Object-Oriented Programming

Inheritance (Cont.)

CST & DS

United International College

Review

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

Outline

- **`final`** modifier
- **`Object`** class
- **`toString`** and **`equals`** methods
- Dynamic Binding
- Subtyping polymorphism
- Type casting: upcasts and downcasts
- **`instanceof`** operator

Last Week: Person

```java
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;
    }
}
```

Person.java

Last Week: Student

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

Student.java

Last Week: Test

```java
public class Test1 {
    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());
        Student t = new Student("Bob", 21);
        System.out.println("Student's info: " + t.getInfo());
    }
}
```

Test1.java

The `final` Modifier

- If the modifier **`final`** is placed before the definition of a ***variable***, the value of this variable cannot be changed.

- If the modifier **`final`** is placed before the definition of a ***method***, then that method cannot be redefined (overridden) in a subclass.

- If the modifier **`final`** is placed before the definition of a ***class***, then that class cannot have subclasses.

- The Java Language Specification recommends listing modifiers in the following order:

  - [public/protected/private]

  - abstract

  - static

  - final

  - ...

Examples

- Make a local variable `final` in the `main` method in the `Test` class: `final int x = 3;` Then, `x = 5;` is a compile-time error.

- Make the `getInfo()` method of the `Person` class `final`: `public final String getInfo(){…}`. Then, the `getInfo()` method in the subclass `Student` becomes a compile-time error.

- Make the `Person` class `final`: `final class Person {…}`. Then, `class Student extends Person{…}` becomes a compile-time error.

The `Object` Class

- In Java, every class is a descendent of the class `Object`:
  - Every class has `Object` as its ancestor.
  - Every object of every class is of type `Object`, as well as being of the type of its own class.
- If a class is defined without explicitly deriving from another class, it is still automatically a derived class of the class `Object`.
- Example: `class Person extends Object { … }`

The **Object** Class
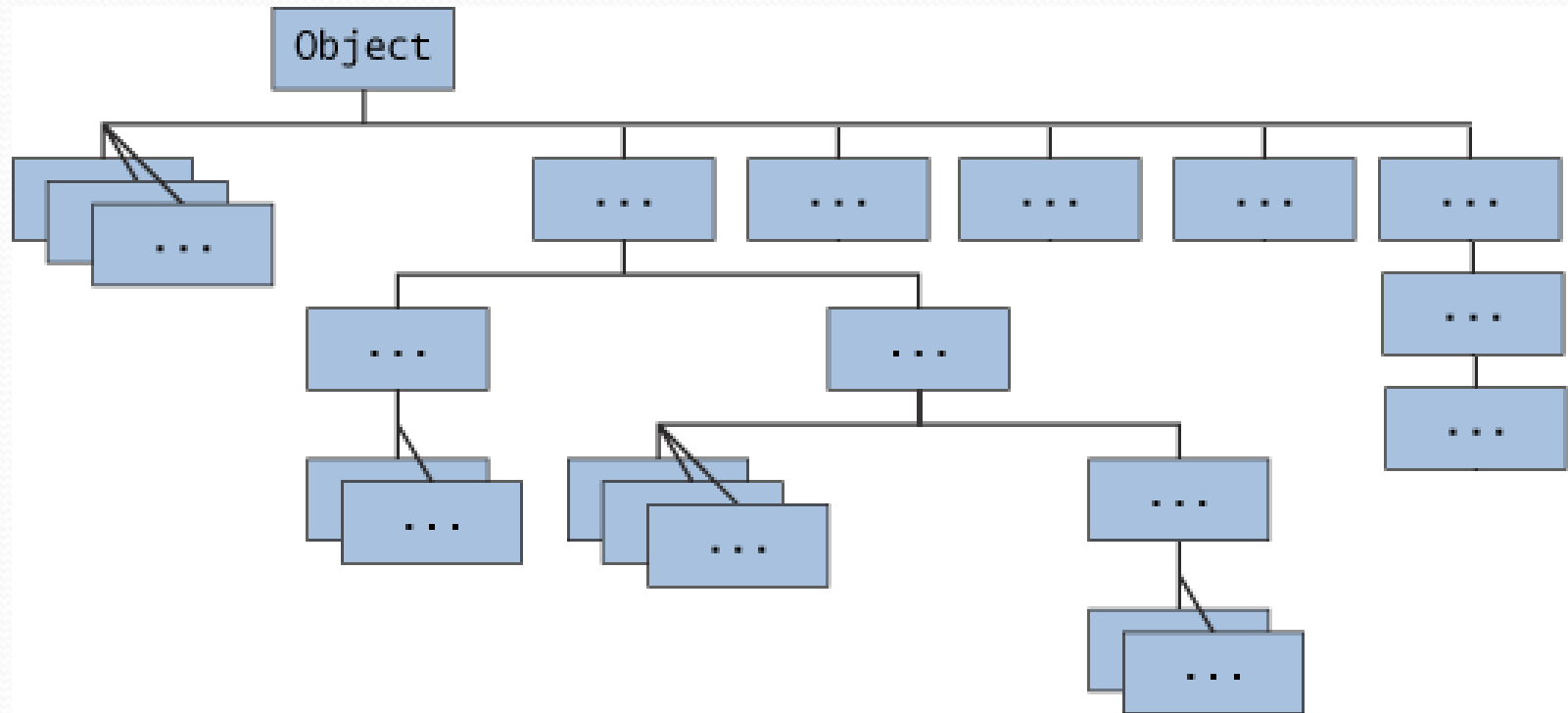
The **Object** Class

