COMP268 STUDY GUIDE

COMP268 Study guide

Table of Contents

Unit 0 Orientation

Overview

You are starting a course that is challenging but rewarding if you complete it successfully.

Learning your first programming language is usually difficult. You will be learning both a language and how to think like a computer as you solve information-processing problems.

When you write an essay, you have to know how to use the language and how to research a topic, develop the ideas, and construct a cohesive piece of work. Programming involves similar types of activities.

Programming is a practical skill that does not come naturally to most people. You have to work at learning programming. Different people require different lengths of time to learn the skills, which is no different from any other practical skill. Learning to drive takes different people very different lengths of time, but it will take you far less time to learn to drive a bus if you can already drive a car, since you already know a lot about driving and using the roads.

Your tutor’s workload assumes you follow the recommended schedule given below. Please do not expect your tutor to mark all your assignments at once, or in a very short period of time.

In a classroom situation or as a professional programmer (a

If you are new to programming, expect to spend between 200 and 300 hours completing the course and assignments. If you have past experience in programming, this time should be significantly reduced.

The course should be paced over three or four months. Do not try and rush it at the last moment, especially if it is your first programming course. You will need time to absorb the ideas in one part of the course before continuing with the next part. Everything software engineer), you usually have colleagues to help you. When you write an essay, it is helpful to have someone else validate your ideas, look for problems, and edit the work. A second pair of eyes is also very helpful in programming. While the mode of instruction remains individualized study, you are welcome to form small groups to study the problem, discuss designs to solve the problem, and share errors encountered. However, we still expect you to develop the code, document the code, and test the code by yourself.

you do will build on earlier parts of the course. Most of your time will be spent learning by programming rather than reading.

TOP

Learning Activities

COMP 268 builds on the concepts introduced in COMP 200 and/or COMP 210, and shows how to use Java language constructs to develop code and solve problems. Each unit introduces a number of topics using a range of material and activities. For each topic, you will design, write, and analyze Java code corresponding to that topic. **You need to pass *each* of the five activities for credit with a mark of at least 50% to pass the course.**

The **two assignments for credit** are available on the home page and are due after Unit 4 and 8. Each is worth 20% of your final grade. You must score 50% in each assignment to pass the course.

You are also required to complete a **midterm quiz**, worth 20% of your final grade. You must score 50% in the quiz to pass the course. Please contact your tutor and request the midterm quiz after your have completed assignment 1.

The **programming profile** activity is also worth 20% of your final grade. A programming profile is nothing but a collection of programs and your comments on each of these programs. You may wish to keep the programming profile as a directory in your computer. When complete, the entire programming profile (i.e., the directory) can be zipped into a single file and submitted on the Programming Profile link after the last unit of the course.

Include problems and solutions that you worked though in the units of the study guide. Minimally, we expect your programming profile to host a minimum of 10 problems, their solutions, and your commentaries on these problems and solutions. Your programming profile will be marked out of 100 marks based on the number of problems you have solved, the quality of your solutions, and, importantly, your reflective commentaries on your solutions. The problems and commentaries you choose to submit for your profile can be drawn from unit activities and/or work completed for credit.

The commentaries should include your personal opinion of the problem. For example, you could discuss whether the problem contributed to improving your coding competency, whether you found the problem difficult to solve, some of the coding challenges you faced, and how you overcame them. You need a minimum of 50% to pass the programming profile activity.

☞ You will also complete an **invigilated online final exam**, worth 20% of your final grade, to allow you to consolidate and demonstrate what you have learned. The final exam includes both programming questions and description questions. You must score 50% to pass the final exam. For more information about online exams, please consult the Online Exams (MuchLearning) section of the Procedures for Applying for and Writing Examinations in your online Student Manual , and visit http://registrar.athabascau.ca/exams/online/student/.

☞ **Tip**: Plan a regular time every day to spend on your course work.

COMP 268 General Discussion Forum

You are encouraged to participate in some collaboration with other students as part of completing this course. You can learn a lot by helping others—teaching is the best way to make sure you understand something yourself. **COMP 268 General Discussion** is for questions and observations about the material, the assignments, and about programming in general. Post to the forum and respond to others' postings on a regular basis for best results.

TOP

Common Problems

The most common mistake in learning to program is trying to write a program before you know how to solve the problem being presented. You first have to design a solution to the problem, and only then can you write the program. Design before you write. Document your design on paper first, and check the logic; then convert it to code.

Feedback from previous students indicates a range of time is needed to complete the assignments. The higher range usually indicates the student is stuck. If you find you are spending more than an hour struggling to get something working, stop and walk away for a while. You may have a blank screen and not know how to start, or you may be looking at your program and only seeing what you want to do, not what is really there.

At this stage you must seek help. Professional programmers rarely work in a vacuum. They routinely discuss problems and potential solutions. If these types of difficulties continue, seek help from your colleagues or your tutor. Do not post source code in your discussions that might deter others from writing code on their own. If you have to, post only the minimal amount of code (say, one or two lines at most) that is essential for the discussion.

Post your problem on the **COMP 268 General Discussion Forum** on the course home page. Often, someone else will have already solved the problem you are working on and will be able to provide help.

If you still are stuck, talk to your tutor. Explain what you have done, what you are trying to do, and what is going wrong. Your tutor will help guide you to solve your problem, but will not provide a working solution. The tutor is already a professional programmer. You are the one learning to program.

You are encouraged to discuss assignments with other students and seek help when needed. **However, it is essential that you fully understand any programs you submit. Your final exam will include questions modeled on the course exercises and assessments.**

The best way to learn a language is to use it. Read the code in the course pages. Discuss solutions. Try things out. The language seems the biggest hurdle at first. You will quickly see, with the help of the compiler and a good IDE (integrated development environment—we prescribe Bluej) that you can help resolve most language errors. However, only you can resolve logic errors. *Problem solving and solution design are the real core of programming*.

Most of all, when you complete this course successfully, remember that you have achieved something very difficult. Congratulate yourself on your achievement. You have started down the road to becoming a software engineer.

When you build a functioning program, you have created your own beautiful thing. Enjoy!

TOP

Suggested Study Schedule

The following schedule is provided to help you organize your time and complete the course successfully. For each unit and each section of the unit, suggested study times are also provided. These are based on the expected average—you may need more or less time depending on your previous exposure to the conventions of computer programming.

Click here for the schedule as a PDF file.

Suggested Weeks of Study	Units
Week 1	Unit 0: Orientation Unit 1. What is Java? What is in Java? Why Java? How Java?
Week 2	Unit 2. Variables, Operators, and Expressions in Java
Weeks 3–4	Unit 3. Control Structures in Java
Weeks 5–6	Unit 4. Classes and Objects in Java Submit **Assignment 1**
Weeks 7–8	Unit 5. Streams in Java
Week 9	**Break week.** It's not too soon to think about ordering your final exam. This has to be done from 20 to 60 days in advance, depending on your location. See http://calendar.athabascau.ca/undergrad/page07_02.php **Midterm Quiz** due
Week 10	Unit 6. Arrays, Vector, ArrayList, LinkedList, Queue
Week 11	Unit 7. Events in Java
Weeks 12–13	Unit 8. Applets Submit **Assignment 2**
Weeks 14–15	Unit 9. Concurrency in Java (aka Java Threads)
Weeks 16–17	Unit 10. Advanced Topics in Java
Week 18	Submit the **Programming Profile** Write the **Final Exam**

TOP

Criteria for Marking Programming Assignments

The following table presents the overall approach to marking a program. The first column lists the five key criteria on which your program will be marked. Functionality is the main criterion. The rest of them will receive proportionally reduced marks depending on the percentage of functionality implemented in the program. If you have implemented only half the functionality expected in the program, don't expect to receive full marks for documentation, test cases, and so on.

Rubric for Marking Programs	**Unsatisfactory < 50%**	**Satisfactory 50%–65%**	**Good 66%–85%**	**Excellent 86%–100%**
Functionality 20%	Completed less than 70% of the requirements. Not delivered on time or not in correct format (zipped files, Moodle submission, etc.)	Completed between 71% and 85% of the requirements. Delivered on time, and in correct format (zipped files, Moodle submission, etc.)	Completed between 86% and 95% of the requirements. Delivered on time, and in correct format (zipped files, Moodle submission, etc.)	Completed between 96% and 100% of the requirements. Delivered on time, and in correct format (zipped files, Moodle submission, etc.)
Coding Standards 20%	No name, date, or assignment title included. Poor use of white space (indentation, blank lines). Disorganized and messy. Poor use of variables (many global variables, ambiguous naming).	Includes name, date, and assignment title. White space makes program fairly easy to read. Organized work. Good use of variables (few global variables, unambiguous naming).	Includes name, date, and assignment title. Good use of white space. Organized work. Good use of variables (no global variables, unambiguous naming)	Includes name, date, and assignment title. Excellent use of white space. Creatively organized work. Excellent use of variables (no global variables, unambiguous naming).
				Clearly and

Documentation 20%	No documentation included.	Basic documentation has been completed including descriptions of all key variables. Purpose is noted for each function.	Clearly documented including descriptions of all key variables. Specific purpose is noted for each function and control structure.	effectively documented including descriptions of all key variables. Specific purpose is noted for each function, control structure, input requirements, and output.
Runtime + test cases 20%	Does not execute due to errors. User prompts are misleading or non-existent. No testing has been completed.	Executes without errors. User prompts contain little information, poor design. Some testing has been completed.	Executes without errors. User prompts are understandable, minimum use of symbols or spacing in output. Thorough testing has been completed.	Executes without errors; excellent user prompts, good use of symbols, spacing in output. Thorough and organized testing has been completed, and output from test cases is included.
Efficiency 20%	A difficult and inefficient solution.	A logical solution that is easy to follow, but it is not the most efficient.	Solution is efficient and easy to follow (i.e., no confusing tricks).	Solution is efficient, easy to understand and maintain.

Delay in assignment submission is not acceptable. If a delay is unavoidable, you should contact your tutor to discuss it. Assignments should be submitted at regular intervals, preferably as prescribed by the study schedule. (If you are unable to complete the course within your contract time, you need a formal extension of the contract. See http://calendar.athabascau.ca/undergrad/current/page06_16.php.)

TOP

Housekeeping

The 18-week schedule is designed to guide the sequence and timing of your course activities. You are not expected to follow the schedule exactly and may complete the course in less, or more, time than it suggests, as long as you stay within your course contract.

Different units may take different amounts of time to complete. Do adjust your study schedule accordingly, and consult your tutor at any time if you feel you may be getting out of step.

All SCIS Undergraduate courses are now delivered through Moodle. This interface should be fairly easy to use, but if you have any questions or concerns, please email scistech@athabascau.ca. Improvements and updates to Moodle are being done on a continuing basis. Log in to your course either through myAU or from the SCIS direct course access link:
http://scis.lms.athabascau.ca/

You will have full access to your active course in Moodle until the course end date. Afterwards, you will have partial access for 4 more months, but can no longer contact the tutor or post in the discussion forums.

Assignments are to be submitted through the upload feature at the bottom of each assignment page. Use Step 1 to attach multiple files, as well as to delete or overwrite uploaded files. Be sure you have completed uploading all files and that all edits are done before you proceed to Step 2, which is to submit the assignment. Once you hit the 'Send for Marking' button, you will no longer be able to add, delete, or edit your assignment files unless your tutor reverts it to Draft status.

Note: After you have completed the assignments, the programming profile problems, or the final exam, do not discuss their contents on any forum or post them anywhere on the Internet. Doing so will be considered cheating and will be dealt with accordingly.

TOP

Unit 1 What is Java? What is in Java? Why Java? How Java?

Overview

> The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing music or poetry.
>
> —Donald Knuth, *The Art of Computer Programming*, Vol. 1

This unit is an introduction to the world of Java programming. We will briefly discuss the Java platform and how to use an IDE, and introduce the concepts of *objects* and *classes*.

Some of the topics in this unit follow a dialogue (Q&A) format. You can imagine them as transcripts of a student—teacher discussion on the subject. The student usually (but not always!) asks the teacher questions. The teacher usually (but not always!) gives the answers.

The dialogues concentrate on the ideas and concepts, and contain many specific examples for each topic. As with any technical subject, the precise meaning of words is often important in computer science. Technical terms are not always defined. Instead, we hope their meaning becomes clear from the context and use of the term. You can also do some Internet research and/or post your questions and observations in the **COMP268 General Discussion Forum**. Be sure to give the URL for any information sources you consult.

TOP

Learning Outcomes

After completing Unit 1, you should be able to

- identify the basic issues and problems of computer programming.
- identify the Java computing platform.
- compile and run simple Java programs.
- examine the syntax for creating and using simple objects.
- use BlueJ to create Java programs.

Have fun!

TOP

Introduction to Programming Languages

::study time: 20 minutes::

JavaOne EarlyHistory from Robyn Rhodes on Vimeo.

Computers are used for solving problems quickly and accurately irrespective of the magnitude of the input. To solve a problem, a sequence of instructions is communicated to the computer. Programming languages were developed to communicate these instructions. The instructions written in a programming language comprise a *program*. A group of programs developed for specific purposes is referred to as *software* whereas the electronic components of a computer are referred to as *hardware*.

Software activates the hardware of a computer to carry out the desired task. In a computer, hardware without software is similar to a body without a brain.

Software can be *system software* or *application software*. System software is a collection of system programs. A system program is a program that is designed to operate, control, and effectively use the processing capabilities of the computer itself.

Application software is a collection of programs meant for specific applications. For example, one can conceive of applications to run financial markets, applications to predict weather, applications for office management, and so on.

Computer hardware can understand instructions only in the form of machine codes, i.e., 0’s and 1’s. A programming language used to communicate with the hardware of a computer is known as *low-level language* or *machine language*. It is very difficult to understand machine language programs because the instructions contain a sequence of 0’s and 1’s only. Also, it is difficult to identify errors in machine language programs. Low-level languages are machine dependent. To overcome the difficulties of machine languages, *high-level languages* such as BASIC, FORTRAN, PASCAL, COBOL, and C were developed.

High-level languages allow some English-like words and mathematical expressions that facilitate better understanding of the logic involved in a program. In solving problems using high-level languages, it is important to develop an *algorithm*, a step-by-step instruction to solve the problem at hand.

The algorithmic approach of yesteryear brought on very many difficulties in solving complex problems. For example, it is extremely difficult and costly to maintain a complex banking application software (say, 500,000 lines of code) written in COBOL language. Object-oriented programming languages such as C++ and Java were evolved and promoted to address these difficulties.

Object-oriented languages are high-level languages that use objects and classes, which are discussed later in this chapter.

Java is an object-oriented programming language. As part of your programs, you will create objects and instruct these objects to communicate with each other to solve the problem at hand.

What is programming?
Programming is the art and science of instructing a computer to do things.

What is an algorithm?
An algorithm describes the step-by-step process that accomplishes some task.

What is a program?
A program is an implementation of an algorithm. For example, the problem could be “add the following three numbers: 25, 88, 1100.” The algorithm then would then be

1. Take the first number 25 and add it to the second number 88.
2. Remember this total.
3. Add the remembered total to the third number.
4. Remember this new total and show it as the solution to the problem.

These four steps put together are the algorithm for the given problem.

Is it true that one problem can have just one algorithm?
Nope. One problem can be solved by many algorithms. For example, the previous problem can be solved by another algorithm as follows:

1. Take the last two numbers and add them.
2. Remember this total.
3. Take the first number and add it to the total.

4. Remember this new total and show it as the solution to the problem.

I see. Can I come up with my own algorithm such as, "Take all three numbers and add them simultaneously"?
Yes, of course. Yours is a much better algorithm.

Okay, now I know roughly what an algorithm is. What is Java?
According to Bill Joy, Sun co-founder, "Java is just a small, simple, safe, object-oriented, interpreted or dynamically optimized, byte-coded, architecture-neutral, garbage-collected, multithreaded programming language with a strongly typed exception-handling mechanism for writing distributed, dynamically extensible programs." [1]

TOP

Uh, okay. Can you simplify that a bit?
Java is a programming language. It is a high-level language, in the sense that Java programs use some English words, and it can be read by humans. It is a general-purpose language because it is not designed for just one or two programming tasks. It can be used to solve a wide variety of programming problems. All those other things that Bill Joy mentions are specific features about the Java language and platform.

What do you mean by "the Java platform"?
Along with the Java programming language itself, the Java platform includes a very large set of library functions (that is, pre-written Java code that you can use in your programs), the Java virtual machine, and various protocols, such as those used for network or web programming.

