CS 10: Problem solving via Object Oriented Programming

Inheritance

Agenda

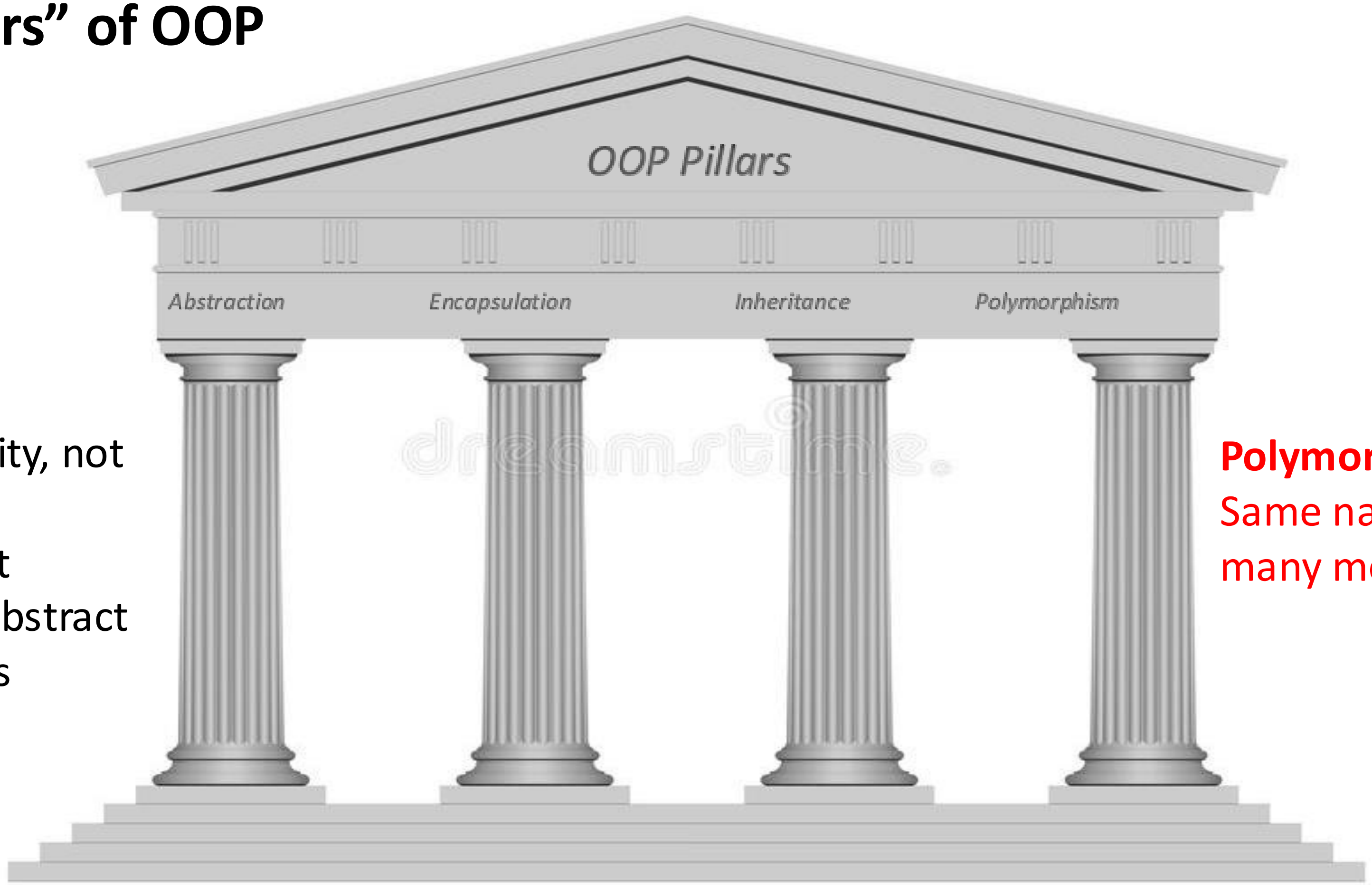
1. Inheritance
2. Comparing objects
3. “Is a” example
4. Access modifiers

Key points:

1. **Create and debug base class**
2. **Create specialty versions of the of the base class (called subclasses) that inherit the code and data from the base class**
3. **Use the keyword “extends” to inherit from the base class**
4. **In Java we can only inherit from one base class (unlike C++)**

OOP relies on four main pillars to create robust, adaptable, and reusable code

Four “pillars” of OOP



Abstraction

- Name functionality, not how to implement
- Leads to Abstract Data Types (ADTs)

Polymorphism

Same name, many meanings

Encapsulation

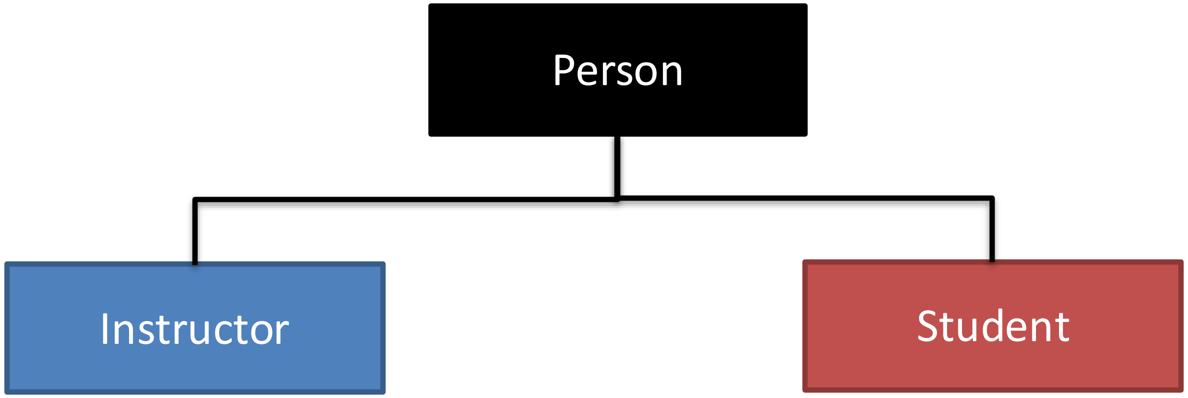
- Bind code and data into one thing called an object
- Code called methods in OOP (not functions)

Inheritance

- Create specialty versions that “inherit” functionality of parent
- Reduces code

Inheritance allows us to reuse code that has already been written and debugged

College application



- In a college application, instructors and students are both people
- As people, there are some things that are common groups
  - Name
  - ID
- We can create a Person class as a “Base class”
- After debugging the person class, we can reuse the code from the base class to create specialty “subclasses” that inherit the instance variables and methods of the base class
- Subclasses can *override* the methods of the base class

The Person base class has instance variables and methods

Person.java

```
public class Person {
   String name;
   String id;

   public Person(String name, String id) {
      this.name = name;
      this.id = id;
   }

   public String getName() { return name; }
   public String getId() { return id; }

  public void setName(String name) {this.name = name; }
   public void setId(String id) { this.id = id;}

   /**
    * Returns a String representation of a Person
    * @return String
    */
   public String toString() {
      String s = "Name: " + name + " (" + id + ")";
      return s;
   }
}
```

Remember: by convention, class names start with capital letter; variable and method names use camelCase (not snake_case like Python or C)
I'll be looking for you to follow this convention

Simple constructor saves name and id

- **Getter and setter methods (note: JavaDoc removed to fit on slide)**
- **Could have other methods that do more complicated things**
- **Here we keep it simple**

Remember: toString returns a String!

Here we add name and id

Don't forget to return the String

Subclasses inherit the instance variables and methods of the base class

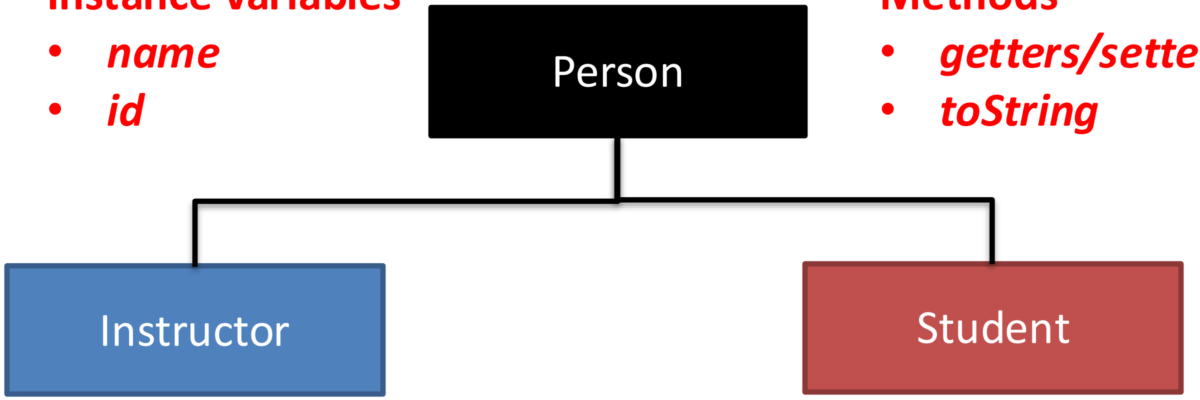
College application

Instance variables

- ***name***
- ***id***

Methods

- ***getters/setters***
- ***toString***

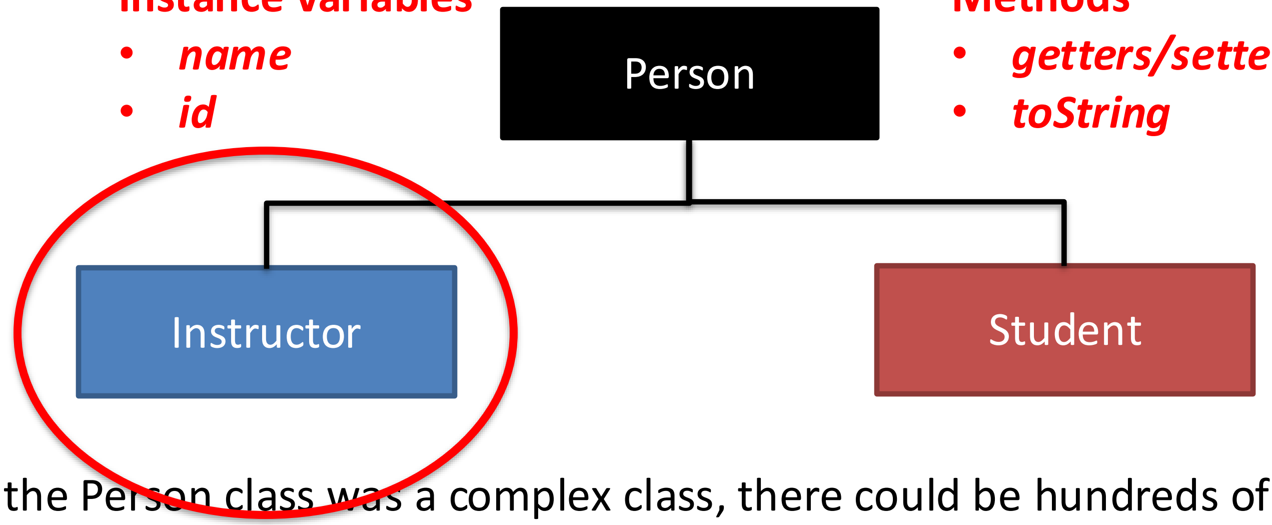


- If the Person class was a complex class, there could be hundreds of lines of code
- No sense duplicating that code
- With inheritance subclasses get the instance variables and methods already written and debugged in the base class
- Ever heard of DRY?
- Don’t Repeat Yourself!
- Duplicating code causes problems if you later make a change
- In that case you must remember to change the code everywhere it is duplicated
- With inheritance, changes in the base class are automatically inherited in subclasses
- An Instructor “is a” Person. A Student “is a” Person too! They are just specialty versions

Note: base class, super class, and parent class all mean the same thing!