- The class **Object** is in the package **java.lang** which is always imported automatically.
- The class **Object** has some methods.
  - Example: the **toString** and **equals** methods.
- Because **Object** is the ancestor of every class, then every class automatically inherits these methods.
- However, these inherited methods should be overridden with definitions more appropriate to a given class.
  - Some Java library classes assume that every class has its own version of such methods.

The **Object** Class

- **Object** class enables methods to be written with a parameter of type **Object**:
  - A parameter of type **Object** can be replaced by an object of any class whatsoever (subtype polymorphism).
  - Example: method println **println(Object x)**
    - **Person p = new Person("Alice", 22);**
    - **Student s = new Student("Bob", 20, "UIC")**
    - **System.out.println(p);**
    - **System.out.println(s);**

toString method

- The java **toString()** method is used when we need a string representation of an object.

- This method can be overridden to customize the string representation of a specific class.

Example

```java
public class Person {
    ...
    @Override
    public String toString() {
        return "I am " + name + " and I am " + age;
    }
}
```

Person.java

```java
public class Test2 {
    public static void main(String arg[]){
        Person p = new Person("Alice", 22);
        System.out.println(p); // Same as:
        System.out.println(p.toString());
    }
}
```

I am Alice and I am 22
I am Alice and I am 22

Test2.java

equals method

- The **equals** method compares the object given as argument to the method with the object on which the method is called: **object1.equals(object2)** compares **object1** and **object2** using the **equals** method of **object1**.

- The result may or may not be the same as: **object2.equals(object1)** because the two objects might be from different classes with different **equals** methods.

equals method

- This method can be overridden to customize the comparison of objects for specific classes.
- The **equals** method should always have a parameter of type **Object** so that we can compare the current object with any other object from any other class:

```
public boolean equals(Object otherObject) {
    ...
}
```

