LEARNER MANUAL

APPLY PRINCIPLES OF CREATING COMPUTER SOFTWARE BY DEVELOPING A COMPLETE PROGRAMME TO MEET GIVEN BUSINESS SPECIFICATIONS

US ID: 115392

NQF LEVEL: 5

CREDITS: 12

NOTIONAL HOURS: 120

CONTENTS

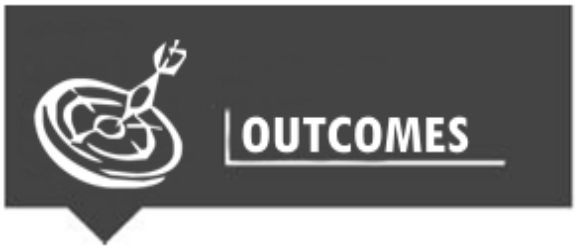
Table of Contents

This manual was compiled by SM Support on behalf of the training provider.

HOW TO USE THIS GUIDE

This workbook belongs to you. It is designed to serve as a guide for the duration of your training programme. It contains readings, activities, and application aids that will assist you in developing the knowledge and skills stipulated in the specific outcomes and assessment criteria. Follow along in the guide as the facilitator takes you through the material, and feel free to make notes and diagrams that will help you to clarify or retain information. Jot down things that work well or ideas that come from the group. Also, note any points you would like to explore further. Participate actively in the skill practice activities, as they will give you an opportunity to gain insights from other people's experiences and to practice the skills. Do not forget to share your own experiences so that others can learn from you too.

ICONS



PROGRAMME OVERVIEW

PURPOSE

This unit standard is intended:

- To provide a expert knowledge of the areas covered
- For those working in, or entering the workplace in the area of Systems Development
- To demonstrate an understanding of how to create (in the computer language of choice) a complete computer program that will solve a given business problem, showing all the steps involved in creating computer software

LEARNING ASSUMPTIONS

The credit value of this unit is based on a person having the prior knowledge and skills to:

- Apply the principles of Computer Programming
- Design, develop and test computer program segments to given specifications

HOW YOU WILL LEARN

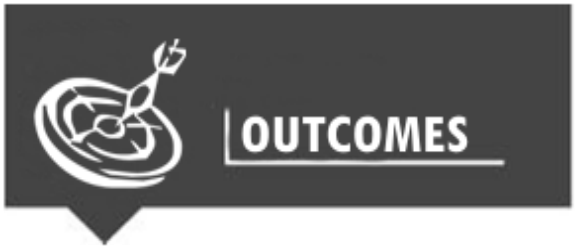
The programme methodology includes facilitator presentations, readings, individual activities, group discussions, and skill application exercises.

HOW YOU WILL BE ASSESSED

This programme has been aligned to registered unit standards. You will be assessed against the outcomes of the unit standards by completing a knowledge assignment that covers the essential embedded knowledge stipulated in the unit standards. When you are assessed as competent against the unit standards, you will receive a certificate of competence and be awarded 12 credits towards a National Qualification.

Prescribed Learning Material / Text	Author	Publisher	Supplier	Price per book/ per page (Vat excluded)
Concurrent Programming in Java: Design Principles and Patterns (Java Series)	Douglas Lea	Addison-Wesley Pub (SD)	ISBN-13: 978-0201695816	R350

SESSION 1: PLANNING A COMPUTER PROGRAM SOLUTION



On completion of this section you will be able to interpret a given specification to plan a computer program solution

1. The plan proposes a description of the problem to be solved by the development of the computer program that is understandable by an end-user and meets the given specification.
2. The plan integrates the research of problems in term of data and functions
3. The plan includes an evaluation of the viability of developing a computer program to solve the problem identified and compares the costs of developing the program with the benefits to be obtained from the program.
4. The plan concludes by choosing the best solution and documenting the program features that will contain the capabilities and constraints to meet the defined problem.

1.1 INTRODUCTION

You may already have used software, perhaps for word processing or spreadsheets, to solve problems. Perhaps now you are curious to learn how programmers write software. A *program* is a set of step-by-step instructions that directs the computer to do the tasks you want it to do and produce the results you want.

There are at least three good reasons for learning programming:

- Programming helps you understand computers. The computer is only a tool. If you learn how to write simple programs, you will gain more knowledge about how a computer works.
- Writing a few simple programs increases your confidence level. Many people find great personal satisfaction in creating a set of instructions that solve a problem.
- Learning programming lets you find out quickly whether you like programming and whether you have the analytical turn of mind programmers need. Even if you decide that programming is not for you, understanding the process certainly will increase your appreciation of what programmers and computers can do.

A set of rules that provides a way of telling a computer what operations to perform is called a programming language. There is not, however, just one programming language; there are many. In this chapter you will learn about controlling a computer through the process of programming. You may even discover that you might want to become a programmer.

An important point before we proceed: You will not be a programmer when you finish reading this chapter or even when you finish reading the final chapter. Programming proficiency takes practice and training beyond the scope of this book. However, you will become acquainted with how programmers develop solutions to a variety of problems.

What Programmers Do

In general, the programmer's job is to convert problem solutions into instructions for the computer. That is, the programmer prepares the instructions of a computer program and runs those instructions on the computer, tests the program to see if it is working properly, and makes corrections to the program. The programmer also writes a report on the program.

These activities are all done for the purpose of helping a user fill a need, such as paying employees, billing customers, or admitting students to college.

The programming activities just described could be done, perhaps, as solo activities, but a programmer typically interacts with a variety of people. For example, if a program is part of a system of several programs, the programmer coordinates with other programmers to make sure that the programs fit together well. If you were a programmer, you might also have coordination meetings with users, managers, systems analysts, and with peers who evaluate your work-just as you evaluate theirs.

The Programming Process

Developing a program involves steps similar to any problem-solving task. There are five main ingredients in the programming process:

- Defining the problem
- Planning the solution
- Coding the program
- Testing the program
- Documenting the program

1.2 PLANNING A PROGRAM SOLUTION

All system development involves the software development cycle to some extent. Some approaches will take more time to define the problem, plan the solution, build the solution, check the solution and modify the solution. Some approaches will be more organized and others may cost very little. No one approach is better than the other but some are more suitable in some situations. We will be looking at the following areas:

- Defining the problem
- Abstraction/refinement
- Data types
- Structured algorithms

As this is a topic with loads of theory, there is no single worksheet. Instead there are individual worksheets for specific points.

1.2.1 DEFINING AND DESCRIBING THE PROBLEM (SO 1, AC 1)

“Defining the problem is the most important part of the software development cycle. If the requirements and the parameters of the problem are clearly understood then the actual output of the development program (process) is far more likely to meet the expected output. The first step in defining a problem is to fully understand the problem which needs to be solved.

This includes:

- Stating the problem in other terms which give greater meaning to the problem.
- explaining the essential qualities or aspects of the problem
- Deciding the boundaries of the problem ".

Information

Simple or wide-encompassing statements like "I want to computerize my payroll system" or "I want a website to sell my wares" or "I want an interesting game that is set in space" are of very limited use when you are the analyst/programmer who's job it is to create the software solution.

And so, irrespective of the software approach you may finally take, it is extremely important to examine and analyse the actual requirements and then to plan how you will achieve the final development before you think of coding anything. The first two stages of development are

- Defining the problem and
- Planning the Solution.

In the Defining the Problem stage, typically a software analyst will sit with management and the system users to discuss and record the requirements of the problem that needs to be solved. It is extremely important to talk to both parties as they both contribute to your understanding of their needs and their expectations. The requirements are then written in a non-technical way and are expressed in terms of what must be done, not how it is to be done.

A Systematic Approach to Defining a Problem

Defining the problem is the first step towards a problem solution. A systematic approach to problem definition, leads to a good understanding of the problem. Here is a tried and tested method for defining (or specifying) any given problem:

Divide the problem into three (3) separate components:

(a) Input or source data provided

(b) Output or end result required

(c) Processing - a list of what actions are to be performed

EXAMPLE 1: A program is required to read three (3) numbers, add them and print their total. It is usually helpful to write down the three (3) components in a defining diagram as shown below:

INPUT	PROCESSING	OUTPUT
Read : Num1, Num2, Num3	Total = Num1 + Num2 + Num3	Print : Total

EXAMPLE 2: Design a program to read in the max. and min. temperature on a particular day and calculate and print the average temp.

INPUT	PROCESSING	OUTPUT
Read : max_ temp, min_ temp	Calculate avg_temp	Print : avg_temp

1.2.2 INTEGRATING RESEARCH PROBLEMS IN TERMS OF DATA AND FUNCTIONS (SO 1, AC 2)

Before beginning a program, you must have a firm idea of what the program should produce and what data is needed to produce that output. Just as a builder must know what the house should look like before beginning to build it, a programmer must know what the output is going to be before writing the program.

Anything that the program produces and the user sees is considered output that you must define. You must know what every screen in the program should look like and what will be on every page of every printed report.

Some programs are rather small, but without knowing where you're heading, you may take longer to finish the program than you would if you first determined the output in detail. Liberty BASIC comes with a sample program called Contact3.bas that you can run. Select File, Open, and select Contact3.bas to load the file from your disk. Press Shift+F5 to run the program and then you should see the screen shown in below No contacts exist when you first run the program, so nothing appears in the *fields* initially. A field, also known as a text box, is a place where users can type data.

Example: Even Liberty BASIC's small Contact Management program window has several fields.

If you were planning to write such a contact program for yourself or someone else, you should make a list of all fields that the program is to produce onscreen. Not only would you list each field but you also would describe the fields. In addition, three Windows command buttons appear in the program window. Table 3.1 details the fields on the program's window.

Table 3.1 Fields That the Contact Management Program Displays

Field	Type	Description
Contacts	Scrolling list	Displays the list of contacts
Name	Text field	Holds contact's name
Address	Text field	Holds contact's address
City	Text field	Holds contact's city
State	Text field	Holds contact's state

Field	Type	Description
Zip	Text field	Holds contact's zip code
Phone #	Text field	Holds contact's phone number
Stage	Fixed, scrolling list	Displays a list of possible stages this contact might reside in, such as being offered a special follow-up call or perhaps this is the initial contact
Notes	Text field	Miscellaneous notes about the contact such as whether the contact has bought from the company before
Filter Contacts	Fixed, scrolling list	Enables the user to search for groups of contacts based on the stage the contacts are in, enabling the user to see a list of all contacts who have been sent a mailing
Edit	Command button	Enables the user to modify an existing contact
Add	Command button	Enables the user to add a new contact
OK	Command button	Enables the user to close the contact window

Many of the fields you list in an output definition may be obvious. The field called Name obviously will hold and display a contact's name. Being obvious is okay. Keep in mind that if you write programs for other people, as you often will do, you must get approval of your program's parameters. One of the best ways to begin is to make a list of all the intended program's fields and make sure that the user agrees that everything is there. As you'll see in a section later this hour named "Rapid Application Development," you'll be able to use programs such as Visual Basic to put together a model of the actual output screen that your users can see. With the model and with your list of fields, you have double verification that the program contains exactly what the user wants.

Input windows such as the Contacts program data-entry screen are part of your output definition. This may seem contradictory, but input screens require that your program place fields on the screen, and you should plan where these input fields must go.

The output definition is more than a preliminary output design. It gives you insight into what data elements the program should track, compute, and produce. Defining the output also helps you gather all the input you need to produce the output.

CAUTION

Some programs produce a huge amount of output. Don't skip this first all-important step in the design process just because there is a lot of output. Because there is more output, it becomes more important for you to define it. Defining the output is relatively easy—sometimes even downright boring and time-consuming. The time you need to define the output can take as long as typing in the program. You will lose that time and more, however, if you shrug off the output definition at the beginning.

The output definition consists of many pages of details. You must be able to specify all the details of a problem before you know what output you need. Even command buttons and scrolling list boxes are output because the program will display these items.

1.2.3 INCLUSION OF EVALUATION OF THE VIABIITY OF THE COMPUTER PROGRAM (SO 1, AC 3)

The plan must include the evaluation of the solution. Remember, once a solution has been designed using computational thinking, it is important to make sure that the solution is fit for purpose.

Evaluation is the process that allows us to make sure our solution does the job it has been designed to do and to think about how it could be improved.

Once written, an algorithm should be checked to make sure it:

- Is easily understood – is it fully decomposed?
- Is complete – does it solve every aspect of the problem?
- Is efficient – does it solve the problem, making best use of the available resources (e.g. as quickly as possible/using least space)?
- meets any design criteria we have been given

If an algorithm meets these four criteria it is likely to work well. The algorithm can then be programmed.

Failure to evaluate can make it difficult to write a program. Evaluation helps to make sure that as few difficulties as possible are faced when programming the solution.

Once a solution has been decided and the algorithm designed, it can be tempting to miss out the evaluating stage and to start programming immediately. However, without evaluation any faults in the algorithm will not be picked up, and the program may not correctly solve the problem, or may not solve it in the best way.

Faults may be minor and not very important. For example, if a solution to the question ‘how to draw a cat?’ was created and this had faults, all that would be wrong is that the cat drawn might not look like a cat. However, faults can have huge – and terrible – effects, eg if the solution for an aeroplane autopilot had faults.

Ways that solutions can be faulty

We may find that solutions fail because:

- it is not fully understood - we may not have properly decomposed the problem
- it is incomplete - some parts of the problem may have been left out accidentally
- it is inefficient – it may be too complicated or too long
- it does not meet the original design criteria – so it is not fit for purpose

A faulty solution may include one or more of these errors.

1.2.4 CHOOSING THE BEST SOLUTION AND DOCUMENTING PROGRAM FEATURES (SO 1, AC 4)

The plan must conclude by choosing the best solution and documenting the program features that will contain capabilities and constraints to meet defined problem.

