Objectives Students will be able to…

* **Create** simple programs with comments and style.
* **List and apply** the steps necessary for avoiding syntax errors.

Assessments Students will...

* **Complete** a worksheet
* **Develop** a personal checklist for spotting syntax errors

Homework Students will...

* **Read** HW 1.4
* **Complete** Ch.1 Exercises 6, 7, 9

# Materials & Prep

* **Projector and computer** (if you are able to/opt to use Eclipse with your students)
* **White paper** **and** **markers**
* **Classroom copies** of WS 1.4
* **Sample punched card** to pass around (available on eBay: <http://tinyurl.com/nnthazu>)
* **Pictures**:
  + - * Punch cards (<http://tinyurl.com/n9zqd3k>)
      * Readers (<http://tinyurl.com/p34mvmb>)
      * Jaquard loom (<http://tinyurl.com/n8tmra3>)
      * Bug (<http://tinyurl.com/ljyguuy>)
* If you are able to laminate student work, or have plastic sleeves available for students that have binders, it would be a good idea to reinforce/preserve student error-correction algorithms (see today’s Activity). Students should be referring to these sheets often in the first few months of the course, so they will get a lot of wear & tear.

# Pacing Guide

|  |  |
| --- | --- |
| Section | Total Time |
| Bell-work and attendance | 5min |
| Vocabulary and history of bugs | 10min |
| Error-checking algorithm | 10min |
| Worksheet | 15min |
| Students trade work, check, and turn in | 5min |

# Procedure

*Today’s lesson will be a combination of drilling the parts of a basic program, and conditioning students to check for common errors. To hook your class, have pictures of punch cards and punch card readers up when students enter. If possible, have physical punch cards available to pass around the room for tactile learners as you explain the origins of the phrase “bug” and “debugging.”*

## Bell-work and Attendance [5 minutes]

## Vocabulary and History of Bugs [10 minutes]

1. Begin with a lecture about the history of computing with punch cards and the origins of “bugs.”

* Before computers had keyboards or touchscreens, all data was input using physical punch cards (pass around cards). In some systems, punch cards were used all the way up through the 1980s!
  + The holes in the cards represent a “0” and the locations without a hole store a “1.”
* Punch cards were originally designed for use in a mechanical loom invented in 1801 (show pictures of loom & tapestry design).
* When something wasn’t working in the physical punch cards that coded the program, users would look for actual bugs in the system (show bug picture).
* Nowadays, since all of our code is digitally stored as 0s and 1s, a “**bug**” means we wrote the code incorrectly. Today we’re going to create checklists of things to look for in our code to make sure its working correctly—or “debugging” our code.
* **Syntax errors**: when you don’t follow the ordering rules of writing Java code, when you misspell something, or leave out punctuation.
  + Analogy: in English, we say “the black bear.” In Spanish, you’d say “el oso negro,” and in Italian “l'orso nero,” both translate to “the bear black.” There are different rules for how you order your words in different languages, and Java has its own set of language rules too. If you write the equivalent of “the bear black” in Java, Java won’t understand it, and you’ll get an error message. (Have students give you an example.)
  + You can also create confusion by writing/saying “the balck bear” (a misspelling), or “the! black, bear?” (incorrect punctuation)
* **Logic errors:** sometimes you might write code that has the right syntax, but doesn’t do what you meant for it to do. In this case, the program will run, but you won’t get the right output. An example of this would be if you wrote a print statement instead of a println.
* **Runtime errors:** these errors can happen if you give Java a code that has no solution, or accidentally causes the computer to calculate an infinite loop.
  + In science fiction, this is usually the way to shut down the evil computer that has come alive to take over humanity. Examples could be asking Java to calculate pi to the last digit, or dividing by zero.
  + If you want to share examples with your class, navigate to this cued Star Trek video clip:

(<https://www.youtube.com/watch?v=5VZRdAUbgCk&feature=youtu.be&t=1m9s>)

or invite students to scan through this list:

(<http://tvtropes.org/pmwiki/pmwiki.php/Main/LogicBomb>)

## Error-Checking Algorithm [10 minutes]

1. Have students distribute paper and markers while you explain that students are going to create a personal algorithm (or specific list of steps) that they will follow each time they write code. A sample algorithm might look something like this:

STEP 1: Check all code for spelling errors.

STEP 2: Check all code for punctuation errors (curly brackets, brackets, parentheses, semicolons).

STEP 3: Check all code for syntax errors.

…

2. Encourage students to write the algorithm as a **checklist, decision tree, or mindmap**. Explicitly contrast the flexibility of the human brain when compared to computers.

*Encourage creativity here—some students may color code their list, or take the assignment home to work on lettering, illustration etc. What may feel like wasted time is actually a spatial and tactile activity that helps students reinforce and memorize the steps needed to check code. The more ownership students take of this list, the more likely they are to use it over the next few months, which will make error-checking habitual.*

3. If you do not have classroom copies of the textbook, list the following errors on the board as required steps for students to have on their code-checking “algorithm.” If you feel that you have enough time, have students put these on the board.

* File name matches class name
* All code is spelled correctly
* All code is capitalized correctly
* All statements end in a semicolon
* Keywords are included
* Strings are enclosed in “quotation marks”
* There are no extra punctuation marks
* All header open-braces are paired with closed-braces

## Worksheet [20 minutes]

1. For 5 minutes go over documentation and proper commenting. Also go over identifiers, camelCase, and do a short introduction to style. Style will be covered in 1.08, but it’s important that they are introduced to it here.

2. Students have 15 minutes to complete WS 1.4. As they solve each problem, students should apply their personal proofreading algorithm to help check their solution for correctness.

## Students trade work, check, and turn in [5 minutes]

1. At the end of class, have students trade their worksheets to check each other’s answers before turning in the worksheet.

# Accommodation and Differentiation

While all students should write their OWN algorithm, you should encourage students to work in pairs or small groups so they can share ideas and help each other organize their thoughts. This is particularly important in ELL classrooms, where emergent English speakers can pair with advanced English learners. If some students want to do this project all on their own, let them.

If you have students who are speeding through this lesson, you should encourage them to:

* Create a mnemonic or acrostic to remember all the steps for checking syntax errors
* Make a poster for the classroom illustrating the mnemonic or acrostic
* Help another student with the worksheet (explain, not solve-for-them)

# About Error Checking in Eclipse

If you are able to use Eclipse with your students during this class period, you may opt to show your students how to interpret the error indicator.

If you do give your students the Eclipse tools at this time, beware that your students will probably copy and paste between Eclipse and Practice-It. On some systems, this may translate the span character into a Unicode space character. This will cause code that worked in Eclipse to produce errors in Practice-It or vice versa. To avoid student frustration, be sure to demonstrate how to correct copied/pasted code before submitting to Practice-It.