بسم الله الرحمن الرحيم

جامعة بحري

UNIVERSITY OF BAHRI

كلية علوم الحاسوب

College of Computer Sciences

Object Oriented Paradigms

College Requirements

CSCR2105

Inheritance

Lecture 3

The Software Crisis

- **software engineering**:
  - The practice of developing, designing, documenting, testing large computer programs.
- Large-scale projects face many issues:
  - getting many programmers to work together
  - getting code finished on time
  - avoiding redundant code
  - finding and fixing bugs
  - maintaining, improving, and reusing existing code
- **code reuse**: The practice of writing program code once and using it in many contexts.

Is-a relationships, hierarchies

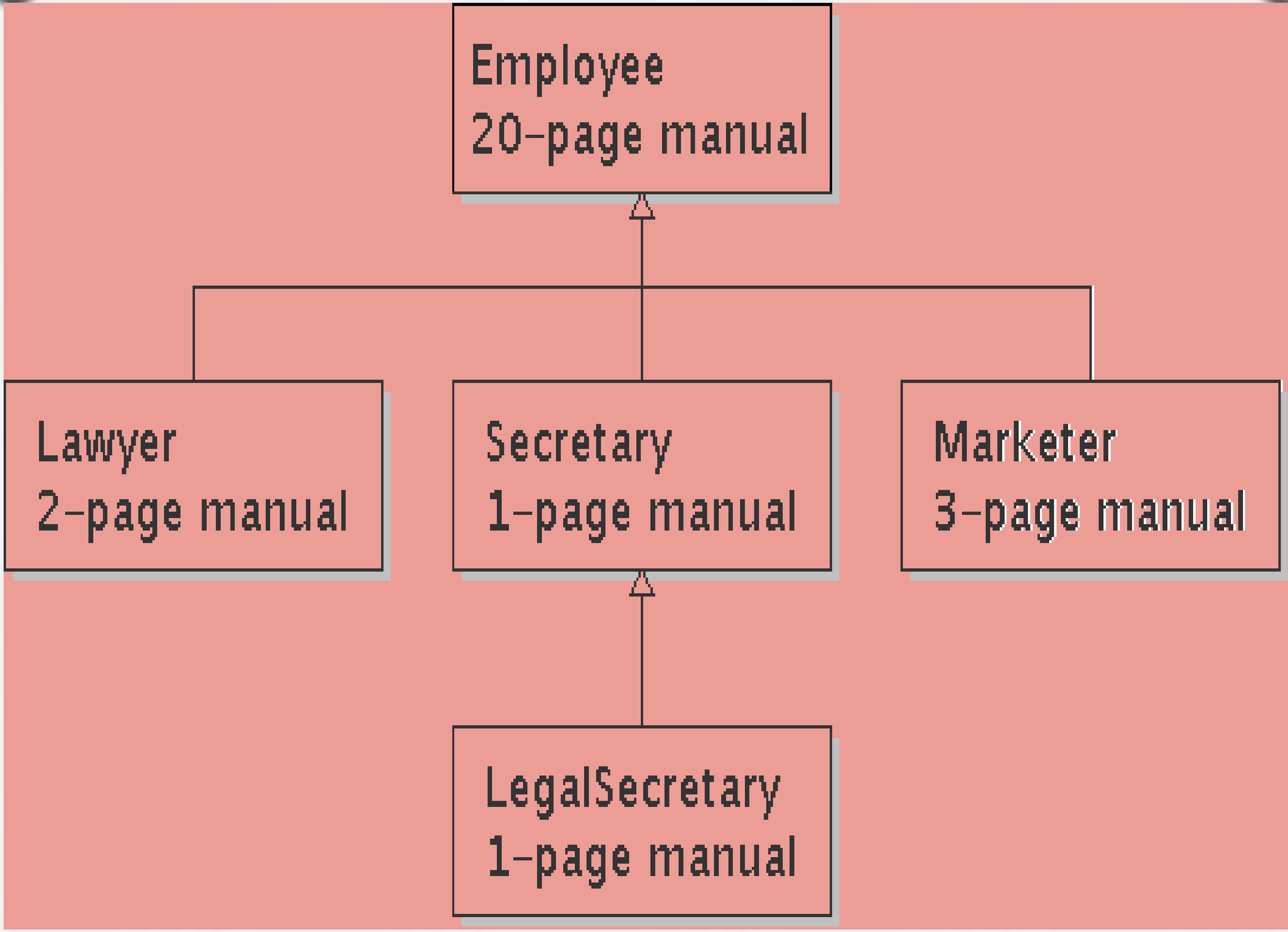
- is-a **relationship:** A hierarchical connection where one category can be treated as a specialized version of another.
  - every Student *is a* Personal.
  - every legal employee *is an* employee .
- **inheritance hierarchy:** A set of classes connected by is-a relationships that can share common code.

