Inheritance

Ye Yang
Stevens Institute of Technology

Review of Objects and Classes

- In object-oriented programming we describe *types of objects* by defining classes.
- A Java class definition contains
  - fields - what properties an object has
    - The values assigned to the fields define the state of the object.
    - Fields should be defined as private (visible to methods of the same class and hidden to methods of other classes).
  - methods - what behaviors (actions) an object can perform
    - Typically these actions supply or modify its state.
- We then can create multiple objects from that class and "fill in" the properties with values specific to each object, and ask the object to perform their behaviors.
  - new operator: returns a reference to the newly created object
  - constructor: initialization
- Every object belongs to one class and is an instance of the class.

Objectives

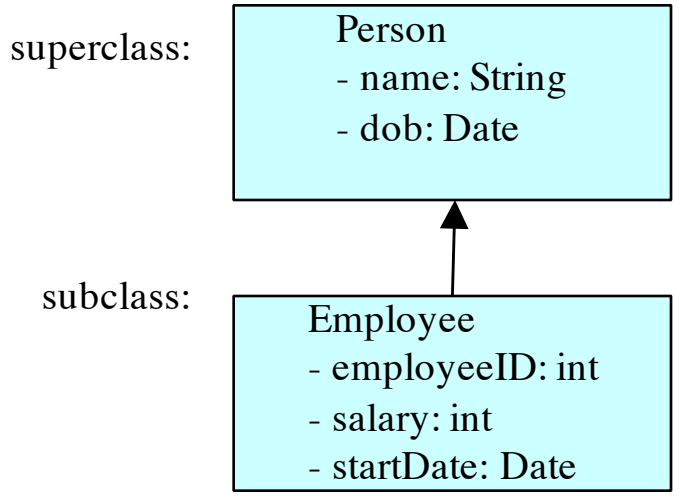
- To define the syntax of inheritance in Java
- To understand the class hierarchy of Java
- To examine the effect of inheritance on constructors

What is Inheritance?

- Inheritance is a fundamental Object Oriented concept
  - Modeling the IS-A relationship among classes
  - For code reuse among classes

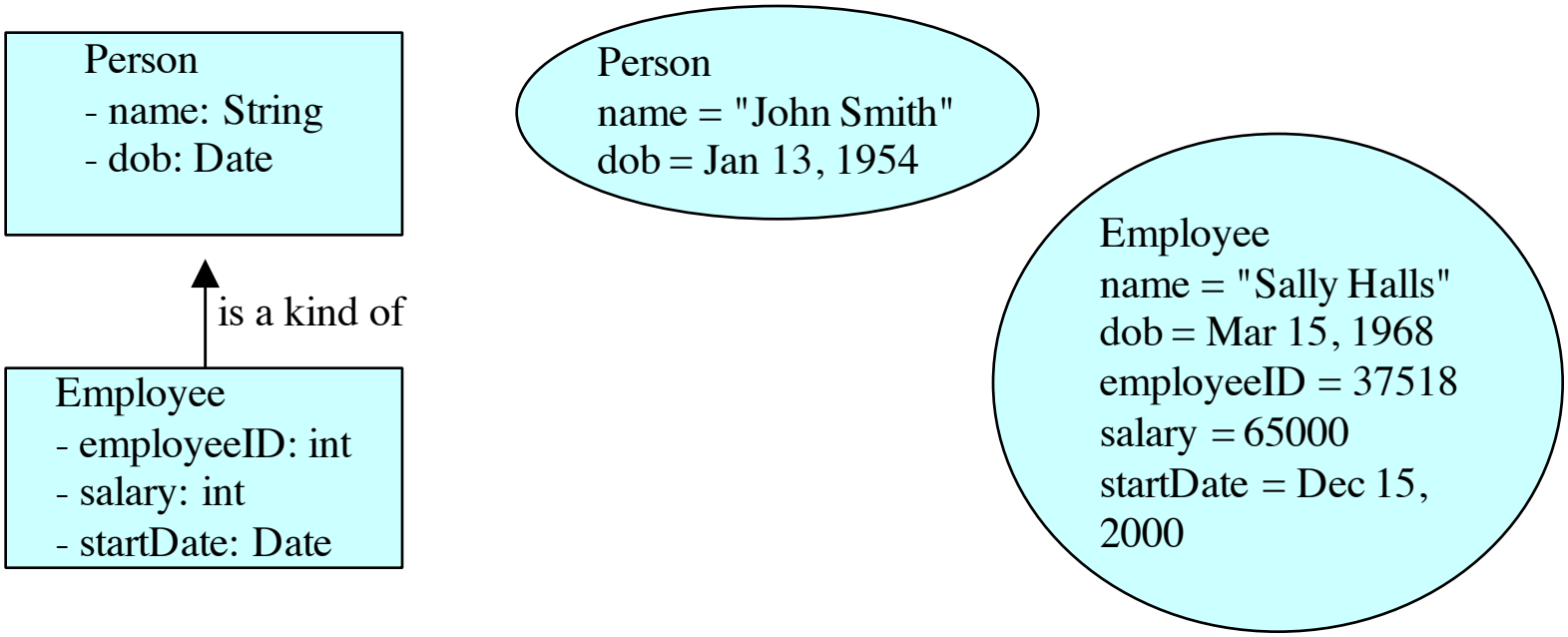
- A class can be defined as a "subclass" of another class.
  - The subclass inherits all data attributes of its superclass
  - The subclass inherits all methods of its superclass
  - The subclass inherits all associations of its superclass

