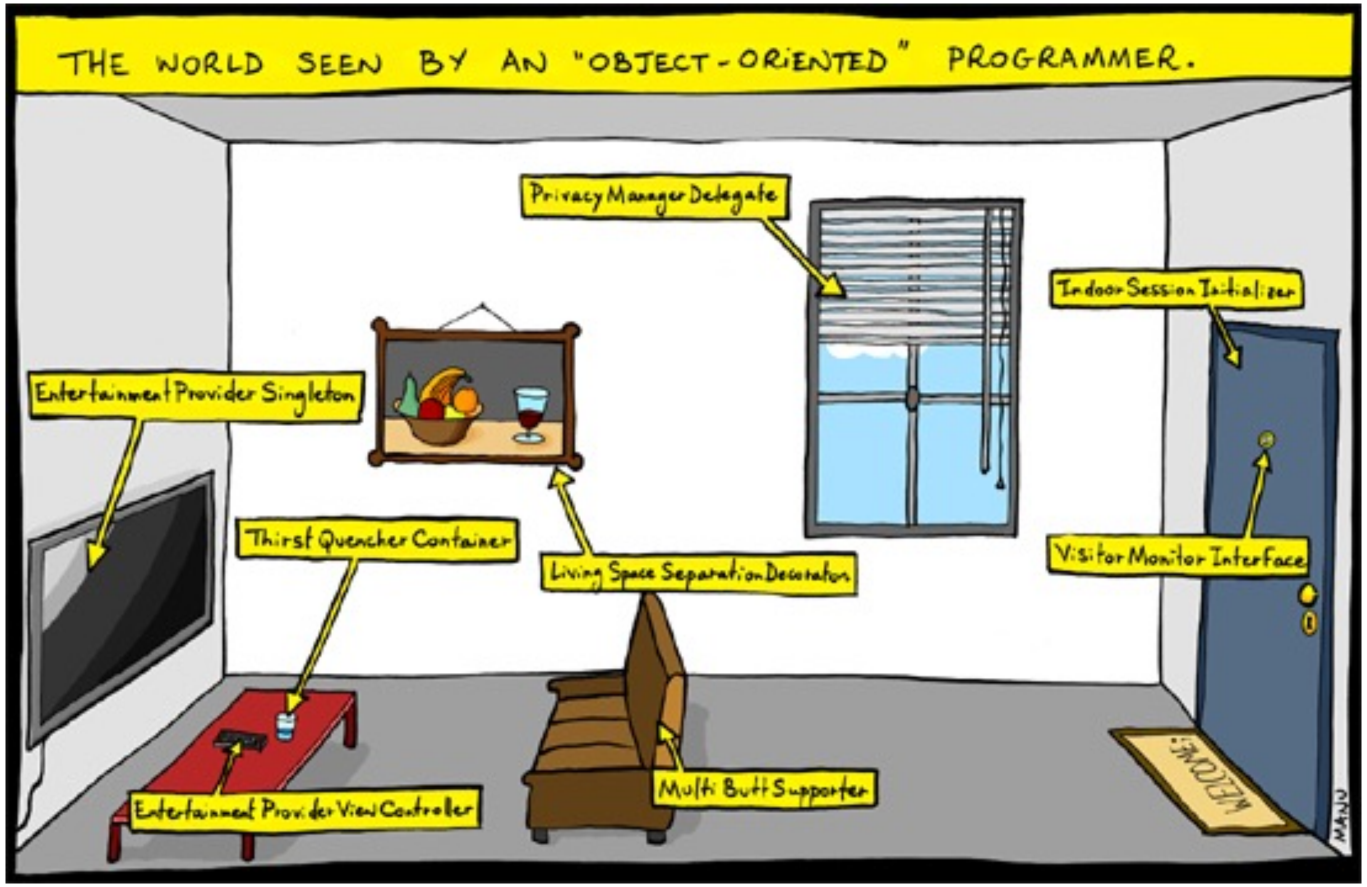
