d and z refer to the SAME object they have
    //different types

    System.out.print("the value of z is: "); z.show();
    System.out.print("the value of b is: "); b.show();
    //which show method will be used?

    myInt d = b.add(b); //this does type check
    System.out.print("the value of d is: "); d.show();
    // w = z.add(b); will not type check
    // w = b.add(z); will not type check
    w = ( (gaussInt) b).add(z);//this does type check
    System.out.print("the value of w is: "); w.show();
    myInt c = z.add(a); //will this typecheck?
    System.out.print("the value of c is: "); c.show();

  }
}
```

Here is what we can say about the variables (TBD means "to be determined"):

```
Name        Declared Type   ActualType


a:          myInt            myInt
z           gaussInt         gaussInt
w           gaussInt         TBD
b           myInt            gaussInt
```

```
d           myInt           myInt
c           myInt           TBD
```

```
myInt a = new myInt(3);
gaussInt z = new gaussInt(3,4);
gaussInt w;
myInt b = z;

System.out.println("the value of z is"+ z.show());

> real part is 3 imag part is 4
```

this prints out the above line because z is declared to be of type `gaussInt`. It passes the type checker as there is a `show` method defined in the `gaussInt` class. At run time it uses the `show` method of `gaussInt` to display the above line.

```
System.out.println("the value of b is :" + b.show());

> real part is 3 and imag part is 4.
```

b is declared to be of type `myInt`. There is a method called `show` in the `myInt` class. The type checker sees that and because of that it passes the type checker, **but** the actual type of b is `gaussInt`. Method lookup is based on actual types of objects and therefor b uses the `show` method in the `gaussInt` class and displays what a `gaussInt` object would have shown.

```
myInt d = b.add(b)

System.out.println("the value of d is:"+ d.show());

> 6
```

b is declared to be of the type `myInt`, the type checker checks to see whether there is an `add` method in the `myInt` class. **Yes** there is one; it takes a `myInt` object and returns a `myInt` object as the result. At run time b's actual type is `gaussInt` the run-time system checks to see if there is an `add` method in the `gaussInt` class which matches the type that it was told by the type-checker. There are two add methods - one that takes a `myInt` and returns a `myInt` (This method has been inherited from the `myInt` class). The other takes a `gaussInt` and returns a `gaussInt`; this is the method that is explicitly defined in the `gaussInt` class. However the latter method does not match what the type-checker told the run-time system to expect.

NOW WHICH ADD METHOD DO WE USE?

since "When there is overloading, it is resolved by typechecking" the method which takes an object of the type `myInt` will be used. This is the method that has been inherited. It takes

in a `myInt` and returns a `myInt`. Hence `b.add(b)` returns a `myInt` object and therefor NOW the actual type of d is `myInt`.

```
//w= z.add(b) -----(i)
//w = b.add(z)-----(ii)
```

These two will not type check

1. z is declared to be of the type `gaussInt`. There are two methods in the `gaussInt` class, the one that takes in a `myInt` object and returns a `myInt` object is used. Why? Once again overloading is resolved by typechecking. Since b is declared to be a `myInt` object it will pick the `add` method that it inherited.

   z is a `gaussInt` which is a subtype of `myInt` and hence is added to b and returns a `myInt`. w is declared to be a `guassInt`. Since `myInt` is not a subtype of `gaussInt` the assignment statement will not accept this for the right hand side, and hence would cause an error.

2. b is declared to be of the type `myInt`. The type checker checks if there is an `add` method in the `myInt` class there is one which expects a `myInt` object and returns a `myInt`object z is a `gaussInt` and since `gaussInt` is a subtype of `myInt`, b is added to z to produce a `myInt` object

   w is declared to be a `gaussInt` Since `myInt` is not a subtype of `gaussInt` it will not accept it and hence would cause an error.

