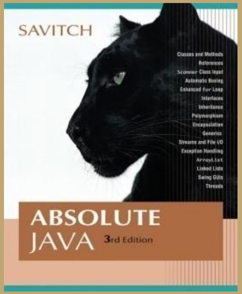
# Comp 249
# Programming Methodology
## Chapter 7 - *Inheritance – Part A*

*Dr. Aiman Hanna*
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of Absolute Java by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.



UNIVERSITÉ Concordia UNIVERSITY

PEARSON Addison Wesley

# Introduction to Inheritance

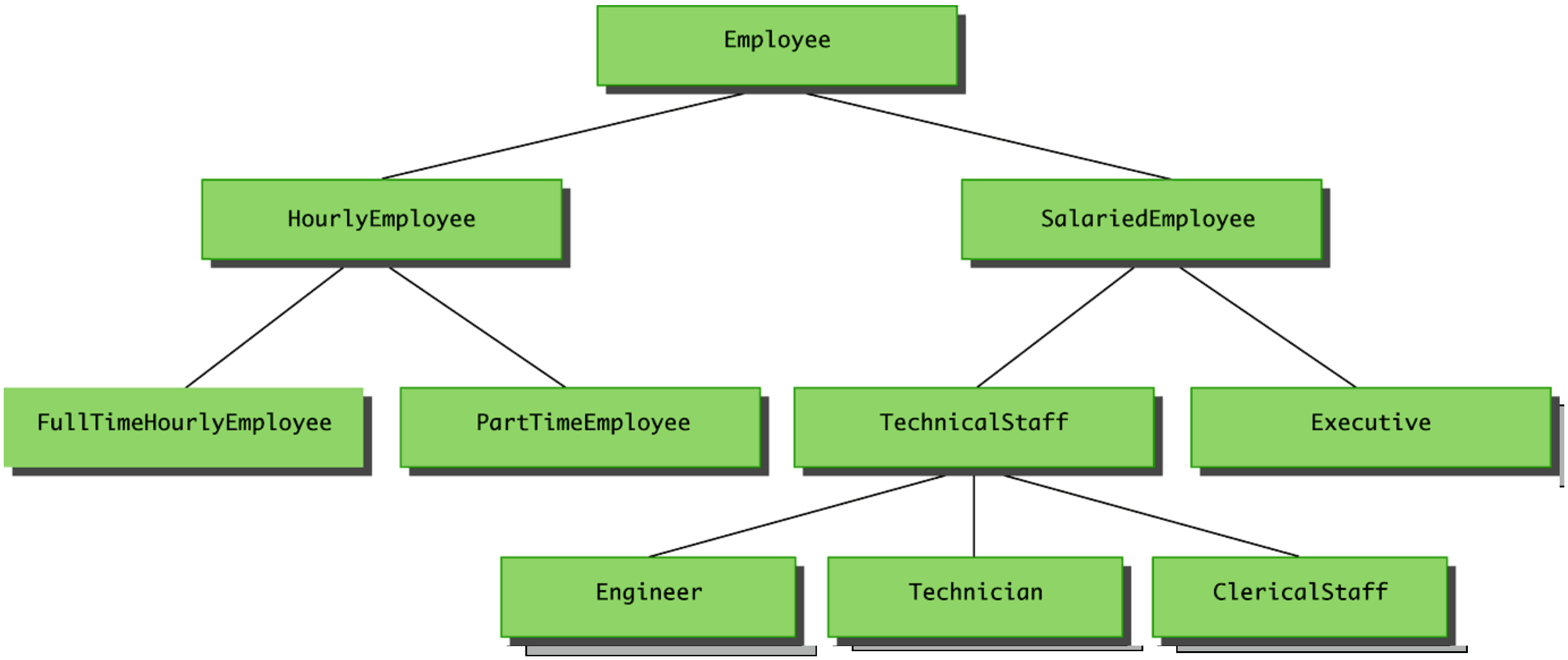
- *Inheritance* is one of the main techniques of object-oriented programming (OOP)
- Using this technique, further classes can be created from existing ones; those classes are said to *inherit* the methods and instance variables of the class they inherited
  - The new class is called a *derived class*
  - The original class is called the *base class*
- Advantage: Reusing existing code

# Derived Classes

- When designing certain classes, there is often a natural hierarchy for grouping them
  - For instance, for the employees of a company, there are hourly employees and salaried employees
  - Hourly employees can be divided into full time and part time workers
  - Salaried employees can be divided into those on technical staff, and those on the executive staff

# A Class Hierarchy

**Display 7.1 A Class Hierarchy**



# Derived Classes

- Since an hourly employee is an employee, it is defined as a *derived* class of the class **`Employee`**
  - A *derived class* is defined by adding instance variables and methods to an existing class
  - The existing class that the derived class is built upon is called the *base class*
  - The phrase **`extends BaseClass`** must be added to the derived class definition:

```
public class HourlyEmployee extends Employee
```

  - ***See Inherit1.java***

# Derived Classes

- Derived classes (also referred to as *subclasses*) inherit all instance variables and methods of the base class (also referred to as *superclass*).
- Any object of a derived class can invoke one of these parent methods, just like any of its own methods
- The derived class can add more instance variables, static variables, and/or methods
- ***See Inherit2.java***