Example

```java
public class Person {

    ...

    @Override
    public boolean equals(Object obj)
    {
        if (this == obj)   // test whether they are same object
            return true;
        if (obj == null)   // input object is null
            return false;
        if (obj instanceof Person) // input object is of type Person
        {
            Person p = (Person)obj; // type cassting
            // compare all the instance variables
            if (this.name.equals(p.getName())&& this.age == p.getAge())
                return true;
        }
        return false;
    }

}
```
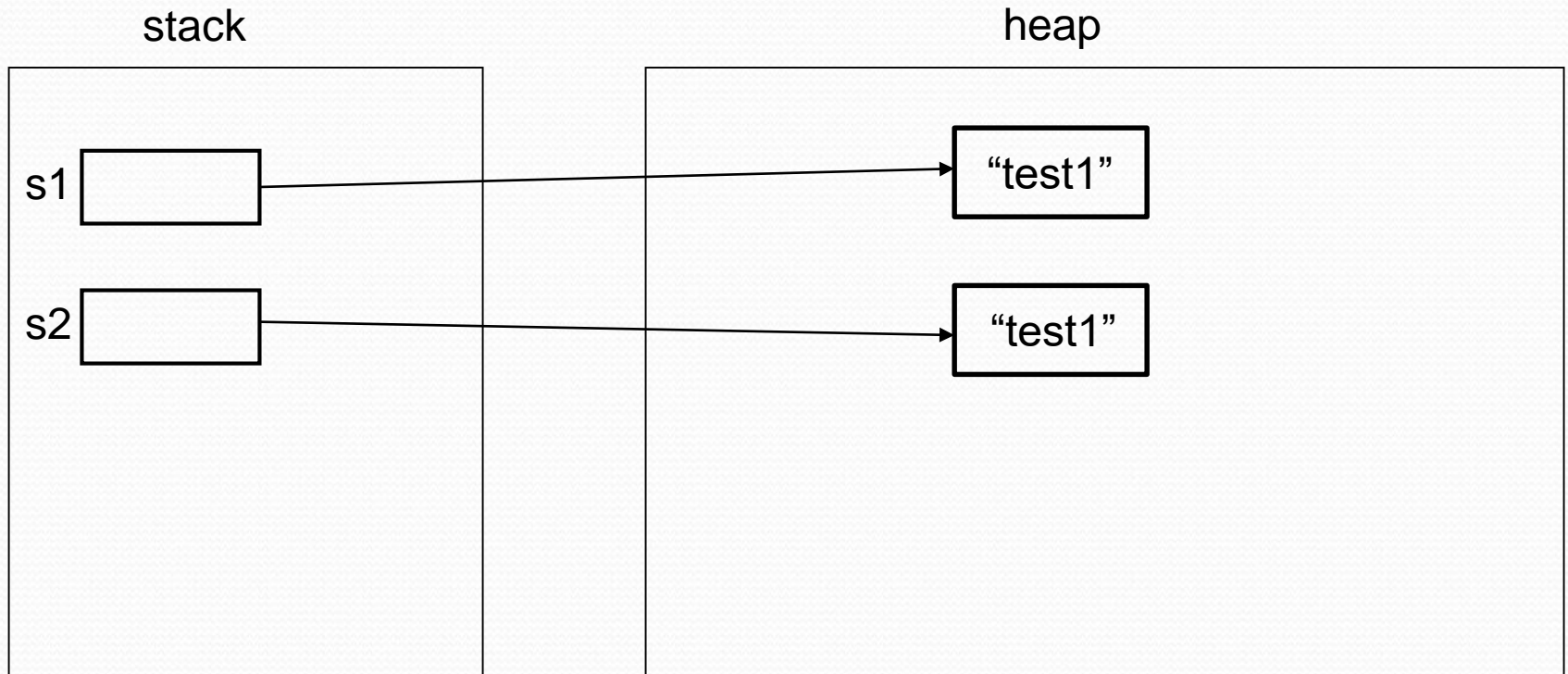
Person.java

