ECE25100: OBJECT ORIENTED PROGRAMMING

Note 8 _ Inheritance and TypeCasting

Instructor: Xiaoli Yang

Introduction

- The proper use of inheritance will allow us to reduce the amount of code that we need to write, resulting in quicker implementation time, less maintenance time and hence reduced costs overall.
- When discussing Inheritance, the notion of Abstraction and Polymorphism come into play as well.
- We will discuss the JAVA notion of Interfaces which is quite a "neat" concept.

Inheritance and Abstraction

- Inheritance is the act of getting (for free) shared state and behavior from more general types of objects.
- A subclass (child) inherits the following from all of its superclasses (ancestors in the class hierarchy):
  - instance variables (fields)
  - methods
- This means that a subclass:
  - is made up of the state (i.e., characteristics) of its superclasses as well as its own unique state.
  - can respond to the messages represented by the methods in its superclasses.

Inheritance and Abstraction (Cont'd)

There are many advantages of using Inheritance:

- allows code to be shared between classes
  - software reusability
- saves time when programming
  - less code needs to be written
- encourages re-use of well designed objects
  - make use of proven and tested code
- helps keep code simple
  - inheritance is natural in real life

Inheritance and Abstraction (Cont'd)

- A subclass:
  - is a class that inherits state (i.e. instance variables) and behavior (i.e., methods) from some other class.
    - can therefore respond to messages defined in its superclasses
  - cannot access private methods or private instance variables from its superclasses.
- A superclass:
  - is a class from which its subclasses inherit state and behavior.
  - is more general than its subclasses. Subclasses are more specific.
- A direct superclass:
  - is the superclass from which the subclass explicitly inherits.
- Some languages allow Multiple Inheritance:
  - is the term given to a class that inherits from more than one direct superclass.
  - is NOT supported in JAVA (but it is in C++).
  - can be partially "faked" through the use of interfaces

Inheritance and Abstraction (Cont'd)

- There are methods in the Object class that allow us to determine what class an object belongs to and also to get the name of the class:
  - getClass() - This is sent to any instance and it returns a Class object representing the class of the object.
  - getName() - This is sent to any Class object and it returns the name of the class.
- Here is an example of how to use these:

```
public class ClassTest {
    public static void main(String args[]) {
        BankAccount     anAccount = new BankAccount();
        Class           aClass = anAccount.getClass();
        System.out.println(aClass);    // prints class BankAccount
        System.out.println(aClass.getName());  // prints BankAccount
    }
}
```

- Notice that the getClass() method is sent to any instance ... while the getName() method can ONLY be sent to a Class object.

Inheritance and Abstraction (Cont'd)

- Another interesting keyword is the instanceof keyword. This allows us to determine if an object is an "instance of" a particular class:

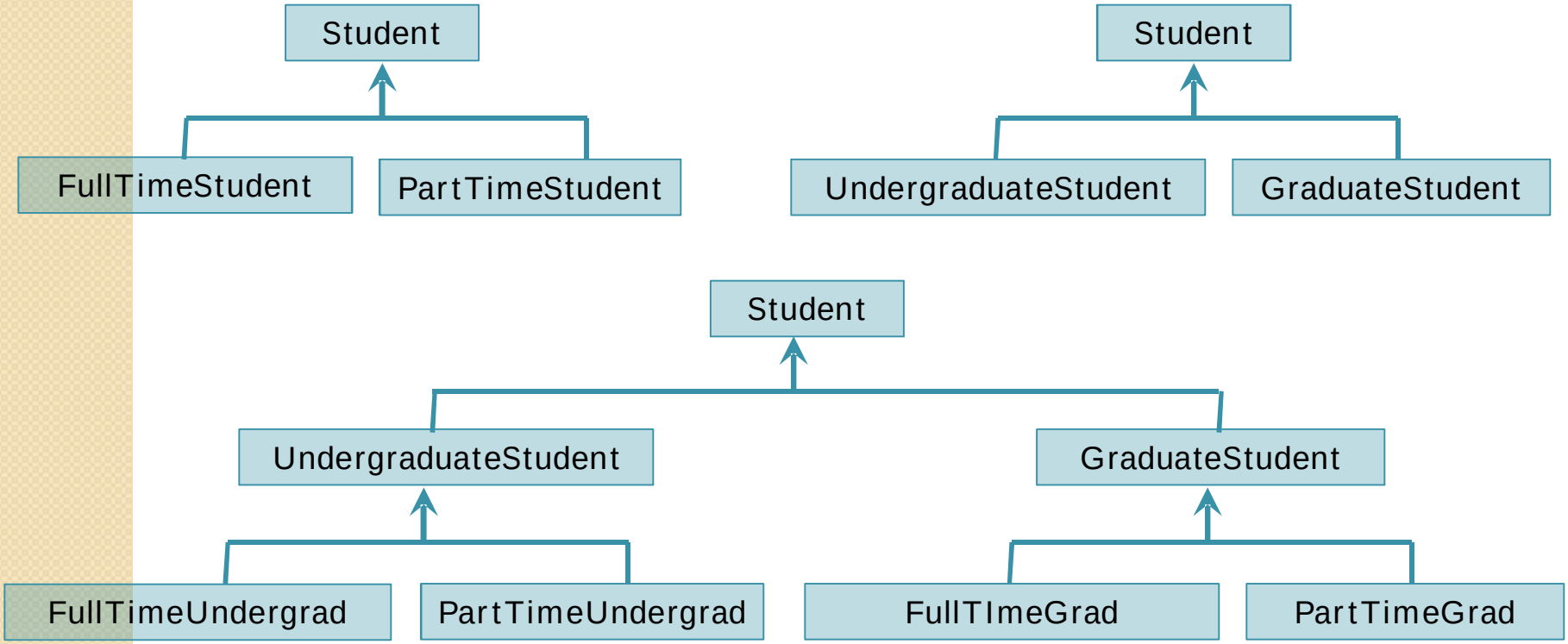
```
BankAccount b = new BankAccount();
if (b instanceof BankAccount) {
    ....
}
```

Inheritance and Abstraction (When Should We Inherit ?)

- How do we go about making our classes inherit from one another?
  - It is actually automatic. As soon as you create a class as a subclass of another class, this new class automatically inherits from it.
- So all we need to decide upon is our class hierarchy. How do we know how deep to make the class hierarchy (i.e., tree) ?
  - Most of the time, any "is a" relationship between objects should certainly result in subclassing.
- Object-oriented code usually involves a lot of small classes as opposed to a few large ones. Often, the code (i.e., hierarchies) ends up being rearranged over time. It depends on the application.

Inheritance and Abstraction (Example 1- Cont'd)

- Students in a university (may be represented many different ways … here are just 3 possibilities):



Inheritance and Abstraction (Example 1- Cont'd)

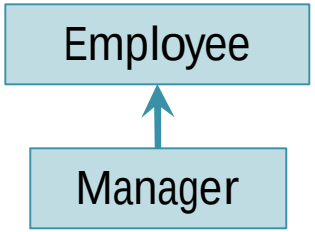
- How do we know which one to use ? It will depend on the state and behavior.
- If we find that the main differences in behavior are between full time and part time students, then we may choose the top left hierarchy (e.g., fee payments).
- If however the main differences are between grad/undergrad, (e.g., privileges, requirements, exam styles etc..) then we may choose the top right hierarchy.
- The bottom hierarchy further distinguishes between full/part time grads/undergrads.
- So ... the answer is ... we often don't know which hierarchy to pick until we thought about which hierarchy allows the maximum sharing of code.

Inheritance and Abstraction (Example 2)

- Consider making an object to represent Employees in a company which maintains: name, address, phoneNumber and employeeNumber
- We may make a single class for this.
- Now what if we want to distinguish between regular Employees and Managers ?
- We must think of what is different between these classes with respect to state and behavior.

Inheritance and Abstraction (Example 2- Cont'd)

- Managers may have:
  - additional state to represent: their responsibilities, the employees that work for them etc...
  - additional (or different) behavior
    - e.g., they may compute their salary differently
    - e.g., they may have different benefit packages
- In these situations, Managers should be represented as a special "kind of" Employee.



Inheritance and Abstraction (Example 2- Cont'd)

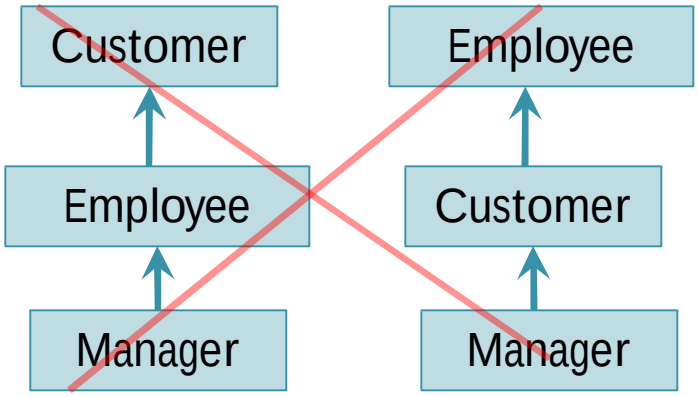
- What if we wanted to represent Customers as well ? Customers probably have state like this: name, address and phoneNumber. This state information is also used by Employees.

- We could just make Employee objects and set their employeeNumber to something like -1 to indicate that it is not an Employee, but in fact a Customer.

```
Employee jim = new Employee("Jim", "13 Elm St.", "123-4567",
762389);
Employee bill = new Employee("Bill", "13 Elm St.", " 123-4567 ", -1);
```

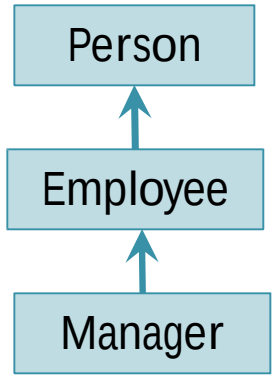
Inheritance and Abstraction (Example 2- Cont'd)

- That's a horrible idea! It actually undoes the whole concept of OOP ! Logically, we need to make:
  - a different class for Employees & Customers
  - a way of having these two classes share their state
- Neither of the following two hierarchies will work since:
  - Employees are not always Customers
  - Customers are not always Employees



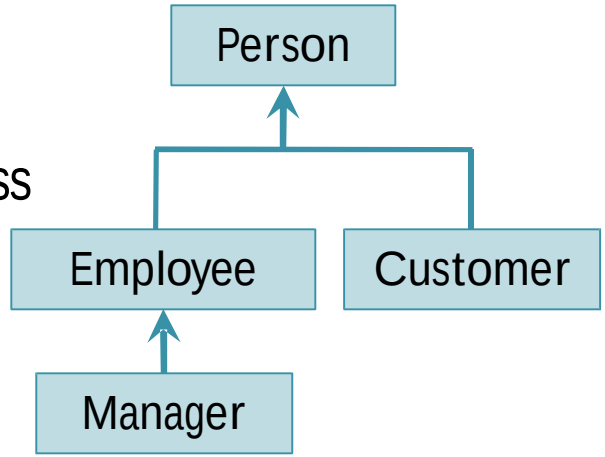
Inheritance and Abstraction (Example 2- Cont'd)

- Perhaps we can name the Customer class Person. So, Customers are just Person objects.
- This is a good solution as long as:
  - ALL state for Customers (i.e., Person objects) is also used by Employees and Managers.
  - there isn't any state or behavior in the Person class that does not apply to Employees and Managers.