- The subclass can:
  - Add new functionality
  - Use inherited functionality
  - Override inherited functionality



What really happens?

- When an object is created using new, the system must allocate enough memory to hold all its instance variables.
  - This includes any inherited instance variables
- In this example, we can say that an Employee "is a kind of" Person.
  - An Employee object inherits all of the attributes, methods and associations of Person



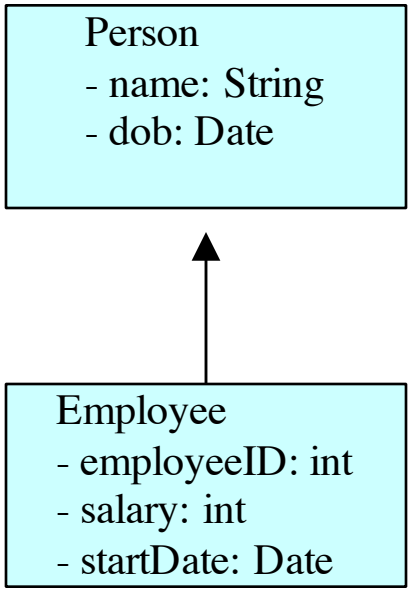
Inheritance in Java

- Inheritance is declared using the "extends" keyword
  - If inheritance is not defined, the class extends a class called Object

```
public class Person
{
   private String name;
   private Date dob;
   [...]
```

```
public class Employee extends Person
{
   private int employeID;
   private int salary;
   private Date startDate;
   [...]
```

```
Employee anEmployee = new Employee();
```



