Inheritance

Parent and Child Classes
protected
Overriding
super
The Object class
The equals method

CLASS DESIGN: A BRIEF REVIEW

Writing Classes

- Classes define a blueprint of what all objects of that class will look like.
  - Once we define a class, we can instantiate as many objects of that class as we need

- Classes define:
  - Characteristics (state)
    - Instance data
  - Functionality (behaviors)
    - Methods

Summary of Classes

- Contain:
  - private instance variables
  - constructors
  - public getters and setters (with appropriate validity checking)
  - toString method
  - public and private class-specific methods

Practice

- Review the AudioItem class.

INHERITANCE

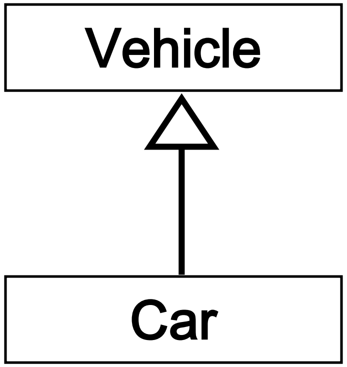
Inheritance

- *Inheritance* allows you to design a new class from an existing class
  - The existing class is called the
    - *parent class*
    - *superclass*
    - *base class*
  - The new class is called the
    - *child class*
    - *Subclass*
    - *derived class*

- Inheritance creates an *is-a* relationship.
  - The child *is a* more specific version of the parent.

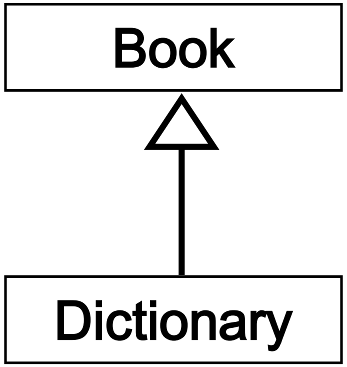
Inheritance

- A car *is-a* specific type of vehicle.



```
public class Car extends Vehicle {
   // class contents
}
```

- A dictionary *is-a* specific type of book.



```
public class Dictionary extends Book
   {
    // class contents
}
```

Inheritance

- The child inherits characteristics of the parent
  - Methods
  - Data
- You can modify the child class by:
  - Adding new methods and variables
  - Modifying inherited methods
- Inheritance is established with the `extends` reserved word

Practice

- Get more specific about what you sell in you store: music and audio books.
- Music also has an artist and genre.
- Audio books also have an author and the number of chapters.

Why Use Inheritance?

- *Software reuse* is a fundamental benefit of inheritance.
  - Reuse everything in the parent class
- Share and reuse common code
- Code is easier to maintain

Programming is like the environment. The more you recycle and reuse, the less junk you have…

What gets inherited?

- Methods
- Instance data variables
  - HOWEVER! You cannot directly access a private instance data variable.
  - It’s there, you just can’t reference it!
  - To access, you need to use the public getters and setters.

The `protected` Modifier

- Variables and methods that are private in the parent class *cannot* be referenced by name in the child class.
- We could make variables public in the parent class… but that would violate encapsulation.

- Instead, we can make variables and methods `protected`.

The `protected` Modifier

- A `protected` variable or method is visible to any class in the same package as the parent class and to a subclass (in any package)
- The `protected` modifier allows a child class to reference a variable or method directly.

- This provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility.

The `protected` Modifier

- Bottom line: keep variable private when you can.
- Only make a variable protected if there is a good reason why the child class needs direct access.

Visibility

- It’s important to understand a subtle issue.
- All variables and methods of a parent class (even private variables and methods) are *inherited* by a child class.
- But private variables and methods cannot be *referenced* by name in the child class.
  - They’re there! You just can’t call them by name.

- You must reference private variables through public methods of the parent class.

Parent Class Constructors

- Constructors are *not* inherited, even though they have public visibility.
  – The child class does not inherit the constructor.
  – The child class needs its own constructor.
  – Often, we still need to set up the “parent’s part” of the object and do additional things for the “child’s part” of the object.
- It’s good practice to call the parent constructor and then add any additional set up needed in the child class.

The `super` Reference

- Use the `super` reference to call the parent’s constructor:
  - The child class’s constructor calls the parent constructor.
  - The first line of the child constructor should use the `super` reference to call the parent’s constructor.
  - Without an explicit call, the default constructor of the parent class will be automatically called.
    - If there is not default constructor in the parent class, you will get a compiler error.