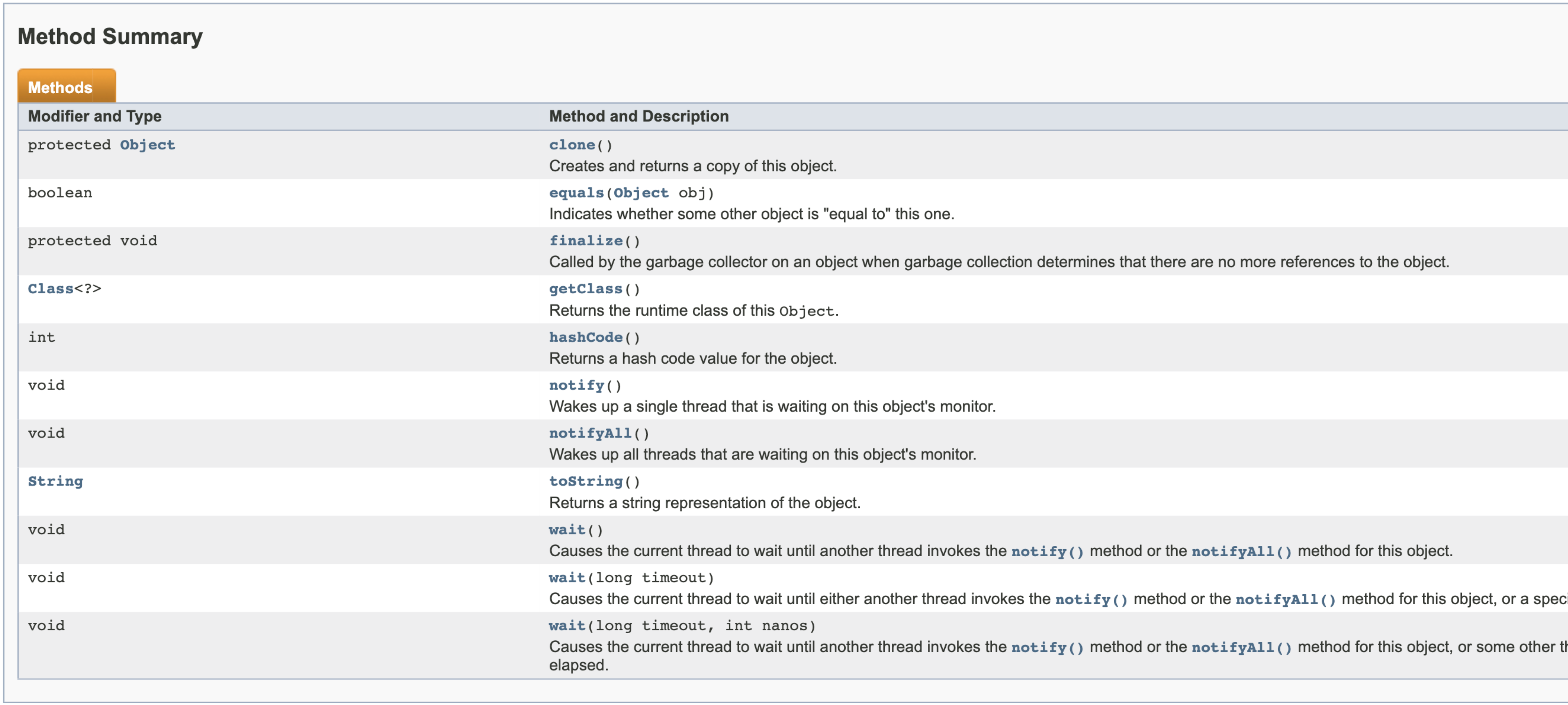

THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
Privacy Manager Delegate
Indoor Session Initializer
Entertainment Provider Singleton
Thirst Quencher Container
Living Space Separation Decorator
Visitor Monitor Interface
Multi Butt Supporter
Entertainment Provider View Controller
WELCOME!
MANU

Object Oriented Programming

Nazim Ashraf

Object class

- Class Object is the root of the class hierarchy.
- Every class has Object as a superclass.
- All objects, including arrays, implement the methods of this class.



Method Summary

Methods

Modifier and Type	Method and Description
`protected Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	`getClass()` Returns the runtime class of this `Object`.
`int`	`hashCode()` Returns a hash code value for the object.
`void`	`notify()` Wakes up a single thread that is waiting on this object's monitor.
`void`	`notifyAll()` Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()` Returns a string representation of the object.
`void`	`wait()` Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
`void`	`wait(long timeout)` Causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a spec
`void`	`wait(long timeout, int nanos)` Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object, or some other th elapsed.

- Always a good idea to override
  - toString
  - equals
  - clone

Things to remember

- Superclass CANNOT access base class's methods!

Has a vs Is a

Inheritance	Composition
Is A	Has a

class Staff extends Person…
 	
Staff is a Person
	class Staff{
	    Person person
	
	Staff has a Person

class Book extends Author
 	
Book is an Author
	class Book{
	  Author author;
	
	Book has an author

Circle and Cylinder class

- Define a Circle class

- Cylinder extends Circle
    - Height
    - getVolume -> uses super class getArea since volume is area of circle x height
    - getArea -> Override getArea: use super class getArea since area of cyclinder is 2 * area of circle + height x circumference of circle

Alternate way to think of the same problem

- Define a Circle class

- Cylinder has a circle and a height
    - getVolume -> uses Circle class getArea since volume is area of circle x height
    - getArea -> use Circle class getArea since area of cyclinder is 2 * area of circle + height x circumference of circle

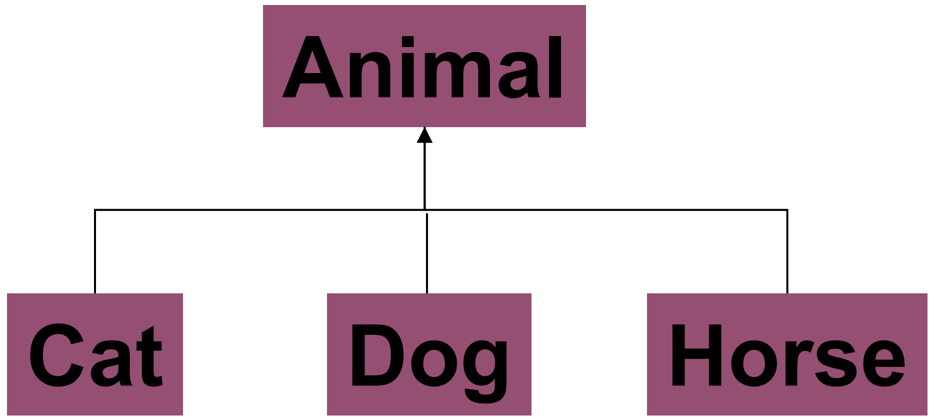
What is the difference?

- Interface is the same
- Implementation issues

Polymorphism

- Super class object can refer to any base class…

Example class hierarchy



Polymorphism

- Normally we have this when we create an object:

```
Dog dog = new Dog();
```

- Polymorphism allows us to also do this:

```
Animal pet = new Dog();
```

  - The object reference variable can be a super class of the actual object type! (Does NOT work the other way around: Dog is an Animal but Animal is not necessarily a Dog)

Where Polymorphism is Helpful

- Arrays
- Passing parameters
- Returning values from a method

Polymorphic Array Example

```
Animal[] myPets = new Animal[5];
myPets[0] = new Cat();
myPets[1] = new Cat();
myPets[3] = new Dog();

for (int i = 0; i < myPets.length; i++) {
    myPets.feed();
}
```

$\}$ You can put any subclass of Animal in the Animal array!

Polymorphic Arguments

```
public class Vet {
    public void giveShot(Animal pet) {
        pet.makeNoise();
    }
}

public class PetOwner {
    Vet vet = new Vet();
    Dog dog = new Dog();
    Cat cat = new Cat();

    vet.giveShot(dog);
    vet.giveShot(cat);
}
```

But…

- We can only call overridden methods
- For instance, if we have:

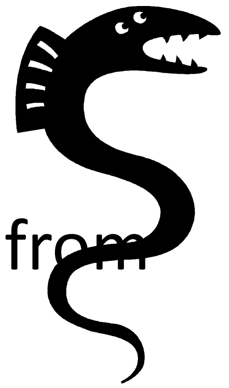
```
Animal pet = new Dog();
```

Then the pet can only access methods/instance variables inside Animal class.

Abstract Classes

- Sometimes we don’t want to allow an object to be created of a certain type.
  - What exactly would an Animal object be?