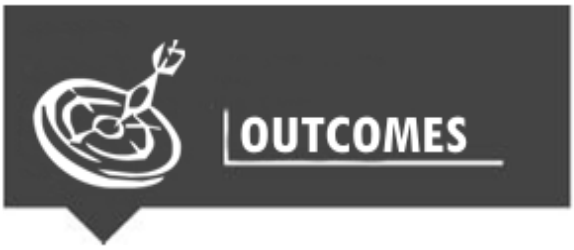
If we all had infinite time to make all the programs in our heads, then they'd all include every feature in our list. But we don't, so in this step, you have to decide which features are the most important, and which features you'll do only if we have time. This will also help you figure out which order to implement features in, from most to least important.

To help you figure out the importance of each feature, ask yourself these questions:

- If I shared this with a friend, which features would I want to make sure were working?
- Which features am I the most excited about building?
- Which features are the most unique to my program?
- Which features will I learn the most from implementing?
- Are there any features that seem too far beyond my current skill level?

Then, go through your feature list from the last step, and either order the list or add a rank to each feature. The ordered list of features will assist you in deciding what is most important to build first. The earlier that you can deliver a feature to get feedback, the better your system will become.

SESSION 2: DESIGNING AND CREATING A COMPUTER PROGRAM



On completion of this section you will be able to design a computer program to meet a business requirement.

1. The design incorporates development of appropriate design documentation and is desk-checked.
2. The design of the program includes program structure components.
3. The design of the program includes program logical flow components.
4. The design of the program includes data structures and access method components.
5. The creation includes coding from design documents, to meet the design specifications.
6. Names created in the program describe the purpose of the items named
7. The creation includes conformance with the design documentation, and differences are documented with reasons for deviations

2.1 INTRODUCTION

Programming is writing computer code to create a program, to solve a problem. Programs are created to implement algorithms. Algorithms can be represented as pseudo code or a flowchart, and programming is the translation of these into a computer program.

To tell a computer to do something, a program must be written to tell it exactly what to do and how to do it. If an algorithm has been designed, the computer program will follow this algorithm, step-by-step, which will tell the computer exactly what it should do.

What is a programming language?

A programming language is an artificial language that a computer understands. The language is made up of series of statements that fit together to form instructions. These instructions tell a computer what to do.

There are many different programming languages, some more complicated and complex than others. Among the most popular languages are:

- Python
- C#
- Java
- C++
- JavaScript
- BASIC

Different languages work in different ways. For example, in Python all instructions are written in lowercase, but in BASIC they tend to be written in uppercase.

Programming languages are designed to be easy for a human to understand and write in. However, a computer cannot run programs written in these languages directly. Most programming languages have to be translated into machine code before the computer can execute the instructions.

What is a program?

Programs are made up of statements that the programming language knows and understands.

Just as words are put together to form a sentence, a program puts one or more statements together to form an instruction. Each statement tells the computer to perform a specific task, and the instructions tell a computer what to do.

Statements

Different **programming languages** use different statements. A few of these are listed in this table:

Statement	Purpose
Print	Output a message on the screen
input	Get data from the user
if…else	A decision
while	A loop controlled by a decision
for	A loop controlled by a counter
def	Create a procedure or function

Examples

The following sentence asks someone to write a message on a whiteboard:

“Please write the words ‘Hello world!’ on the board.”

This sentence is an instruction, which contains a single statement. The statement is ‘write the words’. In **Python** (3.x), the equivalent statement is **print**.

In this manual, we shall use JAVA programming language.

2.2 GUIDELINES FOR DESIGNING COMPUTER PROGRAMS

(SO 2, AC 1, AC 2, AC 3, AC 4)

This manual assumes you're designing a standalone computer program that runs with a conventional GUI or command-line interface, but many of the techniques can also apply to programs that will become part of a bigger system.

A standalone program is one that is justified all by itself, like a word processor or a game, but even if it was a cog in a bigger system it'd still have the same qualities: it would focus on one job, it would take some kind of input that the system produces and transform it into some kind of output that the user or system consumes. Some examples of good standalone programs are:

- Address book
- Spreadsheet
- Calculator
- Trip planner
- Picture editor

Whereas a program designed to be part of a larger system can be something like:

- A program that imports purchase orders and saves them to a database
- A program that prints packing slips for orders stored in a database
- A web browser
- An email client
- An MP3 player

The first set of programs are all useful on their own, but the second set all need something else to complete them such as a web server or email server. Even the MP3 player needs a program that can create new MP3 files for it to play. The impact to you is that the second set of programs have to cooperate with some kind of protocol or file format that you didn't design yourself, and so they become part of your spec. In this tutorial I'm going to ignore that aspect so I can concentrate on the basics.

What is the difference between designing and coding?

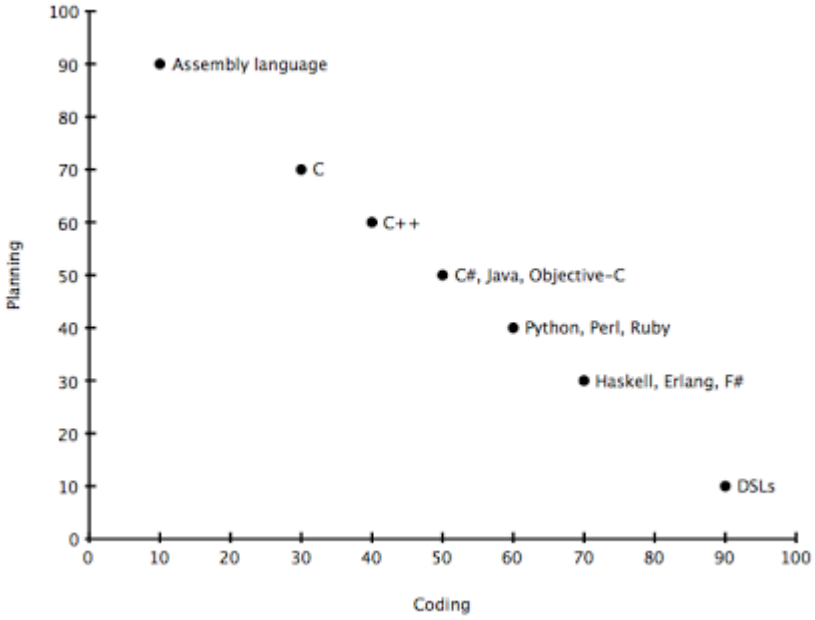
There is no difference between designing a program on paper and coding it, but code tends to be harder to understand and change. So the goal of the paper planning phase is to invent a pattern that helps you understand the code and isolate it from changes made in other parts of the program. In fact, **most of your time will be spent creating ways to isolate code**

from changes made to other code, and most of this tutorial will be about how you can do this one thing.

A popular metaphor for design versus coding is the architect who gives blueprints to the builder, and the builder represents the programmer. But it's an imperfect metaphor because it forgets the compiler, which does the same thing as the builder in the architect metaphor. If we revised the metaphor then the architect would have to go through three phases to design a bridge: come up with a way to organize the blueprints, then come up with a way to prevent changes in the design of the bolts or girders from forcing a redesign of the entire bridge, and *then* design the bridge.

In real life construction projects the builders also co-design the bridge along with the architect because they discover problems that the architect missed, even offering solutions for them based on their expertise as builders. Not just problems with the measurements and materials either, but problems with the landscape, weather and budget. With programming the compiler will be a co-designer because it'll also tell you about problems you missed. Not just problems with the syntax, but problems with the way you use *types*. Many modern compilers will go as far as to suggest ways to fix those problems based on the expertise of the programmers who made it.

This is why the choice of language has a big impact on how much design work you'll do on paper before you begin coding. The more type checking and static analysis the compiler does and the more expressive the language is then the easier it is to organize and isolate code. Imagine that languages could be placed on a graph like the one below. Here we see that assembly languages require the most planning in advance of coding, but DSLs (Domain Specific Languages) can be so specialized for their task that you can jump into coding almost right away.



The numbers on this graph have been exaggerated to illustrate the idea, but the gist is that every language aims to sit somewhere between intense planning and carefree coding. And even though it seems like it would be nice to operate at the bottom-right corner all the time, it isn't necessarily an advantage; one of the trade-offs of expressibility in any language--human or computer languages--is applicability. Every abstraction comes at a price that makes the language less appropriate for some other task.

Even if your problem lives at the bulls eye of a language's target there will still never be a language so expressive that you'll never need any kind of paper-planning. There will only ever be languages that make it safer to move into coding *earlier*.

How to think about the design of a program

All programs transform input into output, even if that input was hard-coded into the source at design-time. It's obvious enough to be forgotten and programmers can get so lost in the fuss of implementing features that they forget what the real purpose of the program was. Proper design can help you and your team mates avoid losing sight of the real goal.

The code that does the conversion is called the **business logic**. The business logic will act on data structures called the **model**. Your program will need to convert (or "deserialize") its input into the model, transform it, and then serialize the model into output. The model will be whatever makes it easiest to perform the transformation and does not need to resemble what the user thinks of the data as. For example, a calculator converts decimal numbers into binary, performs arithmetic on them there, and then converts the binary back into decimal. The binary representation of the decimal numbers is the calculator's model.

Everything else is called **overhead**, and that will be code that implements the user interface, sanitizes the input, creates connections to services, allocates and disposes of memory, implements higher-level data structures like trees and lists and so-on. One of your goals will be to separate the overhead from the business logic so that you can make changes to one without having to modify the other.

Some mistakes to avoid at this stage:

1. Don't create a class or module called "Business Logic" or similar. It shouldn't be a black-box. Later I'll discuss how to structure this code
2. Don't put business logic in a library[1]. You'll want the ability to change it more often than the overhead, and libraries tend to get linked to other programs--multiplying the consequences of changing it
3. Don't put business logic or overhead into the model unless it's only enforcing constraints. The code that defines the model should expose some interfaces that let you attach what you need. More on that later

Some of the things that you can include in the model is code that enforces meaningful relationships. So if your genealogy program has a "Person" class with an "Offspring" collection then you should include code in the model that prevents a person from being their own grandparent. But the function to discover cousin relationships in a family tree is *business logic* and doesn't belong in the model. Although it might make sense that you can't be your own cousin, and that this rule could be enforced in the model, you also need to draw a line between what enforcement is necessary to **prevent common bugs** and what's going to **create complex bugs** through mazes of logic that are difficult to follow.

1 - "Anything Is Appropriate As Long As You Know What you're Doing." One man's business logic is another man's overhead, and your project might be packaged as a library to be consumed by a bigger system. This rule is meant to apply to the point you're at in the food chain.

Top-Down versus Bottom-Up

Top-Down programming is about writing the business logic first in pseudocode or non-compiling code and then working down to the specifics, and it's the first program design technique taught in schools. Let's say that you're designing a coin-counting program, and you write what you want the perfect main () to look like:

```
void main()
{
    CoinHopper hopper = new CoinHopper(Config.CoinHopperHardwarePort);
    decimal deposit = 0;
    while (hopper.count > 0)
    {
        CoinType coin = hopper.next();
        deposit += coin.value;
    }

    decimal processingFee = deposit * 0.08;
    Output.Write("Total deposit: ${0}, processing fee: ${1}, net deposit: ${2}",
                 deposit,
                 processingFee,
                 deposit - processingFee);
}
```

The program doesn't compile yet, but you've now **defined the essence of the program in one place**. With top-down programming you start like this and then you focus on defining what's going on in the CoinHopper class and what abstraction is going on in the Output class.

Bottom-up programming is the natural opposite, where the programmer intuits that they'll need a way to control the coin counting hardware and begins writing the code for that before writing the code for main(). One of the reasons for doing it this way is to discover non-obvious requirements that have a major impact on the high-level design of the program. Let's say that when we dig into the nitty-gritty of the API for the hardware we discover that it doesn't tell us what type of coin is there, it just gives us sensory readings like the diameter and weight. Now we need to change the design of the program to include a module that identifies coins by their measurements.

What usually happens in the real-world is that all of the top-down design is done on paper and the actual coding goes bottom-up. Then when the coding effort falsifies an assumption made on paper the high-level design is easier to change.
"A further disadvantage of the top-down method is that, if an understanding of a fault is obtained, a simple fix, such as a new shape for the turbine housing, may be impossible to implement without a redesign of the entire engine." - from Richard Feynman's report of the Space Shuttle Challenger Disaster

High-level design patterns

"Design patterns" might have come into the programmer's lexicon around the time a book by the same name was published, but the concept has existed since the dawn of man. Given a problem, there are certain traditional ways of solving it. Having windows on two adjacent walls in a room is a design pattern for houses that ensures adequate light at any time of day. Or putting the courthouse on one side of a square park and shops around the other sides is another kind of design pattern that focuses a town's community. The culture of computer

programming has developed hundreds of its own design patterns: high level patterns that affect the architecture of the whole program--forcing you to chose them in advance--and low-level patterns that can be chosen later in development.

High-level patterns are baked into certain types of frameworks. The **Model-View-Controller** (MVC) pattern is ubiquitous for GUIs and web applications and nearly impossible to escape in frameworks like Cocoa, Ruby-on-Rails, and .Net, but that doesn't mean you have to use it for every program; some programs don't have to support random user interaction and wouldn't benefit from MVC.

MVC is well described elsewhere, so here are some of the other high-level patterns and what they're good for.

The 1-2-3

This is the first pattern you learned in programming class: the program starts, asks the user for input, does something with it, prints it out and halts. One-two-three. "Hello world" is a 1-2-3. It's the simplest pattern and still useful and appropriate in many situations. It's good for utilities, housekeeping tasks and one-off programs.

