

Inheritance

- Objectives - when we have completed this set of notes, you should be familiar with:
 - deriving new classes from existing classes
 - the `protected` modifier
 - creating class hierarchies
 - abstract classes
 - indirect visibility of inherited members
 - designing for inheritance

Inheritance

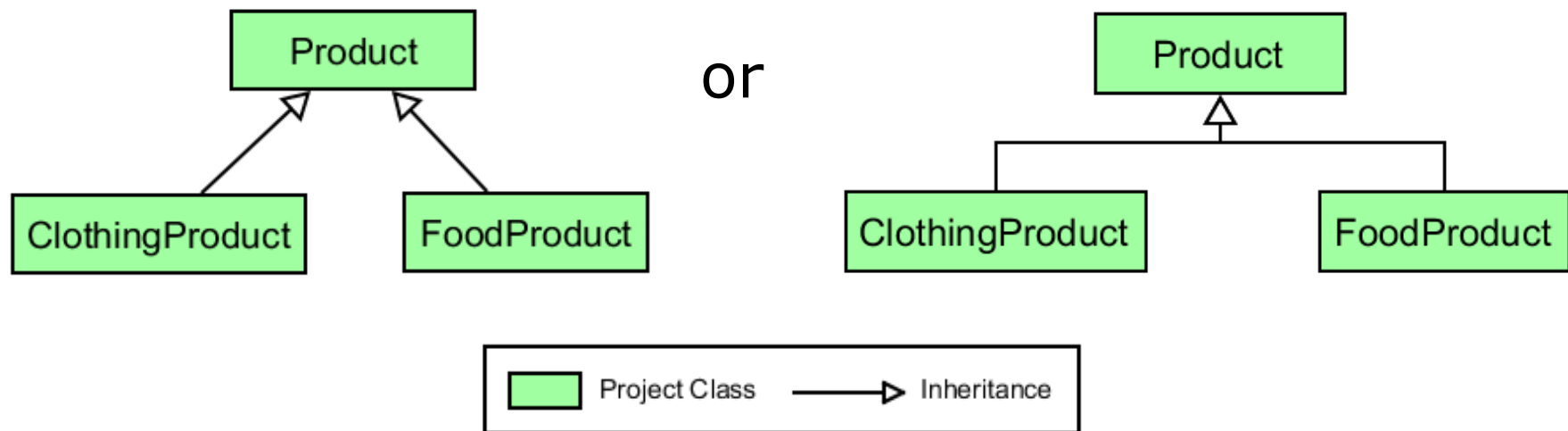
- Suppose that you are creating a program to keep track of products in a store's inventory
- You need to represent the following:
 - General products -> price, name
 - Clothing products -> price, name, **size**
 - Food products -> price, name, **isRefrigerated**
- Each of the above classes needs variables for price and name, but the clothing products and food products classes have additional characteristics

Inheritance

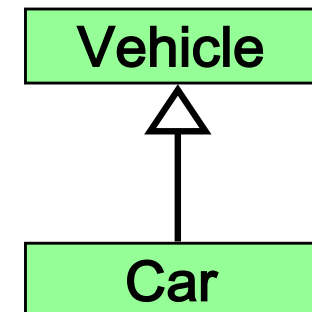
- Possible solutions:
 - Write classes Product, FoodProduct, ClothingProduct and include price and name (and methods) in each
 - Use *inheritance* so that you only have to write common code once [We'll use this approach!]
- The existing class ([Product.java](#)) is the *parent class, superclass, or base class*
- Each derived class (e.g., FoodProduct, ClothingProduct) is the *child class* or *subclass*
- A child classes inherits the variables and methods defined by the parent class

Inheritance

- UML Class Diagram* shows inheritance:



- is-a* relationship: the child *is a* more specific version of the parent



*Open the project file to generate UML Class Diagram

Deriving Subclasses

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

```
public class ClothingProduct extends Product {  
  
}
```

- We say `ClothingProduct` is derived from `Product`; or `ClothingProduct` is subclass of `Product`
- Two children of the same parent are called *siblings*
 - [ClothingProduct](#) and [FoodProduct](#) are siblings