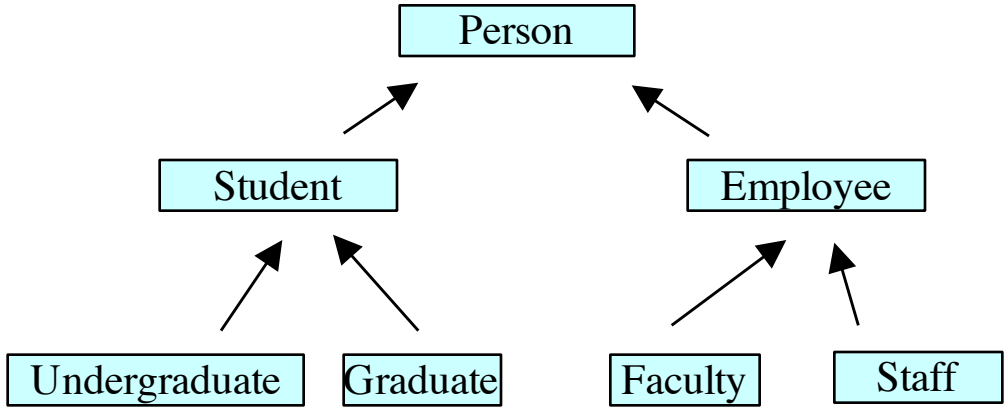
Inheritance Hierarchy

- Each Java class has one (and only one) superclass.
  - C++ allows for multiple inheritance

- Inheritance creates a class hierarchy
  - Classes higher in the hierarchy are more general and more abstract
  - Classes lower in the hierarchy are more specific and concrete

- There is no limit to the number of subclasses a class can have

- There is no limit to the depth of the class tree.



Controlling Inheritance

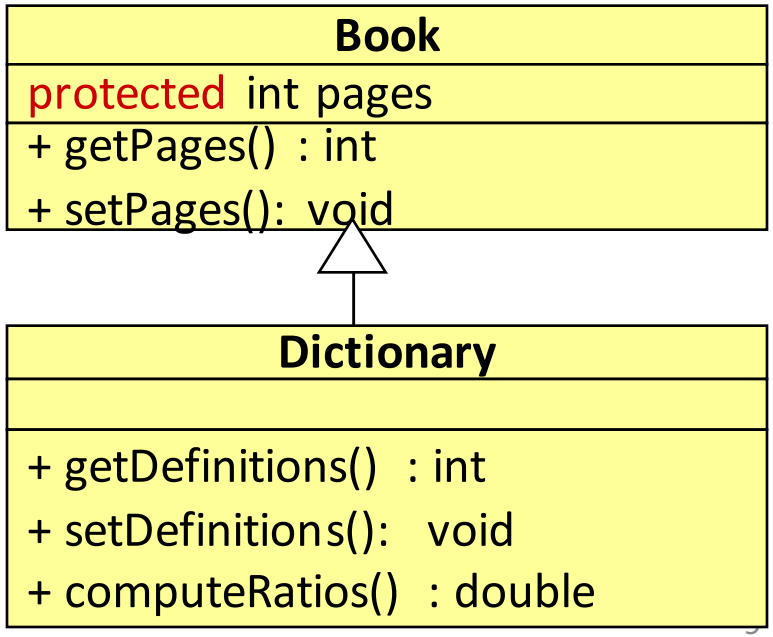
- Visibility modifiers determine which class members are accessible and which do not
- Members (variables and methods) declared with `public` visibility are accessible, and those with `private` visibility are not
- Problem: How to make class/instance variables visible only to its subclasses?
- Solution: Java provides a third visibility modifier that helps in inheritance situations: `protected`

The `protected` Modifier

- The `protected` visibility modifier allows a member of a base class to be accessed in the child
  - `protected` visibility provides more encapsulation than `public` does
  - `protected` visibility is not as tightly encapsulated as `private` visibility

All these methods can access the *pages* instance variable.

Note that by constructor chaining rules, *pages* is an instance variable of every object of class Dictionary.



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, i.e. "common ancestor"
  - even 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
  - the `Object` class is therefore the ultimate root of all class hierarchies
- The `Object` class contains a few useful methods, which are inherited by all classes
  - `toString()`
  - `equals()`
  - `clone()`

The `Object` Class: the `toString` Method

- That's why the `println` method can call `toString` for any object that is passed to it – all objects are guaranteed to have a `toString` method via inheritance

- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class and a hash value

- Every time we have defined `toString`, we have actually been overriding it

The `Object` Class: the `equals` Method

- The `equals` method of the `Object` class determines if two variables point to the same object
- Many classes (which all implicitly extend an `Object` class) override `equals` to define equality in some other way, e.g. `Integer.equals` looks at the represented integer, etc.