Employee regulations

- Consider the following employee regulations:
  - Employees work 40 hours / week.
  - Employees make $40,000 per year, except legal secretaries who make $5,000 extra per year ($45,000 total), and marketers who make $10,000 extra per year ($50,000 total).
  - Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
  - Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.

Law firm employee analogy

- Common rules: hours, vacation, benefits, regulations ...
  - All employees attend a common orientation to learn general company rules
  - Each employee receives a 20-page manual of common rules
- Each subdivision also has specific rules:
  - Employee receives a smaller (1-3 page) manual of these rules
  - Smaller manual adds some new rules and also changes some rules from the large manual


Employee
20-page manual
Lawyer
2-page manual
Secretary
1-page manual
Marketer
3-page manual
LegalSecretary
1-page manual

Separating behavior

- Why not just have a 22 page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?
- Some advantages of the separate manuals:
  - maintenance: Only one update if a common rule changes.
  - locality: Quick discovery of all rules specific to lawyers.
- Some key ideas from this example:
  - General rules are useful (the 20-page manual).
  - Specific rules that may override general ones are also useful.

Employee regulations

- Each type of employee has some unique behavior:
  - Lawyers know how to sue.
  - Marketers know how to advertise.
  - Secretaries know how to take dictation.
  - Legal secretaries know how to prepare legal documents.

An Employee class

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / week
    }
    public double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }
    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }
    public String getForm() {
        return "yellow";     // use the yellow form
    }
}
```

- Exercise: Implement class `Secretary`, based on the previous employee regulations. (Secretaries can take dictation.)

Redundant Secretary class

```
// A redundant class to represent secretaries.
public class Secretary {
    public int getHours() {
        return 40;           // works 40 hours / week
    }
    public double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }
    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }
    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Desire for code-sharing

- `takeDictation` is the only unique behavior in `Secretary`.
- We'd like to be able to say:

```
// A class to represent secretaries.
public class Secretary {
       copy all the contents from the Employee class;
public void takeDictation(String text)
  {
    System.out.println("Taking dictation
  of text: " + text);
     }
}
```

Inheritance

- **inheritance:** A way to form new classes based on existing classes, taking on their attributes/ behavior.
  - a way to group related classes
  - a way to share code between two or more classes
- One class can *extend* another its data/behavior.
  - **superclass:** The parent class that is being extended.
  - **subclass:** The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every <u>field and method </u>from superclass

Inheritance syntax

`public class` **name** `extends` **superclass** `{…}`

- Example:

```
public class Secretary extends Employee {
        ...
  }
```

- By extending `Employee`, each `Secretary` object now:
  - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
  - can be treated as an `Employee` by client code (seen later)

Improved Secretary code

```
// A class to represent secretaries.
public class Secretary extends Employee {
    public void takeDictation(String text) {
        System.out.println("Taking dictation
  of text: " + text);
    }
}
```

- Now we only write the parts unique to each type.
  - `Secretary` inherits `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` methods from `Employee`.
  - `Secretary` adds the `takeDictation` method.

Implementing `Lawyer`

- Consider the following lawyer regulations:
  - Lawyers who get an extra week of paid vacation (a total of 3).
  - Lawyers use a pink form when applying for vacation leave.
  - Lawyers have some unique behavior: they know how to sue.
- Problem: We want lawyers to inherit *most* behavior from employee, but we want to replace parts with new behavior.