Inheritance and Abstraction (Example 2- Cont'd)

- If there is indeed different behavior or state that is unique to Customers, then we must "abstract out" even further and make a Customer class.



- So in the end, the state stored in each Class probably looks something like this:
  - Person: name, address and phoneNumber
  - Employee: employeeNumber
  - Customer: itemsPurchased, and purchaseHistory
  - Manager: responsibilities, and listOfEmployees

Abstract Classes & Methods

- JAVA allows abstract classes and abstract methods.
  An Abstract Class:
  - is a class that can never be instantiated (i.e., never have instances).
  - should always have subclasses

```
new <AbstractClassName>( )
```

- A Concrete Class:
  - is a class that can be instantiated (i.e., can have instances).
  - Until now, we have been creating and using only concrete classes.

Abstract Classes & Methods (Cont.)

Why would we EVER want to create an Abstract class if we cannot make instances of it ?

- can contain common states and behaviors shared (i.e., inherited) by all its subclasses
- helps to organize a programmer
- helps keep code secure by preventing users from instantiating generalized objects. User is forced to be more specific when creating instances:
  - (e.g., you go to a pet store and ask for a Dog, a Cat or a Lizard, but not simply for a Pet).
  - (e.g., you go to a bank and ask for a SavingsAccount or a Chequeing account, but not simply for a BankAccount).

Abstract Classes & Methods (Cont.)

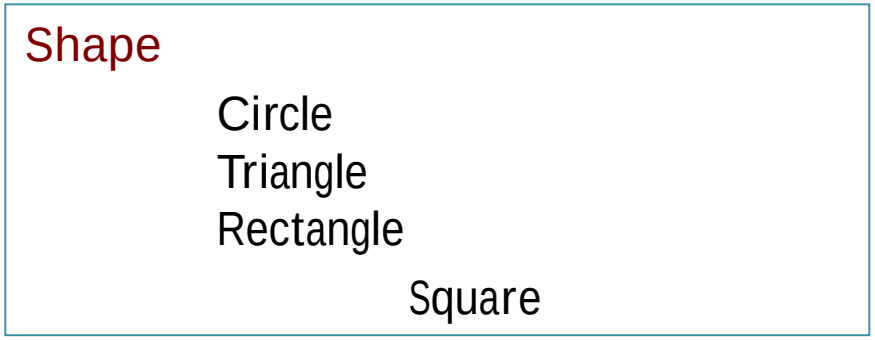
- We define an abstract class by using the abstract keyword in the class definition:

```
public abstract class BankAccount {                }
```

- This class must have subclasses in order to be useful.
  How do we know which classes to make abstract ?
  - whenever there are subclasses that cover all possible types of this object.
  - whenever this class is not specific enough to make sense in our application

Abstract Classes & Methods (Example)

- Consider the following hierarchy:



- The Shape class is most likely abstract since we probably have no instances of shape. Instead, a user may only create instances of one of the subclasses (i.e., Circle, Triangle, Rectangle or Square).

Abstract Classes & Methods (Example Cont.)

- If we only need Circles, Triangles, Rectangles and Squares in our program, then all common states & behaviors for all these classes can be placed (and hence shared) in the Shape class.
- We may need to add more shapes to this hierarchy. What about a Cube or a Sphere or even a Tetrahedron ?
  - we can merely add it as a subclass of Shape, but maybe we need to make a distinction between 2D and 3D objects. We may want to abstract out even more (this is called abstraction):

Abstract Classes & Methods (Example Cont.)

Shape

- TwoDimensionalShape
  - Circle
  - Triangle
  - Rectangle
    - Square
- ThreeDimensionalShape
  - Sphere
  - Cube
  - Tetrahedron

Abstract Classes & Methods (Example Cont.)

- Now we can put the state/behavior that is common for 2D shapes in the TwoDimensionalShape class and that common to 3D shapes in ThreeDimensionalShape.
- Note however, that we will never want to say: new Shape(), new TwoDimensionalShape() etc..., so we make these abstract.
- Perhaps we need to abstract out more later with our objects as we add more shapes. We may end up with something like this:

Abstract Classes & Methods (Example Cont.)

- Shape
  - TwoDimensionalShape
    - Ellipse
      - Circle
    - Polygon
      - Triangle
      - Rectangle
        - Square
  - ThreeDimensionalShape
    - Sphere
    - Polyhedron
      - Cube
      - Tetrahedron

Abstract Classes & Methods (Example Cont.)

- We may never need to create instances of Polygon or Polyhedron. So these may become abstract classes.
- Abstraction is the process we have just been applying to our hierarchy. It is the process of extracting the common features out of a class's subclasses.
- For instance, a polygon may keep a bunch of vertices and line segments. All Triangles, Rectangles and Squares will make use of these features and perhaps share the same drawing method.

Abstract Classes & Methods (Cont.)

- In addition to having abstract classes, JAVA allows us to make abstract methods.
- Only ABSTRACT classes can have abstract methods. However, an abstract class DOES NOT need to have only abstract methods.
- All subclasses inherit all of the "visible" (i.e., non-private) methods from their superclasses, even if the superclasses are abstract.

Abstract Classes & Methods (Cont.)

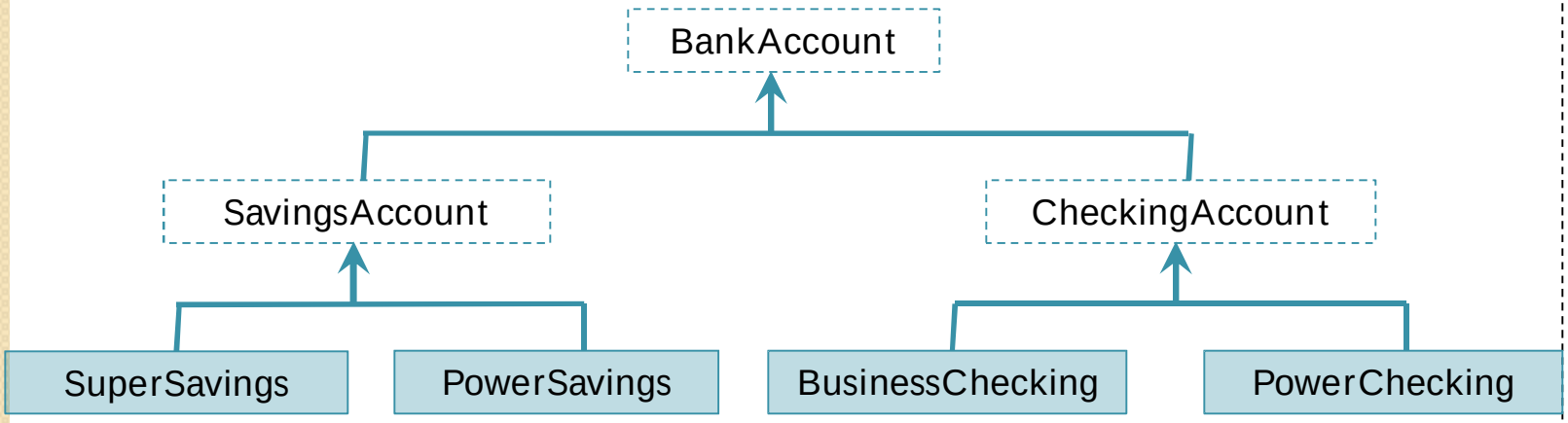
- An abstract method is a method with no code. It is defined using this format:

```
public abstract <returnType><methodName>(...);
```

- Notice the ; used at the end of the definition ... there are no { } characters.
- Abstract methods have no code ! Why would we want a method that has no code ?
  - it is a way of "forcing" the subclasses to implement specific behavior.
  - all concrete subclasses of an abstract class MUST implement the abstract methods defined in their superclasses.

Abstract Classes & Methods (Example 1 Cont.)

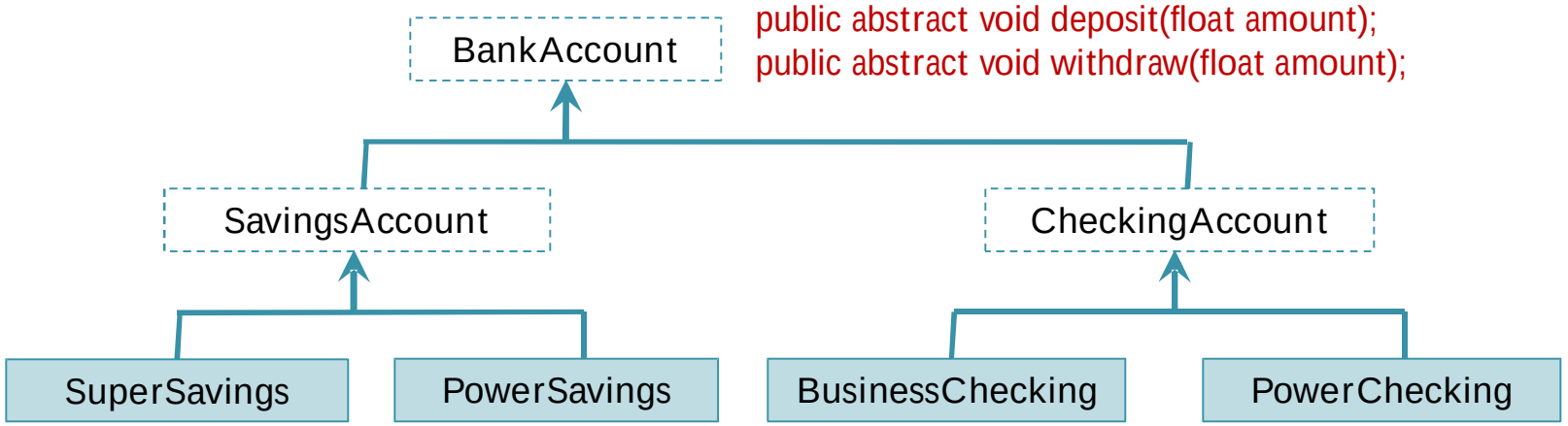
- For BankAccounts, we may wish to make BankAccount, SavingsAccount and CheckingAccount all be abstract:



- This forces the users of these classes to specify the most specific type of bank account required.

Abstract Classes & Methods (Example 1 Cont.)

- We can make deposit(float amount) and withdraw(float amount) methods abstract in the BankAccount class.
- Then, all of its concrete subclasses (SuperSavings, PowerSavings, BusinessChequing and PowerChequing) would be forced to implement them.

