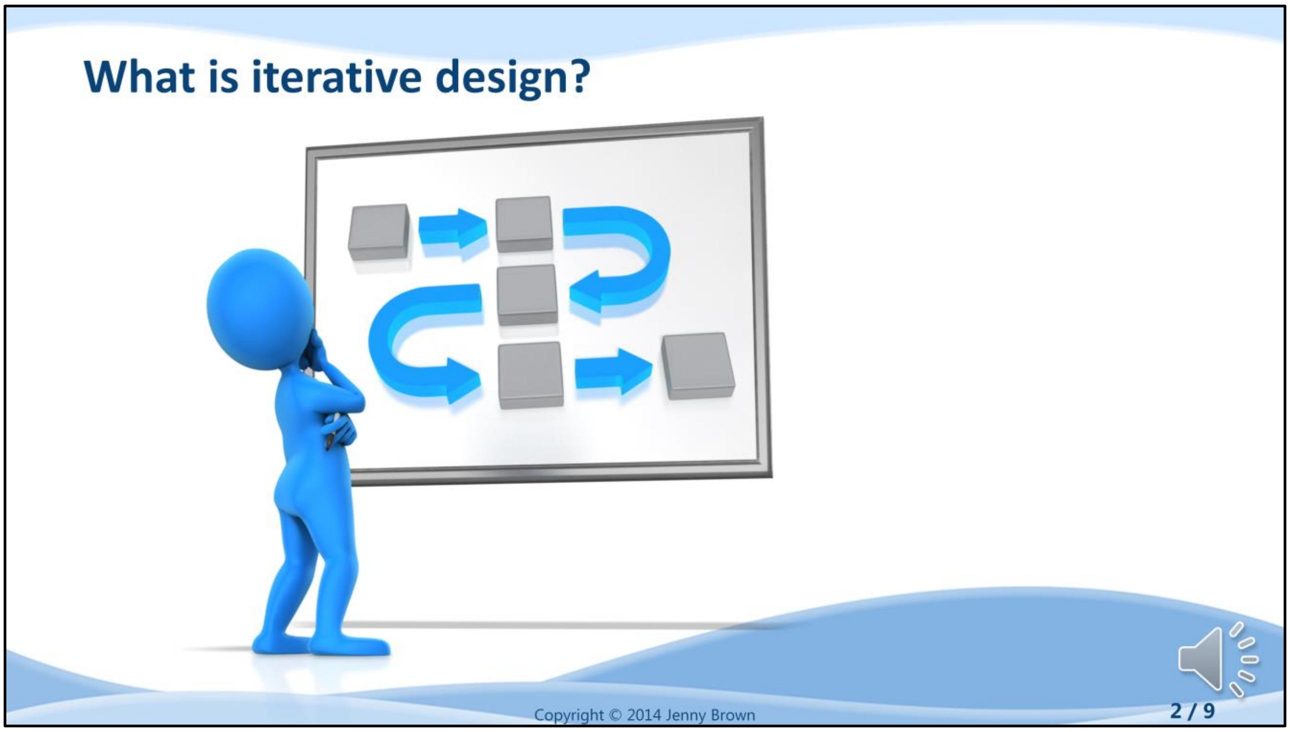
Iterative Design

Iterative design is a way of planning and improving software. This video helps you learn to plan larger programs, especially software for other people.