Subclasses inherit the instance variables and methods of the base class

College application



Instance variables

- ***name***
- ***id***

Methods

- ***getters/setters***
- ***toString***

- If the Person class was a complex class, there could be hundreds of lines of code
- No sense duplicating that code
- With inheritance subclasses get the instance variables and methods already written and debugged in the base class
- Ever heard of DRY?
- Don't Repeat Yourself!
- Duplicating code causes problems if you later make a change
- In that case you must remember to change the code everywhere it is duplicated
- With inheritance, changes in the base class are automatically inherited in subclasses
- An Instructor "is a" Person. A Student "is a" Person too! They are just specialty versions

Note: base class, super class, and parent class all mean the same thing!

Use “extends” to inherit instance variables and methods from base class

Instructor.java

```
public class Instructor extends Person {
   boolean tenured;
   int yearsEmployed;
   String department;

   public Instructor(String name, String id) {
      super(name, id);
      this.tenured = false;   //not required, Java initializes boolean instance variables to false
      this.yearsEmployed = 0; //not required, Java initializes numeric values instance variables to 0
      this.department = null;  //not required, Java initializes objects to null
   }
   public Instructor(String name, String id, boolean tenured, int yearsEmployed, String department) {
      super(name, id);
      this.tenured = tenured;
      this.yearsEmployed = yearsEmployed;
      this.department = department;
   }
```

- **“extends” keyword tells Java this class inherits Person’s instance variables and methods**
- **Note: no *name* and *id* instance variables declared here, but Instructor has them due to “extends”**

Instructors have additional instance variables that the base class Person does not have

Two *overloaded* constructors

- **One takes two parameters**
- **The other takes five parameters**

- ***super* calls the constructor on the base (aka super) class**
- **If the constructor in Person was complex, no need to duplicate that code, just call it**
- **Eliminates code redundancy and reduces likelihood of mistakes**
- **Get any changes made to base class by calling super, rather than duplicating code here**

Subclasses can add instance variables and methods the base class does not have

Instructor.java

```java
/**
 * Getters and setters
 */
public boolean getTenuredStatus() { return tenured;}
public int getYearsEmployed() { return yearsEmployed;}
public String getDepartment() { return department; }

public void setTenured(boolean tenured) { this.tenured = tenured; }
public void setYearsEmployed(int yearsEmployed) { this.yearsEmployed = yearsEmployed; }
public void setDepartment(String department) { this.department = department;}
```

- **Base class Person does not have instance variables**
  - ***tenured***
  - ***yearsEmployed***
  - ***department***
- **Base class also does not have getters/setters defined by subclass**

```java
/**
 * Return a String representation of an instructor
 * @return - string representing the instructor
 */
@Override
public String toString() {
   String s = super.toString() + "\n";
   s += "\tTenured: " + tenured + "\n";
   s += "\tYears Employed: " + yearsEmployed + "\n";
   s += "\tDepartment: " + department;
   return s;
}
```

- **Subclasses can change the behavior of methods defined in the base class**
- **This change is called *overriding* the base class**
- **Here *toString* is defined in the base class and also in the subclass**
- **This version adds additional information to the String returned**
- **Calling *super.toString* calls the base class method**
- **What if this code didn’t say *super*, just *toString*?**
- **Recursively this method!**

Subclasses can add instance variables and methods the base class does not have

Instructor.java

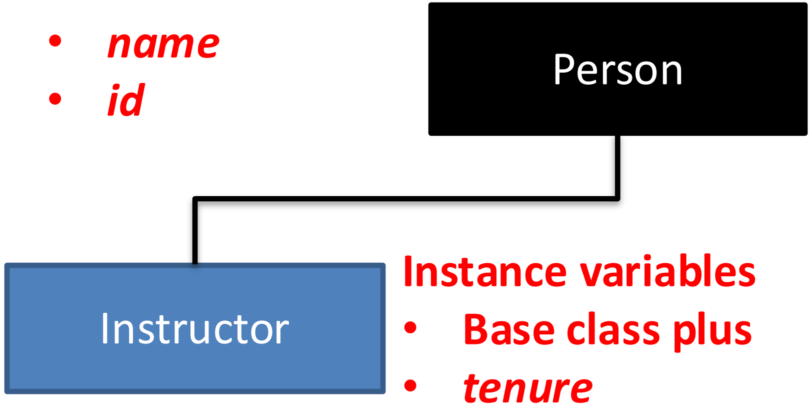
```java
/**
 * Getters and setters
 */
public boolean getTenuredStatus() { return tenured;}
public int getYearsEmployed() { return yearsEmployed;}
public String getDepartment() { return department; }

public void setTenured(boolean tenured) { this.tenured = tenured; }
public void setYearsEmployed(int yearsEmployed) { this.yearsEmployed = yearsEmployed; }
public void setDepartment(String department) { this.department = department;}

/**
 * Return a String representation of an instructor
 * @return - string representing the instructor
 */
@Override
public String toString() {
   String s = super.toString() + "\n";
   s += "\tTenured: " + tenured + "\n";
   s += "\tYears Employed: " + yearsEmployed + "\n";
   s += "\tDepartment: " + department;
   return s;
}
```

- **@Override decorator is not required**
- **Tells Java, “I intend to override the base classes method”**
- **Java will flag an exception if the method does not appear in the base class**
- **Perhaps you made a typo and wrote “toSTring” instead of “toString”**
- **If there is no “toSTring” method in the base class, Java will alert you before you run code**
- **Good habit to include @Override**

Dynamic dispatch hunts up the inheritance chain to find methods

Instance variables

- ***name***
- ***id***



Methods

- ***getters/setters* for *name* and *id***
- ***toString***

Instance variables

- **Base class plus**
- ***tenure***
- ***yearsEmployed***
- ***department***

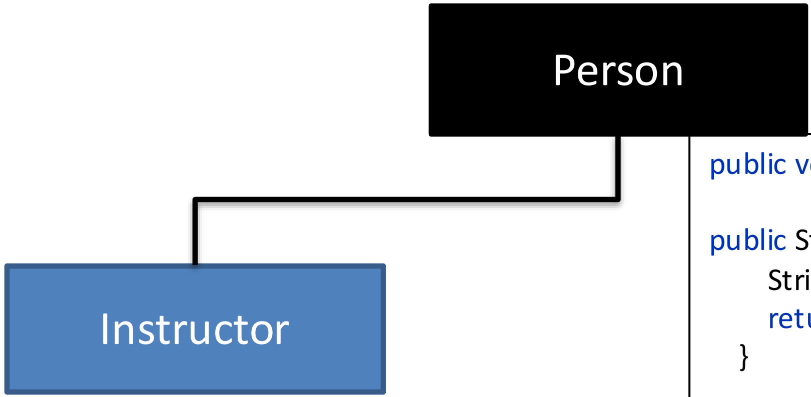
Methods

- **Base class plus**
- ***getters/setters* for new instance variables**
- **Overriden *toString***

- Calling *toString* on an Instructor object will run the Instructor's *toString* code
- Calling *toString* on a Person object will run the Person's *toString* code
- If a method is called on subclass that the subclass does not define, Java hunts up the inheritance chain to look for the method
- For example, *setName* is not defined by Instructor, so calling it on an *Instructor* object will cause Java to first examine the *Instructor* class, when that method is not found, it will check the base class
- In this case *setName* is defined on the base class, so Java will run that code
- This hunting upward is called *dynamic dispatch*
- If the method is never found after hunting upward, Java will throw an exception

Dynamic dispatch hunts up the inheritance chain to find methods

DynamicDispatchExample.java



```
public void setName(String name) {this.name = name; }

public String toString() {
      String s = "Name: " + name + " (" + id + ")";
      return s;
   }
```

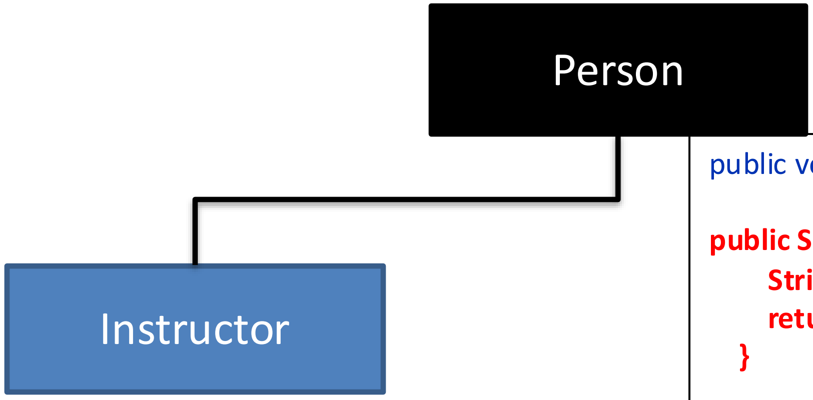
```
@Override
public String toString() {
   String s = super.toString() + "\n";
   s += "\tTenured: " + tenured + "\n";
   s += "\tYears Employed: " + yearsEmployed + "\n";
   s += "\tDepartment: " + department;
   return s;
}
```

```
Person alice = new Person("Alice", "f00xzy");
Instructor bob = new Instructor("Bob","f00abc");
```

Declare two objects *Person alice* and *Instructor bob*

Dynamic dispatch starts at the class the object was declared, runs method if found

DynamicDispatchExample.java



Look for *toString* here

Found! Run this code

```
public void setName(String name) {this.name = name; }

public String toString() {
      String s = "Name: " + name + " (" + id + ")";
      return s;
   }
```

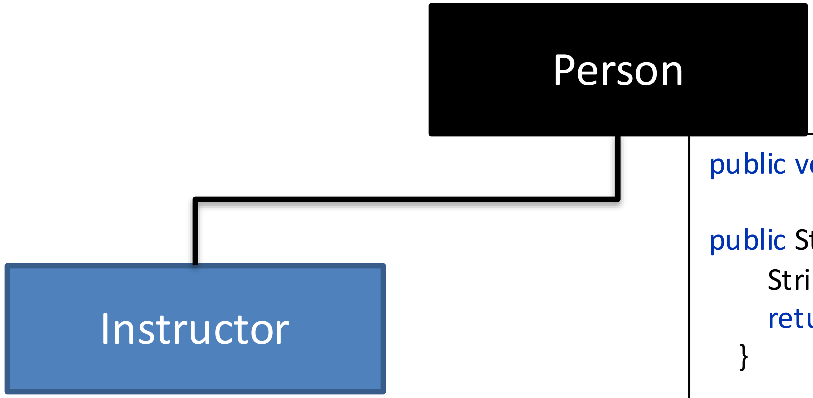
```
@Override
public String toString() {
   String s = super.toString() + "\n";
   s += "\tTenured: " + tenured + "\n";
   s += "\tYears Employed: " + yearsEmployed + "\n";
   s += "\tDepartment: " + department;
   return s;
}
```

```
Person alice = new Person("Alice", "f00xzy");
Instructor bob = new Instructor("Bob","f00abc");
System.out.println(alice);
```