One of the biggest mistakes made by programmers is to get too ambitious. They start designing an MVC program, take weeks or months to code and debug it, but then it *dwarfs the magnitude of the problem*. If someone wants a program that doesn't need to take its inputs randomly and interactively, then produces a straightforward output, then it's a candidate for 1-2-3.

The Read-Execute-Print Loop (REPL)

REPL is the 1-2-3 kicked up a notch. Here the program doesn't halt after printing its output, it just goes back and asks for more input. Command-line interfaces are REPLs, and most interpreted programming languages come with a REPL wrapped around the interpreter. But if you give the REPL the ability to **maintain state** between each command then you have a powerful base to build simple software that can do complex things. You don't even need to implement a full programming language, just a simple "KEYWORD {PARAMETER}" syntax can still be effective for a lot of applications.

The design of a REPL program should keep the session's state separate from the code that interprets commands, and the interpreter should be agnostic to the fact that it's embedded in a REPL so that you can reuse it in other patterns. If you're using OOP then you should also create an Interface to abstract the output classes with, so that you aren't writing to STDOUT directly but to something like myDevice.Write(results). This is so you can easily adapt the program to work on terminals, GUIs, web interfaces and so-on.

The Pipeline

Input is transformed one stage at a time in a pipeline that runs continuously. It's good for problems that transform large amounts of data with little or no user interaction, such as consolidating records from multiple sources into a uniform store or producing filtered lists according to frequently changing rules. A news aggregator that reads multiple feed standards (like RSS and Atom), filters out dupes and previously read articles, categorizes and then distributes the articles into folders is a good candidate for the Pipeline pattern, especially if the user is expected to reconfigure or change the order of the stages.

You write your program as a series of classes or modules that share the same model and have a loop at their core. The business logic goes inside the loop and treats each chunk of data atomically--that is, independent of any other chunk. If you're writing your program on Unix then you can take advantage of the pipelining system already present and write your program as a single module that reads from stdin and writes to stdout.

Some language features that are useful for the Pipeline pattern are coroutines (created with the "yield return" statement) and immutable data types that are safe to stream.

The Workflow

This is similar to the Pipeline pattern but performs each stage through to completion before moving onto the next. It's good for when the underlying data type doesn't support streaming, when the chunks of data can't be processed independently of the others, when a high degree of user interaction is expected on each stage, or if the flow of data ever needs to go in reverse. The Workflow pattern is ideal for things like processing purchase orders, responding to trouble tickets, scheduling events, "Wizards", filing taxes, etc..

You'd wrap your model in a "Session" that gets passed from module to module. A useful language feature to have is an easy way to serialize and deserialize the contents of the session to-and-from a file on disk, like to an XML file.

The Batch Job

The user prepares the steps and parameters for a task and submits it. The task runs asynchronously and the user receives a notification when the task is done. It's good for tasks that take more than a few seconds to run, and especially good for jobs that run forever until cancelled. Google Alerts is a good example; it's like a search query that never stops searching and sends you an email whenever their news spider discovers something that matches your query. Yet even something as humble as printing a letter is also a batch job because the user will want to work on another document while your program is working on queuing it up.

Your model is a representation of the user's command and any parameters, how it should respond when the task has produced results, and what the results are. You need to encapsulate all of it into a self-contained data structure (like "PrintJobCriteria" and "PrintJobResults" classes that share nothing with any other object) and make sure your business logic has thread-safe access to any resources so it can run safely in the background.

When returning results to the user you must make sure you avoid any thread entanglement issues. Framework classes like BackgroundWorker are ideal for marshalling the results back onto the GUI.

Combining architectural patterns

You can combine high-level patterns into the same program, but you will need to have a very good idea of what the user's problem is because one model will always be more dominant than the others. If the wrong model is dominant, then it'll spoil the user's interaction with the program.

For example, it wouldn't be a good idea to make the Pipeline dominant and implement 1-2-3 in one of its modules because you'll force the user to interact excessively for each chunk of data that goes through the pipeline. This is exactly the problem with the way file management is done in Windows: you start a pipeline by copying a folder full of files, but an intermediate module wants user confirmation for each file that has a conflict. It's better to make 1-2-3 dominant and let the user define in advance what should happen to exceptions in the pipeline.

User Interface patterns

A complete discussion of UI patterns would be beyond the scope of this tutorial, but there are two meta-patterns that are common between all of them: **modal** and **modeless**. Modal means the user is expected to do one thing at a time, while modeless means the user can do anything at any time. A word processor is modeless, but a "Wizard" is modal. Regardless of what the underlying architectural pattern is, the UI needs to pick one of these patterns and stick to it.

Some programs can mix the two effectively if the task lends itself, like tax-filing software because it lets you jump to points randomly in the data entry stage, but doesn't let you move onto proofing and filing until the previous stage is complete.

Almost all major desktop software is modeless and users tend to expect it. To support this kind of UI there are two mechanisms supported by most toolchains; event-driven and data binding.

Event driven

Sometimes called "call-backs", but when they're called "Events" it's usually because they've been wrapped in a better abstraction. You attach a piece of code to a GUI control, and when the user does something to the control (click it, focus it, type in it, mouse-over it, etc.) your code is invoked by the control. The call-back code is called a *handler* and is passed a reference to the control and some details about the event.

Event-driven designs require a lot of code to examine the circumstances of the event and update the UI. The GUI is dumb, knows nothing about the data, needs to be spoon-fed the data, and passes the buck to your code whenever the user does something. This can be desirable when the data for the model is expensive to fetch, like when it's across the network or there's more than can fit in RAM.

Events are an important concept in business logic, overhead and models, too. The "PropertyChanged" event is very powerful when added to model classes, for example, because it fits so well with the next major UI mechanism below:

Data Binding

Instead of attaching code to a control you attach the *model* to the control and let a smart GUI read and manipulate the data directly. You can also attach Property Changed and Collection Changed events to the model so that changes to the underlying data are reflected in the GUI automatically, and manipulation on the model performed by the GUI can trigger business logic to run. This technique is the most desirable if your model's data can be retrieved quickly or fit entirely in RAM.

Data binding is complemented with conventional event-driven style; like say your address-book controls are bound directly to the underlying Address object, but the "Save" button invokes a classic Event handler that actually persists the data.

Data binding is extremely powerful and has been taken to new levels by technologies like Windows Presentation Foundation (WPF). A substantial portion of your program's behaviour can be declared in data bindings alone. For example: if you have a list-box that is bound to a collection of Customer objects then you can bind the detail view directly to the list-box's Selected Item property and make the detail view change whenever the selection in the list-box does, no other code required. A perfectly usable viewer program can be built by doing nothing more than filling the model from the database and then passing the whole show over to the GUI. Or like MVC without the C.

What it'll mean for the design of your program is greater emphasis on the model. You'll either design a model that fits the organization of the GUI, or build adaptors that transform your model into what's easiest for the GUI to consume. The style is part of a broader concept known as Reactive Programming.

Designing models

Remember that your data model should fit the solution, not the problem. Don't look at the definition of the problem and start creating classes for every noun that you see, and nor should you necessarily create classes that model the input or output--those may be easily convertible without mapping them to anything more sophisticated than strings.

Some of the best candidates for modelling first are actions and imaginary things, like transactions, requests, tasks and commands. A program that counts coins probably doesn't need an abstract Coin class with Penny, Nickel, Dime and Quarter dutifully sub classing it, but it should have a Session class with properties to record the start time, end time, total coin count and total value. It might also contain a Dictionary property to keep separate counts for each type of coin, and the data type representing the coin type may just end up being an Enum or string if that's all it really needs.

Once you've modelled the actions you can step forward with your business logic a little until you begin to see which nouns should be modelled with classes. For example, if you don't need to know anything about a customer except their name, then don't create a Customer class, just store their name as a string property of an Order or Transaction class[2].

2 - There can be a benefit to creating "wrappers" for scalar values for the sake of strong-typing them. In this case Customer would just store a name, but the fact that it's an instance of Customer means the value can't be accidentally assigned to a PhoneNumber property. But I wouldn't use this technique unless your language supports Generics (more below).

Develop the Logic

After you and the user agree to the goals and output of the program, the rest is up to you. Your job is to take that output definition and decide how to make a computer produce the output. You have taken the overall problem and broken it down into detailed instructions that the computer can carry out. This doesn't mean that you are ready to write the program—quite the contrary. You are now ready to develop the logic that produces that output.

The output definition goes a long way toward describing *what* the program is supposed to do. Now you must decide *how* to accomplish the job. You must order the details that you have so they operate in a time-ordered fashion. You must also decide which decisions your program must make and the actions produced by each of those decisions.

Throughout the rest of this 24-hour tutorial, you'll learn the final two steps of developing programs. You will gain insight into how programmers write and test a program after developing the output definition and getting the user's approval on the program's specifications.

CAUTION

Only after learning to program can you learn to develop the logic that goes into a program, yet you must develop some logic before writing programs to be able to move from the output and data definition stage to the program code. This "chicken before the egg" syndrome is common for newcomers to programming. When you begin to write your own programs, you'll have a much better understanding of logic development.

In the past, users would use tools such as *flowcharts* and *pseudocode* to develop program logic. A flowchart is shown in the diagram that follows It is said that a picture is worth a thousand words, and the flowchart provides a pictorial representation of program logic. The flowchart doesn't include all the program details but represents the general logic flow of the program. The flowchart provides the logic for the final program. If your flowchart is correctly drawn, writing the actual program becomes a matter of rote. After the final program is completed, the flowchart can act as documentation to the program itself.

The flowchart depicts the payroll program's logic graphically.

Flowcharts are made up of industry-standard symbols. Plastic flowchart symbol outlines, called *flowchart templates*, are still available at office supply stores to help you draw better-looking flowcharts instead of relying on freehand drawing. There are also some programs that guide you through a flowchart's creation and print flowcharts on your printer.

Although some still use flowcharts today, RAD and other development tools have virtually eliminated flowcharts except for depicting isolated parts of a program's logic for documentation purposes. Even in its heyday of the 1960s and 1970s, flowcharting did not completely catch on. Some companies preferred another method for logic description

called *pseudocode,* sometimes called *structured English,* which is a method of writing logic using sentences of text instead of the diagrams necessary for flowcharting.

Pseudocode doesn't have any programming language statements in it, but it also is not free-flowing English. It is a set of rigid English words that allow for the depiction of logic you see so often in flowcharts and programming languages. As with flowcharts, you can write pseudocode for anything, not just computer programs. A lot of instruction manuals use a form of pseudocode to illustrate the steps needed to assemble parts. Pseudocode offers a rigid description of logic that tries to leave little room for ambiguity.

Here is the logic for the payroll problem in pseudocode form. Notice that you can read the text, yet it is not a programming language. The indention helps keep track of which sentences go together. The pseudocode is readable by anyone, even by people unfamiliar with flowcharting symbols.

```
For each employee:
 If the employee worked 0 to 40 hours then
 net pay equals hours worked times rate.
 Otherwise,
 if the employee worked between 40 and 50 hours then
  net pay equals 40 times the rate;
  add to that (hours worked -40) times the rate times 1.5.
 Otherwise,
  net pay equals 40 times the rate;
  add to that 10 times the rate times 1.5;
  add to that (hours worked -50) times twice the rate.
 Deduct taxes from the net pay.
Print the paycheck.
```

Writing the Code

The program writing takes the longest to learn. After you learn to program, however, the actual programming process takes less time than the design if your design is accurate and complete. The nature of programming requires that you learn some new skills. The next few hourly lessons will teach you a lot about programming languages and will help train you to become a better coder so that your programs will not only achieve the goals they are supposed to achieve, but also will be simple to maintain.

2.3 GUIDELINES FOR THE CREATION OF COMPUTER PROGRAMS

(SO 3, AC 1, AC 2, AC 3)

There are certain guidelines that apply when creating computer programs. This is irrespective of the programming language used. We shall look at some of them;

- The creation must include coding from the program design documents. That is, the program must meet all the design specifications in accordance with company standards or industry standards for the selected language.
- The names created in the program must describe the purpose of the items named.
- The creation must include conformance with design documentation and any differences must be documented with any reasons for deviations. Conformance relates to 2 key elements i.e performance and maintainability.

Performance

Performance seems to have two meanings:

1. The speed at which a computer program operates, either theoretically (for example, using a formula for calculating Mtops - millions of theoretical operations per second) or by counting operations or instructions performed (for example, (MIPS) - millions of instructions per second) during a benchmark test. The benchmark test usually involves some combination of work that attempts to imitate the kinds of work the computer program does during actual use. Sometimes performance is expressed for each of several different benchmarks.
2. The total effectiveness of a computer system, including throughput, individual response time, and availability.

Maintainability

Program maintainability is defined as the degree to which an application is understood, repaired, or enhanced. Program maintainability is important because it is approximately 75% of the cost related to a project! How many resources are you dedicating to supporting code that could be deleted or improved?

Program maintainability requires more developer effort than any other phase of the development life cycle. A program with these qualities will require increased programming effort:

- Poor Code Quality
- Source Code Defects
- Undetected Vulnerabilities

- Excessive Technical Complexity
- Large Systems
- Poorly Documented Systems
- Excessive Dead Code

2.4 UNDERSTANDING JAVA

Java is a programming language built for the age of the Internet. It was built for a world in which everything that has some sort of electronic component: stereo systems, wireless phones, cars, even your refrigerator, are all on the Internet. This world is right around the corner.

As a language, Java is closely related to C++, which is also object-oriented but retains a lot of idiosyncrasies inherited from its predecessor language C. Java has removed the inconsistent elements from C++, is exclusively object-oriented, and can be considered a modern version of C++.[1] Because of its logical structure Java has quickly become a popular choice as a teaching language,[2] and because of its extensive Internet support and the promise of writing programs once and using them on every operating system Java is becoming more and more accepted in industry.

