# Assignment 2

*CompSci 230 S2 2015*

*Clark Thomborson*

***Submission deadline: 4pm, Friday 1 May 2015***

***Version 1.0 of 15 April 2015: added Timer to Figure 1, adjusted wording of question 1a, corrected label in Figure 2, released hbbv2.01.jar to correctly implement the state diagram of Figure 2, added submission requirements to Q6, required the submission of a screenshot in Q7, removed the requirement for boldface font in Q8, revised the submission requirements in Q8 so that only two methods are listed.***

Total marks on this assignment: 50. This assignment counts 5% of total marks in COMPSCI 230.

**Learning goals**. Basic level of competency in the Java Collections Framework and the Swing Applications Framework. You will *continue* developing the skills and knowledge required for you to figure out "what needs to be changed" and "what should not be changed" in an existing app (such as hbbv2.0), so that it has fewer defects and more desirable features. You will not be expert in such tasks, after you finish this assignment, so you probably will not produce high-quality code. Rather than striving for perfection in your code, I'd encourage you to concentrate on "getting the job done to an adequate standard" and on "learning through experience".

**Getting help.** You may get assistance from anyone if

- you become confused by Java syntax or semantics,
- you don't understand anything I have written in this assignment handout,
- you have difficulty doing the initial (unassessed) steps in a question,
- you don't know how to take screenshots,
- you don't know how to cut-and-paste nicely-formatted listings of code from Eclipse into your assignment submission (hints: replace the tab characters with three or four spaces, or adjust the tab-stops; and be sure you're using a fixed-width font), or
- you have any other difficulty "getting started" on this assignment.

**Working independently.** The homework assignments of this course are intended to help you learn some important aspects of software development in Java, however they are also intended to assess your mastery (or competence) of these aspects. You must construct ***your own answer to every question***. If you are unable to complete the assessed work in any problem, I'd suggest you return to its initial steps – review the readings, and consider looking for additional readings in the Java Tutorials or on the web. Build some relevant sample code from the Java Tutorials, and fiddle with it to adjust its behaviour in some way, to confirm that you understand how to modify that code. If you are still unable to complete a problem after revisiting its initial steps, move on to the next problem – the required changes are (for the most part) independent so you can skip some problems and still complete later ones. You'll be submitting only your final codebase as a jarfile, so you will not lose marks if you remove defects and introduce features in a different sequence than the one I'm suggesting here (in which you'd complete problem 1 before starting problem 2).

**English grammar and spelling.** You will not be marked down for grammatical or spelling errors in your submission. However if your meaning is not readily apparent to the marker, then you will lose some marks.

**Resource requirements.** You will be using the same software as in Assignment 1, with the addition of the HuckleBuckle v2 codebase (available for download on the CompSci230 Assignments webpage). If you're using ArgoUML to produce class diagrams, you will also want to download hbbv2.zargo.

**Submissions.** You must submit electronically, using the Assignment Drop Box (https://adb.auckland.ac.nz/). Your submission must have **one document** (in PDF, docx, doc, or odt format) with your written answers to the questions in this assignment, and one jarfile with the **source** code you developed for the last question.

**Handwritten diagrams.** The scanners in the computer lab will help you convert any handwritten diagrams into images you can include in your submission. However you may have difficulty using the scanner interface the first time you try, so I'd advise you to scan and email a trial document at least *a day before* the submission deadline.

**Style and formatting.** The markers for this course have informed me that they are having difficulty interpreting some students' submissions, due to careless formatting and incorrect numbering. I have advised them to deduct marks from submissions which are unclear or ambiguous. Students should not be given "the benefit of the doubt" if a marker is unsure of what a student is trying to communicate. Students whose writing is unclear should, ideally, be given feedback (through marks, and marker comments) on any significant defects in the understandability or technical accuracy of their submission. Accordingly, to receive full marks: your answers must be formatted well enough to be easily interpreted by the markers. You should take care to number your answers accurately, and to express yourself clearly.

I do not expect you to have a well-developed sense of programming style in Java – this is a new language to most of you, and you have not had the time (or the opportunity) to learn the idioms and stylistic conventions of any of its dialects.

You were exposed (in the first assignment) to the stylistic convention that code-duplication is generally a "bad idea" (with respect to readability and maintainability) whenever it can be easily avoided by a refactoring. Accordingly, you will **lose some marks if you duplicate code** instead of re-using code (e.g. by invoking a method in an existing class or superclass, or by "extracting" a new method from a block of code in a class you are modifying). You are allowed to modify any line of hbbv2.0, and you may add new classes.

