Object-Oriented Programming

Inheritance

United International College

Review

- Classes, Objects
- Members (instance variables, fields)
- Methods
- Constructors
- `new`
- `this`
- `private`, no modifier, `protected`, `public`
- Overloading
- `static`

Review

- Accessors / Mutators
- Encapsulation
- **`package`**

Outline

- Class inheritance
- **`extends`**
- Overriding methods
- Calling parent methods using **`super`**
- **`super()`** parent constructor call
- **`this()`** constructor call

Inheritance

- One of the main techniques of object-oriented programming (OOP).
- A very general form of a class is first defined.
- More specialized classes are defined by adding instance variables and methods.
- The specialized classes ***inherit*** the non-private methods and non-private instance variables of the general class.

Inheritance

- Inheritance is the process by which a new class is created from another class:
  - The new class is called a **subclass** or **derived class** or **child class.**
  - The original class is called the **superclass** or **base class** or **parent class.**
- Inheritance is especially advantageous because it allows code to be **reused**, without having to copy it into the definitions of the subclasses.

Example: Superclass

```
public class Person {
    private String name;
    private int age;
    public Person() {
        this.name = "Alice";
        this.age = 22;
    }
    public String getName(){
        return name;
    }
    public int getAge(){
        return age;
    }
}
```

Person.java

Example: Subclass and Test

```
public class Student extends Person {
    private String school;
    public Student() {
        this.school = "UIC";
    }
    public String getSchool() {
        return school;
    }
}
public class Test {
    public static void main(String arg[]){
        Person p = new Person();
        System.out.println("Person's name: " + p.getName());
        System.out.println("Person's age: " + p.getAge());
        //System.out.println("Person's school: " + p.getSchool());
        Student s = new Student();
        System.out.println("Student's name: " + s.getName());
        System.out.println("Student's age: " + s.getAge());
        System.out.println("Student's school: " + s.getSchool());
    }
}
```

