Encapsulation
Inheritance

Philippe Meunier

Outline

- Encapsulation
  - Java access modifiers
- Inheritance
  - Inheritance and constructors
  - Inheritance and encapsulation
  - Overriding
  - **`Object`**
  - **`final`**
  - **`abstract`**

Encapsulation – What?

Wikipedia:

1) “the bundling of data with the methods that operate on that data”
    - That’s what classes do.

2) “the restricting of direct access to some of an object's components”
    - Information can be hidden inside a class.

Encapsulation – Why?

Once information is hidden inside a class:

1) The code of the class has complete control over what can and cannot be done with the data.
   - Example: in a **`BankAccount`** class, the amount of money must be hidden.

2) The code outside the class has no control beyond what the class allows.
   - Example: in a **`BankAccount`** class, the amount of money can be changed only by calling one of the class's methods with the proper password.

Encapsulation – How?

```
public class Person {
        private String name;
        public Person(String name) {
                this.name = name;
        }
        public String getName() {
                return name;
        }
}
public class Test {
        public static void main(String[] args) {
                Person p = new Person("Alice");
                System.out.println("p's name: " + p.getName());
        }
}
```

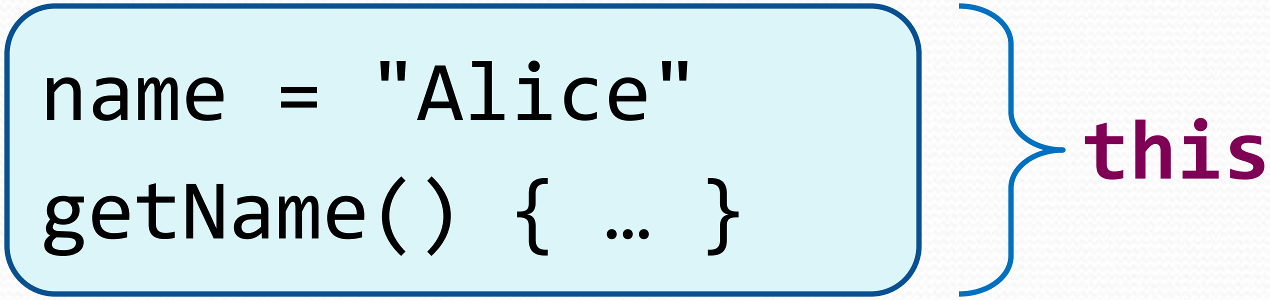
Java Access Modifiers

- Methods and instance variables can be either **`public`** or **`private`**:

Access modifier	same class	other class
`private`	Yes	No
`public`	Yes	Yes

- Good software engineering:
  - keep all the data **`private`** (to keep control over it);
  - provide **`public`** methods to process the data in a controlled manner.

Memory Analysis



```
name = "Alice"
getName() { … }
```

this

Inheritance – What?

Wikipedia: "the mechanism of basing […] a class upon another […] class […], retaining similar implementation"

More practically: inheritance is the process by which a new class is created from an existing class.

Examples:

- A student is a kind of person.
- A teacher is a kind of person.
- A cat is a kind of animal.

Inheritance – Why?

1) We want our Java code to be able to express these relationships between classes, for design clarity.

2) We want such related classes to be able to automatically share code!
   - Example: since a student is a person, we want a student object to have a `getName` method without copy-pasting any code. We want the `Student` class to inherit the `getName` method from the `Person` class.
   - Then you only need to write the `getName` method once!

Inheritance – How?

```
public class Person {
        private String name;
        public Person(String name) {
                this.name = name;
        }
        public String getName() {
                return name;
        }
}
public class Student extends Person {
        private String school;
        public Student(String name, String school) {
                super(name); // Calling the constructor of Person
                this.school = school;
        }
        public String getSchool() {
                return school;
        }
}
```

Inheritance – How?

```
public class Test {
        public static void main(String[] args) {
                Person p = new Person("Alice");
                System.out.println("p's name: " + p.getName());
                Student s = new Student("Bob", "UIC");
                System.out.println("s's name: " + s.getName());
                System.out.println("s's school: " + s.getSchool());
        }
}
```

- The student object **s** has a **getName** method even though no such method appears in the **Student** class.
- All non-private instance variables and methods from **Person** are automatically inherited by **Student**.

Inheritance – How?