**Time budgeting.** I'd suggest you spend 10 to 15 hours on this assignment, so that you still have a significant amount of time (within a 10 hour/week budget for this course) during the month of April for your independent course-related study, and for your lecture and tutorial attendance.

**ADB response time.** The ADB system is sometimes unresponsive, when it is under heavy load due to an impending stage-1 assignment deadline. I will extend your submission deadline by up to one hour, if I see any evidence (from the submission logfiles) of ADB overloading. So: please be patient if the ADB seems slow to respond.

**Checking your own submission.** I strongly recommend you download a copy of your submission, after you have obtained (and retained!) a submission receipt from the ADB. It'll take you only a few minutes to confirm that you have submitted your latest set of answers to this assignment, and that your jarfile meets all requirements. The error rate on assignment submissions in COMPSCI 230 is about 1%, on a historical basis, because students occasionally submit an assignment from another class or the "wrong version" of a jarfile. This is only 4-sigma quality: is it good enough for you?

**Resubmissions.** You may resubmit at any time prior to 4pm Wednesday 6 May 2015. However your markers will see only your latest submission – you will lose marks unnecessarily if you make a partial resubmission, so please be careful to submit *all* required materials on each submission.

**Lateness penalties.** If you submit or resubmit after the deadline (4pm Friday), you will be penalised 8 marks (20% of possible marks) or 20 marks (50% of possible marks), depending on whether your submission or resubmission was within 72 hours of the deadline. No submissions will be accepted more than 120 hours (5 days) after the deadline. Lateness penalties will not reduce your marks on this assignment below 0.

# Part 1: Code Inspection and Reverse Engineering

1) **(5 marks)** *Initial steps, unassessed:* Import hbbv2.0 into your development environment, build it, and run it with no command-line arguments. The console output should show Sue playing a game of Huckle Buckle Beanstalk with Harry on a 5x5 grid. A graphic display window should show Sue's position (as a text-string) on a grey checkerboard.

   *Assessed work:*

   a) Inspect the source code of hbbv2.0 to discover how its classes are related. Modify the class diagram of Figure 1 (on the next page), so that your modified diagram accurately shows all inheritances, realisations (a.k.a. implementations), and associations between classes in this figure. Associations must show navigability and multiplicity. Your diagram must show at least one specialised association (i.e. an aggregation or composition), and you will document your decision to specialise this association in part b of this question. Your modified diagram should not show any additional instance variables or methods.

   **Submit** your modified class diagram as a figure in your submission document. This figure should have a caption indicating that it is your answer to question 1a, and that it depicts "the relationships between classes in version 2.0 of the Huckle Buckle codebase."