What is the Java virtual machine?
It's a computer that exists only in software. To run a Java program, your computer must have a Java virtual machine (they come with Java development environments). The use of a virtual machine is the key reason why Java programs can run on almost any computer. When you write a Java program, you don't need to worry about what computer it will run on. You just need to make it work for the Java virtual machine. We'll cover the details later in the course.

Why use Java?
Java is a good modern language with lots of interest and support. You can find Java books in almost any bookstore, so Java is obviously popular. Once you learn the basics, it's relatively easy to use.

Is Java good for web programming?
What do you mean by web programming?

You know, making websites, stuff like that.
You can use Java to create applets, which are little programs embedded in web pages. Plus, Java has a lot of support for network programming—things such as the ability to communicate information to other computers easily and securely.

What is JavaScript?
JavaScript is a web browser scripting language. It's not the same as Java, although it has some superficial similarities. We won't cover JavaScript in this course.

Who invented Java?
James Gosling and a team of researchers who worked at Sun in the early 1990s.

The Oracle site (http://www.oracle.com) is a great place to surf for Java-related resources. An overview of Java history is given at http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html. Visit this page to find out how Java came about and what it is being used for today.

How do I become a good programmer?
Practice writing lots of programs, unless you were born with the "programming gene."

Is it enough to just read a Java book and understand the examples?
It certainly helps, but reading the book isn't enough. You must learn programming by doing it.

What is Java Technology?
Visit this website: http://download.oracle.com/javase/tutorial/getStarted/intro/index.html

How to install Java?
Visit this website: http://download.oracle.com/javase/tutorial/getStarted/cupojava/index.html and install Java as prescribed. Later in this Unit, there is a Section titled "BlueJ for total beginners" to help you with installation of the BlueJ IDE (Integrated Development Environment). You can write, compile, and track Java programs from within BlueJ.

What next?
Let us move on to the next item for an introduction to how various programming languages evolved into *programming paradigms*.

---

[1]Quoted by Roger Cadenhead in Java unleashed (2nd ed.). Retrieved from http://www.webbasedprogramming.com/Java-Unleashed-Second-Edition/ch1.htm

TOP

Evolution of Programming Paradigms

::study time: 20 minutes::

Information can be computationally extracted from data (e.g., age). Data is represented by a variable or constant in a program (e.g., integer age). To perform an action, an operator acts on the data (e.g., verify age from date of birth). Data and operators are combined to form expressions (e.g., check if the person associated with the age is eligible to vote). Each instruction is written as a statement with the help of expressions (e.g., if eligible, update Elections Canada database; else, do not). A sequence of statements comprises a program (e.g., a program that checks the age of all registered voters for eligibility).

There are different languages for writing programs (e.g., Java, C, C++, C#, Lisp, Prolog, Pascal, COBOL, BASIC, and others). Programming languages evolved based on computational needs and are constrained by the availability of computational technology. At the beginning (1954–1958) the needs were simply numeric. Languages such as FORTRAN I, ALGOL 58, and FLOWMATIC were developed to cater only to numeric computations. These programming languages kept the programs independent of a central repository of data. That is, there was one global set of data accessed by one global set of programs. This is the first-ever programming paradigm: global data and monolithic programs. In this design, the programmer had to repeat the same code many times over in the same program.

Then came the notion of creating subprograms (1959–1961). Subprograms are also known as *functions* or *procedures* or *subroutines*. Subprograms avoided the duplication of code. A single functionality could be encoded in a single subprogram that could be called many times over. FORTRAN II, ALGOL 60, and COBOL are examples of programming languages that use subprograms. This second programming paradigm introduced the notion of *information hiding*, where the implementation details of a subprogram are bundled inside a named package and can be executed many times over by simply calling its name. Still, in the second programming paradigm, the data remained globally accessible.

We then advanced to the third programming paradigm (1962–1970) with languages such as Pascal and C that enabled programs to have local data in addition to global data. Thus, each subprogram could now have local data operated by instructions from a subprogram. Further, each subprogram could also have access to the globally accessible data. Each subprogram could also receive data and send data using parameters. Thus, the third programming paradigm introduced the notion of *data hiding*.

In the 1980s, the complexity involved in the developing large, real-world programs resulted in the fourth programming paradigm of objects with programming languages such as Java, C++, and so on. Here, real-world entities are captured under the notion of *objects*, and the functionalities and data associated with the entities are packaged within the object. For example, if my program deals with cars, then I'd create objects corresponding to cars, where each object encompasses data about the types of cars and also about the functionality of cars.

The contemporary programming paradigm addresses a number of new features such as *distributed/parallel programming* (e.g., two different objects can execute code simultaneously in the same computer), *autonomous programming* (e.g., an object knows, by itself, when exactly to execute a particular code), *Internet programming* (e.g., programs to access and manipulate objects over the Internet), and *ubiquitous programming* (e.g., automatic sunshades, where the code is hidden inside the real-world object).

In the near future, we can expect features such as *cloud computing, nano programming, social computing*, and *quantum computing* to become commonplace.

Cloud computing, for instance, could allow 6 billion human users to access millions of services, through thousands of service providers, over millions of servers, processing exabytes of data, delivered through terabytes of network traffic. Nano programming enables computers that employ extremely small components. Social computing deals with useful, contextual, trustworthy knowledge from collective intelligence. Quantum computing employs quantum states of atoms to represent data in computers, potentially leading to much faster computers. These new programming paradigms are evolving rapidly, and some have already reached the mainstream.

Programming is a skill that must evolve constantly. Learning to program in Java is an excellent choice, since most of the contemporary programming paradigms can be implemented in Java. Let us now get down to the business of learning Java.

TOP

A Lighter Introduction to Objects and Classes

::study time: 20 minutes::

What's an object?
A thing.

Anything?
Pretty much. Here are a few examples of objects: a chair, a person, an idea, a cloud, a number, a car, a leg, a robot, a whistle, a computer, a word, an egg yolk.... If you want to represent things like this in a Java program, you can use Java objects to do so.

Is everything an object?
That depends on your point of view. In Java, you can represent anything as an object if there is some useful reason to do so.

What's a class?
A factory for creating objects.

Do Java programmers do it with class?
Objectively speaking, yes.

Are you trying to be funny?
Did you think that was funny?

Not really.
No then, I wasn't trying to be funny. It must have been a typo.

What's a method?
A recipe for doing something. In non-object-oriented programming, methods are usually called functions.

Do objects own methods?
Yes. Yes. In Java, there is no such thing as a method or function that stands alone, not part of an object. You can't write a function without writing a class to put it in.

What about data?
What about it?

Do objects own any data?
Sure, most objects own some data, in the form of *instance variables*.

So an object is essentially a collection of variables and methods?
That's right. There are more details, of course, and some other ideas, like *inheritance*, that make objects useful, but this is the key point: **objects let you package information and algorithms together.**

What's the funniest computer joke?
While there is no consensus among experts, many programmers believe that a certain rule for access control modifiers may be the funniest thing computer science has ever discovered.

Huh?
Okay, let's go through the basic ideas you'll need in order to objectively evaluate the joke. In a Java class, every method and variable has an associated access level.

What do you mean by "access level"?

The access level of a variable or method is a restriction on what other classes are allowed to access it. For instance, a variable labelled *private* can only be accessed (that is, read from or written to) by methods in the same class. No outside class is allowed to read or write that variable. If instead you label the variable as *public*, then any class is allowed to access it.

What happens if you don't label the variable as private or public?
Then, by default, that variable (or method) is assigned *package access*. That means that the class itself can read or write the variable, and so can any other class in the same package.

TOP

What's a package?
A collection of classes. We'll talk more about packages later in the course.

So a variable or method has either private, public, or package access?
Almost. There's a fourth possibility: a variable or method can be declared as *protected*, which in Java means that any method in the class itself can use the variable or method, and so can any class that inherits from it.

What do you mean by "inherits"?
You can create new classes by extending old ones. This is called *inheritance* in object-oriented programming. We'll talk more about inheritance later in the course.

So that's it?
Yes, that's it. It's not as simple as we might like, but these are important distinctions that will come into play throughout the course.

What's the funny part?
You didn't find that funny?

Uh, no.
Good, just testing. The funny part is related to the public and private access modifiers, and a rule for remembering what they do.

Okay, I'm ready to laugh.
The rule is this: in Java, if you are an object, then only you can touch your private parts, but anyone can touch your public parts.

So that's what passes for funny in Java programming?
It's funnier in C++. In C++, there are the same private/public modifiers, but there's an additional access modifier (which Java lacks) called *friend*. In C++, it turns out that only you and your friends can touch your private parts.

I hope you haven't quit your day job.

Well, I understand your reaction. But continue reading . . . there's another really funny Java joke coming up

So you have another Java joke to tell me?
Do I ever! The best one, I think.

Right.
This one has to do with how objects are created in Java.

This isn't going to be a limerick about constructors, is it?
No, no, no. It's about how new is used to create new objects.

Okay. . .
So, in Java, you construct objects by using *new*. For example, if you have a class called Employee, then you can write this:

```
Employee e = new Employee("Bill Gates");
```

The variable *e* is a reference to an Employee object. You can access the methods and variables of the object it refers to with the "dot" notation, for example

```
e.lowerSalaryBy(5000); // lower Employee's salary
 by $5000
```

☞ The // mean that the comment following is not part of the code for the rest of the line.

The **lowerSalaryBy** method is defined in the Employee class; it takes one **int** parameter and returns nothing; that is, it has a return type of *void*.

This isn't even making me giggle.

Don't worry—the funny part is coming up! Traditionally, objects created with **new** are stored in a region of computer memory called the *heap*. In contrast, non-object variables, like **int** and **doubles**, are stored in a region of memory called the *stack*.

What's the difference between stack memory and heap memory?
Stack memory is very orderly and easy to manage, but heap memory is a constant bad-hair day. Java manages heap memory using a special program called a *garbage collector* tthat automatically deletes objects that are no longer in use. An object is considered to be no longer in use when there are no references to it. The garbage collector makes your life a lot easier. In other languages, such as C++, it is up to the programmer to manage the heap by hand explicitly. Experience shows that this is a major source of subtle programming errors.

That's swell. But I'm not laughing yet.
Okay, get ready, here comes the joke.

Okay!
Are you ready?

Yes.
Really ready?

WHAT'S THE JOKE?!
It's a riddle: Why is the heap the sexiest part of Java?

I don't know.
It's where all the newed variables are!

I have to go now.
Java jokes get no respect, no respect at all.

TOP

Objects

::study time: 20 minutes::

This section introduces you to the world of objects with simple real-world objects. Each object has a state and behaviour. Consider yourself as an object: your name, your age, your hair colour (say black), and so on would make up your state. Your activities such as 'read Google news', 'solve linear algebra problem', 'submit Assignment 1', and so on would make up your behaviour.

You should read through the contents of the following web link on objects before you proceed further, since we will be using the example of a `Bicycle` object quite a few times: http://download.oracle.com/javase/tutorial/java/concepts/object.html. Pictorially, your state information is stored in the inner circle and your behaviour information is stored in the outer circle.

Each object can add to its state and/or its behaviour. For example, you can add a new state called 'your blood type'. You can also add a new behaviour called 'participate in COMP 268 General Discussion forum'.

Each object can delete any or all of its states or behaviours.

Each object can also modify any of its states and/or behaviours. For example, you can make a state change by changing the hair colour from black to brown.

Similarly, you can also make a behaviour change in your object. Suppose the behaviour of ‘submit Assignment 1' is originally defined to be ‘submit Assignment 1 solution using Moodle'. You can change this behaviour by redefining it to be ‘submit Assignment 1 solution by emailing the instructor'. Note that in this example, the name of the behaviour ‘submit Assignment 1' remains the same, but you have changed its functionality.

TOP

Classes

::study time: 20 minutes::

Browse through the following link first before proceeding further: http://download.oracle.com/javase/tutorial/java/concepts/class.html

The link shown above introduces you to a Java class for a bicycle. It also shows an *implementation* of the class in Java language. Each object in java is created from a corresponding class. One can create many objects from a single class.

Suppose you want to create `Bicycle` objects—one bicycle for yourself, one bicycle for your younger sister, and a third bicycle for your grandpa.

As you can easily surmise, each of these three bicycles has its own states and behaviours. For instance, the ‘height of seat' of your bicycle could be 150 cm, that of your sister could be 90 cm, and that of your grandpa could be 170 cm (your grandpa is really a tall man). In this case, ‘height of seat' is a state datum.

Each of the three bicycles can have a number of state data such as ‘cadence', ‘number of gears', ‘current gear', ‘width of tires', ‘colour of frame', ‘current speed' and so on. So, each Bicycle object has the same set of states, but the values the states hold can be different.

State data of your bicycle:

cadence = 170

height of seat = 150 cm

number of gears = 24

current gear = 5

width of tires = 7 cm

colour of frame = black

current speed = 40 km/h

State data of your sister's bicycle:

cadence = 80

height of seat = 90 cm

number of gears = 12

current gear = 8

width of tires = 5 cm

colour of frame = yellow

current speed = 20 km/h

State data of your grandpa’s bicycle:

cadence = 120

height of seat = 170 cm

number of gears = 2

current gear = 2

width of tires = 12 cm

colour of frame = teal

current speed = 5 km/h

Enough about state. Let us now look at behaviour.

All three of these objects must share the same set of behaviours. Say, all of them know how to 'calculate maximum speed', 'adjust seat height', 'changeCadence', 'changeGear', 'speedUp', 'applyBrakes', and so on. If you change the functionality of one particular behaviour, it changes the corresponding functionality for all bicycles.

For example, the functionality of the 'speedUp' behaviour could be defined as 'increase the cadence by 10 more revolutions per minute', 'increase the current gear value by 1', and 'lean forward by another 20 degrees'. Now, try to think about the behaviour (in terms of a list of specific functionality) for all the other behaviours that we listed earlier.

Equipped with this knowledge about objects and behaviour, we are now ready to understand class. A class is a blueprint of all the bicycles. A *class* defines all the state data and all the behaviour functionality. From what you have read so far, we can now construct the following **`Bicycle`** class.

> Class: Bicycle
>
> State: 'cadence', 'height of seat', 'number of gears', 'current gear', 'width of tires', 'colour of frame', 'current speed'
>
> Behaviour: 'calculate maximum speed', 'adjust seat height', 'changeCadence', 'changeGear', 'speedUp', 'applyBrakes'

There you go. These three statements define the blueprint of the Bicycle class—name, state, and behaviour.

So this **`Bicycle`** class is the blueprint for all the bicycle objects you want to create—you first define a class and then instantiate the objects. The word *instantiate* in essence means *to create*. Now that you have three bicycle objects for yourself, your sister, and your grandpa, how about creating another bicycle for your friend? Yes, you can simply instantiate a fourth bicycle object using the same class blueprint.

TOP

Inheritance

::study time: 20 minutes::

Browse through the following web page for a quick introduction to inheritance using the **`Bicycle`** example: http://download.oracle.com/javase/tutorial/java/concepts/inheritance.html

Inheritance is one of the key motivations behind the development of object-oriented programming. Suppose you are asked to develop classes of various bicycles. You could start with a list of all possible types, say, common bicycle, foldable bicycle, mountain bicycle, road bicycle, electric bicycle, and tandem bicycle. These six types of bicycles have some common states and some common behaviours. Also, each of these six types of bicycle has some unique features.

One possible way to create the class blueprints for these six types of bicycles is by creating six different classes corresponding to each of the types. If you do that, first of all, you can expect plenty of overlap among these classes. For example, all six classes would share states such as 'current speed', 'current cadence', 'current gear', and so on. Further, these six classes would also share behaviour such as 'apply brakes', 'oil the chains', and so on. Secondly, if you attempt to change one of these common features in one class, whether it is a state or a behaviour, you would be forced to make similar changes in all six classes.

Do you know why it is so? Because, all common features are supposed to hold similar properties across all classes, and if you change a common feature in one class then you would have to make a similar change in the common feature in all remaining classes to maintain commonality. In a real-world project, maintaining such independent classes would become too complex.

To alleviate such maintenance complexity, inheritance could be used in designing the class blueprints. Using the same example, one could create a root/parent **`Bicycle`** class that holds all common states and behaviours. The remaining five classes (**`Foldable, Mountain, Road, Electric`**, and **`Tandem`**) could be the leaf/children bicycle classes. Having this root/leaf or parent/child relationship between classes avoids most duplicity.