- **`Person`** is called the superclass / base class/ parent class.
- **`Student`** is called the subclass / derived class / child class.

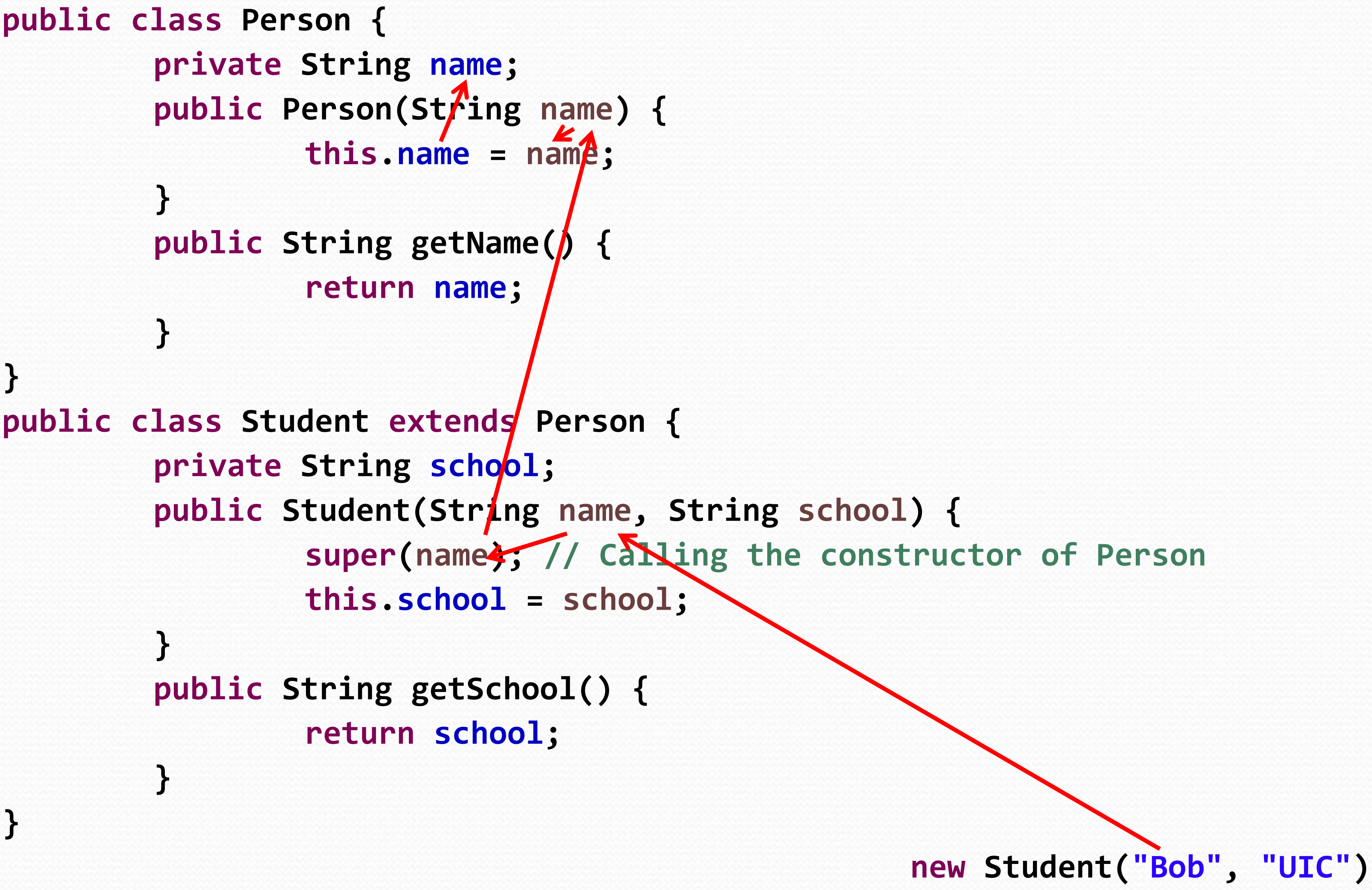
- A class can have many children (and grandchildren, and grandgrandchildren, etc.)
  - **`Person`** might have subclasses **`Student`** and **`Teacher`**;
  - **`Student`** itself might have a **`GradStudent`** subclass.
- A class can have only one parent (in Java).