- **Printing Person object *alice* calls *toString* behind the scenes**
- ***Person* class *toString* runs because *alice* is declared as a Person object**
- **NOTE: this is an example of Polymorphism (same name, many meanings)**
- **Same name *toString*, different results**

Dynamic dispatch hunts up the inheritance chain if method is not found

DynamicDispatchExample.java



```
public void setName(String name) {this.name = name; }

public String toString() {
        String s = "Name: " + name + " (" + id + ")";
        return s;
   }
```

```
@Override
public String toString() {
   String s = super.toString() + "\n";
   s += "\tTenured: " + tenured + "\n";
   s += "\tYears Employed: " + yearsEmployed + "\n";
   s += "\tDepartment: " + department;
   return s;
}
```

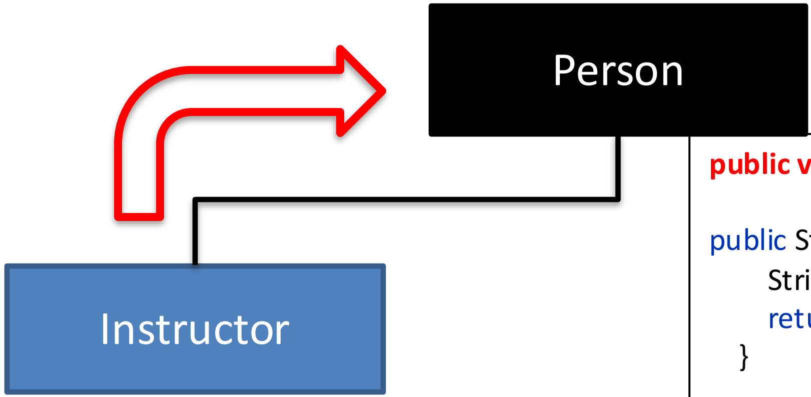
First look for *setName* here
Not found
Check base class

```
Person alice = new Person("Alice", "f00xzy");
Instructor bob = new Instructor("Bob","f00abc");
System.out.println(alice);
bob.setName("Bobby");
```

- **Call *setName* on *Instructor bob***
- **Instructor does not define *setName***

Dynamic dispatch hunts up the inheritance chain if method is not found

DynamicDispatchExample.java



Second, look for *setName* here
Found! Run this code

```
public void setName(String name) {this.name = name; }

public String toString() {
      String s = "Name: " + name + " (" + id + ")";
      return s;
   }
```

```
@Override
public String toString() {
   String s = super.toString() + "\n";
   s += "\tTenured: " + tenured + "\n";
   s += "\tYears Employed: " + yearsEmployed + "\n";
   s += "\tDepartment: " + department;
   return s;
}
```

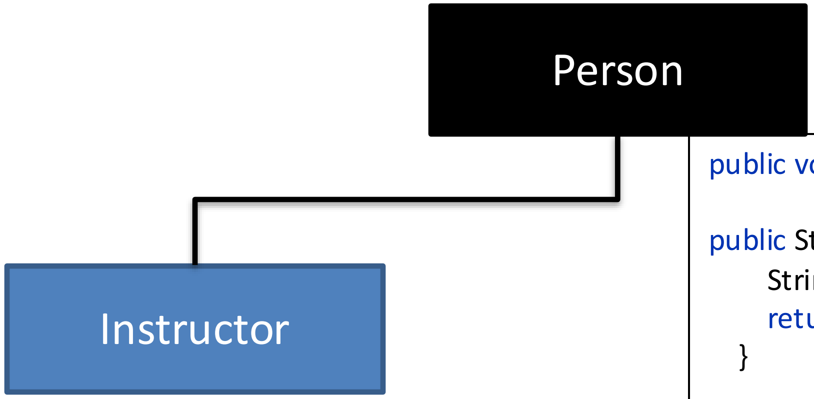
First, look for *setName* here
Not found
Check base class

```
Person alice = new Person("Alice", "f00xzy");
Instructor bob = new Instructor("Bob","f00abc");
System.out.println(alice);
bob.setName("Bobby");
```

- **Call *setName* on *Instructor bob***
- **Instructor does not define *setName***

Run subclass code if a method is overriden

DynamicDispatchExample.java



```
public void setName(String name) {this.name = name; }

public String toString() {
      String s = "Name: " + name + " (" + id + ")";
      return s;
   }
```

```
@Override
public String toString() {
   String s = super.toString() + "\n";
   s += "\tTenured: " + tenured + "\n";
   s += "\tYears Employed: " + yearsEmployed + "\n";
   s += "\tDepartment: " + department;
   return s;
}
```

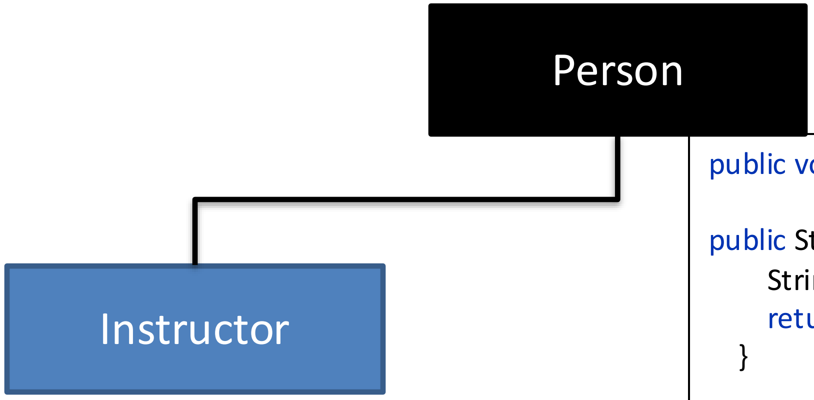
First look for *toString* here
Found!
Run this code

```
Person alice = new Person("Alice", "f00xzy");
Instructor bob = new Instructor("Bob","f00abc");
System.out.println(alice);
bob.setName("Bobby");
System.out.println(bob);
```

Printing Instructor *bob*
Instructor class overrides *toString*
Use the most specific method
Here use Instructor's *toString* method

Dynamic dispatch starts at the class the object was declared, runs method if found

DynamicDispatchExample.java



```
public void setName(String name) {this.name = name; }

public String toString() {
      String s = "Name: " + name + " (" + id + ")";
      return s;
   }
```

```
@Override
public String toString() {
   String s = super.toString() + "\n";
   s += "\tTenured: " + tenured + "\n";
   s += "\tYears Employed: " + yearsEmployed + "\n";
   s += "\tDepartment: " + department;
   return s;
}
```

```
Person alice = new Person("Alice", "f00xzy");
Instructor bob = new Instructor("Bob","f00abc");
System.out.println(alice);
bob.setName("Bobby");
System.out.println(bob);
```

Output

```
Name: Alice (f00xzy)
Name: Bobby (f00abc)
     Tenured: false
     Years Employed: 0
     Department: null
```

Name changed to Bobby by Person's *setName*

From Instructor *toString*

Multiple classes can inherit the same base class, each providing a specialty version

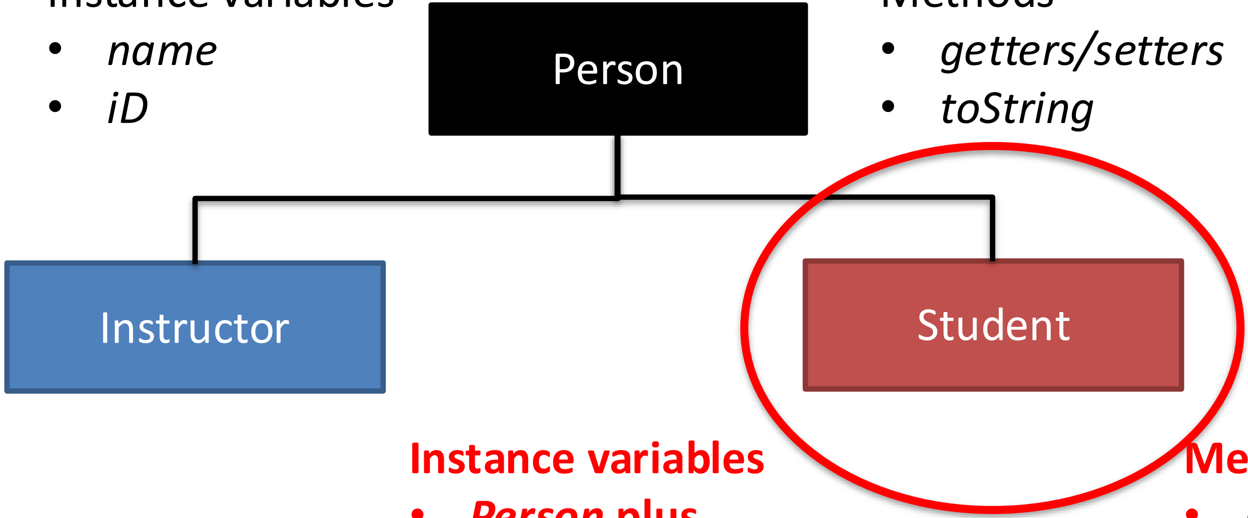
College application

Instance variables

- *name*
- *iD*

Methods

- *getters/setters*
- *toString*



Instance variables

- ***Person* plus**
- ***graduationYear***
- ***studyHours***
- ***classHours***

Methods

- ***Person* plus**
- ***New getters/setters***
- ***study***
- ***attendClass***

The Student class also inherits from the Person class, but behaves differently

Student.java

```
public class Student extends Person {
    protected Integer graduationYear;
    double studyHours;
    double classHours;

    public Student(String name, String id) {
        super(name, id);
        graduationYear = null;
        studyHours = 0;
        classHours = 0;
    }

    public double study(double hoursSpent) {
        System.out.println("Hi Mom! It's " + name + ". I'm studying!");
        studyHours += hoursSpent;
        return studyHours;
    }

    public double attendClass(double hoursSpent) {
        System.out.println("Hi Dad! It's " + name +". I'm in class!");
        classHours += hoursSpent;
        return classHours;
    }

    @Override
    public String toString() {
        String s = super.toString() + "\n";
        s += "\tGraduation year: " + graduationYear + "\n";
        s += "\tHours studying: " + studyHours + "\n";
        s += "\tHours in class: " + classHours;
        return s;
    }
```

By using *extends*, Students have name and id from Person, just like Instructors got them by using *extends*

But, Students have additional information

- ***graduationYear***
- ***studyHours***
- ***classHours***

- **Students also have methods Persons and Instructors do not have**
  - ***study***
  - ***attendClass***
- **Student also *overrides toString* so output is different for Students than for Persons and Instructors**