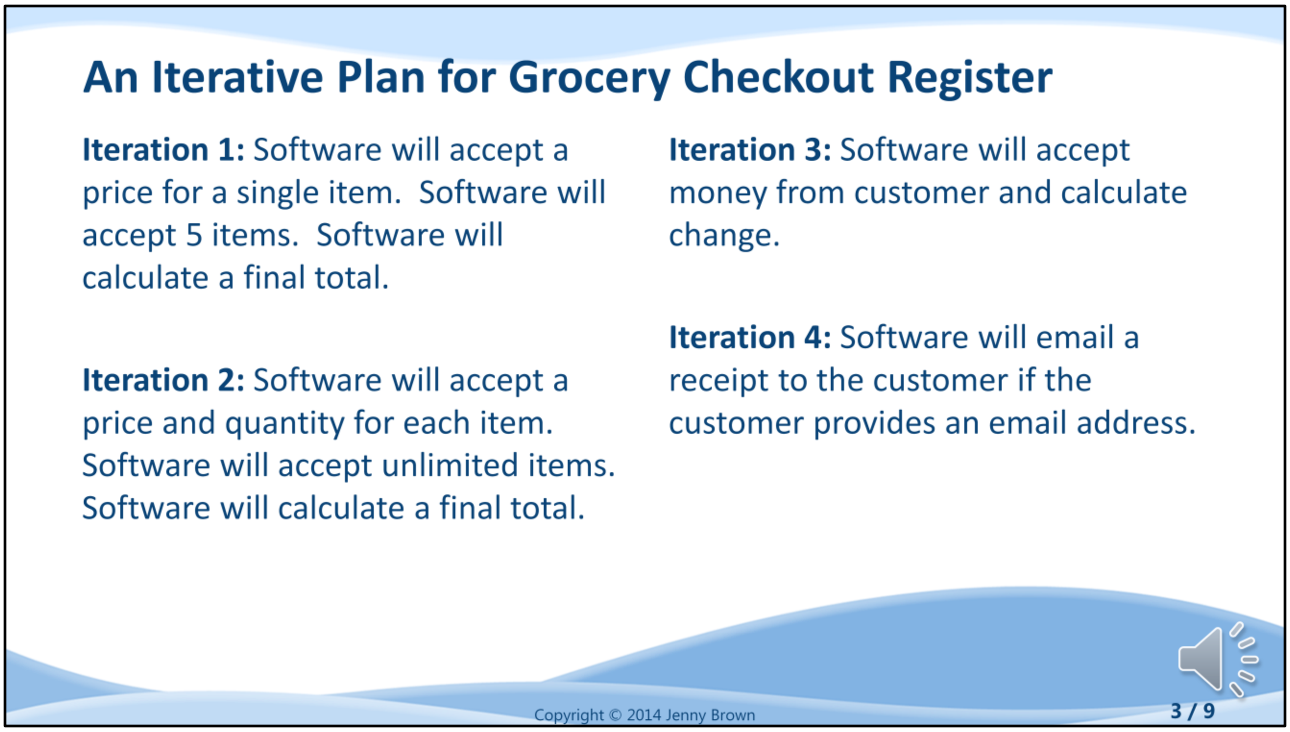
What is iterative design?

2 / 9

What is iterative design? A design is a plan. An iterative design is a plan you keep improving as you make progress towards it.

The word "iterate" means to do something again and again, or to do each of several small steps one after another.

In the context of software development, an iterative design means gradually enhancing a program to be better and better. It can also refer to adjusting your plan before you start making software. In the first iteration, you come up with the overall general idea, and a small set of behaviors to work on. In the next iteration, you figure out what else is still needed and improve on your earlier work. Each iteration makes the result better and more complete. At some point, you decide it's done enough, and tested enough, and it's useful as is.



An Iterative Plan for Grocery Checkout Register

Iteration 1: Software will accept a price for a single item. Software will accept 5 items. Software will calculate a final total.

Iteration 2: Software will accept a price and quantity for each item. Software will accept unlimited items. Software will calculate a final total.

Iteration 3: Software will accept money from customer and calculate change.

Iteration 4: Software will email a receipt to the customer if the customer provides an email address.

Here's an example of an iterative plan for a piece of software. This is software to operate a cash register.

