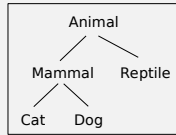


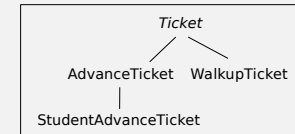
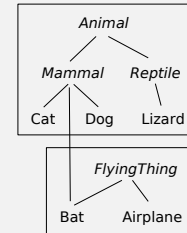
Type Hierarchies

- different kinds of things are not always completely unrelated
 - e.g. apples and fruit – apples are a kind of fruit, though there is fruit that's not apples
 - e.g. savings accounts and checking accounts are both kinds of bank accounts
- type hierarchies capture “is-a” relationships between types
 - “is-a” in the sense of adds state and behavior or modifies existing behavior – cannot remove state or behavior



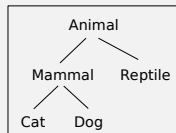
Type Hierarchies in Java

- not all types in a type hierarchy are suitable for instantiating objects
 - use an *abstract class* when the more specialized types have state and/or behaviors in common
 - use an *interface* when the type groups otherwise dissimilar things



Polymorphism

- polymorphism* allows instances of a type lower in the hierarchy to be used when a value of a type higher in the hierarchy is expected
 - allows avoiding repeated code when the same operation is applied to similar kinds of objects
 - essential for writing reusable, flexible code



Inheritance

- inheritance defines an “is-a” relationship between classes

```
public class Apple extends Fruit {  
    ...  
}
```

 - an apple is a (kind of) fruit
- subclasses inherit everything – instance variables and methods – *except* constructors
 - even private things, though they cannot be accessed directly in the subclass
 - like a safe deposit box at the bank – you own the box and things can be stored in it, but you have to ask someone else to access it for you
 - new access modifier: `protected` allows only* the class and its subclasses to access
 - * `protected` also allows classes in the same package to access, but if you only want same-package access, use `package-private` (no modifier keyword) instead

Inheritance

Subclasses –

- *can* add new elements (instance variables and methods)
 - a new method has a different header (name and/or number/type of parameters)
- *can* redefine (override) or extend methods
 - same header, new body
 - to extend, also invoke superclass version
- *must* define one or more constructors (in most cases)
 - constructor should first call superclass constructor, then initialize only the instance variables for its own class
 - the superclass constructor is responsible for initializing its own instance variables
- *cannot* redefine instance variables
- *cannot* remove instance variables or methods already defined

40

Inheritance

- an object is like an onion, with each class in the inheritance hierarchy describing a layer
- top-level class is at the core



- this refers to the current layer of the onion
- super refers to the next layer in

CPSC 329: Software Development • Fall 2026

41

Polymorphism

- 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);
```
 - the compiler checks the declared type of the object
 - `acct.withdraw(100)` is legal because `acct` is declared as a `BankAccount` and `BankAccount` has a `withdraw` method that takes a `double`
 - this will be legal 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 run 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`

42

Polymorphism

These two things are very powerful.
Consider 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, reflecting 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

CPSC 329: Software Development • Fall 2026

43

Abstract Classes

- 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 way to write a body that makes sense for all”
 - must be overridden in a subclass or else the subclass is also abstract
 - note the difference between `;` for an abstract method (with no body) 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 object
 - can (should) be protected because only subclasses will use them
 - can also have instance variables and methods with bodies

Interfaces – Defining

- syntax
 - `public interface InterfaceName { ... }`
 - `public returntype methodname ( paramlist );`
 - like an abstract method in that no body is supplied, but abstract keyword is not needed
 - cannot have instance variables, constructors, or (most) methods with bodies
 - since Java 8, default and static methods are allowed to have bodies
 - default method body is inherited by classes implementing the interface if they don't provide their own version
 - since Java 9, private methods are allowed to have bodies (helpers for the default and static methods)
- semantics
 - an interface defines a type
 - the type can be used anywhere a type is needed e.g. in variable and parameter declarations or as the base type of an array
 - it is not possible to create instances of an interface type
 - no `new ...` for an interface

Interfaces – Implementing

- classes *implement* interfaces
- syntax
 - `public class ClassName implements Interface1, Interface2, ... { ... }`
 - a class can implement any number of interfaces (it can also extend another class)
 - the class must provide a body for every method in the interface(s) or else it must be declared abstract
 - the class must provide a body to resolve conflicts for any default method coming from two or more interfaces or else it's a compiler error
- semantics
 - an object of a type implementing an interface can be used anywhere the interface type is expected
 - e.g. `Interface1 obj = new ClassName();`