The Student class also inherits from the Person class, but behaves differently

Student.java

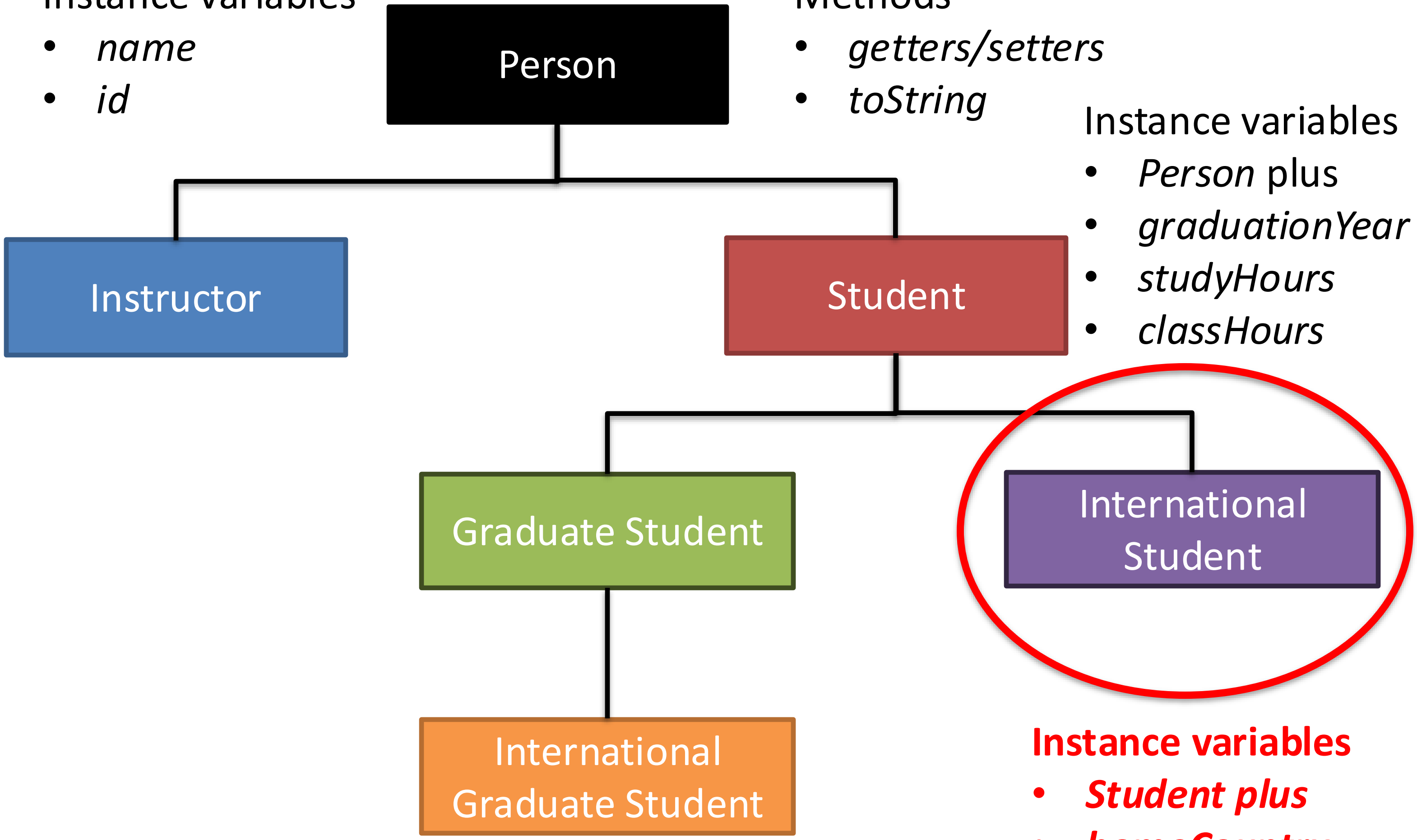
```
public class Student extends Person {
   protected Integer graduationYear;
   double studyHours;
   double classHours;

   public Student(String name, String id) {
      super(name, id);
      graduationYear = null;
      studyHours = 0;
      classHours = 0;
   }

   public double study(double hoursSpent) {
      System.out.println("Hi Mom! It's " + name + ". I'm studying!");
      studyHours += hoursSpent;
      return studyHours;
   }

   public double attendClass(double hoursSpent) {
      System.out.println("Hi Dad! It's " + name +". I'm in class!");
      classHours += hoursSpent;
      return classHours;
   }

   @Override
   public String toString() {
      String s = super.toString() + "\n";
      s += "\tGraduation year: " + graduationYear + "\n";
      s += "\tHours studying: " + studyHours + "\n";
      s += "\tHours in class: " + classHours;
      return s;
   }
```

Note: *graduationYear* is of type Integer (autoboxed version of primitive data type) so we can set it to null (instead of 0) if we do not have a value

Otherwise *graduationYear* would be 0 by default, but in some cases 0 might be a valid value

Null is different from 0, it indicates that we do not have a value

Classes can inherit from other inherited classes, forming a chain



Person

Instructor

Student

Graduate Student

International Student

International Graduate Student

Instance variables

- *name*
- *id*

Methods

- *getters/setters*
- *toString*

Instance variables

- *Person* plus
- *graduationYear*
- *studyHours*
- *classHours*

Instance variables

- ***Student plus***
- ***homeCountry***

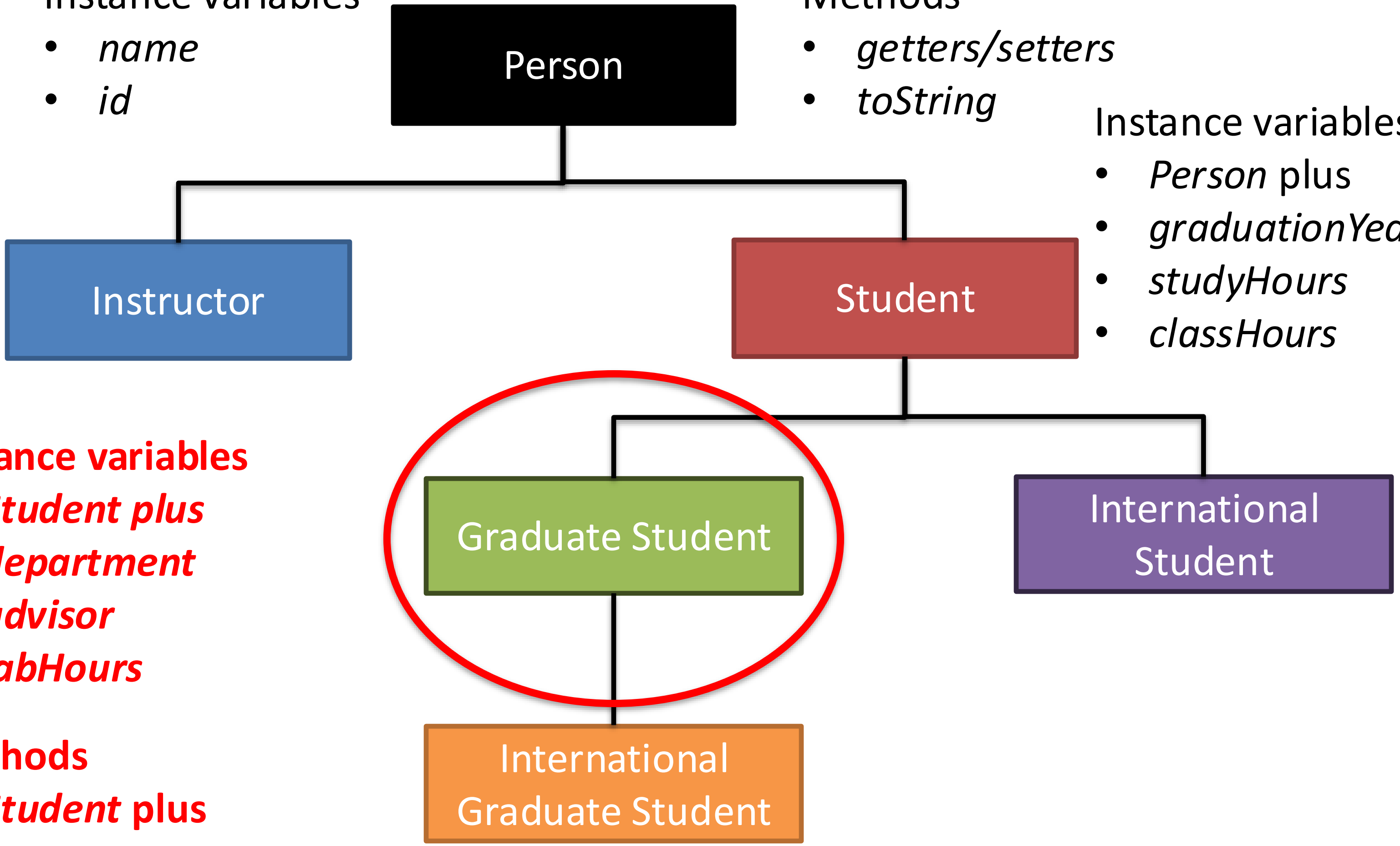
Classes can inherit from other inherited classes, forming a chain

Instance variables

- *name*
- *id*

Methods

- *getters/setters*
- *toString*



Instance variables

- *Person* plus
- *graduationYear*
- *studyHours*
- *classHours*

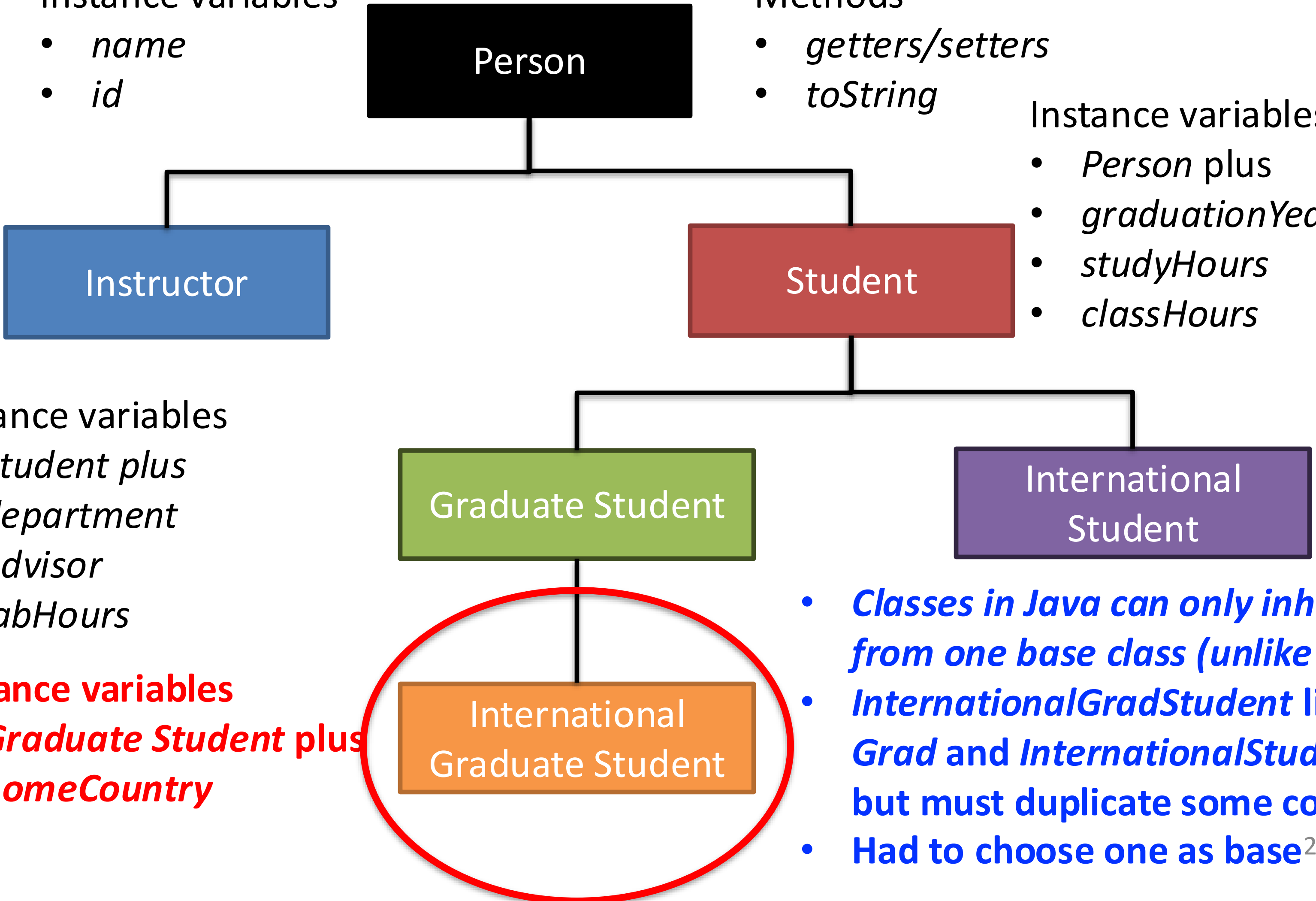
Instance variables

- ***Student plus***
- ***department***
- ***advisor***
- ***labHours***

Methods

- ***Student* plus**
- ***experiment***

Classes can inherit from other inherited classes, forming a chain



Instance variables

- *name*
- *id*

Methods

- *getters/setters*
- *toString*

Instance variables

- *Person* plus
- *graduationYear*
- *studyHours*
- *classHours*

Instance variables

- *Student plus*
- *department*
- *advisor*
- *labHours*

Instance variables

- ***Graduate Student* plus**
- ***homeCountry***

- ***Classes in Java can only inherit from one base class (unlike C++)***
- ***InternationalGradStudent* like *Grad* and *InternationalStudent*, but must duplicate some code**
- **Had to choose one as base**