Example

```java
public class Test3 {
    public static void main(String arg[]){
        String s1 = new String("test1");
        String s2 = new String("test1");
        System.out.println(s1 == s2);        // Same object?
        System.out.println(s1.equals(s2));  // Same values?
         Person p1 = new Person("Alice", 22);
        Person p2 = new Person("Alice", 22);
        Person p3 = new Person("Bob", 20);
        System.out.println(p1 == p2);        // Same object?
        System.out.println(p1.equals(p2));  // Same values?
        System.out.println(p1.equals(s1));  // Same values?
        System.out.println(p1.equals(p3));  // Same values?

    }
}
```
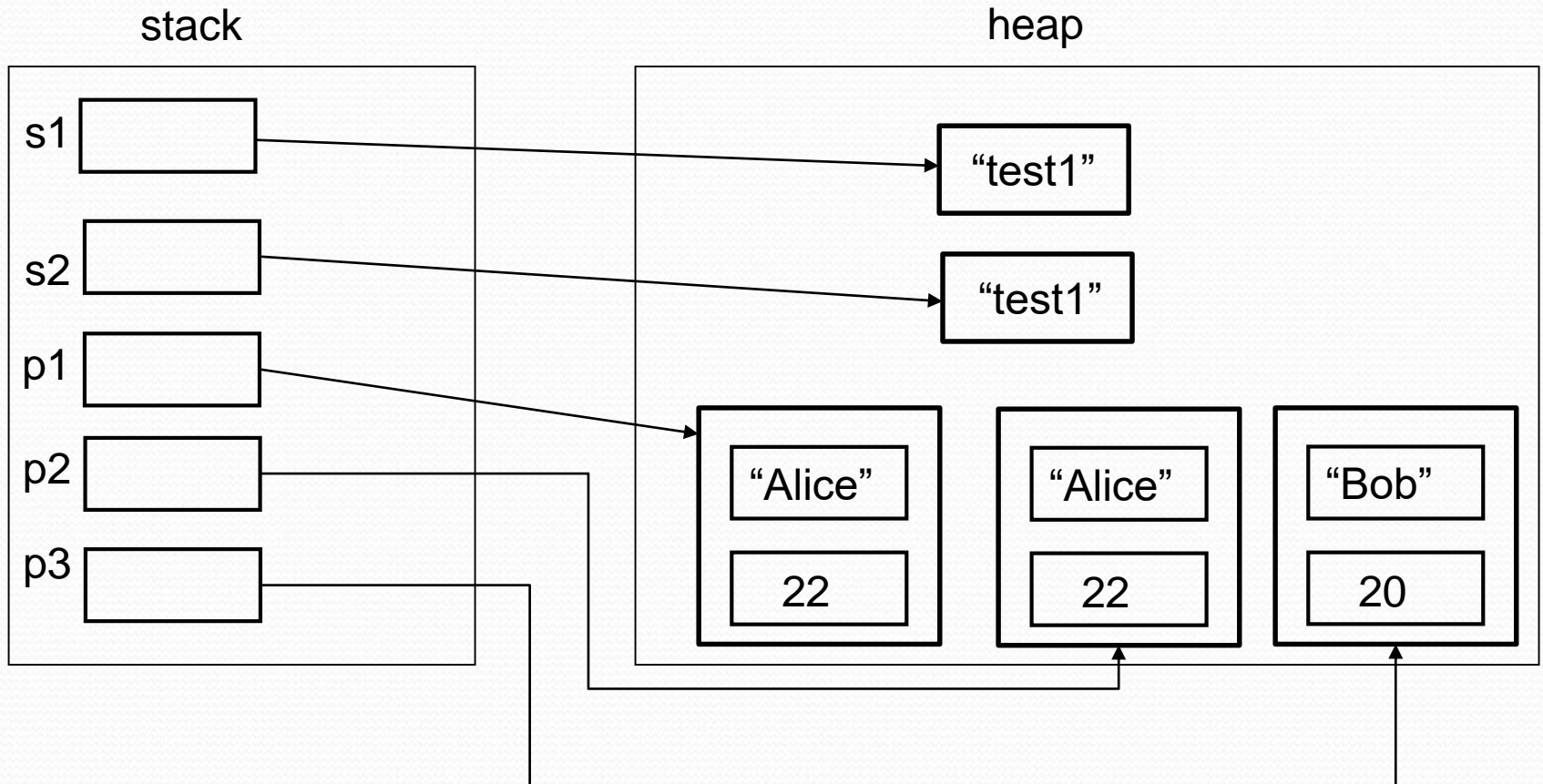
Test3.java

Example

```java
String s1 = new String("test1");
String s2 = new String("test1");
```

stack   heap

s1 [ ] ────────────────────→ ["test1"]

s2 [ ] ────────────────────→ ["test1"]

```java
System.out.println(s1 == s2);        // Same object?
System.out.println(s1.equals(s2));   // Same values?
```

