Slide 1

Administrivia

- *Please do not* reboot the machines in HAS 340! If a previous user has left a machine in the "locked by screensaver" state, you can bail out by pressing control-alt-backspace to restart X (the graphical subsystem) without disturbing background processes.
- Are your prox cards giving you access to the labs? Supposedly all known problems have been resolved.
- Reminders:

  Homework 1 design due today at 11:59pm. ("Turn in" by sending me mail saying your game is ready to grade. Putting "csci 1321" or similar in the subject line helps me not misplace your message!)

  Homework 1 code due Thursday. Open lab today and tomorrow. After today's lecture you should know enough.
- Details for all homeworks on Web now; Homework 2 due next week.

Slide 2

UML Class Diagrams

- "Unified Modeling Language" — formal graphic representation of software analysis and design.

  Many types of diagrams, some of which you'll probably encounter in other courses. Tools exist for drawing them, but worth noting that they were designed to be whiteboard-friendly.
- We will mainly use class diagrams:
  - Box representing a class has name, attributes, operations.
  - Subclass points to its superclass (represents the path to follow to figure out inheritance).

Slide 3

Inheritance (Short Version)

- Given a class, it can be useful to define specialized versions — "subclasses".
- A subclass inherits attributes and operations from its superclass (which can in turn have a superclass . . .).
- Subclasses also form "subtypes" — e.g., if `CheckingAccount` is a subclass of `Account`, can use a `CheckingAccount` anywhere we need a `Account`.

Slide 4

Polymorphism (Short Version)

- "Many shapes" — something that works with many types.
- E.g., a function that works on `Accounts` should work on `CheckingAccounts`, `SavingsAccounts`, . . .

Slide 5

Intermezzo — Immutable Objects

- Some classes are "immutable" — once created, objects can't be changed. Example — `String` — if you look at the API, you notice that methods that "change" the string actually return a new one.
- This sounds inconvenient, right? What advantages might it have?

Slide 6

Inheritance and Code Reuse

- If class `Account` defines

  ```
  private double balance;
  public double getBalance();
  ```

  then if `SavingsAccount` is a subclass of `Account`, `SavingsAccount` also has variable `balance` and method `getBalance`.
- This can be a good way to reduce code duplication.
- If it's not what you want, subclasses can "override" methods (or variables — but this is not usually a good idea).
- Or a superclass can leave methods unimplemented; subclasses must then define — for `Account`, `addInterest` could be abstract.

Slide 7

Inheritance and Subtypes

- In the "shapes" example, class `Account` defines a type, and `SavingsAccount` and `CheckingAccount` are subtypes. Anywhere we need a `Account`, we can use a `SavingsAccount` — e.g.,
  `Account s = new SavingsAccount();`
  (but not `SavingsAccount s = new Account()`;)
- So we could have an array of `Account`s, whose elements could be `SavingsAccount`s or `CheckingAccount`s. (More about arrays soon.)

Slide 8

Multiple Inheritance Versus Interfaces

- What if you want a class to inherit from multiple classes? C++ allows this ("multiple inheritance"). To avoid possible confusion/ambiguity, Java doesn't.
- Instead, define "interfaces" — classes in which *all* methods are abstract.
- In `Account` example, we could define a `HasPersonName` interface with method `getPersonName`. Not obviously useful — unless there's another kind of object that could have a person's name but shouldn't be a subclass of `Account`. (A prospective customer?)
- A class can "implement" as many interfaces as you like.

Slide 9

Interfaces and Types

- Interfaces also define types. So if `Account` implements interface `HasPersonName`, we can use a `Account` anywhere a `HasPersonName` is required.

```
HasPersonName o = new Account();
```

- This is "inclusion polymorphism" — and is what will allow your project code to plug neatly into Dr. Lewis's framework. (The framework is written in terms of interfaces such as `Block` and `Screen`; your classes will implement those interfaces.)

Slide 10

A Few Words About Generics

- Java library has many useful "container classes" (for vectors, sets, linked lists, etc.) that can hold any kind of object — which is useful, but also sometimes inconvenient (e.g., no way to say "I want a vector of `Account`s only").
- A solution — Java "generics", new with 1.5/5.0.
- Syntax uses angle brackets, e.g., a `Vector` that can hold only `Account`s:

```
Vector<Account> v = new Vector<Account>();
```

- Most of Dr. Lewis's game framework uses this feature — e.g., in Homework 1 you need not a `MainFrame` object but a

```
MainFrame<BasicBlock, BasicEntity>
```

object.

Slide 11

Minute Essay

- None — sign in.