```
w = ( (gaussInt) b).add(z)
```

This does type check as it is just a little modification to case 2 above. Now since w is a `gaussInt`, it better get a `gaussInt` on the right hand side. However, now, because of the cast, the typechecker knows that b is really a `gaussInt`. Thus, it now has to choose between two possible `add` methods. To resolve the overloading it uses the declared types; z has declared type `gaussInt`. Thus when it resolves the overloading of the `add` method it figures out to use the `gaussInt` to `gaussInt` version.

15 Exception Handling in Java

Exception handling is a Java mechanism that serves two purposes: it allows the programmer to deal with runtime errors and it can be used to indicate special error conditions. Both uses of exception handling will be discussed below.

15.1 The Exception Object

In Java, runtime errors are called *exceptions*. An exception is a Java object. All exceptions inherit from the `Exception` class and its subclasses. You can find the definition of class `Exception` and a list of its direct subclasses at

http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Exception.html

Java divides exceptions into two groups: unchecked exceptions and checked exceptions. All unchecked exceptions inherit from the `RunTimeException` class, which is a subclass of the `Exception` class. Note that the name `RunTimeException` is misleading since all exceptions occur at runtime.

Runtime errors that occur because of programming errors correspond to unchecked exceptions. Unchecked exceptions happen because of the programmer's carelessness and can be prevented by the programmer through the introduction of checks in the code. Unchecked exceptions thus do not require exception handling. Two common unchecked exceptions are `ArrayIndexOutofBoundsException` and `NullPointerException`. Both can be prevented by simple checks in the code: the former exception would not occur if the programmer checks that an array index actually exists before accessing it, while the latter exception can be prevented by checking whether a reference contains `null` before using it to access fields or invoke methods.

All other runtime errors are categorized as checked exceptions. Checked exceptions are caused by situations out of the programmer's control. Checked exceptions require exception handling. Two common checked exceptions are `FileNotFoundException` and `IOException`. The former exception occurs when the user provides an input file that does not exist. The latter exception indicates that an error occurred while writing to or reading from a file.

15.2 Creating a New Exception

New exception classes can be created if none of Java's predefined exception classes correspond to a certain error. To create a new exception class, extend either the `Exception` class or any one of its subclasses other than `RunTimeException`. Recall that RunTimeException and its subclasses correspond to unchecked exceptions. Instances of any new exception class should be checked exceptions because of reasons you will discover below.

15.3 Throwing an Exception

An exception is *thrown* to indicate the occurrence of a runtime error. Only checked exceptions should be thrown, as all unchecked exceptions should be eliminated.

An exception is explicitly thrown using the `throw` statement. To throw an exception, first determine the type of exception you need to throw, then throw a new instance of that exception. You can throw one of Java's predefined exceptions or define a new exception class and throw its instance. Let us throw a `FileNotFoundException`.

```
throw new FileNotFoundException();
```

A method must add a `throws` clause in its header to indicate the types of all the checked exceptions that it may throw. This clause follows the parameter list in a method's header. Commas separate multiple exception types in the `throws` clause. Here is the header of a `main` method that throws an `IOException` and a `FileNotFoundException`:

```
public static void main(String[] args) throws IOException,
                                               FileNotFoundException
```

A method's header advertises the checked exceptions that may occur when the method executes. When creating a new exception class, it is important that its instances be checked exceptions, so that others are aware of the existence of a new type of exception that may be thrown. If a method's header does not contain a `throws` clause, then the method throws no checked exceptions.

Although exceptions can be thrown to indicate runtime errors such as a missing input file, one can also throw exceptions to indicate special error conditions. Consider a method called `search` that looks for an element in an array. The method either returns the first array index at which the element is found or -1 to indicate that the search was unsuccessful. One can alternatively throw an exception to indicate that an element was not found in the array. Let us write `search` and create a new exception class:

```
class ElementNotFoundException extends Exception{

  //Constructor

  /* This is a simple constructor that just calls the constructor of
  its superclass */
  public ElementNotFoundException(){
    super();
  }
}

public class bar{

  //Static methods
  public static int search(int[] A, int key) throws ElementNotFoundException{

    boolean found = false;
    int index = 0;

    while ((! found) && (index < A.length)){
      if (A[index] == key)
```

```
        found = true;
      else
        index ++;
    }

    if (found) return index;
    else throw new ElementNotFoundException();
  }//search


  public static void main (String args[]) throws ElementNotFoundException{

   /* Note that an ElementNotFoundException must be declared in the
   throws clause of this method as any of the calls to method search
   can cause this exception to be thrown */

    int[] A = new int[50];
    int k = 50;