TOP

Interface

::study time: 20 minutes::

Let us say that you have just created a **`Bicycle`** class and would like to inform your friends about your code. You could send the entire Java code to all your friends and invite them to look at the code to understand what your **`Bicycle`** class offers. Alternatively, you could only send them the key behaviours of your **`Bicycle`** class. If your friends want to use the **`Bicycle`** class, they could do so only in terms of the key behaviours you have published. This alternative is the preferred way of developing code. You have essentially published an *interface* to your **`Bicycle`** class.

If someone wants to create various **`Bicycle`** objects and put them to use in their code, they do so using only the published behaviours. The interface is a contract between your **`Bicycle`** class and anyone from the rest of the world who wants to use your **`Bicycle`** class. The contract specifies exactly what you offer in the **`Bicycle`** class and what someone else can do with your **`Bicycle`** class.

Think about what happens when someone wants to add a completely new behaviour to your **`Bicycle`** class. If that person has full access to your **`Bicycle`** class code, then he or she would be able to add that new behaviour. In doing so, he/she would have created a slightly different `Bicycle` class. Alternatively, that person could ask you to include this new behaviour in your `Bicycle` class and publish it in the interface, thus maintaining a single `Bicycle` class for everyone to use.

Browse through the following web page to consolidate the notion of an interface with the `Bicycle` example: http://download.oracle.com/javase/tutorial/java/concepts/interface.html

TOP

BlueJ for Total Beginners

::source: http://www.bluej.org/::

::study time: 60 minutes::

Now what?
Now try running Java.

How do I do that?
That's a bit involved. In this course, we'll be mostly using the BlueJ integrated development environment (IDE). It's essentially a supercharged editor designed to make programming easier. But there are many little details that we need to attend to before running even the simplest program. First, you must install BlueJ onto your computer. You can download BlueJ for Java from http://www.bluej.org/download/download.html. Installation instructions are available from http://www.bluej.org/download/install.html.

Can you show me how you did it?
Sure, click here for a tutorial on using BlueJ to create and run a Java program.

How to write Java code in BlueJ?
Go to this website: http://www.bluej.org/tutorial/tutorial-201.pdf It will get you started. Take your time to go through this in detail, and get used to BlueJ. After installing BlueJ, you may want to install an extension called MILE.

What is MILE?
MILE is an extension to BlueJ offered exclusively to AU students. It allows you to track your own coding habits and share them with your tutor.

Is MILE a requirement?
Yes and no. With MILE, it would be easy to discuss your programming problems with your tutor. However, it you are not keen to track your coding habits then you can simply ignore MILE.

Is there a download site for MILE?
You can download the MILE extension from http://mile.athabascau.ca.

Then what?
Copy the ‘MILE_BlueJ_Tracer_Extension.jar’ file from this website to the BlueJ installation directory under /lib/extensions, or into an extensions directory specified in your BlueJ user configuration directory (/.bluej/extensions or \bluej\extensions), or into an extensions directory in the single BlueJ project you wish the extension to be active for.

The extension should start when you next run BlueJ after installing the extension or when you open the BlueJ project into which you installed the extension. Every time BlueJ runs after installing the extension, it will ask you if you want to use the extension. The choices are Yes and No. The first time the extension runs, it asks for Student id #, Assignment id #, and Path to where the MTD files will be saved. These values can be changed in Tools -> Preferences -> Extensions. The Assignment ID can only be changed in the preferences for future assignments.

MTD files are XML files that record your personal coding process. MILE records time-stamps and snapshots of code submitted for BlueJ compilation. It may also store statistics on errors and warnings, if any.

Is there a MILE file for each program?
Yes, there is a file with an .mtd extension for each program you write in BlueJ.

Can I include the .mtd file along with my program submission?
That would be ideal. You can also include the .mtd file in your programming profile.

What next?
Take a well-deserved break and then move ahead to the next item when ready.

TOP

Hello, World!

::study time: 20 minutes::

What’s “Hello, World!”?
Traditionally, this is the first program programmers write. A Hello, world! program is a program that does nothing but print “Hello, world!” on the computer screen.

How do I write a Hello, world! program?
See http://download.oracle.com/javase/tutorial/getStarted/cupojava/index.html, which shows

a basic Java Hello world! program. Download one that runs on your system before continuing! Make sure to download the file that has the .java extension.

I'm having trouble getting it to work. What should I do?
Play with things to find out how they work. Pay attention to the details: Java does not tolerate spelling errors in variable names, and it is case sensitive, so `main` is different from `Main`. Watch out for missing semicolons at the end of lines. Every Java statement must end with a semicolon (;).

I still can't get it to work!
There's no magic solution. You need to find out what part of the process of compiling and running the program is not working for you. Be a detective, and gather clues about why things are not working the way you think they should. Work through the whole process, step by step, and be careful not to make unwarranted assumptions. Or, talk to your tutor.

I still have problems—help please!
Visit this webpage:
http://download.oracle.com/javase/tutorial/getStarted/problems/index.html

I got it working
Excellent. There you go...

There are a few who still couldn't get it working. What about them?
Well, they would have contacted their tutor by now. No worries.

What more is there?
How about trying the following programs just for the fun of it, in BlueJ !!

http://www.cis.temple.edu/~ingargio/cis67/code/Hello.java

http://www.cs.uic.edu/~sloan/CLASSES/java/HelloDate.java

http://www.cis.temple.edu/~ingargio/cis67/code/Helloi.java

http://www.cis.temple.edu/~ingargio/cis67/code/Hellox.java

http://www.cis.temple.edu/~ingargio/cis67/code/Helloix.java

TOP

A Quick Introduction to String Objects and String Methods

::study time: 20 minutes::

This animation highlights string objects and methods used by string objects. Here we deal with two string objects and a number of string methods that are employed on these two objects.

To start the animation, first type in two strings. These two strings correspond to two **`String`** class objects named **`string1`** and **`string2`**. They can be identical or different. When you press Start, a two-column table will be displayed.

Each cell in the table corresponds to a method (behaviour) in the **`String`** class. For instance, the top left cell corresponds to a method called 'equals'. This 'equals' method in the **`String`** class tests if the first string object (**`string1`**) is similar to the second string object (**`string2`**). The expression is as follows:

```
string1.equals (string2)
```

This is of the format **`firstStringObject.methodName (secondStringObject)`**.

If you click on this cell, the result will be shown at the bottom of the animation. If the two string objects are similar then you will get TRUE; else FALSE.

Click on each cell in the table and see the results.

Try this animation a few times with different strings each time. Include spaces and other non-alphabetic characters.

TOP

Concluding Unit 1

You have done extremely well to come this far. Keep it going.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming Profile entry for Unit 1. The icon on the left will help to remind you to update it.

You may also want to make notes on the definitions of the italicized words from Unit 1.

Assessment Activities

Answer the questions in Assignment 1 that belong to Unit 1, and update your programming profile.

TOP

Unit 2 Variables, Operators, and Expressions in Java

Overview

This unit introduces Java's basic built-in data types for numbers and strings. It covers the important concept of programming variables, and shows you how to use Java as a kind of super desktop calculator.

Numbers are an important component of computing. To see an example of what happens if we take things easy when it comes to computing with numbers, visit http://en.wikipedia.org/wiki/Ariane_5_Flight_501 for a quick report on the Ariane 5 rocket failure.

TOP

Learning Outcomes

After completing Unit 2, you should be able to

- create, initialize, and use variables.
- use basic numeric types.
- perform basic arithmetic calculations using Java expressions.
- use the Number class.
- use strings, including substrings and individual character access.
- use code to read values from the console and print values to the console.
- use operators and expressions.

TOP

Variables

::study time: 30 minutes::

A *variable* is a named placeholder to contain specific items.

Let's say that a fridge is a placeholder for food items. A cup is a placeholder for drinks. A bookcase is a placeholder for books. There are a number of fridges, cups, and bookcases in existence.

We need to assign specific names so that you know which placeholders you are referring to. So, you can now say that a fridge named 'mint' is a placeholder for food items; a cup named 'pepper' is a placeholder for drinks; a bookcase named 'cinnamon' is a placeholder for books. Suppose you have placed a book, say Dan Brown's *The Lost Symbol*, in the bookcase; then, to access the book you must access 'cinnamon'. Suppose you have placed lemonade in a cup; then to drink the lemonade you access 'pepper'. Suppose someone had put the milk container in the fridge; if you need milk you have to access 'mint'.

We also need to know exactly what these placeholders can contain. A fridge can contain anything, not just food items. But you are forcing 'mint' to store only food items. Similarly, you are restricting 'pepper' to store only drinks and 'cinnamon' only books. So 'food items', 'drinks', and 'books' can be well-defined objects.

One can store not only objects, but also simpler items such as numbers or strings. For example, you can create a variable called 'age' and restrict it to only store numbers between 0 to 125. Another variable called 'my_name' can be limited to store only 'Vive(k) Kumar'.

Thus, a variable is **a named placeholder to contain objects or simple data**.

Read the following web page for a detailed introduction to variables and how to name them: http://download.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

Variables in Java can take only data that are of specific types—mostly! We'll see exceptions for this rule at a later time. For the time being, an integer variable can hold only integer values; a string variable can hold only a string value; and a `Bicycle` variable can hold a `Bicycle` object.

Java has eight primitive types of data: **byte, short, int, long, float, double, boolean**, and **char**. You can assign a variable to belong to one of these types. For instance, `int age;` declares a variable named 'age' to be of type 'int'. You can assign an integer value to this variable as follows: `age = 33;`.

You can also assign a value to a variable when the variable is declared. For example, `double pi = 3.14;`. Here, 'double' is the data type, 'pi' is the name of the variable, '=' is the assignment operator, and '3.14' is the value assigned to the variable.

TOP

If you don’t assign a value to a declared variable, then such variables are assigned default values. Visit http://download.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html for a complete list of default values for these eight primitive data types. Remember, primitive data types are NOT objects :). As part of your programming profile activity, you may wish to keep track of the data types that you have not used in your programs yet.

One can also assign specific literal values to variables, as in `char capitalC = 'C';` or `boolean worldIsFlat = false;`.

Shown below is a demonstration of how variables are declared and stored in Java. The area on the right represents your computer’s memory (RAM). Every time you click a button on the left, a new variable is declared and placed somewhere in memory. Note that a string consists of two parts: a reference and the actual string itself.

Click on each of the primitive data types listed on the left pane. The animation will show you where in memory the name of the variable is stored along with the value assigned to it.

That is enough about variables for now. We'll get to know more about variables as we go further.

TOP

Java Arithmetic Example: Voice Data

::study time : 30 minutes::

Let us use variables in a simple java application called "VoiceExample.java."

When you speak into a telephone, your voice gets sampled 8000 times a second, and each sample is one of 256 possible different levels. Essentially, one second of the sound of your voice is converted into a sequence of 8000 bytes, and those bytes are sent through cables and wires to the telephone of the person you are speaking to. Amazingly, they usually get sent so quickly that there are no noticeable pauses or delays—in sharp contrast to delay-filled Internet communications!

Let’s write a Java program that will answer this question: How many kilobytes does s seconds of telephone voice communication require? The user types in a number of seconds, and then the program prints out how many bytes are required by that many seconds of voice.

Before writing the program, it’s a wise idea to work out the details by hand. We know that 1 second of voice needs 8000 bytes, so we can write that as a simple equation: 1s = 8000 bytes. If we want to know how many bytes 10 seconds requires, then we just multiply both sides by 10, giving 10s = 80,000 bytes. This is useful, but it is not quite the answer we want, because we need to know the number of kilobytes, not just bytes.

A kilobyte is 1024 bytes; that is, 1 kb = 1024 bytes. Given some number of bytes, we just divide by 1024 to find how many kilobytes that is. For example, 8000 bytes is 8000/1024 = 7.8125 kb.

With this in mind, let’s sketch a Java program that solves the original problem. We want the user to type in a number of seconds, and then the program will display how many kilobytes that many seconds of voice take up. The steps are as follows:

1. Get the number of seconds, *s*, from the user.
2. Using the formulas above, calculate the number of kilobytes that *s* seconds of voice requires.
3. Print the result on the screen.

Sketching out a solution like this is not really necessary for such a simple problem, but it’s a good habit to get into because this initial design is one of the most important steps in writing longer programs.

Now let’s write a Java program that solves the problem. First, write a shell program like this:

```java
public class VoiceExample {
        public static void main(String[] args) {
                System.out.println("Doesn't do anything yet!");
        } // end of main method
} // end of class VoiceExample
```

The `main` method here is an example of a stub. Stubs are quite useful, and you should get into the habit of using them. As you can see, all it does is print the message “Doesn't do anything yet!” to the screen. But the code around it is correct, so you can concentrate on writing main’s body.

For the next step, we can replace this message statement with print statements that print out the steps of the program outlined above.

```
public class VoiceExample {
        public static void main(String[] args) {
                System.out.println("Using the formulas above, calculate the number of

kilobytes that s seconds of voice requires");
                System.out.println("Get the number of seconds, s, from the user.");


                System.out.println("Print the result on the screen. ");


        } // end of main method
} // end of class VoiceExample
```

Now, we can go a step at a time, replacing each print statement with its corresponding code.

Note that the original print statement has been replaced by a comment indicating the purpose of that first section of code.

Now let’s write the code corresponding to each of the three comments:

```
importjava.io*;
public class VoiceExample {
        public static void main(String[] args) {
                // first comment
                // Get the number of seconds from the user.
                // Create a ConsoleReader object to simplify input.
                ConsoleReader console = new ConsoleReader(System.in);
                System.out.print("Please enter a number of seconds: ");
                double sec = console.readDouble();

                // second comment
                // Calculate number of kilobytes.
                final int KBYTE = 1024;
                // 1 kb = 1024 bytes
                final int VOICE_BYTES_PER_SECOND = 8000;
                final int VOICE_KB_PER_SECOND = VOICE_BYTES_PER_SECOND / KBYTE;
                double voiceKB = sec / VOICE_KB_PER_SECOND;

                // third comment
                // Print the result on the screen
                System.out.println();
        }

} // end of class VoiceExample
```

TOP

The code “import java.io.*;” is required so that VoiceExample.java can have access to all the Java classes from the “io” library. ‘println’ is one of the methods defined in the “io” library that you can readily use.

Notice that the code uses a class called “ConsoleReader”. This is not a pre-built Class in Java. But, this class (i.e., the Java source code) was written by someone and published at http://www.cs.indiana.edu/classes/a202-oley/ConsoleReader.java. You can copy this file from this website under the name “ConsoleReader.java” and keep it available in the same directory where “VoiceExample.java” resides. The ConsoleReader class simplifies reading strings and numbers input by the user.

The source of ConsoleReader is presented below:

```
/* ConsoleReader.java - use to read input from the keyboard.
 * Examples of use:
 * First, declare and initialize a ConsoleReader object:
 * ConsoleReader console = new ConsoleReader(System.in);
 * Next, use the desired method:
 * console.readLine() // returns a String
 * console.readInt() // returns an int
 * console.readDouble() // returns a double
 */
import java.io.InputStreamReader;
import java.io.InputStream;
import java.io.BufferedReader;
import java.io.IOException;

public class ConsoleReader {
        private BufferedReader reader;

        public ConsoleReader(InputStream inStream) {
                reader = new BufferedReader(new InputStreamReader(inStream));
        }

        public String readLine() {
                String inputLine = "";
                try {
                        inputLine = reader.readLine();
                } catch (IOException e) {
                        System.out.println(e);
                        System.exit(1);
                }
                return inputLine;
        }
        public int readInt() {
                String inputString = readLine();
                int n = Integer.parseInt(inputString);
                return n;
        }

        public double readDouble() {
                String inputString = readLine();
                double x = Double.parseDouble(inputString);
                return x;
        }
}
```

Notice that, in VoiceExample.java, the second two constants have very long names: **`VOICE_BYTES_PER_SECOND and VOICE_KB_PER_SECOND`**. It can be irritating to type such long names, but the advantage is that these are *self-documenting constants*. Everywhere you use them, it should be clear to a reader what their meaning is. Many of the standard Java classes use similarly long names in an effort to make programs more readable.

However, it doesn't always make sense to use long, self-descriptive variable names. Sometimes short names are better. For instance, if you are writing a program to solve the quadratic equation x^2 + 4x - 4 = 0, then it makes more sense to use the variable x in your program, as opposed to something like **`unknownValue`** or **`theVariableWhoseValueWeAreTryingToFind`**.

In practice, we wouldn't bother writing three constants like this to solve such a short and simple problem. But this program is meant to illustrate good programming style and practice. When you write longer programs, naming and collecting all the important constants like this can save you a lot of work, prevent errors, and make your program easier to read and understand. For example, if the phone companies decide to sample voice 10,000 times a second, then you only need to change one value in your program if you've been careful to use constants everywhere.

