Unit 5 is worth 30% of your portfolio grade

Podcast - Introduction to Unit 5

This is the largest element of the course, both in marks and the time we expect it to take. This is the unit in which you really start to learn how to program, building on the preliminary skirmishes with the subject you began in Unit 4. We think that, for an average student in search of an average mark, this will take a good 30 hours of effort, much of it involving practice, debugging, and fixing of errors. For those hoping for more, it may take longer.

In keeping with the approach used so far in this course, we provide a process, a few links and a bit of scaffolding to help you on your way, but much of the work here involves visiting other sites to find tutorials that work for you. As and when you hit problems, tutors and other students can be reached through the Landing to seek help and assistance. Please remember to ask for this when you need it, and please help others when you can—to teach is a great way to learn, especially in dialogue with others.

No Pain, No Gain

We have already warned you a little of what to expect from this experience in Unit 4, observing that programming is sometimes an angst-ridden process involving confusion, frustration, a lot of tedious detailed and painstaking work and, after a great deal of practice and error, delight. I expect you have already noticed this, even when simply re-using someone else's code.

The 80/20 Rule

If you have not programmed before, it will probably take a while to understand the rhythms of the programming process. In the first place, Pareto's Principle applies here: about 80% of it will cause you few problems and maybe take 20% of the time. That's good. Unfortunately, the remaining 20% is likely to take 80% of the time (and maybe more—there is a strange quantum field that surrounds coding that bends the usual rules of time).

As an experienced programmer I learned, when estimating required effort, to first start with how long I think it will take, double that, and change the units to the next one up. So, if a program *should* take an hour to write, the chances are that it *will* take two days before it really works. If it should take a couple of weeks, it will probably take four months. It is normal and expected that you will grind your teeth and want to destroy your computer in despair for days over something that will eventually turn out to be as simple as a missing bit of punctuation or a forgotten keyword. We all do it, even the professionals although, if you aim for getting it right in the first place, it gets easier as you go along.

The Rewards

Given the dangers to our mental and physical health that this brings, why do we do it? I'm guessing that (beyond the financial rewards) it might be something to do with the equally intense feeling of satisfaction and immense pleasure to be gained from seeing the code you have written behave as it should, like a well-oiled machine that you have built from its component parts. In fact, even when you see “Hello world” appear for the first time as a result of your labours of love, it is a mighty fine feeling.

So—are you ready for the ride?

Let's go . . .

Learning Outcomes

When you have completed this unit, you should be better able to

- write well-structured, easily maintained JavaScript code following accepted good practice, including
    - general appearance and form: well-commented, properly laid out, appropriate capitalization.
    - structure: modular, using functions, classes and objects effectively, making effective use of variables.
    - standards-compliant: avoiding proprietary elements where possible and working with the vast majority of web browsers.
    - accessible: usable in some form by people with a wide range of disabilities or, if not, failing gracefully and offering alternatives to achieve similar functionality.
- make effective and efficient use of a full range of programming constructs including sequence, selection and iteration.

- effectively use variables, including passed parameters, local and global variables and arrays, to improve the efficiency, re-use, and maintainability of the code.
- use a wide range of programming features and commands to improve the efficiency, re-use and maintainability of the code.
- write JavaScript code that works in all major browsers (including IE, Mozilla-based browsers such as Firefox, Opera, Konqueror, Safari, Chrome).
- effectively debug JavaScript code, making use of good practice and debugging tools.

Problem

Important: do not begin this task until you have completed all of the previous units, including reflective learning diary entries.

This unit requires you to analyse a problem related to your website, design a solution, then code it. There are two stopping points along the way: identifying requirements and creating a design, where you will have to await feedback from your tutor before continuing, and several things that require you to reflect in your learning diary. Make sure you do this! Fuller information is available in the Process Guide section of this document below.

Part One: Analysis and Design

A vital part of good programming practice is careful and methodical design—this will reduce errors to a minimum and make it vastly easier to fix, improve, and update later on. Much of the time, unless the task has been allocated wholly to someone else, a programmer should also be extremely familiar with the needs that his or her program will satisfy. You have already gone some way towards dealing with the analysis part of this and some of the design elements, but now it is necessary to think about the code that will be needed to meet those needs.