- We use the keyword abstract to prevent a class from ever being instantiated.



```
abstract public class Animal
```

Abstract Classes

- Can still use abstract classes as a reference variable, for the purposes of polymorphism.
- An abstract class has no use until it is extended!
- A class that is not abstract is called concrete.

Abstract Methods

- An abstract method has no body and is marked with the keyword **`abstract`**.

  ```
  public abstract void eat();
  ```

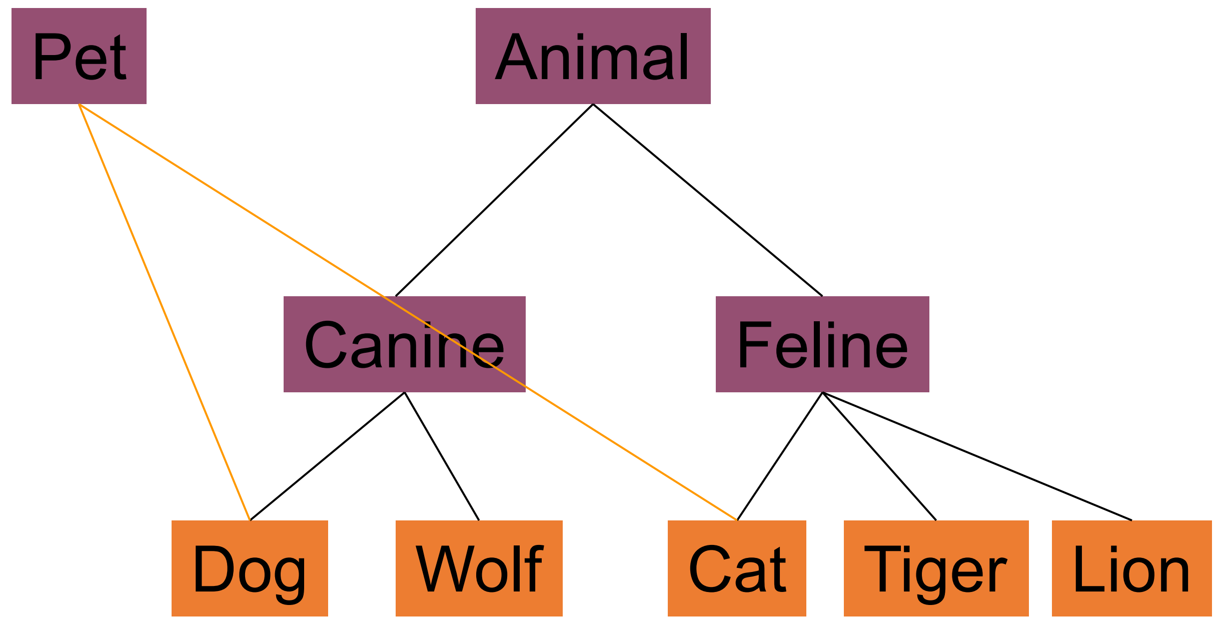
- If a method is abstract, the class it is contained in must also be abstract.

- Abstract methods help the programmer to provide a protocol for a group of subclasses.

- The first concrete class in the inheritance hierarchy must implement the abstract method (i.e. override it and provide it a body)

Side Effects of Polymorphism

```
ArrayList pets = new ArrayList();
Dog dog = new Dog();
pets.add(dog);
int index = pets.indexOf(dog);
Dog dog1 = pets.get(index);         // won't work

Object dog2 = pets.get(index);
dog2.bark();                        // won't work

((Dog)dog2).bark();                 // works because of casting

if (dog2 instanceof Dog) {          // being careful
 ((Dog)dog2).bark();
}

Dog dog3 = (Dog) pets.get(index);  // works because of casting

if (dog2 instanceof Dog) {          // being careful
 Dog dog4 = (Dog) dog2;
```



But remember we said that Java does not support multiple inheritance. There is a solution however: interfaces.