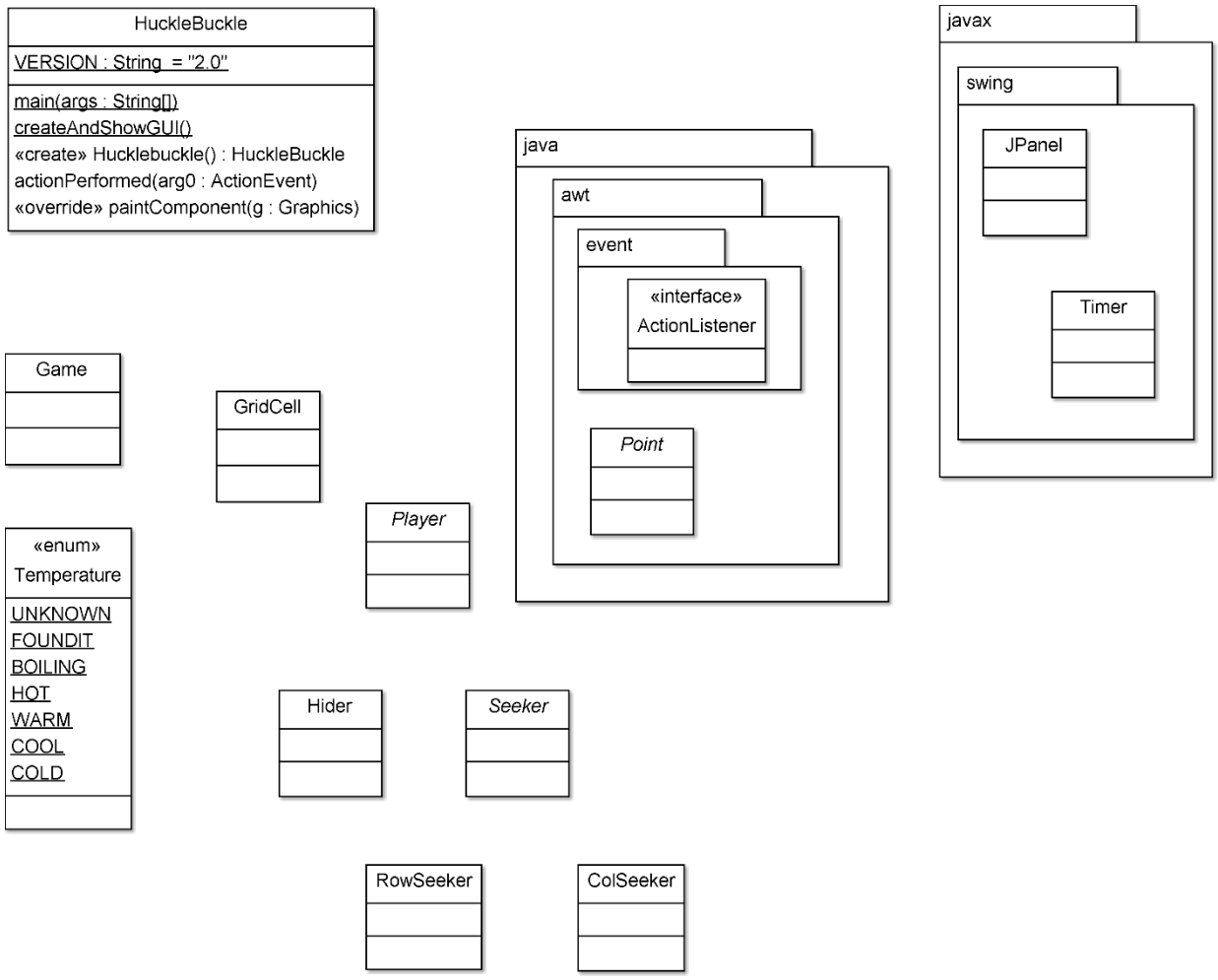


Figure 1. Classes in HuckleBuckle v2.0

   b) **Submit** an English-language paragraph, explaining (very briefly) why you specialised one of the associations in your class diagram into an aggregation or composition. To receive full marks, your paragraph must appropriately and understandably use the words "own" and "destroy"; it must clearly indicate which

association you are talking about; and it must give some understandable reason "why" you chose to specialise this association. Please also note that English-language syntax is *not* on our course syllabus, and that our markers are neither trained nor expected to detect, correct, or mark your minor syntax errors in e.g. the conjugation of English verbs (own, owns, owned, …). When answering this question, you are "implementing" English sentences, not Java statements; but in either language, an implementation is not acceptable if it is ambiguous or incomprehensible to its interpreter.

## Part 2: Learning about the Set interface

2) **(5 marks)** *Initial steps:* Read through The Set Interface lesson in the Java Tutorials, skipping over its discussion of coding suggestions for JDK 8 or later. Also read the block-comment following the JavaDoc definition of the Set<E> interface, and the block-comment for its add() method. Your goal in this reading should be to get a rough idea of the semantics of `Set`, and some idea of how to use these semantics in your coding: you should not attempt to memorise this material, but if you run across a sentence you don't understand, you should take note of this difficulty … then move on! I am encouraging you to learn about `Sets` mostly through your experience, rather than trying to learn the theory first and the practice later, because some of the information in this tutorial lesson and JavaDoc block-comment is quite advanced! In particular, software contracts are entirely outside the scope of this course. The JavaDoc entry is very careful to specify the "stipulations" in the "contract" which (formally) defines the behaviour of the Set interface, but I don't expect students in this course to take a formalist approach to software development. However… if you *are* curious about software contracts, I'd encourage you to read the Wikipedia entry on Design by contract.

   *Assessed work:*

   Read the TODO comment in the `HuckleBuckle()` constructor, and also the TODO comment in the Player class. These comments refer to a defect in the behaviour of `hbbv2.0`. The `HuckleBuckle` constructor attempts to add a second Seeker to the set of allSeekers, but this `add()` operation fails – for a reason which you should at least vaguely understand from your initial steps on this problem.

   Create `hbbv2.1` with an improved implementation of the `Player` class (but no changes to any other class, aside from the removal of the TODO comment in HuckleBuckle's constructor), so that multiple seekers can play `HuckleBuckle`

   **Submit:** a listing of all methods which you changed or added, when revising the `Player` class in `hbbv2.1`.

   To receive full marks, there should be accurate and informative block-comments on all methods in your listing. There should be no duplication of code, that is, you should call a method rather than duplicating its functionality in another method. Note: you should not attempt to determine the "best" semantics for Player, because there are several "reasonable choices" for an improved Player semantics which would allow Sally to join the game as a Seeker (even though there is already one Seeker in the set of seekers).