The protected Modifier

- Variables / methods / constants declared as *private* cannot be referenced in a child class
 - This is fine unless the child class needs to reference a specific variable or method
- Variables / methods / constants declared with *public* access **can** be referenced in a child class
 - But declaring variables as public violates encapsulation!
- Solution: the `protected` access modifier
 - Only allows subclasses (child classes) and classes in the same package to access the variable

The protected Modifier

- If instance variables for price and name need to be accessed directly in subclasses of Product, we can use the protected modifier:

```
public class Product {  
    protected String name;  
    protected double price;  
  
}
```

- Variables name and price can now be accessed by FoodProduct and ClothingProduct:

```
public class ClothingProduct extends Product  
  
public class FoodProduct extends Product
```

The super Reference

- Constructors are **not** inherited
- However, you can avoid repeating all of the code in the parent's constructor using the reserved word `super`
- The first line of a child's constructor can use the `super` reference to call the parent's constructor (See [ClothingProduct](#) constructor)
- The `super` reference can also be used to reference variables and methods defined in the parent class (See `toString` in [FoodProduct](#))

Parameterless Constructors

- Recall that Java provides a parameterless constructor for your class if you do not provide a constructor.
- If a constructor in a subclass does not call the super constructor directly, the parameterless constructor of the superclass is automatically called - - - all the way up the hierarchy.
 - If there is no parameterless constructor in the superclass (parent), then you **must** call the super constructor in the child class; otherwise a compile-time error will occur. Modify ClothingProduct so that the super constructor is not called to see this error.

[InheritanceExample.java](#) (Open project file to generate UML Class Diagram)

Overriding Methods

- A child class can *override* the definition of an inherited method
- The new method must have the same signature as the parent's method, but can have a different body
- For example, suppose that food items do not factor tax into their total price
 - The totalPrice method is redefined in [FoodProduct](#)

Overriding

- The concept of overriding can be applied to data and is called *shadowing variables*
 - For example, ClothingProduct could also have a variable called name
 - You would have to use super.name to access the name variable in the parent class
 - Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

The exception would be for class constants since they are usually qualified with the class name

Overloading vs. Overriding

- Recall that overloading deals with multiple methods with the same name but with **different signatures**
 - Defines a method of the same name as an existing method but with different parameters
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
 - Redefines a method of the parent class (same name and parameters)

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
- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

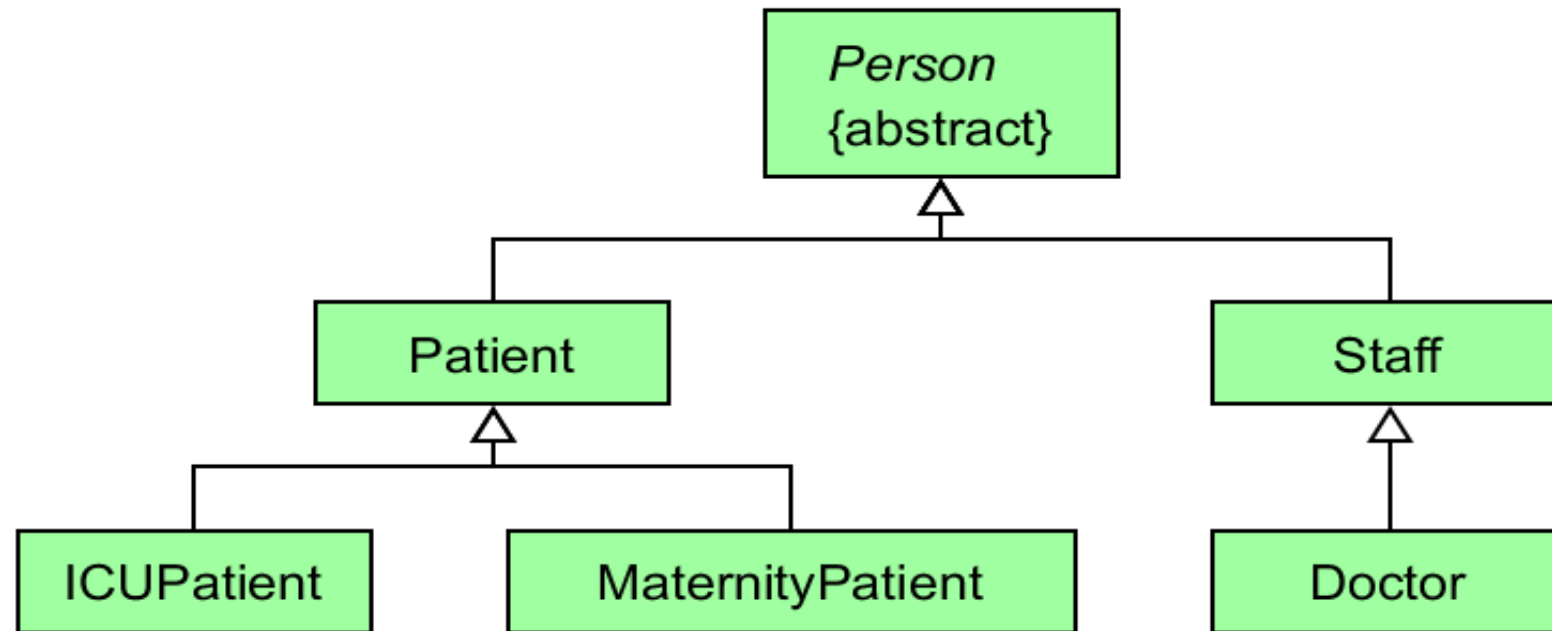
- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we define the `toString` method, we are actually overriding an inherited definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class along with the hash code for the object

