CSD221 Lecture 4

Lecture 4 Polymorphism

[**Objectives**](#_ev84ew4t3es9) **2**

[**Class Types**](#_c7uzkfdpkgc4) **2**

[Activity/Feedback 1](#_1c7mocu0z6td) 6

[**Substitutability**](#_pkbotf8q2qn6) **8**

[**Polymorphism**](#_8uuirzul1m1y) **9**

[The CashTill class](#_7w2aahkibssl) 9

[Activity/Feedback 2](#_pv61r7gxny15) 14

[Activity/Feedback 3](#_hmv368oz33wt) 18

[Activity/Feedback 4](#_dthw7rwdv41y) 21

[**Interfaces**](#_81n6f64s03bl) **23**

[Tickets](#_uvhx49x2ot6x) 27

[**Extensibility**](#_12sxruprh8q2) **34**

[Activity/Feedback 5](#_2fs4uw2mt442) 35

[Activity/Feedback 6](#_zfulti1ovorn) 37

[Distinguishing Subclasses.](#_7f65kb25c2mo) 39

# Objectives

* understand how polymorphism allows us to handle related classes in a generalized way.
* understand the implications of polymorphism with overridden methods
* define interfaces to extend polymorphism
* appreciate how polymorphism provides for extensibility

# 

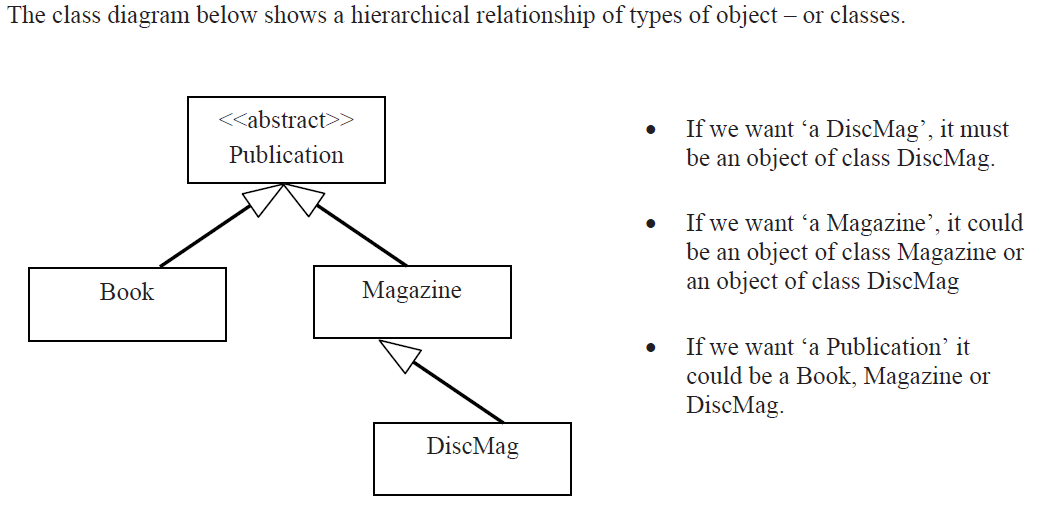
# Class Types

Within the hierarchical classification of animals

* pinky is a pig (species sus scrofa)
* pinky is also (more generally) a mammal
* pinky is also (even more generally) an animal