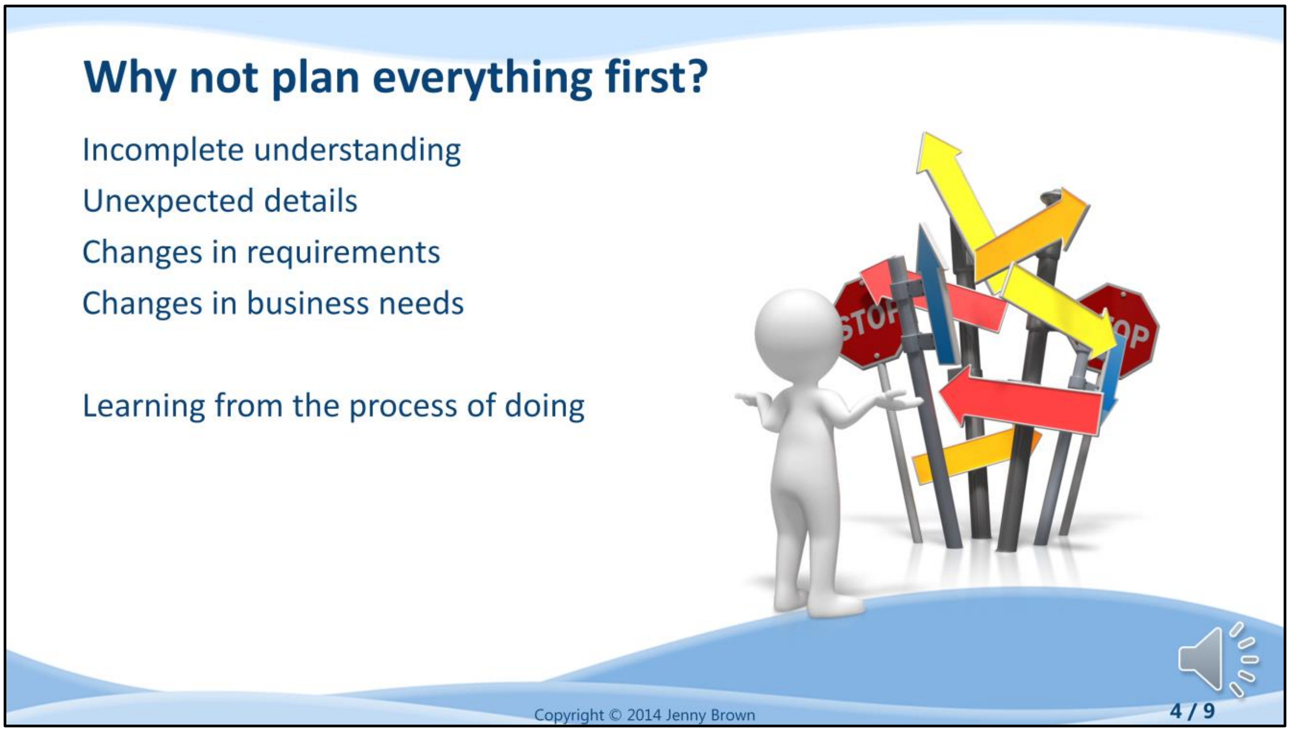
In the first iteration, the software will accept a price for a single item at a time, and up to 5 items. It will add up the price of the 5 items to come up with a total cost.

Then, in the second iteration, the programmer will change the same software. Now it will accept a price and a quantity for the items. So if there are 3 loaves of bread at $2 each, it will know that means $6 total for the bread. It will still add up the totals across all the groceries to produce a final total. But this time, instead of accepting just 5 grocery items, it will accept as many as needed.

In the third iteration, the programmer will change the software again. Now it will also accept an amount of money from the customer and calculate how much change to give them.

Finally, in the fourth iteration, the software will also gain the feature of emailing a receipt to the customer.

See how each iteration builds on the previous work, and adds new behaviors? This lets the programmer focus on smaller, easier steps each time.



Why not plan everything first?