Inheritance and Constructors

```
public class Person {
        private String name;
        public Person(String name) {
                this.name = name;
        }
        public String getName() {
                return name;
        }
}
public class Student extends Person {
        private String school;
        public Student(String name, String school) {
                super(name); // Calling the constructor of Person
                this.school = school;
        }
        public String getSchool() {
                return school;
        }
}
```

```
new Student("Bob", "UIC")
```

Inheritance and Constructors



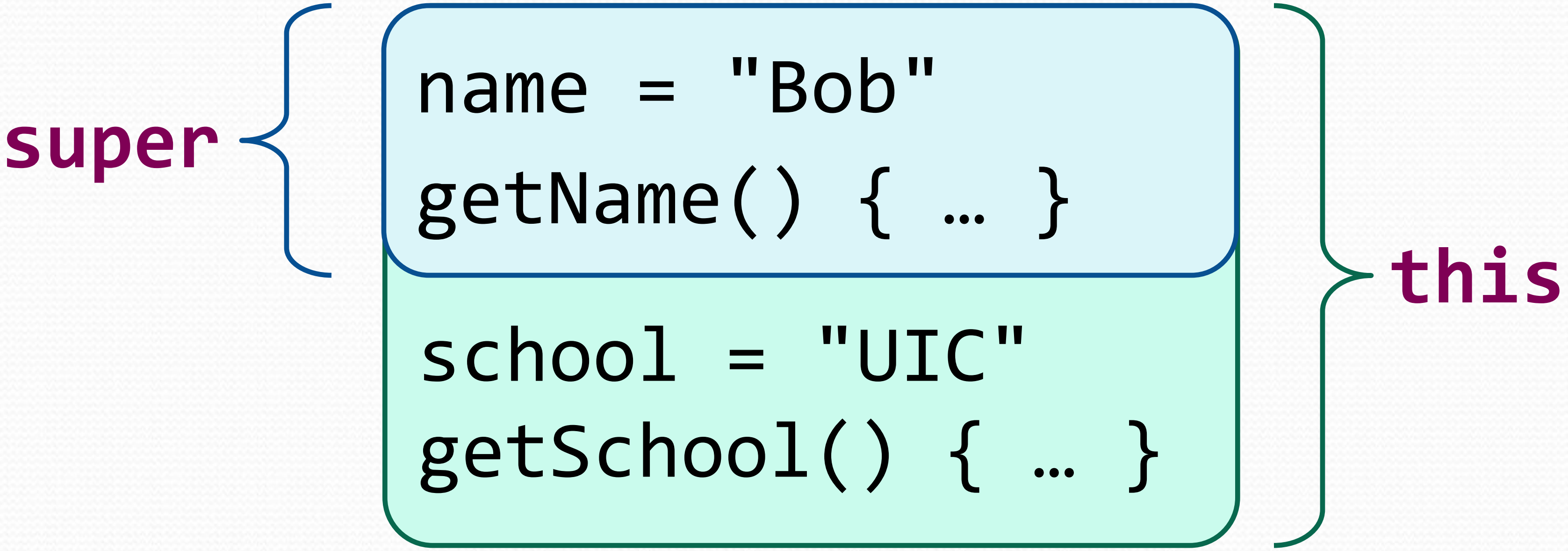
```
public class Person {
        private String name;
        public Person(String name) {
                this.name = name;
        }
        public String getName() {
                return name;
        }
}
public class Student extends Person {
        private String school;
        public Student(String name, String school) {
                super(name); // Calling the constructor of Person
                this.school = school;
        }
        public String getSchool() {
                return school;
        }
}
```

```
new Student("Bob", "UIC")
```

Inheritance and Constructors

- The constructor of `Student` calls the constructor of its superclass `Person` by using `super(name)`
  - Obviously, when calling the `super` constructor, the type and number of arguments must match the type and number of arguments of the constructor of `Person`.
- The call to `super` must always be first inside the constructor of `Student`. That's just the way Java is.
- The call to `super` is optional if the constructor of the superclass takes zero argument.
  - In practice: just always do it.

Memory Analysis



super

```
name = "Bob"
getName() { … }
```

```
school = "UIC"
getSchool() { … }
```

this

Memory Analysis

- `super` corresponds to the part of the object built from the superclass, `this` is the whole object.

- You cannot use **`this.name`** (or just **`name`**) in the code of the **`Student`** class because **`name`** is private!
  - You cannot use **`super.name`** either.