Practice

- Write constructors for the new classes.

OVERRIDING METHODS

Overriding Methods

- A child class can *override* the definition of an inherited method and provide its own implementation.
- The new method has the same signature (name and parameters) as the parent’s method, but will have a different implementation.

@Override Annotation

- Annotations contain metadata about programs.
- You can use annotations to create new compiler checks.
- @Override will ensure that the method header is correct.
- It’s good practice to always use this annotation when overriding a method!

Practice

- Override the playSample method in the child classes.

Overriding Methods

- When you override a method, you want to either:
  - Do something different
  - Do what the parent method does, but do more
- To *add on* to the parent's method, you can use the `super` reference.
  - Invoke the parent version of the method with `super.method()`

Practice

- Override the toString method in the child classes.
  - We want to print out the “parent part” of the object, but then print out *more* about the “child part.”

The `super` Reference

- Two uses of `super`:
  - Invoke the parent constructor (using super(…))
    - Must be first line of the constructor!
  - Invoke the parent’s version of an overridden method (using super.parentMethod(…))
    - Can be used anywhere!
    - Can be used to invoke *any* parent method.

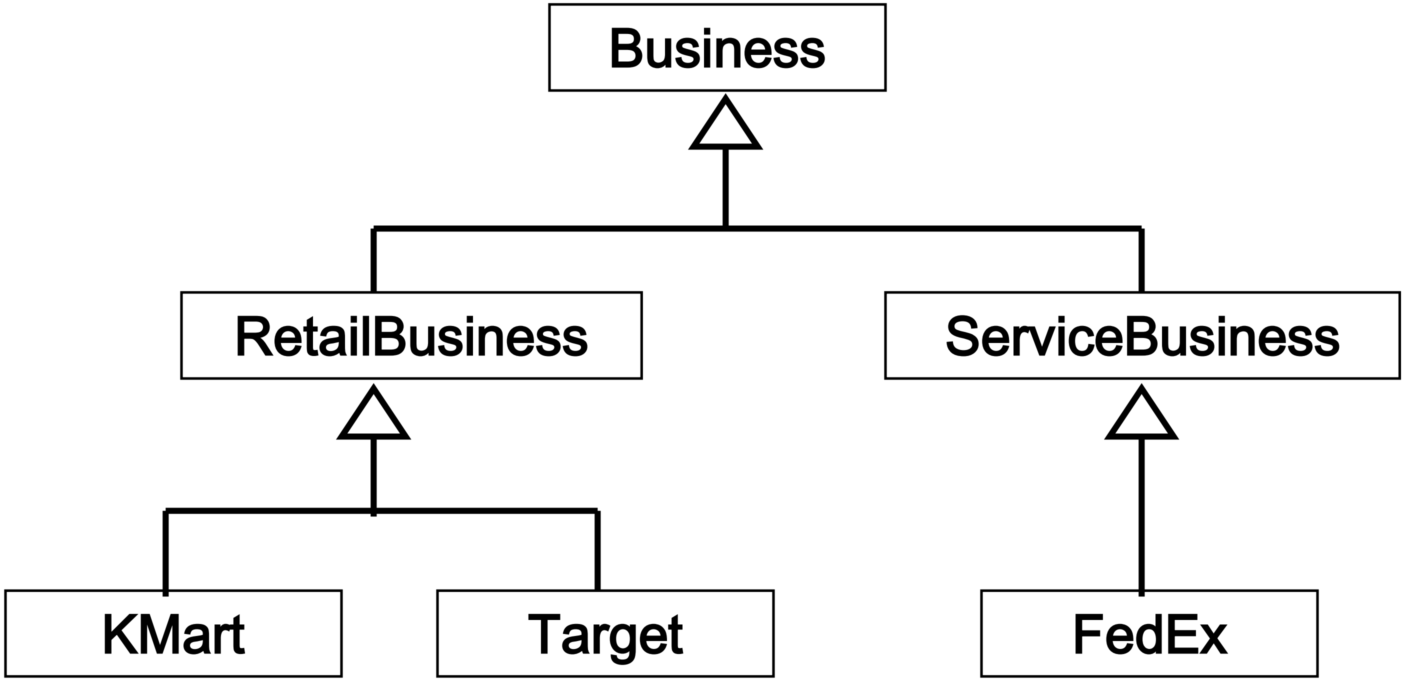
Overloading vs. Overriding

	Overloading	Overriding
Basic idea:	multiple methods in the same class with the **same** name and **different** signatures	two methods (one in the parent class and one in the child class) with the **same** name and **same** signature
Use for…	providing a similar operation in different ways by accepting different parameters	providing a similar operation with additional or altered functionality