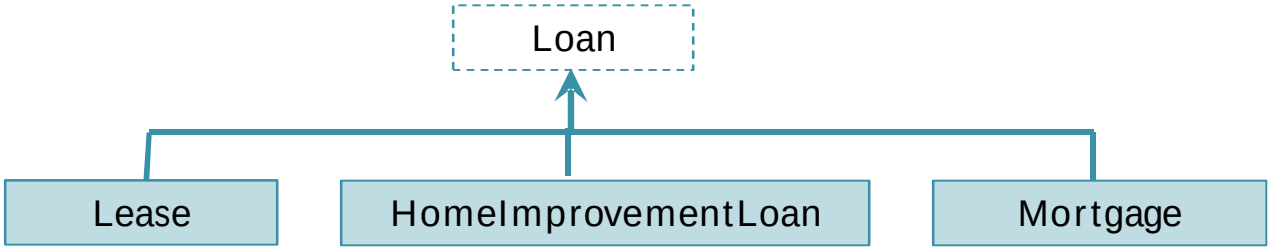


Abstract Classes & Methods (Example 1 Cont.)

- Note that abstract classes SavingsAccount and ChequingAccount do NOT NEED TO implement the abstract methods in BankAccount since they are not concrete classes.
- Alternatively, we could write non-abstract methods deposit(float amount) and withdraw(float amount) in the BankAccount class and have the subclasses either use or override them. They would thus not be forced to re-implement the methods.
- Abstract classes are often used to specify some "standard" behavior for use by its subclasses.

Abstract Classes & Methods (Example 2 Cont.)

- For example, we can make a Loan class abstract and then specify behavior that all Loan objects should have:



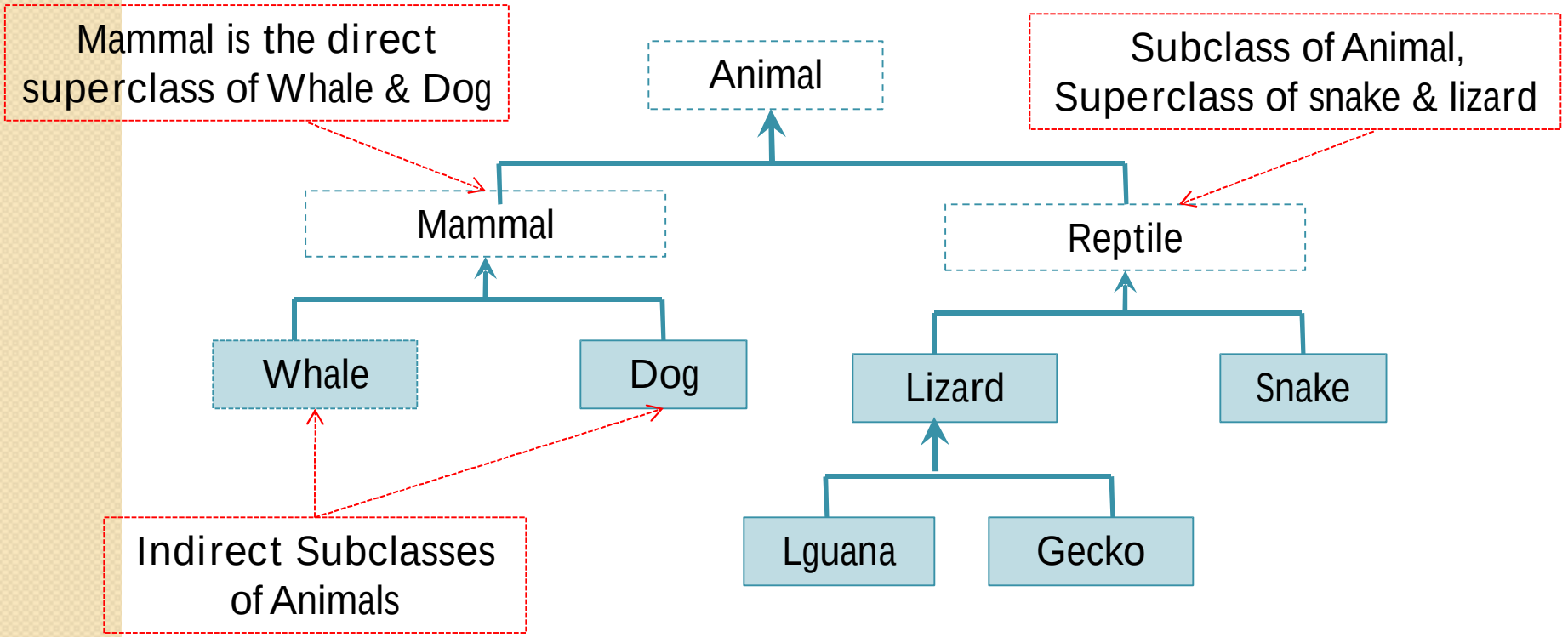
Abstract Classes & Methods (Example 2 Cont.)

- Note that all concrete subclasses of Loan MUST implement the 3 abstract methods and would inherit the non-abstract method getClientInfo().

```
public abstract class Loan {
    public abstract float calculateMonthlyPayment(); //abstract method
    public abstract void makePayment(float amount);  //abstract method
    public abstract void renew();                    //abstract method
    public Client getClientInfo() {                  //non-abstract method
        ...
    }
    ....
}
```

Abstract Classes & Methods (Example 3)

- Consider the Animal class hierarchy :



Abstract Classes & Methods (Example 3 Cont.)

- Assume that we make the Animal class abstract, with some abstract methods and some non-abstract methods as follows:

```
public abstract class Animal {
    public abstract void putInMouth(Food someFood);
    public abstract void chew(Food someFood);
    public abstract void swallow(Food someFood);
    public void eat(Food someFood) {
        putInMouth(someFood);
        chew(someFood);
        swallow(someFood);
    }
    public void sleep(int minutes) {   ....   }
    ... }
```

Abstract Classes & Methods (Example 3 Cont.)

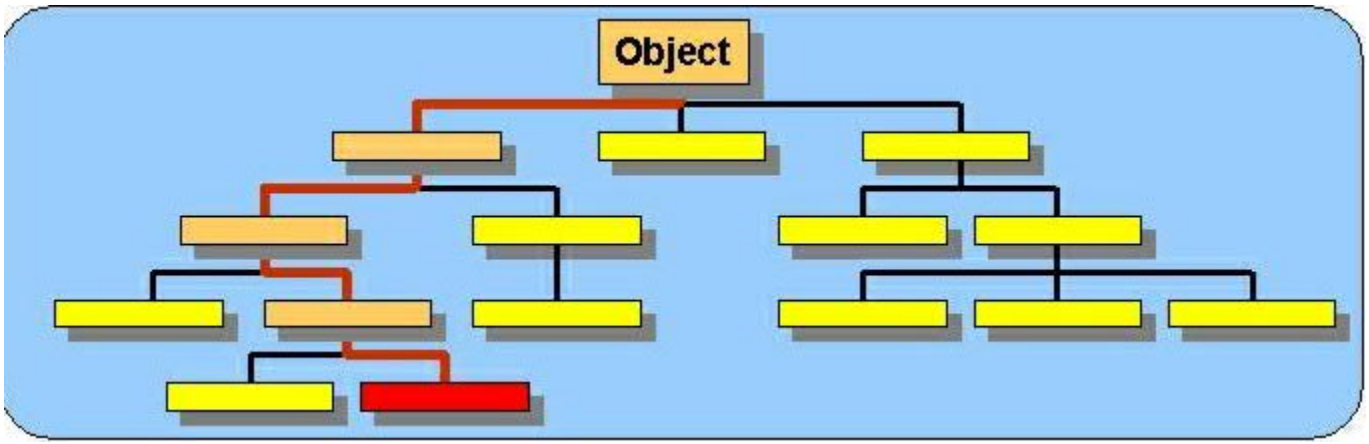
- In this code, the putInMouth(), chew(), and swallow() methods are all declared as abstract. Note that the code is not given.
- By doing this, we are forcing all subclasses of Animal to implement these methods. The compiler will complain if the methods are not implemented in the concrete subclasses.
- That means, the Reptile and Mammal classes must implement these methods. Note that some methods (such as eat()) may not be abstract.

Abstract Classes & Methods (Example 3 Cont.)

- So by writing an abstract method, we force all subclasses to adhere to some "standard". As we can see, the eat() method requires the putInMouth(), chew() and swallow() methods to be written in the subclasses. That is, we provide a guarantee that the subclasses will implement all of the required methods.

Method Lookup and Overriding

- The amount of inheritance an object receives is based on its position in the hierarchy.
- An object inherits the state and behavior from all the superclasses up the tree along the path to Object:



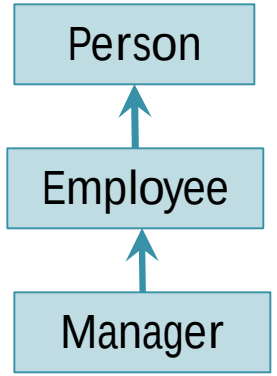
Method Lookup and Overriding (Cont.)

- Since Object is at the top of the hierarchy, every class inherits from Object.
- What kind of neat behaviors do we inherit from Object ? toString(), equals(), clone()
- To cause inheritance in our own classes, we merely make our class to be a subclass of the class we want to inherit from.

Method Lookup and Overriding (Cont.)

- Consider the inheritance in this example:

```
public class Person  {
        public String name;
        public String getName(){
            return name;}
    }
public class Employee  extends Person  {
        public int employeeNumber;
        public int getEmployeeNumber(){
            return employeeNumber;}
    }
public class Manager  extends Employee  {
        public Vector responsibilities;
        public Vector getResponsibilities(){
            return responsibilities;}
    }
```



Method Lookup and Overriding (Cont.)

- Employee inherits name and getName() from Person.
- Manager inherits name and getName() from Person as well as employeeNumber and getEmployeeNumber() from Employee.
- Both Employee and Manager objects can use the fields and methods of their superclass as if it was defined in their own class:

Method Lookup and Overriding (Cont.)

```
public class Person  {
        public String name;
        public String getName(){
            return name; }
    }
public class Employee  extends
    Person  {
        public int employeeNumber;
        public int
    getEmployeeNumber(){
            return employeeNumber; }
    }
public class Manager  extends
    Employee  {
        public Vector responsibilities;
        public Vector
    getResponsibilities(){
            return responsibilities; }
    }
```

```
Employee jim = new Employee();
jim.name = "Jim";
jim.employeeNumber = 123456;
System.out.println(jim.getName());


Manager betty = new Manager();
betty.name = "Betty";
betty.employeeNumber = 543469;
betty.responsibilites.add("Internet project");
System.out.println(betty.getName());
System.out.println(betty.getEmployeeNumber());
```

Method Lookup and Overriding (Cont.)

- A subclass cannot, however, access private methods or private fields from its superclasses:

```
public class B  {
      public int a = 10;
      private int b = 20;
      protected int c = 30;
      public int getB()  { return b; }
}
public class  A extends B  {
      public int d;
      public void tryVariables()  {
          System.out.println(a);            //access allowed
          System.out.println(b);            //access NOT allowed
          System.out.println(getB());    //can get private variable value
                                                 //through this public method
          System.out.println(c);            //access allowed
      }
}
```

Method Lookup and Overriding (Note: Cont.)

- Even though a class cannot access the `private` fields of its superclasses, it still inherits these fields as part of its own state.

- We can also use the `protected` keyword for fields:
  - `protected` fields can be accessed by a class, its subclasses and any classes in the same package. So it is a kind of "semi-private" access.