Here is a complete VoiceExample.java program. As you can see, it includes another import statement that uses a class named "NumberFormat." There are many such libraries with pre-existing classes available for your use. You will learn more about these libraries as you learn more about Java. The **`NumberFormatclass`** is used to limit to two the number of values to the right of the decimal place that are displayed. Most users will not be interested in the answer beyond one or two decimal places.

`java.io.*` implies that you are interested in using all classes, hence the '*', available under the library of classes java.io.

`java.text.NumberFormat` implies that you are interested in using only the class named 'NumberFormat', available under the library of classes **`java.text`**.

```
import java.io.*;
import java.text.NumberFormat;

public class VoiceExample {
        public static void main(String[] args) {
                // first comment
                // Get the number of seconds from the user.
                // Create a ConsoleReader object to simplify input.
                ConsoleReader console = new ConsoleReader(System.in);
                System.out.print("Please enter a number of seconds: ");
                double sec = console.readDouble();

                // second comment
                // Calculate the number of kilobytes.
                final int KBYTE = 1024; // 1 kb = 1024 bytes
                final int VOICE_BYTES_PER_SECOND = 8000;
                final int VOICE_KB_PER_SECOND = 8000 / KBYTE;
                double voiceKB = sec / VOICE_KB_PER_SECOND;

                // third comment
                // Print the result on the screen.
                NumberFormat formatter = NumberFormat.getNumberInstance();
                formatter.setMaximumFractionDigits(2);
                System.out.println("\n\n" + formatter.format(sec)
                                   + " seconds of voice " + "requires "
                                   + formatter.format(voiceKB) + "kb of " + "storage\n");
        }
} // end of class VoiceExample
```

The final program shown above is an example of a good program. It is easy to read, self-documenting, and user-friendly, as opposed to the following program:

```
import java.io.*;
public class VoiceExample {
        public static void main(String[] args) {
                System.out.println(new ConsoleReader(System.in).readDouble() / 0.128);
        }
}
```

This program works, and is quite short, but its purpose is not at all clear. It’s hard to read because nothing is explained. It’s hard to use because there are no friendly prompts explaining the input and output to the user. Even though this performs the same calculation as the previous program, and maybe it even runs a little faster, it is clearly a program of much lower quality.

TOP

Operators

::study time: 140 minutes::

Operators enable specific operations on operands. In the expression, 2 + 3, ‘2’ and ‘3’ are operands and '+' is the operator.

Read this web page for a summary of all Java operators: http://download.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html

Operator Precedence

Let us try to evaluate this expression, 10 − 4 * 2. Would it be (10 − 4) * 2 or 10 − (4 * 2)? The first option gives you a value of 12, and the second 2. To have consistent evaluation, Java enforces an order of precedence. For example, in the example above, * has a higher precedence than −, and hence Java would always evaluate this expression as 10 − (4 * 2).

Read this page for an introduction to operator precedence: http://download.oracle.com/javase/tutorial/java/nutsandbolts/operators.html.

Here is an example program to illustrate the precedence of operators:

```
class TestPrecedence {

        public static void main(String[] args) {

                System.out.println(" 5 + 3 * 2 = \t " + (5 + 3 * 2));

                System.out.println("(5 + 3) * 2 = \t " + ((5 + 3) * 2));



                int a;

                int b = 5;

                int c = 1;

                a = b = c; // output is 1




                System.out.println(" 5 - 2 + 1 = \t " + (5 - 2 + 1));

                System.out.println(" a = b = c: \t " + a);

        }

}
```

Some sample expressions using the operators are given below. You can run each of these examples by saving them in respective files.

Prefix and Postfix Operators Examples

Postfix operators work on single operands. The operand is evaluated first in its current context and then incremented. In the following example, when printing a++, a’s current value of 5 is printed, and then its value is incremented so that it prints 6 when a’s value is printed the next time.

```
class PostfixOperators {

        public static void main(String[] args) {

                int a = 5;

                System.out.println("now a = " + a++); // at this time, a = 5; a is

                // incremented after it is evaluated to be 5

                System.out.println("then a =" + a);  // at this time, a = 6

        }

}
```

::source:http://www.janeg.ca/scjp/oper/prefix.html::

```
class PrefixAndPostfixOperators {

        public static void main(String[] args) {

                int x = 0;

                int y = 0;

                 // A compile error occurs if used directly with a value,

                // e.g., ++5;.

                // ++x offers the same result as x = x + 1.
```

```
                  ++x; // prefix

                  System.out.println("x=0; ++x; \t\t\t -> " + x);



                  x = 0;

                  x++; // postfix

                  System.out.println("x=0; x++; \t\t\t -> " + x);

                   // Same result as x = x – 1.

                  x = 0;

                  --x;  // prefix

                  System.out.println("x=0; --x; \t\t\t -> " + x);

                  x = 0;

                  x--;  // postfix

                  System.out.println("x=0; x--; \t\t\t -> " + x);

                  x = 0;

                  y = 0;

                  y = ++x;  // prefix

                  System.out.println("x=0; y=0; y = ++x; \t\t -> " + "y = " + y +

                  " x = " + x);

                  x = 0;

                  y = 0;
```

```
y = x++;  // postfix

System.out.println("x=0; y=0; y = x++; \t\t -> " + "y = " + y +

" x = " + x);

// did you anticipate these results?

x = 0;

x = ++x; // prefix

System.out.println("x=0; x = ++x; \t\t\t -> " + x

                + "[Value of x changes]");

x = 0;

x = x++; // postfix

System.out.println("x=0; x = x++; \t\t\t -> " + x

                + "[Value of x does not change]");

// Test on char type to see if prefix and postfix operators

// could be used on char datatypechar c = 'a';

c++;

System.out.println("char c = 'a'; c++; \t\t -> " + c);

--c;

System.out.println("char c = 'b'; --c; \t\t -> " + c);

// prefix and postfix test on bytes:byte b = 2;

byte b1;
```

```
                System.out.println("byte b = 2; ++b; \t\t -> " + ++b);

                b1 = ++b;

                System.out.println("byte b1 = 3; b1 = ++b; \t\t -> " + b1);

                b = 127;

                b1 = ++b;

                System.out.println("byte b1 = 127; b1 = ++b; \t -> " + b1);

                b = -128;

                b1 = --b;

                System.out.println("byte b1 = -128; b1 = --b; \t -> " + b1);

        }

}
```

TOP

Assignment, Arithmetic, and Unary Operators

Read this web page for a detailed introduction to assignment, arithmetic, and unary operators: http://download.oracle.com/javase/tutorial/java/nutsandbolts/op1.html. Implement all the programs introduced in this page as part of your Programming Profile.

Assignment Operators Example

There are 12 assignment operators:

=	*=
/=	%=
+=	−=
<<=	>>=
>>>=	&=
^=	|=

All operators, except = , are *right associative*: a = b = c is grouped as a = (b = c).

All operators operate only on primitive data types except = and += , which can be used with `String` objects.

Examples:

x += y is equivalent to x = x + y

x %= y is equivalent to x = x % y

x |= y is equivalent to x = x | y

Browse through http://www.janeg.ca/scjp/oper/assignment.html for a quick introduction to assignment operators. Include the example program “TestAssignment.java” available at the end of the page in your programming profile.

Arithmetic Operators Example

The arithmetic operators are as follows:

+ additive operator (also used for String concatenation)

− subtraction operator

* multiplication operator

/ division operator

% remainder operator

The + and − have the same precedence and are *left associative*. They operate only on primitive numeric types, except when the + operator is used to concatenate `String` objects.

The * , / , and % (called *modulo*) have the same precedence and are left associative as well. They operate strictly on primitive numeric types.

Integer division rounds towards 0 by truncating the result.

Division by 0 in integers, such as 10 / 0, results in an exception (error). On the other hand, division by 0 in variables of the float data type yields the following:

- division of a positive floating-point value: POSITIVE_INFINITY
- division of a negative floating-point value: NEGATIVE_INFINITY
- division of a floating-point value by −0: POSITIVE_INFINITY

```
 10.34 / 0 // Result: infinity

-10.34 / 0 // Result: -infinity

 10.34 / -0 // Result: infinity

 0 / 0 // Result: not a number (NaN)
```

The modulo operator (%) returns the remainder or the fraction. This operator can be used by both integers and floats:

- The result is negative if the dividend is negative.
- The result is positive if the dividend is positive.
- If the divisor is zero, a runtime exception is thrown.
- If the dividend is a floating-point value and the divisor is zero, no exception is thrown and the result is not a number.

```
 5 % 3 = 2

-5 % 3 = -2

 5.0 % 3 = 2.0

-5.0 % 3 = -2.0

 5.0 % 0 = NaN
```

Take a quick look at this web page for some more examples: http://www.janeg.ca/scjp/oper/arithmetic.html. Don’t forget to include the example program “TestArithmetic.java” available at the end of the page in your programming profile.

Unary Operators Example

Unary operators also work on only a single operand, and are *prefixed*. For example, ++a would increment the value of a first before it is evaluated. The ! unary operator works only on operands of the Boolean type and simply inverts the Boolean value. The ~ unary operator works only on binary operands.

::source:http://www.janeg.ca/scjp/oper/unary.html.::

The ~ bitwise complement is used only on integers. It inverts the bits. That is, a 0-bit becomes a 1-bit, and vice-versa. In all cases, ~x = (−x) −1:

```
byte b0 = 7; // Binary: 0000 0111byte b1 = ~b0; // Binary: 1111 1000 (-8)

~7 = -7 -1 = -8


~3578 = -3578-1 = -3579


~-1234 = -(-1234)-1 = 1233
```

The ! operator returns the logical complement of a Boolean type:

```
!(false) = true; // Complement of 'false' is 'true'.


!(true) = false; // Complement of 'true' is 'false'.
```

The result of the unary plus (+) operator is a value not a variable:

```
byte b = +5; // Result: 5
```

The unary plus operator has no effect on the sign of a value; it is included for symmetry only and to allow the declaration of constants.

The result of the unary minus (−) operator is a value that is the same as subtraction from zero; −x is equal to (~x) + 1:

```
byte b;


b = -5;  // Result: -5
```

```
b = (~5) + 1;  // Result: -5
```

Negation of the maximum negative `int` or `long` value results in the same number. An overflow occurs, but no exception is thrown:

```
int i;

 long l = 0L;

i = -(-2147483648); // Result: -2147483648

l = -(-9223372036854775808L) // Result: -9223372036854775808;
```

TOP

For variables of float datatypes, the unary minus operator merely negates the sign of the value:

```
double d = 0D;

 d = -(15.63); // Result: -15.63

 d = -(-15.63); // Result: 15.63 class UnaryOperators {

        public static void main(String[] args) {

                /** Variables **************************************************/byte b;

                byte b1;

                int i;

                long l = 0L;

                float f = 0F;

                double d = 0D;



                /** Unary ~ ****************************************************/
                b = 7;

                System.out.println("b = 7; ~b = \t\t\t -> " + (~b));



                /** Unary ! ************************************************/

                System.out.println("!(false) = \t\t\t -> " + !(false));

                System.out.println("!(true) = \t\t\t -> " + !(true));



                /** Unary + ************************************************/

                b = +5;

                System.out.println("b = +5 \t\t\t\t -> " + b);



                /** Unary - ************************************************/

                b = -5;

                System.out.println("b = -5 \t\t\t\t -> " + b);

                b = (~5) + 1;

                System.out.println("b = (~5)+1 \t\t\t -> " + b);

                i = -(-2147483648);

                System.out.println("i = -(-2147483648) \t\t -> " + i);

                i = (~i) + 1;

                System.out.println("i = (~i) + 1 \t\t\t -> " + i);
                l = -(-9223372036854775808L);

                System.out.println("l = -(-9223372036854775808L) \t -> " + l);

                l = (~l) + 1;

                System.out.println("l = (~l) + 1 \t\t\t -> " + l);

                d = -(15.63);

                System.out.println("d = -(15.63) \t\t\t -> " + d);

                d = -(-15.63);

                System.out.println("d = -(-15.63) \t\t\t -> " + d);

        }

}
```

TOP

Equality, Relational, and Conditional Operators

Read the following web pages for a detailed introduction to equality, relational, and conditional operators: http://download.oracle.com/javase/tutorial/java/nutsandbolts/op2.html and http://www.janeg.ca/scjp/oper/comparison.html. Implement all the programs introduced in these pages as part of your Programming Profile.

Comparison Operators Example

```
class ComparisonDemo1 {

        public static void main(String[] args) {

                int value1 = 1;

                int value2 = 2;
                if (value1 == value2)

                        System.out.println("value1 == value2");

                if (value1 != value2)

                        System.out.println("value1 != value2");

                if (value1 > value2)

                        System.out.println("value1 > value2");

                if (value1 < value2)

                        System.out.println("value1 < value2");

                if (value1 <= value2)

                        System.out.println("value1 <= value2");

        }

}



 class ComparisonDemo2 {

        public static void main(String[] args) {

                int x = 5;

                int y = 6;

                int z = 5;

                float f0 = 0.0F;

                float f1 = -0.0F;
                float f2 = 5.0F;

                String s1 = "hello";

                String s2 = "hello";

                String s3 = s1;

                String s4 = new String("hello");



                /** Relational *********************************************/

                System.out.println();

                System.out.println("Relational operators: ");

                System.out.println("----------------------");

                System.out.println();// Insert a blank line for neatness.

                System.out.println("\t Less than: 5 < 6 \t\t " + (x < y));

                System.out.println("\t Less than or equal to: 5 <= 5 \t\t "

                                + (x <= z));

                System.out.println("\t Greater than: 5 > 6 \t\t " + (x > y));

                System.out.println("\t Greater than or equal to: 5 >= 5 \t\t "

                                + (x >= z));

                System.out.println();

                System.out.println("\t Less than: -0.0 < 0.0 \t\t " + (f1 < f0));

                System.out.println("\t Less than or equal to: -0.0 <= 0.0 \t\t "
                            + (f1 <= f0));

                System.out.println("\t Greater than: 5 > NaN \t\t "

                            + (x > (f0 / f1)));



                /** Equality ***********************************************/

                System.out.println();

                System.out.println("Equality operators: ");

                System.out.println("-------------------");

                System.out.println();

                System.out.println("\t Equals: 5 == 5.0 \t\t " + (x == f2));

                System.out.println("\t Not Equal: 5 != 5.0 \t\t " + (x != f2));

                System.out.println("\t Equals: s1 == s2 \t\t " + (s1 == s2)

                            + " [same literal]");

                System.out.println("\t Equals: s1 == s3 \t\t " + (s1 == s3)

                            + " [same object reference]");

                System.out.println("\t Equals: s1 == s4 \t\t " + (s1 == s4)

                            + " [s4 is new object]");

            }

    }
```

Unary Operators Example

Unary operators also work on only a single operand, and are *prefixed*. For example, ++a would increment the value of a first before it is evaluated. The ! unary operator works only on operands of the Boolean type and simply inverts the Boolean value. The ~ unary operator works only on binary operands.

::source:http://www.janeg.ca/scjp/oper/unary.html. ::

The ~ bitwise complement is used only on integers. It inverts the bits. That is, a 0-bit becomes a 1-bit, and vice-versa. In all cases, ~x = (-x) -1:

```
byte b0 = 7; // Binary: 0000 0111byte b1 = ~b0; // Binary: 1111 1000 (-8)
```

```
~7 = -7 -1 = -8

~3578 = -3578-1 = -3579

~-1234 = -(-1234)-1 = 1233
```

The ! operator returns the logical complement of a Boolean type:

```
!(false) = true; // Complement of 'false' is 'true'.

!(true) = false; // Complement of 'true' is 'false'.
```

The result of the unary plus (+) operator is a value not a variable:

```
byte b = +5; // Result: 5
```

The unary plus operator has no effect on the sign of a value; it is included for symmetry only and to allow the declaration of constants.