Based on your personas and scenarios developed for Unit 1, in your learning diary identify ways that JavaScript elements might improve your site. It is important that the code you develop meets the learning outcomes for this unit as well as being justified and useful in the context of the site. Examples might include (but are by no means limited to) or contain:

- menus, collapsible lists, tab sets, accordions

- forms with error checking, auto-filling of forms
- shopping cart
- games
- adaptive elements that show different content for different users
- opportunities for visitors to customize their view of your site
- lessons (courseware)
- site search
- musical instruments
- tools and utilities such as calculators, calendars, email forms, convertors, HTML generators, and so on
- text or graphics editors
- dynamic graphs

You are encouraged to be creative in this, so don't feel that you have to use any of these—they are just examples of the kinds of system that might be of value. But the value is dependent on the scenarios, personas, and purposes of your site, and you should never add something for the sake of it or because it shows off how well you can code. There must always be a well-justified reason for creating a particular bit of code.

Do spend some time checking to see what others have done if it is available on the Landing through shared blogs, wiki entries, site links, and so on. If you have a good idea yourself, add it to the wiki! Feel free to get inspiration from sites used in Unit 4, but do not simply re-use or slightly modify such code for the work required in this unit. The object of this exercise is to learn to program, not just to modify or re-use what already exists. We strongly encourage *legal and properly cited* re-use of code but, the more you use of others' work, the better we expect your program to be on top of it. Standing on the shoulders of giants is great, but you have to reach higher as a result.

The next step is to identify the data, objects, functions, methods, and properties you will need to assemble in your program and, once you have gathered these together, to design the logical process model. There are several ways to do this of varying complexity. We would suggest that you choose one or more of:

- a flow diagram or similar
- pseudo-code
- a structure chart showing relationships and flow between functions and objects.

There are many other ways to represent your program—feel free to explore other alternatives. The purpose is to think about the design before you get down to coding it so, if it achieves this goal, it is probably fine. If in doubt, ask. If you cannot decide, then pseudo-code may be your safest bet and involves a relatively low learning curve. If you do use this process, we suggest that you start by sketching out the broad functionality first with general headings and functions then drill down to the details, rather than trying to design your code a statement at a time.

Part Two: Code It

We expect that you should use (at least) the following concepts and constructs in the language, and you will lose marks if any are not present:

- comments – extensively and descriptively throughout
- sequence, selection, iteration
- variables and variable declarations
- arrays
- functions
- classes and objects

If any of this is confusing, step back a bit and see the previous unit for more information about what these terms mean.

Extra marks will be given for effective use of more complex constructs and tools such as:

- regular expressions
- recursion

We expect your code to be very well integrated with the HTML and CSS of your pages, making full use of events and objects in the DOM.

Important: we are looking for well-designed, well-constructed, well-formatted, well-commented, elegant, readable, maintainable, bug-free code that does what it is intended to do and that you have written yourself. Marks will be lost if any of those things are not true.

Upload your code to your SCIS site, and, of course, reflect on the process in your learning diary. **Important**: create a zipped-up version of the code (no need for the images), and save it as an attachment to your diary entry. Remember that you must do this at the end of each unit. This will help to provide us with a clearer image of how your site has developed over the course, what you have learned and how you have developed your skills.

Process Guide

Please follow this process carefully. In particular, note that **you are expected to consult with your tutor when choosing your project, and you must submit your design to the tutor before coding it**.

Learning The Basics

The ability to learn new things about programming and new ways to code by yourself is perhaps the most important outcome we would wish for you in this course. We are not leaving you entirely to your own devices and will offer support when needed as well as a structured process for those that need it (see below), but the more you can help yourself, the better it will be for you.

Start teaching yourself to program: you may use available sites and books from our **Essential Resources** list or the course bookmarks, or feel free to seek out better resources and, if you find any, share them with the course via the Landing bookmarks.

Choosing the Project

Research the possibilities. You should already have started this in Unit 4, but you will probably want to spend longer finding out what kind of things others have done. We are looking for a fairly substantial amount of code, in the order of 200–300 lines (potentially in multiple smaller chunks), so think beyond a simple component or interface tweak.

Always keep your site purpose, scenarios, and personas in mind: some things might be fun to do but, if they don't fit with the needs of your site, they are useless to you and will get you few if any marks. Also bear in mind that we are seeking evidence that you have used a good variety of code structures, objects, methods, and so on, so your project should be sufficiently complex that you can show off your skills. Your tutor will give feedback when you have reached the end of this process that will help to ensure you have taken on a suitable challenge.