Overriding methods

- **override:** To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```
  public class Lawyer extends Employee {
// overrides getForm method in Employee class
      public String getForm() {
          return "pink";
      }
      ...
  }
```

  - Exercise: Complete the `Lawyer` class.
    - (3 weeks vacation, pink vacation form, can sue)

Lawyer class

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacationDays() {
        return 15;           // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

- Exercise: Complete the `Marketer` class. Marketers make $10,000 extra ($50,000 total) and know how to advertise.

Marketer class

```
// A class to represent marketers.
public class Marketer extends Employee {
    public void advertise() {
      System.out.println("Act now while supplies
  last!");
    }

    public double getSalary() {
        return 50000.0;     // $50,000.00 / year
    }
}
```

Levels of inheritance

- Multiple levels of inheritance in a hierarchy are allowed.
  - Example: A legal secretary is the same as a regular secretary but makes more money ($45,000) and can file legal briefs.

    ```
    public class LegalSecretary extends
    Secretary {
            ...
    }
    ```
  - Exercise: Complete the `LegalSecretary` class.

LegalSecretary class

```
// A class to represent legal secretaries.
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all
  day!");
    }

    public double getSalary() {
        return 45000.0;      // $45,000.00 / year
    }
}
```

Interacting with the superclass

Changes to common behavior

- Let's return to our previous company/employee example.
- Imagine a company-wide change affecting all employees.

  Example: Everyone is given a $10,000.

  - The base employee salary is now $50,000.
  - Legal secretaries now make $55,000.
  - Marketers now make $60,000.
- We must modify our code to reflect this policy change.

Modifying the superclass

```
// A class to represent employees (20-page
  manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours /
  week
    }

    public double getSalary() {
        return 50000.0;      // $50,000.00 / year
    }

    ...
}
```

- Are we finished?

- The `Employee` subclasses are still incorrect.
  - They have overridden `getSalary` to return other values.

Bad solution

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 55000.0;
    }
    ...
}

public class Marketer extends Employee {
    public double getSalary() {
        return 60000.0;
    }
    ...
}
```

- Problem: The subclasses' salaries are based on the Employee salary, but the `getSalary` code does not reflect this.

Calling overridden methods

- Subclasses can call overridden methods with `super`

  `super.`**method**`(`**parameters**`)`

- Example:

```
public class LegalSecretary extends
Secretary {
    public double getSalary() {
     double baseSalary=super.getSalary();
     return baseSalary + 5000.0;
    }
    ...
}
```

- Exercise: Modify `Lawyer` and `Marketer` to use `super`.

Improved subclasses

```
public class Lawyer extends Employee {
    public String getForm() {
        return "pink";
    }

    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
public class Marketer extends Employee {
    public void advertise() {
     System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

Inheritance and constructors

- Imagine that we want to give employees more vacation days the longer they've been with the company.
  - For each year worked, we'll award 2 additional vacation days.
  - When an Employee object is constructed, we'll pass in the number of years the person has been with the company.
  - This will require us to modify our `Employee` class and add some new state and behavior.
  - Exercise: Make necessary modifications to the `Employee` class.

Modified `Employee` class

```
public class Employee {
    private int years;
      public Employee(int initialYears) {
        years = initialYears;    }
      public int getHours() {
        return 40;    }
      public double getSalary() {
        return 50000.0;    }
      public int getVacationDays() {
        return 10 + 2 * years;    }
    public String getVacationForm() {
        return "yellow“;    }
}
```

Problem with constructors

- Now that we've added the constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
       ^
```

  - The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.
  - The long explanation: (next slide)

The detailed explanation

- Constructors are not inherited.
  - Subclasses don't inherit the `Employee(int)` constructor.
  - Subclasses receive a default constructor that contains:

```
public Lawyer() {
    super();  // calls Employee() constructor
}
```

- But our `Employee(int)` replaces the default `Employee()`.
  - The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.

