Slide 1

Administrivia

- Homework 2 design due today. Code due next Tuesday.
- Homework 1 grades coming soon. Probably full credit for everyone.

Slide 2

Homework 2 — Design (Again)

- Interfaces `YourBlock`, `YourEntity`: In project API, referred to as "general block type" and "general entity type". You will use these as replacements for `BasicBlock` and `BasicEntity`, and everywhere else you use one of the framework's generic classes.
- Player and game setup classes. Copy code from `BasicPlayer` and `BasicGameSetup` and edit (change `package` line, block and entity types). May want to change game setup more during code phase. Also edit your main class from the first assignment.

  Don't worry about player for now — you will start writing your own in the next assignment.

Slide 3

Homework 2 — Design Continued

- Block class(es). These are blocks that make the playing field for your game. Should have one class for each kind of block (floor, walls, ladders, anything that doesn't move). Try to define as many as you can. Copy code from `BasicBlock`.
- Screen class (class implementing `Screen` interface). This is the most work in this assignment. Eclipse can make stub methods for you. Copy and paste comments from API.

Slide 4

How to Approach Defining a Class

- What methods do I need? If implementing an interface, you at least need the methods in the interface. May want additional methods. If making a subclass, remember you automatically inherit all methods from superclass. Can override them and/or provide additional methods.
- What variables do I need to implement the needed methods? e.g., if defining a `Rectangle` class that has a `getArea` method, probably need either area or width and height.

Slide 5

Homework 2 — Code

- Go back through copied code and look for things you know you want to change. E.g., you should be able to figure out how to draw something okay-for-now for your game's blocks by changing `getImage`.
- For your screen class, decide what variables you need to implement the methods in `Screen`. (You'll probably want a two-dimensional grid (array) of blocks (your block type) and a list of entities (your entity type).)

Slide 6

"Good Style" Tips

- Make methods public if needed by code that uses your class, private otherwise.
- Make variables private unless there's a good reason not to — prevents unwanted/inconsistent access.
- Try to choose good names for variables — ones that make it clear what they're for.
- Use named constants (static final variables) rather than hard-coded values.
  E.g., `private static final screenWidth = 20;`
  Also remember that you can get the size of an array from its `length` field (variable).
- Follow Java conventions — class names start with a capital latter, method and variable names with lower case.

Slide 7

"Good Style" Tips, Continued

- Make sure your code is indented in a consistent and readable way. (Eclipse tip: control-I cleans up indenting on whatever text is selected. So, control-A followed by control-I may be a good idea after editing that messes up indentation.)
- Remember that you're writing for three audiences: the compiler, a human reading your code, and a human reading the generated HTML documentation.

Slide 8

Commenting Code

- "Documentation" (javadoc) comments generally describe how someone would use your class. For examples, look at Java API, game framework, examples from class. Note that some comments for game framework describe how to "use" the class in the sense of extending or modifying it. Eventually these should disappear from your copies/versions.

  I've said that for code you turn in you should provide at least a brief comment on every class and every method (and also any public variables), with the possible exception of things that are "totally obvious". A little subjective, so getting it right will be an iterative process (you try something, I suggest changes).
- Internal comments can be helpful within the code, to clarify any tricky bits, or to improve readability. Eclipse convention seems to be to have `// TODO` comments to indicate things that will need attention at some point. Probably useful.

Slide 9

String Class — Example of Using a Class

- In C, "strings" are just arrays of characters, terminated by null character. In Java, there's a library class, `String`.
- To see what's available, look at the API . . .

Slide 10

String Class, Continued

- In general, no operator overloading in Java, with one exception — "+" for strings. Non-string objects converted using (their) `toString` method. Primitives converted in the "obvious" way.
- To compare two strings, "==" is rarely what you want. Instead, use `equals`.
- Strings are "immutable" — once created, can't be changed. (Why? allows them to be safely shared.) Methods you would think might change the value return a new string.
- Use `StringBuffer` if you need something you can change, or for efficiency.
- Let's do some examples . . .

Slide 11

Sidebar — Immutable Objects

- `String` is an example of a class that's "immutable" — once created, objects can't be changed. If you look at the API for `String`, you notice that methods that "change" the string actually return a new one.
- This sounds inconvenient, right? What advantages might it have? (Hint: What did we say a few classes ago about what really happens when you "pass an object to a method"?)

Slide 12

Minute Essay

- None — quiz.

Slide 13

Minute Essay Answer

- FIX THIS