Interfaces

- Interface: A collection of constants and abstract methods that cannot be instantiated.
- A class implements an interface by providing method implementations for each of the abstract methods defined in the interface.

```
public class Dog extends Canine implements Pet
```

Interfaces

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}

public class Dog extends Canine implements Pet {
    public void beFriendly() {
        wagTail();
    }

    public void play() {
        chaseBall();
    }

    . . .  all the other Dog methods . . .
}
```

Explicitly typing in public and abstract is not necessary since they MUST be public and abstract

Must implement these methods since they are in Pet

Interfaces vs. Subclasses

- Make a subclass only when you want to make a more specific version of a class.

- Use an interface when you want to define a role that other classes can play, regardless of where those classes are in the inheritance tree.

Polymorphism via Interfaces

- An interface reference variable can be used to refer to any object of any class that implements that interface.
- This works the same with superclasses.

```
Pet myPet = new Dog();
```

- The same side effects of polymorphism occur with interfaces as with inheritance.

Comparable Interface

- Defined in the `java.lang` package
- Only contains one method: `compareTo` which takes an object as a parameter and returns an integer. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Provides a common mechanism for comparing one object to another.
- http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html

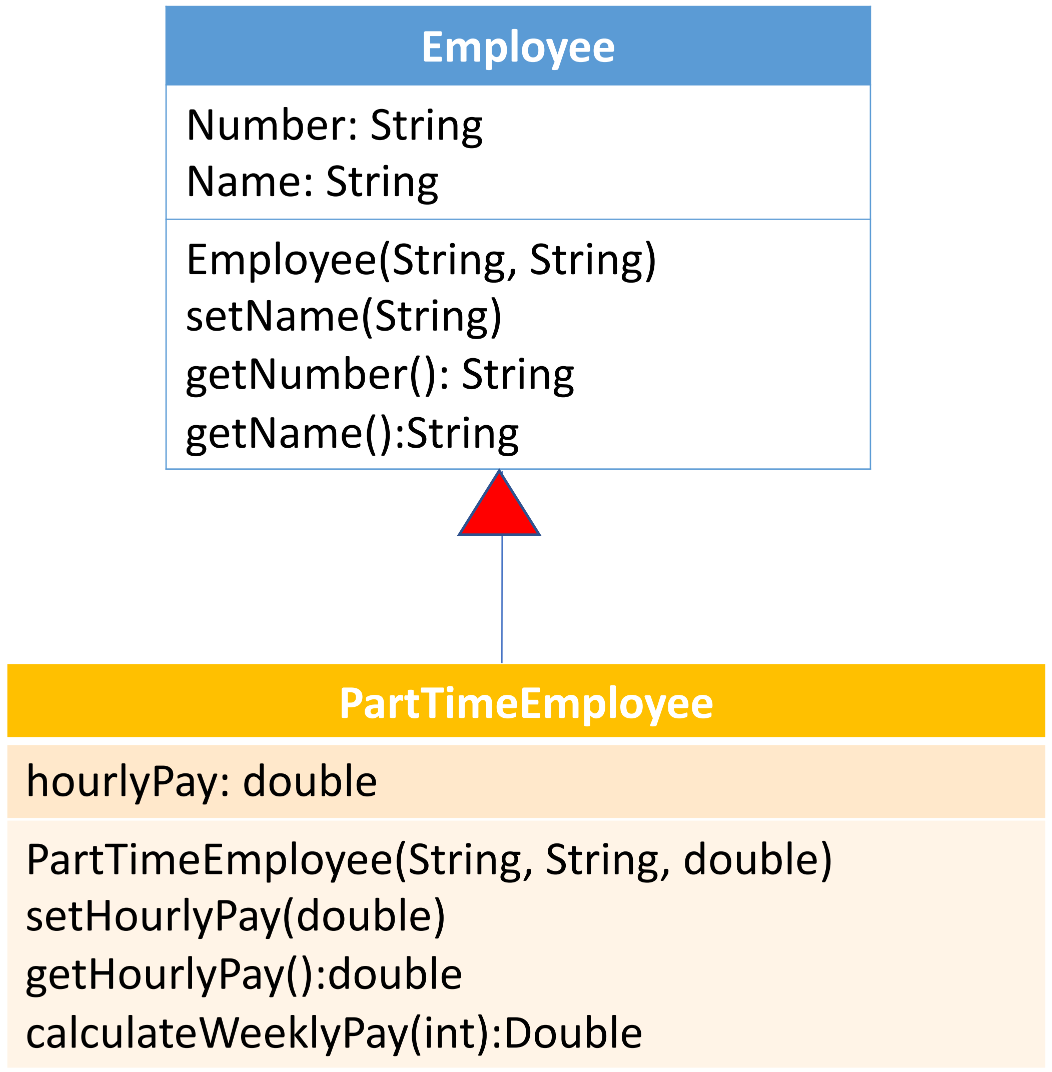
Inheritance cont’d

- A superclass can have multiple subclasses
- Subclasses can be superclasses of other subclasses
- A subclass can inherit directly from only one superclass
- All classes inherit from the *Object* class
- An important concept in inheritance is that an object of a subclass is also an object of any of its superclasses.

Superclasses and Subclasses

- A big advantage of inheritance is that we can write code that is common to multiple classes once and reuse it in subclasses
- A subclass can define new methods and instance variables, some of which may override (hide) those of a superclass

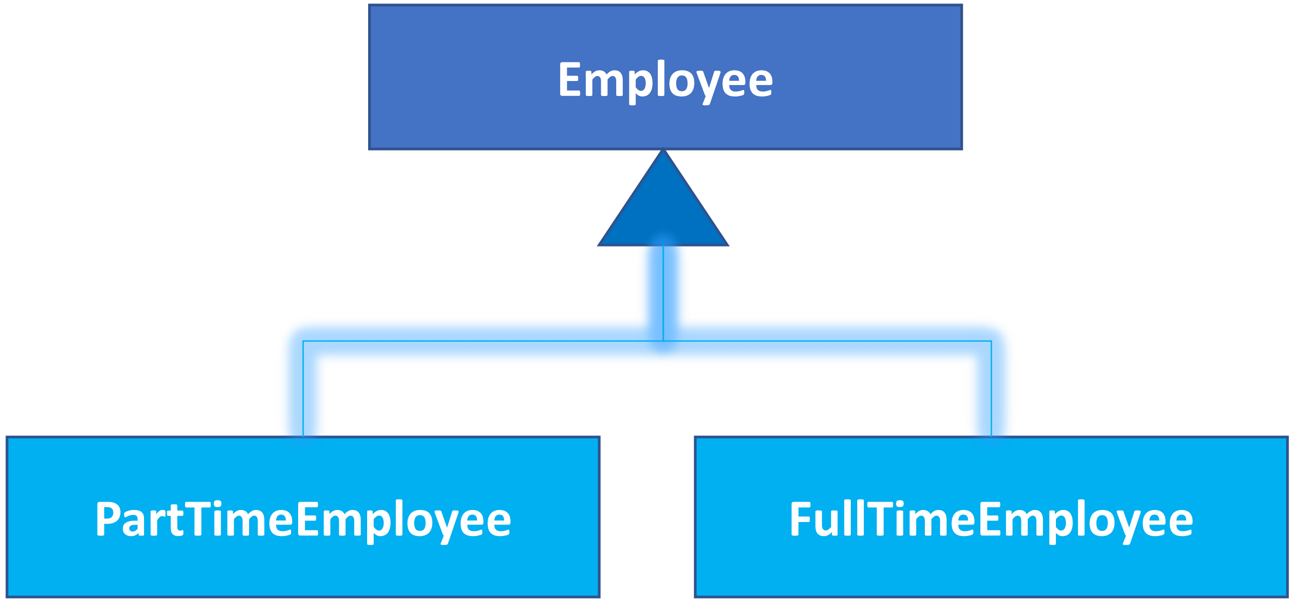
Abstract Classes



Situation

- Our business expands and we now employee full time employees as well as part-time employees.
- Difference between full time and part time?
  - Instead of hourly play will have an annual salary
  - Might need a method that calculates monthly pay

New Hierarchy



Lets Identify the attributes