Method Lookup and Overriding (Overriding)

- As mentioned, through inheritance, a class inherits all the methods of its superclasses. That means, a subclass can use any methods of its superclasses as if they were its own.
- It may be the case that a subclass does not want to inherit some methods from its superclass. If this is the case, then the subclass can choose to ignore the superclass method by making its own method with the same name (and same parameter list) that does something else (perhaps nothing).

Method Lookup and Overriding (Overriding: Cont.)

- A method in a class is overridden by one of its subclasses if the subclass implements the method using the exact same signature.
- Overriding is used when the subclass method needs to:
  - do something different from what the superclass is doing.
  - disable the behavior since it may not be useful (or makes no sense) in the subclass:
  - do something more than what is done in the superclass.
- Lets take a look in more detail at these three scenarios ...

Method Lookup and Overriding (Overriding: Cont.)

1. Do Something Different: (e.g., consider the computation of a pay check which gets "done" every two weeks:
   - regular Employees may be paid on an hourly basis:

```
public float computePay() {
        return (regularHoursWorked * payRate) +
            (overTimeHoursWorked * 1.5f * payRate);
    }
```

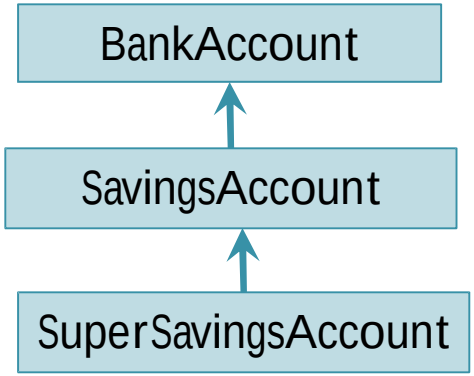
   - Managers may be paid on a salary basis:

```
public float computePay() {
        return (yearlySalary / 26f); //26 pay checks per year
    }
```

   - The Manager method overrides the one in Employee.

Method Lookup and Overriding (Overriding: Cont.)

2. Disable the Behavior: (e.g., assume that the SuperSavings account cannot be withdrawn from)
   - Write withdraw(float amount) in BankAccount class as usual.
   - Don't write withdraw(float amount) in SavingsAccount
     - allow it to be inherited from BankAccount
   - Override the method in SuperSavingsAccount:

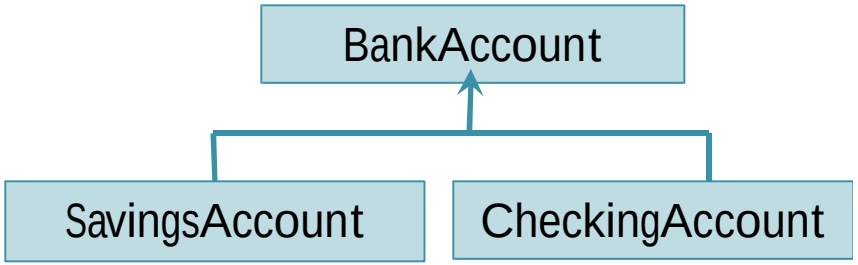



```
public void withdraw(float amount)  {
// Do Nothing
}
```

Method Lookup and Overriding (Overriding: Cont.)

3. Do More: (e.g., Assume that withdraws from a checking account have a surcharge fee of $0.75).
   - Write method withdraw(float amount) in BankAccount as before
     - inherited by SavingsAccount.
   - Override this method in CheckingAccount to make use of superclass one:

```
public void withdraw(float amount)  {
super.withdraw(amount + 0.75f);
}
```



   - (Note: The keyword super is used to indicate that you would like to use the superclass method instead of the one in this class)

Method Lookup and Overriding (Overriding: Cont.)

- The final/protected keywords:
  - If a method is declared as final, it CANNOT be overridden, the compiler will stop you:

```
public final void withdraw(float amount) {
    ...
}
```

  - Why would we want to do this ? Perhaps the behavior defined in the method is very critical and overriding this behavior improperly may cause problems with the rest of the program.
  - If a class is declared as final, it CANNOT have subclasses:

```
public final class Manager {
    ...
}
```

Method Lookup and Overriding (Overriding: Cont.)

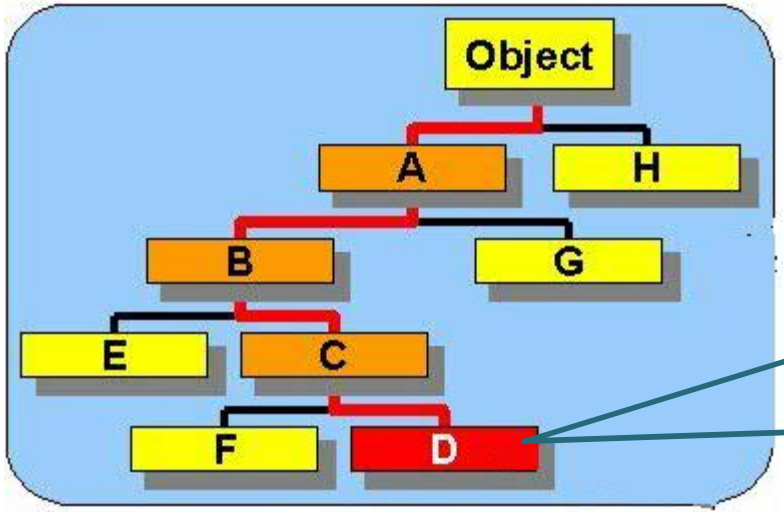
- Why would we want to do this ? Maybe because it is too complicated and may easily be misused.
- Many of the JAVA classes (e.g., Vector) are declared final which means that we cannot make a subclass of them. ... its a kind of security issue to prevent us from "messing up" the design and intentional usage of those classes.

Method Lookup and Overriding (Method Look-Up)

- When a message (method) is sent to an instance, JAVA must look up the method in the class hierarchy and then evaluate it.
- It is important to understand how JAVA "looks up" methods when we send messages to objects.
- Assume that we send a message `methodTesting()` to an instance of class D:

Method Lookup and Overriding (Method Look-Up: Cont.)

- Method lookup for all instance methods works as follows:
  - If a method methodTesting() exists in class D, then it is evaluated.
  - Otherwise, JAVA checks the superclass of D for the method (in this case class C).
  - If not found there, JAVA continues looking up the hierarchy until Object is reached:



Call to methodTesting() appears here:

```
Public void x( ){
this. methodTesting() ;
//or just methodTesting() ;
...
}
```

Lookup begins here in class D.

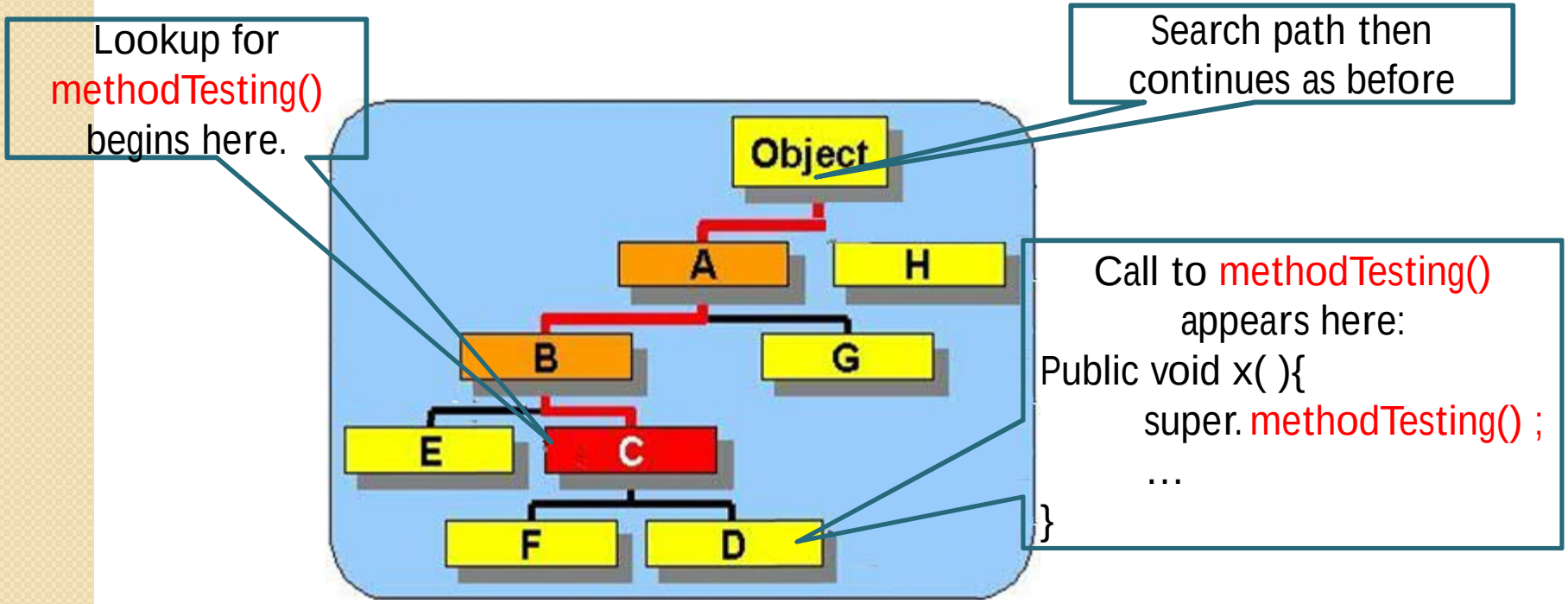
Method Lookup and Overriding (Method Look-Up: Cont.)

- If not found at all during this search up to the Object class, the compiler will catch this and inform you that it can't find the method methodTesting() for the object you are trying to sending it to:

  C:\Test.java:20: cannot resolve symbol: method methodTesting()

- The use of super merely specifies where the method lookup should begin the search.
  If for example, we are in a method belonging to class D that calls super. methodTesting() ,
  the lookup for methodTesting() will start in class C first:

Method Lookup and Overriding (Method Look-Up: Cont.)



Method Lookup and Overriding (Method Look-Up: Cont.)

- The super keyword is also used in constructors to call a constructor in the superclass.
- JAVA automatically calls the superclass' default constructor if you do not specify a call yourself.
- If you do want to call a superclass constructor, it MUST be the first line of your constructor code.

```
public BankAccount(String own, float bal) {
        owner = own;
        balance = bal;
    }
public SavingsAccount(String own){
     super(own, 0.0f); //Calls above constructor, otherwise the default
                       //constructor in BankAccount would have been called.
}
```

Method Lookup and Overriding (Method Look-Up: Cont.)

- JAVA does lookup for static methods differently
  - static method lookup is done at compile time.
  - this becomes important when type-casting (more later).

Type-Casting

- We have already seen type-casting with primitives. Type-casting with objects is a little more complicated.
- Many classes in JAVA make use of automatic type-casting, so we must understand:
  - how to type-cast
  - when it is done automatically
  - the rules of type-casting
- For primitives, type-casting is the process of converting a value from one form to another:

```
(int)871.34354;   // results in 871
(char)65;         // results in 'A'
(long)453;        // results in 453L
```

Type-Casting (Cont'd)

- Type-casting of objects is done using a similar syntax (i.e, with the round brackets). Here are a couple of examples:

```
Customer      c = (Customer)anArrayList.get(i);
SavingsAccount b = (SavingsAccount)aBankAccount;
```

- For objects, type casting does not convert, but merely causes the object to be "treated" more generally.
- When objects are type-casted:
  - they DO NOT change their type!
  - they are ONLY "treated" differently by the compiler.
  - they usually have less methods available to them.