- You can use **`this.getName()`** (or just **`getName()`**) in the code of the **`Student`** class because of inheritance.
  - You can also use **`super.getName()`** but avoid doing that because it will confuse people reading your code.

Inheritance and Encapsulation

- Methods and instance variables can be either **`public`** or **`private`** or **`protected`**:

Access modifier	same class	subclass	other class
`private`	Yes	No	No
`protected`	Yes	Yes	No
`public`	Yes	Yes	Yes

- Good software engineering:
  - still keep all the data **`private`** (to keep control over it);
  - provide **`public`** methods to process the data in a controlled manner;
  - use **`protected`** only if you really need it (rare).

Overriding – What?

- Wikipedia: "a language feature that allows a subclass [...] to provide a specific implementation of a method that is already provided by [its superclass]."

- More practically: a subclass can hide a method inherited from its superclass by providing its own implementation of it.
  - The new implementation must have the same name, same type, and same number of arguments.

Overriding – Why?

- All subclasses automatically inherit the (non-private) methods of the superclass.

- In most cases that's what you want: code sharing.

- In some cases a subclass might want to do its own thing: then you override.

Overriding – How?

```
public class Person {
        private String name;
        public Person(String name) {
                this.name = name;
        }
        public String getName() {
                return name;
        }
        public String getInfo() {
                return "Person "+ name;
        }
}
```

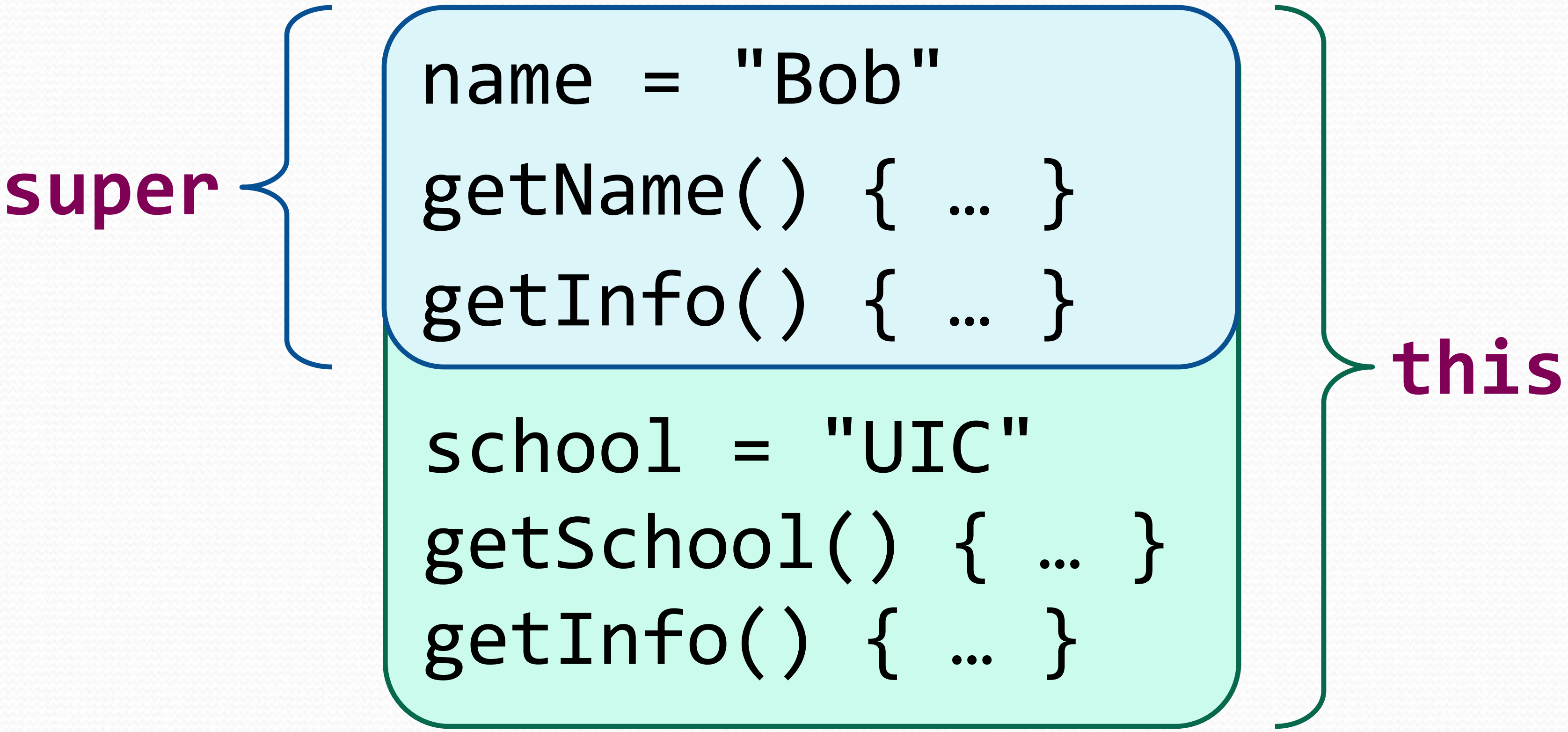
Overriding – How?

```
public class Student extends Person {
        private String school;
        public Student(String name, String school) {
                super(name);
                this.school = school;
        }
        public String getSchool() {
                return school;
        }
        @Override
        public String getInfo() {
                return "Student " + getName() + " from " + school;
        }
}
```