3) **(6 marks)** *Assessed work:*

   Create hbbv2.1a. This is an "experimental" branch of your `hbbv2.1` codebase, in which you will try various implementations and types for the `allSeekers` variable in the constructor for the `HuckleBuckle` class. Note that this is a local variable; you should review slide 21 of the seventh set of lecture slides for a rough idea of the semantics of such variables. Also note that there are just two "uses" of this variable: in the invocation of `setSeekers()` of the `Game` class, and in the invocation of the `pleasePlayWith()` method of the `Hider` class. There cannot be any other uses because a local variable isn't accessible outside the method in which it is declared.

Try the other two implementations for Set in the Collections framework: TreeSet and LinkedHashSet, by calling the relevant constructor in the third line of the HuckleBuckle constructor. (Of course, you'll have to import any package you use!) Do these changes cause any compilation errors, any runtime errors, or any observable change in runtime behaviour?

Try a few other types for `allSeekers`: it could be declared as a `Collection<Seeker>` rather than a `Set<Seeker>`, it could be declared as a `List<Seeker>`, and it could be declared as a `List`. After you adjust the import statements, are there any compilation errors in any of these three cases? If there were any compilation errors, can you easily repair the defect by making simple changes (such as selecting a more appropriate constructor) to the HuckleBuckle class, or is it necessary to change other classes as well? If you have to change other classes, are these changes "improving" the whole codebase, or are you merely changing multiple classes in order to adapt them to your new HuckleBuckle implementation (with the risk that some future developer might want to "change them again" to suit some other implementation)? After you get each of these three experimental ver**rs**ions to run, determine whether or not there is any observable change in runtime behaviour as a result of your changes to the type of the `allSeekers` reference variable.

Now review the paragraph in The Set Interface lesson of the Java Tutorials which starts with the phrase "Note that the code always refers to the `Collection` by its interface type…"

**Submit:** a paragraph discussing your experimental findings, and your recommendations (if any) for changing the way the Java Collection Framework is used in hbbv2. Note that you are experimentally evaluating the extensibility of hbbv2, by determining whether or not some "simple" changes to the HuckleBuckle constructor could be made without causing the codebase to "break" (until other classes are changed to suit a changed implementation of HuckleBuckle).

## Part 3: An Experiential Introduction to State Diagrams

4) **(6 marks)** *Initial steps:* Examine the source code for the `Seeker` class of `hbbv2.0`, and look at the state diagram of Figure 2 on the next page. Note that we haven't discussed UML state diagrams in this course, and they are not examinable except with respect to what you'd reasonably learn about them from this question. However I think you'll find this diagram helpful in understanding – possibly with someone else's assistance – the "life cycle" of a `Seeker`. Unlike a real child, a Seeker immediately starts `WAITING` after it is created, and (after receiving certain method calls) it shifts to other states such as `STARTING` or `MOVING`.

To confirm (or develop) your understanding of this state diagram, set a breakpoint (using Eclipse, possibly with someone else's assistance) at the first line of the Seeker's `move()` method. Examine the `myState` member of the Seeker every time the breakpoint is encountered. Hint: if your `Seeker` override its `toString()` method by appending `myState` to the result of its overridden ("`super`") `toString()` method, then it's easier to monitor this field in the Variable window of Eclipse's Debug view. Also note that you can use the F8 key to "resume" processing after stopping at a breakpoint.

*Assessed work:* Create a new project called `hbbv2.2`, and import your `hbbv2.1` source code (and not your `hbbv2.1a` experimental code!) into this project. Add a new state, `ASKING`, to the `SeekerState enum`. Adjust the implementation of the Seeker's `move()` method, so that a `Seeker` will enter the `ASKING` state (instead of the `SEEKING` state) after she has taken enough steps to reach her current destination `(nextX, nextY)`. In the `ASKING` state, the seeker should ask the `Hider` for her current temperature. If her temperature isn't `FOUNDIT`, then she should proceed to her (revised) `SEEKING` state – in which she either determines new values for `(nextX, nextY)`, or decides to quit, depending on whether or not she has already looked at all possible hiding places. Note that the effect of this change is to slow down the seeker's progress, for she'll spend two timer-ticks

in each cell before moving to the next. This change also makes it possible for a new type of seeker (which you'll develop later in this assignment) to move rapidly through a cell (i.e. without `ASKING` the hider for its temperature) if the temperature is already known.