Incomplete understanding

Unexpected details

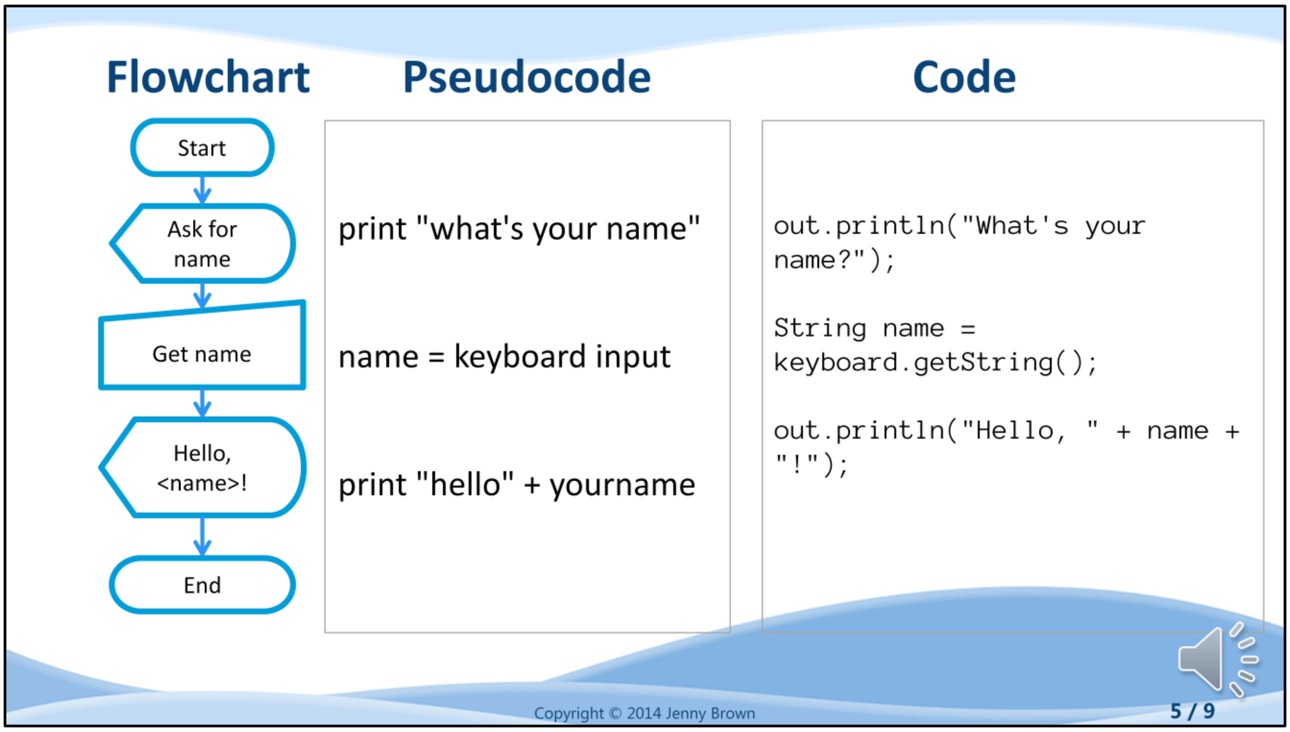
Changes in requirements

Changes in business needs

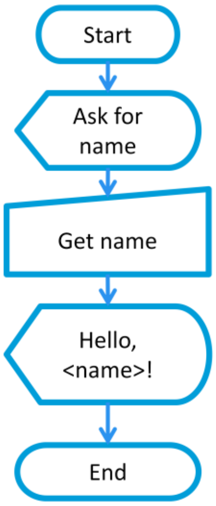
Learning from the process of doing

Why not plan everything first, get it right, and then write the software? It turns out, that's actually fairly difficult to do, especially as software gets bigger. Most of the time, you'll be working on a project where you have an incomplete understanding of what's needed and how to do it. You'll want the option to adapt as you learn more. You may encounter details in the code that aren't what you expected, and so your earlier plan isn't on target anymore. Once you're building software on the job, for other people, you'll find that during the time it takes to build the software, the business needs and technical requirements sometimes change, so you need to adjust it to a different goal.

You'll also find, especially while you're a student, that you learn a lot from the process of coding, and that improves your understanding of your own plan. It's good to start out with a plan. It's necessary, even. And once you start to build, you'll find that your plan also needs improvements. That's fine! It means you're paying attention and thinking carefully.



Flowchart



Pseudocode

```
print "what's your name"

name = keyboard input

print "hello" + yourname
```