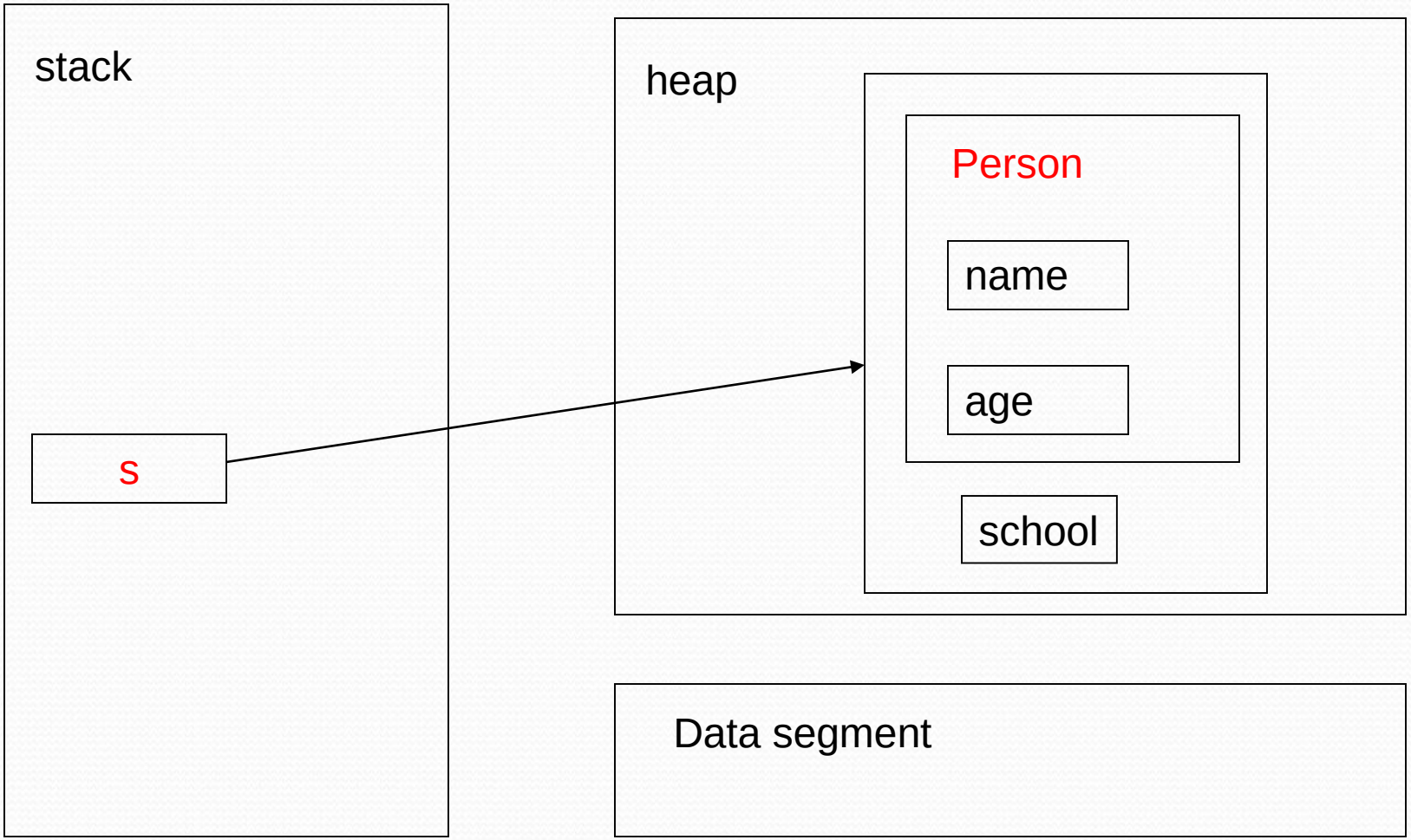
Student.java

Test.java

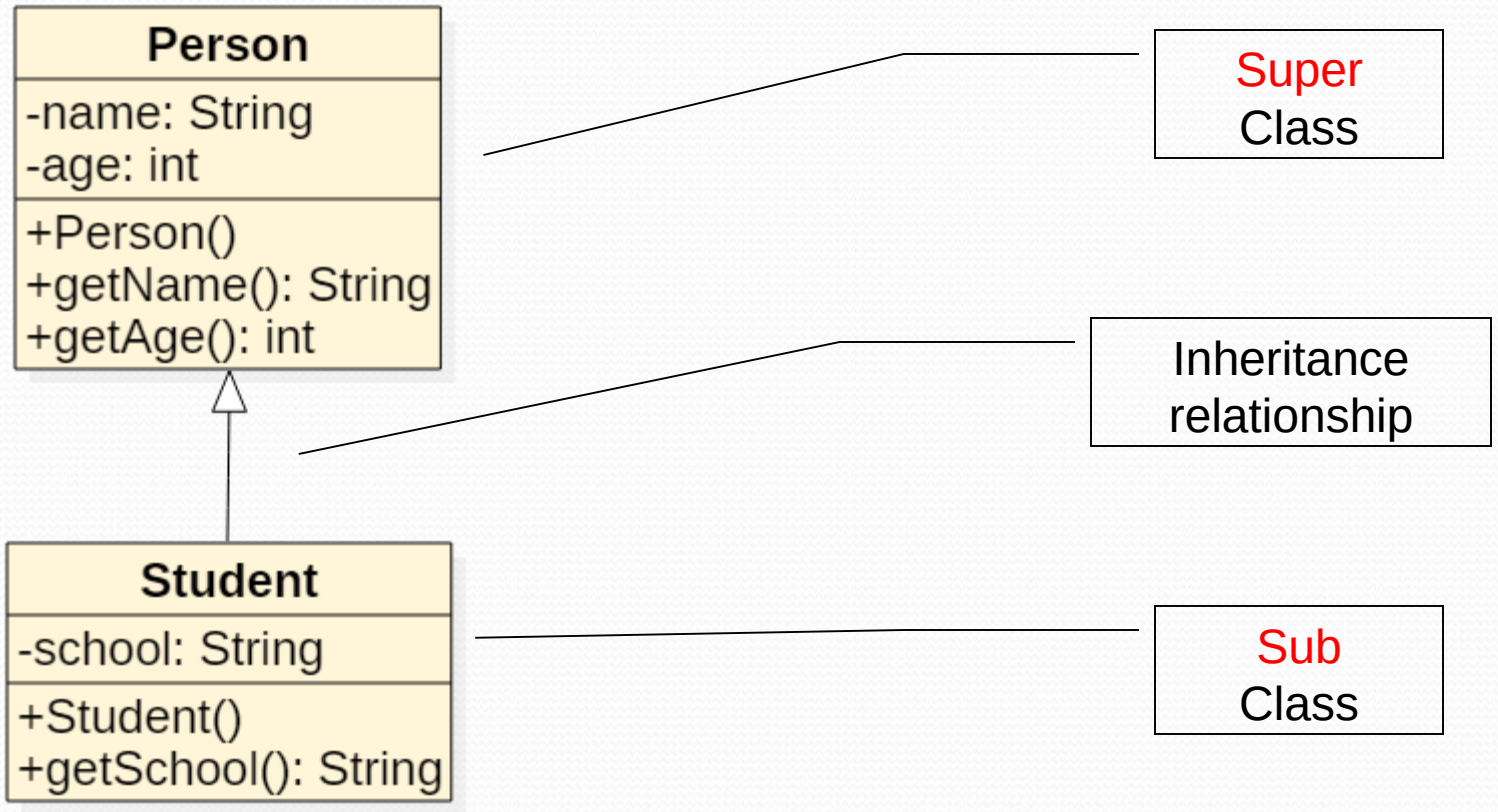
Inheritance

A subclass automatically has all the non-private instance variables and methods of the superclass and it can have additional methods and/or instance variables as well.

Memory Analysis



Inheritance in UML



Example: Derived Person Classes

- Use the **`extends`** keyword, followed by the name of the class to inherit from:

```
public class Student extends Person {
    // new fields and methods
    // specific to a student go here
}
```

- This gives **`Student`** all the same fields and methods as **`Person`**, yet allows its code to focus exclusively on the features that make a student unique.

SubClasses

- When a subclass is defined, it is said to **inherit** the instance variables and methods of the superclass that it extends.
- So the class **`Student`** inherits all the (non-private) instance variables and methods of **`Person`**:
  - The class **`Student`** inherits the methods **`getName`** and **`getAge`** from the class **`Person`**.
  - Any object of the class **`Student`** can use the two methods.
- **`Student`** has one more method **`getSchool`**.
  *Note that this method is **not** available to the superclass Person!*

Inheritance Summary

- A **subclass** automatically has all the non-private instance variables and methods of the **superclass**.
  - If we use `protected` instead of `public` for the `getName` method then the subclass can still use `getName` but code in other packages cannot.
  - If we use `private` instead of `public` for the `getName` method then the subclass cannot inherit the method.
- Definitions for the inherited variables and methods do not appear in the subclass:
  - The code is reused without having to explicitly copy it.

