Slide 1

Administrivia

- No class Thursday; I plan to be at a conference. If you need help with homework, okay to ask Dr. Lewis. Open lab Thursday 3:30pm–5:30pm.
- Comments on Homework 1 coming soon by e-mail. Code due a week from today. Notice changed recommendations about package names. (Okay to keep what you have this time, but change for Homework 2.) More about packages today.
- Remaining homeworks online, for those who want to know more about what's ahead.
- (Review minute essay from last time.)

Slide 2

Inheritance (Short Version, Recap)

- 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 3

Polymorphism (Short Version, Recap)

- "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 4

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 5

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. (More about arrays soon.)

Slide 6

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 7

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 8

Packages and Importing

- Library classes grouped into "packages" — e.g., `java.util`, `java.net`.
- For classes in `java.lang` and "default package", reference using their names only. For other classes, can use full name or `import`. (`import` looks like `#include`, but works differently.)
- 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.
- 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`.

Slide 9

"Generics" in Java

- Java library includes classes for collections of things (`ArrayList`, e.g. — like an expandable array). Originally, could put any kind of `Object` in one of these. Nice, except that then there's no way to know anything about types of objects inside except by using reflection (*much* later, if at all) or `instanceof` operator. Must also use explicit casts to do much with objects retrieved from collection.
- So Java 1.5 (a.k.a 5.0) introduced "generics" — Java's answer to C++ template classes, though not exactly the same. Idea is to allow you to specialize a collection — so, a `ArrayList` of `Integer` objects only, or a `ArrayList` of `Account` objects only, etc., etc. Syntax uses angle brackets, e.g., a `ArrayList` that can hold only `Account`s:

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

- Also look at API for `MainFrame` in the game framework . . .

Slide 10

Minute Essay

- What problems did you have doing the design phase of Homework 1?