Constructors and Initialization

- Classes use constructors to initialize instance variables
  - When a subclass object is created, its constructor is called.
  - It is the responsibility of the subclass constructor to invoke the appropriate superclass constructors so that the instance variables defined in the superclass are properly initialized
- Superclass constructors can be called using the "super" keyword in a manner similar to "this"
  - It must be the first line of code in the constructor
- If a call to super is not made, the system will automatically attempt to invoke the no-argument constructor of the superclass.

Constructors - Example

```
public class BankAccount
{
   private String ownersName;
   private int accountNumber;
   private float balance;

   public BankAccount(int anAccountNumber, String aName)
   {
         accountNumber = anAccountNumber;
         ownersName = aName;
   }
   [...]
}

public class OverdraftAccount extends BankAccount
{
   private float overdraftLimit;

   public OverdraftAccount(int anAccountNumber, String aName, float aLimit)
   {
         super(anAccountNumber, aName);
         overdraftLimit = aLimit;
   }
}
```

Method Overriding

- Subclasses inherit all methods from their superclass
  - Sometimes, the implementation of the method in the superclass does not provide the functionality required by the subclass.
  - In these cases, the method must be overridden.
- To override a method, provide an implementation in the subclass.
  - The method in the subclass MUST have the exact same signature as the method it is overriding.

Method overriding - Example

```
public class BankAccount
{
   private String ownersName;
   private int accountNumber;
   protected float balance;

   public void deposit(float anAmount)
   {
         if (anAmount>0.0)
                  balance = balance + anAmount;
   }

   public void withdraw(float anAmount)
   {
         if ((anAmount>0.0) &&
   (balance>anAmount))
                  balance = balance - anAmount;
   }

   public float getBalance()
   {
         return balance;
   }
}
```

```
public class OverdraftAccount extends
   BankAccount
{
   private float limit;

   public void withdraw(float anAmount)
   {
         if ((anAmount>0.0) &&
   (getBalance()+limit>anAmount))
                  balance = balance -
   anAmount;
   }

}
```

Overloading vs. Overriding

- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overloading lets you define a similar operation in different ways for different data

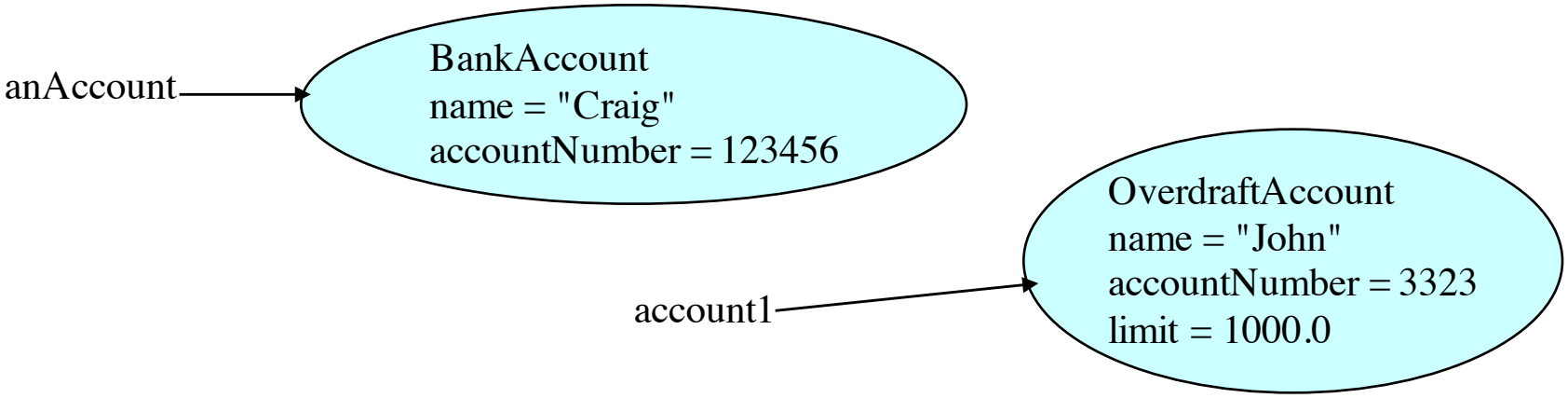
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overriding lets you define a similar operation in different ways for different object types

