1. Design patterns: <https://en.wikipedia.org/wiki/Design_Patterns#Patterns_by_type>
   1. More like a toolkit and each of the tools solves a very specific problem
   2. It’s better to understand them with comparison to naive approaches/designs
   3. Initial 23 classic design patterns, but now there are tons of them: <https://github.com/iluwatar/java-design-patterns>
2. The ones that we already talked about:
   1. Iterator design pattern
   2. Recall how we apply iterator to traverse the engineering department in midterm
   3. We can also create an iterator on the conservatory in the Bird lab
   4. We can even build iterators to traverse files in a file system, or machines in a data center, etc.
3. Singleton pattern: ensure only one instance of a certain class.
   1. Imagine we are building a website which has several webpages. We want to have a counter to accumulate the clicks on all the pages.
      1. Say each page is an object of Page class
      2. Say we have a counter class
      3. How to guarantee all the pages use the same counter object
   2. The keyword *static* – a static method or member variable belongs to the class
   3. Pre-create an instance and mark the constructor as private
   4. Ref: <https://en.wikipedia.org/wiki/Singleton_pattern#Java_implementation[8>]
4. Builder pattern
   1. Use case: construct a class that has many member variables, of which many are optional.
   2. Naive solution:
      1. an army of overloading constructors
      2. Or, create an object with default member variables and provide setters for all of them – the problem is we don’t want to have setters for some of them
   3. Builder pattern
      1. Module 3.6 on Canvas
5. Command design pattern
   1. Say you want to build a video game. The protagonist can move, shoot and jump.
      1. There is a keyboard module that issues commands
      2. And there is a game logic module that receives the keyboard commands, and translates them to protagonist action
   2. One design: the keyboard module sends string/enum commands to the game logic module, which has a switch statement
   3. Command design pattern: one command class for each command.
      1. Followup Q1: what if we want to add a new command called jump shoot?
      2. Followup Q2: what if both jump and shoot commands have 1000 LoC in its exec()?
   4. OOD:
      1. Encapsulation: put everything related to a command to one place.
      2. Can use inheritance to share some common code among commands.
      3. Can put a sequence of commands into a collection
   5. Fundamental idea: let java polymorphism/dynamic dispatch do the branching instead of our hand-rolled switch cases
   6. Ref: <https://course.ccs.neu.edu/cs5004/lecturecommands.html#%28part._.The_.Command_.Design_.Pattern%29>
6. Adapter pattern:
   1. Use case: you have an existing class and a function that takes some existing interface as an argument. How to pass the existing class into the function when the class doesn’t implement the required interface?
      1. Example: you have a forward iterator, but the function you want to call requires a reverse iterator
   2. One more example: <https://en.wikipedia.org/wiki/Adapter_pattern#Java>
7. Factory method pattern:
   1. Lab 2 - role playing game
   2. Basic idea: creates object without specifying the exact class
   3. Ref: <https://en.wikipedia.org/wiki/Factory_method_pattern>
8. Decorator pattern:
   1. <https://en.wikipedia.org/wiki/Decorator_pattern#First_example_(window/scrolling_scenario)>
   2. Example: Basic window, VerticallyScrollableWindow, HorizontallyScrollableWindow, BorderedWindow
   3. Now if I want to create VerticallyScrollableBorderedWindow, HorizontallyScrollableBorderedWindow, etc. I have to create many different child classes
   4. Composition-over-inheritance principle:
      1. <https://en.wikipedia.org/wiki/Composition_over_inheritance>
      2. *“It is more natural to build business-domain classes out of various components than trying to find commonality between them and creating a family tree.”*