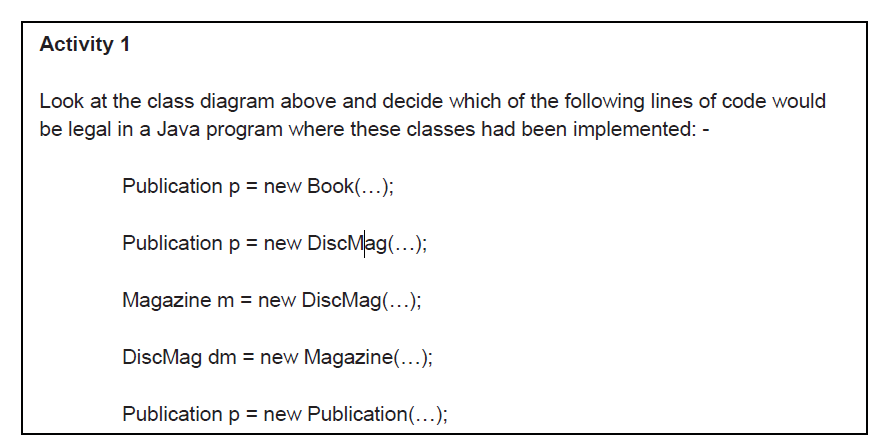
At different levels of detail

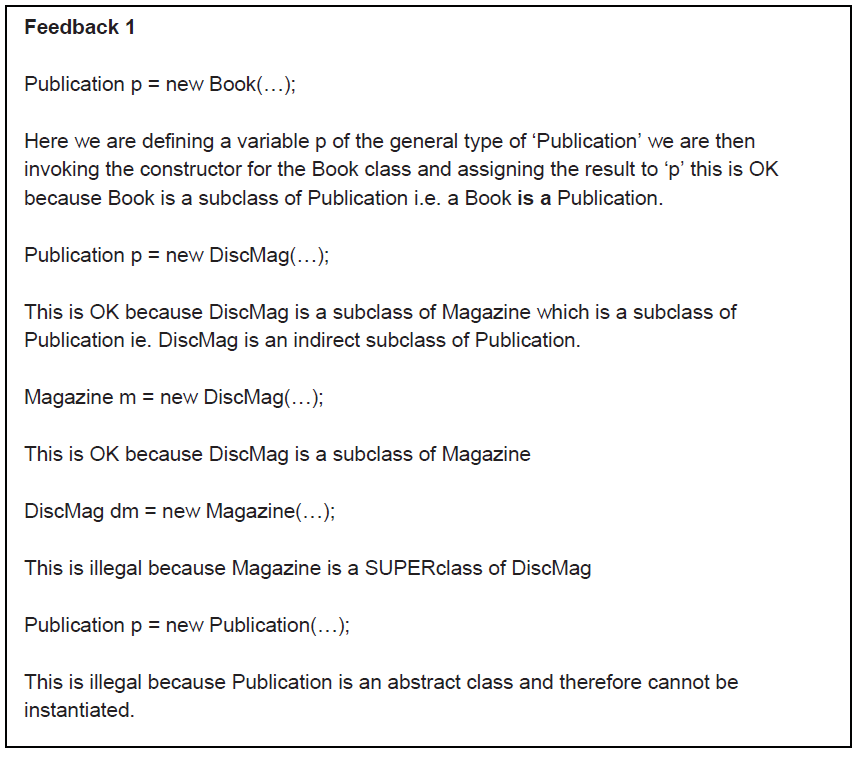
* higher level = less specific
* lower level = more specific
* If someone asks for a Pig you can give them Pinky or any other Pig
* If someone asks for a mammal you can give them Pinky, any other Pig or any other animal like a lion or a dog.
* That's because Pinky is a Pig, which is a mammal, which is an animal, etc... (the "is a" relationship)
* in OOP, a subtype class can be substituted wherever a supertype is expected.



* you can substitute a subclass wherever a superclass is required but not the other way around.

## Activity/Feedback 1





# Substitutability

* substitutability:
* a subclass can be used wherever a superclass is required
* this has the implication that the subclass is truly a substitute for its superclass, that is, that it provides at least the same functionality of the superclass.
  + when overriding methods, you must be careful to ensure that the overridding method does not change the functionality of the overridden method. It should enhance the functionality but not change its meaning.
    - recvNewIssue in discmag still does the same basic job (fullfills the contract) of updating the number of copies and the current issue but adds the job of checking for the disc.
* What is a Publication?
  + An object which supports (at least) these operations
    - void sellCopy()
    - double getPrice()
    - int getCopies()
    - void setCopies(int copies)
    - String toString()
  + Inheritance guarantees that all subclasses of Publication support these operations.
  + a subclass may NEVER remove an inherited operation it can only enhance (extend) them.

# Polymorphism

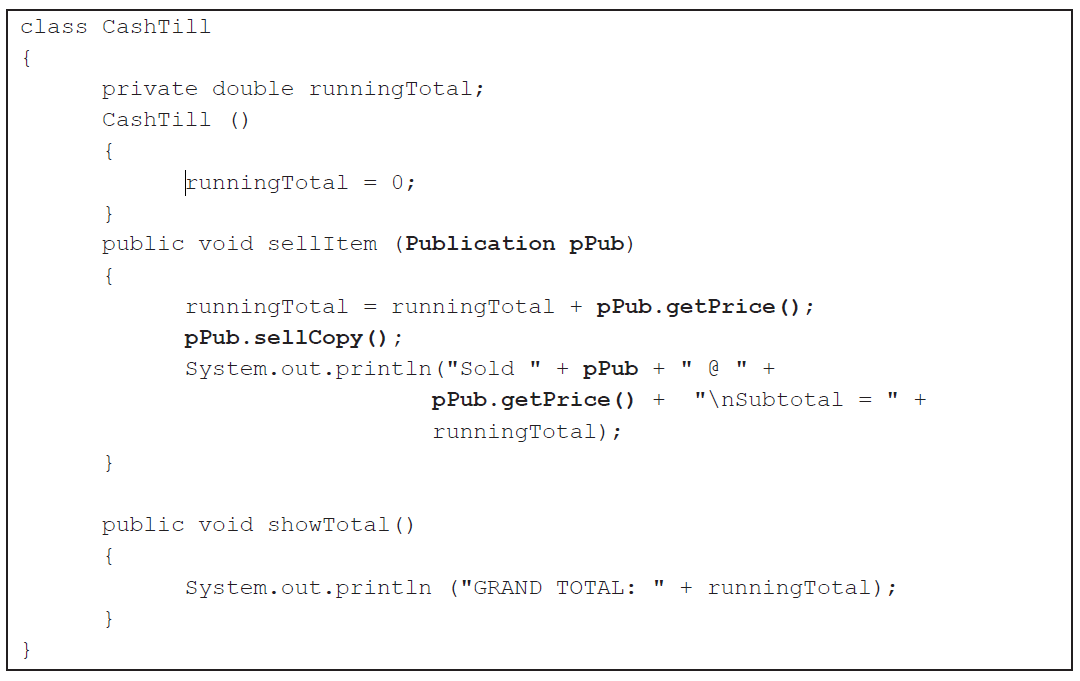
* since a subclass **"IS A"** superclass and a subclass guarantees the same operation of the superclass, you can call operations on the subclass without worry, that the subclass fulfills the contract of its superclass.
  + This is called POLYMORPHISM
* A Publication comes in different "flavours"
  + it can be a book, a magazine or a discmag
  + we can call sellCopy() on any of these publications.
  + p.sellCopy() will work as expected regardless of what "p" is.
  + That’s polymorphism.

