Slide 1

Administrivia

- Tentative dates for quizzes on Web. First one next Thursday. I will put some practice problems on the Web.
- Homework 1 design due next Thursday. More next time, but you should know enough at this point to start if you like.

Slide 2

Parameter Passing in Java

- As in C, all parameters in Java are passed “by value”. (Some languages provide other options, e.g., passing “by reference”.)
- C has pointers, which can point to any data type, and this allows you fake passing parameters by reference. Not possible in Java — Java has references, which can only point to objects.
- *However*, when you pass an object reference by value, both caller and callee have references to the same object, so in some ways you appear to be passing the object by reference.

Slide 3

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.)
- Tip: When writing code with Eclipse, if it can't find a particular class because it needs an `import`, select the reference to the class and press shift-control-M, and it will try to generate an appropriate `import`. Shift-control-M "organizes" `import`s (removes any not needed).

Slide 4

Packages, Continued

- You can define your own packages. Convention is to use your e-mail/Web address, in reverse order (e.g., Dr. Lewis's framework is `edu.trinity.cs.gamecore`). For your game, I'm recommending `edu.trinity.cs.yourusername.yourgame` (`yourgame` is something descriptive). Call the main class something with `Main` in its name.
- Packages and filesystem hierarchy are related — for an example, create a package in Eclipse and then use another tool to look at the resulting directories and files.

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 `CheckingAccount` is a subclass of `Account`, can use a `CheckingAccount` 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 `CheckingAccount`s, `SavingsAccount`s, . . .

Slide 8

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 9

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 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 `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.

  (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 `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 12

Minute Essay

- Last time I asked you to try writing a method to compute and add interest, assuming the interest rate was stored in an instance variable. But that might not be the best way to provide for paying interest on accounts, and anyway now we have both an `Account` class and subclasses. What would you add to these classes (variables, methods, etc.) to allow for paying interest? (Your answer might be “it depends” — if so, on what?)

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 accounts, 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.