Calling superclass constructor

```
super(parameters);
```

- Example:

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
     super(years);// calls Employee constructor
    }
    ...
}
```

- The `super` call must be the first statement in the constructor.
- Exercise: Make a similar modification to the `Marketer` class.

Modified Marketer class

```
// A class to represent marketers.
public class Marketer extends Employee {
    public Marketer(int years) {
        super(years);
    }
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

Modified Secretary class

```
// A class to represent secretaries.
public class Secretary extends Employee {
    public Secretary() {
        super(0);      }
    public void takeDictation(String text) {
     System.out.print("Taking dictation of text:“+text);
    }
   }
```

- Since `Secretary` doesn't require any parameters to its constructor, `LegalSecretary` compiles without a constructor.
  - Its default constructor calls the `Secretary()` constructor.

Inheritance and fields

- Try to give lawyers $5000 for each year at the company:

```
public class Lawyer extends Employee {
    ...
    public double getSalary() {
        return super.getSalary()+5000 * years;
    }
    ...
}
```

- Does not work; the error is the following:

```
Lawyer.java:7: years has private access in
  Employee return super.getSalary()+500*year;
```

  Private fields cannot be directly accessed from subclasses.

  - One reason: So that subclassing can't break encapsulation.
  - How can we get around this limitation?

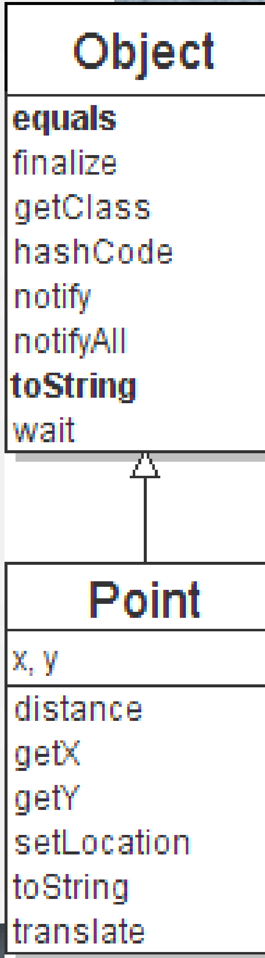
Improved `Employee` code

Add an accessor for any field needed by the subclass.

```
public class Employee {
    private int years;
    public Employee(int initialYears) {
        years = initialYears;    }
    public int getYears() {
        return years;    }
    ...
}
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);  }
    public double getSalary() {
        return super.getSalary() + 5000 * getYears(); }
    ...
}
```

Class `Object`

- All types of objects have a superclass named `Object`.
  - Every class implicitly extends `Object`
- The `Object` class defines several methods:
  - `public String toString()`
    Returns a text representation of the object, often so that it can be printed.
  - `public boolean equals(Object other)`
    Compare the object to any other for equality. Returns `true` if the objects have equal state.