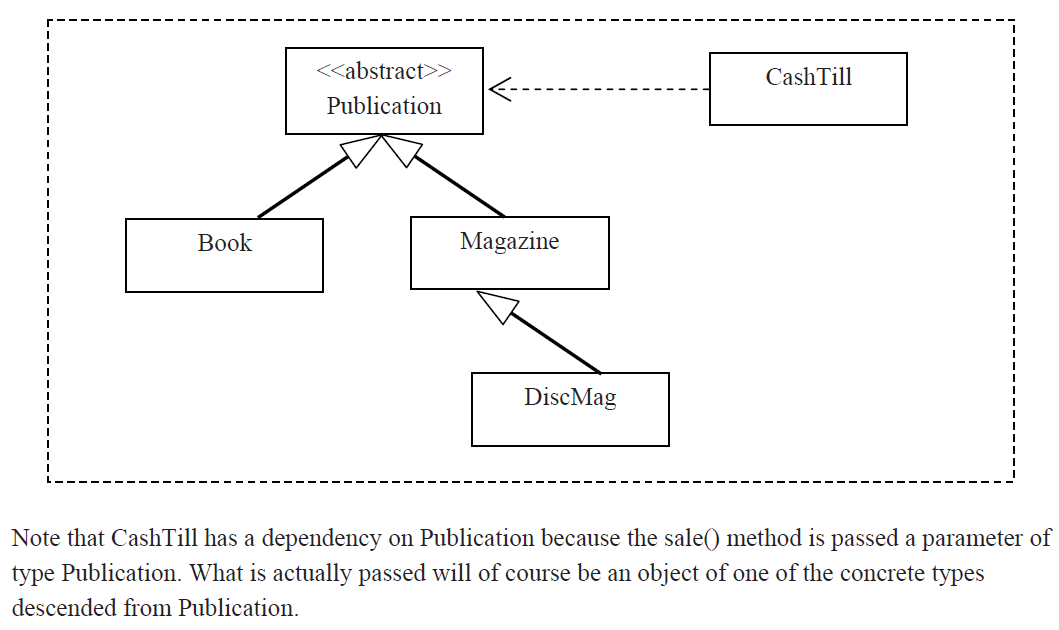
Polymorphism in Practice

## **The CashTill class**

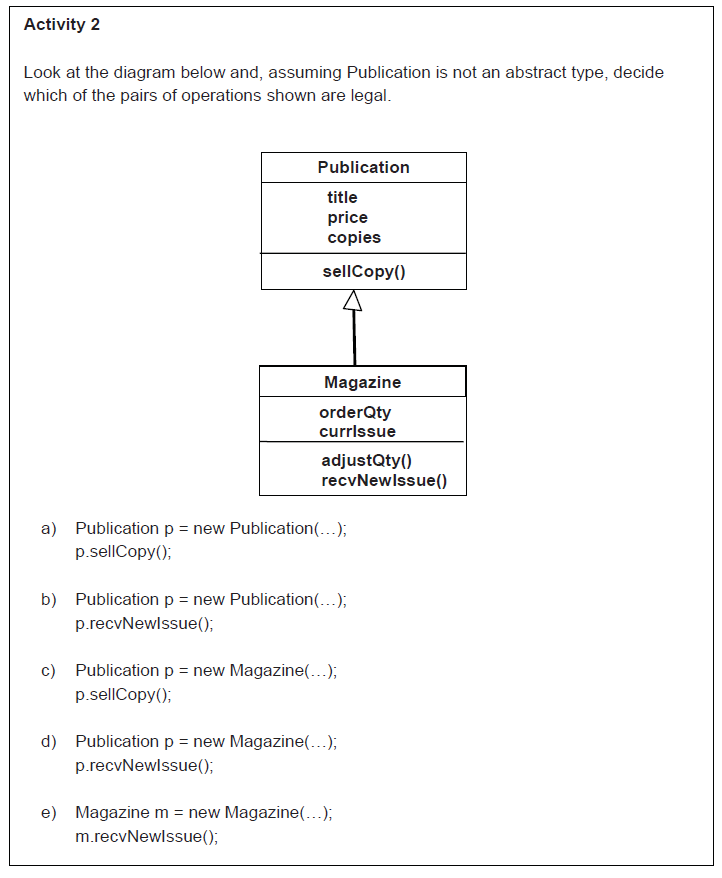
* CashTill is a class that processes a sequence of items being sold.
* **Without polymorphism,**
  + you would need a separate method for each type of item being sold:  
    sellBook(Book book)  
    sellMagazine(Magazine magazine)  
    sellDiscMag(DescMag discMag)
* **With polymorphism:**
  + you only need one method  
      
    sellItem(Publication pub)
  + this works because every sub class of Publication IS A superclass, so you can pass a book, a magazine or a disc magazine and the method will sell it.
* this enables "extensibility":
* we can extend our application by creating new subclasses without having to write a "sellNewsublass" method. The new subclass is still a publication which means you can just use the existing sellItem(Publication pub) method.
* without polymorphism, we would have to check the type of the Publication, whether it’s Book, Magazine, etc... and then call the appropriate method:
  + if p is a book call sellCopy() for book
  + if p is a magazine, call sellCopy() for magazine
  + if p is a DiscMag, call sellCopy() for DiscMag
* with polymorphism:
  + p.sellCopy();