What is so unique about Java that has propelled its rapid, wide acceptance?

- It's available on more devices world-wide than any other language. Notice that I say devices - not just computers. Java is currently being used not only on mainframe systems in the enterprise and personal computers in the office and at home - it's also running in cellphones.
- It was carefully designed to eliminate many of the most common causes of programming errors - bugs. Java programs that compile error-free tend to work! Strong data typing and complete memory management are two features that make this possible.
- It provides for secure programs that can be executed on the Internet without worry of them infecting your system with some virus or planting a Trojan horse.

WHAT IS A JAVA PROGRAM AND HOW DO I CREATE ONE?

Let's look at what makes up a Java program. A Java program is built by writing (and referencing already available) things called *classes*. In the simplest sense, a Java program is a bunch of classes. You will construct at least one, typing its source code into a file.

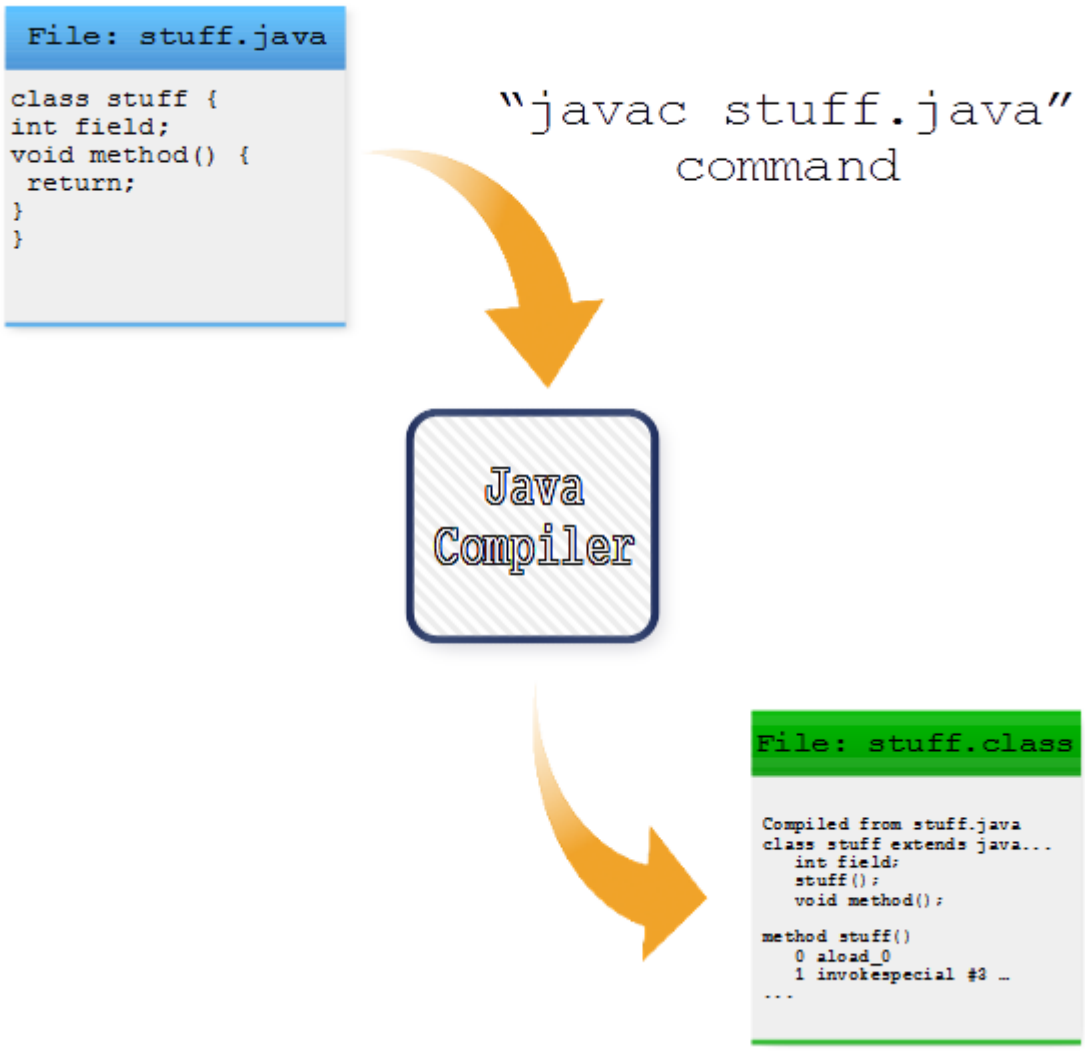
The stuff you will enter (text) has a very specific structure (its **syntax**) that the Java compiler expects. You create your Java programming language files with an editor that is available on

[1] Java does have disadvantages. For example, programs written in Java are generally slower than those in C++ and it is difficult to accomplish system-level tasks in Java.

[2] Java compilers and tools are available for free, an important consideration for academic and student budgets.

your computer. On a PC running Windows, Wordpad or Notepad will work just fine. On a Sun workstation, textedit is a nice editor.

Once you have some complete Java source code in a file, you compile it. The Java compiler turns your file full of characters into another file which contains instructions that a JVM (Java Virtual Machine) can interpret, a ".class" file.



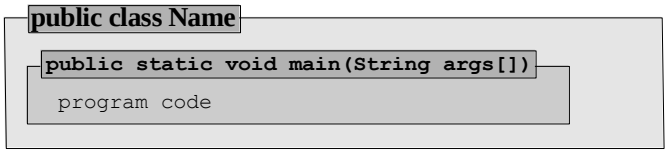
BASIC JAVA PROGRAMMING GUIDELINES

Every Java program must follow these guidelines:

- *Java is case sensitive, i.e. the word* Program *is different from* program*.*
- *Curly brackets* { *and* } *are used to group statements together.*
- *An executable Java program must contain at least the following lines as a framework:*

```
public class Name
{  public static void main(String args[])
   {  ... program code ...
   }
}
```

- *Every statement whose next statement is not a separate group must end in a semicolon.*
- *A Java program containing the above framework must be saved using the filename* Name.java*, where* Name *(including correct upper and lower cases) is the word that follows the keywords* public class *and the file extension is* .java*.*



In other words, to create a Java program you first create a text file containing the lines

```
public class Name
{
   public static void main(String args[])
   {
  ... more lines ...
   }
}
```

The file containing our code is called the source code file.

Source Code

A Java source code file is a text file that contains programming code written according to the Java language specifications, resembling a mixture of mathematical language

and English. A computer cannot execute source code, but humans can read and understand it.

Java source code files should be saved as Name.java*, where* Name *is the name that appears in the first line of the program:* public class Name*. That* Name *is referred to as the name of the class, or program. By convention its first letter is capitalized.*

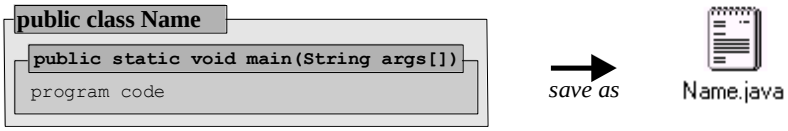


Figure: Saving a Java source code file

Here is an example of a Java source code file. We will later explain what the various lines mean; for now it is simply a text file that looks as shown.

Example: The first source code file

Create a source code file containing the necessary Java code to get the computer to write "Hi – this is my first program" on the screen.

Our first Java program looks as follows:

```
public class Test
{
  public static void main(String args[])
  {
     System.out.println("Hi – this is my first program");
  }
}
```

This program, or class, is called Test and must be saved under the file name Test.java.

Compiling a Java Program or Class

A source code file, which is more or less readable in plain English, needs to be transformed into another format before the computer can act upon it. That translation process is called *compiling* and is accomplished using the Java compiler javac from the Java Developer's Kit (JDK), which could be invoked by an IDE such as BlueJ.

Compiling

Compiling is the process of transforming the source code file into a format that the computer can understand and process. The resulting file is called the byte-code, or class, file. The name of the class file is the same as the name of the program plus the extension .class*. The program* javac *from the Java Developer's Kit is used to transform a source code file into a class file.*

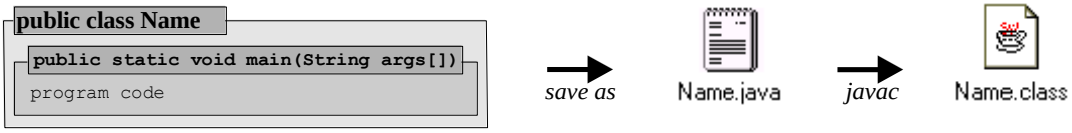


Figure: Compiling and creating class file

If a source code contains any errors, they are flagged by the compiler. You need to fix them and re-compile until there are no further errors.

Tip: In case of an error, the javac compiler shows the line number and position of where *it thinks* the error occurred in your source code.

- If the compiler points out an error, then there is an error *at or before* the indicated position.
- If the compiler reports a certain number of errors, than this is *the least* amount of errors.
- If one error is fixed, other errors may automatically disappear or new ones may appear.

Fix your source code a few errors at a time. Recompile often to see if the number of errors and the error messages change until no errors are reported. If you can not find an error at the position indicated by the compiler, look at the code *before* that position.

Executing a Java Program or Class

The Java compiler does not produce an executable file, so Java programs can not execute under the operating system of your machine. Instead they execute inside a *Java Virtual Machine*, which is invoked using the java program of the JDK.

Executing a Class File

To execute a Java program the Java Developer's Kit provides a program called java*. When executing that program with your class file as parameter the following happens:*

- *the Java Virtual Machine (JVM) is created inside your computer*
- *the JVM locates and reads your class files*
- *the JVM inspects your class file for any security violations*
- *the JVM executes, or interprets, your class file according to its instructions if possible*

Under Windows and Unix, execute a program by typing at the command prompt java Name*, where* Name *is the name of the program (no extension). On a Macintosh, double-click the* java *icon and select the appropriate class file.*

Most IDE's allow for a convenient way to execute a file. In BlueJ you right-click on a compiled class and select the "main" method.

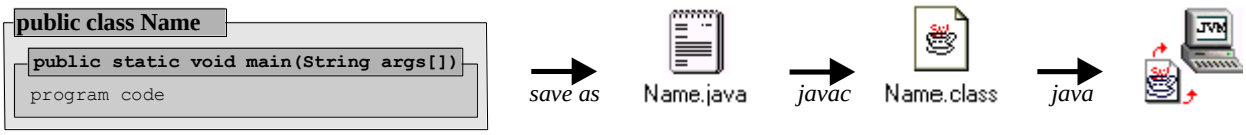


Figure: Executing a class file

A good question at this point is which line in a Java program executes *first*.

Default Program Entry Point

The default program entry point is that part of a class (or program) where execution begins. For every Java class (or program), the standard program entry point consists of the line:

public static void main(String args[])

If that line is not present in your source code, the JVM can not execute your program and displays an error message.

At this point, we need to explain what the Java Virtual Machine is and how it relates to the operating system and to Java class files.

Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is a platform-independent engine used to run Java applets and applications. The JVM knows nothing of the Java programming language, but it does understand the particular file format of the platform and implementation independent class file produced by a Java compiler. Therefore, class files produced

by a Java compiler on one system can execute without change on any system that can invoke a Java Virtual Machine.[3]

When invoked with a particular class file, the JVM loads the file, goes through a verification process to ensure system security, and executes the instructions in that class file.

The JVM, in other words, forms a layer between the operating system and the Java program that is trying to execute. That explains how *one* Java program can run without change on a variety of systems: it can not! A Java program runs on only *one* system, namely the Java Virtual Machine. That virtual system, in turn, runs on a variety of operating systems and is programmed quite differently for various systems. To the Java programmer, it provides a unified interface to the actual system calls of the operating system.[4]

You can include graphics, graphical user interface elements, multimedia, and networking operations in a Java program and the JVM will negotiate the necessary details between the class file(s) and the underlying operating system. The JVM produces exactly the same results – in theory – regardless of the underlying operating system. In the Basic (or C, or C++) programming language, for example, you can create code that specifies to multiply two integers 1000 and 2000 and store the result as another integer. That code works fine on some systems, but can produce negative numbers on others.[5] In Java, this can not happen: either the code fails on all platforms, or it works on all platforms.

[3] In general, the Java Virtual Machine is an abstractly specified class file interpreter that can be realized by different software makers. The JVM that comes with the JDK was created by SUN Microsystems, but any other JVM is also able to run the same class files. Different JVM's can vary in efficiency, but they all must run the same class files.

[4] This is somewhat similar to old Basic programs: a simple Basic program can run on virtually any system that has a Basic interpreter installed since the interpreter mediates between the program trying to run and the operating system.

[5] Programming languages have a *largest possible* integer whose value can differ on different systems. A C++ program executing on a machine with a largest integer bigger than 2,000,000 produces the correct result, but on a system where the largest integer is, say, 32,767 it fails. The JVM has the *same* largest integer on every platform.