In your reflective learning diary, write three ideas for how you want to improve your site with JavaScript. For each idea, explain the idea and how you want to it to work in a paragraph or two (pictures may help). For each idea, briefly explain how it relates to the purposes, personas, and scenarios of the site. Save that entry.

We will be looking at the timestamp on this, so do not go back and edit it after you have made the final edits. You will be using at least one of those ideas for your program and will not, without good reason and tutor consent, be able to change your mind.

Inform your tutor that you have done this and wait for feedback using the 'Review of your three ideas for using JavaScript' link on this Moodle site. If this feedback takes more than a few days (likely over weekends or holidays or if the tutor is away) do feel free to start investigating the ways to go about this. Your tutor will comment, typically briefly, on your ideas. He or she may suggest that you combine them, or reduce the scope, or make some modifications so that you have the best chance to achieve successful learning outcomes.

Designing

The next step is to **stop**. Before you go any further, you will need to design your program. Do not, under any circumstances, leap straight into the code. A full design process is beyond the scope of this course, so we provide a simple formula, much as we have done for the interaction design part of the course. This is a cut-down version of a full design process but should be sufficient for your needs.

Your design should show

- data: names of variables, arrays, and so on, indicating their scope (e.g., are they private to particular functions or objects).
- functions and classes: a full list of the various functions and objects that you will create.
- program flow: this may be created as a diagram or pseudo-code in a form that you feel comfortable with. A full flow chart or class diagram is not necessary, but your diagram should show the main processes and functions of your program and how they relate together, including any loops or selections along the way.

Unless it is a purely textual format such as pseudo-code, please ensure that this is saved in PDF, PNG, or SVG format. Do not provide proprietary formats such as MS Word or Omnigraffle; do not save this as a JPEG image (because JPEG is a lossy format that will make your diagram unclear). Please ensure that the diagram is readable.

The more detailed your design at this stage, the easier it will be when you come to code it, but you should be a little wary of starting at too fine a granularity. For instance, if you produce the most detailed design notation of all in the form of pseudo-code, it is almost trivial to adapt that into JavaScript. It is quite hard to see the big picture, and it is very easy to miss important chunks of structure if you start at too fine a level. But if you just show a structure chart, you will have much more work to do when it comes to coding. A good rule of thumb is that, if it takes almost as much effort to write the design documentation as it will to write the final code, you should not bother.

A sensible approach for most would be to combine methods as appropriate: where the coding requirements are simple, you may only need a box on your chart to show what is needed: for example, if you are just going to generate a linear table or menu, then it is probable that you could get by with a simple loop, so there would be no great need to go into more detail. If a bit of coding is likely to be more complex, a flow diagram or pseudo-code would be more useful and would help you to understand what is needed better. For instance, if multiple loops and branches are needed, or recursion, or calls to multiple functions or objects, then you would probably be wise to sketch the program flow in advance.

Present your design in your reflective learning diary.

Once you have completed your design, **share it with your tutor using the 'Review of your JavaScript program design(s)' link on this Moodle site and, again, wait for feedback**. Your tutor may have suggestions that will improve the design and maximize your chances of achieving a good mark. Feedback will usually arrive within seven to eight business days, though it may take longer under rare circumstances. Be patient and, while you are waiting, carry on trying more JavaScript tutorials and honing your coding skills.

Note that a design is seldom precisely the same structure as the final program—you will almost certainly want to tweak it and may find constraints of the language or your skills limit some of your ideas. This is fine, as long as the broad structure of the design remains intact when it comes to coding it.

Coding

Start coding. You will certainly need tools and references at hand, as well as ongoing tutorials.

Remember: use comments to explain your code very liberally—we will pay a great deal of attention to these.

Your reflective learning diary entry for this part of the course is especially important. We need to know how you went about learning JavaScript, what obstacles you faced, how and whether you overcame them, what you would do differently next time. This is a vital part of your own learning process, but if you feel willing to share this with others, it would be especially useful and may help them when faced with similar difficulties.

In your reflective learning diary, explain in detail how the code improves the experience of the personas you created in Unit 1, and how it helps with the scenarios you presented. If, as a result of this, you need to make changes to Unit 1, you should explain what changes were made and why you made them. Do not modify the original—make a copy and post it as well.

Submit your code by adding it to your area of the SCIS site.

⚠ Important: create a zipped-up version of the code (no need for the images), and save it as an attachment to your diary entry. Remember that you must do this at the end of each unit. This will help to provide us with a clearer image of how your site has developed over the course, what you have learned and how you have developed your skills.