Code

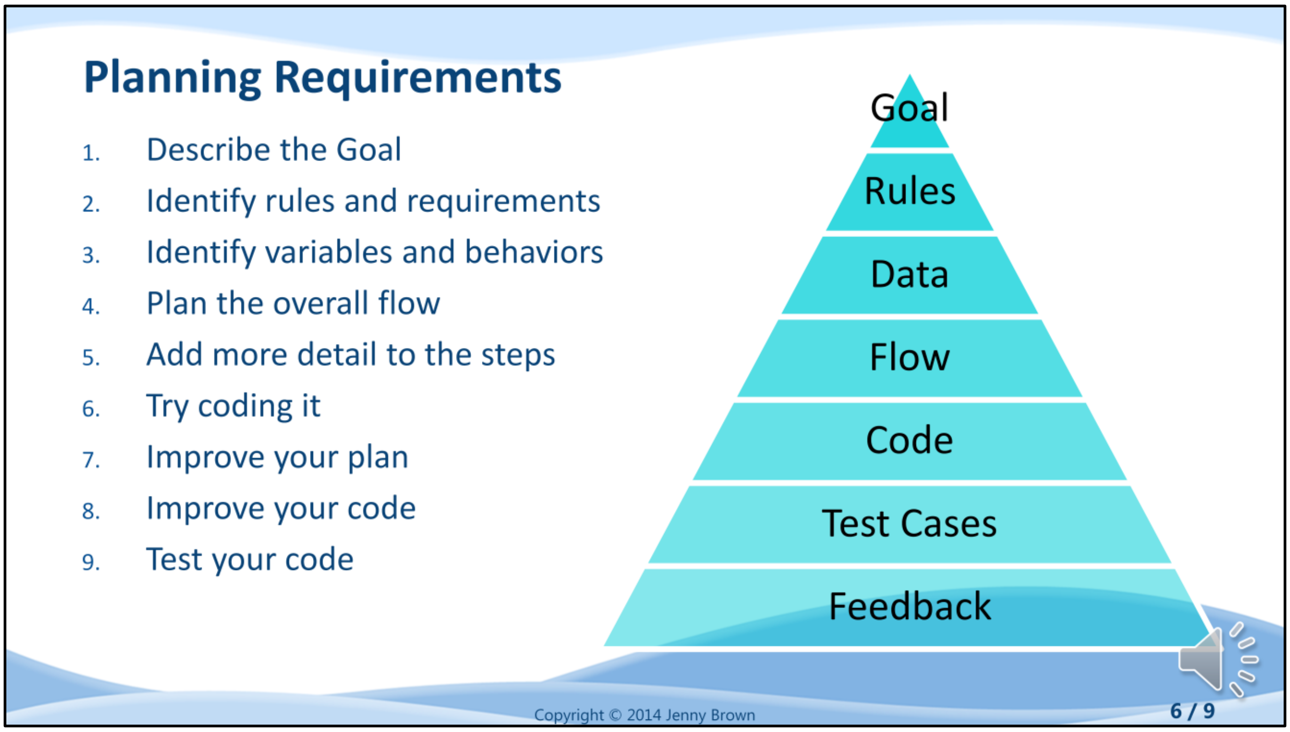
```
out.println("What's your
name?");

String name =
keyboard.getString();

out.println("Hello, " + name +
"!");
```

5 / 9

Pseudocode is a way of writing English words and phrases as if it were source code. Pseudo, here, means fake; basically you're writing fake code that still accurately describes what the real code will do, but without fussing about exact syntax or grammatical details. Developers often write pseudocode as a way to think through an approach and decide if it will work. They also use it to describe what code does without having to slow down to read actual code.

You can start writing pseudocode by adding detail beside a flow chart, like shown in this slide. You can also write pseudocode without creating a flowchart first. It's meant to be a tool to help you think and plan, and you can write it any way that makes sense to you. The main thing is to focus on small, step by step instructions, so that it's a good fit to source code when you're ready to start coding.

This slide shows just one version of the pseudocode because this is a really short, simple example. As you work on more complex programs, you may find you write and re-write your pseudocode as you work on the problem. It's okay to start general and then add more detail as you understand it. It's always easier to change your pseudocode than to change actual code, so take the time to work on your plan while it's still easy to change.