Inheritance summary

By simply adding “extends”, a subclass gets all (non-private) base class:

- Instance variables (no need to redefine name and id)
- Methods

Subclass can *override* base class method to create specialty versions

- Give same method name in the subclass as in the base class
- Java will run the subclass’s method when called
- Subclass method can call base class method super.<methodName>
- Dynamic dispatch hunts upward if subclass does not define method

Inheritance reduces duplicate code

- Just use the code written and debugged for the base class
- Changing base class updates subclass

Agenda

1. Inheritance
2. Comparing objects
3. "Is a" example
4. Access modifiers

Key points:

1. **Compare primitive types with ==**
2. **Compare objects with *equals* method**

Use == when comparing primitives

CompareTest.java

```
public class CompareTest {
   public static void main(String[] args) {
      int a = 7;
      int b = 5;
      System.out.println("Check primitive variables");
      System.out.println("a=" + a + " b=" + b + " same:" + (a==b));
```

***a* and *b* are not equal**

Output

```
Check primitive variables
a=7 b=5 same:false
```

Use == when comparing primitives

CompareTest.java

```
public class CompareTest {
  public static void main(String[] args) {
    int a = 7;
    int b = 5;
    System.out.println("Check primitive variables");
    System.out.println("a=" + a + " b=" + b + " same:" + (a==b));
    b = 7;    a and b are now equal
    System.out.println("a=" + a + " b=" + b + " same:" + (a==b));
```

Output

```
Check primitive variables
a=7 b=5 same:false
a=7 b=7 same:true
```

Using == when comparing objects checks to see if they reference the same address

CompareTest.java

```java
public class CompareTest {
    public static void main(String[] args) {
        int a = 7;
        int b = 5;
        System.out.println("Check primitive variables");
        System.out.println("a=" + a + " b=" + b + " same:" + (a==b));
        b = 7;
        System.out.println("a=" + a + " b=" + b + " same:" + (a==b));

        System.out.println("Check object variables");
        Person alice = new Person("Alice","f00abc");
        Person ally = alice;
        System.out.println("alice == ally: " + (alice==ally));
```

Output

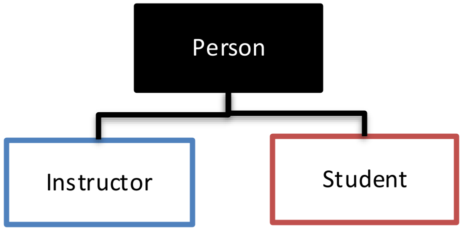
```
Check primitive variables
a=7 b=5 same:false
a=7 b=7 same:true

Check object variables
alice equals ally: true
```

***alice* and *ally* point to the same memory address on the heap**

== compares memory addresses and decides they are the same
(if yes, they are exactly the same memory location on the heap!)

The right way to compare equality of objects is the *equals* method

Person.java

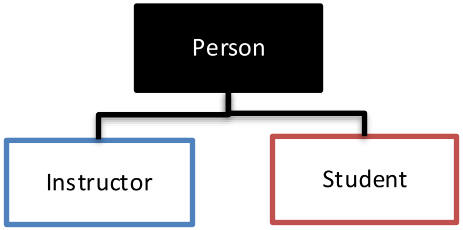


```
/**
 * Comare two Person objects and decide if they are the same.
  * Use id to decide, assume each person has unique id
 * @param other compare this person's id
 * @return true if ids are the same, false otherwise
 */
public boolean equals(Person other) {
    if (id.length() != other.id.length()) {
        return false;
    }
    for (int i = 0; i < id.length(); i++) {
        if (id.charAt(i) != other.id.charAt(i)) {
            return false;
        }
    }
    return true;
}
```

- **Java does not know the semantic meaning of objects we create**
- **Thus, Java does not know how to compare them**
- **We can provide an *equals* method to tell Java if we consider two objects to be equal**
- **We create an *equals* method in the Person base class, all subclasses will use this method due to dynamic dispatch if they do not override *equals***
- **We use *id* to decide if two Person (or subclass) objects are equal**
- **Because IDs are Strings, we check the length and ensure each character matches**
- **Return true if same length and each character matches, false otherwise**

The right way to compare equality of objects is the *equals* method

Person.java



```
/**
 * Comare two Person objects and decide if they are the same.
  * Use id to decide
 * @param other compare this person's id
 * @return true if ids are the same, false otherwise
 */
public boolean equals(Person other) {
//       if (id.length() != other.id.length()) {
//           return false;
//       }
//       for (int i = 0; i < id.length(); i++) {
//           if (id.charAt(i) != other.id.charAt(i)) {
//               return false;
//           }
//       }
//       return true;
      return id.equals(other.id);
   }
```

- **Java has already provided an *equals* method for autoboxed types and Strings**
- **We can just use their *equals* method instead**
- **Thanks Java developers!**

The right way to compare equality of objects is the *equals* method

CompareTest.java

```
public static void main(String[] args) {
    int a = 7;
    int b = 5;
    System.out.println("Check primitive variables");
    System.out.println("a=" + a + " b=" + b + " same:" + (a==b));
    b = 7;
    System.out.println("a=" + a + " b=" + b + " same:" + (a==b));

    System.out.println("\nCheck object variables");
    Person alice = new Person("Alice","f00abc");
    Person ally = alice;
    System.out.println("alice == ally: " + (alice==ally));
    System.out.println("alice equals ally: " + alice.equals(ally));
```

Output

```
Check primitive variables
a=7 b=5 same:false
a=7 b=7 same:true

Check object variables
alice == ally: true
alice equals ally: true
```

Because *alice* and *ally* both point to the same memory address, they each have the same *id* String

***equals* returns true here**

The right way to compare equality of objects is the *equals* method

CompareTest.java

```
public static void main(String[] args) {
    int a = 7;
    int b = 5;
    System.out.println("Check primitive variables");
    System.out.println("a=" + a + " b=" + b + " same:" + (a==b));
    b = 7;
    System.out.println("a=" + a + " b=" + b + " same:" + (a==b));

    System.out.println("\nCheck object variables");
    Person alice = new Person("Alice","f00abc");
    Person ally = alice;
    System.out.println("alice == ally: " + (alice==ally));
    System.out.println("alice equals ally: " + alice.equals(ally));
    ally = new Person("Ally", "f00xyz");
    System.out.println("alice == ally: " + (alice==ally));
    System.out.println("alice equals ally: " + alice.equals(ally));
```

Output

```
Check primitive variables
a=7 b=5 same:false
a=7 b=7 same:true

Check object variables
alice == ally: true
alice equals ally: true
alice == ally: false
alice equals ally: false
```

Now *ally* is instantiated as new object (so new memory address on heap) and different id from *alice*

Both checks now return false

The right way to compare equality of objects is the *equals* method

CompareTest.java

```
public static void main(String[] args) {
    int a = 7;
    int b = 5;
    System.out.println("Check primitive variables");
    System.out.println("a=" + a + " b=" + b + " same:" + (a==b));
    b = 7;
    System.out.println("a=" + a + " b=" + b + " same:" + (a==b));

    System.out.println("\nCheck object variables");
    Person alice = new Person("Alice","f00abc");
    Person ally = alice;
    System.out.println("alice == ally: " + (alice==ally));
    System.out.println("alice equals ally: " + alice.equals(ally));
    ally = new Person("Ally", "f00xyz");
    System.out.println("alice == ally: " + (alice==ally));
    System.out.println("alice equals ally: " + alice.equals(ally));
    ally.setId("f00abc");
    System.out.println("alice == ally: " + (alice==ally));
    System.out.println("alice equals ally: " + alice.equals(ally));
```

Output

```
Check primitive variables
a=7 b=5 same:false
a=7 b=7 same:true

Check object variables
alice == ally: true
alice equals ally: true
alice == ally: false
alice equals ally: false
alice == ally: false
alice equals ally: true
```

ally **now gets same id as alice**

== false (different addresses)

equals **method true (same id)**

instanceof lets you check an object’s type

CompareTest.java

```
public static void main(String[] args) {

    <snip>

    //instanceof tests
    Instructor bob = new Instructor("Bob", "f00000");
    Student carol = new Student("Carol", "f11111");
```

```
Bob is an Instructor
Carol is a Student
```

Output

instanceof lets you check an object’s type

CompareTest.java

```
public static void main(String[] args) {

    <snip>

    //instanceof tests
    Instructor bob = new Instructor("Bob", "f00000");
    Student carol = new Student("Carol", "f11111");
    if (bob instanceof Instructor) {
        System.out.println("Bob is an instructor");
    }
    if (carol instanceof Instructor) {
        System.out.println("Carol is an instructor");
    }
}
```

Output

```
Bob is an Instructor
Carol is a Student
```

instanceof
checks type,
returns boolean

instanceof lets you check an object’s type

CompareTest.java

```
public static void main(String[] args) {

    <snip>

    //instanceof tests
    Instructor bob = new Instructor("Bob", "f00000");
    Student carol = new Student("Carol", "f11111");
    if (bob instanceof Instructor) {
        System.out.println("Bob is an instructor");
    }
    if (carol instanceof Instructor) {
        System.out.println("Carol is an instructor");
    }
}
```

Bob is an Instructor

Carol is a Student

Output

```
Bob is an instructor
```

instanceof **checks type, returns boolean**

bob prints

instanceof lets you check an object’s type

CompareTest.java

```
public static void main(String[] args) {

    <snip>

    //instanceof tests
    Instructor bob = new Instructor("Bob", "f00000");
    Student carol = new Student("Carol", "f11111");
    if (bob instanceof Instructor) {
        System.out.println("Bob is an instructor");
    }
    if (carol instanceof Instructor) {
        System.out.println("Carol is an instructor");
    }
}
```