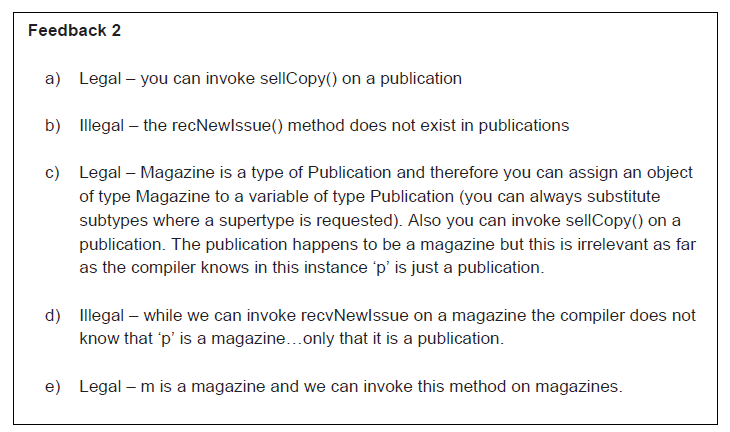
**The CashTill class : utilizes polymorphism**

****

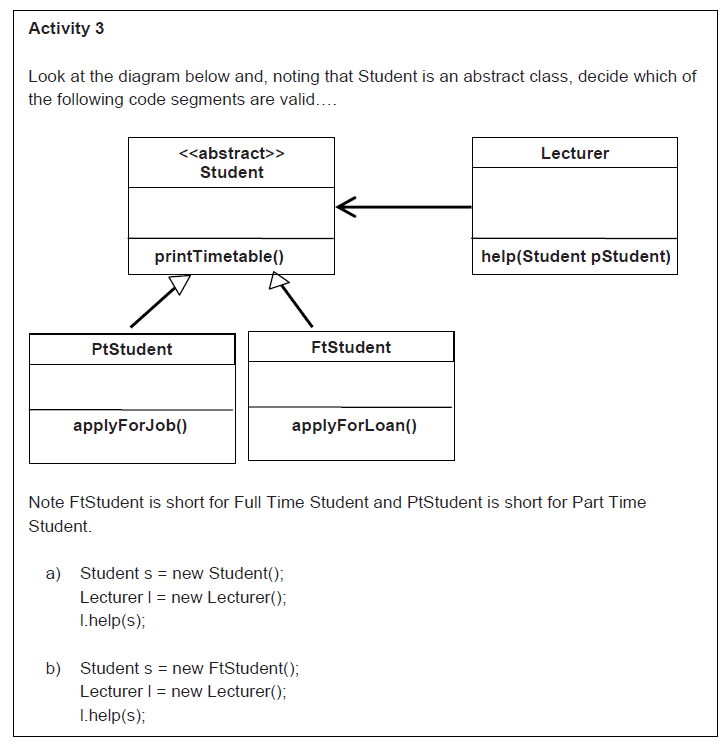
****

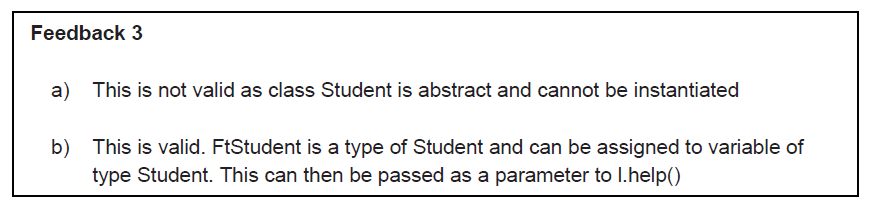
## Activity/Feedback 2

****

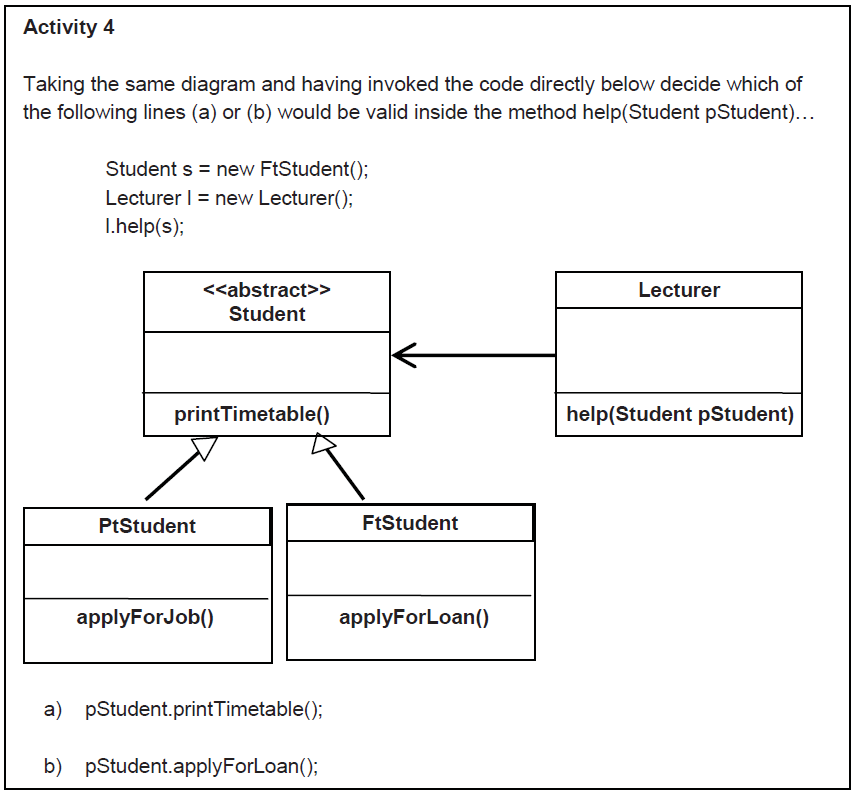
****

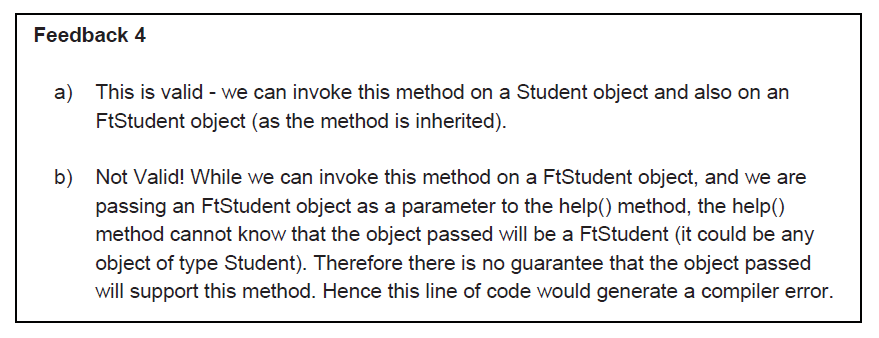
## Activity/Feedback 3

****

****

## Activity/Feedback 4

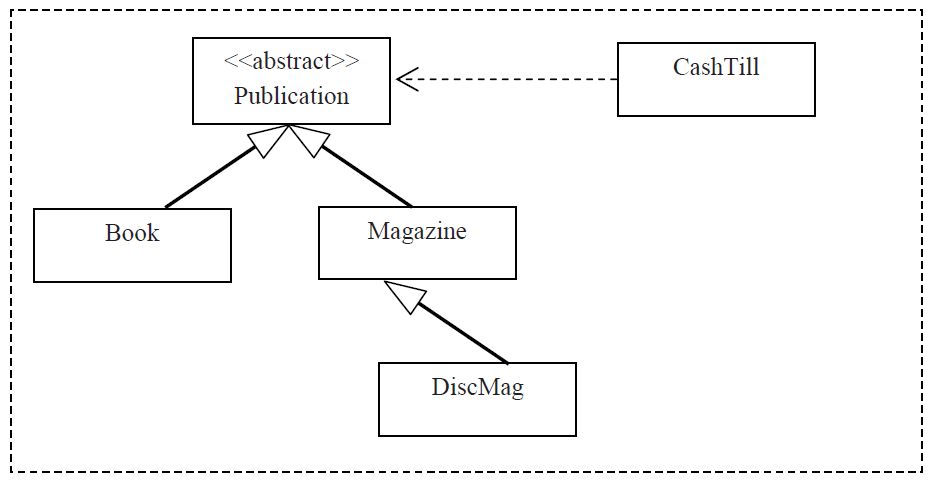
****

****

# Interfaces

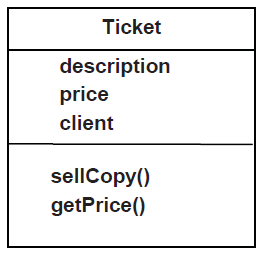
* first, review inheritance
* there are 2 aspects to inheritance
  + a subclass inherits the "interface", access to public methods
  + a subclass inherits the implementation of its superclass
  + the keyword "extends" automatically applies both these aspects
  + **public class Book extends Publication {}**
  + a subclass is a subtype
    - its interface must include all of the interface of its supertype although it may modify the interface through overloading methods
  + sometimes we want two classes to share an interface BUT we don’t want them in the same class hierarchy because
    - they aren't really related by a true "is a" relationship
    - we want a class to have interfaces shared with more than one superclass, but java does not support Multiple Inheritance
    - we want to create a "plug and socket" arrangement between software components that don't exist yet.
      * for example, we want to make sure two cars have the same "interface", that is they operate the same way but we leave the implementation (exactly "how" it’s done) up to the engineers.
        + 2 cars may have an accelerator pedal, that accelerates the car when you depress the pedal (same interface), but one might be electric, so it’s implemented by increasing the current to the motor while the other is gasoline, so it’s implemented by increasing fuel flow to the carb/injector.
* "interface" has 3 meanings in java
  + the public members of a class
  + the "user interface" or GUI
  + a specific java construct described below.