The result of the unary minus (-) operator is a value that is the same as subtraction from zero; -x is equal to (~x) + 1:

```
byte b;

b = -5;  // Result: -5

b = (~5) + 1;  // Result: -5
```

Negation of the maximum negative `int` or `long` value results in the same number. An overflow occurs, but no exception is thrown:

```
int i;

 long l = 0L;

i = -(-2147483648); // Result: -2147483648

l = -(-9223372036854775808L) // Result: -9223372036854775808;
```

TOP

For variables of float datatypes, the unary minus operator merely negates the sign of the value:

```
double d = 0D;

 d = -(15.63); // Result: -15.63

 d = -(-15.63); // Result: 15.63 class UnaryOperators {

        public static void main(String[] args) {

                /** Variables *************************************************/byte b;

                byte b1;

                int i;

                long l = 0L;

                float f = 0F;

                double d = 0D;



                /** Unary ~ ****************************************************/

                b = 7;

                System.out.println("b = 7; ~b = \t\t\t -> " + (~b));
```

```
/** Unary ! **************************************************/

System.out.println("!(false) = \t\t\t -> " + !(false));

System.out.println("!(true) = \t\t\t -> " + !(true));



/** Unary + **************************************************/

b = +5;

System.out.println("b = +5 \t\t\t\t -> " + b);



/** Unary - **************************************************/

b = -5;

System.out.println("b = -5 \t\t\t\t -> " + b);

b = (~5) + 1;

System.out.println("b = (~5)+1 \t\t\t -> " + b);

i = -(-2147483648);

System.out.println("i = -(-2147483648) \t\t -> " + i);

i = (~i) + 1;

System.out.println("i = (~i) + 1 \t\t\t -> " + i);

l = -(-9223372036854775808L);

System.out.println("l = -(-9223372036854775808L) \t -> " + l);
```

```
                l = (~l) + 1;

                System.out.println("l = (~l) + 1 \t\t\t -> " + l);

                d = -(15.63);

                System.out.println("d = -(15.63) \t\t\t -> " + d);

                d = -(-15.63);

                System.out.println("d = -(-15.63) \t\t\t -> " + d);

        }

}
```

TOP

Bitwise and Bit Shift Operators

::study time: 60 minutes::

I presume that you already know about bits in computing. Inside the computer everything is represented in binary digits, aka bits – 0 and 1. For example, 1001 could represent the number 9. Similarly, characters, strings, objects, and everything else represented inside the computer's memory is represented in terms of bits.

First, visit http://www.janeg.ca/scjp/oper/binhex.html for a review of decimal, binary, octet, and hex representations.

Browse through this page for a quick introduction to bitwise and bit shift operators: http://download.oracle.com/javase/tutorial/java/nutsandbolts/op3.html. Also include BitDemo.java from this page in your programming profile.

As mentioned earlier, there are three bitwise operators: & AND, | OR, ^ exclusive OR. These are used for operations on integer and Boolean values (logical bitwise). They return 1 if the corresponding bits in both operands have a 1, otherwise they return a 0. For example, assuming a 32-bit machine (each number is represented by 32 bits), using the

AND operator on 63 and 252 results in a value of 60.

```
63 = 00000000 00000000 00000000 00111111

252 = 00000000 00000000 00000000 11111100

------------------------------------

00000000 00000000 00000000 00111100 --> 60
```

The OR operator returns a 0 if the corresponding bits in both operands are 0, otherwise it returns a 1. This OR is also known as *inclusive OR*.

```
63 = 00000000 00000000 00000000 00111111

252 = 00000000 00000000 00000000 11111100

------------------------------------

00000000 00000000 00000000 11111111 --> 255
```

TOP

The XOR operator returns a 0 if the corresponding bits of both operands are both 0 or both 1; otherwise it returns a 1.

```
63 = 00000000 00000000 00000000 00111111

252 = 00000000 00000000 00000000 11111100

------------------------------------

00000000 00000000 00000000 11000011 --> 195
```

```
class TestBitwise {

        public static void main(String[] args) {

                System.out.println("63 & 252 \t\t -> " + (63 & 252));

                System.out.println("63 | 252 \t\t -> " + (63 | 252));

                System.out.println("63 ^ 252 \t\t -> " + (63 ^ 252));

        }

}
```

The unary bitwise ~ operator is only used with integer values. It inverts the bits, i.e., a 0-bit becomes 1-bit and vice-versa. In all cases, ~x equals (−x)−1:

```
     byte b0 = 7; // binary: 0000 0111byte b1 = ~b0; // binary: 1111 1000 (-8)




     ~7 = -7 -1 = -8

     ~3578 = -3578-1 = -3579

     ~-1234 = -(-1234)-1 = 1233
```

TOP

Expressions, Statements, and Blocks

::study time: 30 minutes::

Visit the following page for an introduction to expressions, statements, and blocks in Java. Include the sample programs from this page in your Programming Profile.

::source:http://download.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html::

TOP

Numbers

Number class in Java, along with the **`PrintStream`**, **`DecimalFormat`**, and **`Math`** classes. Continue the trail of this web page and read the following topics: "The Numbers Classes," "Formatting Numeric Print Output," and "Beyond Basic Arithmetic."

::source: http://download.oracle.com/javase/tutorial/java/data/numbers.html::

TOP

Strings

::study time: 80 minutes::

Strings are made up of characters. In Java, strings are objects.

The **`String`** class represents character strings. All string literals in Java, such as 'abc', are implemented as instances of the **`String`** class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because **`String`** objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Visit the following page for an introduction to strings in Java: http://download.oracle.com/javase/tutorial/java/data/strings.html.

Continue reading the trails of this web page for an introduction to

- "Converting Between Numbers and Strings,"
- "Manipulating Characters in a String,"
- "Comparing Strings and Portions of Strings," and
- "The StringBuilder Class."

Also visit the following page, and take a look at the constructor methods and instance methods of the **`String`** class:
http://download.oracle.com/javase/1.3/docs/api/java/lang/String.html

Implement and run the following code samples before adding them to your Programming Profile.

TOP

```
public class ConstructString {
        public static void main(String[] args) {
                StringBuffer buf = new StringBuffer("Vive"); // Vive
                buf.append(" Kumar"); // Vive Kumar
                buf.append(" PhD"); // Vive Kumar PhD
                int index = 14;
                buf.setCharAt(index, '.'); // Vive Kumar PhD
                index = 5;
                buf.insert(index, "SSresh ");// Vive SSresh Kumar PhD
                int start = 6;
                int end = 7;
                buf.replace(start, end, "u"); // Vive Suresh Kumar PhD
                start = 5;
                end = 11;
                buf.delete(start, end); // Vive Kumar PhD
                String s = buf.toString(); // Converts StringBuffer to String.
                System.out.println(s); // Prints the entire string.
        }
}

public class CompareStrings {
        public static void main(String[] args) {
                String s1 = "vive";
                String s2 = "VIVE";
                // Check if s1 and s2 are identical.
                boolean b = s1.equals(s2); // False
                System.out.println(b); // Check while ignoring case.
                b = s1.equalsIgnoreCase(s2); // True
                System.out.println(b); // Compare String with StringBuffer;.
                StringBuffer sbuf = new StringBuffer("vive");
                b = s1.contentEquals(sbuf); // True
                System.out.println(b);
        }
}

public class StringHasSubstring {
        public static void main(String[] args) { // Note: class name starts
                //with uppercase - String.
                // Note: variable name starts with lowercase - string.
                String string = "Java is Cool"; // Starts with a substring
                Boolean b;
                b = string.startsWith("Java is"); // True
                System.out.println(b); // Ends with a substring.
                b = string.endsWith("is Cool"); // True
                System.out.println(b); // Find substring anywhere.
                b = string.indexOf("va is") > 0; // True
                System.out.println(b);
                int start = 1;
                int end = 4;
                String substr = string.substring(start, end); // Str
                System.out.println(substr); // First occurrence of a character.
                int index = string.indexOf('C');
                System.out.println("first C is at " + index); // Last occurrence
                // of a character.
                index = string.lastIndexOf('s');
                System.out.println("last s is at " + index); // No such
                //character.
                index = string.lastIndexOf('z');
                System.out.println("z is not there " + index); // first
                //occurrence of a substring
                index = string.indexOf("Coo");
                System.out.println("first occurrence of Coo is at " + index);
                // Last occurrence of a substring.
                index = string.lastIndexOf("ool");
                System.out.println("last occurrence of ool is at " + index);
                // Substring not found.
                index = string.lastIndexOf("zzzzzzz"); // -1
                System.out.println("substring is not there " + index);
                // Remember, string objects are immutable; you can't
                // change them once created.
                // replace() method creates a new string with the replaced
                // characters.
                // Replace all occurrences of 'o' with 'a'
                String newString = string.replace('o', 'a');
                System.out.println(newString);
                System.out.println("variable string still has " + string);
                // Convert a string to uppercase.
                String upper = string.toUpperCase(); // Convert a String to
                //lowercase string.
                lower = string.toLowerCase();
                System.out.println("Uppercase = " + upper + "Lowercase = "
                                + lower);
        }
}
```

TOP

You can convert a primitive data type value into a **`String`** object either by using a **`String`** method called **`valueOf()`** or using the string’s concatenation operator "+".

```
public class PrimitiveToString {
        public static void main(String[] args) {
                System.out.println("With valueOf() method");
                String s = String.valueOf(true);
                System.out.println(s);
                s = String.valueOf((byte)0x12);
                System.out.println(s); s = String.valueOf((byte)0xFF);
                System.out.println(s); s = String.valueOf('J');
                System.out.println(s); s = String.valueOf((short)12345);
                System.out.println(s); s = String.valueOf(123456);
                System.out.println(s); s = String.valueOf(123456L);
                System.out.println(s); s = String.valueOf(1.23456F);
                System.out.println(s); s = String.valueOf(1.23456D);
                System.out.println(s);
                System.out.println("Now with + concatenation operator"); s = ""
                                + true; // true
                System.out.println(s); s = "" + ((byte)0x12);
                System.out.println(s); s = "" + ((byte)0xFF);
                System.out.println(s); s = "" + 'J';
                System.out.println(s); s = "" + ((short)12345);
                System.out.println(s); s = "" + 123456;
                System.out.println(s); s = "" + 123456L;
                System.out.println(s); s = "" + 1.23456F;
                System.out.println(s); s = "" + 1.23456D;
                System.out.println(s);
        }
}
}
```

A *console* is a command line window. Java library has a class called Scanner that enables reading from console and printing out to console. A **Scanner** object called 'in' is created. This 'in' object enables you to accept input from the console.

```
import java.util.Scanner;
public class ReadFromConsole {
        public static void main(String[] args) {
                String aString;
                int aNumber;
                Scanner in = new Scanner(System.in);
                aString = in.nextLine(); // Read the string from console.
                aNumber = in.nextInt(); // Read an integer.
                in.close();
                System.out.println("aString: " + aString);
                System.out.println("aNumber " + aNumber);
        }
}
```

When you run this program, it will wait for you to type in a string followed by an integer; you can separate the two with a return.

TOP

Enum Data Types

::study time: 30 minutes::

An *enum* type is a type whose fields consist of a fixed set of constants. Common examples include compass directions (values of north, south, east, and west) and the days of the week.

Browse through the following web page for an introduction to **`Enum`** types: http://download.oracle.com/javase/tutorial/java/javaOO/enum.html

TOP

Concluding Unit 2

You may wish to review some of the key concepts introduced in this unit by studying this animation. Now that you have the basics of Java, you are ready to proceed to using control structures in the next unit.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming Profile for Unit 2.

You may also want to make notes on the definitions of the italicized words from Unit 2.

Assessment Activities

Answer questions in the first assignment that belong to Unit 2, and update your Programming Profile.

TOP

Unit 3 Control Structures in Java

Overview

This unit introduces Java's control structures that enable the flow of code execution within the program.

If statements specify how computers make decisions. A key aspect of if statements is that they are based on evaluating *Boolean expressions*, which are expressions that are either true or false. To master programming, it's vital that you master if-statements and Boolean expressions.

Loops make the world go around—or, at least, they make your Java programs go around. They are the final key concept of programming, and loops, along with if statements, variables, and methods, let you do pretty much anything you can imagine. Of course, there are more technical details to learn, but they really are just details compared to the fundamental concepts of loops. So spend as much time as you can mastering loops.

TOP

Learning Outcomes

After completing Unit 3, you should be able to

- demonstrate and write if statements, including *else* and *if-else* statements.
- demonstrate and write *while*, *do-while*, and *for* loop statements.
- write nested control structures.
- write *break, continue*, and *return* statements.
- write *try-catch-finally* statements.

TOP

Control Statements – Introduction

::study time: 10 minutes::

Normally, computer statements are executed in a serial and incremental fashion, one at a time and one after the other. However, at times your algorithm may require you to jump out of the serial execution of statements to a statement that is not next in line for execution. Control statements enable you to perform such jumps in execution.

An ATM banking machine is one situation where you would require such controlled jumps.

When a client inserts the client-card, the software checks if it is a valid card or not. If valid, the client can proceed further; if not, the transaction is rejected.

Suppose the client-card is valid, then the software asks for the client password. If the password entered is correct, then the client can proceed further; else the client is asked to enter the password again.

If the password is a correct one, then the client would be asked to choose from a list of options (such as withdraw cash, deposit cash, check balance, and so on).

In a single ATM transaction you encounter many such controlled jump situations. You can imagine many other real-world situations where you would require similar controlled jumps.

Listed below are Java’s control statements. The following sections in this unit elaborate on a number of control statements.

- if-else
- for
- while
- do-while
- exceptions
- try-catch-finally
- switch
- break
- continue
- return
- labels

Of the lot, the last one should be thoroughly and absolutely avoided, if at all possible: 'labels' are known to lead to out-of-control situations when employed in complex software.

TOP

If-else Statements

::study time: 50 minutes::

We’ll put if-else control statements to use in animated examples. To start with, let us code a pretty uninteresting program. Follow the code and see what it would print!

```
class BoringIf {
        public static void main(String[] args) {
                int x = 5;
                int y = 117;
                if (x == y) {
                        System.out.println(x);
                } else if (x > y) {
                        System.out.println(y);
                } else {
                        System.out.println("x not > or = to y");
                }
        }
}
```

TOP

Coin-Flipping Algorithm

This example shows an animated flowchart for a simple coin-flipping algorithm. A random number from 0 to 1 is chosen, and if that number is less than 0.5, then “heads” is chosen, otherwise “tails” is chosen.

The animation shows the essential nature of the *if statement*: it lets you choose between two or more alternatives. In Java, the choice is always based on a boolean expression, that is, an expression that evaluates to true or false. In contrast, an expression like (3 * x − y) is an arithmetic expression, because it evaluates to a number, e.g., an int or a double depending upon the types of x and y.

Here’s a Java program that could be used to simulate a single coin flip.

```
// Need the Random class to generate random numbers.
import java.util.Random;
public class BigDecisions {
        static public void main(String[] args) {
                // Create a random number object.
                // Calling numberGenerator.nextDouble() will return a
                // random double between 0 and 1.
                Random numberGenerator = new Random();
                // The probability that the coin lands heads up is 0.5.
                final double headProb = 0.5;
                if (numberGenerator.nextDouble() < headProb)
                        System.out.println("Heads");
                else
                        System.out.println("Tails");
        }
}
```

Of course, this isn’t a very useful or interesting program yet because it flips a coin once and then ends. When we learn about loops, you can modify this program to flip a coin any number of times.

TOP

Coin Counting

This animation shows an if-else-if structure in action. To run the animation, select one of the coins on the right side of the animation. Clicking on a coin button causes that coin to be added to the pile. The if-else-if structure then determines which counter to increment. Note that the last coin (with the square hole in the middle) is an unknown coin, and it’s handled by a final else statement.

This animation is drawn as a flowchart, which is one graphical method for diagramming the structure of a program. Flowcharts work well for diagramming if statements, because they clearly show the two or more possible choices. In practice, though, they are often inconvenient because they take up a lot of space and it’s not always easy to write out decisions into those little diamonds! However, they are a simple and intuitive way of representing programs.

TOP

Here’s a Java program that does something similar to coin counting:

```
public class BigDecisions {
        static public void main(String[] args) {
                // Instruction for the user.
                System.out.print("Please choose a coin. Enter \n\n" +
                                " 1 for a penny\n" +
                                " 5 for a nickel\n" +
                                " 10 for a dime\n" +
                                " 25 for a quarter\n" +
                                " 100 for a loonie\n" +
                                " 200 for a toonie\n");
                System.out.println("\nIf you do not enter a whole number,
                                an error " + "will result!");
                                System.out.print("--> ");
                                // Read the user's input.
                                ConsoleReader console = new ConsoleReader(System.in);
                                int coin = console.readInt();
                                // Print the beginning of a response.
                                System.out.print("\nYou chose a ");
                                if (coin == 1) // Is it a penny?
                                        System.out.println("penny.");
                                else if (coin == 5) // Otherwise, is it a nickel?
                                        System.out.println("nickel.");
                                else if (coin == 10) // Otherwise, is it a dime?
                                        System.out.println("dime.");
                                else if (coin == 25) // Otherwise, is it a quarter?
                                        System.out.println("quarter.");
                                else if (coin == 100) // Otherwise, is it a loonie?
                                        System.out.println("loonie.");
                                else if (coin == 200) // Otherwise, is it a toonie?
                                        System.out.println("toonie.");
                                else
                                        // Otherwise, it doesn't match any Canadian coin.
                                        System.out.println("non-existent coin.");
        }
}
```

In the animation, counters are incremented, but here, a message is printed instead. That's because it only checks the value of one coin, so there's no reason to increment a counter.