Hint: if you're having trouble debugging and you had based your development on `hbbv2.0`, I'd suggest you consider looking at `hbbv2.01`. In that code I have correctly implemented the state diagram of Figure 2 in the `move()` method, I have deleted the `moveTo()` method, and I have adjusted the output to indicate that a `Seeker` takes just one step at a time.

**Submit:** a listing of your revised `move()` method; and a listing of the console output for a run of `HuckleBuckle` v2.2 with its default arguments. You should *not* submit listings of the other methods you changed when creating `hbbv2.2`: you will be marked only on your `move()` method and your console output. If you are unable to your program to work correctly, you should still submit your revised `move()` – because the majority of marks are based on this, and not on the output listing.

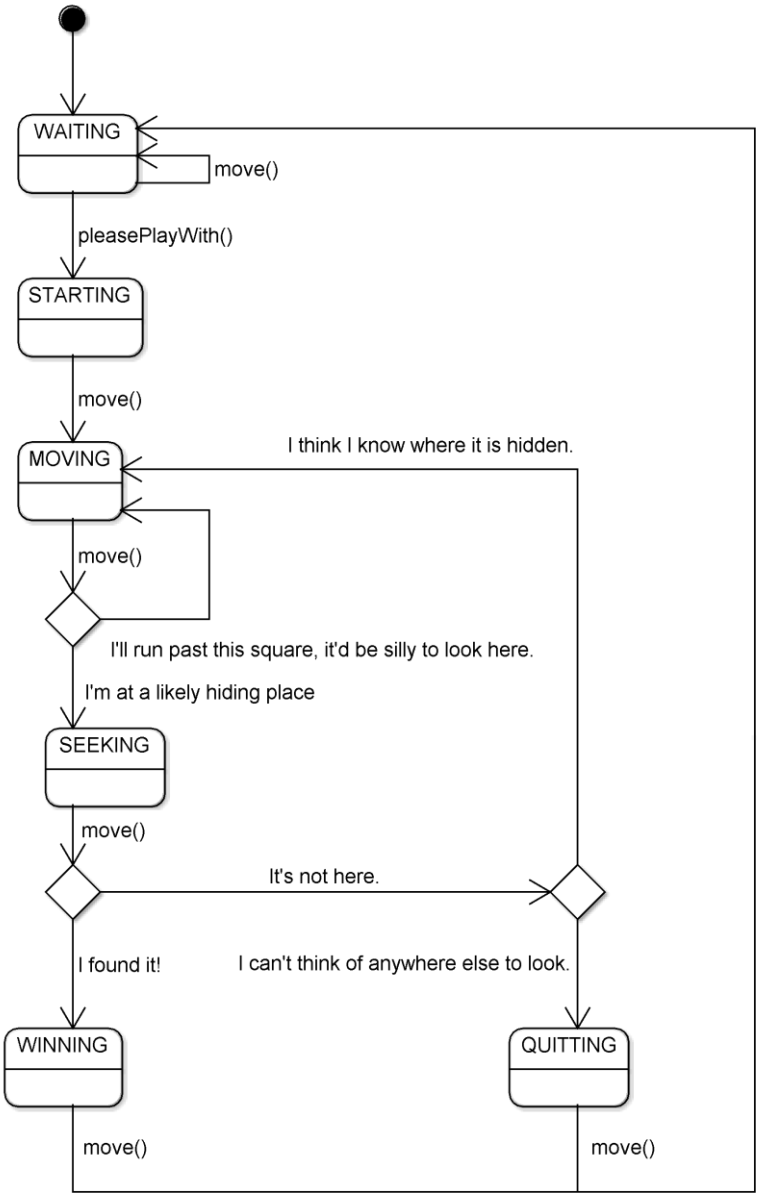


**Figure 2. State diagram for Seeker in hbbv2.0.**

## Part 4: Adding Three Buttons

5) **(6 marks)** *Initial steps:* Read the first few pages in the Java Tutorials lesson on Using Swing Components, skipping the pages on Using Text Components and the page on How to Make Applets, and stopping after you finish the page on How to Use Buttons, Check Boxes, and Radio Buttons.

   Import, compile, and run the Top-level Demo, as suggested in your reading. The setup code for even a simple Swing app, such as this one, is complex. There are many optional parameters, but (eventually!) you will recognise the idioms in Swing-app setups. There aren't very many of these idioms, and none are examinable. Don't despair! Almost all Swing apps are programmed "by example" and I won't expect you to write full-custom setups, nor will I expect you to understand "what's going on" in any setup other than the one in `hbbv2.0`.

