OOJ Lecture 4
Inheritance

- Reading: Savitch, Chapter 7
- Reference: Big Java, Horstman, Chapter 11

Objectives

- To know how to convert from Subclasses to Superclasses
- To understand Abstract classes & Abstract methods
- To learn about protected and package access control

What is the "Type" of a subclass?

- Subclasses have more than one type
- Of course they have the type of the extending class (the subclass they define)
- They also have the type of every ancestor class
  - all the way to the top of the class hierarchy
- *All* classes extend from the original, predefined class `Object`

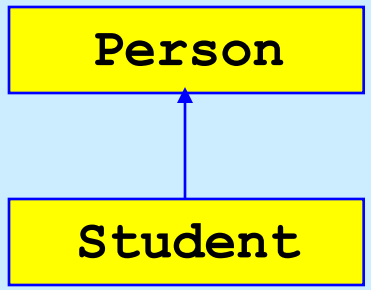
Assignment Compatibility

- **Can** assign an object of a derived class to a variable of any ancestor type

```
Person john;
Student Top1 = new
  Student("eric");
john = top1;  //OK
```

- Can **not** assign an object of an ancestor class to a variable of a derived class type

```
Person john = new Person();
Student Top1;
top1 = john;
```



Person is the parent class of Student in this example.

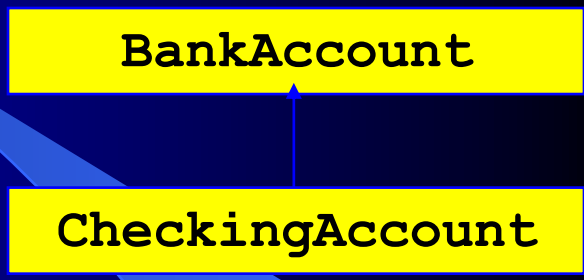
"Is a" and "Has a" Relationships

- Inheritance is useful for "is a" relationships.
  - A student "is a" person.
  - `Student` inherits from `Person`.
- Inheritance is usually **not** useful for "has a" relationships.
  - A student "has a(n)" enrollment date.
  - Add a `Date` object as an instance variable of `Student` instead of having `Student` inherit from `Date`.
- If it makes sense to say that an object of `Class1` "is a(n)" object of `Class2`, then consider using inheritance.

Converting from Subclasses to Superclasses

```
//Reference: Horstmann, Chapter 11
class BankAccount{..
   void transfer(BankAccount other,
                 double amount)
   {   withdraw(amount);
       other.deposit(amount);
   }
..}

   BankAccount momsAccount = new BankAccount();
   CheckingAccount HarryCheck = new
     CheckingAccount(10);
   momsAccount.transfer(harryCheck, 100);
```



Converting from Subclasses to Superclasses

- It is OK to pass a CheckAccount reference to a method expecting a BankAccount
  - Then the method *deposit* in *transfer* will be actually called from CheckAccount
  - The method called always by the types of the actual object, not the reference

Abstract Classes & Abstract Methods

- For software engineering planning purposes, we need abstract methods and classes.
- An "abstract" method
  - is a method that is not implemented, defined with modifier "abstract".
- An "abstract" class
  - is a class containing at least an abstract method, also defined with "abstract" keyword.
  - Cannot be instantiated
- Precisely the opposite of the "final" keyword.
- An abstract class must be extended to non-abstract class before it can be instantiated.

Abstract Classes & Abstract Methods

- A class can only extend one class. But an interface can extend many interfaces

- An abstract class does not have to contain only pure abstract methods (comparing with the *interface)*
  - It may contain some non-abstract methods (i.e. implemented methods) and one or more abstract methods
  - While an *interface* does not contain any implementation code. It is PURE abstract

Abstract Classes Example

```
abstract class Animals {    // abstract class
  int useless = 0;
  abstract void wish();     // abstract method
  void speech()
  { System.out.print("This is an abstract class");
    wish();
  }
…}
```

Remember an abstract class cannot be instantiated

```
Animals Tom = new Animals(); // Wrong!

class Cat extends Animals {  // Now not abstract
      void wish() {
      System.out.println("Good Luck");}
}
Cat Tom = new Cat();   // Legal
```

Modifier “final” versus “abstract”

- The “final” modifier defines a class/method/variable which can’t be extended. A final method can’t be overridden.

- It is opposed to the ‘abstract’ modifier. An abstract class must be extended before it can be used to generate objects

Access Control Level

- Variables/fields and methods can be modified by access control modifiers, to define their accessibility
- Java has four levels of controlling access to fields, methods and classes:
  1. `public`
  2. `private`
  3. `protected` (accessible by subclasses and package)
  4. package access (the default, no modifier)

Access Control Level

- `public`:
  - can be used everywhere, by any packages or classes
- `private`:
  - private modifier = information encapsulation.
  - used in its own class & can't be accessed outside the class (not even by its subclasses).
  - keeps secret information and only provides this to controlled access methods.
  - when subclassing: non-private can't become private.
- `protected`:
  - can be accessed within its own package & subclasses.

Access Control Level

- Fields and methods that are not declared as `public`,`private` or `protected` can be accessed by all classes in the same package
  - This default is *package access*.

- Package access is a good default for classes, but not desirable for fields.
  - Instance and static fields of classes should always be private

Recommended Access Levels

- Fields: Always `private`
- Exceptions
  - Constants: `public static final` -- safe
  - Object: `System.out` should be `public`
- Methods: `public` or `private`
- Classes: `public` or package
- Beware of accidental package access (forgetting `public` or `private`)

Recommended Access Levels

- If a superclasses declares a method to be public, subclasses cannot override it to be more private
- The compiler doesn't allow this

Example

```
public class BankAccount
{ ...
   public void withdraw(double amount)
    ..
public class CheckingAccount extends BankAccount
{ ...
   private void withdraw(double amount)   //WRONG
    ...
```

Protected Access

- `protected` modifier allows access to its own package & its subclasses.

If the subclass wants to access the balance from BankAccount, the balance should be defined as protected

Example

```
public class BankAccount
{ ...
   protected double balance; ...}
```

`protected` fields are hard to change and maintain

- May be used by a subclass
- May be corrupted by a subclass
- Other classes in the same package

Class Exercise

- Write a subclass *HourlyEmp* for the following Employee class. The hourly employees are paid by hours with an hourly rate.
- How do you get the salary of hourly employees?

```
public class Employee
{  private double salary;
   private String name;
   public Employee (String aName, double aSalary)
   {          ...                    }
   public double getSalary()
   {     return salary;               }
   public void setSalary(double newSalary)
   {    salary = newSalary;           }
   public void writeOutput()
   {    System.out.println("Name: " + name);
        System.out.println("Salary: " + salary);}
```

Class Exercise

```
public class HourlyEmp extends Employee
{
  private int hours;
  public HourlyEmp(String aName, int hours)
  { // fill in the constructor  }
  public double calculateSalary()
  {
     // NOTE: Salary is a private variable
     // of Employee.
```

You may use setSalary(newSalary) of Employee to assign the salary of hourly employees.

```
  }
}
```