– One of the main advantages of type-casting is that it allows for polymorphism.

Polymorphism

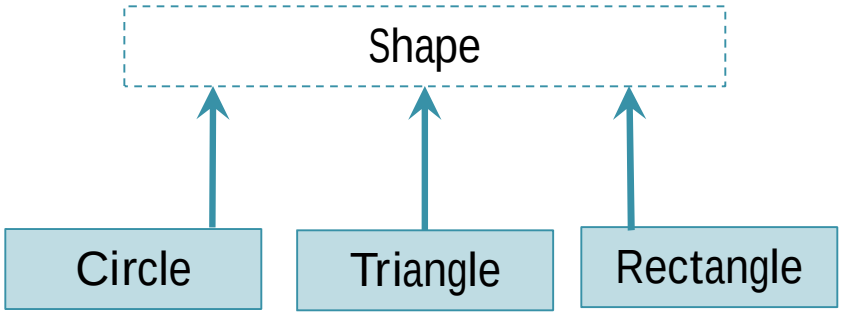
- Polymorphism is the ability to use the same behavior for objects of different types.
- That is, it allows different objects to respond to the exact "same" message.
- We have already seen some cases of polymorphism:
  - all Objects understand the toString() message
  - all Collections (e.g., ArrayList, Vector) understand the add() message
  - all BankAccounts understand the deposit() message

Polymorphism (Cont'd)

- A BIG advantage of polymorphism is that by treating objects more generally (i.e., type-casting to superclasses) we just need to understand a few commonly understood messages that all these objects understand. For example, we can:
  - ask all Person objects what their name is. This is independent as to whether or not they are instances of Employees, Managers, Customers etc...
  - deposit to any BankAccount, independent of its type.
- Polymorphism is the main motivation for type-casting objects. This is because all objects that can be type-casted to the same type will be able to respond to the same messages. The code is therefore shorter and easier to understand.

Polymorphism (Cont'd)

- Consider the following hierarchy:



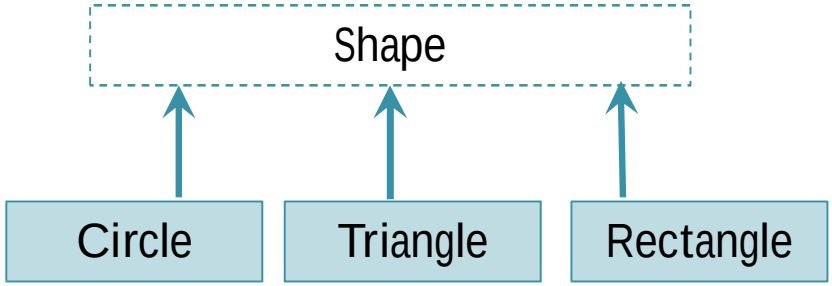
- We can store less accurate primitives (e.g., int) into a variable of more precision (e.g., float):

```
int   amount = 43;
float total = amount;
```

Polymorphism (Cont'd)

- Well when dealing with Objects, we can store more specific objects in variables of a more general type. That is, we can store instances of a class into variables of a type of one of its superclasses:

```
Triangle  t = new Triangle();
Shape     s = (Shape)t;
Object    obj1 = (Object)t;
Object    obj2 = (Object)s;
```



- Note that the explicit type-cast is not necessary, it is automatic if left out:

```
Triangle  t = new Triangle();
Shape     s = t;
Object    obj1 = t;
Object    obj2 = s;
```

Polymorphism (Cont'd)

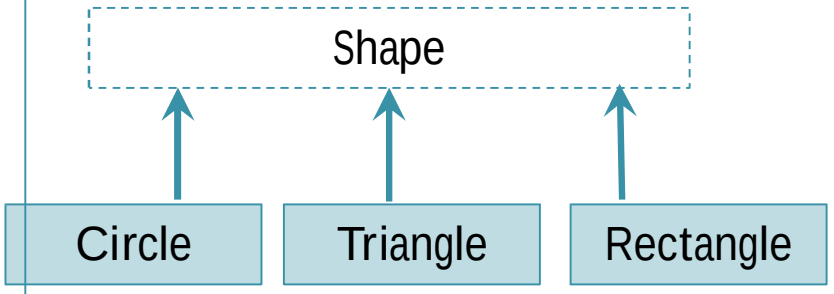
- Of course, typecasting to classes that are not superclasses of the original object is not allowed:

```
Rectangle  r = new Triangle();                  // Not allowed
Rectangle  r = (Rectangle) new Triangle();      // Not allowed
Circle     c = new Shape();                     // Not allowed
Circle     c = (Shape) new Shape();             // Not allowed
```

- Once type-casted to something else, we can always type-cast "back to the original class" again:

```
Triangle  t;
Shape     s;

t = new Triangle();
s = (Shape)  t;
t = (Triangle) s;
```



Polymorphism (Cont'd)

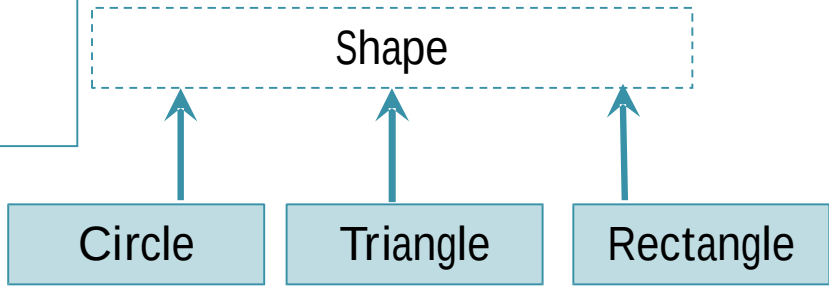
- Assume that we are given a variable defined as follows:

```
Shape   aShape;
```

- This variable may hold instances of Shape, Circle, Triangle or Rectangle.
- Assume now that we want to draw the shape or perhaps ask how many sides it has.
- However, we may not know exactly which kind of shape is currently stored in the aShape variable, so we would have to make a check. Perhaps a bunch of if statements could be used:

Polymorphism (Cont'd)

```
aString = aShape.getClass().getName();
if (aString.equals("Circle"))
    aShape.drawCircle();
if (aString.equals("Triangle"))
    aShape.drawTriangle();
if (aString.equals("Rectangle"))
    aShape.drawRectangle();
```



- or even better:

```
if (aShape instanceof Circle)
    aShape.drawCircle();
if (aShape instanceof Triangle)
    aShape.drawTriangle();
if (aShape instanceof Rectangle)
    aShape.drawRectangle();
```

Polymorphism (Cont'd)

- The shape must be drawn differently according to the type of shape it actually is. In the code above, we have three different methods for drawing the shape and we are calling the appropriate one according to the type of shape.
- There is a better way to write the code. We could name the drawing method "draw()" for each of the three shape classes. Having the same method name (and signature) for different multiple classes is an example of Polymorphism.

Polymorphism (Cont'd)

- Why do we want this ? Well, replace the method names in the code:

```
if (aShape instanceof Circle)
    aShape.draw();
if (aShape instanceof Triangle)
    aShape.draw();
if (aShape instanceof Rectangle)
    aShape.draw();
```

- Clearly, we do not even need the if statements anymore since we are calling the same method despite the type of shape. We can just write:

```
aShape.draw( );
```

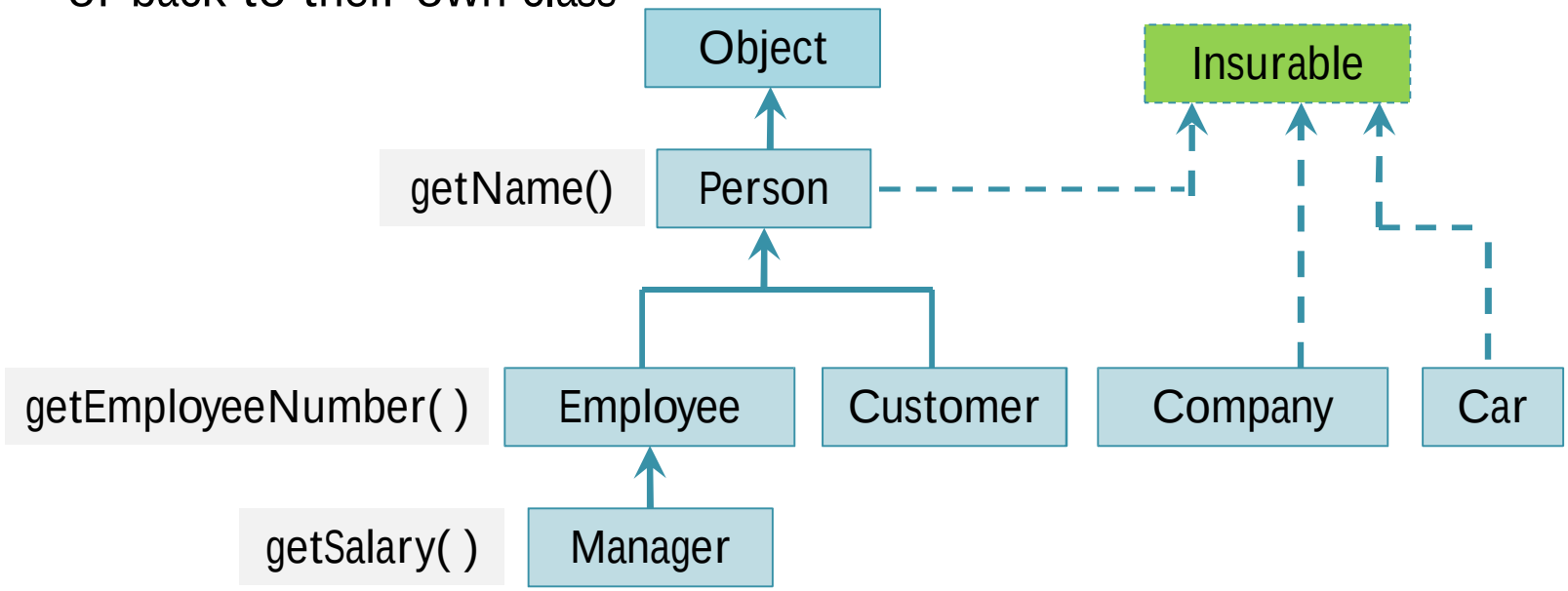
This works because all different shapes are subclasses of the Shape class

Polymorphism (Cont'd)

- There are many advantages to using polymorphism:
  - The amount of code that needs to be written is reduced
  - The code is easier to understand
  - It helps the programmer remember available functionality
- We have seen polymorphism already. Here are some examples:
  - toString()
  - size()
  - isEmpty()
- Many objects have these methods.

