COMP 202 – Weeks 10 & 11

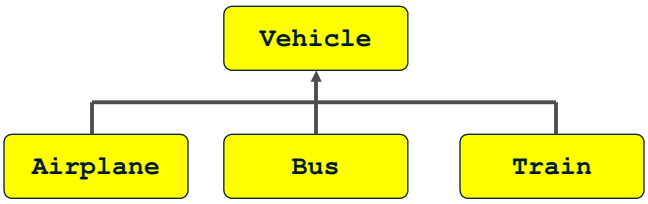
- Another fundamental object-oriented technique is called inheritance. It enhances software design and promotes reuse.
- This week we focus on:
  - deriving new classes
  - the `protected` modifier
  - creating class hierarchies
  - abstract classes
  - polymorphism via inheritance

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class

Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Vehicle
{
   // class contents
}

class Airplane extends Vehicle
{
   // class contents
}
```

Controlling Inheritance

- Visibility modifiers determine which class members get inherited and which do not
- Variables and methods declared with `public` visibility are inherited, and those with `private` visibility are not
- But `public` variables violate our goal of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The `protected` Modifier

- The `protected` visibility modifier allows a member of a base class to be inherited into the child
- But `protected` visibility provides more encapsulation than `public` does
- However, `protected` visibility is not as tightly encapsulated as `private` visibility
- The details of each modifier are given in Appendix F

The `super` Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and is often used to invoke the parent's constructor
- See Vehicle.java
- See Bus.java
- See ABusIsAVehicle.java

Single vs. Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- In most cases, the use of interfaces gives us the best aspects of multiple inheritance without the overhead

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- That is, a child can redefine a method that it inherits from its parent
- The new method must have the same signature as the parent's method, but can have different code in the body
- The type of the object executing the method determines which version of the method is invoked

Overriding Methods

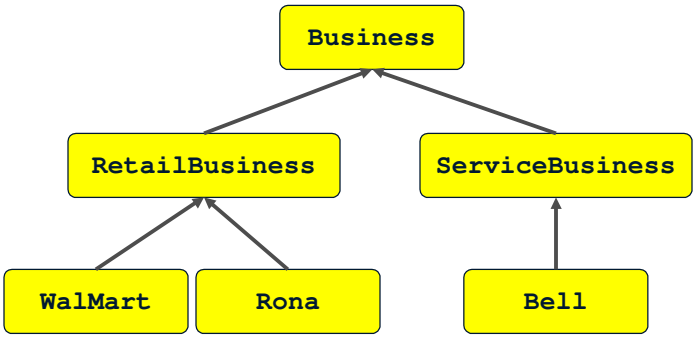
- See PastVsPresent.java
- See Past.java
- See Present.java
- Note that a parent method can be explicitly invoked using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data (called *shadowing variables*), there is generally no need for it

Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Class Hierarchies

- A child class of one parent can be the parent of another child, forming *class hierarchies*



Class Hierarchies

- Two children of the same parent are called *siblings*
- Good class design puts all common features as high in the hierarchy as is reasonable
- An inherited member is continually passed down the line
- Class hierarchies often have to be extended and modified to keep up with changing needs
- There is no single class hierarchy that is appropriate for all situations

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- One interface can be used as the parent of another
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the parent and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

The `Object` Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- The `Object` class is therefore the ultimate root of all class hierarchies

The `Object` Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we have defined `toString`, we have actually been overriding it
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class and a hash value

The `Object` Class

- That's why the `println` method can call `toString` for any object that is passed to it – all objects are guaranteed to have a `toString` method via inheritance
- See Student.java
- See GradStudent.java
- See Academia.java
- The equals method of the `Object` class determines if two references are aliases
- You may choose to override `equals` to define equality in some other way

Indirect Access

- An inherited member can be referenced directly by name in the child class, as if it were declared in the child class
- But even if a method or variable is not inherited by a child, it can still be accessed indirectly through parent methods
- See FoodItem.java
- See EarthWorm.java
- See FoodAnalysis.java

Abstract Classes

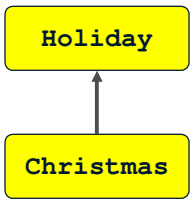
- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract
- An abstract class often contains abstract methods (like an interface does), though it doesn't have to

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as final (because it must be overridden) or static (because it has no definition yet)
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is too general to instantiate

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Holiday` class is used to derive a child class called `Christmas`, then a `Holiday` reference could actually be used to point to a `Christmas` object




```
Holiday day;
day = new Christmas();
```


References and Inheritance

- Assigning a predecessor object to an ancestor reference is considered to be a widening conversion, and can be performed by simple assignment
- Assigning an ancestor object to a predecessor reference can also be done, but it is considered to be a narrowing conversion and must be done with a cast
- The widening conversion is the most useful

Polymorphism via Inheritance

- Earlier, we saw how an interface can be used to create a *polymorphic reference*
- Recall that a polymorphic reference is one which can refer to different types of objects at different times
- Inheritance can also be used as a basis of polymorphism
- An object reference can refer to one object at one time, then it can be changed to refer to another object (related by inheritance) at another time

Polymorphism via Inheritance

- Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrode it
- Now consider the following invocation:

```
day.celebrate();
```

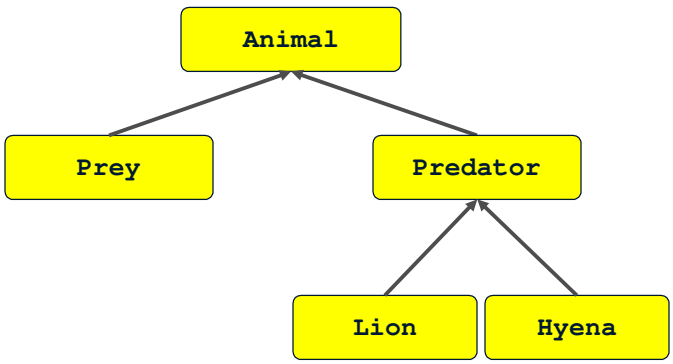
- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version

Polymorphism via Inheritance

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Note that, if an invocation is in a loop, the exact same line of code could execute different methods at different times
- Polymorphic references are therefore resolved at run-time, not during compilation

Polymorphism via Inheritance

- Consider the following class hierarchy:



Polymorphism via Inheritance

- **Now consider the task of feeding all animals**

- **See Animal.java**
- **See Prey.java**
- **See Predator.java**
- **See Lion.java**
- **See Hyena.java**
- **See Fauna.java**
- **See TheBush.java**