`protected` members

- To allow subclass methods to access a superclass field, define it `protected`. But be cautious!
- Making methods `protected` makes more sense, if the subclasses can be trusted to use the method correctly, but other classes cannot.

What `protected` really means

Precisely, a `protected` member is accessible

- Within the class itself
- within code in the same package
- it can also be accessed from a class through object references that are of at **least the same type as the class** – that is , references of the class's type or one of its subtypes

What `protected` really means

```
 public class Employee {
      protected Date hireDay;
      . . .
}

 public class Manager extends Employee {
      . . .
      public void printHireDay (Manager p) {
        System.out.println("mHireDay: " +
                               (p.hireDay).toString());
      }
      // ok! The class is Manager, and the object reference type is also Manager



       public void printHireDay (Employee p) {
         System.out.println("eHireDay: " +
                         (p.hireDay).toString());
      }
      // wrong! The class is Manager, but the object reference type is Employee
      // which is a supertype of Manager
      . . .
}
```

super example

```
public class BankAccount {
  private double myBalance;
  public BankAccount() { myBalance = 0; }
  public double getBalance() { return myBalance;
  }
  public void withdraw(double amount) {
    myBalance -= amount;}
   public String toString() {
    return getID() + " $" + getBalance();
} }

public class FeeAccount
                    extends BankAccount {
  public void withdraw(double amount) {
    super.withdraw(amount);
    if (getBalance() < 100.00)
      super.withdraw(2.00); // charge $2 fee
} }
```

Which method gets called?

```
BankAccount b = new FeeAccount("Ed", 9.00);
b.withdraw(5.00);
System.out.println(b.getBalance());
```

- Will it call the `withdraw` method in `BankAccount`, leaving Ed with $4?

- Will it call the `withdraw` method in `FeeAccount`, leaving Ed with $2 (after his $2 fee)?

The answer: dynamic binding

- The version of `withdraw` from `FeeAccount` will be called
- The version of an object's method that gets executed is always determined by that object's type, *not* by the type of the variable
- The variable should only be looked at to determine whether the code would compile; after that, all behavior is determined by the object itself

Static and Dynamic Binding

- **static binding**: methods and types that are hard-wired at compile time
  - static methods
  - referring to instance variables
  - the types of the reference variables you declare

- **dynamic binding**: methods and types that are determined and checked as the program is running
  - non-static (a.k.a virtual) methods that are called
  - types of objects that your variables refer to

Polymorphism

- inheritance provides a way to achieve polymorphism in Java
- **polymorphism**: the ability to use identical syntax on different data types, causing possibly different underlying behavior to execute
  - example: If we have a variable of type `BankAccount` and call `withdraw` on it, it might execute the version that charges a fee, or the version from the checking account that tallies interest, or the regular version, depending on the type of the actual object.

Type-casting and objects

- You cannot call a method on a reference unless the reference's type has that method

```
Object o = new BankAccount("Ed",9.00);
o.withdraw(5.00);  // doesn't compile
```

- You can cast a reference to any subtype of its current type, and this will compile successfully

```
((BankAccount)o).withdraw(5.00);
```

Converting Between Subclass and Superclass Types

- Occasionally you need to convert from a superclass reference to a subclass reference

```
BankAccount anAccount = (BankAccount) anObject;
```

- This cast is dangerous: if you are wrong, an exception is thrown

The `instanceof` keyword

- Performs run-time type check on the object referred to by a reference variable
- Usage: *object-reference* `instanceof` *type* (result is a boolean expression)
  - if *type* is a class, evaluates to **true** if the variable refers to an object of *type* or any subclass of it.
  - if *type* is an interface, evaluates to true if the variable refers to an object that implements that interface.
  - if ***object-reference* is null, the result is false**.
- Example:

```
Object o = myList.get(2);
if (o instanceof BankAccount)
   ((BankAccount)o).deposit(10.0);
```

Converting Between Subclass and Superclass Types

- Solution: use the `instanceof` operator
- `instanceof`: tests whether an object belongs to a particular type

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

Down-casting and runtime

- It is illegal to cast a reference variable into an unrelated type (example: casting a `String` variable into a `BankAccount`)

- It is legal to cast a reference to the wrong subtype; this will compile but crash when the program runs
  - Will crash even if the type you cast it to has the method in question

```
((String)o).toUpperCase();        // crashes
((FeeAccount)o).withdraw(5.00);   // crashes
```

Some `instanceof` problems

```
Object o = new BankAccount(...);
BankAccount c = new CheckingAccount(...);
BankAccount n = new NumberedAccount(...);
CheckingAccount c2 = null;
                                                  T/F
  ???
o instanceof Object
  _______
o instanceof BankAccount
  _______
o instanceof CheckingAccount
  _______
c instanceof BankAccount
  _______
c instanceof CheckingAccount
```

A dynamic binding problem

```
class A {
  public void method1() { System.out.println("A1"); }
  public void method3() { System.out.println("A3"); }
}

class B extends A {
  public void method2() { System.out.println("B2"); }
  public void method3() { System.out.println("B3"); }
}

A var1 = new B();
Object var2 = new A();

var1.method1();____________
var1.method2();___________
var2.method1();____________
((A)var2).method3();_______________
```

OUTPUT???