Polymorphism (Cont'd)

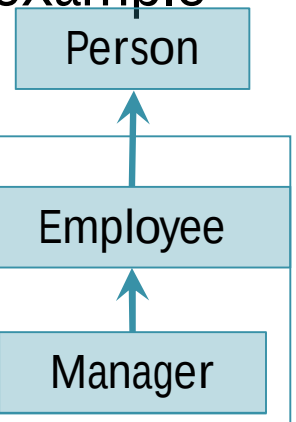
- There are certain rules for type-casting objects. Objects may ONLY be type-casted to:
  - a type which is ANY one of its superclasses
  - an interface which the class implements (more on interfaces soon)
  - or back to their own class



- Attempts to type-cast to anything else will generate a ClassCastException.

Polymorphism (Cont'd)

- Managers may be type-casted to Employee, Person or Object but not to Customer, Company or Car.
- Some of these restrictions make sense, after all, why would we treat aManager as a Company or a Car ? Here is an example of some type-casting:




```
Manager  man = new Manager();
man.getName();
man.getEmployeeNumber();
man.getSalary();

Employee  emp = (Employee)man;  // emp is actually pointing to a Manager object
emp.getName();                 // can use this method since it is also defined in Person
emp.getEmployeeNumber();       // can use this method since it is also defined for Employee
emp.getSalary();               // COMPILE ERROR: emp cannot respond to Manager methods anymore
```

Polymorphism (Cont'd)

- Recall that explicit type-casting is not always required. An object may be automatically type-casted when it:
  - is assigned to a variable

```
Manager   man;
Employee  emp;

man = new Manager();
emp = man;      // automatic ... same as saying emp = (Employee)man;
```

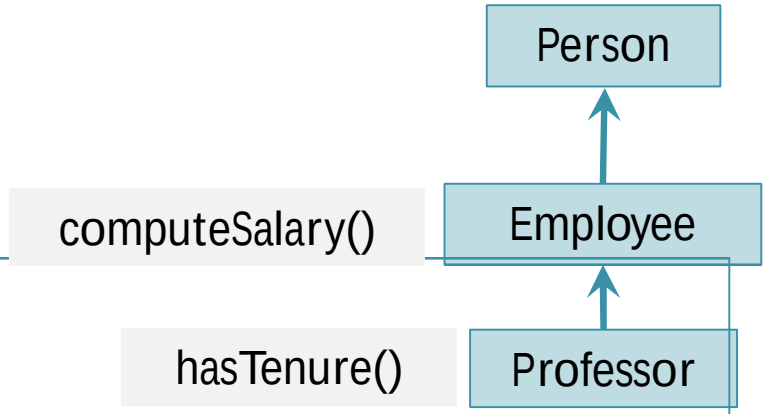
  - is sent as a parameter to a method

```
Manager man = new Manager();
aCompany.doStandardHiringProcess(man);//Passed in as a Manager object
public void doStandardHiringProcess(Employee emp) {
   ...//emp is type-casted to Employee upon entering this method.
```

- REMEMBER: JAVA ALWAYS knows the "original" type of every object ... and this NEVER changes.

Polymorphism (Cont'd)

- IMPORTANT: You are only allowed to use methods which are AVAILABLE in the class being type-casted to. However, when methods are called, lookup begins in the original class.
- Here is another example:




```
Professor   prof;
Employee    anEmployee;

prof = new Professor();
anEmployee = (Employee)prof;   //From now on, treat the Professor as an Employee

anEmployee.computeSalary();    //This will work
anEmployee.hasTenure();  //This won't work since hasTenure() is only defined for
                                 // Professors
```

Polymorphism (Cont'd)

- Note that we have type-casted the Professor to be an Employee object. From that point onwards, we may only send Employee methods to the object.
- Make sure to keep in mind that type-casting does NOT do any kind of conversion. In fact, the anEmployee variable in the above example actually points to a Professor object.
- That is, the type casting does not change the fact that the object is a Professor object. The type-casting process for objects merely informs the compiler that from now on, the object is to be "treated as" an Employee.
- In fact, we can always type-cast back again:

Polymorphism (Cont'd)

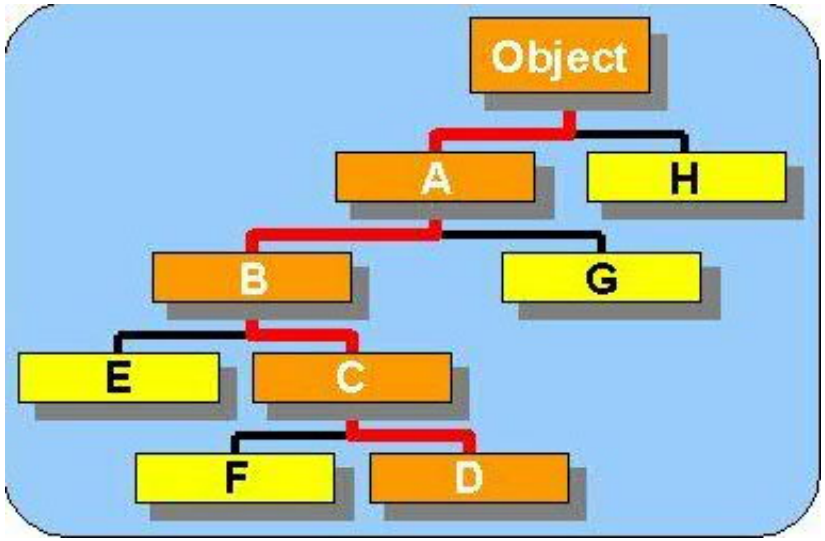
```
Professor  aProf;
aProf = (Professor)anEmployee; //From now on, treat the object again as a professor
aProf.computeSalary();        //This will still work
aProf.hasTenure();            //Works now
```

- So...an object may be type-casted back again at any time.
  Look again at the Manager example:

```
Manager   man = new Manager();
Employee  emp = (Employee)man;
emp.getName();
emp.getEmployeeNumber();
((Manager)emp).getSalary();
```

Polymorphism (Cont'd)

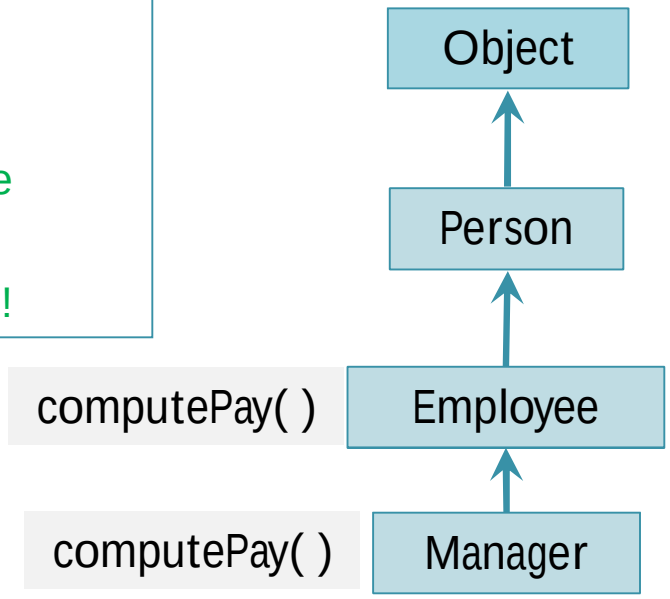
- Type-casting may occur up and down the hierarchy along the path from the original type of the object to the Object class:



Polymorphism (Cont'd)

- How does method lookup occur for type-casted objects ?
- For instance methods, method lookup ALWAYS begins in the class of the original object. Lookup then proceeds as normal up the hierarchy.

```
Manager   man = new Manager();
Employee  emp1 = new Employee();
Employee  emp2 = (Employee)man;
emp1.computePay(); //uses method in Employee
man.computePay();  //uses method in Manager
emp2.computePay(); //uses method in Manager!!
```



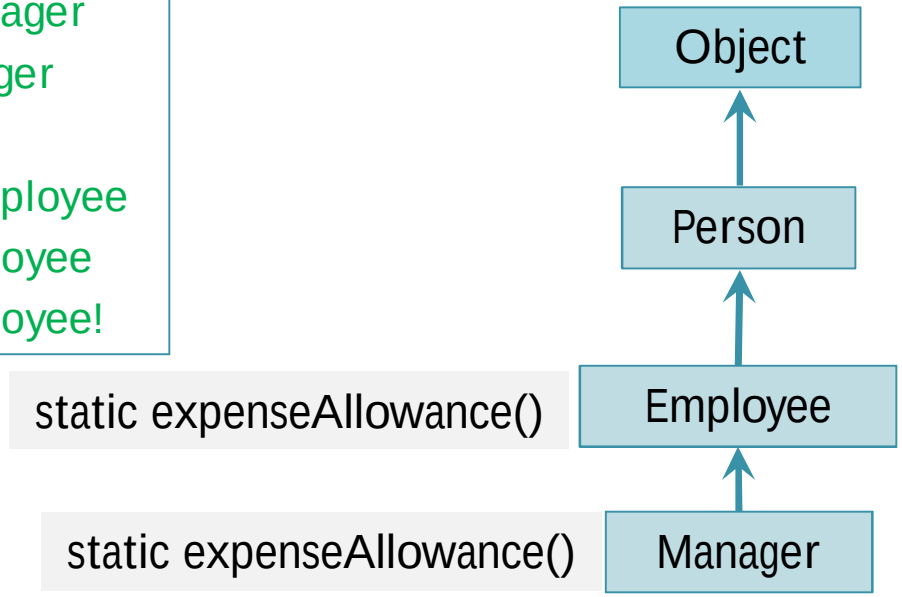
Polymorphism (Cont'd)

- For static methods, lookup is done at compile time, and so JAVA always begins in the class of the defined variable.

```
Manager    man = new Manager();
Employee  emp1 = new Employee();
Employee  emp2 = (Employee)man;

Manager.expenseAllowance();  //in Manager
man.expenseAllowance();      //in Manager

Employee.expenseAllowance(); //in Employee
emp1.expenseAllowance();     //in Employee
emp2.expenseAllowance();     //in Employee!
```



Polymorphism (Cont'd)

- Many JAVA methods work with arbitrary objects. (e.g., ArrayLists can hold various kinds of objects.)
- Many of these methods take Objects or return them as parameters:

```
public boolean equals(Object obj) {...}
public void add(Object obj) {...}
public Object get(int i) {...}
```

- In these cases, any object can be passed into the method (or returned from it) and the object is automatically type-casted to type Object.
- Thus, for Object parameters, only messages that Object understands can be sent to the incoming obj when inside the method.

Polymorphism (Cont'd)

- As we have already seen, type-casting was also necessary when extracting items from a Collection:

```
ArrayList employees = getEmployees();
for (int i=0; i<employees.size(); i++) {
  Employee emp = (Employee)employees.get(i);
  emp.doSomething();
}
```

- We must type-cast here because both of these are true:
  - the get() method of ArrayList always returns type Object, and
  - we want to do something "Employee-specific" with this returned Object.

Polymorphism (Cont'd)

- we don't have to type-cast it if we are just sending Object-specific messages to it:

```
ArrayList employees = getEmployees();
for (int i=0; i<employees.size(); i++) {
    System.out.println(employees.get(i));
}
```

- we don't have to type-cast it if we want to treat it as a general Object:

```
ArrayList result = new ArrayList();
for (int i=0; i<employees.size(); i++) {
    result.add(employees.get(i));
}
```

Polymorphism (Cont'd)

- Objects define the following commonly used methods:

```
public Class getClass();
public String toString();
public boolean equals(Object obj);
public void clone();
public int hashCode();
```

- Every object inherits these methods from object. The last 4 of these are often overridden.