# Parent and Child Classes

- A base class is often called the *parent class*
  - A derived class is then called a *child class*
- These relationships are often extended such that a class that is a parent of a parent . . . of another class is called an *ancestor class*
  - If class **`A`** is an ancestor of class **`B`**, then class **`B`** can be called a *descendent* of class **`A`**

# Overriding a Method Definition

- Although a derived class inherits methods from the base class, it can change or *override* an inherited method if necessary
  - In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class
  - ***See Inherit3.java***

# Changing the Return Type of an Overridden Method

- Ordinarily, the type returned may not be changed when overriding a method
- However, if it is a class type, then the returned type may be changed to that of any descendent class of the returned type
- This is known as a *covariant return type*
  - *Covariant return types* are new in Java 5.0; they are not allowed in earlier versions of Java

# Covariant Return Type

- Given the following base class:

```
public class BaseClass
{ . . .
   public Employee getSomeone(int someKey)
   . . .
```

- The following is allowed in Java 5.0:

```
public class DerivedClass extends BaseClass
{ . . .
   public HourlyEmployee getSomeone(int
   someKey)
   . . .
```

# Changing the Access Permission of an Overridden Method

- The access permission of an overridden method can be changed from private in the base class to public (or some other more permissive access) in the derived class
- However, the access permission of an overridden method can not be changed from public in the base class to a more restricted access permission in the derived class

# Changing the Access Permission of an Overridden Method

- Given the following method header in a base case:

```
private void doSomething()
```

- The following method header is valid in a derived class:

```
public void doSomething()
```

- However, the opposite is not valid
- Given the following method header in a base case:

```
public void doSomething()
```

- The following method header is not valid in a derived class:

```
private void doSomething()
```

# Pitfall: Overriding Versus Overloading

- Do not confuse *overriding* with *overloading*
  - When a method is overridden, the new method definition given in the derived class has the exact same number and types of parameters as in the base class
  - When a method in a derived class has a different signature from the method in the base class, that is overloading
  - Note that when the derived class overloads the original method, it still inherits the original method from the base class as well

# The `final` Modifier

- If the modifier **`final`** is placed before the definition of a *method*, then that method may not be overridden in a derived class
- It the modifier **`final`** is placed before the definition of a *class*, then that class may not be used as a base class to derive other classes
- ***See Inherit4.java***

# The `super` Constructor

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class
  - In order to invoke a constructor from the base class, it uses a special syntax:

```
public derivedClass(int p1, int p2, double
p3)
{
   super(p1, p2);
   instanceVariable = p3;
}
```

  - In the above example, **`super(p1, p2);`** is a call to the base class constructor

  - ***See Inherit5.java***

# The `super` Constructor

- A call to the base class constructor can never use the name of the base class, but uses the keyword **`super`** instead
- A call to **`super`** must always be the first action taken in a constructor definition
- Notice that if **`super`** is not used, then a call to the default constructor of the base class is automatically issued
- Consequently, a compilation error would occur if the base class has no default constructor

- ***See Inherit6.java***
- ***See Inherit7.java***

# The `this` Constructor

- Within the definition of a constructor for a class, **`this`** can be used as a name for invoking another constructor of the same class
  - The same restrictions on how to use a call to **`super`** apply to the **`this`** constructor
- If it is necessary to include a call to both **`super`** and **`this`**, the call using **`this`** must be made first, and then the constructor that is called must call **`super`** as its first action
- ***See Inherit8.java***

# The `this` Constructor

- Often, a no-argument constructor uses **`this`** to invoke an explicit-value constructor
  - No-argument constructor (invokes explicit-value constructor using **`this`** and default arguments):

    ```
    public ClassName()
    {
       this(argument1, argument2);
    }
    ```

  - Explicit-value constructor (receives default values):

    ```
    public ClassName(type1 param1, type2 param2)
    {
       . . .
    }
    ```

# The this Constructor

Example:

```
public HourlyEmployee()
{
   this("No name", new Date(), 0, 0);
}
```

- The above constructor will cause the constructor with the following heading to be invoked:

```
public HourlyEmployee(String theName,
   Date theDate, double theWageRate, double
   theHours)
```

# Tip: An Object of a Derived Class Has More than One Type

- An object of a derived class has the type of the derived class, and it also has the type of the base class
- More generally, an object of a derived class has the type of every one of its ancestor classes
  - Therefore, an object of a derived class can be assigned to a variable of any ancestor type
- An object of a derived class can be plugged in as a parameter in place of any of its ancestor classes
- In fact, a derived class object can be used anyplace that an object of any of its ancestor types can be used
- Note, however, that this relationship does not go the other way
  - An ancestor type can never be used in place of one of its derived types

- ***See Inherit9.java***

# Pitfall: The Terms "Subclass" and "Superclass"

- The terms *subclass* and *superclass* are sometimes mistakenly reversed
  - A superclass or base class is more general and inclusive, but less complex
  - A subclass or derived class is more specialized, less inclusive, and more complex
    - As more instance variables and methods are added, the number of objects that can satisfy the class definition becomes more restricted