There is no formal feedback given at this point, but you may ask for a very brief appraisal by your tutor if you wish, and look at the work of others, if available. If you comment on their code and they comment on yours, you will all benefit immensely. There are few better ways to learn than to teach, and many eyes can suggest improvements no single individual such as your tutor could uncover.

Essential Resources

Learning is best done when you need it. This stage in the course is about that time.

We have selected a default set of resources and paths that you may wish to follow if there are too many or too few choices, but do not feel constrained by this. We have also provided some self-test quizzes that will help you to assess whether or not you have sufficient grasp of the subject. We suggest the following sequence to make learning easier, but feel free to choose a different path if it suits you better:

JavaScript Topic	**Recommended Default Tutorial**	**Alternative Tutorials**

1	Inserting JavaScript on an HTML page	http://www.w3schools.com/ js/js_howto.asp http://www.w3schools.com js/js_whereto.asp http://www.w3schools.com js/js_guidelines.asp	http://www.echoecho.com/ javascript1.htm
2	Comments	http://www.w3schools.com/ js/js_comments.asp	http://www.tizag.com/javascriptT/ javascriptcomment.php
3	Variables	http://www.w3schools.com js/js_variables.asp	http://www.echoecho.com/ javascript5.htm
4	Arrays	http://www.tizag.com/javascriptT javascriptarray.php	http://www.echoecho.com/ javascript10.htm
5	Operators	http://www.w3schools.com/ js/js_operators.asp	http://www.tizag.com/javascriptT/ javascriptoperators.php

6	Comparisons	http://www.w3schools.com/js/js_comparisons.asp	http://www.tizag.com/javascriptT/javascriptoperators.php
7	If...Else and Switch statements	http://www.w3schools.com/js/js_if_else.asp	http://www.echoecho.com/javascript6.htm
		http://www.w3schools.com/js/js_switch.asp	http://www.tizag.com/javascriptT/javascriptif.php
8	Functions	http://www.w3schools.com/js/js_functions.asp	http://www.echoecho.com/javascript7.htm
9	For loops	http://www.w3schools.com/js/js_loop_for.asp	http://www.echoecho.com/javascript9.htm
		http://www.w3schools.com/js/js_loop_for_in.asp	http://www.tizag.com/javascriptT/javascriptfor.php
10	While loops	http://www.w3schools.com/js/js_loop_while.asp	http://www.echoecho.com/javascript9.htm

			http://www.tizag.com/javascriptT/javascriptwhile.php
11	Events	http://www.w3schools.com/js/js_events.asp	http://www.echoecho.com/javascript8.htm
12	Error handling (Try...catch etc)	http://www.w3schools.com/js/js_try_catch.asp http://www.w3schools.com/js/js_throw.asp	http://www.javascriptkit.com/javatutors/trycatch.shtml
13	Objects and classes	http://www.w3schools.com/js/js_obj_intro.asp	http://www.echoecho.com/jsquickref.htm
14	Regular expressions	http://www.w3schools.com/js/js_obj_regexp.asp	http://www.javascriptkit.com/javatutors/redev.shtml
15	Recursion	http://www.c-point.com/javascript_tutorial/recursion.htm	http://www.java2s.com/Tutorial/JavaScript/0140__Function/Recursivefunction.htm

TOP

Alternatively, you may prefer to follow a more structured path using a book. There are thousands of such books available of unequal quality—if you find a good one, please add a bookmark to the Landing group for this course and, of course, do look there for recommendations. The following are free and are pretty good:

- **Eloquent JavaScript** (better on JavaScript, weaker on teaching it): http://eloquentjavascript.net/ (also available in the Files section of the course Landing group)
- **SAMS Tech Yourself JavaScript in 24 hours** (weaker on JavaScript, stronger on teaching it): http://www.informit.com/library/library.aspx?b=STY_JavaScript_24_hours.

In addition to the basics, you will definitely need to know about some of the functions and methods that JavaScript provides that help you to manipulate HTML pages. Some specific JavaScript tricks, objects, and methods that you might want to explore include:

- using timers (e.g. http://www.w3schools.com/js/js_timing.asp)
- animation
- image manipulation
- dragging and dropping
- using cookies - writing and reading (e.g. http://www.w3schools.com/js/js_cookies.asp)
- drop-down menus
- form validation (e.g. http://www.w3schools.com/js/js_form_validation.asp)
- redirection
- getElementById()
- date handling (e.g. http://www.w3schools.com/js/js_obj_date.asp)
- string manipulation
- browser detection (e.g. http://www.w3schools.com/js/js_browser.asp)
- error handling

However, you may well come up with many other things that will help in your program. The suggested tutorials will show a range of possibilities, so some of this will be learned as you go along, but you are also advised to browse through JavaScript references in order to discover what the language is capable of.