   Read the following notes on my theory of code development:

   - The starting point for most software development projects is a working codebase from some previous project. Incompetent developers will sometimes start from an example they discover through a web-search, without considering whether or not this is a trustworthy baseline for their development. Strong developers will always base their development on codes that are trustworthy, i.e. ones which have been carefully constructed by a reputable developer and are obtained from a trustworthy source (e.g. the Java Tutorials). There are many possible starting-points for a new development effort, and it is (of course) most convenient to select one which is vaguely like the one you're currently trying to develop. After confirming that the baseline software is working properly after its import, a competent developer will generally make a series of small changes to this software by adjusting its parameters, and by adding and deleting method-calls, until their app's baseline functionality (e.g. a display window with no rendered content) is appropriate for the task at hand. Features can then be integrated, sequentially, into this application; after each integration, the developer can (and should!) confirm that it is "still working", and they soon learn to be *very* careful to retain copies of previous versions -- just in case they want to "start again" on the current feature, rather than to continue on their current (unpromising, possibly even hopelessly-botched) current line of development on this feature.
   - I believe that scaffolding from a standard template, followed by idiomatic ("boilerplate") customisation, is the usual case in competent professional practice. This practice improves maintainability, because other developers will easily recognise your idioms, rather than puzzling over your "clever" coding of a function-point, and becoming annoyed when they realise this could have been handled idiomatically. It also improves correctness, *if* the professional is competent enough to deploy an idiom only in situations where it is appropriately deployed. Trainee software developers have a significant advantage over trainee lawyers with respect to correctness, because our compilers and our run-time systems will help us detect errors in our pre-release code. By contrast, a trainee lawyer must seek expert advice when they are unsure of the legal implications of their drafts.

   Now try to import, compile, and run the Button Demo, as suggested in your reading. You're likely to run into difficulty with this step. The downloadable zipfile for this project was developed in NetBeans; and you're using Eclipse. You would have to rearrange the files for compilation in Eclipse, if you import the entire Netbeans project. This is certainly possible to do, and it's not very difficult (after you learn how) to import foreign projects into a development environment. However this is a very simple project with only one source file and three resource files. I suggest you keep it simple, by downloading the ButtonDemo.java file and importing this into a new Eclipse project. You can then try to import the gifs; and if you aim Eclipse at your `ButtonDemo` project and ask it to import the imagery (three `gif` files), these "should" end up in the right place – as in Figure 3 below. However if you can't get the imagery to load, don't despair, we won't be rendering any images in this

assignment, so we can just 'hack these out' of the demo so that it will compile and run without imagery. To do this, you can comment-out (or delete) the first three lines of code in the `ButtonDemo()` constructor, so that this method makes no references to `gif` files. Then you should replace all uses of the three `ImageIcon` reference variables by `null`. For example:

```
b1 = new JButton("Disable middle button", null);
```

Now that the images have been hacked-out of the demo, it should compile and run in Eclipse without errors.

