Slide 1

Administrivia

- Reminder: Homework 1 design due today. Code due next Thursday. Next two classes should cover enough material for you to do what's needed.
- A request: You will turn in almost all work for this course by e-mail. Please do include the name or number of the course in the subject line of your message, plus something about which assignment it is, to help me get it into the correct folder for grading.

Slide 2

More Administrivia

- Two options for transferring files between machines:
  - Simple, not particularly smart: Just copy `.java` files between machines. Several options.
  - More complicated to get started, more professional: Use CVS (versioning software, Eclipse has built-in support).
- Some instructions in project description . . .

Slide 3

Java Basics Example, Continued

- Let's try to quickly write some more of the methods of that `Account` class . . . .

Slide 4

Arrays, Briefly

- Syntax is like C, except for explicit `new`:
  - `int[] x = new int[10];` creates 10 integers.
  - `String[] args = new String[20];` creates 20 *references* to strings.
- Arrays are "first-class" objects, with `length` variable.
- Java checks for out-of-bounds array references.

Slide 5

Miscellaneous Other Stuff

- No operator overloading (except "+" for `String` class).
- On reference variables, `=` and `==` operate on references, not objects. (So, you may instead want copy constructors or `equals()`.)
- No C-style strings, but a `String` class.
- A little about packages, and Java "generics" (new with 5.0), later.

Slide 6

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 7

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 an `Account`.

Slide 8

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 9

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 (maybe differently for different classes). E.g., for `Account`, if we make `deposit` and `withdraw` abstract, each subclass must provide its own code.

Slide 10

Inheritance and Subtypes

- In the bank-account 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.
- Let's write more code for that example . . .

Slide 11

Minute Essay

- What changes would you make to the bank-account example to allow paying interest on accounts? (Would you add variables and/or methods? To what class(es)?)

Slide 12

Minute Essay Answer

- There should probably be a method `addInterest`, and I think it should be in `Account`, at least as an abstract method. Beyond that — there are some decisions to make:
- Do all accounts pay interest the same way, or is it different for different types? if it's the same for all, code can go in `Account`, otherwise it needs to go in subclasses.
- Do all accounts pay interest at the same rate, or is it different for different types, or even for different individual accounts? if it's different for different account, it probably should be an instance variable.
- Is the rate the same every time, or does it change (e.g., varies from month to month)? If it changes, it probably should be a parameter to `addInterest` rather than being an instance variable.