Slide 1

Administrivia

- (None?)

Slide 2

Packages and Importing

- Packages are simply a way of grouping related code and providing restricted scope for class names. Package names are (somewhat) hierarchical, with levels separated by dots — look at Java library API for examples.
- For classes in `java.lang` and current package reference using the class name only (e.g., `System`). For other classes, can use full name (e.g., `java.util.Vector`), or use `import`. (`import` looks like `#include`, but works differently.)

Slide 3

Packages, Continued

- You can define your own packages. Convention is to start with your e-mail/Web address, in reverse order (e.g., Dr. Lewis's framework is `edu.trinity.cs.gamecore`). (For simple throwaway programs you might use simpler names.)
- Packages and filesystem hierarchy are related — for an example, create a package in BlueJ/Eclipse and then use another tool to look at the resulting directories and files.

Slide 4

Another Example

- Let's start sketching another example — `Account` class representing bank accounts.
- What variables seem useful? what methods?

Slide 5

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.
  - Different kinds of arrows showing relationships among classes and interfaces.

Slide 6

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 `PersonalAccount` is a subclass of `Account`, can use a `PersonalAccount` anywhere we need an `Account`.

Slide 7

Polymorphism (Short Version)

- "Many shapes" — something that works with many types.
- E.g., a function that works on `Account`s should work on `PersonalAccount`s, `BusinessAccount`s, ...

Slide 8

Inheritance and Code Reuse

- If class `Account` defines

  ```
  private String accountID;
  public void deposit();
  ```

  then if `BusinessAccount` is a subclass of `Account`, `BusinessAccount` also has variable `accountID` and method `deposit`.
- 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 `withdraw` abstract, each subclass must provide its own code.

Slide 9

Inheritance and Subtypes

- In the bank-account example, class `Account` defines a type, and `BusinessAccount` and `PersonalAccount` are subtypes. Anywhere we need a `Account`, we can use a `BusinessAccount` — e.g.,

  ```
  Account s = new BusinessAccount();
  ```

  (but not `BusinessAccount s = new Account();`)
- So we could have an array of `Account`s, whose elements could be `BusinessAccount`s or `PersonalAccount`s.
- Let's write more code for that example . . .

Slide 10

Inheritance Versus Interfaces

- What if you don't need/want the superclass to provide any code? you just want it to define a "contract" that all subclasses must meet (i.e., a list of methods they must provide?) then you want a Java *interface*.
- In the `Account` example, we could define a `PaysInterest` interface with method `addMonthlyInterest`. This would let us decide for each type of account whether it should pay interest — e.g., `BusinessAccount` and some subclasses of `PersonalAccount`.
- A class can "implement" as many interfaces as you like.

  (This helps if you want a class to inherit from multiple classes — Java, unlike some languages (e.g., C++), doesn't allow that because of possible confusion/ambiguity, but you can fake it by implementing multiple interfaces.)

Slide 11

Interfaces and Types

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

```
PaysInterest p = new BusinessAccount();
```

- 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 12

Minute Essay

- In class we started writing a `PaysInterest` interface with a method `payMonthlyInterest`. What changes would you need to make to `BusinessAccount` to include this method? do you need more variables in the class, or does the method need parameters, or what? and what would the code look like?

Slide 13

Minute Essay Answer

- There are actually several things you might want to think about first . . .
- Is the rate the same every time you pay interest (monthly?), or does it change from month to month? (If it stays the same, maybe it should be a variable within the object; if it changes, maybe it should be a parameter to the method.)
- Is the rate the same for all accounts, or different for different accounts? if the former, it could be a class (static) variable.