CLASS HIERARCHIES

Class Hierarchies

- A parent class can be a child of another parent class and on and on, forming a *class hierarchy.*



Class Hierarchies

- Common features should be put up as high in the hierarchy as possible to increase the amount of reuse…
  - Write once, use often!
- An inherited variable or method is passed continually down the line
  - A child class inherits from all of its ancestor classes
  - Example: `Target` inherits all the methods/variables from `RetailBusiness` *and* from `Business`

The `Object` Class

- The `Object` class is the ultimate root of all class hierarchies
- `Object` is defined in the `java.lang` package
- *All* classes in Java are derived from `Object`
- If a class does not explicitly extend a class, it is automatically a child of the `Object` class

The `Object` Class Methods

- There are a few useful methods in `Object` that are inherited by all classes:
- `public String toString()`
  - Every time you write your own `toString` method, you are actually overriding this method from the `Object` class.
  - This is why the method header must match exactly!
  - In the `Object` class, this method returns a `String` with the name of the object's class and some other information (the memory address).
  - You should override this method with a text representation of your object.

The `Object` Class Methods

- `public boolean equals(Object obj)`
  - The inherited version returns true if two references are aliases
  - You can override this method to define *meaningful* or *logical* equality in a more appropriate way
    - Examples: same ID, same name, etc.
  - Example: `String` overrides equals to return true if two `String` objects contain the same characters

Implementing Equals

- We need to first check that we've been given an object with the right type.
  - We do this with the `instanceof` operator
  - `instanceof` returns a boolean that represents whether an object is an *instance of* a class
- If the parameter is the right type, we use a *cast* to tell the compiler that we want to treat the parameter as that type.
- Then we compare whatever information we are using for equality.
  - Will often use the equals method (or equalsIgnoreCase method) to compare String varaibles and == to compare numeric variables.

Implementing Equals

```
@Override
public boolean equals(Object obj) {
      if(obj instanceof MyClass) {
            MyClass otherObject = (MyClass) obj;
            // compare whether the variables are the same
      } else {
            return false;
      }
}
```

The `equals` Method header

- A common error is to accept a parameter of your class, instead of object.
- This is incorrect because this method no longer overrides the inherited method!
- Be very careful not to do this!

X

```
public boolean equals(Employee e)
```

Practice

- Write an equals method for the AudioItem class.
- Write an equals method for the Employee class.

DESIGNING FOR INHERITANCE

Multiple Inheritance

- Java supports *single inheritance*
  - A child class can have only *one* parent class
- Other languages allow *multiple inheritance*
  - A class can be derived from multiple classes
  - A child class inherits the data and methods of all parents
  - Collisions (e.g., the same variable name in two parents) must be resolved

- You can use interfaces to get some of the functionality of multiple inheritance without the overhead

Designing for Inheritance

1. Inheritance should always represent an *is-a* relationship.
2. Find common characteristics of classes and push them as high in the class hierarchy as possible.
    a. Maximum reuse!
3. Override methods to tailor or change the functionality of a child.
    a. If you aren’t changing anything about the parent’s method, do not override it!

Designing for Inheritance (cont.)

4. Allow each class to manage its own data.
    a. Use the `super` reference to invoke the parent’s constructor to set up its data.
    b. Use the `super` reference to invoke the parent’s existing methods.
5. Even if there are no current uses for them, override general `Object` methods like `toString` and `equals`.
6. Use visibility modifiers carefully to provide needed access without violating encapsulation.

Class Core Components

- Contain:
  - private instance variables
  - constructors
  - public getters and setters (with appropriate validity checking)
  - toString method (overridden!)
  - equals method (overridden!)
  - public and private class-specific methods

Restricting Inheritance

- The `final` modifier can be used to prevent inheritance
- If the `final` modifier is applied to a method, then that method cannot be overridden in any child classes
- If the `final` modifier is applied to a class, then that class cannot have any child classes

Practice

- Review the Employee class.
- Write classes to represent paid employees (some hourly, some salaried) and unpaid employees (interns).
  - Override appropriate methods.