    //Initializing A
    for (int i = 0; i < A.length; i++){
      A[i] = k;
      k--;
    }

    k = 23;
    int result = search(A, k);
    System.out.println(k + " found at " + result);

    k = 88;

    //This will cause an exception
    result = search(A, k);
    System.out.println(k + " found at " + result);

    k = 35;
    result = search(A, k);
    System.out.println(k + "  found at " + result);
  }//main
}//class bar
```

Compile the code above and run class `bar`. The source code must be saved in a file called `bar.java` as class `bar` is a `public` class.

We defined a new exception class called `ElementNotFoundException`, which extends the `Exception` class. An instance of the `ElementNotFoundException` class is a checked exception and must be declared in the header of a method that throws it. Since we explicitly throw an `ElementNotFoundException` in the `search` method of class `bar`, we declare that exception in the `throws` clause of the method's header.

An exception can occur in two ways: explicitly through the use of a `throw` statement or implicitly by calling a method that can throw an exception. We explicitly throw an exception in the `search` method of class `bar`. The `main` method of class `bar` implicitly throws an exception by calling `search`, a method that can throw an exception. Thus, the `main` method of class `bar` must also declare an `ElementNotFoundException` in its `throws` clause.

In general, it is not a good idea to indicate an error condition that will occur often by throwing an exception. This is so because catching exceptions takes a lot of time.

15.4 Catching an Exception

When an exception occurs, the normal flow of execution stops as the Java runtime system starts looking for the appropriate exception handling code. Code that handles an exception is found in a *catch* clause. To handle an exception, one must catch it. One catches an exception using a `try-catch` block. The general syntax of a `try-catch` block is as follows:

```
try{
   code that could cause exceptions
}
catch (Exception e1){
   code that does something about exception e1
}
catch (Exception e2){
   code that does something about exception e2
}
```

Code that can throw exceptions is placed in a `try` block as above. When an exception occurs, the normal flow of execution stops as the rest of the code in the `try` block is skipped. The Java runtime system starts looking for an appropriate `catch` clause for the exception that occurred.

The `catch` clause for an exception need not be in the method in which the exception is thrown. It can be in any of the methods that were called before the current method. Indeed, if a `catch` clause is not found in the method that throws the exception, then each of the calling methods is examined in turn for a `catch` clause. If no `catch` clause is found in any of the methods on the calling stack, then a default exception handler catches the exception. The default exception handler handles all exceptions in the same way: execution is stopped and the name of the exception and a stack trace are printed on the screen. This is the scenario that takes place when class `bar` is run.

If a `catch` clause for the exception is found, then the code in that `catch` clause is executed. After the execution of the `catch` clause, the first line of code not inside a `catch` clause is executed. A method that handles an exception need not declare that exception in its `throws` clause. If no exception is thrown by the code in a `try` block, then all the code in the `try` block is executed and the all the `catch` clauses associated with that `try` block are skipped.

Let us modify the `main` method of class `bar` to catch the exception thrown by the `search` method.

```
class ElementNotFoundException extends Exception{

  //Constructor
  public ElementNotFoundException(){
    super();
 }

}

public class bar{

  //Static methods
  public static int search(int[] A, int key) throws ElementNotFoundException{

    boolean found = false;
    int index = 0;

    while ((! found) && (index < A.length)){
      if (A[index] == key)
        found = true;
      else
        index ++;
    }

    if (found) return index;
    else throw new ElementNotFoundException();
  }//search

  public static void main(String args[]){

    int[] A = new int[50];
    int k = 50;