Parent and Child Classes

- ***JAVA doesn't allow multiple inheritance***
  - One child class can only inherit from one parent class.
  - One parent class can derive many child classes.
- This relationship is often extended such that a class that is a parent of a parent . . . of another class is called an ***ancestor class***:
  - A class `A` can have a child class `B` which itself has a child class `C` which itself has a child class `D` which…
  - If class `A` is an ***ancestor*** of class `D` then class `D` is a ***descendent*** of class `A`.

Overriding a Method Definition

- Method Overriding is achieved when a subclass overrides (hides) non-static methods defined in the superclass with its own methods.
- The new method definition must have the same method signature (i.e., method name and parameters) and compatible return type.
- The new method definition cannot narrow the accessibility of the method, but it can widen it.

Example: Superclass

```
public class Person {
    private String name;
    private int age;
    public Person() {
        this.name = "Alice";
        this.age = 22;
    }
    public String getName(){
        return name;
    }
    public int getAge(){
        return age;
    }
    public String getInfo() {
        return "Person "+ name + " is " + age;
    }
}
```

Example: Subclass

```
public class Student extends Person {
    private String school;
    public Student() {
        this.school = "UIC";
    }
    public String getSchool() {
        return school;
    }
    @Override // Asks Java to check for overriding errors.
    public String getInfo() { // Overrides the inherited getInfo().
        return "Student "+ getName() + " is " + getAge() +
                " and at " + school;
    }
}
```

