The Big Picture

- a subclass object can be used any place a superclass type is expected
  - e.g.

```
BankAccount acct =
    new InvestmentAccount(2,"Arthur",500,.015);
acct.withdraw(100);
```

  - compiler checks the declared type of the object
    - `acct.withdraw(100)` is legal because `BankAccount` has a withdraw method that takes a `double`
    - works at runtime because a subclass has all of the elements (instance variables and methods) that its superclass does – so `InvestmentAccount` also has a `withdraw` method that takes a `double`
- at runtime, the version called belongs to the actual type of the object (*polymorphism*)
  - `acct.withdraw(100)` will result in a fee if the number of withdrawals exceeds the limit because `acct` actually refers to an `InvestmentAccount`

Abstract Classes

Establishing is-a relationships between types is important for flexible, reusable code – but the specific semantics of `extends` creates some problems.

All animals eat, sleep, and make noise, but how they make noise varies - cows moo, ducks quack, horses neigh, etc. If you were designing a collection of classes for barnyard animals, what would be the best choice?

- make Animal a class, with Cow, Duck, and Horse extending Animal
- ★ make Animal an abstract class, with Cow, Duck, and Horse extending Animal
- just make Cow, Duck, and Horse classes (no Animal)
- none of these are appropriate choices

observations –

nothing is just an animal – it is always some kind of animal (cow, duck, horse, ...)

all animals have a common ability (making noise) but there is not a common implementation (moo, quack, neigh, ...)

`Animal` as a concrete class is not appropriate because there is no such thing as an animal that isn't also some kind of animal – shouldn't be able to create `Animal` objects

a body for `makeNoise` in `Animal` is not appropriate because there isn't a way to make noise shared by all animals

`Animal` as an abstract class allows for reuse of code common to all kinds of animals as well as using `Animal` as a type (allows coding to the interface)

These two things are very powerful.

Consider the book's example of a shape-drawing program –

- can *code to the interface* – can have a single collection of `Shapes`, with a loop to go through and paint them all
  - avoids repeated code resulting from needing a separate collection for each kind of shape
  - makes it possible to write new `Shape` classes and use them with the existing shape-drawing program without changing the existing code
- can *encapsulate what varies* – each `Shape` subclass can have its own `paint` method, capturing the differences between kinds of shapes
  - separates the rest of the program from the details of individual shapes, limiting the impact of change – updating how one shape is painted only affects that one shape's class

Abstract Classes

- *abstract classes* handle this situation
- syntax
  - `public abstract class` *`ClassName`* `{ … }`
    - `abstract` means that it is not possible to create instances of `ClassName` – "nothing is just a `ClassName`"
    - `abstract` is required if there is at least one abstract method
  - `public abstract` *`returntype methodname`* `(` *`paramlist`* `);`
    - `abstract` means that no body is supplied – "no common way of doing the operation"
      - must be overridden in a subclass or else the subclass is also abstract
    - note the difference between `;` for an abstract method and `{}` for a (not abstract) method with an empty body
  - abstract classes must still have one or more constructors
    - can't use directly to create a standalone object, but subclass constructors still need to be able to create the core of the onion
    - can be `protected` because only subclasses will use them
  - can also have instance variables and methods with bodies

Deciding on Abstract Classes and Methods

Language cues –

- the unifying concept isn't talked about as its own thing → abstract class
  - compare “A bank account …, a checking account …, a savings account ...” (bank account is its own thing – not abstract) and “All kinds of bank accounts …; a checking account …, a savings account …” (bank account is a category, not its own thing – abstract)
- there is a common operation, but how it works depends on the kind of thing → abstract method
  - e.g. “All tickets have a price. The price of a walkup ticket is …, the price of an advance ticket is …, “ (abstract method, unless the only difference is the amount)
  - e.g. “The price of a ticket depends on the type of ticket and potentially other properties of the ticket.” (abstract method)