Polymorphism (Cont'd)

- Remember though that since JAVA 1.5, there is now an automatic type-cast when extracting items from ArrayLists as long as we declare the type when defining the ArrayList. In this case, the type-casting is hidden from the programmer.

```
ArrayList<Employee>  employees = getEmployees();
ArrayList<Object>    result = new ArrayList<Object>();
for (Employee emp: employees) {  // items automatically type-casted to Employee
  emp.doSomething();
  System.out.println(emp);
  result.add(emp); // emp automatically type-casted to Object
}
```

Polymorphism (Cont'd)

- Example 1: Typecasting PersonTester

- Note: The lifespan() method did not work the way that we expected. That's because for class methods, method look-up occurs at compile time.
- The lifeSpan() method in the Person class is used by both the Boy and Person classes. In this case, since the method is static and declared in the Person class, the ageFactor from the Person class is used.
- However, the Girl class has its own lifeSpan() method, so the ageFactor within the Girl class is used in that case.

Double-Dispatching

- Let us look back again at our shape-drawing example. Consider now a Pen object which is capable of drawing shapes. We would like to use code that looks something like this:

```
aPen.draw(aCircle);
aPen.draw(aTriangle);
aPen.draw(aRectangle);
```

Double-Dispatching (Cont'd)

- However, this is not so straight forward. We would have to define a draw method in the Pen class for each kind of shape in order to satisfy the compiler with regards to the type of the parameter:

```
public void draw(Circle aCircle) {
    // Do the drawing of the given Circle
}
public void draw(Triangle aTriangle) {
    // Do the drawing of the given Triangle
}
public void draw(Rectangle aRectangle) {
    // Do the drawing of the given Rectangle
}
```

Double-Dispatching (Cont'd)

- Well, since the drawing code is likely different for all 3 shapes ... then don't we need 3 different methods anyway ?
- YES. However, the way the code is arranged, we have to write all the draw methods in the Pen class.
- So, if we later want to add some more Shapes (such as polygons, ellipses, parallelograms etc...), we would have to go into the Pen class and add a draw method for each Shape.
- This is BAD ... because the code in the Pen class becomes highly dependent on the different kinds of Shape objects.
- We would like a way of separating the Pen class completely, so that we don't have to come back to it later and add more draw methods.

Double-Dispatching (Cont'd)

- In fact, we can indeed declare a single method as long as all the Shapes inherit from a common superclass (or implement a common interface as we'll see later). We are fortunate here since all of our shapes are subclasses of the Shape class. So, we can write one method that takes a Shape parameter:

```
public void draw(Shape anyShape) {
    // Do the drawing
}
```

- This is actually a VERY powerful concept. It prevents us from having to write a method for every kind of parameter object, since we can pass in any object that is a subclass of Shape !! Now we simply write one draw method that handles all shapes :).

Double-Dispatching (Cont'd)

- Wait a minute! We still have a minor problem though. What does the code look like inside the method ?

```
public void draw(Shape anyShape) {
if (anyShape instanceof Circle)
   // Do the drawing for circles
if (anyShape instanceof Triangle)
   // Do the drawing for triangles
if (anyShape instanceof Rectangle)
   // Do the drawing for rectangles
}
```

Double-Dispatching (Cont'd)

- That is, we seem to still have to decide how to draw the different Shapes.
- So then when new Shapes are added, we still need to come into the Pen class and make changes :(.
- We can correct this problem by shifting the drawing responsibility to the Shapes themselves, as opposed to it being the Pen's responsibility.
- This "shifting" (or flipping) of responsibility is called double dispatching (the name implies that a message is "dispatched" (or sent) twice in order for the behavior to be completed).

Double-Dispatching (Cont'd)

- We perform double-dispatching by making a method in each of the specific Shape classes that allows the shape to draw itself using a given Pen object:

```
//This method should be in the Circle class
public void drawWith(Pen  aPen) {
    // Do the drawing with the given pen
}
//This method should be in the Triangle class
public void drawWith(Pen  aPen) {
    // Do the drawing with the given pen
}
//This method should be in the Rectangle class
public void drawWith(Pen  aPen) {
    // Do the drawing with the given pen
}
```

Double-Dispatching (Cont'd)

- Now, we reduce the code in the Pen class to the following:

```
public void draw(Shape anyShape) {
    anyShape.drawWith(this);
}
```

- So ... when we ask the Pen to draw the Shape, it is as if it says: "No way! let the shape draw itself using me". This is a very common technique which is used in object-oriented programming as it helps to reduce and simplify the code. Polymorphism makes it all work.
- NOTE: In order for this to compile, you must have a drawWith(Pen p) method declared in class Shape since it is declared as abstract.

Interfaces

- In addition to polymorphism, there is another concept that allows the programmer to simplify code and to remember class functionality. This is called an interface.

- An Interface:
  - is simply a specification (i.e., a list) of a set of behaviors
    - specifies ONLY behavior, no state.
  - represents a "stamp of approval" to all classes that implement it
    - provides a guarantee that any class that implements this interface will implement (provide) these methods.
  - like classes, they are types and are defined in their own .java files

Interfaces (Cont'd)

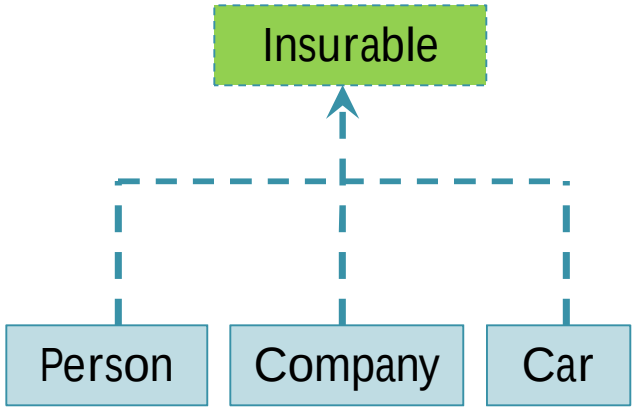
- We use the interface keyword instead of the class keyword when defining an interface and then we simply write out all the method signatures:

```
public interface interfaceName {
    <modifiers> <returnType1> <method1Name> (<t1> <name1>, <t2> <name2>,...<tn> <nameN>);
    <modifiers> <returnType2> <method2Name> (<t2> <name1>, <t2> <name2>,...<tn> <nameN>);
    ...
    <modifiers> <returnTypeN> <methodNName> (<t3> <name1>, <t2> <name2>,...<tn> <nameN>);
}
```

- Defining an interface, involves determining the common behavior for all implementers of the interface.

Interfaces (Cont'd)

- Consider, for example, an insurance company application that defines an Insurable interface and has different kinds of objects implement the interface.
  - Insurable behavior: get the policy number, get amount of coverage, calculate the premium, get the expire date
  - Here is an example of the code that may be defined within an Insurable interface:




```
public interface Insurable {
    public int getPolicyNumber();
    public int getCoverageAmount();
    public double calculatePremium();
    public Date getExpiryDate();
}
```

Interfaces (Cont'd)

- Unlike classes, you cannot make instances of an interface.

```
new <InterfaceName>( );
```

- Instead, once the interface is written, saved and compiled, classes may then implement the interface by specifying the implements keyword in their definition:

```
public class className implements interfaceName {
    // Implementation of the interface's methods
    // Own data and methods
}
```

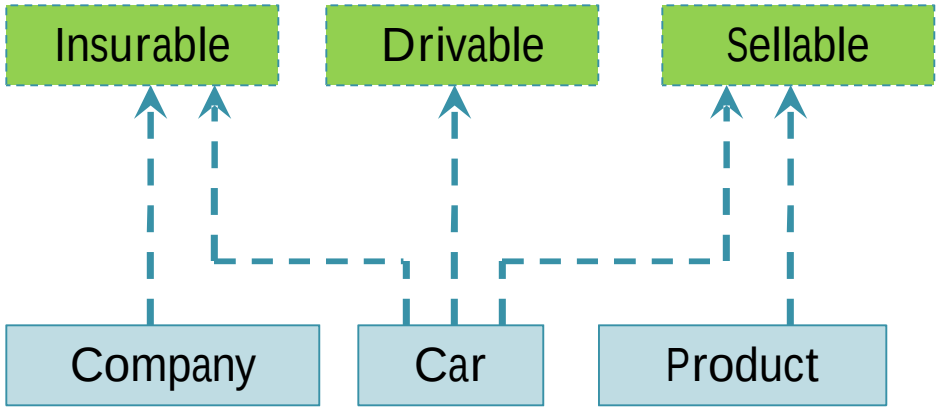
Interfaces (Cont'd)

- Classes that implement the interface must implement ALL of the interface methods:

```
public class Car implements Insurable {
    public int getPolicyNumber() {
        // write code here
    }
    public double calculatePremium() {
        // write code here
    }
    public Date getExpiryDate() {
        // write code here
    }
    public int getCoverageAmount() {
        // write code here
    }
}
```

Interfaces (Cont'd)

- Classes may implement more than one interface:



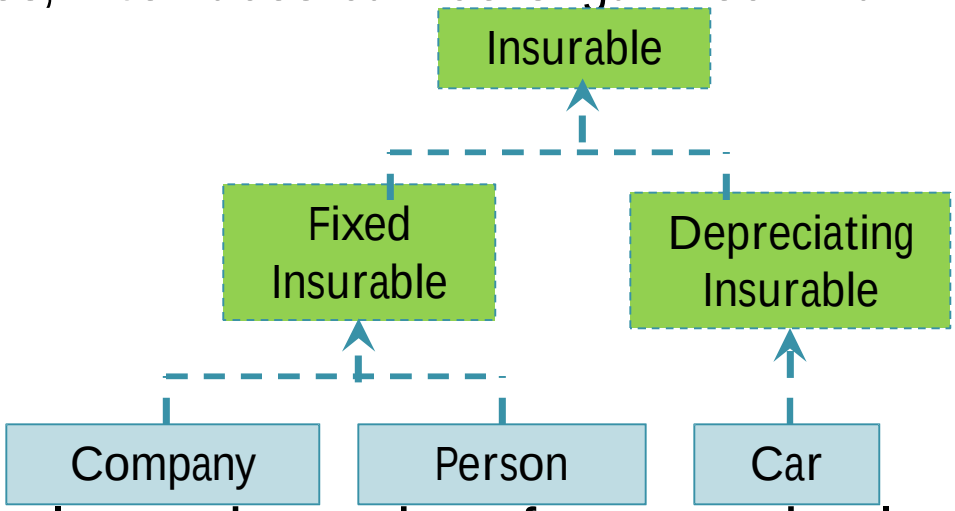
- To allow this in your code, just specify each implemented interface in your class definition:

```
public class Car implements Insurable, Drivable,
Sellable {
   ...
}
```

- Here, Car would have to implement ALL of the methods defined in each of the three interfaces.