Object

equals
finalize
getClass
hashCode
notify
notifyAll
toString
wait

Point

x, y

distance
getX
getY
setLocation
toString
translate

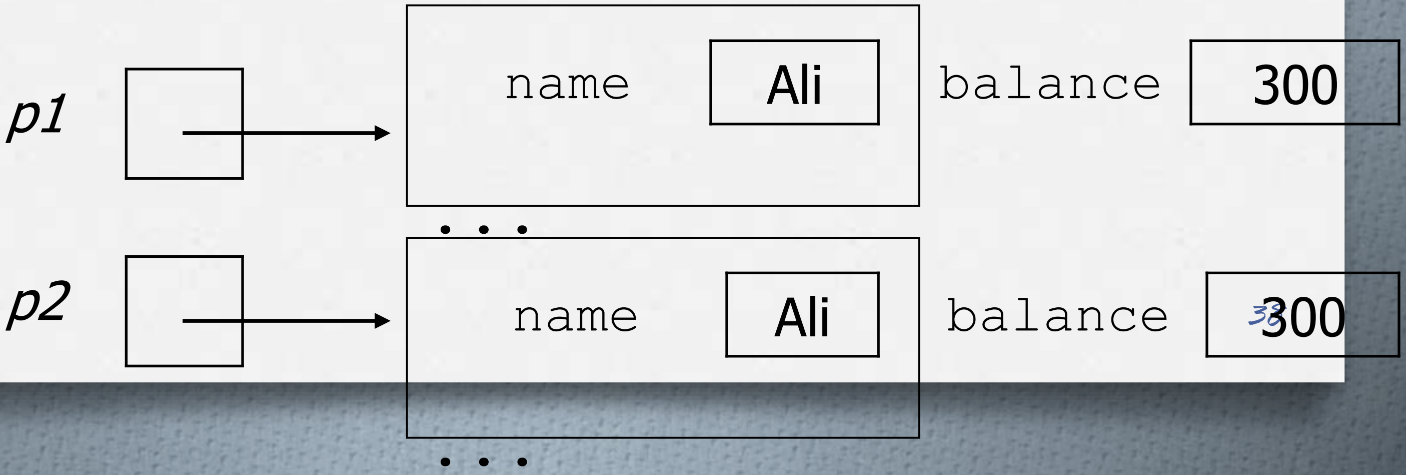
Recall: comparing objects

- The == operator does not work well with objects.

  == compares references to objects, not their state.

  It only produces `true` when you compare an object to itself.

```
AccountBank p1 = new AccountBank(“Ali”,300);
AccountBank p2 = new AccountBank(“Ali”,300);
  if (p1 = = p2) {   // false
      System.out.println("equal");
  }
```



p1

name Ali balance 300

. . .

p2

name Ali balance 300

. . .

The equals method

- The equals method compares the state of objects.

```
if (str1.equals(str2)) {
   System.out.println("the strings are equal");
}
```

- But if you write a class, its equals method behaves like ==

```
if (p1.equals(p2)) {    // false :-(
      System.out.println("equal");
}
```

  - This is the behavior we inherit from class Object.
  - Java doesn't understand how to compare Points by default.

Flawed `equals` method

- We can change this behavior by writing an `equals` method.
  - Ours will *override* the default behavior from class `Object`.
  - The method should compare the state of the two objects and return `true` if they have the same x/y position.
- A flawed implementation:

```
public boolean equals(Point other) {
    if (x == other.x && y == other.y) {
        return true;
    } else {
        return false;
    }
}
```

Flaws in our method

- The body can be shortened to the following:

```
// boolean zen
return x == other.x && y == other.y;
```

- It should be legal to compare a `Point` to any object (not just other `Point`s):

```
// this should be allowed
Point p = new Point(7, 2);
if (p.equals("hello")) {    // false
    ...
```

  - `equals` should always return `false` if a non-`Point` is passed.

equals and Object

```
public boolean equals(Object name) {
        statement(s) that return a boolean value ;
}
```

- The parameter to `equals` must be of type `Object`.
- `Object` is a general type that can match any object.
- Having an `Object` parameter means *any* object can be passed.
  - If we don't know what type it is, how can we compare it?

Another flawed version

- Another flawed `equals` implementation:

```
public boolean equals(Object o) {
    return x == o.x && y == o.y;
}
```

- It does not compile:

```
Point.java:36: cannot find symbol
symbol  : variable x
location: class java.lang.Object
return x == o.x && y == o.y;
             ^
```

  - The compiler is saying,
    "`o` could be any object. Not every object has an `x` field."

Type-casting objects

- Solution: *Type-cast* the object parameter to a `Point`.