Bob is an Instructor
Carol is a Student

Output

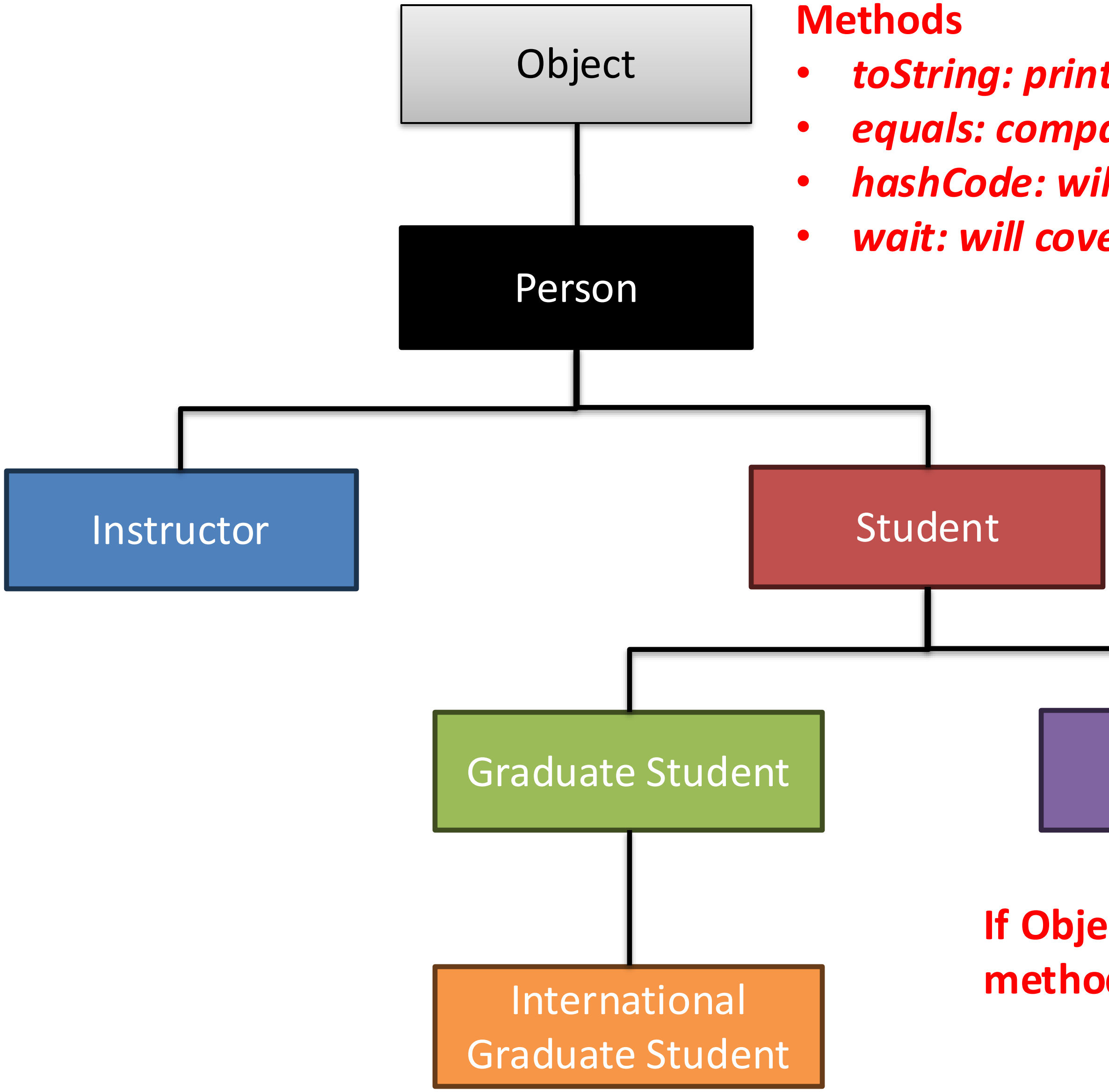
```
Bob is an instructor
```

instanceof
checks type,
returns boolean

bob prints

***Carol* does not print**
because *Carol* is a *Student*

Our classes inherit from Java's Object class behind the scenes



Methods

- ***toString: prints memory address***
- ***equals: compares memory address***
- ***hashCode: will cover soon***
- ***wait: will cover soon***

If you don't implement methods that Java's base class implements, then calling these methods on your classes hunts upward to the base class and runs Object's implementation

If Object doesn't implement the method, Java throws an exception

Agenda

1. Inheritance
2. Comparing objects
3. "Is a" example
4. Access modifiers

Key points:

1. **A subclass "is a" type of the base class (just a specialty version)**

An array that holds Person objects can also hold objects of a subclass type

CollegeApp.java

```
public class CollegeApp {
  public static void main(String[] args) {
    //define some people
    int numberOfPeople = 5;
    Person[] people = new Person[numberOfPeople];
    Instructor tjp = new Instructor("Tim Pierson", "f00zzz");
    tjp.setDepartment("Computer Science");
    people[0] = tjp;
```

Create an array of Person objects

Arrays hold one type of object

Remember: an Instructor is a Person, so is a Student

tjp **can go into a** ***Person*** **array because** ***tjp*** **in an** ***Instructor*** **and instructors are people too! (e.g., Instructor is a subclass of Person, so it "is a" Person)**

An array that holds Person objects can also hold objects of a subclass type

CollegeApp.java

```
public class CollegeApp {
   public static void main(String[] args) {
      //define some people
      int numberOfPeople = 5;
      Person[] people = new Person[numberOfPeople];
      Instructor tjp = new Instructor("Tim Pierson", "f00zzz");
      tjp.setDepartment("Computer Science");
      people[0] = tjp;
      people[1] = new Student("Alice", "f00xyz");
      people[2] = new GraduateStudent("Bob", "f00abc", "Computer Science", "Tim Pierson");
```

There is no need to create a temporary value like *tjp*, can just assign an array slot to a new object if you'd like to

***Alice* (*Student*) and *Bob* (*GraduateStudent*) can go into a *Person* array because they are also Persons (due to subclass)**

A GraduateStudent "is a" Student and a Student "is a" Person!

An array that holds Person objects can also hold objects of a subclass type

CollegeApp.java

```java
public class CollegeApp {
    public static void main(String[] args) {
        //define some people
        int numberOfPeople = 5;
        Person[] people = new Person[numberOfPeople];
        Instructor tjp = new Instructor("Tim Pierson", "f00zzz");
        tjp.setDepartment("Computer Science");
        people[0] = tjp;
        people[1] = new Student("Alice", "f00xyz");
        people[2] = new GraduateStudent("Bob", "f00abc", "Computer Science", "Tim Pierson");
        ((Student)people[2]).setYear(2028);
```

Must *cast people[2]* to a Student to access *graduationYear* because Person does not have a *graduationYear* instance variable

Casting does not change the type of variable stored in array

An array that holds Person objects can also hold objects of a subclass type

CollegeApp.java

```
public class CollegeApp {
   public static void main(String[] args) {
      //define some people
      int numberOfPeople = 5;
      Person[] people = new Person[numberOfPeople];
      Instructor tjp = new Instructor("Tim Pierson", "f00zzz");
      tjp.setDepartment("Computer Science");
      people[0] = tjp;
      people[1] = new Student("Alice", "f00xyz");
      people[2] = new GraduateStudent("Bob", "f00abc", "Computer Science", "Tim Pierson");
      ((Student)people[2]).setYear(2028);
      people[3] = new InternationalStudent("Charlie", "f00123", "Germany");
      people[4] = new InternationalGraduateStudent("Denise", "f00987");
```

Now array *people* holds:

- **An Instructor**
- **A Student**
- **A GraduateStudent**
- **An InternationalStudent**
- **An InternationalGraduateStudent**

That is ok because they are all *Persons*

Add more people to *Person* array

This time we add an *InternationalStudent* and an *InternationalGraduateStudent*, they are people too

An array that holds Person objects can also hold objects of a subclass type

CollegeApp.java

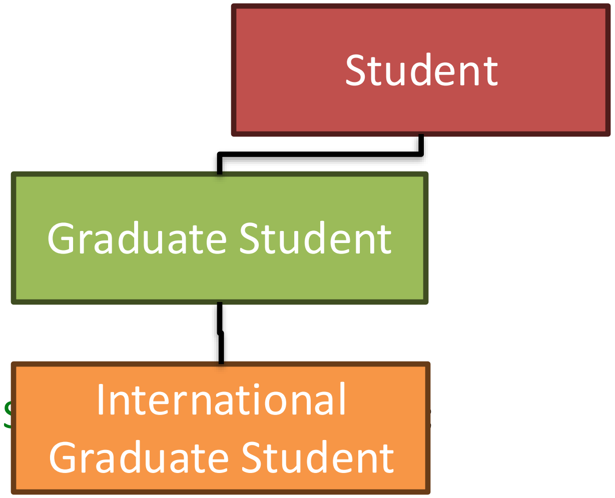
```
public class CollegeApp {
  public static void main(String[] args) {
    //define some people
    int numberOfPeople = 5;
    Person[] people = new Person[numberOfPeople];
    Instructor tjp = new Instructor("Tim Pierson", "f00zzz");
    tjp.setDepartment("Computer Science");
    people[0] = tjp;
    people[1] = new Student("Alice", "f00xyz");
    people[2] = new GraduateStudent("Bob", "f00abc", "Computer Science", "Tim Pierson");
    ((Student)people[2]).setYear(2028);
    people[3] = new InternationalStudent("Charlie", "f00123", "Germany");
    people[4] = new InternationalGraduateStudent("Denise", "f00987");
    ((InternationalGraduateStudent)people[4]).setDepartment("Computer Science");
    ((InternationalGraduateStudent)people[4]).setAdvisorName("Alan Turing");
    ((InternationalGraduateStudent)people[4]).setHomeCountry("Spain");
```

Must cast *people[4]* to *InternationalGraduateStudent* to access class-specific instance variables

An array that holds Person objects can also hold objects of a subclass type

CollegeApp.java

```
public class CollegeApp {
  public static void main(String[] args) {
    //define some people
    int numberOfPeople = 5;
    Person[] people = new Person[numberOfPeople];
    Instructor tjp = new Instructor("Tim Pierson", "f00zzz");
    tjp.setDepartment("Computer Science");
    people[0] = tjp;
    people[1] = new Student("Alice", "f00xyz");
    people[2] = new GraduateStudent("Bob", "f00abc", "Computer S
    ((Student)people[2]).setYear(2028);
    people[3] = new InternationalStudent("Charlie", "f00123", "Germany");
    people[4] = new InternationalGraduateStudent("Denise", "f00987");
    ((InternationalGraduateStudent)people[4]).setDepartment("Computer Science");
    ((InternationalGraduateStudent)people[4]).setAdvisorName("Alan Turing");
    ((InternationalGraduateStudent)people[4]).setHomeCountry("Spain");
```



Could we cast to GraduateStudent instead for InternationalGraduateStudent Denise?
Yes! An InterationalGraduateStudent “is a” GraduateStudent (and “is a” Student)
GraduateStudent defines department and advisor (but not home country!)
InternationalGraduateStudents inherit these from GraduateStudent