TOP

Which Would You Choose?

If statements require an understanding of Boolean logic, and one way to practice logic is to try your hand at solving logic puzzles. Here's a very clever one devised by the logician Raymond Smullyan. It's an example of what he calls "coercive logic."

Folow the discussion thread to find the puzzle. Don't peek at the solution until you've settled on what your answer would be!

Visit http://download.oracle.com/javase/tutorial/java/nutsandbolts/if.html to see how if-else control statements can be used in the context of the `Bicycle` class that we explored earlier.

TOP

Loops – For, While, and Do-While

::study time: 60 minutes::

Number Guessing with While Loops

This animation shows a simple guessing game that uses a loop. You pick a number by clicking on one of the digits, and then the program begins guessing numbers until it guesses the number you selected. The guesses are random, so sometimes it will take many wrong guesses before it gets your number.

Here is a Java program that is similar to the program in the animation:

```
import java.util.Random;
public class Loops {
        public static void main(String[] args) {
                Random randGen = new Random();
                ConsoleReader console = new ConsoleReader(System.in);
                // Ask the user to enter a number from 1 to 9.
                //If they enter a number outside that range (e.g. 0, -34, 10,
                // ...), then the program asks them again to enter a number.
                //If the user enters something that is not a number (e.g., the
                // string "one"), then the program aborts with an error.
                int  secret = 0;
                // The user is choosing a number.
                while  ((secret < 1) || (secret > 9)) {
                        System.out.print("Please enter a number from 1 to 9 --> " );
                        secret = console.readInt();
                }
                System.out.println(); // Insert a blank line for neatness.
                // Keep guessing until the guess matches the secret number.
                int  guess = 0;
                // The program will guess.
                while  (guess != secret) {
                        // Generate a random guess from 1 to 9 and print it out.
                        guess = 1 + Math.abs(randGen.nextInt()) % 9;
                        System.out.println(" My guess is "  + guess + "." );
                }
                System.out.println("\nI guessed right! (Finally.)");
        }
}
```

TOP

Two different while loops are used here: one to get the input from the user and one to do the guessing. The animation shown earlier makes it impossible for you to guess a number outside the range 1 to 9, but in this program nothing stops the user from entering invalid data.

Notice how the computer makes its guess. It generates a random **`int`** using **`randGen.nextInt()`**, and then it takes the absolute value of that number to avoid having to worry about negative numbers. Then it uses the mod operator % to calculate the remainder when the number is divided by 9. This remainder is 0, or 1, or 2, . . . or 8 (think for a moment about why the remainder is never 9 or higher), so adding 1 will give an `int` in the range 1 to 9. Remember this technique: it's a useful one for generating random numbers.

One problem with this source code is that it is confined to the range of 1 and 9. If you want to change the range to 10 to 20, say, then you must go through the program and change every instance of 1 and 9 to 10 and 20. If you think the program will change the range frequently, than a better solution is to define constants lo and hi that store the lowest and highest values of the range of number. Then all you need to do to change the range is to set lo to 10 and hi to 20.

Do-While Loops – Encoding Animation Example

This animation shows a simple algorithm for creating secret messages. To use it, first click on the "shift" value. Picking 0 will cause the message to be encoded to itself, so pick something other than 0 to start. Then, pick a short word, like 'mop', and type 'm' into the input box. When you click the button, you will see how 'm' is processed. Then type in 'o' and click the button, and then 'p' followed by a button click. Finally, press return in the character box (that is, enter a \n character) to end the loop and see the coded version of your word.

The essential difference between a while loop and a do-while loop is that the test happens at the bottom of the do-while loop, instead of at the top as in a while loop. That means that the body of a while loop is always executed at least one time, but a while loop's body will not be executed at all if the condition is initially false.

It turns out that any while loop can be written as a do-while loop and vice-versa. In practice, though, do-while loops are not as common as while loops, and it is not always easy to find examples where the do-while loop is the best loop to use.

Visit http://download.oracle.com/javase/tutorial/java/nutsandbolts/while.html for a review of while and do-while loop concepts and implementation details. You should also include all the examples you have studied so far in this section in your Programming Profile.

For Loops

For loops are a common and useful sort of loop. At first they can be confusing because they pack so much into the top of the loop, but most programmers view that as a major advantage of for loops: all the important information about the loop is located in one place.

A common use of for loops is to process every character in a string or some other sequence. Suppose you want to count how many times the character 'e' occurs in a string. You can do it like this:

```
String s = "I like cheese!";
int eCount = 0;
// Begin for loop.
for(int i = 0; i < s.length(); ++i) {
        if (s.charAt(i) == 'e')
                ++eCount;
}
// End for loop.
System.out.println("\"" + s + "\" has " + eCount + " es\n");
```

This is a good code template to keep in mind. Many programming problems involving strings can be solved using variations of this loop.

Here is an animation of a for-loop execution:

TOP

Visit http://download.oracle.com/javase/tutorial/java/nutsandbolts/for.html for a review of for loop concepts and implementation details.

TOP

Break, Continue, Return, and Label Statements

::study time: 30 minutes::

Visit http://download.oracle.com/javase/tutorial/java/nutsandbolts/branch.html and study the material for an introduction to branching statements Do implement all the examples in this page and include them in your Programming Profile.

TOP

Switch Statements

::study time: 30 minutes::

Visit http://download.oracle.com/javase/tutorial/java/nutsandbolts/switch.html and study the material for an introduction to switch statements Do implement all the examples in this page and include them in your Programming Profile.

TOP

Exceptions Statements

::study time: 60 minutes::

Visit http://download.oracle.com/javase/tutorial/essential/exceptions and study the material for an introduction to branching statements. Do implement all the examples in this page, and include them in your programming profile. This lesson includes the following topics – "What Is an Exception?," "The Catch or Specify Requirement," "How to Throw Exceptions," "The try-with-resources Statement," "Unchecked Exceptions," and "Advantages of Exceptions."

TOP

Recursion – An Introduction

::study time: 60 minutes::

What is recursion?
Recursion is a technique where a method calls itself, under specific conditions, multiple times, to solve a problem.

What's a recursive method?
It's a method that calls itself.

If a method calls itself, won't that lead to an infinite loop?
Possibly. But with care, you can write recursive functions that do very useful things in surprisingly simple ways.

What's a base case?
The stopping condition for a recursive method.

Should every recursive method have a base case?

Absolutely! Recursion without a base case results in an infinite loop that will probably crash your computer. Interestingly, you will often get *stack overflow errors* if you forget the base case. This is because each recursive call takes up some memory on the computer's internal stack, and if too many method calls happen in a row, then the stack fills up and overflows. Stack overflow is really bad.

Is recursion practical?

Sometimes yes, although the bookkeeping that the compiler needs to do to keep track of the recursion can slow things down and eat up a lot of memory. Typically, loops are faster and use less memory than recursion.

Is recursion ever necessary?

No. Any recursive program can be written as a program that uses loops instead of recursion. It goes the other way too; any program that uses loops can be rewritten using only recursion.

If recursion is inefficient and unnecessary, why do we bother with it? Is recursion ever useful?

Yes—absolutely! It's an important tool that every programmer should be aware of. Sometimes recursion is the best tool for the job. For example, there's a very practical sorting algorithm called *quicksort* that is easiest to implement and understand recursively. We won't talk about quicksort in this course, but it is one example of where recursion is commonly used in practice. However, in most kinds of practical programming, loops are used much more often than recursion.

Recursion seems to work almost like magic!

That's a common first reaction to recursion. The trick is to think about recursion in the right way. Tracing recursive functions is tricky because you need to keep track of so many different method calls and returns. But when you are solving a problem with recursion, you should not think about tracing the function. Instead, think about the general structure: decide what the base case is, and decide what the recursive case is, and don't worry about how recursion is implemented in Java. Until you've implemented the program, just trust that recursion works. Then you should start to trace through the recursion to see all the details.

Are there examples that could help me consolidate my understanding of recursion?

Of course. I suggest you visit http://danzig.jct.ac.il/java_class/recursion.html for an introduction to recursion in Java with examples.

TOP

Concluding Unit 3

::study time: 120 minutes::

Control structures are essential for the development of professional programs. You should be absolutely comfortable using them in your code.

Visit this page to consolidate your understanding of the control structures introduced in this section.

Also visit http://www.faqs.org/docs/javap/c3/exercises.html and implement the problems and their solutions as part of your Programming Profile.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming Profile for Unit 3.

You may also want to make notes on the definitions of the italicized words from Unit 3.

Assessment Activities

Answer questions in the first assignment that belong to Unit 3, and update your Programming Profile.

TOP

Unit 4 Classes and Objects in Java

Overview

This unit covers a number of additional topics related to classes and methods, some more important than others. Ideas such as recursion and commenting could have their own entire chapters, while the discussion of static methods and variables is a much narrower programming issue.

Further, in this unit we will cover two very important ideas: *inheritance* and *interfaces*.

In Java, you can create new classes by extending (that is, inheriting from) existing subclasses. Inheritance is a key idea for reusing classes.

Interfaces are essentially classes where the methods have no bodies. They let you specify how a class ought to behave without imposing any particular implementation.

TOP

Learning Outcomes

After completing this unit, you will have learned to

- use static variables and methods.
- pass parameters to methods.
- create classes.
- create objects.
- use objects.
- create and use interfaces.
- create and use class hierarchies.

TOP

Objects and Parameters

::study time: 60 minutes::

What is parameter passing?
It’s how methods communicate.

How many kinds of parameter passing does Java have?
Two, essentially.

What are they?
When you are passing primitive types (such as **`boolean, char, int, double`**, and so on), they are passed in a way called *pass by value*. When you are passing objects

(e.g., `String`, `Employee`, or any class of your own creation), then the parameter passing works as if it were pass by reference.

What does "pass by value" mean?
When you pass a parameter by value, the method to which you pass it makes a copy of the original parameter and never alters that original value. Here's an example of a method that illustrates the basic nature of pass by value:

```
    void add1(int n) { ++n; }
```

Suppose you have a variable called x, and if x == 5, and further you call `add1(x)`. What's the new value of x?

Is this a trick question?
Not if you understand pass by value!

Then I'll say 5.
Right! That's because when you pass an `int` as a parameter to a method, the method makes a copy of that `int` value and never touches the original `int`. In the add1 method, this copied `int` is called `n`, and it is `n` that gets incremented, not x. Because `n` is local to add1, it disappears as soon as add1 terminates. The variable x is left unchanged.

Okay. But I'm curious—is there a way to write add1 so that it works correctly?
No, not really. It's possible to play tricks that sort of do what you want, but there is no clean and simple way to write a function in Java that increments a variable. You could write it like this:

```
    int add1(int n) { return n + 1; }
```

But to increment x, you need to write `x = add1(x)`, which is clumsy and inefficient—it's wasteful to make a copy of x and pass its value to the `add1()` method if all you want to do is increment it!

TOP

All right. What about passing objects? How does that work?
In Java, recall that variables refer to objects; they don't label them directly.

For instance, suppose `boss` and `worker` refer to different `Employee` objects. If you write `boss = worker` (for example, because the worker is promoted), then both `boss` and `worker` now refer to the same object. Calling `boss.raiseSalary(1000`) and then calling `worker.raiseSalary(500)`will change the same object, because `boss` and `worker` now refer to the same object.

Okay, that's the basics of how object variables work in Java. What does this have to do with parameter passing?

Now, suppose you have a function called `giveBonus(Employee e, double bonus)`; suppose it's part of some other class (maybe a class called `Company`). If you want to give worker a $500 bonus, then you write `giveBonus(worker,500)`, and then worker's salary is increased by $500.

I see—the value of `worker` is actually changed, unlike the pass by value.
Right. This behaviour is known as *pass by reference*, since we are passing a reference to the method, as opposed to a copy of the object's value.

Remember these two key facts:

> **Pass by value**: when you pass a primitive Java type (e.g, `int, double, boolean,` . . .) to a method, a copy of the value is made and given to the method. The method has no access to the original value, and so cannot change it in any way.
>
> **Pass by reference**: when you pass an object to a method, the method does get access to the original object and it can change the object if necessary.

Okay, that's fab. Can I go now?
One more thing.

Sigh. What is it?
When you pass an object, it turns out that technically speaking, it really is still pass by value. A copy of the reference is made, but the object it points to is not copied. But practically, object parameter passing behaves as if it were pass by reference.

Check this page out for more clarification :): http://www.javaworld.com/javaworld/javaqa/2000-05/03-qa-0526-pass.html.

AAArrrrrrggggghhhhhhhhhh!!
Okay, okay, take it easy now. Read the following web page for a good introduction to objects: http://www.faqs.org/docs/javap/c5/s1.html.

Okay. I've read that. Now, when is a local primitive variable born?
Primitive types (like `int, double, boolean` . . .) are born when you see initialization statements like '`double trouble = 4.99`' or '`boolean hungry = true`'. For example, if x is declared at the top of a method, then x is born every time the method is called.

TOP

When does a local primitive variable die?
At the end of its scope. For example, if x is born at the beginning of a method, then x dies when the method terminates. Unlike people, x can be reborn if the method is called again, although x has no memory of its previous lives. As another example, consider the scope of `i` in the loop `for(int i = 0; i < 10; ++i) { /* ... */ }`. The

variable `i` does not exist before the loop nor after the loop. In general, it's a good idea to minimize the scope of a variable: don't have unused variables hanging around. At best they do nothing, but at worst they could end up causing you grief, for example, due to accidental assignment.

Why do you keep saying "local"? Don't all variables live and die the same way?
Not quite: non-local class variables don't follow the same scoping rules as local variables. A variable defined in a class—not inside a method—is born when the class is created and dies when the class is deleted (e.g., when the garbage collector deletes it or the program terminates). This kind of class variable is visible anywhere in the class, and it doesn't matter if you define it at the top, bottom, or middle of the class. However, for neatness and readability, it's usually best to put all class variables at the top of the class or at the bottom.

When are objects born?
Objects are born wherever you see the keyword new; new means that an object of a given class should be constructed. Classes can declare new objects when they are constructed, so sometimes you can get access to an object without explicitly calling `new`.

Wait a minute. What about declarations like '`String name = "Bonnie";`'. If strings are objects, why don't you need to use `new` to create a new `String`?
Good question. In Java, strings have special privileges that other objects don't get. Since strings are so common, Java lets you avoid using `new` when you initialize a `String` variable. If you want to do extra typing, you can use `new`, and write '`String name = new String("Bonnie");`'.

When do objects die?
That's a tricky question because it depends on what exactly you mean by "die." In Java, if an object has no references to it, then there is no way to access that object. It still exists and takes up memory, but it may as well be dead; such objects might be called zombies. Zombies die either when the program ends or when the garbage collector comes along and deletes them, freeing up the memory they used for use by new objects.

When do objects party?
Never. Objects don't have much of a social life, so they get very few invitations.

TOP

Class

::study time: 180 minutes::

Study this lesson on classes:

http://download.oracle.com/javase/tutorial/java/javaOO/classes.html.

Do implement the `Bicycle` class example in this page and include it in your Programming Profile. There are multiple pages under this lesson. Study all the associated pages on “Declaring Classes,” “Declaring Member Variables,” “Defining Methods,” “Providing Constructors for Your Classes,” and “Passing Information to a Method or a Constructor.”

TOP

Objects – Creation and Using

::study time: 60 minutes::

Study this lesson on creating and using objects: http://download.oracle.com/javase/tutorial/java/javaOO/objects.html.

Do implement all the examples in this lesson and include them in your Programming Profile. There are two pages under this lesson—“Creating Objects” and “Using Objects”—study them both.

:

TOP

More on Classes

::study time: 240 minutes::

Study this lesson: http://download.oracle.com/javase/tutorial/java/javaOO/more.html.

Implement all the examples in this lesson, and include them in your Programming Profile. There are multiple pages under this lesson; study them all including the sections “Nested Classes” and “Annotations.”