Example: Test

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

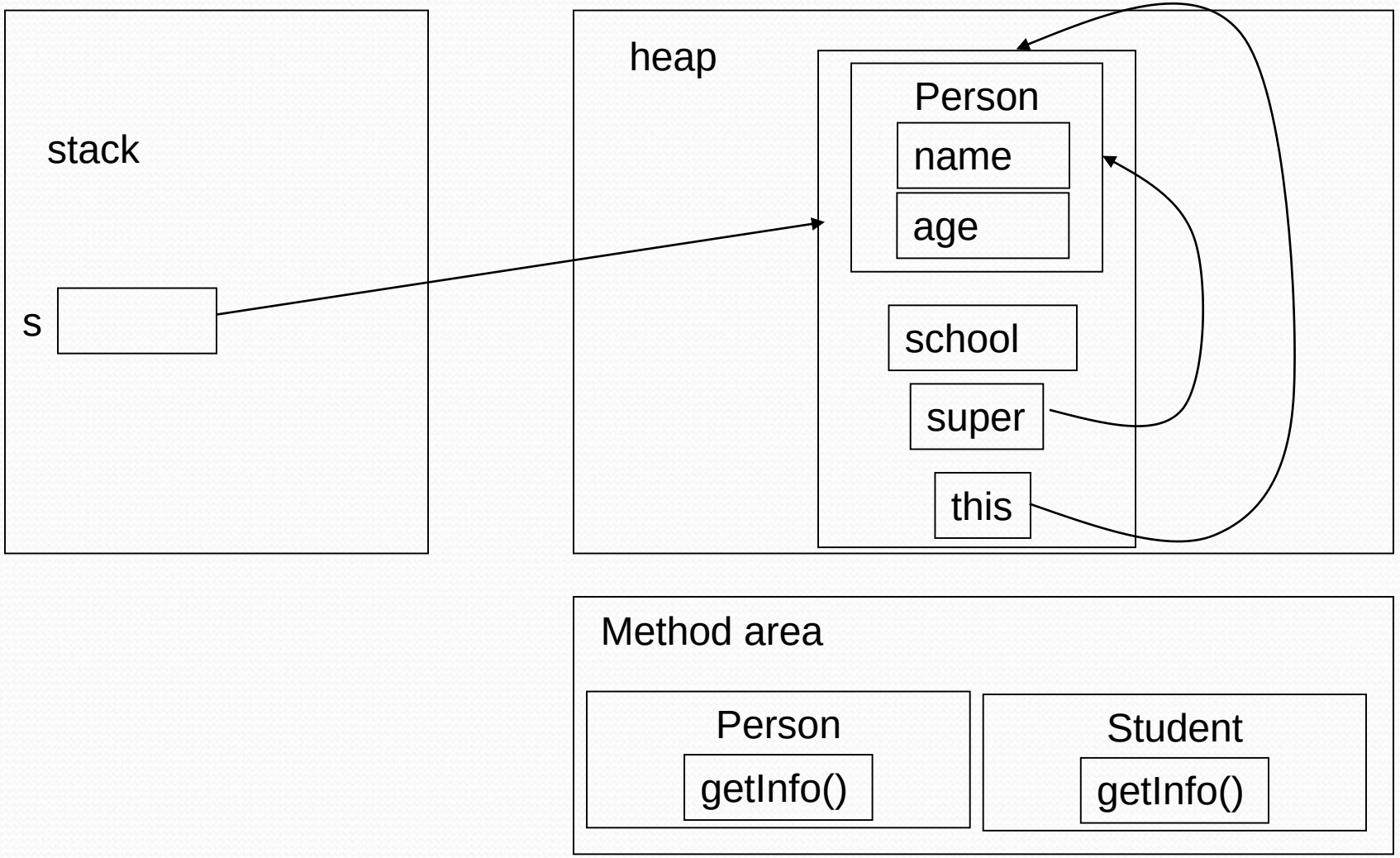
The Keyword: super

```
public class Student extends Person {
    private String school;
    public Student() {
        this.school = "UIC";
    }
    public String getSchool() {
        return school;
    }
    @Override
    public String getInfo() {
        return "Student "+ getName() + " is " + getAge() +
                " and at " + school;
    }
    public String getParentInfo() {
        return super.getInfo(); // Call getInfo() of parent.
    }
}
```