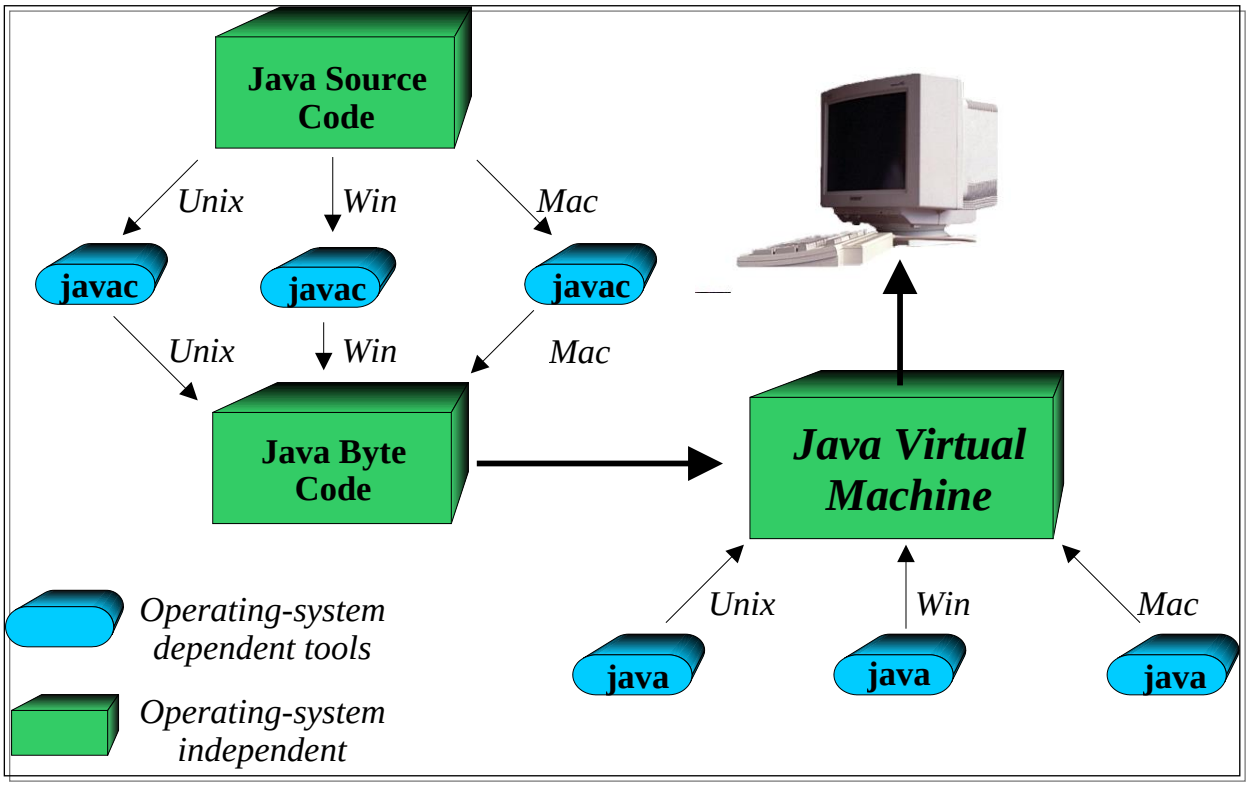


Figure 1.09: Illustrating the machine dependent/independent parts of Java programs

Because the JVM is in effect its own computer, it can shield the actual computer it is running on from potentially harmful effects of a Java program. This is especially important because Java programs known as *applets* can *automatically* start executing on your machine when you are surfing the web if the appropriate feature of your web browser is enabled. If these programs were allowed to meddle with your system, you could accidentally execute a program that would proceed to erase your entire disk. That, of course, would prompt people to disable Java on their web browser, which in turn would be bad news for anyone who supports the Java concept.

Basic Data Types:

Primitive Java Data Types

Java supports the following primitive, or basic, data types:

- **int**, long, *or* short — *to represent integer numbers*
- **double** *or* float — *to represent decimal numbers*
- **char** — *to represent character values*
- **boolean** — *to represent logical values*
- **void** — *to represent "no type"*

Each numeric type has a largest and smallest possible value, as indicated in table 1.10.[6]

Most programs use int for integers and double for decimal numbers, while long, short, and float are needed only in special situations.

Type	Range	
double	largest positive/negative value:	±1.7976931348623157E308
	smallest non-zero value:	±4.9E-324
	significant digits:	16 digits after decimal point
float	largest positive/negative value:	±3.4028235E38
	smallest non-zero value:	±1.4E-45
	significant digits:	8 digits after decimal point
int	largest value	2147483647
	smallest value:	-2147483648
short	largest value	32767
	smallest value:	-32768
long	largest value	9223372036854775807
	smallest value:	-9223372036854775808

Table: Ranges for valid decimal types

Each type can contain values called literals or unnamed constants in a particular format.

Literals

Literals are constant values for the basic data types. Java supports the following literals:

- int, short: *digits only, with possible leading plus (+) or minus (-) sign*
- long: *like* int *literals, but must end with an "*L*"*
- double: *digits including possible periodic point or leading plus (+) or minus (-) sign, or numbers in scientific notation* #.################E±###,[7] *where each # represents a digit*
- float: *like* double *literals, but must end with an "*F*"*
- char: *Single Unicode characters enclosed in single quotes, including the special control sequences described in table 1.11*[8]

[6] Java also supports a basic type `byte`, which we do not need currently, and another very useful type `String`, introduced later

[7] For example, the `double` number `1.23456E002 = 1.234562 = 123.456`

- boolean: true *or* false

In addition, Java has an object literal called null *for object references.*

Character literals include the following special characters called control sequences:

Control Sequence	Meaning	Control Sequence	Meaning
\n	new line	\t	tab character
\b	backspace	\r	return
\f	form feed	\\	backslash
\'	single quote	\"	double quote

Table: Common character control sequences

The ranges for the numeric types are the same, regardless of the underlying operating system (after all, programs run under the JVM, not the native operating system). In languages such as C or C++ an integer sometimes has a range similar to a Java short, and sometimes that of a Java int, depending on the underlying operating system, which can cause different results if the same program runs on different systems.

To use the basic data types to store information, we must define variables that have one of these types:

Declaration of Variables

To declare a variable that can store data of a specific type, the syntax:

```
type varName [, varName2,..., varNameN];
```

is used, where type *is one of the basic data types,* varName *is the name of the variable, and* varName2, ..., varNameN *are optional additional variables of that type. Variables can be declared virtually anywhere in a Java program.*

Variables must have a name and there are a few rules to follow when choosing variable names:

Valid Names for Variables

A variable name must start with a letter, a dollar sign '$', or the underscore character '_', followed by any character or number. It can not contain spaces. The reserved

[8] Unicode characters support characters in multiple languages and are defined according to their "Unicode Attribute table" (see `http://www.unicode.org/`). Every character on a standard US keyboard is a valid Unicode character.

keywords listed in the table below can not be used for variable names. Variable names are case-sensitive.

Java Reserved Keywords					
Abstract	boolean	break	byte	case	catch
Char	Class	const	continue	default	do
Double	Else	extends	false	final	finally
Float	For	goto	if	implements	import
instanceof	Int	interface	long	native	new
null	Package	private	protected	public	return
short	Static	super	switch	synchronized	this
throw	Throws	transient	true	try	void
volatile	While				

Table 1.12: Reserved keywords in Java

Example: Declaring variables

Declare one variable each of type int, double, char, and two variables of type boolean.

This is an easy example. We declare our variables as follows:

```
int anInteger;
double aDouble;
char aChar;
boolean aBoolean, anotherBoolean;
```

Assigning a Value to a Variable

To assign a value to a declared variable, the assignment operator "=" is used:

```
varName = [varName2 = ... ] expression;
```

Assignments are made by first evaluating the expression on right, then assigning the resulting value to the variable on the left. Numeric values of a type with smaller range are compatible with numeric variables of a type with larger range (compare table 13). Variables can be declared and assigned a value in one expression using the syntax:

```
type varName = expression [, varname2 = expression2, ...];
```

The assignment operator looks like the mathematical equal sign, but it is different. For example, as a mathematical expression

$$x = 2x + 1$$

is an equation which can be solved for *x*. As Java code, the same expression

```
x = 2*x + 1;
```

means to first evaluate the right side 2*x + 1 by taking the current value of x, multiplying it by 2, and adding 1. Then the resulting value is stored in x (so that now x has a new value).

Value and variable types must be compatible with each other, as shown in table 1.13.

Value Type	Compatible Variable Type
double	double
int	int, double
char	char, int, double
boolean	boolean

Table 1.13: Value types compatible to variable types

Example: Declaring variables and assigning values

Declare an int, three double, one char, and one boolean variable. Assign to them some suitable values.

There are two possible solutions. Variables can be declared first and a value can be assigned to them at a later time:

```
int anInteger;
double number1, number2, number3;
anInteger = 10;
number1 = number2 = 20.0;
number3 = 30;

char cc;
cc = 'B';
boolean okay;
okay = true;
```

Alternatively, variables can be declared and initialized in one statement:

```
int anInteger = 10;
double number1 = 20.0, number2 = 20.0, number3 = 30;
char cc = 'B';
boolean okay = true;
```

Software Engineering Tip: Variables serve a purpose and the name of a variable should reflect that purpose to improve the readability of your program. Avoid one-letter variable names[9]. Do not reuse a variable whose name does not reflect its purpose.

Whenever possible, assign an initial value to every variable at the time it is declared. If a variable is declared without assigning a value to it, all basic types except boolean are automatically set to 0, boolean is set to false, and all other types are set to null.[10]

Declare variables as close as possible to the code where they are used. Do not declare all variables at once at the beginning of a program (or anywhere else).

Example: Using appropriate variable names

The code segment below computes the perimeter of a rectangle and the area of a triangle. Rewrite that segment using more appropriate variable names and compare the readability of both segments:

```
double x, y, z;
x = 10.0;
y = 20.0;
z = 2*(x + y);
w = 0.5 * x * y;
```

This code computes the perimeter z of a rectangle with width x and length y and the area w of a triangle with base x and height y, so the variable names should reflect that. In addition, variables should be assigned a value when they are declared, so the code segment should be rewritten as follows:

```
double width = 10.0, height = 20.0;
```

[9] If a variable serves a minor role in a code segment (such as a counter in a loop) it can be declared using a one-letter variable name.

[10] The compiler may display an error message if it encounters variables that are not explicitly initialized.

```
double perimeterOfRectangle = 2*(width + height);
double base = width;
double areaOfTriangle = 0.5 * base * height;
```

It is immediately clear that the formulas used are correct. Choosing appropriate variable names clarifies the code significantly and makes it easy to locate potential problems.

Basic Arithmetic for Numeric Types

> *Java support the basic arithmetic operators + (addition), - (subtraction), * (multiplication), / (division), and % (remainder after integer division) for numeric variables and literals. The order of precedence is the standard one from algebra and can be changed using parenthesis.*
>
> *Each operator has a left and right argument and the type of the result of the computation is determined according to the rules outlined in the table below*

Left Argument	Right Argument	Result
int	int	int
int	double	double
double	int	double
double	double	double

Table: Resulting types of the basic arithmetic operations

Example: A Temperature Conversion Program

Create a complete program that converts a temperature from degrees Fahrenheit in degrees Celsius. Use comments to explain your code.

```
public class Converter
{
  public static void main(String args[])
  {
    // Printing out a welcoming message
    System.out.println("Welcome to my Temperature Converter.");
    System.out.println("\nProgram to convert Fahrenheit to Celcius.\n");

    // Defining the temperature value in Fahrenheit
    double temp = 212.0;
```

```
    // Applying conversion formula and storing answer in another variable
    double convertedTemp = 5.0 / 9.0 * (temp - 32.0);

    // Printing out the complete answer
    System.out.println(temp + " Fahrenheit = " + convertedTemp + " Celcius.");
    }
}
```

2.5 COMPILING THE PROGRAM - JAVA

A program has to be converted to a form the Java VM can understand so any computer with a Java VM can interpret and run the program. Compiling a Java program means taking the programmer-readable text in your program file (also called source code) and converting it to byte codes, which are platform-independent instructions for the Java VM.

The Java compiler is invoked at the command line on Unix and DOS shell operating systems as follows:

```
javac ExampleProgram.java
```

Note: Part of the configuration process for setting up the Java platform is setting the class path. The class path can be set using either the -classpath option with the javac compiler command andjava interpreter command, or by setting the CLASSPATH environment variable. You need to set the class path to point to the directory where the ExampleProgram class is so the compiler and interpreter commands can find it.

Interpreting and Running the Program

Once your program successfully compiles into Java bytecodes, you can interpret and run applications on any Java VM, or interpret and run applets in any Web browser with a Java VM built in such as Netscape or Internet Explorer. Interpreting and running a Java program means invoking the Java VM byte code interpreter, which converts the Java byte codes to platform-dependent machine codes so your computer can understand and run the program.

The Java interpreter is invoked at the command line on Unix and DOS shell operating systems as follows:

```
java ExampleProgram
```

At the command line, you should see:

```
I'm a Simple Program
```

Here is how the entire sequence looks in a terminal window:

Common Compiler and Interpreter Problems

Common Error Messages on Microsoft Windows Systems

`'javac' is not recognized as an internal or external command, operable program or batch file`

If you receive this error, Windows cannot find the compiler (`javac`).

Here's one way to tell Windows where to find `javac`. Suppose you installed the JDK in `C:\jdk1.8.0`. At the prompt you would type the following command and press Enter:

```
C:\jdk1.8.0\bin\javac HelloWorldApp.java
```

If you choose this option, you'll have to precede your `javac` and `java` commands with `C:\jdk1.8.0\bin\` each time you compile or run a program. To avoid this extra typing, consult the section Updating the PATH variable in the JDK 8 installation instructions.

`Class names, 'HelloWorldApp', are only accepted if annotation processing is explicitly requested`

If you receive this error, you forgot to include the `.java` suffix when compiling the program. Remember, the command is `javac HelloWorldApp.java` not `javac HelloWorldApp`.

Common Error Messages on UNIX Systems

`javac: Command not found`

If you receive this error, UNIX cannot find the compiler, `javac`.