The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- The `String` class has overridden the `equals` method inherited from `Object` in favor of a more useful version returns true if two `String` objects contain the same characters
- Note: If you override the `equals` method, our Checkstyle rules require the `hashCode` method to be overridden as well)

Class Hierarchies

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



(Open the Hospital.gpj project file to generate UML Class Diagram)

Class Hierarchies

- Common features should be put as high in the hierarchy as is reasonable (see [Person](#))
- A child class inherits from all its ancestor classes
 - [Doctor](#) inherits all protected and public fields and methods from [Staff](#) **and** [Person](#)
 - See the toString method in [Doctor.java](#). It accesses firstName and lastName from Person.java as well as phone in Staff.java

(Open Hospital project file to generate UML Class Diagram)

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that defines certain variables and behavior
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:
 - Example: We would never really need a “Person” object, but it can define fields and methods common to Patients and Staff

```
public abstract class Person
```

Abstract Classes

- An abstract class can contain abstract methods with no definitions (similar to an interface)
 - The `abstract` modifier must be applied to each abstract method
- The child of an abstract class must override the abstract methods of the parent or it must be declared to be abstract as well
 - `getId` from `Person` is defined in `Staff`, `Doctor`, and `Patient`
 - Note that it is **not** defined in `ICUPatient` and `MaternityPatient` since it was handled by `Patient`

Abstract Classes

- Why define abstract methods?
 - The hospital is never going to instantiate a Person object, but methods like getName are self-explanatory and will be the same for all child classes.
 - The generation of an id is necessary for all child classes, but it's going to be different for patients and staff
- An abstract method cannot be defined as `final` or `static`

Inheritance

- Discussion: What are the benefits of inheriting methods and variables from an existing class?
 - Avoiding redundancy
 - Code reuse
 - Testing
 - Maintainability

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
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces

Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data
- Override the `toString` and `equals` methods from the `Object` class with appropriate definitions as needed
- Use abstract classes to represent general concepts that lower classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

Accessibility Revisited

- Variables and methods of a parent class, are inherited by its children
- Private variables and methods inherited by subclass instances cannot be referenced directly
- However, the subclass can reference them indirectly using its parent's public methods (e.g., getters, setters)
- The `super` reference can be used to refer to the parent class, even if no object of the parent exists

Restricting Inheritance

- The `final` modifier can be used to restrict inheritance
- If the `final` modifier is applied to a class, then that class cannot be used to derive subclasses (e.g., if class A is final then class B cannot extend A)
 - Thus, an abstract class cannot be declared as final
- If the `final` modifier is applied to a method, then that method cannot be overridden in any descendent classes
- These are key design decisions, establishing that a method or class should be used as is