Employee	PartTimeEmployee	FullTimeEmployee
Number:String	hourlyPay:double	annualSalary:double
Name: String		

Lets Identify the methods

Employee	PartTimeEmployee	FullTimeEmployee

Employee(String, String)
setName(String)
getNumber():String
getName(): String
	PartTimeEmployee(string, String, double)
	setHourlyPay(double)
	getHourlyPay():double
	calculateWeeklyPay(int):double
		FullTimeEmployee(String, String, double)
		setAnnualSalary(double)
		getAnnualSalary(): double
		calculateMonthlypay():double

Will there be any employee who is neither FULL TIME, nor PART TIME, rather just an EMPLOYEE???

How to prevent people from creating an object of EMPLOYEE Class??

- Declare the class as abstract
  - Public abstract class Employee
- Once a class is declared in this way, it means that it is not allowed to create objects of that class.
- Why we need Employee class then?
  - Acts as basis on which to build other classes
- An object reference to an *abstract* class can be declared.

Restrictions for Defining *abstract* Classes

- Classes must be declared *abstract* if the class contains any *abstract* methods
- *abstract* classes can be extended
- An object reference to an *abstract* class can be declared
- *abstract* classes cannot be used to instantiate objects

Wish to know that whether a Employee is PartTime or FullTime

- Need a method getStatus()
  - Will return a String declaring the status

Employee	PartTimeEmployee	FullTimeEmployee
Employee(String, String) setName(String) getNumber():String getName(): String **getStatus():String**	PartTimeEmployee(string, String, double) setHourlyPay(double) getHourlyPay():double calculateWeeklyPay(int):double **getStatus():String**	FullTimeEmployee(String, String, double) setAnnualSalary(double) getAnnualSalary(): double CalculateMonthlypay():double **getStatus():String**

Discussion

- Does the Employee Objects(if any) has any status???
  - NO
- Does the Employee Class needs any status message??
  - YES and NO
  - Yes as we need it to have method overriding
  - NO as there is no behavior
- How to prevent this having its own behavior
  - Declare it as **abstract**

Abstract Method

- abstract public String getStatus();
- Note that its only a header but no body
- Force all subclasses of our class to implement this method
- In this case PartTimeEmployee, Full TimeEmployee, and any other future subclass of Employee will have to have a method called getStatus, if they didn’t , program will not compile
- Each subclass will OVERRIDE this method

Restrictions for Defining *abstract* Methods

- *abstract* methods can be declared only within an *abstract* class