Different learners have different needs and different interests at different times. Our suggested path through may not be the one for you. You might, for example, prefer to start with a problem and work your way through. If you have used a different programming language before, you may find some of the tutorials a bit too easy. Feel free to adopt a different approach and, if you find a good one, please share it with the rest of us—there are sure to be others with similar needs and interests who would benefit from your discoveries. Do bear in mind that the further you depart from the methods we suggest, the less easy it may be for us to help you with problems along the way, but don't let that stop you if it works for you—we will do our best to help if you run into difficulties.

Further Guidance

Practice. Practice. Practice.

It is *really* important that you practice then practice some more. Don’t just paste code and see what happens. Doing so is OK from time to time if you want a very quick overview of what is intended and is an acceptable way to get started sometimes—just don't make a habit of it.

Experienced programmers tend to make very little use of copied code, though they certainly try to re-use complete libraries, components, and objects whenever they can. As you become more experienced, you will probably find that you will far more frequently read other people’s code to see how it works and then write your own using what you have learned as a result, rather than simply copying it in.

Generally speaking, it is far better to type the code in yourself—by doing this you are teaching your fingers to write code. It’s a bit like practicing scales on a musical instrument. It’s a process that helps code writing to become second nature. It might be more boring, more long-winded, and less interesting than rushing forward to the next exercise, but you will not learn much JavaScript just by reading about it and running example code. Programming is a craft: learning the craft is achieved by doing, not just by reading.

When you *do* type in code from the various exercises and tutorials, always try to see what happens when you tweak it: change the values of variables, replace text with your own, change the variable names to ones that suit you better.

As you learn more, try to work out different ways of achieving the same thing. For example, if you come across a 'for' loop, try to change it into a 'while' loop, or if you come across a form that uses radio buttons, try to change it to one that uses a drop-down list. In doing this kind of exercise, you are drilling yourself to think more and more like a programmer.

Use the debugging tools in your browser to try out pieces of code, or if you prefer, use the 'try it yourself' tools provided at W3Schools that will run some kinds of code that you create so that you can see the results immediately. However, remember that a program is an intricate machine in which there are dependencies between the parts, so you will only learn so much from executing simple statements. This is a bit like learning to write: the fact that you can write words and sentences does not make you able to write novels—at a larger scale, different skills are involved.

Look for tutorials about particular things that interest you on the Web—they often provide step-by-step guides to building complex programs from which you can learn a great deal. As always, remember to cite any code that you use as a result—we encourage and reward intelligent re-use, but you must remember to tell us where it came from.

If we find something you have used from elsewhere without proper citation, we will treat it as plagiarism. If you can find it, then we can probably find it too, and we have motivation, tools, and techniques for doing so.

Getting Support

TOP

While we do not wish to hold your hand and tell you what to do at every juncture, this **is** a supported process: if you have problems, post a message to the course forum on the Landing and/or ask your tutor for help. He, she, or your course-mates may not have all the answers, but the process of talking problems through is half way to solving them. We should emphasize that the tutor will often try to prod you to find the answers by yourself and will, under most circumstances, not give them directly to you—a tutor’s role is not just to keep you comfortable (at least, not all the time) but also to challenge you when necessary. On the other hand, your fellow students might give you the answers: there's no harm in asking. If you can help a student in this way, note that you may use the help you give as evidence towards having achieved the learning outcomes for this course—helping others means more marks!

Some Notes on Debugging

All programs beyond the most elementary have bugs. Some of these are simple, such as syntax errors. Most (but by no means all) of these can be discovered fairly easily when you try to run them because the interpreter will fail, giving you an error message which often (but by no means always) will highlight where things have gone wrong.

It is not always very obvious, however; for example, if you have ended a loop in the wrong place, or forgotten to close a statement, then the interpreter may encounter an error somewhere else in the program, not where the mistake occurred. Other errors relate to problems with the logic of the code, leading to run-time errors. In other words, the program will run and do what you have told it to do, but what you have told it to do will not be what you intend. For instance, you may create a loop of commands that never ends, or never begins. Or you may try to add a number and a string of text (which JavaScript, as a weakly typed language, may accept and treat as best it can).