Overriding – How?

```
public class Test {
        public static void main(String[] args) {
                Person p = new Person("Alice");
                System.out.println("p's name: " + p.getName());
                System.out.println("p's info: " + p.getInfo());
                Student s = new Student("Bob", "UIC");
                System.out.println("s's name: " + s.getName());
                System.out.println("s's school: " + s.getSchool());
                System.out.println("s's info: " + s.getInfo());
        }
}
```

Memory Analysis



super

```
name = "Bob"
getName() { … }
getInfo() { … }
```

```
school = "UIC"
getSchool() { … }
getInfo() { … }
```

this

Memory Analysis

- You can use **`this.getName()`** (or just **`getName()`**) in the code of the **`Student`** class because of inheritance.
  - You can also use **`super.getName()`** but avoid doing that because it will confuse people reading your code.

- You can use **`this.getInfo()`** (or just **`getInfo()`**) in the code of the **`Student`** class to call the **`getInfo`** method of the **`Student`** class itself.

- If you really need it, you can use **`super.getInfo()`** in the code of the **`Student`** class to call the **`getInfo`** method inherited from the **`Person`** class. Avoid doing that unless strictly necessary.

Object

- The truth: in Java every class must have a superclass.
  - So what’s the superclass of **`Person`**?

- The code

```
public class Person { … }
```

is in fact the same as

```
public class Person extends Object { … }
```

Object

- The **`Object`** class is provided to you by Java and is automatically the superclass of every class that does not specify a superclass explicitly.

- Therefore **`Object`** is the ancestor class of all classes.
  - All classes are organized as a single tree with **`Object`** at the root.

- Therefore every (non-private) method of the **`Object`** class is inherited by every class.

Final

- A class which is declared as `final` cannot have a subclass.
    - Example: you do not want anyone to be able to create their own kind of bank account by subclassing your **`BankAccount`** class.
- A method which is declared as `final` cannot be overridden.
    - Example: you do not want a subclass to be able to override the method in your **`BankAccount`** class that checks passwords.
- An instance variable which is declared as `final` becomes constant.
    - It can still be initialized in a constructor.

Abstract – What?

- A method which is declared as **`abstract`** cannot have code.
  - It is still inherited by subclasses and can then be overridden.
- A class which is declared as **`abstract`** cannot be instantiated to create objects from it.
  - It can still have subclasses.
- A class which has at least one **`abstract`** method (including inherited methods) must be declared **`abstract`**.

Abstract – Why?

1) Sometimes you do not have enough information to implement a method in a superclass.
   - Example: how do you compute the area of a shape? It depends on which specific kind of shape you're talking about (square, circle, etc.)