TOP

Interfaces and Inheritance in Java

::study time: 120 minutes::

Interfaces and inheritance contribute to two major characteristics of object-oriented programming—*encapsulation* and *abstraction*. Study this lesson on interfaces and inheritance in Java and work through each and every example program: http://download.oracle.com/javase/tutorial/java/IandI/index.html. Do not forget to add these example programs as part of your Programming Profile.

TOP

Concluding Unit 4

::study time: 180 minutes::

This unit forms the basis of object-oriented programming in Java. You should fully understand and be competent enough to start writing code corresponding to each topic introduced in this unit. It is perfectly all right to revisit this unit to ensure that you are comfortable with the Java code and concepts introduced here. You may also wish to study http://www.faqs.org/docs/javap/c5/s3.html and this animated introduction to consolidate your understanding of classes and objects in Java.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far as your Programming Profile for Unit 4.

You may also want to make notes on the definitions of the italicized words from Unit 4.

Assessment Activities

Answer questions in the first assignment that belong to Unit 4, **submit Assignment 1**, and update your Programming Profile.

TOP

Unit 5 Streams in Java

Overview

This unit covers input/output (I/O) streams and file I/O that enable data to flow between input sources and output destinations. The Java tutorial pages cover a number of topics, and you are expected to cover every topic addressed in the tutorials for this unit.

TOP

Learning Outcomes

After completing this unit, you should be able to

- apply streams to read and write files.
- write programs that use byte streams, character streams, buffered steams, data streams, and object streams.
- write programs that use file I/O classes.

TOP

I/O Streams

::study time: 300 minutes::

Study the following Java tutorial lesson in its entirety, along with the example programs in these web pages: http://download.oracle.com/javase/tutorial/essential/io/streams.html.

TOP

File I/O

::study time: 300 minutes::

Study the following Java tutorial lesson in its entirety, along with the example programs in these web pages: http://download.oracle.com/javase/tutorial/essential/io/fileio.html.

You may find some of the terminologies confusing, but go right ahead and start working on the examples. These are important topics that require keen attention on your part to not only understand the syntactic mechanisms but also to apply these mechanisms appropriately in real-world coding contexts.

TOP

Concluding Unit 5

This unit introduced you to streams and file input/output operations. Most advanced programs and applications would require you to routinely employ these two techniques. Try to write as many sample programs corresponding to these topics as possible, and keep an index of these programs for your future reference.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming Profile for Unit 5.

You may also want to make notes on the definitions of the italicized words from Unit 5.

Assessment Activities

Answer questions in the second assignment that belong to Unit 5 and update your Programming Profile.

Review material from the first five units and **take the online Midterm Quiz**.

TOP

Unit 6 Arrays in Java

Overview

Arrays, vectors, lists, stacks, and queues offer different ways to store collections of things. While simple variables can store only one value at a time, these data types can store many different values and offer different mechanisms to retrieve the stored elements.

TOP

Learning Outcomes

After completing this unit, you should be able to

- use uni-dimensional and multi-dimensional arrays.
- use **`ArrayList`** and **`LinkedList`** data structures.
- use vectors, queues, and stacks.

TOP

Arrays

::study time: 40 minutes::

What is an array?
An array is an ordered sequence of objects.

Can you give me another example?
Sure. Here's an array of three doubles in Java:

```
int[] ia= newint[3];
```

There are three slots in **`ia`**, each holding one **`int`**. Initially, all the locations in the array are set to 0. Java’s primitive types, such as **`int`**, each have default values when used in arrays.

*Why did you call it “**`ia`**”?*
`ia` stands for “**`int array`**.”

How do I access individual array elements?
Use the square-bracket notation: **`ia[0]`** is the first element of **`ia, ia[1]`** is the second, and **`ia[2]`** is the third. You can read or write values as if they were ordinary **`int`** variables. The number in brackets is called the *index value*.

What is ?

`ia[0]`
0

TOP

What is `ia[1]`*?*
0

What is `ia[2]`*?*
0

What is `ia[3]`*?*
An error! This raises this exception: `IndexOutOfBoundsException`. Similarly with `ia[-1], ia[4], ia[343]`, etc.

Is 0 always the first index of an array?
Yes, at least in Java.

Isn’t that kind of weird? Why don’t arrays start at 1?
Java follows C’s style of indexing arrays, and the designers of C chose to begin all arrays at index location 0. There’s no earth-shattering reason why arrays should start at 0 instead of 1. Experience shows that starting at 0 makes a few calculations easier, for example, operations involving the mod operator (%).

How can I make an array bigger?
You can’t. For example, `ia` will always have exactly 3 elements, no more, no less. You must create a new array of a different size if you want `ia` to shrink or grow. For example, here’s how you can copy `ia` into a new array of size 5:

```
int[] nia = newint[5];
for(int i=0; i < ia.length; ++i)
        nia[i] = ia[i];
```

Study the following web page for additional introduction to arrays in Java: http://download.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html.

Multidimensional Arrays

You can also declare an array of arrays (also known as a *multidimensional array*) by using two or more sets of square brackets, such as `String[][]` names. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimensionalArray program:

```
class MultiDimensionalArray {
        public static void main(String[] args) {
                String[][] names = { { "Mr. ", "Mrs. ", "Ms. " },
                                { "Smith", "Jones" } };
                System.out.println(names[0][0] + names[1][0]); // Mr. Smith
                System.out.println(names[0][2] + names[1][1]); // Ms. Jones
        }
}
```

The output from this program is

```
         Mr. Smith
         Ms. Jones
```

Java arrays are quite flexible and, for instance, easily let you create tables – arrays of arrays. For example, here's how you can define the first few rows of Pascal's triangle:

```
int[][] triangle = { {1},
                {1,1},
                {1,2,1},
                {1,3,3,1},
                {1,4,6,4,1}
};
for(int i = 0; i < triangle.length; ++i) {
        for(int j = 0; j < triangle[i].length; ++j) {
                System.out.print(triangle[i][j] + " ");
        } // end of inner for
        System.out.println();
} // end of outer for
```

In Pascal's triangle each interior number is equal to the sum of the two numbers above it. The variable triangle is a two-dimensional array of `int` with five rows. Each row is itself an array of `int`, and each row has a different length.

The code under the declaration of triangle is a nested for-loop structure that prints a triangle to the screen. Two-dimensional arrays and doubly-nested loops often work together like this: the outer loop moves along the rows, while the inner-loop moves through each element of a row.

TOP

Vectors

::study time: 40 minutes::

Vectors are commonly used instead of arrays in Java because they expand automatically when new elements are added to them. Vectors can hold only objects. They cannot hold primitive data types such as **`int`** and **`char`**.

You can create a new vector with a default initial size (e.g., **`Vector v = new Vector();`**) or with a defined initial size (e.g., **`Vector v = new Vector(250);`**).

In the animation below, you can play with an example vector. You first create the vector, and then add, change, or remove elements from it.

Here is a simple program that uses the **`Vector`** class to store strings. This code uses *generics* in Java. We'll study generics in the next unit.

```
import java.util.Vector;
public class VectorExample {
        public static void main(String[] args) {

                //  Element type of Vector, e.g., String, Integer, Object ...
                // The code below creates a Vector that only contains Strings
                Vector vc = new Vector();

                // Add vector elements.
                vc.add("Vector Object 1");
                vc.add("Vector Object 2");
                vc.add("Vector Object 3");
                vc.add("Vector Object 4");
                vc.add("Vector Object 5");

                // Add vector element at index 3.
                vc.add(3, "Element at fix position");

                // vc.size() gives the number of elements in Vector.
                System.out.println("Vector Size :" + vc.size());

                // Get elements of Vector.
                for (int i = 0; i < vc.size(); i++) {
                        System.out.println("Vector Element " + i + " :" + vc.get(i));
                }
        }
}
```

TOP

ArrayList

::study time: 40 minutes::

Study the following web page for an introduction to the `ArrayList` data structure in Java: http://www.eecs.qmul.ac.uk/~mmh/DCS128/notes/arrayLists.html. Also implement all the sample programs from this page in your Programming Profile.

TOP

LinkedList

::study time: 40 minutes::

Visit the following page and take a look at different types of linked lists: http://www.mycstutorials.com/articles/data_structures/linkedlists.

A **`LinkedList`** class is built into Java. Shown below is an example (from http://www.roseindia.net) that uses Java's **`LinkedList`** class. This example creates an object of the **`LinkedList`** class and performs various operations on the linked list such as adding and removing objects using the following six methods:

- **`add(Object o)`**: Appends the specified element to the end of this list. It returns a Boolean value.
- **`size()`**: Returns the number of elements in this list.
- **`addFirst(Object o)`**: Inserts the given element at the beginning of this list.
- **`addLast(Object o)`**: Inserts the given element at the last of this list.
- **`add(int index,Object o)`**: Inserts the specified element at the specified position in this list. It throws **`IndexOutOfBoundsException`** if index where element can be added is out of range.
- **`remove(int index)`**: Remove the element at the specified position in this list. It returns the element that was removed from the list. It throws **`IndexOutOfBoundsException`** if index is out of range.

```
import java.util.*;
public class LinkedListDemo {
        public static void main(String[] args) {
                LinkedList link = new LinkedList();
                link.add("a");
                link.add("b");
                link.add(new Integer(10));
                System.out.println("The contents of linkedlist is" + link);
                System.out.println("The size of linkedlist is" + link.size());

                link.addFirst(new Integer(20));
                System.out.println("The contents of linkedlist is" + link);
                System.out.println("The size of linkedlist is" + link.size());
                link.addLast("c");
                System.out.println("The contents of linkedlist is" + link);
                System.out.println("The size of linkedlist is" + link.size());

                link.add(2, "j");
                System.out.println("The contents of linkedlist is" + link);
                System.out.println("The size of linkedlist is" + link.size());

                link.add(1, "t");
                System.out.println("The contents of linkedlist is" + link);
                System.out.println("The size of linkedlist is" + link.size());

                link.remove(3);
                System.out.println("The contents of linkedlist is" + link);
                System.out.println("The size of linkedlist is" + link.size());
        }
}
```

TOP

Here is another example that demonstrates the use of the `LinkedList` class.

```
import java.util.*;
public class LinkedListExample {
        public static void main(String[] args) {
                System.out.println("Linked List Example!");
                LinkedList list = new LinkedList();
                int num1 = 11, num2 = 22, num3 = 33, num4 = 44;
                int size;
                Iterator iterator;
                // Add data to the list.
                list.add(num1);
                list.add(num2);
                list.add(num3);
                list.add(num4);
                size = list.size();
                System.out.print("Linked list data: ");
                // Create an iterator.
                iterator = list.iterator();
                while (iterator.hasNext()) {
                        System.out.print(iterator.next() + " ");
                }
                System.out.println();
                // Check whether the list is empty or not.
                if (list.isEmpty()) {
                        System.out.println("Linked list is empty");
                } else {
                        System.out.println("Linked list size: " + size);
                }
                System.out.println("Adding data at 1st location: 55");
                // Add data at the first position.
                list.addFirst(55);
                System.out.print("Now the list contains: ");
                iterator = list.iterator();
                while (iterator.hasNext()) {
                        System.out.print(iterator.next() + " ");
                }
                System.out.println();
                System.out.println("Now the size of list: " + list.size());
                System.out.println("Adding data at last location: 66");
                // Addlast or append.
                list.addLast(66);
                System.out.print("Now the list contains: ");
                iterator = list.iterator();
                while (iterator.hasNext()) {
                        System.out.print(iterator.next() + " ");
                }
                System.out.println();
                System.out.println("Now the size of list: " + list.size());
                System.out.println("Adding data at 3rd location: 55");
                // Add data at the third position.
                list.add(2, 99);
                System.out.print("Now the list contains: ");
                iterator = list.iterator();
                while (iterator.hasNext()) {
                        System.out.print(iterator.next() + " ");
                }
                System.out.println();
                System.out.println("Now the size of list: " + list.size());
                // Retrieve the first data.
                System.out.println("First data: " + list.getFirst());
                // Retrieve the last data.
                System.out.println("Last data: " + list.getLast());
                // Retrieve specific data
                System.out.println("Data at 4th position: " + list.get(3));
                // Remove the first data.
                int first = list.removeFirst();
                System.out.println("Data removed from 1st location: " + first);
                System.out.print("Now the list contains: ");
                iterator = list.iterator();
                // After removing the first data element from the list,
                // do the following
                while (iterator.hasNext()) {
                        System.out.print(iterator.next() + " ");
                }
                System.out.println();
                System.out.println("Now the size of list: " + list.size());
                // Remove the last data.
                int last = list.removeLast();
                System.out.println("Data removed from last location: " + last);
                System.out.print("Now the list contains: ");
                iterator = list.iterator();
                // After removing the last data from the list,
                // do the following
                while (iterator.hasNext()) {
                        System.out.print(iterator.next() + " ");
                }
                System.out.println();
                System.out.println("Now the size of list: " + list.size());
                // Remove the second data.
                int second = list.remove(1);
                System.out.println("Data removed from 2nd location: " + second);
                System.out.print("Now the list contains: ");
                iterator = list.iterator();
                // After removing data from 2nd location, do the following
                while (iterator.hasNext()) {
                        System.out.print(iterator.next() + " ");
                }
                System.out.println();
                System.out.println("Now the size of list: " + list.size());
                // Remove all data.
                list.clear();
                if (list.isEmpty()) {
                        System.out.println("Linked list is empty");
                } else {
                        System.out.println("Linked list size: " + size);
                }
        }
}
```

TOP

Queue

::study time: 40 minutes::

The Queue interface in Java implements the Collection framework and supports ordering on a first-in-first-out (FIFO) basis. The main difference between a list and a queue is that in the former you can choose what item to get, while in a queue you only get access to the "first in line" object.

The following example from http://www.javadb.com/using-a-queue-or-linkedlist uses the `LinkedList` class to create Queue objects. It enables you to add items using the `add()` method or the `offer()` method. The difference between them is that the `add()` method will throw an exception should the adding of the item fail, but the `offer()` method will simply return `false`. The example also shows how to remove items from the queue by using either of the two methods for removing: `remove()` and `poll()`.

```
import java.util.LinkedList;
import java.util.Queue;
/**
 *
 * @author javadb.com
 */

public class Main {
        /**
         * Example method for using a Queue
         */

        public void queueExample() {
                Queue queue = new LinkedList();
                // Use the add method to add items.
                // Should anything go wrong, an exception will be thrown.
                queue.add("item1");
                queue.add("item2");
                // Use the offer method to add items.
                // Should anything go wrong, it will just return false.
                queue.offer("Item3");
                queue.offer("Item4");
                // Remove the first item from the queue.
                // If the queue is empty, a java.util.NoSuchElementException will
                // be thrown.
                System.out.println("remove: " + queue.remove());

                // Check what item is first in line without removing it.
                // If the queue is empty, a java.util.NoSuchElementException will
                // be thrown.
                System.out.println("element: " + queue.element());
                // Remove the first item from the queue.
                // If the queue is empty, the method just returns false.
                System.out.println("poll: " + queue.poll());
                // Checkwhat item is first in line without removing it.
                // If the queue is empty, a null value will be returned.
                System.out.println("peek: " + queue.peek());
        }
        /**
         * @param args
         * the command line arguments
         */

        public static void main(String[] args) {
                new Main().queueExample();
        }
}
```

TOP

Given below is another example from http://www.roseindia.net/java/example/java/util/QueueImplement.shtml that offers a queue that handles only Integer objects.

```
import java.io.*;
import java.util.*;
public class QueueImplement {
         LinkedList list;
        String str;
        int num;
        public static void main(String[] args) {
                QueueImplement q = new QueueImplement();
        }
        public QueueImplement() {
                try {
                         list = new LinkedList();
                         InputStreamReader ir = new InputStreamReader(System.in);
                         BufferedReader bf = new BufferedReader(ir);
                         System.out.println("Enter number of elements : ");
                         str = bf.readLine();
                         if ((num = Integer.parseInt(str)) == 0) {
                                 System.out.println("You have entered either zero
                                                                   or null.");
                                 System.exit(0);
                         } else {
                                 System.out.println("Enter elements : ");
                                 for (int i = 0; i < num; i++) {
                                         str = bf.readLine();
                                         int n = Integer.parseInt(str);
                                         list.add(n);
                                 }
                         }
                         System.out.println("First element :" + list.removeFirst());
                         System.out.println("Last element :" + list.removeLast());
                         System.out.println("Rest elements in the list :");
                         while (!list.isEmpty()) {
                                 System.out.print(list.remove() + "\t");
                         }
                } catch (IOException e) {
                         System.out.println(e.getMessage() + " is not a legal entry.");
                         System.exit(0);
                }
        }
}
```

TOP

Stack

::study time: 40 minutes::

Stacks implement a last-in-first-out (LIFO) mechanism to handle data. That is, you have access to only the last element added to the list. Read the following two pages for a quick introduction to the `Stack` data structure with some examples.