The Keyword: **super**

```
public class Test {
    public static void main(String arg[]){
        Person p = new Person();
        System.out.println("Person's name: " + p.getName());
        System.out.println("Person's age: " + p.getAge());
        System.out.println("Person's info: " + p.getInfo());
        Student s = new Student();
        System.out.println("Student's name: " + s.getName());
        System.out.println("Student's age: " + s.getAge());
        System.out.println("Student's school: " + s.getSchool());
        System.out.println("Student's info: " + s.getInfo());
        System.out.println("parent's info: " + s.getParentInfo());
    }
}
```

Memory Analysis



The `super` Constructor

- A subclass uses a constructor from the superclass to initialize all the data from the superclass.
- In order to invoke a superclass constructor from the subclass, use again `super` with the correct number of arguments to call the constructor of the superclass.

Example

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

Example

```
public class Student extends Person {
    private String school;
    public Student(String name, int age, String school) {
        super(name, age); // Calls the constructor of Person.
        this.school = school;
    }
    public String getSchool() {
        return school;
    }
    @Override
    public String getInfo() {
        return "Student "+ getName() + " is " + getAge() +
                " and at " + school;
    }
    public String getParentInfo() {
        return super.getInfo();
    }
}
```

Example

```
public class Test {
    public static void main(String arg[]){
        Person p = new Person("Alice", 22);
        System.out.println("Person's name: " + p.getName());
        System.out.println("Person's age: " + p.getAge());
        System.out.println("Person's info: " + p.getInfo());
        Student s = new Student("Alice", 22, "UIC");
        System.out.println("Student's name: " + s.getName());
        System.out.println("Student's age: " + s.getAge());
        System.out.println("Student's school: " + s.getSchool());
        System.out.println("Student's info: " + s.getInfo());
        System.out.println("parent's info: " + s.getParentInfo());
    }
}
```

The `super` Constructor

- Every constructor always automatically calls its superclass constructor.
- Java implicitly adds a call to `super()` in each constructor which does not explicitly call `super()` as its first statement (or does not call `this()`: see later).
- The implicit `super()` can be replaced by an explicit `super()`. The `super` statement must be the *first* statement of the constructor.

Example

```
public class Person {
    private String name;
    private int age;
    public Person() {
        this.name = "Alice";
        this.age = 22;
    }
    ...
}
public class Student extends Person {
    private String school;
    public Student() {
        //super(); // Optional.
        this.school = "UIC";
    }
    ...
}
```

The `super` Constructor

If a class only defines non-default constructors, and its subclasses does not include an explicit **`super()`** call, this will be flagged as a compile-time error.

Example

```
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    ...
}
public class Student extends Person {
    private String school;
    public Student(String name, int age, String school) {
        super(name, age); // Mandatory!
        this.school = school;
    }
    ...
}
```

The `this` Constructor

- Within the definition of a constructor for a class, `this` can be used as a name for invoking another constructor in the same class.
- The same restrictions on how to use a call to `super` apply to the `this` constructor.

Example

```
public class Student extends Person {
    private String school;
    public Student(String name, int age, String school) {
        super(name, age); // Calls the constructor of Person.
        this.school = school;
    }
    public Student(String name, int age) {
        // Calls the other constructor of Student.
        this(name, age, "UIC");
    }
    ...
}
public class Test {
    public static void main(String arg[]){
        ...
        Student s = new Student("Alice", 22, "UIC");
        ...
        Student t = new Student("Bob", 21);
        System.out.println("Student's info: " + t.getInfo());
    }
}
```

`this` and `super`

- To call a method in the class itself:

  ```
  this.methodName(…)
  ```

- To call a constructor in the class itself:

  ```
  this(…)
  ```

- To call a method from the parent class:

  ```
  super.methodName(…)
  ```

- To call a constructor from the parent class:

  ```
  super(…)
  ```

Summary

- Class inheritance
- **`extends`**
- Overriding methods
- Calling parent methods using **`super`**
- **`super()`** parent constructor call
- **`this()`** constructor call