Here's one way to tell UNIX where to find `javac`. Suppose you installed the JDK in `/usr/local/jdk1.8.0`. At the prompt you would type the following command and press Return:

```
/usr/local/jdk1.8.0/javac HelloWorldApp.java
```

Note: If you choose this option, each time you compile or run a program, you'll have to precede your `javac` and `java` commands with `/usr/local/jdk1.8.0/`. To avoid this extra typing, you could add this information to your PATH variable. The steps for doing so will vary depending on which shell you are currently running.

`Class names, 'HelloWorldApp', are only accepted if annotation processing is explicitly requested`

If you receive this error, you forgot to include the `.java` suffix when compiling the program. Remember, the command is `javac HelloWorldApp.java` not `javac HelloWorldApp`.

Syntax Errors (All Platforms)

If you mistype part of a program, the compiler may issue a *syntax* error. The message usually displays the type of the error, the line number where the error was detected, the code on that line, and the position of the error within the code. Here's an error caused by omitting a semicolon (`;`) at the end of a statement:

```
testing.java:14: `;' expected.
System.out.println("Input has " + count + " chars.")
                                                    ^
1 error
```

Sometimes the compiler can't guess your intent and prints a confusing error message or multiple error messages if the error cascades over several lines. For example, the following code snippet omits a semicolon (`;`) from the bold line:

```
while (System.in.read() != -1)
    count++
System.out.println("Input has " + count + " chars.");
```

When processing this code, the compiler issues two error messages:

```
testing.java:13: Invalid type expression.
        count++
              ^
testing.java:14: Invalid declaration.
    System.out.println("Input has " + count + " chars.");
                  ^
2 errors
```

The compiler issues two error messages because after it processes `count++`, the compiler's state indicates that it's in the middle of an expression. Without the semicolon, the compiler has no way of knowing that the statement is complete.

If you see any compiler errors, then your program did not successfully compile, and the compiler did not create a `.class` file. Carefully verify the program, fix any errors that you detect, and try again.

Semantic Errors

In addition to verifying that your program is syntactically correct, the compiler checks for other basic correctness. For example, the compiler warns you each time you use a variable that has not been initialized:

```
testing.java:13: Variable count may not have been initialized.
        count++
        ^
testing.java:14: Variable count may not have been initialized.
    System.out.println("Input has " + count + " chars.");
                                  ^
2 errors
```

Again, your program did not successfully compile, and the compiler did not create a `.class` file. Fix the error and try again.

Runtime Problems

Error Messages on Microsoft Windows Systems

`Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp`

If you receive this error, `java` cannot find your bytecode file, `HelloWorldApp.class`.

One of the places `java` tries to find your `.class` file is your current directory. So if your `.class` file is in `C:\java`, you should change your current directory to that. To change your directory, type the following command at the prompt and press Enter:

```
cd c:\java
```

The prompt should change to `C:\java>`. If you enter `dir` at the prompt, you should see your `.java` and `.class` files. Now enter `java HelloWorldApp` again.

If you still have problems, you might have to change your CLASSPATH variable. To see if this is necessary, try clobbering the classpath with the following command.

```
set CLASSPATH=
```

Now enter `java HelloWorldApp` again. If the program works now, you'll have to change your CLASSPATH variable. To set this variable, consult the Updating the PATH variable section in the JDK 8 installation instructions. The CLASSPATH variable is set in the same manner.

`Could not find or load main class HelloWorldApp.class`

A common mistake made by beginner programmers is to try and run the `java` launcher on the `.class` file that was created by the compiler. For example, you'll get this error if you try to run your program with `java HelloWorldApp.class` instead of `java HelloWorldApp`. Remember, the argument is the *name of the class* that you want to use, *not* the filename.

`Exception in thread "main" java.lang.NoSuchMethodError: main`

The Java VM requires that the class you execute with it have a `main` method at which to begin execution of your application. A Closer Look at the "Hello World!" Application discusses the `main` method in detail.

Error Messages on UNIX Systems

`Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp`

If you receive this error, `java` cannot find your bytecode file, `HelloWorldApp.class`.

One of the places `java` tries to find your bytecode file is your current directory. So, for example, if your bytecode file is in `/home/jdoe/java`, you should change your current directory to that. To change your directory, type the following command at the prompt and press Return:

```
cd /home/jdoe/java
```

If you enter `pwd` at the prompt, you should see `/home/jdoe/java`. If you enter `ls` at the prompt, you should see your `.java` and `.class` files. Now enter `java HelloWorldApp` again.

If you still have problems, you might have to change your CLASSPATH environment variable. To see if this is necessary, try clobbering the classpath with the following command.

```
unset CLASSPATH
```

Now enter `java HelloWorldApp` again. If the program works now, you'll have to change your CLASSPATH variable in the same manner as the PATH variable above.

`Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp/class`

A common mistake made by beginner programmers is to try and run the `java` launcher on the `.class` file that was created by the compiler. For example, you'll get this error if you try to run your program with `java HelloWorldApp.class` instead of `java HelloWorldApp`. Remember, the argument is the *name of the class* that you want to use, *not* the filename.

`Exception in thread "main" java.lang.NoSuchMethodError: main`

The Java VM requires that the class you execute with it have a `main` method at which to begin execution of your application. A Closer Look at the "Hello World!" Application discusses the `main` method in detail.

Applet or Java Web Start Application Is Blocked

If you are running an application through a browser and get security warnings that say the application is blocked, check the following items:

Verify that the attributes in the JAR file manifest are set correctly for the environment in which the application is running. The Permissions attribute is required. In a NetBeans project, you can open the

manifest file from the Files tab of the NetBeans IDE by expanding the project folder and double-clicking `manifest.mf`.
Verify that the application is signed by a valid certificate and that the certificate is located in the Signer CA keystore.
If you are running a local applet, set up a web server to use for testing. You can also add your application to the exception site list, which is managed in the Security tab of the Java Control Panel.

Code Comments

Code comments are placed in source files to describe what is happening in the code to someone who might be reading the file, to comment-out lines of code to isolate the source of a problem for debugging purposes, or to generate API documentation. To these ends, the Java language supports three kinds of comments: double slashes, C-style, and doc comments.

Double Slashes

Double slashes (//) are used in the C++ programming language, and tell the compiler to treat everything from the slashes to the end of the line as text.

```
//A Very Simple Example
class ExampleProgram {
  public static void main(String[] args){
    System.out.println("I'm a Simple Program");
  }
}
```

C-Style Comments

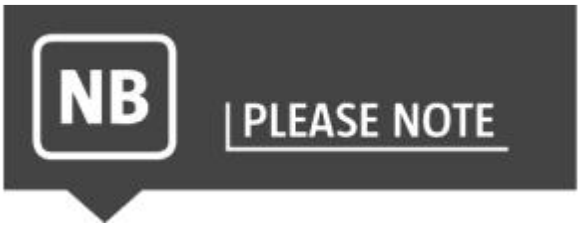
Instead of double slashes, you can use C-style comments (/* */) to enclose one or more lines of code to be treated as text.

```
/* These are
C-style comments
*/
class ExampleProgram {
  public static void main(String[] args){
    System.out.println("I'm a Simple Program");
  }
}
```

Doc Comments

To generate documentation for your program, use the doc comments (/** */) to enclose lines of text for the javadoc tool to find. The javadoc tool locates the doc comments embedded in source files and uses those comments to generate API documentation.

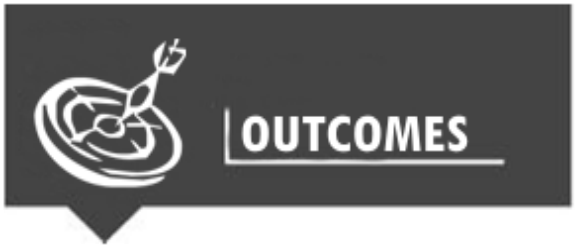
```
/** This class displays a text string at
*  the console.
*/
class ExampleProgram {
  public static void main(String[] args){
    System.out.println("I'm a Simple Program");
  }
}
```



To have a good understanding of JAVA you need to read the following textbook.

Prescribed Learning Material /	**Author**	**Publisher**	**Supplier**	**Price per book/ per page (Vat excluded)**
Concurrent Programming in Java: Design Principles and Patterns (Java Series)	Douglas Lea	Addison-Wesley Pub (SD)	ISBN-13: 978-0201695816	R350

SESSION 3: TESTING A COMPUTER PROGRAM



On completion of this section you will be able to test a computer program against the business requirements.

1. The testing includes assessment of the need to develop a testing program to assist with stress testing
2. The testing includes the planning, developing and implementing a test strategy that is appropriate for the type of program being tested.
3. The testing includes the recording of test results that allow for the identification and validation of test outcomes.

3.1 INTRODUCTION

Some experts insist that a well-designed program can be written correctly the first time. In fact, they assert that there are mathematical ways to prove that a program is correct. However, the imperfections of the world are still with us, so most programmers get used to the idea that their newly written programs probably have a few errors. This is a bit discouraging at first, since programmers tend to be precise, careful, detail-oriented people who take pride in their work. Still, there are many opportunities to introduce mistakes into programs, and you, just as those who have gone before you, will probably find several of them.

Eventually, after coding the program, you must prepare to test it on the computer. This step involves these phases:

- **Desk-checking.** This phase, similar to proofreading, is sometimes avoided by the programmer who is looking for a shortcut and is eager to run the program on the computer once it is written. However, with careful desk-checking you may discover several errors and possibly save yourself time in the long run. In desk-checking you simply sit down and mentally trace, or check, the logic of the program to attempt to ensure that it is error-free and workable. Many organizations take this phase a step further with a walkthrough, a process in which a group of programmers-your peers-review your program and offer suggestions in a collegial way.
- **Translating.** A translator is a program that (1) checks the syntax of your program to make sure the programming language was used correctly, giving you all the syntax-error messages, called diagnostics, and (2) then translates your program into a form the computer can understand. A by-product of the process is that the translator tells you if you have improperly used the programming language in some way. These types of mistakes are called syntax errors. The translator produces descriptive error messages. For instance, if in FORTRAN you mistakenly write N=2 *(I+J))-which has two closing parentheses instead of one-you will get a message that says, "UNMATCHED PARENTHESES." (Different translators may provide different wording for error messages.) Programs are most commonly translated by a *compiler*. A compiler translates your entire program at one time. The translation involves your original program, called a source module, which is transformed by a compiler into an object module. Prewritten programs from a

system library may be added during the link/load phase, which results in a load module. The load module can then be executed by the computer.

- **Debugging.** A term used extensively in programming, debugging means detecting, locating, and correcting bugs (mistakes), usually by running the program. These bugs are logic errors, such as telling a computer to repeat an operation but not telling it how to stop repeating. In this phase you run the program using test data that you devise. You must plan the test data carefully to make sure you test every part of the program.

What do we test for?

The first test of a system is to see whether it produces correct outputs. After this a variety of other tests are performed:

1. Online response:
   Online system must have a response time that will not cause problems to the users. One way to test is to input transactions on as many CRT screens s would normally be used in peak hours and time the response to each outline function to establish the performance level.
2. Volume:
   In this test, we create as many records as would normally be produced to verify that the hardware and software will function correctly. The users are asked to provide test data for volume testing.
3. Stress testing:
   The purpose of stress testing is to prove that the candidate system does not malfunction under peak loads. Unlike volume testing, where time is not a factor, here we subject the system to a high volume of data over a short period of time.
4. Recovery and security:
   A forced system failure is induced to test a backup recovery procedure for file integrity. Inaccurate data are entered into the system to see how the system responds in terms of error detection and protection. Another test is performed to check whether that data and programs are secured from unauthorized access.
5. Usability documentation and procedure:
   The usability test verifies the user-friendly nature of the system. This relates to normal operating and error handling procedures. For example the user is asked to use only the documentation and procedure as a guide to determine whether the system can system can run smoothly. One aspect of user friendliness is accurate and complete documentation.

3.2 DEVELOPING A TESTING PROGRAM

A **test case** is an individual set of **variables** or conditions that is used to see if features of the **software** work as expected. In the **test plan**, various test cases may be planned and then tested. When designing test cases, it is a good idea to design some test cases that will check if the software will fail and some that will check if it succeeds.

Use cases

A **use case** is an individual set of instructions or steps in which the user might use the software. A use case might also feature as part of a test plan.

Test criteria

Each area of the test plan needs criteria so you know if it has failed and why. The conditions for passing or failing each test need to be decided by the tester.

Acceptance criteria

The acceptance test is usually carried out by the intended users of the system, the people who requested or commissioned the software. The purpose of the acceptance test is to check that the system performs exactly as required.

The acceptance criteria should be clearly defined in advance. It is the criteria for which the user decides that the software is acceptable.

Test data

Before performing a test, you need to decide what data you are going to include in your **test case**. It is not normally possible to perform tests with every single possible piece of data. So, instead the developers will choose from a limited range of data such as:

- **valid** - the most obvious or common data that should work
- **valid extreme** - unusual, extreme or unexpected data, eg the highest and lowest (data that tests the limits but that should work)
- **invalid** - data that should definitely fail
- **invalid extreme** - data that is at the edge of failure and is nearly acceptable
- **erroneous** - data that is the wrong data type

Tests should find that the program works as expected. Obvious input data should confirm that the **software** works as expected. Extreme test data will be chosen to test what breaks the system.

For example, if you were developing a number-guessing game, you might have a unit of code that asks the user to choose a number in a specific range, eg "Choose a number between 1 and 10". To test this unit, you could try a whole range of inputs to see what happens:

3, 4.5, three, -99, 10.00001

Invariant

Sometimes software developers use something called an **invariant** when they are testing or fixing **bugs**. An invariant is a **value** or condition that can be relied upon to be true when a program is being **executed**.

Using something that can be relied upon helps the developer to isolate the units that are working from the units that are not.