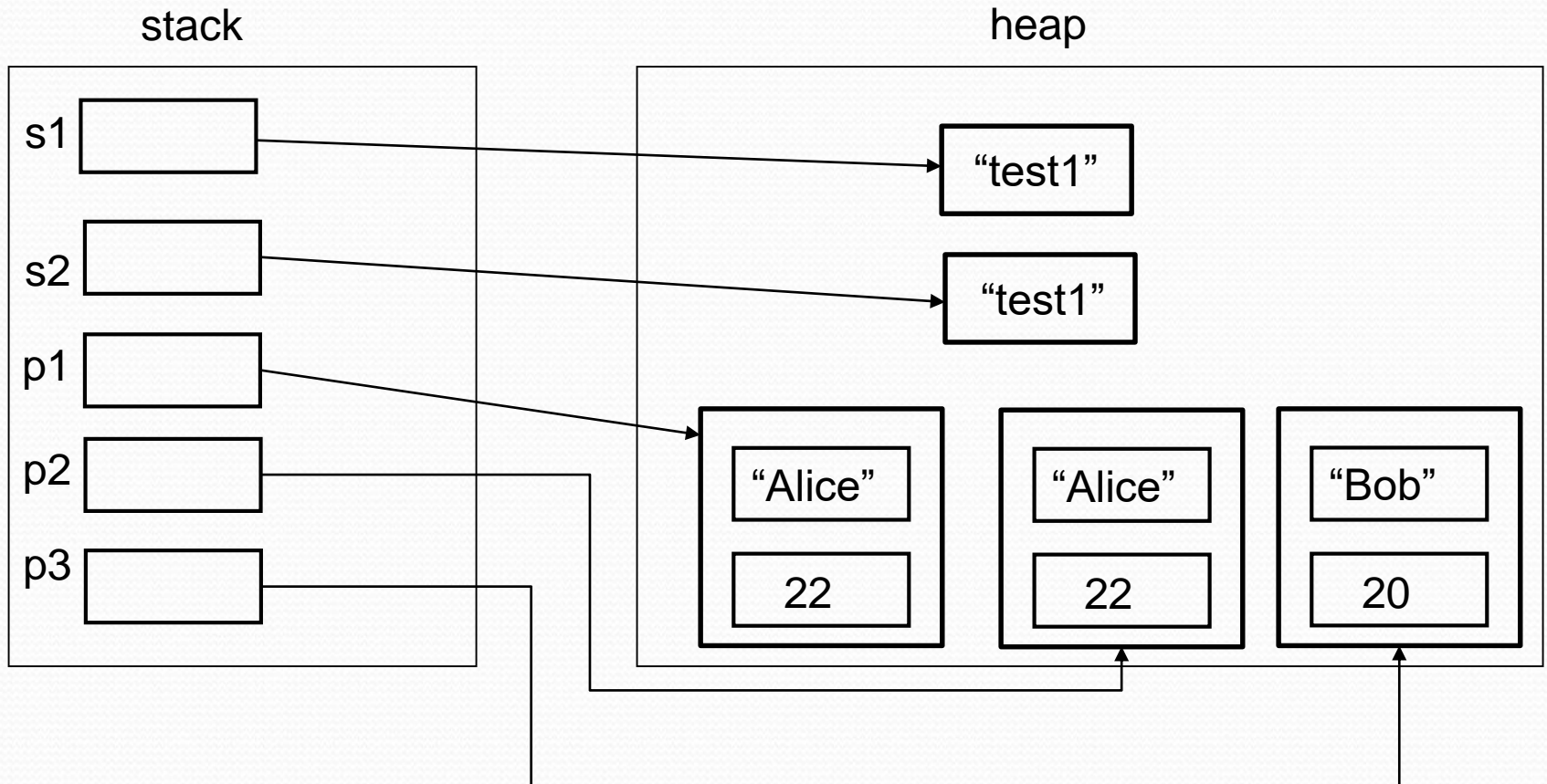# Example

```
Person p1 = new Person("Alice", 22);
Person p2 = new Person("Alice", 22);
Person p3 = new Person("Bob", 20);
```

stack   heap

s1   "test1"

s2   "test1"

p1

p2   "Alice"   "Alice"   "Bob"

p3   22   22   20

Example

```
System.out.println(p1 == p2);         // Same object?
System.out.println(p1.equals(p2));    // Same values?
System.out.println(p1.equals(s1));    // Same values?
System.out.println(p1.equals(p3));    // Same values?
```

Dynamic Binding

- Connecting a method call to a method body is called *binding*.

- Automatically selecting the appropriate method at runtime is called dynamic binding or late binding. When binding is performed before the program is run (by the compiler, if there is one), it's called early binding.

- All method binding in Java uses late binding unless the method is **static** or **final** (**private** methods are implicitly **final**).

Dynamic Binding

- When the program runs and uses dynamic binding to call a method, then the virtual machine must call the version of the method that is appropriate for the actual type of the object.
  - If the actual type class defines the matched method, then the method is called.
  - Otherwise, the superclass of the actual type class is searched, and so on.
  - Finally, the **Object** class is searched.

```
Person p = new Student("Alice", 22,"UIC");
System.out.println(p.getInfo());
// First, search Student class for getInfo() method
// If not found, search Person class for getInfo() method
```

Dynamic Binding and Method Overriding

- Multiple implementations of the same method occur in different classes along the same hierarchy.