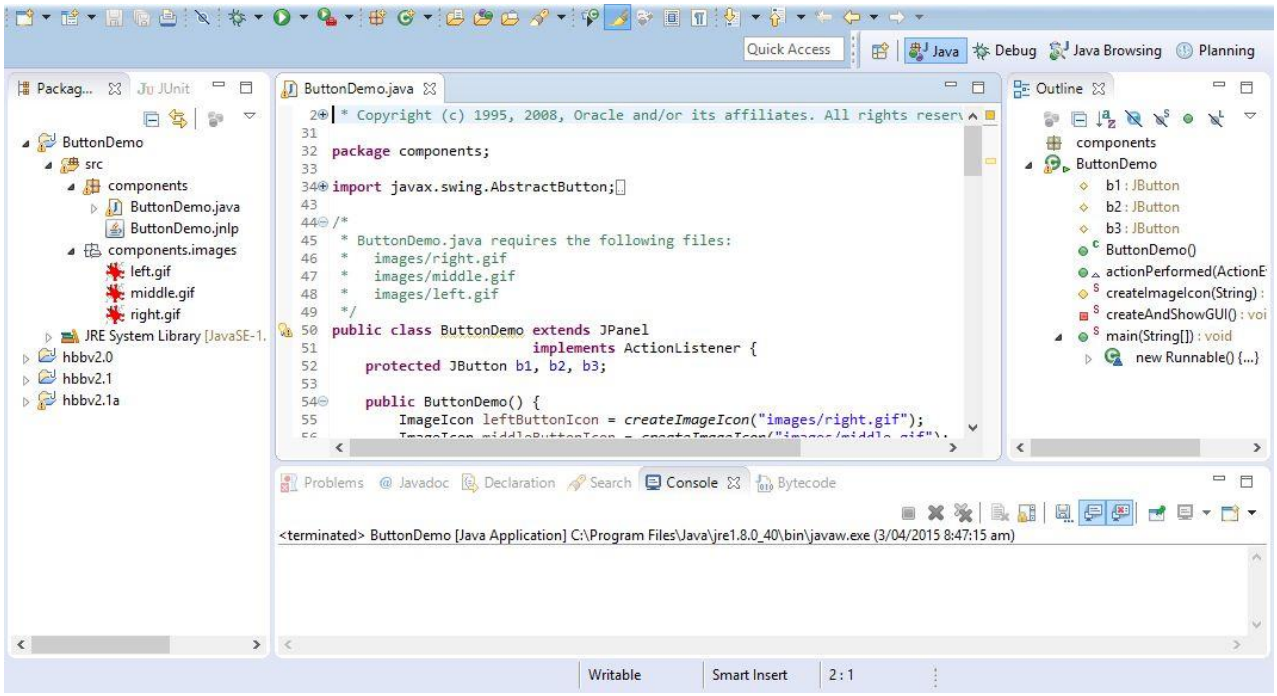


**Figure 3. ButtonDemo, as an Eclipse project.**

Confirm your understanding of the `ButtonDemo` by adding a fourth button, using the idioms you find in this code for setting up its three existing buttons. You should not memorise any of these idioms – they are not examinable! As a professional developer, you will "copy" idioms many times before you "learn" the ones you are commonly deploying in your code. Furthermore, very few of you will ever become Swing developers, and memorising Swing idioms would be a complete waste of time for anyone who isn't a Swing dev. So… instead of memorising anything, I'd encourage you to concentrate on understanding "what's going on" in this code. Identify the idiom which instantiates a button, the idioms which modify its default behaviour and appearance (e.g. by labelling the event that is thrown when this button is clicked by the user), the idiom which registers it as a listener, and the idiom which adds the button to its GUI container (this is a `JPanel`). Also look at how these button's `actionPerformed()` event handler is coded.

*Assessed work.* Create a new project, import your previous version of the `hbbv2.x` codebase, and modify this code until it displays three buttons. One should be labelled "Pause", one should be labelled "Resume", and one should be labelled "Single Step". The action handlers for these buttons should not actually pause, resume or single-step the game; however you should

- place these buttons along the top edge of the `JPanel` (with the grid displayed below),
- assign appropriate labels to events (e.g. "pause" for the Pause button)
- assign appropriate keyboard shortcuts (aka mnemonics), e.g. "P" for the Pause button. (Note that, by convention, an underlined letter in a menu or button label indicates the keystroke which will invoke this button; this demo is defective in that it responds to an `<Alt>p` but not an `<Alt>P`, even though the "P" is capitalised in the "Pause" label on its Pause button.)
- ensure that the Resume button is initially disabled, and is enabled after Pause is clicked

- ensure that that the Pause button is initially enabled, and is disabled after Pause is clicked

When answering the next problem, you'll be modifying the button enable/disable behaviour, so you should not attempt to develop appropriate button-click behaviour (aside from what is specified above) in this version of your code.

**Submit:** a listing of any method you modified or added, when implementing the features described above, and two screenshots showing the appearance of your app's GUI window before, and after, the Pause is clicked for the first time.

6) **(6 marks)** *Initial steps:* Read the Java tutorial on How to Use Swing Timers, and the block comment in the JavaDoc for the javax.swing.Timer class.

*Assessed work.* Invoke the stop(), start(), and setRepeats(false) methods on the game timer, in the actionPerformed() handler for your buttons, so that your buttons have the appropriate behaviour of pausing, resuming, and single-stepping the simulation of a game of Huckle Buckle Beanstalk. To receive full marks, your app should disable any button which shouldn't be pressed, e.g. the Resume and Single-Step buttons should be disabled when the simulation is running.

**Submit:** a listing of any method you modified or added, when implementing the features described above, and two screenshots showing the appearance of your app's GUI window before, and after, the Pause is clicked for the first time.

## Part 5: Adding Instance Variables and Methods to enums, and Custom Painting

7) **(6 marks)** *Initial steps:* Read the Java tutorial on enum types, concentrating on the `Planet` example. Make sure you understand the syntax of an `enum` class which has `private final` instance variables (in this case, `mass` and `radius`), getters (in this case, `mass()` and `radius()`) which allow these variables to be read, and a constructor (in this case, `Planet(double mass, double radius)`) which is invoked, at compile-time, when the `public static final` instances of the `enum` class are created from the comma-separated list at the beginning of the body of the `enum` declaration.