- An *abstract* method must consist of a method header followed by a semicolon
- *abstract* methods cannot be called
- *abstract* methods cannot be declared as *private* or *static*
- A constructor cannot be declared *abstract*

final Modifier

- What is the use of final modifier??
  - Recall final int x = 15;
- What if it is places before a class or method??

public final class Someclass
{
    //code goes here
}
	public final void someMethod()
	{
	    //code goes here
	}

Class cannot be subclassed	Method cannot be overridden

A mixed list

- Try to create an Employee List
  - Where some employees are part time and some are full time

Interfaces

A class can inherit directly from only one class, that is, a class can *extend* only one class.

To allow a class to inherit behavior from multiple sources, Java provides the interface.

- An interface typically specifies behavior that a class will *implement*. Interface members can be any of the following:
  - classes
  - constants
  - *abstract* methods
  - other interfaces

Interface Syntax

To define an interface, use the following syntax:

```
accessModifier interface InterfaceName
{
   // body of interface
}
```

All interfaces are *abstract*; thus, they cannot be instantiated. The *abstract* keyword, however, can be omitted in the interface definition.

Finer Points of Interfaces

- An interface's fields are *public*, *static,* and *final.* These keywords can be specified or omitted.
- When you define a field in an interface, you must assign a value to the field.
- All methods within an interface must be *abstract,* so the method definition must consist of only a method header and a semicolon. The *abstract* keyword also can be omitted from the method definition.

Interface

- A class in which all methods are abstract

Inheriting from an Interface

To inherit from an interface, a class declares that it implements the interface in the class definition, using the following syntax:

```
accessModifier class ClassName
     extends SuperclassName
     implements Interface1, Interface2, …
```

- The *extends* clause is optional.
- A class can *implement* 0, 1, or more interfaces.
- When a class *implements* an interface, the class must provide an implementation for each method in the interface.

Example

We define an *abstract* class *Animal* with one *abstract* method

```
public abstract void draw( Graphics g );
```

We define a *Moveable* interface with one abstract method:

```
public interface Moveable
{
   int FAST = 5; // static constant
   int SLOW = 1; // static constant

   void move( ); // abstract method
}
```