- A child class overrides the implementation of a method provided by its parent class.

- Example:
  `Student.getInfo()` overrides `Person.getInfo()`

- Dynamic binding then uses the first method with the right signature when searching bottom-up in the class hierarchy.

Overriding vs. Overloading

Do not confuse overriding with overloading:

- Overriding takes place in the subclass: a new method with the same name and <u>same signature</u> hides the method inherited from the parent class.

- Overloading takes place in the same class: a new method with the same name but a <u>different signature</u> is defined and does not hide the existing method.

- Note that the method signature consists of the method name and the parameter list

Subtyping Polymorphism

- Subtyping polymorphism: an object from a subclass can be used as if it were an object from a superclass.
  - Also called the *Liskov substitution principle*.
- **Student** inherits all the non-private methods of **Person**.
- So a **Student** object has all the methods necessary to act as a **Person** object!
- So Java allows us to use a **Student** object in any place where a **Person** object would work too.

Example

```
public class Test4 {
    public static void main(String arg[]){
        Student s = new Student("Alice", 22, "UIC");
        Person p = s; // Using s as a Person object.
        System.out.println("Person's name: " + p.getName());
        System.out.println("Person's age: " + p.getAge());
        // Student's getInfo method is used, not Person's!
        System.out.println("Person's info: " + p.getInfo());
    }
}
```

Person's name: Alice
Person's age: 22
Person's info: Student Alice is 22
and at UIC

When calling the `p.getInfo()` method, dynamic binding starts searching in the `Student` class, not in the `Person` class, even though `p` is of type `Person`, because the object really is an object from the `Student` class. So Alice's school is printed!

Example

```
public class Test4 {
    public static void main(String arg[]){
      Student s = new Student("Alice", 22, "UIC");
      Person p = s; // Using s as a Person object.
       ...
      System.out.println("parent's info:"+ p.getParentInfo());
    }
}
```

The method getParentInfo() is undefined for the type Person

The **Student** object has a **getParentInfo** method but you cannot use this method when using the object with the **Person** type, because the **Person** class does not have such a method. The type system of Java forbids this, even though **p** and **s** both refer to the same object!

Example

```
public class Test4 {
    public static void main(String arg[]){
        Student s = new Student("Alice", 22, "UIC");
        Person p = s; // Using s as a Person object.
        ...
        Object o = s; // Using s as an Object object.
        // Person's toString method is used, not Object's!
        System.out.println(o); // Same as:
        System.out.println(o.toString());
    }
}
```
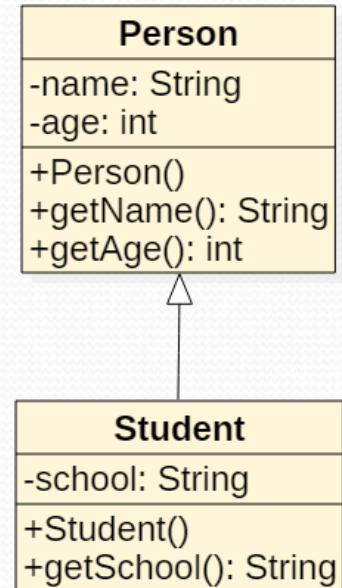
- **Student** is a subclass of **Person** (explicitly) and **Person** is a subclass of **Object** (implicitly), therefore an object from the **Student** class can be used as an object of the **Person** class or as an object of the **Object** class.

- As before, when calling the **o.toString()** method, dynamic binding starts searching in the **Student** class, not in the **Object** class. The **Student** class does not define a **toString** method but the **Person** class does, so method **toString()** of class Person is then called here.

Type Casting

- **Object typecasting** is when the type used for an object is changed, usually by assigning the object to a variable of a different type.

- The cast does not change the object itself, it only changes the type through which the object is used.

- There are two types of casts:

  - Upcast: the type of the object is changed to the type of a superclass.