**Review the Publication hierarchy**

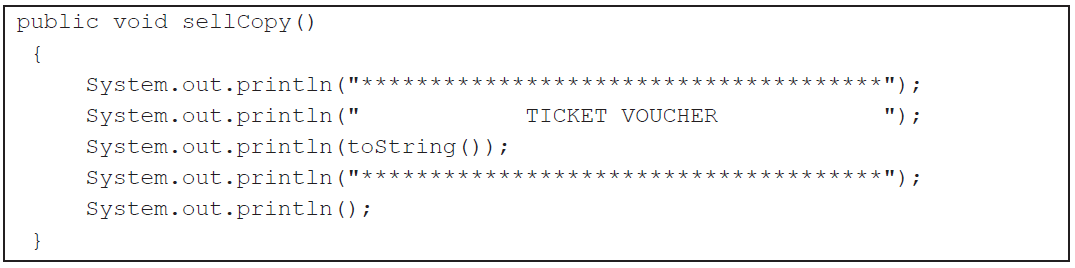


## Tickets

* assume that in addition to publications, we also want to sell Tickets
* Tickets are not Publications so we can’t subclass Publication but we still want to be able to sell them just like we sell Publications
* we want Tickets and Publications to be treated the same way (be able to sell them) but they are not alike so we cant include Tickets in the same class hierarchy as a Publication.
  + In short we want them to have the same interface but not the same implementation
* Tickets are different because
  + we don’t have a finite 'stock' but print them on demand at the till
  + tickets consist of a description, price, client
  + ticket sales are really a service not a product
* Tickets and Publications are similar though
  + they both have getPrice()
  + they both have sellCopy()
  + both methods are needed by CashTill



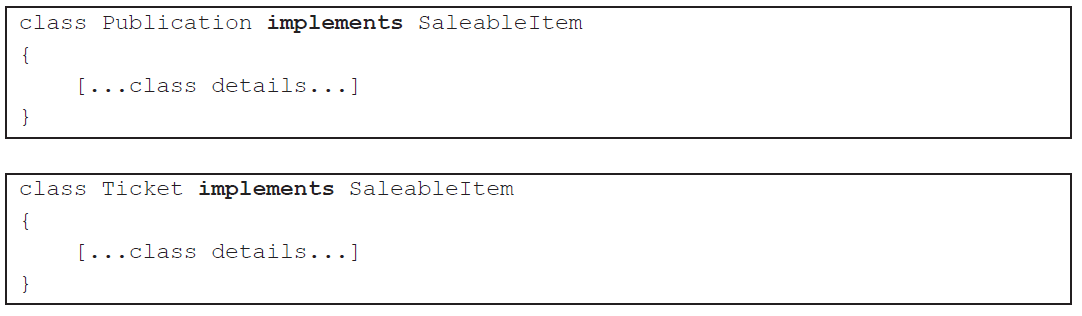
* both Tickets and Publications have a sellCopy() method (same interface) but are implemented very differently
  + Ticket.sellCopy()
    - just print one
  + Publication.sellCopy()
    - reduce stock by one



* the implementation of sellCopy() for Tickets and Publications are very different so we don’t want Ticket in the same class hierarchy as Publication (dont want to inherit its functionality) but we do want to inherit the same interface (we want to sell Tickets just like we sell Publications)
* to be able to sell Tickets through CashTill all we need is sellCopy() and getPrice()
* since Ticket has these, we can sell them.
* in fact any class that has sellCopy() and getPrice() can be used by CashTill and hence can be sold.
* to enable this functionality, we define a set of operations or "an Interface" called SaleableItem, which defines sellCopy() and getPrice() as required interface operations
  + now, any Object we wish to be saleable, MUST "implement" the SaleableItem interface

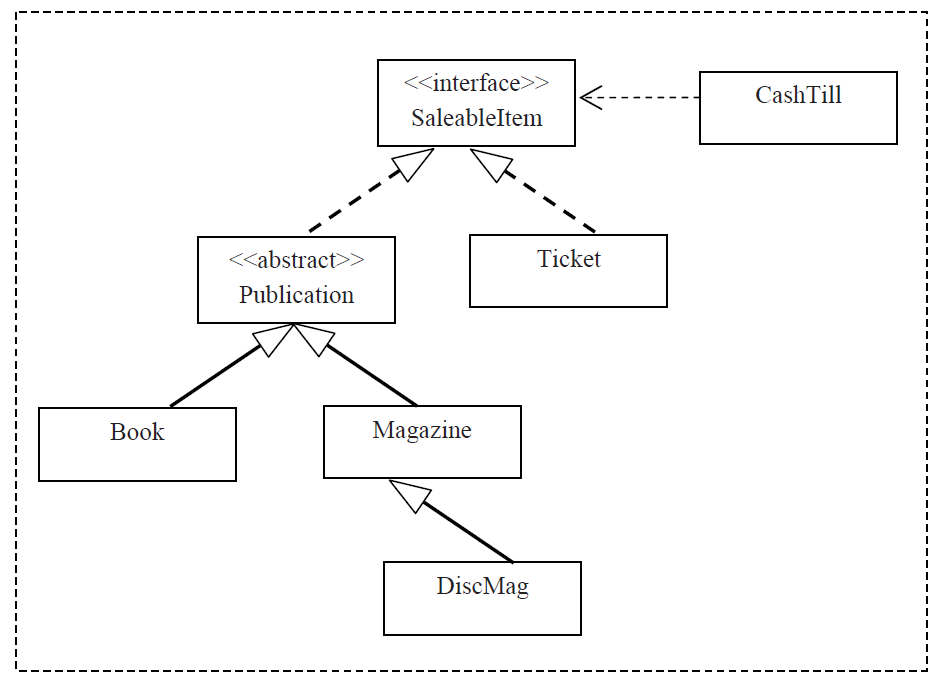


* NOTE\*\* an interface defines operations only. The method signature is defined but the body of the method is empty.
* subclasses "extend" superclasses
* subclasses may "implement" interfaces
* if a subclass extends a superclass then the subclass "IS A" superclass (a Book IS A Publication)
* if a sublcass implements an interface then the subclass "IS A" interface (a Book IS A SaleableItem and so is a Ticket)
* so a Book IS A Publication and it IS A SaleableItem too, which means, because of Polymorphism, we can sellCopy() on both Books and Tickets because both are SaleableItems.. Look, multiple inheritance without all the problems associated with true multiple inheritance.