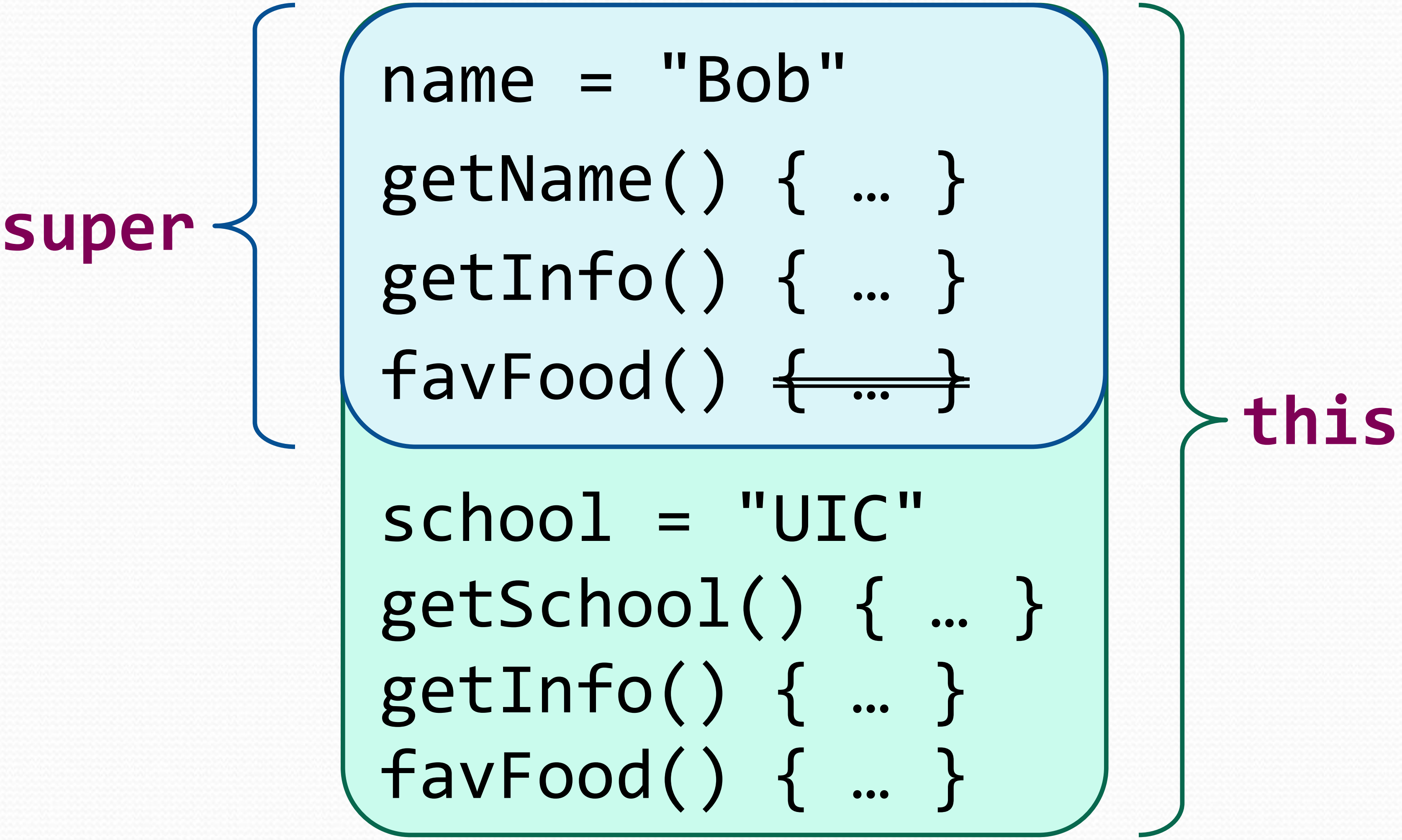
2) You still want the superclass to have that method.
   - Then subclasses must either override the method or themselves be abstract.

3) Therefore you make the method **`abstract`**.

Abstract – How?

```
public abstract class Person {
        // Everything else as above.
        public abstract String favFood();
}
public class Student extends Person {
        // Everything else as above.
        @Override
        public String favFood() {
                return "pizza";
        }
}
public class Test {
        public static void main(String[] args) {
                // Person p = new Person("Alice");
                Student s = new Student("Bob", "UIC");
                // Everything else as above.
                System.out.println("s's food: " + s.favFood());
        }
}
```

Memory Analysis



super

name = "Bob"

getName() { … }

getInfo() { … }

favFood() ~~{ … }~~

school = "UIC"

getSchool() { … }

getInfo() { … }

favFood() { … }

this

Summary

- Encapsulation
    - Java Access Modifiers
- Inheritance
    - Inheritance and constructors
    - Inheritance and encapsulation
    - Overriding
    - **`Object`**
    - **`final`**
    - **`abstract`**