Planning Requirements

1. Describe the Goal
2. Identify rules and requirements
3. Identify variables and behaviors
4. Plan the overall flow
5. Add more detail to the steps
6. Try coding it
7. Improve your plan
8. Improve your code
9. Test your code

Goal
Rules
Data
Flow
Code
Test Cases
Feedback

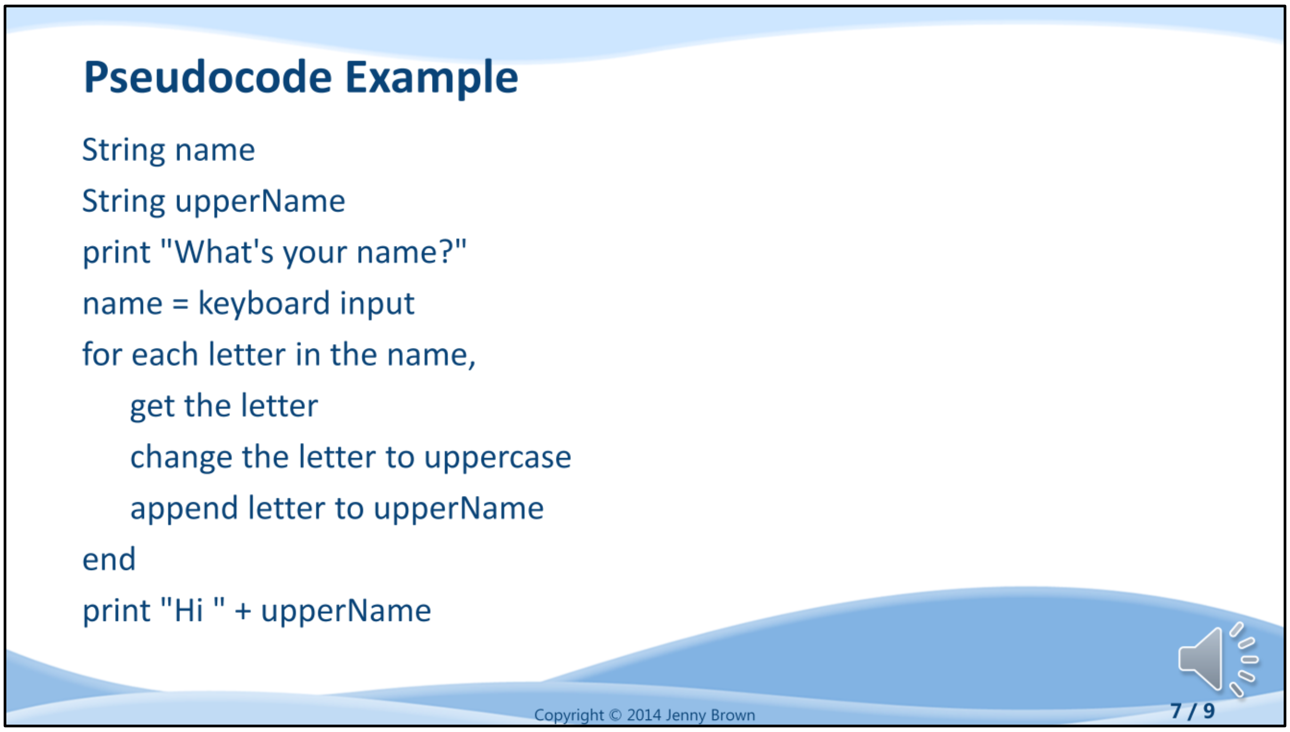
So how can you reasonably go about planning your software? You need to start somewhere. When you're working on class assignments, the requirements will usually be given to you, already written out. And hopefully they're clear!

It's not as simple when you're the one responsible for interviewing the customer and determining what they need. You'll start out by identifying the main goal of the program. This should be something you can describe in one or two sentences. It should talk about how the program serves the person who will use it.

Then, unless someone else has already given it to you, work on writing down all of the rules about how the program should behave. This can include the wording of messages on screen, the kinds of questions it asks and what it does with the answers, any math calculations it runs, and a description of its intended behavior. This may be several paragraphs of English, whatever it takes to understand the planned software. While you should try to make it reasonably complete, understand that there will always be some uncertainty in it. That's why you start by identifying the main goal; it helps you make good choices when other things are ambiguous. This is also a good time to identify the kinds of things you'll eventually type in to test your program and make sure it works right. These test cases are very helpful later on. At that point you have a reasonably good description of the software's purpose.

Next you need to start thinking about the technical details. This includes variables, comparisons, if statements, and the overall structure of your code. Here's where you start to bridge between an idea, in English, over to a code structure. You may find a flow chart to be a useful tool for this, as is pseudocode. In any case, you start to create a technical plan that bridges from English ideas to something that almost resembles code. Then you start coding it, and see where you get stuck or have questions. Those are places where you may need to go back to your plan, ask more questions, try to understand the details better, and then come back to your code and try again.