**Polymorphic CashTill**

* currently, CashTill can sell any Publication
* public void sellItem(Publication p)
* we also want to sell tickets. Tickets and Publications are both SaleableItems so do this
* public void sellItem(SaleableItem p)

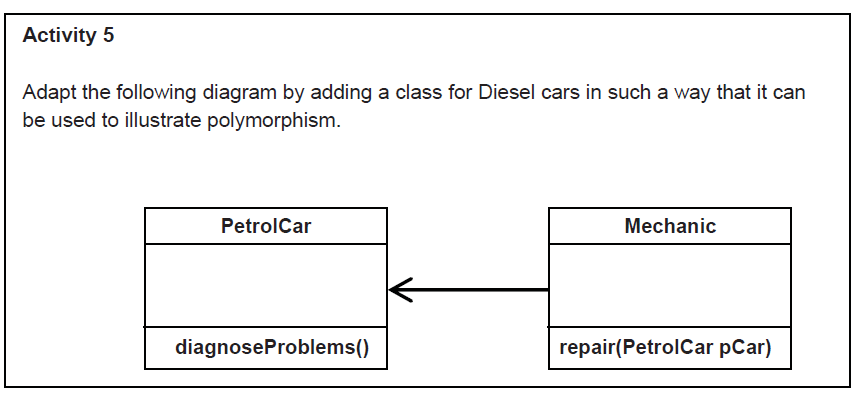


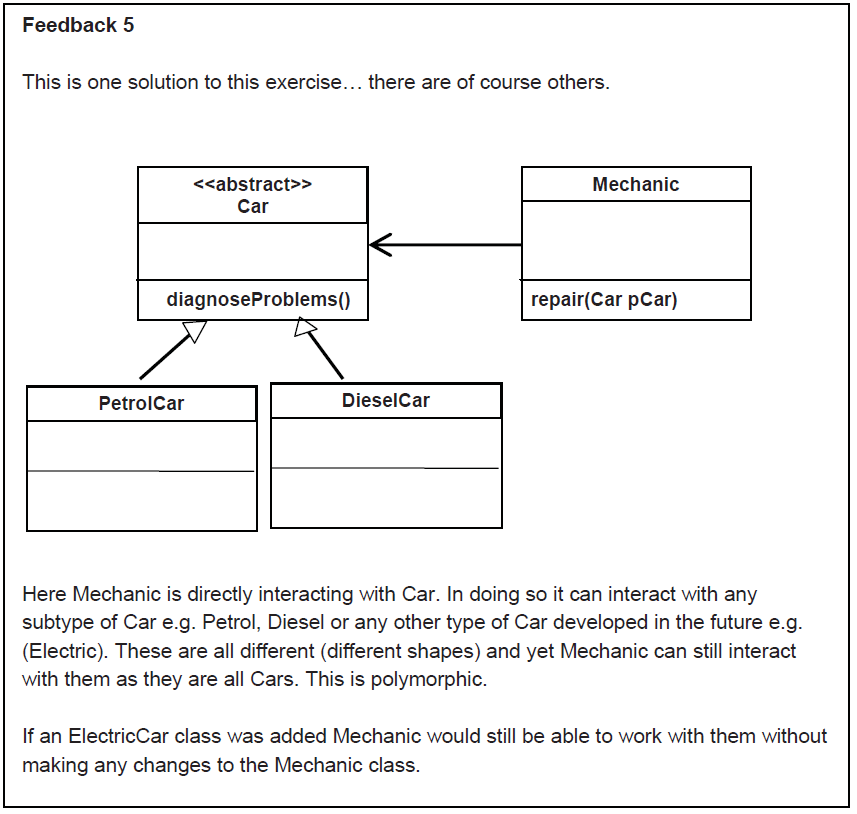
* the relationship from Publication and Ticket to SaleableItem are like inheritance arrows except they're dotted indicating each class implements the interface.