  - Downcast: the type of the object is changed to the type of a subclass.
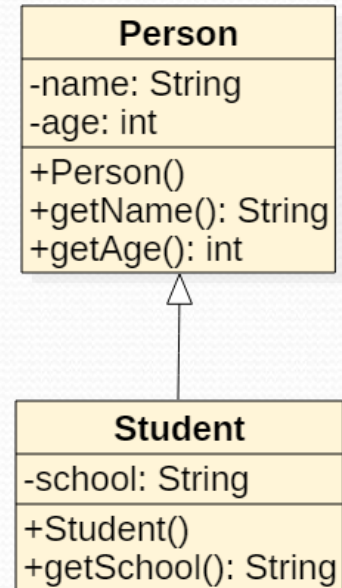
Upcast Example

```java
public class Test5 {
    public static void main(String arg[]){
        Student s = new Student("Alice", 22, "UIC");
        Person p1 = s; // Implicit upcast.
        // Or:
        Person p2 = (Person)s; // Explicit upcast.
    }
}
```

Person
-name: String
-age: int
+Person()
+getName(): String
+getAge(): int

Student
-school: String
+Student()
+getSchool(): String

- The upcast can be implicit (added by Java) or explicit (added by the user).
- All upcasts always work, because of subtyping polymorphism.
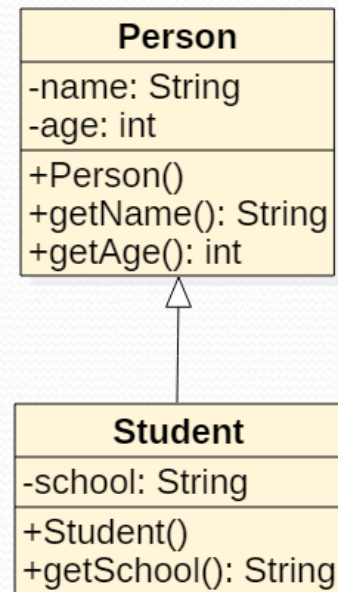
Downcast Example

```java
public class Test5 {
    public static void main(String arg[]){
        Student s = new Student("Alice", 22, "UIC");
        Person p1 = s; // Implicit upcast.
        // Or:
        Person p2 = (Person)s; // Explicit upcast.
        // Explicit downcast.
        Student s2 = (Student)p1;
    }
}
```

Person
-name: String   -age: int
+Person()   +getName(): String   +getAge(): int

Student
-school: String
+Student()   +getSchool(): String

- The downcast must be explicit (added by the user).
- A downcast only works if an object is downcasted back to its original type after there was an upcast!

Downcast Problem

```java
public class Test5 {
    public static void main(String arg[]){
        ......
        Person p = new Person("Alice", 22);
        // Explicit downcast.
        Student s2 = (Student)p;
    }
}
```

Person
-name: String
-age: int
+Person()
+getName(): String
+getAge(): int

Student
-school: String
+Student()
+getSchool(): String

- It is not possible to transform a person into a student by doing a downcast from **Person** to **Student**. The Java compiler will accept the downcast but the JVM will detect the problem at runtime and stop the program!

- This is because a **Person** object might not have all the required methods (such as **getSchool**) to work as a **Student** object.

Downcast Problem

```
class Person { ... }
class Student extends Person { ... }
class Teacher extends Person { ... }
...
Student s = new Student("Alice", 22, "UIC");
Person p = s; // Implicit upcast.
Teacher t = (Teacher)p; // Explicit downcast.
```

- It is not possible to transform a student into a teacher by doing an upcast from **Student** to **Person** followed by a downcast from **Person** to **Teacher**. The Java compiler will accept the downcast but the JVM will detect the problem at runtime and stop the program!

- This is because a **Student** object might not have all the required methods to work as a **Teacher** object.

instanceof

- Since the type used for an object can be changed, how do we know what kind of object it really is?
- The **instanceof** operator can be used to test if an object is from a class which is a descendent of the tested class
  - **object *instanceof* class**
    - For compile, class could be parent class/children class, otherwise *"incompatible conditional operand types"*
    - For run time, return true if the class is the parent class of the object type; otherwise return false

instanceof

```java
public class Test6 {
    public static void main(String arg[]){
        Person p = new Person("Alice", 22);
        System.out.println(p instanceof Object);  // true
        System.out.println(p instanceof Person);  // true
        System.out.println(p instanceof Student); // false
        Student s = new Student("Alice", 22, "UIC");
        System.out.println(s instanceof Teacher); // error
    }
}
```

Incompatible conditional operand types Student and Teacher

Summary

- **`final`** modifier
- **`Object`** class
- **`toString`** and **`equals`** methods
- Dynamic binding
- Subtyping polymorphism
- Type casting: upcasts and downcasts
- **`instanceof`** operator