Interfaces (Cont'd)

- Like classes, interfaces can be organized in a hierarchy:

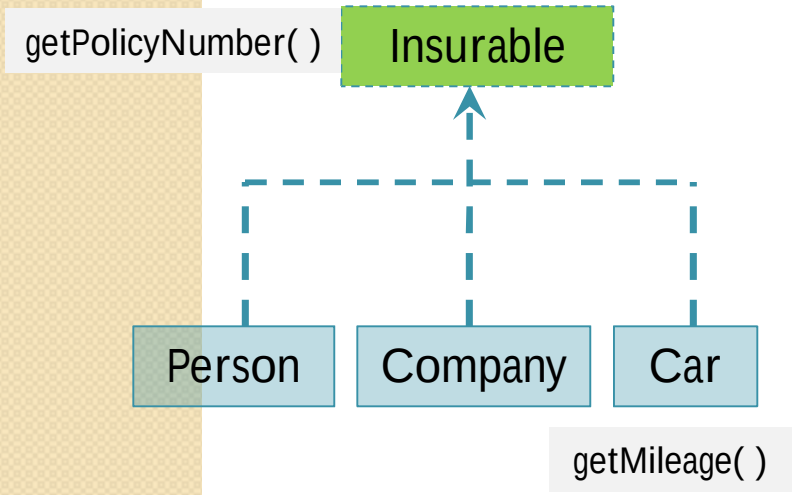


- Classes implementing an interface must implement its "super" interfaces as well.

```
public interface DepreciatingInsurable extends Insurable{
    public double computeFairMarketValue();
}
public interface FixedInsurable extends Insurable {
    public int getEvaluationPeriod();
}
```

Interfaces (Cont'd)

- Objects can also be type-casted to an interface type, provided that the class implements that interface.




```
Car       jetta = new Car();
Insurable  item = (Insurable)jetta;
// We can now only send Insurable messages to item
item.getPolicyNumber();
item.calculatePremium();

jetta.getMileage();       // This is OK
item.getMileage();        // This is NOT OK !!
((Car)item).getMileage(); // This is OK too
```

- Notice that defining an interface is similar to defining an abstract class which has no state and only abstract methods.

Interfaces (Cont'd)

- Recall the abstract methods in the Animal class:

```
public abstract class Animal {
    public abstract void putInMouth(Food someFood);
    public abstract void chew(Food someFood);
    public abstract void swallow(Food someFood);
    public void sleep(int minutes) { ... }
    public void eat(Food someFood) { ... }
    ...
}
```

- Similarly, we could have defined the following interface:

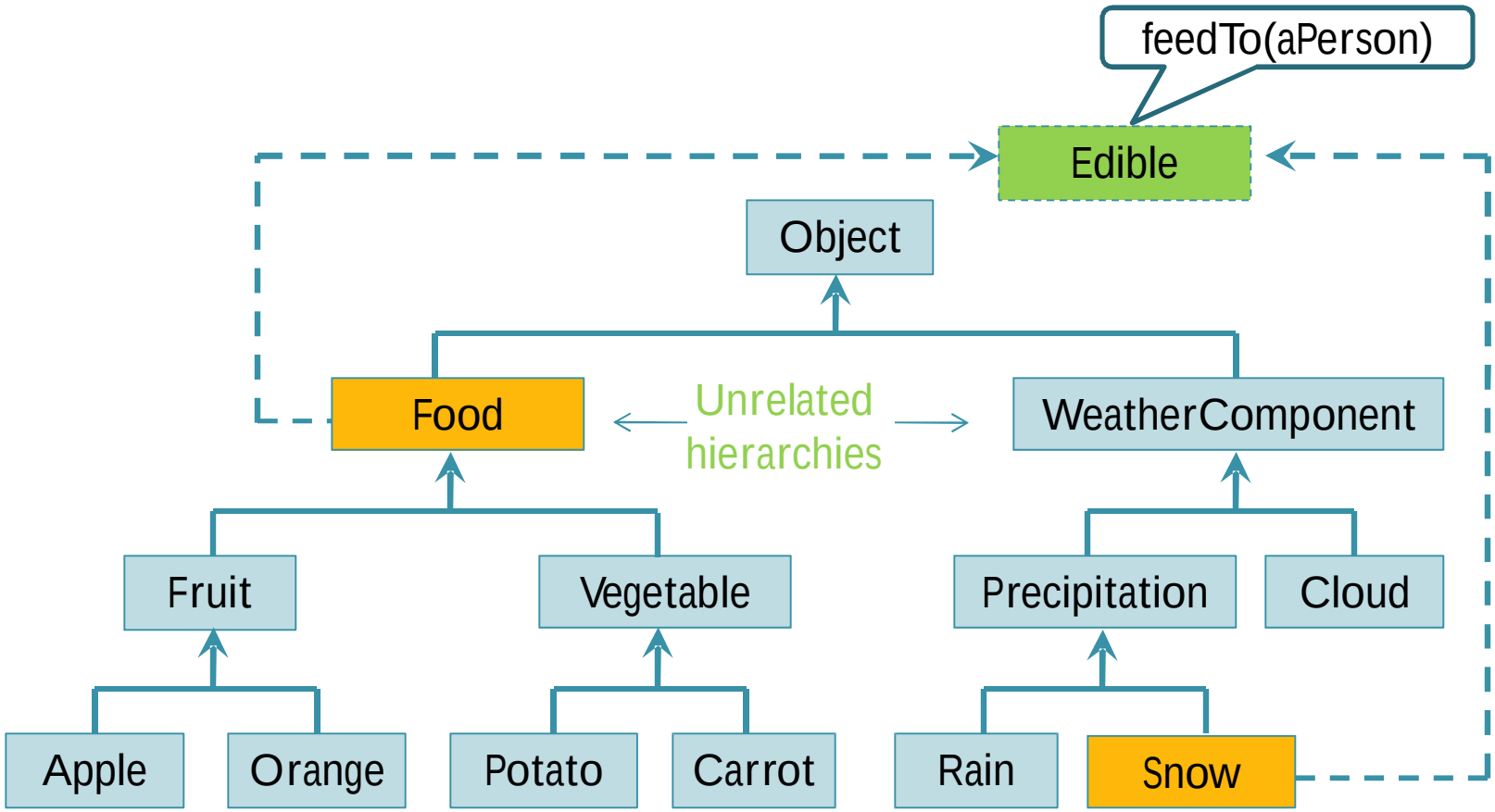
```
public interface AnimalInterface {
    public void putInMouth(Food someFood);
    public void chew(Food someFood);
    public void swallow(Food someFood);
}
```

- If we then made the subclasses of Animal implement this interface, then this is another way of guaranteeing that they will have the required behavior.

Interfaces (Cont'd)

- How are the two strategies similar ? Both provide a guarantee that the methods will be implemented !
- So which way should we write our code ... using abstract classes with abstract methods or using interfaces instead ?
- Interfaces are more flexible in that any class can implement the behavior whereas with abstract classes, we only get to specify this common behavior for classes that lie beneath Animal in the hierarchy.
- Hence, the abstract class solution is more restrictive on the classes that will get this guarantee. Thus, interfaces allow us to specify common behavior between seemingly unrelated objects:

Interfaces (Cont'd)



Interfaces (Cont'd)

- But abstract classes DO have an advantage over interfaces in that they allow us to define inherited state as well as non-abstract methods.

- So, in summary, how does having an interface help us ?
  - helps us extract common behavior among similar objects (kind of like inheritance through abstraction).
  - makes our code more robust and easier to understand in that it ensures that similar objects have at least certain behaviors available.

Interfaces (Cont'd)

- Step 1: Consider typical operations on machines that move. The interface on such objects may look as follows:

```
public interface  MovableObject {
public boolean    start();
public void       stop();
public boolean    turn(int degrees);
public double     fuelRemaining();
public void       changeSpeed(double kmPerHour);
}
```

- This interface essentially defines what all MovableObjects should be able to do. It is possible that they will do more, but this is the minimum.

Interfaces (Cont'd)

- Now, consider a Plane. It is a MovableObject and so it should implement this interface. We can implement the interface for planes as follows:

```
public class Plane implements  MovableObject{
public int      seatCapacity;
public Company  owner;
public Date     lastRepairDate;
public boolean start() {
    //Do whatever is necessary to start the plane. Return true if it actually started.
}
public void stop() {
   //Do whatever is necessary to stop the plane.
}
public boolean turn(int degrees) {
   //Do whatever is necessary to turn the plane. Return true if it actually worked.
}
public double fuelRemaining() {
   //Return the amount of plane fuel remaining.
}
public void changeSpeed(double kmPerHour) {
   //Do whatever is necessary to accelerate or decelerate by the given amount
}
   // There will likely also be some other methods which are plane-specific behaviors
public Date getLastRepairDate() {
....}
public double calculateWindResistance() {
....}
.....}
```

Interfaces (Cont'd)

- What about a Car, Train, Boat or even a Lawnmower ? We can implement the MovableObject interface for each of these as well.
- Suppose we'd like to set up a handheld remote control for MovableObjects. We can then treat all of the objects (Planes, Cars, Trains, etc...) as a single type of object ... a MovableObject.

```
public class RemoteControl {
    private MovableObject    machine;
    public RemoteControl(MovableObject m) {
        machine = m;
    }
    ...
    //When the start button is pressed on the remote
    boolean okay = machine.start();
    if (!okay) display("No Response on start");
    ...
}
```

Interfaces (Cont'd)

- Notice that the remote control constructor is given an object that is of type MovableObject. It could be a Plane, Car, Train, Boat, Lawnmower, ... We can create remote controls for more than one class and each will operate correctly using its own implementation of the MovableObject interface.

```
Plane         aPlane = new Plane();
Boat          aBoat = new Boat();
RemoteControl planeRemote = new RemoteControl(aPlane);
RemoteControl boatRemote = new RemoteControl(aBoat);
```

Interfaces (Cont'd)

- Recall that we did something identical with the Shape class when we allowed different types of shapes to be passed as parameters to a draw method for the Pen class.
- Guess what ? We can ALSO type-cast objects into their implemented interface:

```
Plane          aPlane = new Plane();
MovableObject    m = (MovableObject)aPlane;
m.start();
m.stop();
m.getLastRepairDate();//This won't work now
```

Interfaces (Cont'd)

- So, everything works as it did in the Shape example. Once we type-cast to an interface type, we can only send interface methods to the Plane. Although it is still indeed a Plane object, we cannot send Plane-specific methods to the object anymore (unless it is type-casted back to Plane).
- We may also use an interface name when specifying the type of objects being stored in our collections:

Interfaces (Cont'd)

```
ArrayList<MovableObject>  toys = new ArrayList<MovableObject>();
toys.add(new Plane());
toys.add(new Boat());
toys.add(new Car());

for (MovableObject toy: toys) {
   toy.start();
   toy.turn(120);
   System.out.println(toy.fuelRemaining());
   toy.stop();
}
```

References

- Carleton University COMP 1005 and 1006:
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1405/Notes/
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1406/Notes/
- Armstrong Atlantic State University
  http://www.cs.armstrong.edu/liang/intro10e/
- Oracle Java tutorial:
  http://docs.oracle.com/javase/tutorial/
- MIT opencourseware
  https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/