An array that holds Person objects can also hold objects of a subclass type

CollegeApp.java

```
public class CollegeApp {
  public static void main(String[] args) {
    //define some people
    int numberOfPeople = 5;
    Person[] people = new Person[numberOfPeople];
    Instructor tjp = new Instructor("Tim Pierson", "f00zzz");
    tjp.setDepartment("Computer Science");
    people[0] = tjp;
    people[1] = new Student("Alice", "f00xyz");
    people[2] = new GraduateStudent("Bob", "f00abc", "Computer Science", "Tim Pierson");
    ((Student)people[2]).setYear(2028);
    people[3] = new InternationalStudent("Charlie", "f00123", "Germany");
    people[4] = new InternationalGraduateStudent("Denise", "f00987");
    ((InternationalGraduateStudent)people[4]).setDepartment("Computer Science");
    ((InternationalGraduateStudent)people[4]).setAdvisorName("Alan Turing");
    ((InternationalGraduateStudent)people[4]).setHomeCountry("Spain");

    //print all people
    for (Person person: people) {
      System.out.println(person + "\n");
    }
  }
}
```

Print all people using a for-each loop

The most specific *toString* method is called for each object

An array that holds Person objects can also hold objects of a subclass type

CollegeApp.java

```java
public class CollegeApp {
   public static void main(String[] args) {
      //define some people
      int numberOfPeople = 5;
      Person[] people = new Person[numberOfPeople];
      Instructor tjp = new Instructor("Tim Pierson", "f00zzz");
      tjp.setDepartment("Computer Science");
      people[0] = tjp;
      people[1] = new Student("Alice", "f00xyz");
      people[2] = new GraduateStudent("Bob", "f00abc", "Computer Science'
      ((Student)people[2]).setYear(2028);
      people[3] = new InternationalStudent("Charlie", "f00123", "Germany");
      people[4] = new InternationalGraduateStudent("Denise", "f00987");
      ((InternationalGraduateStudent)people[4]).setDepartment("Computer S
      ((InternationalGraduateStudent)people[4]).setAdvisorName("Alan Turin
      ((InternationalGraduateStudent)people[4]).setHomeCountry("Spain");

      //print all people
      for (Person p: people) {
         System.out.println(p + "\n");
      }
   }
}
```

```
Name: Tim Pierson (f00zzz)
          Tenured: false
          Years Employed: 0
          Department: Computer Science
Name: Alice (f00xyz)
          Graduation year: null
          Hours studying: 0.0
          Hours in class: 0.0
Name: Bob (f00abc)
          Graduation year: 2028
          Hours studying: 0.0
          Hours in class: 0.0
          Hours in the lab: 0.0
          Department: Computer Science
          Advisor: Tim Pierson
Name: Charlie (f00123)
          Graduation year: null
          Hours studying: 0.0
          Hours in class: 0.0
          Home country: Germany
Name: Denise (f00987)
          Graduation year: null
          Hours studying: 0.0
          Hours in class: 0.0
          Hours in the lab: 0.0
          Department: Computer Science
          Advisor: Alan Turing
          Home country: Spain
```

Agenda

1. Inheritance
2. Comparing objects
3. “Is a” example
4. Access modifiers

Key points:

1. **Access modifiers allow you to control access to an object’s data**

Java allows us to break up major portions of code into Projects, Packages and Classes

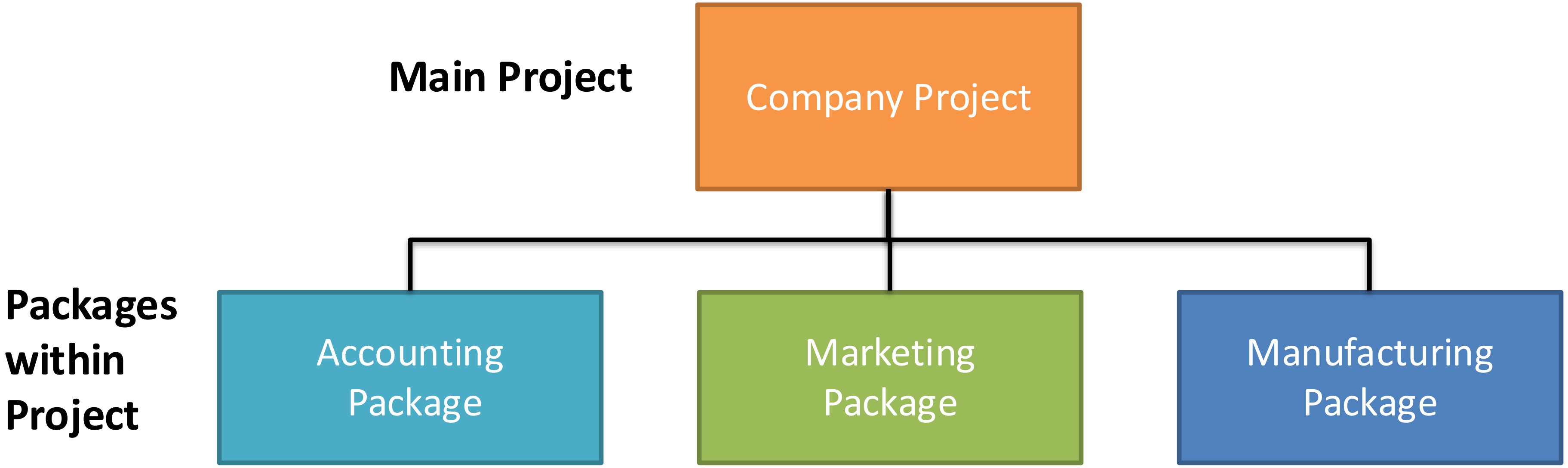
Example of master project for a company

Main Project

Company Project

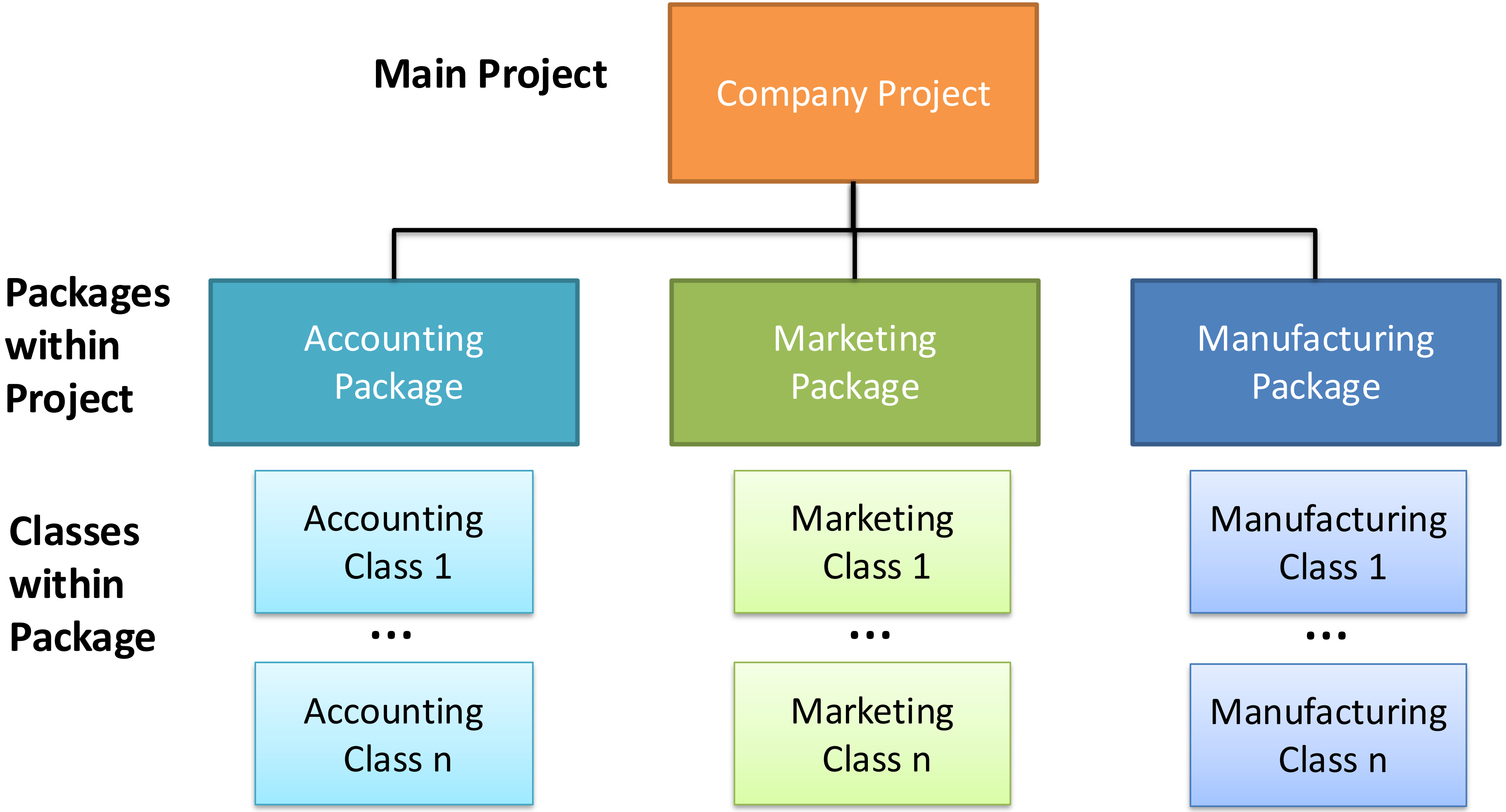
Java allows us to break up major portions of code into Projects, Packages and Classes

Example of master project for a company



Java allows us to break up major portions of code into Projects, Packages and Classes

Example of master project for a company

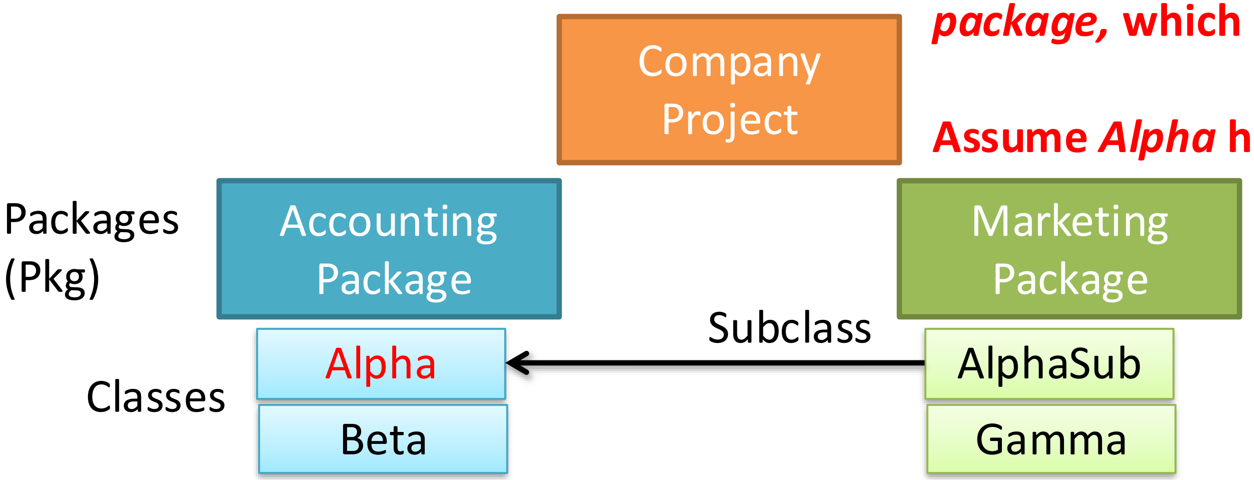


Visibility depends on modifier applied

Example: Visibility of Alpha class

Alpha is a class in *Accounting package,* which is in *Company project*

Assume *Alpha* has instance variable *x*



Company Project

Packages (Pkg)