Trace tables

When testing more complex examples of software, it is sometimes necessary to test a number of conditions and sub-conditions at the same time.

A **trace table**, also called a **trace matrix**, can be used to record the outcomes of the test. The trace table for a very simple example, such as x=y+2, would look like this:

y	x
13	15
25	27
1200	1202

A trace table might appear to look very similar to a test plan, but without the other headings like 'expected outcome' and 'actual outcome'.

Types of test

There are many different tests that can be carried out.

Validation and verification tests

Validation tests make sure that the system only allows 'sensible' data to be entered. Unless the input data is sensible, the computer could crash or produce results that don't make sense.

Verification tests clarify that the program is working as intended and is producing the **outputs** that were expected. This will be based on criteria set out in the first stage of the **development cycle**.

Functional and non-functional tests

Functional tests look at a specific function within the program. The purpose of a functional test is to see if all of the system or units work in the way that a user would expect them to

work. It involves checking against a list of the functions the **software** was designed to perform (a specification) and to see if it does what it was intended to do.

Non-functional tests look at the bigger picture of how a program performs. For example, a program might perform as expected (functional) with one user, but then crash unexpectedly if multiple users use it (non-functional).

Typical non-functional tests include:

- **load test** - testing how well the system performs if accessed by many users at the same time
- **volume test** - testing to check whether they can withstand large amounts of data
- **performance test** - testing how quickly the software can perform tasks or testing over very long periods of time
- **security test** - testing for unauthorised access

Unit and system tests

A unit test looks at a small section of the overall computer program. A system test checks that the whole program works.

Unit tests

As units of the program are created and developed, they are then tested and slowly integrated into a much larger working piece of code to make the computer system. Units are sets of code made up of one or more **module**.

As each new part of the system is completed, it is tested to make sure that it works properly - otherwise other parts of the system that will rely on it could fail. This is called **unit testing**. If the new part of the system works, it can be used by other parts of the system.

A unit test is usually undertaken early on in the development cycle. One benefit of unit testing is that it helps you to discover problems early when the problems are small. Some **programming languages**, including **Java**, **Python** and **Ruby**, have built-in features to support unit testing.

During a unit test, testing will make use of different **values** of data and different types of operating conditions.

System tests

Once you reach a point in the development cycle when a number of units are ready and tested, a **system test** can take place. The purpose of a system test is to ensure that all the individually-developed units of **software** work together as intended.

When problems are found in a system test, it might indicate that more unit development is required. Once this is complete, more unit tests and system tests will be necessary.

One benefit of system testing is that it helps you to discover problems or inconsistencies between the software units that are intended to work together.

Version control (Recording test results)

It is really important to keep detailed records of all the tests that have been carried out as well as any changes that have been made after the tests.

A **version control** system is used to track changes in the development of your software. Sometimes errors can be found in the software later on. The tracking that version control provides enables developers to go back and see what changes were made.

3.3 DEVELOPING A TESTING STRATEGY

It is important to develop a testing strategy before conducting the actual testing process. Test strategy is a plan for defining the testing approach, and it answers to questions like what you want to get done and how you are going to accomplish it. It is most important document for any QA team in software testing, and effectively writing this document is a skill that every tester develops with experience.

Test planning activities guides team to define test coverage and testing scope. It also aids test managers to get clear picture of the project at any instance. The possibility of missing any test activity is very low when there is a proper test strategy in place. Testing strategy plan must be conversed with the entire team so that the team will be consistent on approach and responsibilities.

Test Plan Vs Test Strategy

There is a great confusion about Test Plan and Test Strategy documents. Different organization have their unique processes and standards to manage these documents. For example, some organization include test strategy facts in test plan itself while some organization include strategy as a subsection within the testing plan.

Test Plan	Test Strategy
• In Test Plan, test focus and project scope are defined. It deals with test coverage, scheduling, features to be tested, features not to be tested, estimation and resource management.	• Test strategy is a guideline to be followed to achieve the test objective and execution of test types mentioned in the testing plan. It deals with test objective, test environment, test approach, automation tools and strategy, contingency plan, and risk analysis

The first step in the system testing is to prepare a plan that will test all aspects of the system in a way that promotes its credibility among potential users.

A test plan includes the following the following activities:

1. Prepare test plan:

A test plan includes the following:

a. Output expected from the system.

b. Criteria for evaluating outputs.

c. A volume of test data
d. Procedure for using test data.
e. Personnel and training requirements.

2. Specify condition for user acceptance testing:

Planning for user acceptance testing calls for the analyst and the user to agree on the conditions of test. Many of the condition can be derived from the test plan. Agreements on the test schedules, test duration, persons designated for the test are done. The start and end dates for the tests should be specified in advance.

3. Prepare test for program testing:

As each program is coded, test data are prepared and documented to ensure that all aspects of the program are properly tested. The data are filled for future reference.

4. Prepare test data for transaction path testing:

This activities develops the data required for testing every condition and transaction to be introduced into the system. The path for each transaction from the origin to the destination is carefully tested for reliable result.

5. Plan user training:

User training is designed to prepare the user for testing and concerting the system. Here, provisions are included for developing training materials like training manuals or documents to complete the training activity. The supervisors are trained for the following reasons:

a. User supervisors are knowledgeable about the capabilities of their staffs.
b. Staff members usually responds more favourably to accept instructions better from supervisors than from outsiders.
c. Familiarity of users with their particular problems make them better candidate for handling user training than system analyst. The analysts get feedback to ensure proper training is provided.

6. Compile and assemble program:

All programs have to be compiled/assembled for testing. Before this, a complete program description should be available. It should include the purpose of the program, its use, the programmers who prepared it and the amount of computer time it takes to run it. The type if activities done in this phase are:

a. Desk checking: The source code uncovers programming errors.
b. A run order schedule: It specifies the transactions to test and the order in which they should be tested.
c. Test scheme: It further consists of two approaches:

i. Bottom up approach: It tests small scale program modules which are linked to a higher level module and so on until the the program is complete.
ii. Top down approach: Here general program is tested first, followed by addition of program modules one level at a time to the lowest module.

7. Prepare job performance aids:

In this activity the materials to be used by personnel to run the system are specified and scheduled. This includes a display of materials such as program codes, a list of input codes attached to the CRT terminals, a posted instruction schedule. This aids reduce the training and employ personnel at lower levels.

8. Prepare operational documents:

During the test plan stage, all operational documents are finalized. Also section on the experience, training and educational qualification of personnel for proper operation of the new system are documented.

Types of system tests:

System testing consists of the following steps:

1. Program testing:

A program represents the logical element of the system. When a program is tested, the actual output is compared to the expected output. Achieving an error free program is the responsibility of the programmer. Program testing checks two types of errors:

a. Syntax errors: It occurs when program statement violates one or more rules of the programming language in which it is written.
b. Logical error: It deals with incorrect data fields, out of range items, invalid combinations.

2. String testing:

Programs are sequentially related to each other and interact in a total system. Each program is tested to see whether it confirms to related programs in the system. Each portion of the system is tested against entire module with both test data and live data before the entire system is ready to be tested.

3. System testing:

It is done to find flaws and weakness that were not found earlier. This includes forced system

failures and validation of the total system as it will be implemented by its users in the operational environment. The total system is tested for recovery and fall back after various major failures to ensure no data are lost.

4. System Documentation:

All design and test documentation should be finalized and entered into the library for future reference. The library is the central location for maintenance of the new system.

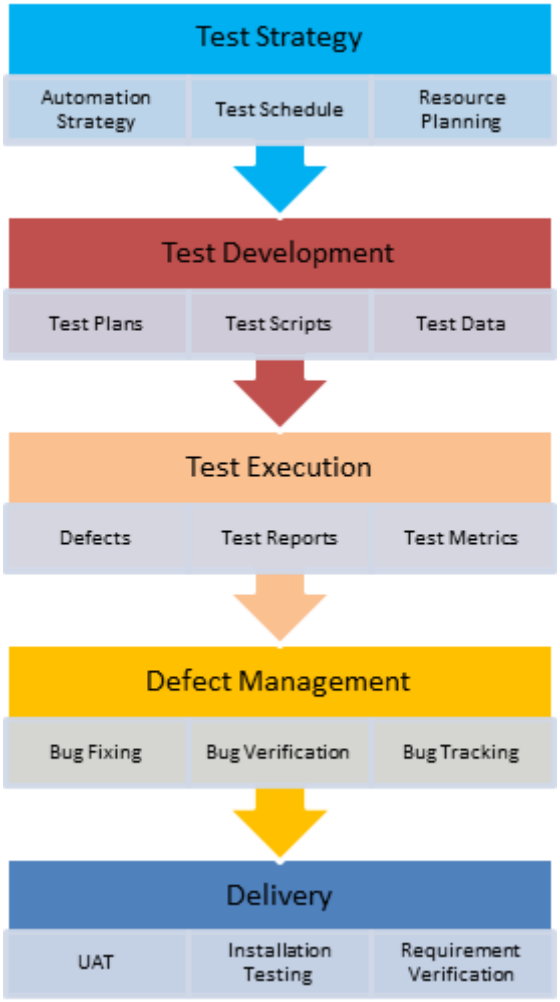
5. User acceptance testing:

An acceptance test has the objective of selling the user on the validity and reliability of the system. It verifies that the system's procedures operates to system specification and the integration of vital data is maintained.

How to prepare a good test strategy document

Every organization has their unique priority and set of rules for software designing, so do not copy any organization blindly. Always ensure that their document is compatible and adds value to your software development before following the template.

Test Strategy in STLC:



Step#1: Scope

It defines parameters like

- Who will review the document?
- Who will approve this document?
- Testing activities carried out with timelines

Step#2 Test Approach

It defines

- Process of testing
- Testing levels
- Roles and responsibilities of each team member
- Types of Testing (Load, Security, Performance testing etc.)
- Testing approach & automation tool if applicable
- Adding new defects, re-testing, defect triage, regression testing and test sign off

Step#3 Test Environment

- Define number of requirement and setup required for each environment
- Define backup of test data and restore strategy

Step#4 Testing Tools

- Automation and Test management tools needed for test execution
- Figure out number of open-source as well as commercial tools required, and determine how many users are supported on it and plan accordingly

Step#5 Release Control

- Release management plan with appropriate version history that will make sure test execution for all modification in that release

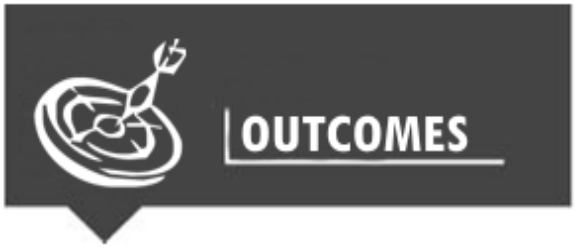
Step#6 Risk Analysis

- List all risks that you can estimate
- Give a clear plan to mitigate the risks also a contingency plan

Step#7 Review and Approvals

- All these activities are reviewed and sign off by the business team, project management, development team, etc.
- Summary of review changes should be traced at the beginning of the document along with approved date, name, and comment

SESSION 4: IMPLEMENTING A COMPUTER PROGRAM



On completion of this section you will be able to implement the program to meet business requirements.

1. The implementation involves checking the program for compliance with user expectations and any other applicable factors
2. The implementation involves training of users to enable them to use the software to their requirements
3. The implementation involves planning of installation of the program that minimises disruption to the user.

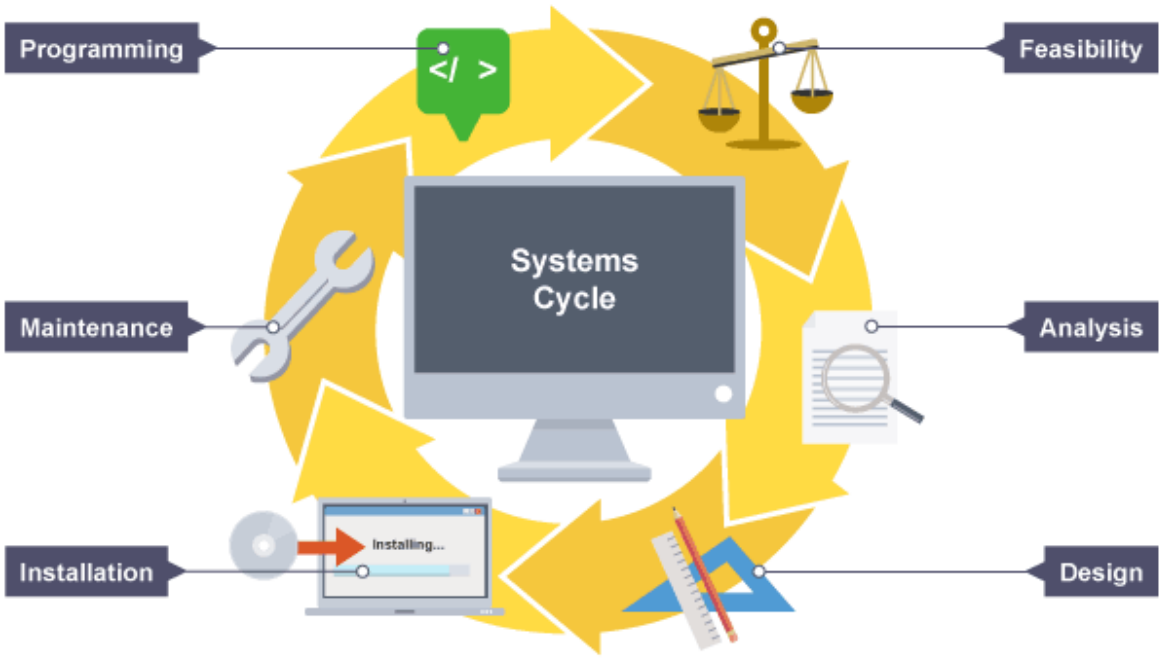
4.1 INTRODUCTION

Developing and activating a new computer system is a long process. It is important to break the process down into smaller stages, each one requiring a different set of professional skills. An important part of this process is the first stage - **analysis**. During this stage the systems analyst will investigate how the current system works and what needs to be improved. This involves finding out whether a new computer system is needed at all and exactly what it will be used for.