The Object & Class Classes

Object Class : Top class in Java

equals() method

boolean equals(Object obj)

getClass() method

Class getClass()

toString() method

String toString()

Class Class

getName() method

String getName()

getSuperClass() method

Class getSuperClass()

forName() method

Static Class forName (String cIsName) throws ClassNotFoundException

The Object & Class Classes

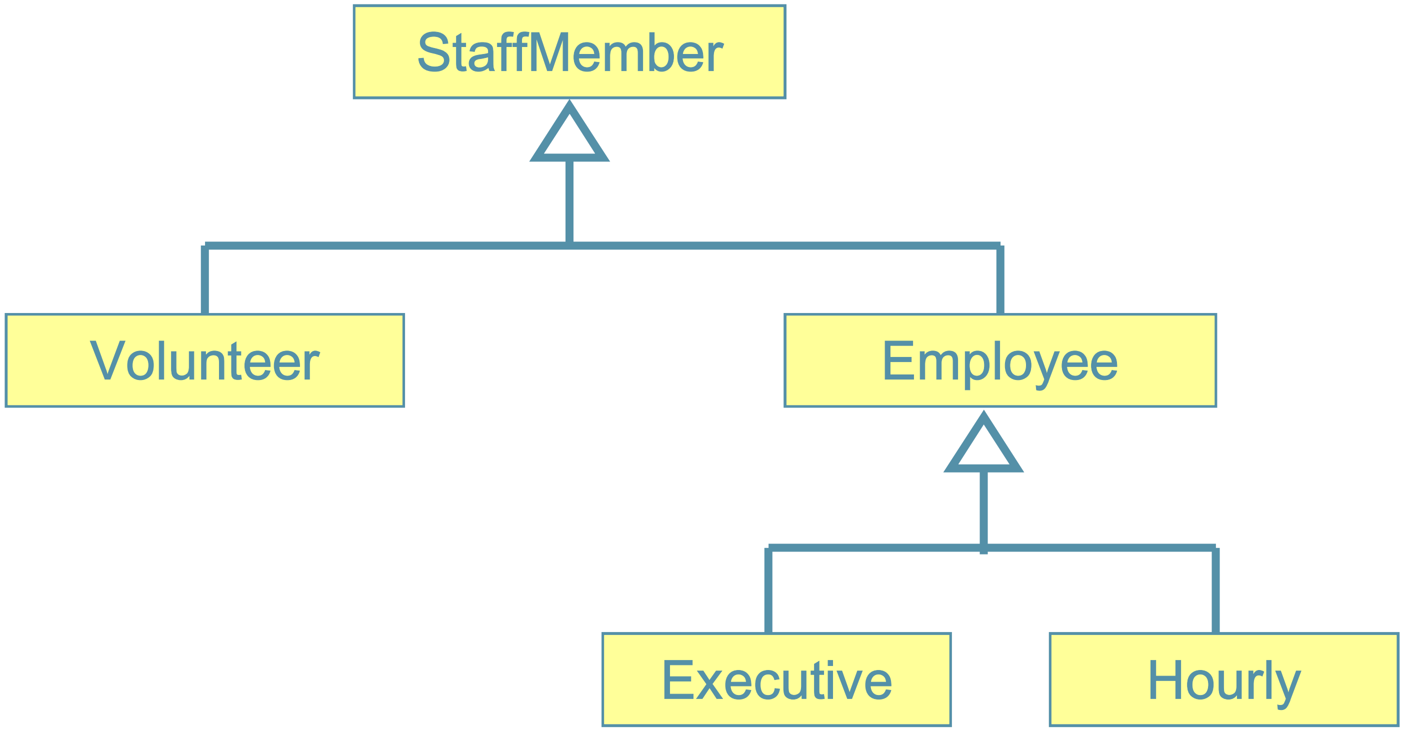
```
class ClassDemo {

  public static void main(String args[]) {

    Integer obj = new Integer(8);
    Class cls = obj.getClass();
    System.out.println(cls);
  }
}
```

Result :

```
Class
java.lang.Integer
```

A payment problem

- **Consider the following class hierarchy: How to pay all staff members using their pay() methods.**


StaffMember
Volunteer
Employee
Executive
Hourly