    //Initializing A
```

```
    for (int i = 0; i < A.length; i++){
      A[i] = k;
      k--;
    }

    try{
      k = 23;
      int result = s2(A, k);
      System.out.println(k + " found at " + result);

      k = 88;
      //This will cause an exception
      result = search(A, 88);
      System.out.println(k + " found at " + result);

      k = 35;
      result = search(A, k);
      System.out.println(k + "  found at" + result);
    }
    catch(ElementNotFoundException e1){
      System.out.println(k + " not found in array A");
    }

    System.out.println("End of main method of class bar reached.");
  }//main
}//class bar
```

We enclosed all the calls to `search` in a `try` block, since any of these calls could result in an `ElementNotFoundException`. The second call to `search` causes an `ElementnotFoundException`. The remaining code in the `try` block is skipped. The code in the `catch` clause corresponding to an `ElementNotFoundException` is executed. After the execution of the `catch` clause, normal execution resumes as the print statement after the `catch` clause is executed. Note that because the second call to `search` throws an exception, the third call to `search` is never reached. Indeed, if the first call to `search` caused an exception, then both the second and third calls to `search` would be skipped. The above `try-catch` block is poorly designed. It is much better to wrap each call to `search` in a `try` block. Thus, for each call to `search`, one can handle the possibility of an unsuccessful search.

```
class ElementNotFoundException extends Exception{

  //Constructor
  public ElementNotFoundException(){
    super();
```

```
  }

}

public class bar{

  //Static methods
  public static int search(int[] A, int key) throws ElementNotFoundException{

    boolean found = false;
    int index = 0;

    while ((! found) && (index < A.length)){
      if (A[index] == key)
        found = true;
      else
        index ++;
    }

    if (found) return index;
    else throw new ElementNotFoundException();
  }//search

  public static void main(String args[]){

    int[] A = new int[50];
    int k = 50;

    //Initializing A
    for (int i = 0; i < A.length; i++){
      A[i] = k;
      k--;
    }

    try{
      k = 23;
      int result = s2(A, k);
      System.out.println(k + " found at " + result);
    }
    catch(ElementNotFoundException e1){
      System.out.println(k + " not found in array A");
    }
```

```
    try{
      k = 88;
      //This will cause an exception
      result = search(A, k);
      System.out.println(k + " found at " + result);
    }
    catch(ElementNotFoundException e1){
      System.out.println(k + " not found in array A");
    }

    try{
      k = 35;
      result = search(A, k);
      System.out.println(k + "  found at" + result);
    }
    catch(ElementNotFoundException e1){
      System.out.println(k + " not found in array A");
    }

    System.out.println("End of main method of class bar reached.");
  }//main
}//class bar
```

Each call to `search` is wrapped in its own `try` block and each call gets executed. If any of the calls to `search` was not wrapped in a `try` block, then an `ElementNotFoundException` would have to be declared in the `throws` clause of the `main` method. Since we handle each `ElementNotFoundException` that may be thrown, we need not declare this exception in a `throws` clause.

16 Conclusions

This is the end of our brief survey of Java. To learn more about the language you should take a course on programming languages. There are many subtle issues that we have not looked at most notably subtyping and related issues. There are many details of various predefined Java classes, libraries and packages that we have not even mentioned. What we have said about the language is hopefully stable.

Acknowledgements

We would like to thank Haroon Ali Agha, Jacob Eliosoff, Etienne Gagnon, Patrick Lam, Moses Mathur and Maria Olaguera for helpful comments. We are very grateful to Alan

Shaver, Dean of the Faculty of Science for providing the funding that made these notes possible. The second author would like to thank his wife for reminding him to shave from time to time.

References

[AG98] Ken Arnold and James Gosling. *The Java Programming Language, Second Edition.* Addison-Wesley, 1998.

[HC99] Cay S. Horstmann and Gary Cornell. *Core Java 1.2.* Sun Microsystems, 1999.

[LL98] John Lewis and William Loftus. *Java Software Solutions.* Addison-Wesley, 1998.

[vdL99] Peter van der Linden. *Just Java 1.2, Fourth Edition.* Sun Microsystems Press, 1999.