Now read the Overview of the Java 2D API Concepts in the Java Tutorials, stopping when you reach the Images page – we won't be rendering any images in this assignment, nor will we be rendering graphics to a printing device (we'll only be rendering geometric primitives and text to a display device).

Now examine the `paintComponent()` method of `HuckleBuckle`, to discover its use of a `public static final` instance of *`awt.Color`* as an argument to the `setColor()` method of a `Graphics` object, when it is setting the colour of a pen (a.k.a. paintbrush). This pen will be used by our GUI's `Graphics` object, when we invoke its `fillRect()` method to render rectangular shapes to its display device.

Now work through the Performing Custom Painting lesson of the Java Tutorials, stopping after you complete Step 3. Don't worry about the clipped `repaint()` calls in this assignment – the custom-painting code in hbbv2.0 is very inefficient in its repaints, and you are not required to remedy this defect. (This inefficiency is not a fatal defect, because the `Timer` in `HuckleBuckle` throws only a few events per second, and there are only a few dozen invocations of `Graphic` methods per `repaint()` of the entire window. Maintaining such a simple graphic display, at such a low frame rate, is not a significant CPU load on any modern computer.) Note that the `repaint()` invocation in `HuckleBuckle` is made by its `actionPerformed()` method, which is registered (by a `setTimer()` invocation in the `HuckleBuckle` constructor) to handle the events thrown by a newly-

constructed instance of Timer. Also note that an instance of `HuckleBuckle` is added to a `contentPane`, in the `createAndShowGUI()` method of `HuckleBuckle`.

*Assessed work*. Develop a new version of Huckle Buckle, by modifying the `Temperature enum` in your previous version so that it associates a `Color` with each instance of the `enum`, and by modifying the custom-painting code in HuckleBuckle so that it paints a temperature-appropriate color on each square after Harry has revealed its temperature to either of the seekers.

**Submit**: a listing of your `Temperature enum`, a listing of any custom-painting method that you modified (or added), a listing of the method(s) you modified or added when recording the hider's temperature reports in the Game object, and a screenshot showing at least four different colors for GridCells.

## Part 6: Should a Player have a Point, or should she be a Point?

8) **(4 marks)** *Assessed work:* Consider the appropriateness of a significant change to the Huckle Buckle codebase: a Player might have-a Point rather than being-a Point. As discussed in lecture, it is generally a bad idea to use inheritance, when a composition is feasible. Explore this design decision, by creating a new version of Huckle Buckle in which a Player has-a Point. Keep a record of the lines of code you change, when creating this new version. **Submit:** a paragraph evaluating this design change: does composition (of a Point, when defining a Player) have any significant advantages or disadvantages over inheritance? Also submit a paragraph discussing any significant difficulties (such as a large number of code-changes) you encountered, when shifting this inheritance to a composition.

## Part 7: Only one seeker per GridCell!

9) **(6 marks)** *Assessed work:* Adjust the "rules of the game" so that, with the exception of the initial cell (0,0), a seeker is not allowed to execute its translate() method if this would put it into the same cell as some other seeker. A seeker can thus get "stuck" in its MOVING state, until another seeker moves out of the way; and there can be only one winner in a game (unless the object is hidden in the initial cell).

- You **should** add a counter to the MOVING state, so that a seeker will enter the QUITTING state if she can't move for an extended period of time (say, 8 timer-events).
- You **should** test your program with many seekers; so you are required to add a second command-line argument which defines the number of seekers (default two; maximum 10; minimum one).
- You **should** add a new type of seeker, one which "looks ahead" when picking her next goal (nextX, nextY) – she shouldn't enter an ASKING state on any cell whose temperature is already known. You will not be marked on any additional "cleverness" you add to this seeker, so I'd encourage you not to get "too clever" until you have completed a version with the required features.
- Seekers **should** be of diverse types in a multi-seeker game, so your app should instantiate your clever seeker first, then a row-seeker, then a col-seeker, and repeating this sequence until it has instantiated the number of seekers specified by the command-line argument.
- Seekers **should** output informative messages, so that the console output is diagnostic of their current position and state.
- You **may** base this development on the code you developed before starting question 8, that is, your Players may be Points, rather than having a Point.

- **Submit:**
    - A listing of the move() method of your modified Seeker
    - A listing of the tryNewPosition() method of your “clever” Seeker subtype, and
    - A sample of the console and GUI output of your final codebase, for some interesting set of commandline arguments (e.g. perhaps with four seekers on a 7x7 grid), with the snapshot of your GUI being taken at some interesting time in the simulation.

In addition to your written answers (as specified above), you should also **submit** a jarfile containing your **source code** for your final version.