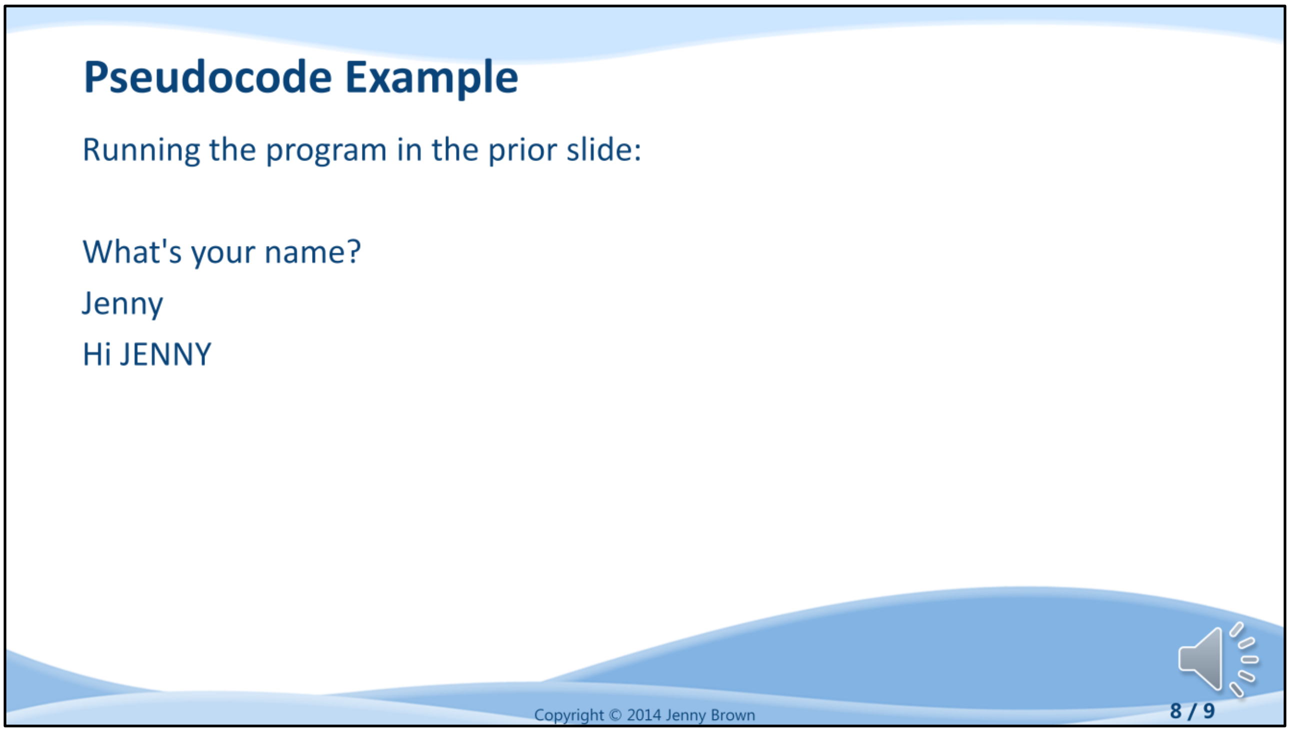
Once you have your code generally working, then you figure out if there are any special rules or weird circumstances it needs to handle, that you haven't already solved. You fix those. You test your code against all the test cases you wrote down earlier, plus any new ones you've thought of. Fix anything that's wrong, and then show your result to the person who will be using it. They may have further suggestions, which result in you adjusting your plans, and you launch right back into another iteration of improving your code some more, testing it some more, and sharing it again.



Pseudocode Example

```
String name
String upperName
print "What's your name?"
name = keyboard input
for each letter in the name,
    get the letter
    change the letter to uppercase
    append letter to upperName
end
print "Hi " + upperName
```

Pseudocode can also be useful when you're planning something that you don't know how to code yet. Read through this example, and see if you can figure out what it does. Notice how it's easier to think about the steps in English that is almost, but not quite, code. Go ahead and pause the video to read it. The next slide will show the answer.