# Extensibility

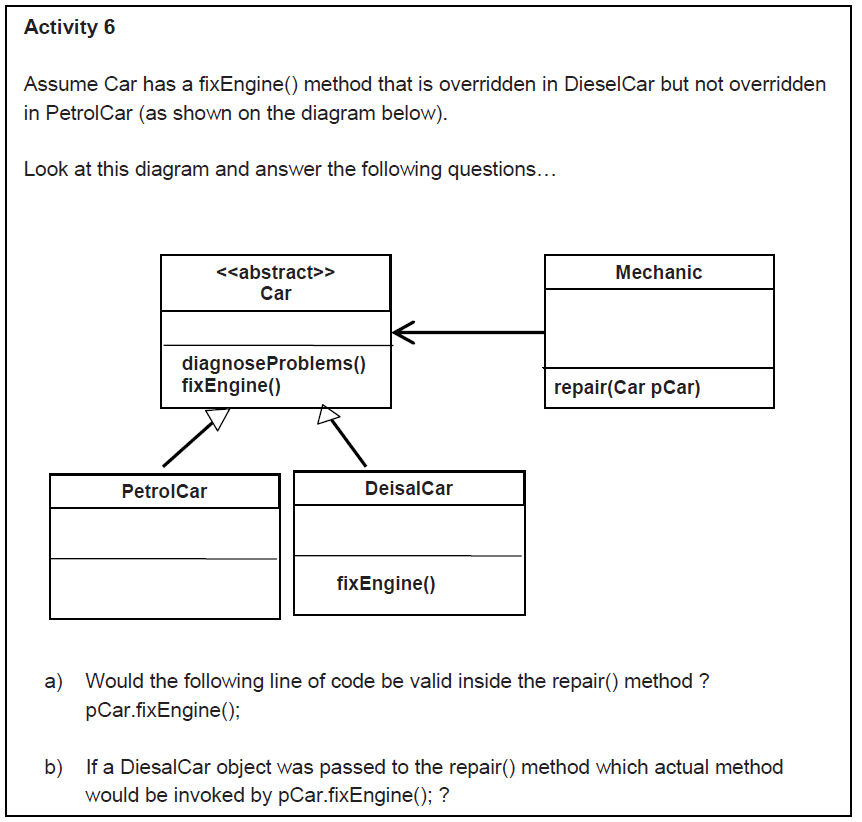
* consider this
  + new products and services can be created and CashTill can process them without adding any new code to CashTill.
  + all you have to do is extend a Publication OR implement SaleableItem
  + That's how polymorphism allows for extensibility and how "extending" classes and "implementing" interfaces implements polymorphism.
  + "interface" allow software components to plug together more flexibly and extensibly just like the standard power plug interface allow you to plug in many different kinds of device.

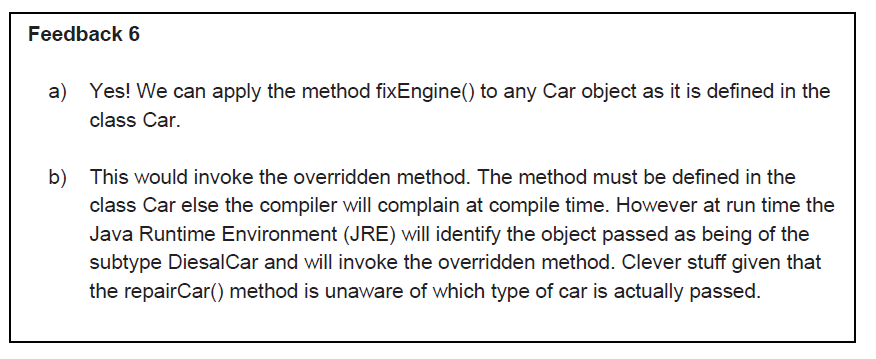
## Activity/Feedback 5





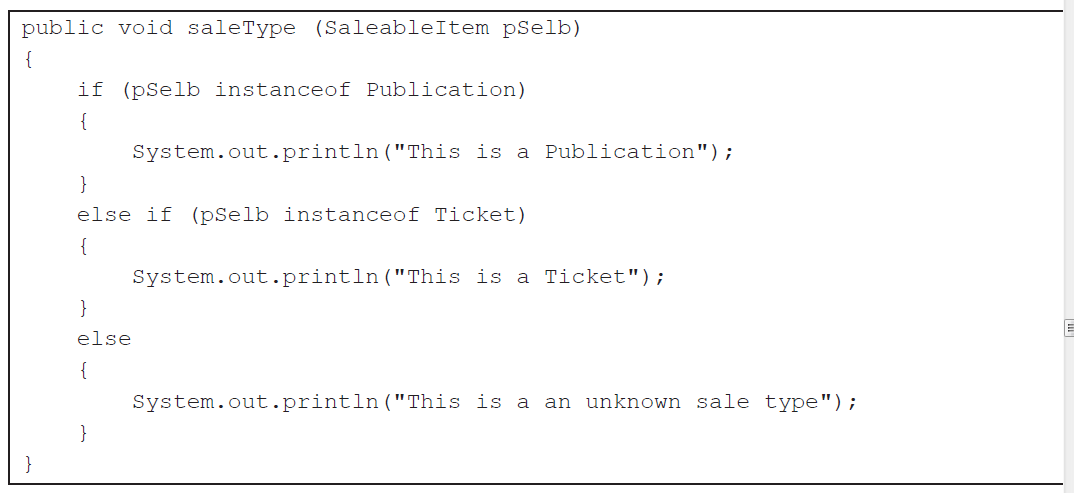
## Activity/Feedback 6





## Distinguishing Subclasses.

* use "instanceof" to determine type



* the above example compromises polymorphism.
  + if we create a new class we now have to modify the "if" statements above, which creates a maintenance nightmare.
  + rather than use "instanceof" its better to package different behaviours in the subclasses themselves. We do this by adding a "describeSelf()" method in the interface SaleableItem, so that all saleable items would have to implement a method that describes itself instead of using instanceof to do it.