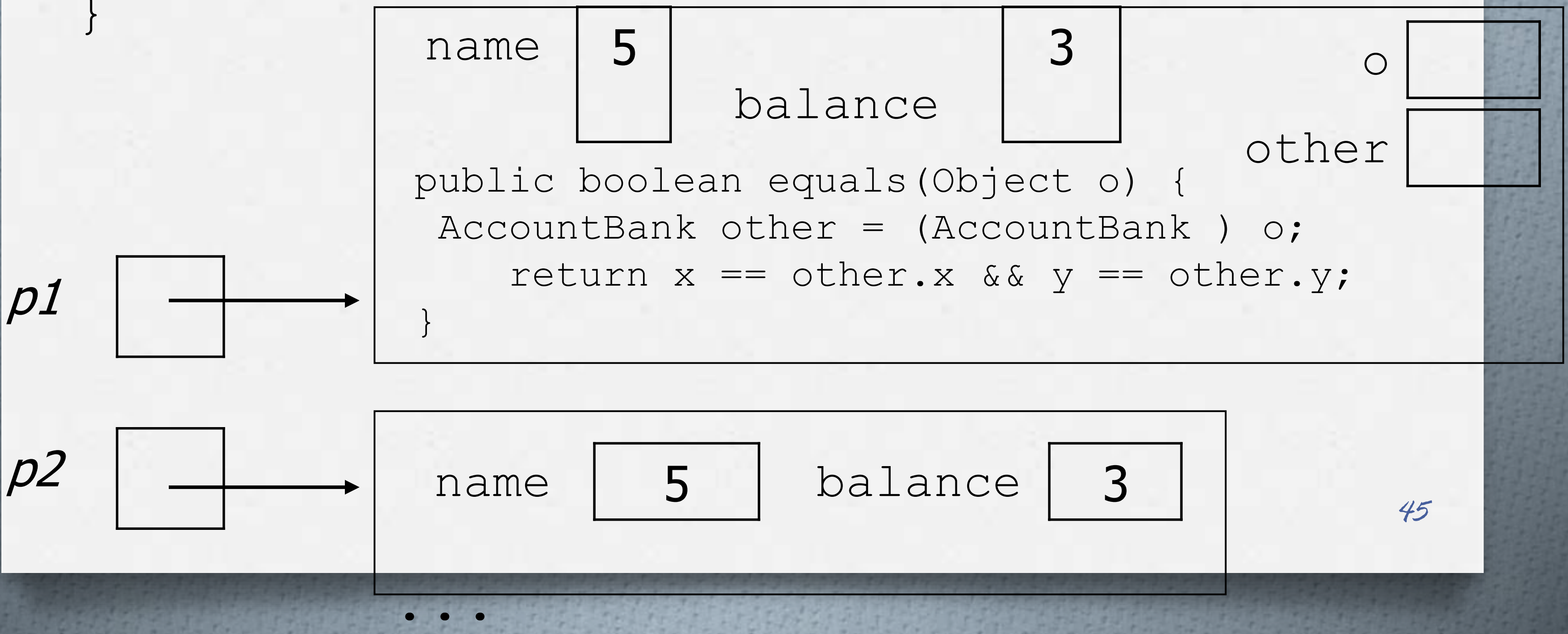
```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- Casting objects is different than casting primitives.
  - Really casting an `Object` reference into a `Point` reference.
  - Doesn't actually change the object that was passed.
  - Tells the compiler to *assume* that `o` refers to a `Point` object.

Casting objects diagram

- Client code:

```
 AccountBank p1 = new AccountBank(“Ali”,3);
 AccountBank p2 = new AccountBank(“Ali”,3);
if (p1.equals(p2)) {
    System.out.println("equal");
}
```



```
public boolean equals(Object o) {
 AccountBank other = (AccountBank ) o;
    return x == other.x && y == other.y;
}
```

Comparing different types

```
 AccountBank  p = new AccountBank(“Ali”, 2);
if (p.equals("hello")) {// should be false
    ...
}
```

- Currently our method crashes on the above code:

```
Exception in thread "main"
java.lang.ClassCastException:
java.lang.String
        at AccountBank .equals(AccountBank
.java:25)
        at PointMain.main(PointMain.java:25)
```

- The culprit is the line with the type-cast:

```
public boolean equals(Object o) {
 AccountBank other = (AccountBank) o;
```

The instanceof keyword

```
if (variable instanceof type) {
        statement(s);
}
```

- Asks if a variable refers to an object of a given type.
  - Used as a boolean test.

```
String s = "hello";
Point p =new Point();
```

expression	result
s instanceof Point	false
s instanceof String	true
p instanceof Point	true
p instanceof String	false
p instanceof Object	true
s instanceof Object	true
null instanceof String	false
null instanceof Object	false

Final equals method

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point.
  public boolean equals(Object o) {
      if (o instanceof Point) {
          // o is a Point; cast and compare it
          Point other = (Point) o;
          return x == other.x && y == other.y;
      } else {
          // o is not a Point; cannot be equal
          return false;
      }
  }
```

NOW:

Waiting for your questions and comments

Lecture 3

Object-Oriented Programming: Polymorphism