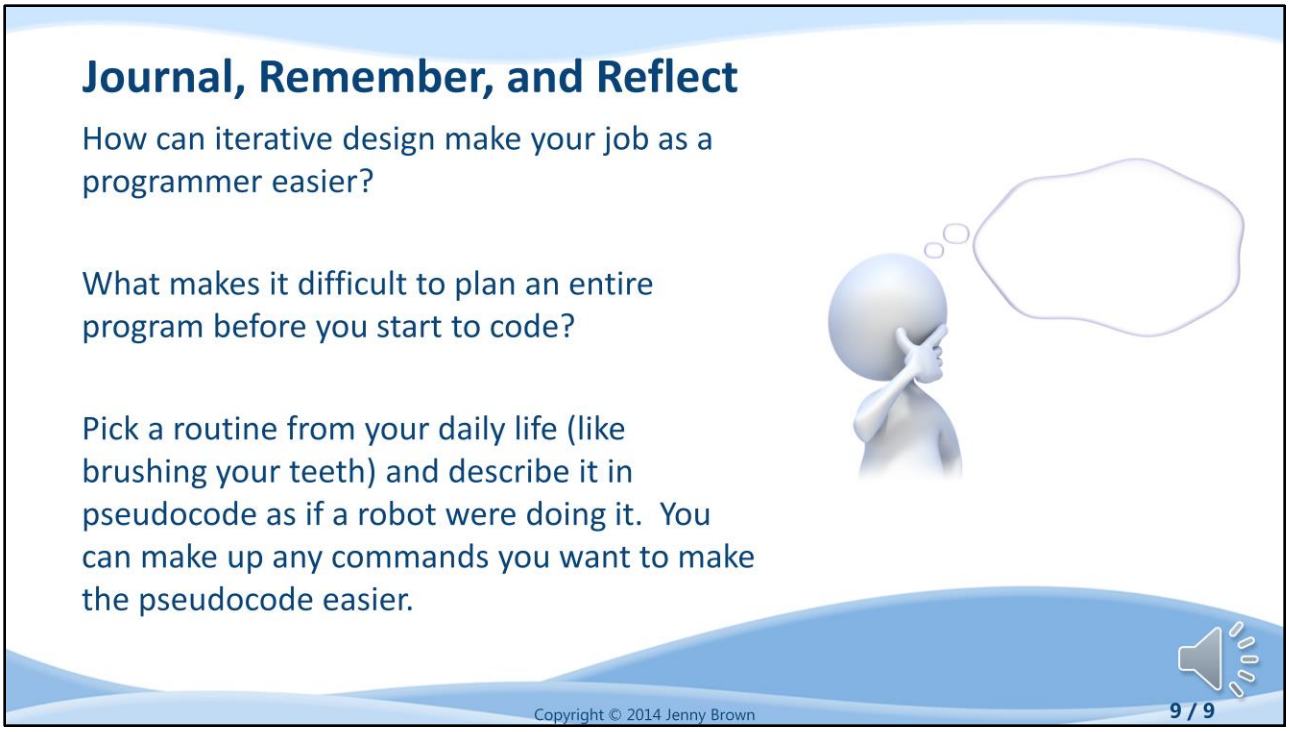


Pseudocode Example

Running the program in the prior slide:

```
What's your name?
Jenny
Hi JENNY
```

The pseudocode in the prior slide describes changing a name to all uppercase. It starts with the letter on the left, J in this case, and converts each letter to uppercase. It saves the result in a separate variable called upperName. Then it prints a greeting using the changed name.



Journal, Remember, and Reflect

How can iterative design make your job as a programmer easier?

What makes it difficult to plan an entire program before you start to code?

Pick a routine from your daily life (like brushing your teeth) and describe it in pseudocode as if a robot were doing it. You can make up any commands you want to make the pseudocode easier.

Journal time.

How can iterative design make your job as a programmer easier?

What makes it difficult to plan an entire program before you start to code?

Pick a routine from your daily life (like brushing your teeth) and describe it in pseudocode as if a robot were doing it. You can make up any commands you want to make the pseudocode easier.