Accounting Package

Marketing Package

Subclass

Classes

Alpha

Beta

AlphaSub

Gamma

Y = can access

N = cannot access

If Alpha.x is:	Alpha.x can be accessed by:	Accounting Pkg		Marketing Pkg	
		Alpha	Beta	AlphaSub	Gamma
public	Any class	Y	Y	Y	Y
protected	Pkg + Subclass	Y	Y	Y	N
No modifier	Pkg - Subclass	Y	Y	N	N
private	This class only	Y	N	N	N

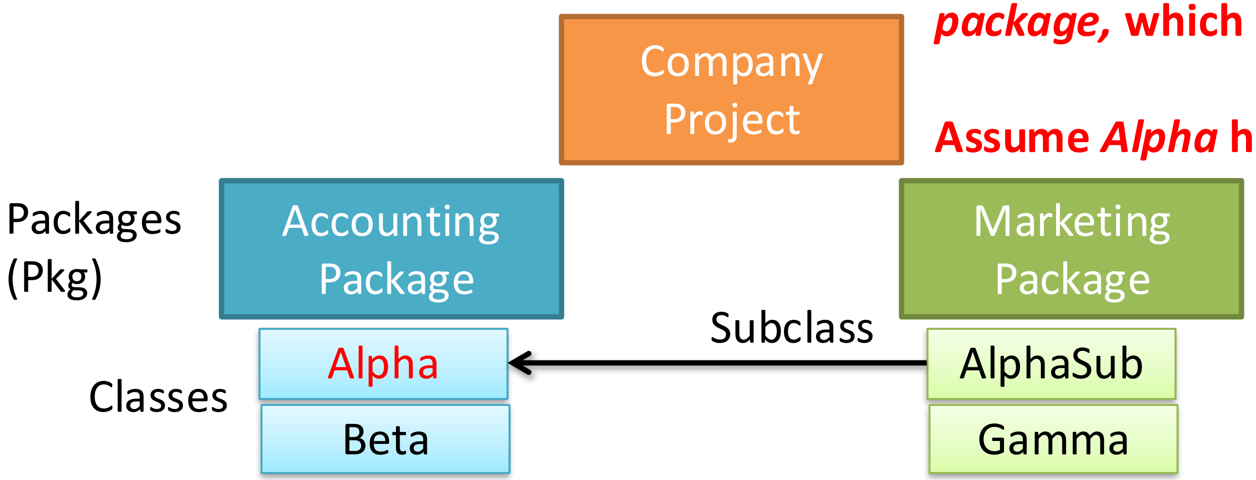
Adapted from Java documentation

Visibility depends on modifier applied

Example: Visibility of Alpha class

Alpha is a class in *Accounting package,* which is in *Company project*

Assume *Alpha* has instance variable *x*



Y = can access
N = cannot access

If Alpha.x is:	Alpha.x can be accessed by:	Accounting Pkg		Marketing Pkg	
		Alpha	**Beta**	**AlphaSub**	**Gamma**
public	**Any class**	**Y**	**Y**	**Y**	**Y**
protected	Pkg + Subclass	Y	Y	Y	N
No modifier	Pkg - Subclass	Y	Y	N	N
private	This class only	Y	N	N	N

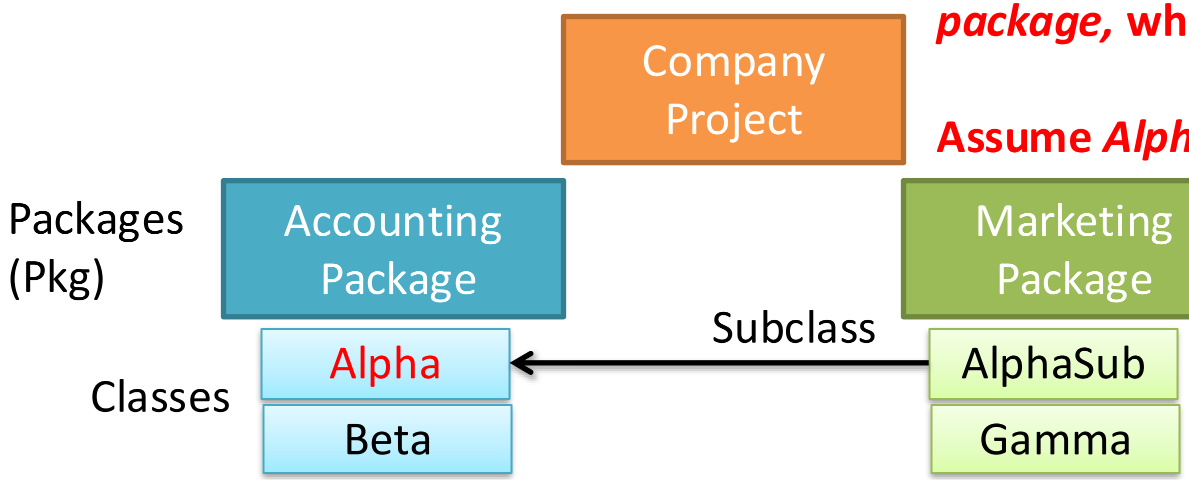
Adapted from Java documentation

Visibility depends on modifier applied

Example: Visibility of Alpha class

Alpha* is a class in *Accounting package,* which is in *Company project

Assume *Alpha* has instance variable *x*



Y = can access

N = cannot access

If **Alpha.x** is:	**Alpha.x** can be accessed by:	Accounting Pkg		Marketing Pkg	
		Alpha	**Beta**	**AlphaSub**	**Gamma**
public	Any class	Y	Y	Y	Y
protected	**Pkg + Subclass**	**Y**	**Y**	**Y**	N
No modifier	Pkg - Subclass	Y	Y	N	N
private	This class only	Y	N	N	N

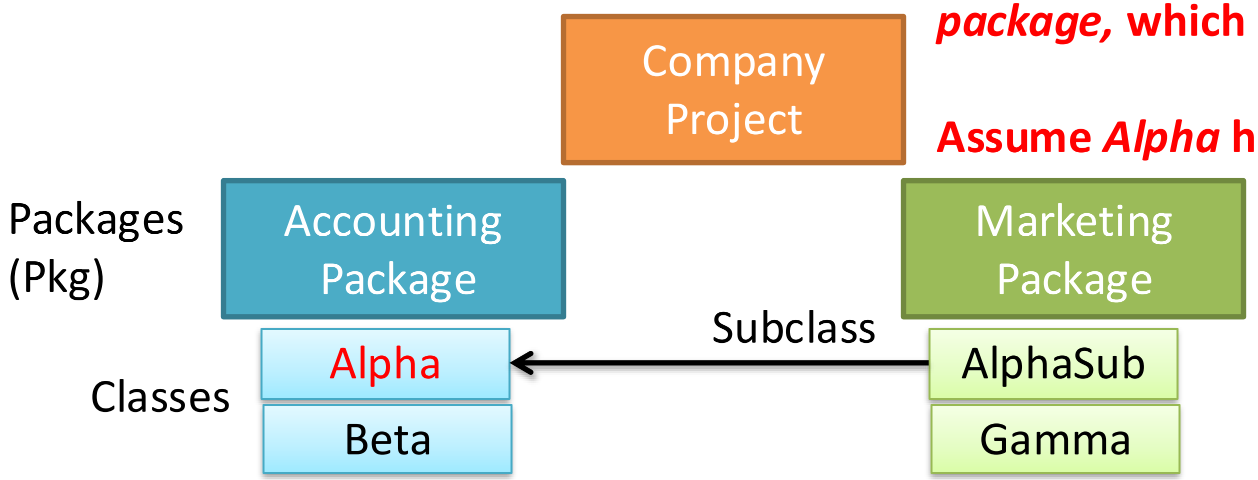
Adapted from Java documentation

Visibility depends on modifier applied

Example: Visibility of Alpha class

Alpha* is a class in *Accounting package,* which is in *Company project

Assume *Alpha* has instance variable *x*



Y = can access

N = cannot access

		Accounting Pkg		Marketing Pkg	
If Alpha.x is:	**Alpha.x can be accessed by:**	**Alpha**	**Beta**	**AlphaSub**	**Gamma**
public	Any class	Y	Y	Y	Y
protected	Pkg + Subclass	Y	Y	Y	N
No modifier	**Pkg - Subclass**	**Y**	**Y**	N	N
private	This class only	Y	N	N	N

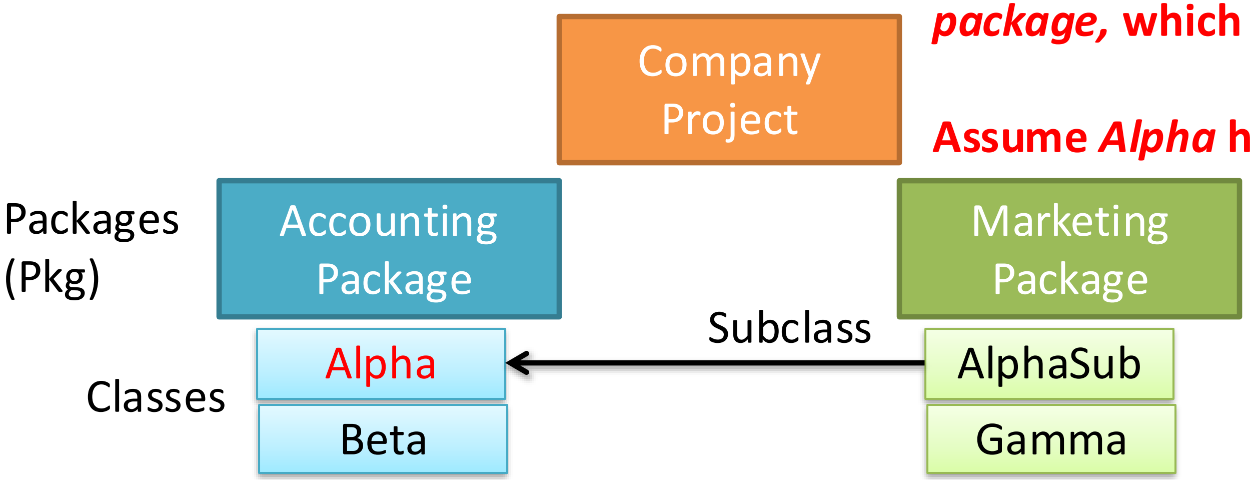
Adapted from Java documentation

Visibility depends on modifier applied

Example: Visibility of Alpha class

Alpha **is a class in** ***Accounting package*****, which is in** ***Company project***

Assume ***Alpha*** **has instance variable** ***x***



Y = can access
N = cannot access

		Accounting Pkg		Marketing Pkg	
If Alpha.x is:	**Alpha.x can be accessed by:**	**Alpha**	**Beta**	**AlphaSub**	**Gamma**
public	Any class	Y	Y	Y	Y
protected	Pkg + Subclass	Y	Y	Y	N
No modifier	Pkg - Subclass	Y	Y	N	N
private	**This class only**	**Y**	N	N	N

Adapted from Java documentation

Key points

1. Create and debug base class
2. Create specialty versions of the of the base class (called subclasses) that inherit the code and data from the base class
3. Use the keyword “extends” to inherit from the base class
4. In Java we can only inherit from one base class (unlike C++)
5. Compare primitive types with ==
6. Compare objects with *equals* method
7. A subclass “is a” type of the base class (just a specialty version)
8. Access modifiers allow you to control access to an object’s data