Analysis is all about looking at how a job is done at present and seeing if the job could be done better or more efficiently by upgrading or developing a new system.

With this goal in mind, the systems analyst might:

- observe staff at work
- interview staff about their work
- send out questionnaires about working practices
- inspect documents such as user guides, data capture forms and any printouts the current system creates



4.2 FEASIBILITY

The implementation of a computer program must involve checking the program for compliance with user expectations and the operating environment. This is done via a feasibility study.

Having investigated the present system, the systems analyst will produce a **feasibility study**. This will look at whether the new system is:

- **Technically feasible** - is the new system technically possible to implement in the time available and will it meet the needs of the users?
- **Economically viable** - will the cost of the new system be offset by savings once it is implemented, ie will it save the organisation time, money or increase its performance?

The project will only continue to the next stage if the answer to both of these questions is yes. At this point the decision makers in the organisation, e.g. the board of directors, decide whether or not to go ahead.

4.3 TRAINING USERS ON THE NEW PROGRAM

The implementation process must involve training the end users of the program so that they will be able to use the software. Training of the end users is one of the most important steps for a successful system implementation. The end users should be utilized during parallel testing, so training will need to be rolled out prior to that. Getting the end users involved at this point is also a good way to get them excited about the system, as many of them may not have been involved with the project prior to training. Their assistance in parallel testing will help them prepare for when the system goes live. End users are good at using the system in more of a "real world" situation and can judge when process flows are not working. When everyone involved with using the system is included in the training, they will feel more confident about using it as they go into production and the user community will view the implementation as successful. The system may have been tested for functionality and all customizations are working accurately, but if the end users do not know how to use it or feel comfortable with it, then the launch of the new system will be viewed as unsuccessful.. Therefore, the timing of the end user training is critical and must be planned for and implemented prior to the start of the parallel test phase to ensure a successful implementation.

There are two possible solutions for training. The first is to use project team members to develop and deliver the end user training and the second is to identify a training partner to support the development and delivery of end user training, including a train the trainer component. Both options will be fully explored during the next phase of the project.

Training Design

Identify Training Needs:

- Have a Key Message on what Training will be provided for the End Users (Like a Mission Statement on what will be delivered to the users)
- Decide when training should be started. It should be rolled out before parallel testing so the end users can be a part of that system testing right away.
- Identify what areas should be trained

- Develop a Content Outline on what Functionality of the system should be covered in the training, what modules of the system will be included and who (the end users) should be trained.
- Establish a Training Team

Design Training Materials/Documents

- Design a layout of the training manual, and what each functional and technical areas of the system should be included. An overview section should also be included for the areas that do not need detailed training. Also query and report writing training should be included.

Design Support Materials

- Design any support materials that may be needed for use during training.

Finalize Design Document

- Publish the training document outline and have the team review and finalize**.**

Update Training and Support Materials Plan

- Update the Project Plan to accommodate any changes to the Training Schedule

4.4 PLANNING THE INSTALLATION OF THE PROGRAM

You might typically associate the word "installation" with that of a program or application, such as the one on the right that you have downloaded off the internet. You typically see some sort of loading screen, indicating the progress of the installation and have a variety of options that you can set in order to get the most out of the application.

A new system can be thought of as similar to that of the installation of a program on your computer, in that it is placed in a new environment. However, **the two must not be confused**. Installing a computer program is not the same as installing a new system, as the latter is about the *development* of a new system, the focus of this section, rather than just unpackaging files for execution.

Identify the context for which a new system is planned.

In its most general sense, an organisation is the collective goal which linked to its environment, or "context". The environment, therefore, defines the limits of an organisation. When systems are built, they are defined by the needs of their creators, and as such, systems are not independent entities but exist in an environment. This environment affects the functioning and the performance of the system. Sometimes, the environment may be thought of as a system in its own right, but more generally, it consists of a number of other systems which interact with each other.

Describe the need for change management.

Moore's law describes the natural progression of technology, which essentially indicates that there is a natural development of new systems. These new systems will be developed in response to a problem or to replace an older system. In the case of the latter, these new systems will require a fluid transition from one system to another, requiring management. This management will require the technical ability to oversee this transition and have a say in how it is best implemented.

Outline compatibility issues resulting from situations including legacy systems or business mergers.

A whole host of errors or incompatibilities may stem from transition of software, and the insightful knowledge of an external party or internal expert may reduce the friction between the implementation of the newer systems.

Legacy Systems

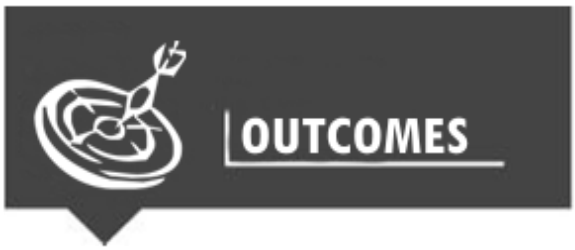
Legacy systems refer to outdated computer systems that have been superceded by newer technologies. Such systems have many disadvantages. First, costs for maintaining older systems may be higher as the systems may fail more frequently with age. Second, technical support for these systems may no longer be available. Additionally, legacy systems tend to be more vulnerable to security threats due to a lack of security updates.

Compatibility Issues

When a system changes, there are repercussions. These repercussions can be felt locally by the users who use the system on a day-to-day basis, or internationally by those who rely on it. During the design process of a new system, developers must take into account the influence of this change. Following the earlier example of the spreadsheet revolution, people who began to use the software had to undergo a learning curve. This is the most basic and fundamental sense of the term "compatibility issue." If they could not use it, they could not work or had to continue using their old system, only to see it become more and more obsolete.

Compatibility issues these days can have greater repercussions, especially when systems are relied upon. If a system is developed and does not take into account the fact that other users, perhaps even third-parties, rely on it, then it might experience a whole host of problems. Going back to the spreadsheets, if an organisation began using spreadsheet software to manage the orders for a particular good, and it begins to co-operate with another organisation who uses ye old pen-and-paper, they are going to have great difficulty working together. The new system did not take into account users who still managed their orders in this manner, and does not allow these people to easily transfer the details. Because the software is not bespoke and is developed generally, some end-users may find it difficult to utilise the new system. Whilst a solution may be to provide comprehensive and easy-to-read documentation so much so that it eliminates obscurities found by people, in some cases, data can no longer transfer between the two parties.

SESSION 5: DOCUMENTING A COMPUTER PROGRAM



On completion of this section you will be able to Document the program according to industry standards.

1. The documentation includes annotation of the program with a description of program purpose and design specifics
2. The documentation includes the layout of the program code including indentation and other acceptable industry standards
3. The documentation includes full internal and external documentation, with a level of detail that enables other programmers to analyse the program
4. The documentation reflects the tested and implemented program, including changes made during testing of the program

5.1 INTRODUCTION

Software documentation is written text or illustration that accompanies computer software. It either explains how it operates or how to use it, and may mean different things to people in different roles.

Documentation is an important part of software engineering.

Types of documentation include:

1. Requirements – Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what will be or has been implemented.
2. Architecture/Design – Overview of software. Includes relations to an environment and construction principles to be used in design of software components.
3. Technical – Documentation of code, algorithms, interfaces, and APIs.
4. End user – Manuals for the end-user, system administrators and support staff.
5. Marketing – How to market the product and analysis of the market demand.

Good software documentation, whether a specifications document for programmers and testers, a technical document for internal users, or software manuals and help files for end users, helps the person working with the software understand its features and functions. Good software documentation is specific, concise, and relevant, providing all the information important to the person using the software.

5.2 SOFTWARE DOCUMENTATION FOR TECHNICAL USERS

The following are steps when writing program documentation for technical users;

1. **Determine what information needs to be included.** Software specification documents serve as reference manuals for designers of the user interface, programmers who write the code, and testers who verify that the software works as intended. The exact information depends on the program in question but may include any of the following:
    - Key files within the application. This may include files created by the development team, databases accessed during the program's operation, and third-party utility programs.
    - Functions and subroutines. This includes an explanation of what each function or subroutine does, including its range of input values and output values.
    - Program variables and constants, and how they're used in the application.
    - The overall program structure. For a disc-based application, this may mean describing the program's individual modules and libraries, while for a Web application, this may mean describing which pages use which files.

2. **Decide how much of the documentation should be within the program code and how much should be separate from it.** The more technical documentation is developed within the program's source code to begin with, the easier it will be to update and maintain along with the code, as well as to document various versions of the original application. At a minimum, documentation within the source code needs to explain the purpose of functions, subroutines, variables, and constants.
    - If the source code is particularly lengthy, it can be documented in the form of a help file, which can be indexed or searched with keywords. This is a particular advantage for applications where the program logic is fragmented over many pages and includes a number of supplemental files, as with certain Web applications.
    - Some programming languages, such as Java and the .NET Framework (Visual Basic.NET, C #), have their own standards for documenting code. In these cases, follow the standards as to how much of the documentation should be included with the source code.

3. **Choose the appropriate documentation tool.** To some extent, this is determined by the language the code is written in, be it C++, C#, Visual Basic, Java, or PHP, as specific tools exist for these and other languages. In other cases, the tool to use is determined by the type of documentation required.

- Word-processing programs for Microsoft Word are adequate for creating separate text files of documentation, as long as the documentation is fairly short and simple. For long, complex text files, many technical writers prefer a documentation tool such as Adobe FrameMaker.
- Help files for documenting source code can be produced with any help authoring tool, such as RoboHelp, Help and Manual, Doc-To-Help, MadCap Flare, or HelpLogix.

5.3 PROGRAM DOCUMENTATION FOR END USERS

The following are steps when developing program documentation for end users;

1. **Determine the business reasons for your documentation.** Although the functional reason for documenting software is to help users understand how to use the application, there are other reasons as well, such as assisting in marketing the software, enhancing the company image, and most notably, reducing technical support costs. In some cases, documentation is necessary to comply with certain regulations or other legal requirements.
    - In no case, however, should software documentation substitute for poor interface design. If an application screen requires reams of documentation to explain it, better to change the screen design to something more intuitive.

2. **Understand the audience you're writing the documentation for.** In most cases, software users have little knowledge of computers outside of the tasks the applications they use enable them to do. There are several ways to determine how to address their needs with your documentation.
    - Look at the job titles your prospective users hold. A system administrator is likely expert with a number of software applications, while a data entry clerk is more likely to know only the application he or she currently uses to enter data.
    - Look at the users themselves. Although job titles generally indicate what people do, there can be considerable variation in how certain titles are used within a given organization. By interviewing prospective users, you can get a feel for whether your impressions of what their job title indicates are accurate or not.
    - Look at existing documentation. Documentation for previous versions of software, as well as functional specifications, provide some indication as to what the user will need to know to use the program. Keep in mind, however, that end users are not as interested in how the program works as they are in what it can do for them.
    - Identify the tasks needed to do the job, and what tasks need to be done before those tasks can be done.

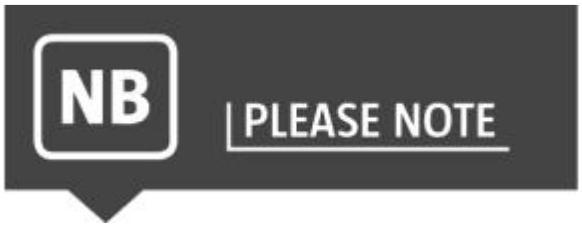
3. **Determine the appropriate format(s) for the documentation.** Software documentation can be structured in 1 of 2 formats, the reference manual and the user guide. Sometimes, a combination of formats is the best approach.
    - A reference manual format is devoted to explaining the individual features of a software application (button, tab, field, and dialog box) and how they work. Many help files are written in this format, particularly context-sensitive help that

displays a relevant topic whenever a user clicks the Help button on a particular screen.

- A user guide format explains how to use the software to perform a particular task. User guides are often formatted as printed guides or PDFs, although some help files include topics on how to perform particular tasks. (These help topics are usually not context-sensitive, although they may be hyperlinked to from topics that are.) User guides often take the form of tutorials, with a summary of the tasks to be performed in the introduction and instructions given in numbered steps.

4. **Decide what form(s) the documentation should take.** Software documentation for end users can take 1 or several of many forms: printed manuals, PDF documents, help files, or online help. Each form is designed to show the user how to use each of the program's functions, whether in the form of a walkthrough or a tutorial; in the case of help files and online help, this may include demonstration videos as well as text and still graphics.
    - Help files and online help should be indexed and keyword-searchable to allow users to quickly find the information they're looking for. Although help file authoring tools can generate indexes automatically, it is often better to create the index manually, using terms users are likely to search for.

5. **Choose the appropriate documentation tool.** Printed or PDF user manuals can be written with a word-processing program like Word or a sophisticated text editor like FrameMaker, depending on their length and complexity. Help files can be written with a help authoring tool like RoboHelp, Help and Manual, Doc-To-Help, Flare, HelpLogix, or HelpServer.



Note

Any program documentation be it for technical users or end-users must;

- Annotation of the program with description of program purpose and design specifics.
- Have a layout of the program code.
- Include full internal and external documentation, with level of detail that enables other programmers to analyse the program.
- Reflect the tested and implemented program, including changes made during testing of the program.