::source:http://www.javacoffeebreak.com/faq/faq0037.html::

::source:http://www.roseindia.net/answers/viewqa/Java-Beginners/10513-java-using-Stack.html::

TOP

Concluding Unit 6

This unit introduced you to a few common data structures available in Java. By now you should be comfortable using any of these data structures in your programs. What we did not introduce is the relation between specific problems and specific data structures. That is something you will have to pick up in an advanced programming course.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming Profile for Unit 6.

You may also want to make notes on the definitions of the italicized words from Unit 6.

Assessment Activities

Answer questions in Assignment 2 that belong to Unit 6, and update your Programming Profile.

TOP

Unit 7 Events in Java

This unit provides an introduction to

- event handling in Java.
- types of events (action, item, and change).
- dealing with multiple events handling.

TOP

Learning Outcomes

After completing this unit, you should be able to

- describe a Swing resource.
- use Swing components.
- apply event listeners to Swing components.
- apply multiple listeners to a single Swing component.

TOP

Event Handling – Introduction to Swing

::study time: 240 minutes::

Java provides two sets of components to deal with graphical user interfaces: AWT and Swing. These form the basis of Java's event handling. Let us focus only on Swing components. Let us start with a simple demo.

Event-Handling Demo

This animation provides an analogy to help you think about how Java's event handling works. In order to get a script, an actor must register with the agency. Different actors favour different kinds of scripts, and whenever a script comes into the agency, it's forwarded to all the actors interested in it. Scripts are the events, the agency is the event source, and the actors/actresses are the event handlers and listeners. An actor registering with an agency is like installing a listener. Multiple actors could register with the agency, and multiple scripts could come in a sequence.

A major goal of Java is to be platform-independent, i.e., the designers didn’t want Java to work just on Sun computers or just on Windows computers. That is, you should be able to write a program on one computer and have it behave identically on any other computer that implements the Java virtual machine.

This is a laudable goal, but it adds a lot of complexity to the graphical user interface (GUI) libraries. Since you don’t know what sort of computer is running your Java program, the designers of Java’s GUI library could only use generic features that they were sure would be common to all computers that might run Java programs. Java designers’ first attempt is called the AWT, and it was not very successful; it was replaced by the Swing library. The AWT would use the windowing toolkit on your computer, which means that Java programs would have a different look and feel when run on different computers.

In contrast, Swing directly draws its own GUI components, so Swing applets and applications look the same on different platforms. Doing so makes it difficult, or impossible, to reliably “port” a program to another computer. Both the AWT and Swing are included in the current versions of Java, since not all Java programmers have adopted the newer Swing library.

Study the material from the following links that introduce you to the Swing-based GUI and handling of events in the GUI. All the links are part of a lesson trail from the Oracle Java tutorials.

Getting Started with Swing

Study the contents of http://download.oracle.com/javase/tutorial/uiswing/start/about.html to learn about the key features of Java Foundation Classes (JFC). As you can see, Swing is just one of the features of JFC.

Compiling and Running Swing Programs

Study the contents of http://download.oracle.com/javase/tutorial/uiswing/start/compile.html which offers steps to compile and run Swing programs. The page also contains a tutorial link to using the NetBeans IDE. You can skip it and continue using the BlueJ IDE to develop Swing programs.

Include the HelloWorldSwing.java program introduced in this web page as part of your programming profile. This program uses Java's package mechanism. We'll thoroughly introduce packages in the last unit.

The following link shows the Swing-based code for a Celsius-to-Fahrenheit conversion application: CelsiusConverterGUI.java. As you can see, Swing components look rather complicated. However, you will come to understand that most of the code is rather standard material that you typically cut and paste.

Here is another Swing application that you can try running from within BlueJ. Simply cut and paste the code, compile, and run:

```
import javax.swing.JOptionPane;

 public classBodyMassSwing

{

        public static void calculateMyBodyMass()
        {

                //Declare and construct variables.

                String height, weight;

                int inches, pounds;

                double kilograms, meters, index;


                //Print prompts and get input.

                System.out.println("\tTHE SUN FITNESS CENTER BODY MASS
 INDEX CALCULATOR");

                height = JOptionPane.showInputDialog(null, "Enter your height to the
 nearest inch: ");

                inches = Integer.parseInt(height);

                weight = JOptionPane.showInputDialog(null, "Enter your weight to the
 nearest pound: ");

                pounds = Integer.parseInt(weight);


                //Do calculations.

                meters = inches / 39.36;

                kilograms = pounds / 2.2;

                index = kilograms / Math.pow(meters,2);


                //Output.

                JOptionPane.showMessageDialog(null, "YOUR BODY MASS INDEX IS " +
                                   Math.round(index) + ".",

                                   "Body Mass Calculator" , JOptionPane.PLAIN_MESSAGE );

                //System.exit(0); not needed.

        }

}
```

Introduction to Swing Components

Visit http://download.oracle.com/javase/tutorial/ui/features/components.html and take a look at various Swing components that you can include in your programs.

As outlined in http://download.oracle.com/javase/tutorial/uiswing/components/toplevel.html, all Swing components must be part of a containment hierarchy. The top level of this hierarchy must be one of the following three container classes: JFrame, JDialog, or JApplet. Each of these top-level containers has a content pane. All other Swing components can be added to these top-level containers, that is, to their respective content panes. Optionally, each of these top-level containers can have the menu bar. In this unit, we will focus only on the JFrame top-level container to develop Swing applications.

Each GUI Swing component can be contained only once. That is, each component can be added to a container only once. If a component is already inside a container and you try to add it to another container, the component will be removed from the first container and then added to the second.

Implement the TopLevelDemo.java program from TopLevelDemo.java. It uses a JFrame object, called frame, as the top-level container. The application has just two methods, a `main()` method and a `createAndShowGUI()` method. The `main()` method invokes the `createAndShowGUI()` method in a Java Thread. We'll study about threads later on. The `createAndShowGUI()` method creates the JFrame object, customizes it, creates a Menu object (called `greenMenuBar`), creates a custom Label object (called `yellowLabel`) , adds the Menu object to the frame, adds the Label object to the content pane of the JFrame object, and finally displays them all.

All Swing components whose names begin with the letter "J" (e.g., JLabel, JMenuBar, JScrollPane, JButton, and JTable) have JComponent class as their parent. The

JComponent class provides a range of functionality to its descendants including mouse event handling, key bindings, drag and drop, support for accessibility, and look and feel.

Using Text Components

Study http://download.oracle.com/javase/tutorial/uiswing/components/text.html, and implement all the example programs presented in the page. The TextSamplerDemo.java application introduces you to JTextField, JPasswordField, JFormattedTextField, JTextArea, JEditorPane, and JTextPane.

JTextComponent is the foundation class for all Swing text components. The following web page presents an example that shows some of the functionality offered by JTextComponent to all Swing text components: http://download.oracle.com/javase/tutorial/uiswing/components/generaltext.html. At this time, we only expect you to run the TextxComponentDemo using the “Launch” button in this web page. You are welcome to look at the .java files to see how this program has been implemented.

TOP

Introduction to Event Listeners

::study time: 30 minutes::

Study http://download.oracle.com/javase/tutorial/uiswing/events/intro.html for an introduction to event listeners. Review Beeper.java and study how the `ActionListener` event has been implemented for a button. Then, review MultiListener.java and study how multiple events have been implemented in a single program.

TOP

General Information about Writing Event Listeners

::study time: 30 minutes::

Study http://download.oracle.com/javase/tutorial/uiswing/events/generalrules.html for information concerning design considerations, event objects, types of events, event adapters, inner classes, and the `eventhandler` class.

TOP

Listeners Supported by Swing Components

::study time: 30 minutes::

Study http://download.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html to understand the types of event listeners associated with various Swing components.

TOP

Implementing Listeners for Commonly Handled Events

::study time: 300 minutes::

Study http://download.oracle.com/javase/tutorial/uiswing/events/handling.html, and implement the following listeners as part of your programming profile:

- How to Write an Action Listener
- How to Write a Change Listener
- How to Write a Component Listener
- How to Write a Container Listener
- How to Write a Document Listener
- How to Write an Item Listener
- How to Write a Key Listener
- How to Write a List Data Listener
- How to Write a List Selection Listener
- How to Write a Mouse Listener
- How to Write a Mouse-Motion Listener
- How to Write a Table Model Listener
- How to Write a Tree Expansion Listener
- How to Write Window Listeners

Now that you are comfortable writing event listeners, we'll get back to implementing some Swing components.

TOP

Introduction to Swing Continues

::study time: 300 minutes::

Study http://download.oracle.com/javase/tutorial/uiswing/components/componentlist.html, and implement the following Swing components as part of your programming profile:

- How to Use Buttons, Check Boxes, and Radio Buttons
- How to Use Combo Boxes
- How to Make Dialogs
- How to Use Editor Panes and Text Panes
- How to Use File Choosers
- How to Use Formatted Text Fields
- How to Make Frames (Main Windows)
- How to Use Internal Frames
- How to Use Labels
- How to Use Lists
- How to Use Menus
- How to Use Password Fields
- How to Use Progress Bars
- How to Use Scroll Panes
- How to Use Sliders
- How to Use Spinners
- How to Use Split Panes
- How to Use Tabbed Panes
- How to Use Tables
- How to Use Text Areas
- How to Use Text Fields
- How to Use Tool Bars
- How to Use Tool Tips
- How to Use Trees

By now you should be absolutely comfortable writing Swing programs that are capable of handling events.

TOP

Concluding Unit 7

This unit introduced you to the world of Swing and events in Java. Using simple Swing-based interfaces, one could develop world-class professional applications.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming Profile for Unit 7.

You may also want to make notes on the definitions of the italicized words from Unit 7.

Assessment Activities

Answer questions in Assignment 2 that belong to Unit 7, and update your Programming Profile.

TOP

Unit 8 Applets

Overview

Applets offer a novel method to deploy Java applications. Further, a significant number of real-world applications now have been implemented as Applets because of their ease of use, wider applicability, and security.

TOP

Learning Outcomes

After completing this unit, you should be able to

- describe applets in terms of their lifecycle.
- describe applets' execution environment.
- develop and deploy Applets.
- develop and deploy JApplets.
- use the various functionalities of Applets and JApplets.

TOP

Applets

::study time: 420 minutes::

There is no better introduction to Applets than the following tutorial from http://download.oracle.com/javase/tutorial/deployment/applet/index.html. Study the entire trail without leaving anything behind.

TOP

JApplets

::study time: 60 minutes::

Study this web page for an introduction to JApplets: http://download.oracle.com/javase/tutorial/uiswing/components/applet.html.

TOP

Concluding Unit 8

Applets belong to one of the new paradigms in computing in recent years. By now, you should be absolutely comfortable in discussing, designing, developing, and deploying applets.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming Profile for Unit 8.

You may also want to make notes on the definitions of the italicized words from Unit 8.

Assessment Activities

Answer questions in the second assignment that belong to Unit 8, **submit Assignment 2**, and update your Programming Profile.

TOP

Unit 9 Concurrency in Java (aka Java Threads)

Overview

Threads is an important topic in Java. It allows Java programs to run multiple processes or perform multiple tasks at the same time. Almost all professional Java applications would have used threads. Programs that use threads are commonly known as *concurrent programs* and the underlying notion is called *concurrency*.

TOP

Learning Outcomes

After completing this unit, you should be able to

- describe threads.
- apply threads in applets.
- use synchronization methods in threads.
- demonstrate the liveness of threads.
- demonstrate a strategy to define immutable objects.

TOP

Concurrency – An Introduction

::study time: 450 minutes::

Study http://en.wikibooks.org/wiki/Java_Programming/Threads_and_Runnables for a quick introduction to threads and *runnables*, the two techniques that enable concurrency in Java.

Study “Thread Objects,” “Synchronization,” “Liveness, “Guarded Blocks,” and “Immutable Objects” in the following lesson: http://download.oracle.com/javase/tutorial/essential/concurrency/index.html.

TOP

Concurrency in Swing

::study time: 200 minutes::

Study the following lesson on concurrency in Swing: http://download.oracle.com/javase/tutorial/uiswing/concurrency/index.html.

For your additional reading, http://www.buyya.com/java/Chapter14.pdfoffers a full chapter on Socket Programming.

TOP

Concluding Unit 9

One cannot claim to be a professional programmer without a thorough understanding and practical knowhow about implementing concurrent programs. Not every Java program needs to be threaded. However, most real-world applications would demand concurrency, hence threads.

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming

Profile for Unit 9.

You may also want to make notes on the definitions of the italicized words from Unit 9.

Assessment Activities

Update your Programming Profile.

TOP

Unit 10 Advanced Topics in Java

Overview

Java provides a wide range of advanced features to address a number of real-world techniques. The following lists some of these features:

- Custom networking – The Java platform's powerful networking features.
- The extension mechanism – Custom APIs available to all applications running on the Java platform.
- Full-screen exclusive mode API – Applications that more fully utilize the user's graphics hardware.
- Generics – An enhancement to the type system that supports operations on objects of various types while providing compile-time type safety.
- Internationalization – Designing software so that it can be easily be adapted (localized) to various languages and regions.
- JavaBeans – The Java platform's component technology.
- JDBC Database Access – An API for connectivity between the Java applications and a wide range of databases and a data sources.
- JMX – Java Management Extensions provides a standard way of managing resources such as applications, devices, and services.
- JNDI – Java Naming and Directory Interface enables accessing the naming and directory services such as DNS (domain-name system) and LDAP (lightweight directory access protocol).

RMI – The Remote Method Invocation API allows an object to invoke methods of an object running on another Java Virtual Machine.
- Reflection – An API that represents ("reflects") the classes, interfaces, and objects in the current Java Virtual Machine.
- Security – Java platform features that help protect applications from malicious software.
- Sound – An API for playing sound data from applications.
- 2D Graphics – Displaying and printing 2D graphics in applications.

This unit reviews only a handful of features from this list.

TOP

Learning Outcomes

After completing this unit, you should be able to

- use *packages*.
- use abstract methods and classes.
- use *sockets* for client–server network programming in Java.

TOP

Packages

::study time: 180 minutes::

Study http://download.oracle.com/javase/tutorial/java/package/index.html for a detailed introduction to packages in Java.

TOP

Abstract Methods and Classes

::study time: 30 minutes::

Study the following web page for an introduction to abstract methods and classes:

http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

TOP

Socket Programming

::study time: 300 minutes::

Earlier in I/O streams, we studied that data can move from a source to a destination, say from the console input to a file output. Similarly, we can send data from one computer to another using socket programming. Once you establish a socket connection between two computers, data can transmit between the two. Sockets allow you to treat network connections as you would any other I/O.

Study the following page for a brief introduction to networking basics: http://download.oracle.com/javase/tutorial/networking/overview/networking.html.

Study the following page for a quick introduction to URLs and URLConnections: http://docs.oracle.com/javase/tutorial/networking/urls/index.html.

Then, study the following lesson about sockets: http://download.oracle.com/javase/tutorial/networking/sockets/index.html. This lesson includes the topics “What Is a Socket?,” “Reading from and Writing to a Socket,” and “Writing a Client/Server Pair.”

Finally, study the following lesson about datagrams: http://download.oracle.com/javase/tutorial/networking/datagrams/index.html. This lesson includes the topics “What Is a Datagram,” “Writing a Datagram Client and Server,” and “Broadcasting to Multiple Recipients.”

For your additional reading, http://www.buyya.com/java/Chapter13.pdf offers a full chapter on socket programming.

TOP

Concluding Unit 10

This is the end of the course. Or, you can think of it as the beginning of a journey in Java programming

Programming Profile

::study time: 60 minutes::

Include all the programs you have worked on so far in your Programming Profile for Unit 10.

You may also want to make notes on the definitions of the italicized words from Unit 10.

Assessment Activities

Update your Programming Profile, and **submit the entire profile** (at least 25 programming problems, solutions for these problems, your personal comments, and preferably the MILE .mtd file corresponding to each of these 25 problems) in a single zip file.

Start reviewing the material in preparation for the final exam. Best wishes for success.

TOP