Sometimes, everything will work perfectly well, but the output will not be what you want—anything from poor formatting to utter nonsense. Sometimes, everything will work fine until the user does something unexpected. The interpreter will sometimes (not always) alert you to the problem, though it is less likely to be helpful in determining the cause of the error.

At this point you need to follow a method in fixing your code and, to make your life easier, enlist the help of a *debugger*. A debugger is a program that lets you look more closely at what your code is doing, such as telling you the values of variables as the program runs or letting you step through it line by line. Beware, however, that even the best debuggers may not be able to point at the precise problem with your code. For instance, a comma or semicolon missing in one place may not cause problems until later in the code.

Did we mention this already? *Practice*

TOP

This is the most significant part of the course that requires you to come to terms with programming. For some people, programming comes naturally. For others, it will be a struggle. For everyone, the key to success is always going to be practice: try, try, and try again. Discover what works, discover what doesn't, do it until it is second nature.

It's a bit like learning a musical instrument: you can only learn so much by reading and observing. The only way to really learn is to practice, get it wrong, try again, practice some more, getting better slowly and continuously (well—sometimes it may feel like a step backwards!) as you go. Do spend time attempting exercises on JavaScript tutorial sites. Do not skimp here.

Important: make a note of what you do and how you deal with bugs for your reflective learning diary. While reflection is always important as part of this learning process, it is doubly so when coping with the emotional ups and downs of debugging code. Don't be afraid to write about how you felt about it as well as the things that you did, would do differently next time, would do the same. This will help considerably both to consolidate the learning experience and as a form of self-healing therapy so that you can better understand what is involved in the programming/debugging process and approach it with confidence next time.

Making mistakes can be helpful in learning to code, but this is only true if you are able to reflect upon how they came about, what you learned from them and how you would cope with them or avoid them next time.

Indicative Grading Criteria

Grade	Criteria
A	• Rich use of a wide variety of code constructs, objects, and functions, each appropriately used and elegantly constructed, with appropriate methods and designs for each task the code performs. • Effective and intelligent use of advanced techniques including objects, recursion, and regular expressions. • Well integrated with the HTML and CSS of the site, with good separation of data, process, and presentation. • Well-commented, well laid-out and maintainable. • The need for the code is strongly driven by the purposes, personas, and scenarios of Unit 1 and notably enhances the experience of the end user in that context. • No apparent bugs, logic or run-time errors; any errors or exceptions treated well. • Good interface design—highly accessible and usable. • Works (or fails gracefully) with all browsers.

B

- Use of a varied range of code constructs, objects, and functions, each appropriately used, with appropriate methods and designs for each task the code performs.
- Some use of advanced techniques including objects, recursion, and regular expressions.
- Well integrated with the HTML and CSS of the site.
- Well-commented, well laid-out and maintainable.
- The need for the code clearly derives from the purposes, personas, and scenarios of Unit 1 and is explicitly shown to improve the experience of the end user in that context.
- No glaring bugs, logic errors, or run-time errors; any errors captured and treated appropriately.
- Good interface design—accessible and usable.
- Works with all common desktop browsers.

C

- A range of common code constructs, objects , and functions, mostly appropriately used, with appropriate methods and designs for each task the code performs.
- Limited use of advanced techniques including objects, recursion, and regular expressions.
- HTML and CSS integration works, with some separation of data, process, and presentation.
- Adequately commented, mostly properly laid out, and generally maintainable.
- Some relationship with the purposes, personas, and scenarios of Unit 1.
- Limited error handling.
- The occasional small bug, logic, or run-time error.
- Mediocre interface design but generally usable and accessible.
- Works in most modern desktop browsers.

D	● Use of a limited range of common code constructs, objects, and functions.
	● Those that are attempted are appropriately used by and large.
	● Very limited or buggy use of advanced techniques including objects, recursion, and regular expressions.
	● HTML and CSS integration works, but may not be very elegant.
	● Some comments and mostly well laid out code, but limited maintainability.
	● Tenuous or sketchy but explicit relationship between the purposes, personas, and scenarios of Unit 1.
	● Poor error handling, the occasional show-stopping bug, logic, or runtime error, but mostly works well.
	● Works in at least two common desktop browsers from different families (e.g., two or more of Firefox, Chrome, Safari, Opera, and IE).
	● Mostly accessible and usable but some small problems here and there.