Object References and Inheritance

- Inheritance defines "a kind of" relationship.
  - In the previous example, OverdraftAccount "is a kind of" BankAccount
- Because of this relationship, programmers can "substitute" object references.
  - A superclass reference can refer to an instance of the superclass OR an instance of ANY class which inherits from the superclass.

```
BankAccount anAccount = new BankAccount(123456, "Craig");

BankAccount account1 = new OverdraftAccount(3323, "John", 1000.0);
```



Polymorphism - 1

- In the previous slide, the two variables are defined to have the same type at compile time: BankAccount
  - However, the types of objects they are referring to at runtime are different
- What happens when the withdraw method is invoked on each object?
  - anAccount refers to an instance of BankAccount. Therefore, the withdraw method defined in BankAccount is invoked.
  - account1 refers to an instance of OverdraftAccount. Therefore, the withdraw method defined in OverdraftAccount is invoked.
- Polymorphism is: The method being invoked on an object is determined AT RUNTIME and is based on the type of the object receiving the message.
  - This is called dynamic binding or dynamic dispatch

Packages

- The Java API is organized into packages
- The package to which a class belongs is declared by the first statement in the file in which the class is defined using the keyword package followed by the package name
- All classes in the same package are stored in the same directory or folder
- All the classes in one folder must declare themselves to be in the same package
- Classes that are not part of a package may access only public members of classes in the package

The No-Package-Declared Environment and Package Visibility

- There exists a default package
  - Files that do specify a package are considered part of the default package
- If you don't declare packages, all of your packages belong to the same, default package
- Package visibility sits between private and protected
  - Classes, data fields, and methods with package visibility are accessible to all other methods of the same package but are not accessible to methods outside of the package
  - Classes, data fields, and methods that are declared protected are visible to all members of the package

Visibility Supports Encapsulation

- The rules for visibility control how encapsulation occurs in a Java program
- Private visibility is for members of a class that should not be accessible to anyone but the class, not even the classes that extend it
- Package visibility allows the developer of a library to shield classes and class members from classes outside the package
- Use of protected visibility allows the package developer to give control to other programmers who want to extend classes in the package

Visibility Supports Encapsulation (continued)

TABLE 3.3
Summary of Kinds of Visibility

Visibility	Applied to Classes	Applied to Class Members
`private`	Applicable to inner classes. Accessible only to members of the class in which it is declared.	Visible only within this class.
Default or package	Visible to classes in this package.	Visible to classes in this package.
`protected`	Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared.	Visible to classes in this package and to classes outside the package that extend this class.
`public`	Visible to all classes.	Visible to all classes. The class defining the member must also be public.

Final Methods and Final Classes

- Methods can be qualified with the final modifier
  - Final methods cannot be overridden.
  - This can be useful for security purposes.

```
public final boolean validatePassword(String  username, String Password)
{
   [...]
```

- Classes can be qualified with the final modifier
  - The class cannot be extended
  - This can be used to improve performance. Because there an be no subclasses, there will be no polymorphic overhead at runtime.

```
public final class Color
{
   [...]
```

Using Libraries

- Java programmers need to learn how to search the class libraries, identify classes they need and the class methods, as well as figuring out how to use them.
- To do this you need to understand object-oriented principles, as well as how to `implement` interfaces and `extend` classes. The basics of these topics are in this lecture.

Searching the Libraries

- To search the class libraries, a good starting place is java.sun.com. Any link will quickly be obsolete as new versions of Java are released, but an example of the libraries was at https://docs.oracle.com/javase/7/docs/api/
- Later, we will introduce the `javadoc` utility, which catalogs the functionality that you acquire by using classes from the Java libraries.