

# CIS 3022 – Lecture #39 – 201011.24

## Quiz #4

- \* Name and lab period in upper right corner
- \* Assume a class Polynomial (as in last week's lab); implement the method `degree()`, which quickly and efficiently returns the degree of the **Polynomial** on which it is called.

## Implementation inheritance

### Person-Student example cont.

Last time we ended with... now let's include

```
class Student extends Person {
    private int id;

    public Student( String name, int age, int id ) {
        super( name, age );

        this.id = id;
    }

    public int getID() {
        return id;
    }
}
```

### In action

```
Student chris = new Student( "Chris", 19, 1234 );
```

then all of the follow are legal

```
System.out.println( chris.getName() );
System.out.println( chris.getAge() );
System.out.println( chris.getID() );
```

```
Student stillChris = chris;
Person chrisToo = chris;
```

then which of the following are legal?

```
System.out.println( stillChris.getName() ); ✓
System.out.println( stillChris.getAge() ); ✓
System.out.println( chris.getID() ); ✓
```

```

System.out.println( chrisToo.getName() ); ✓
System.out.println( chrisToo.getAge() ); ✓
System.out.println( chrisToo.getID() ); ✗

chris = stillChris; ✓
chris = chrisToo; ✗

```



class **Person** neither defines nor inherits the operation **getID()**, therefore, when the *compiler* sees you are trying to call an operation that the variable's *declared type* does not support, it gives an error.

While *all* instances of *class Student* are also instances of *type Person*, *not* all instance of *type Person* are of *type Student*

#### IMPORTANT CONCEPT:

An object may be an instance of *multiple types*, but it is an instance of *exactly one class*—that being the name of the constructor that was used with the keyword **new**.

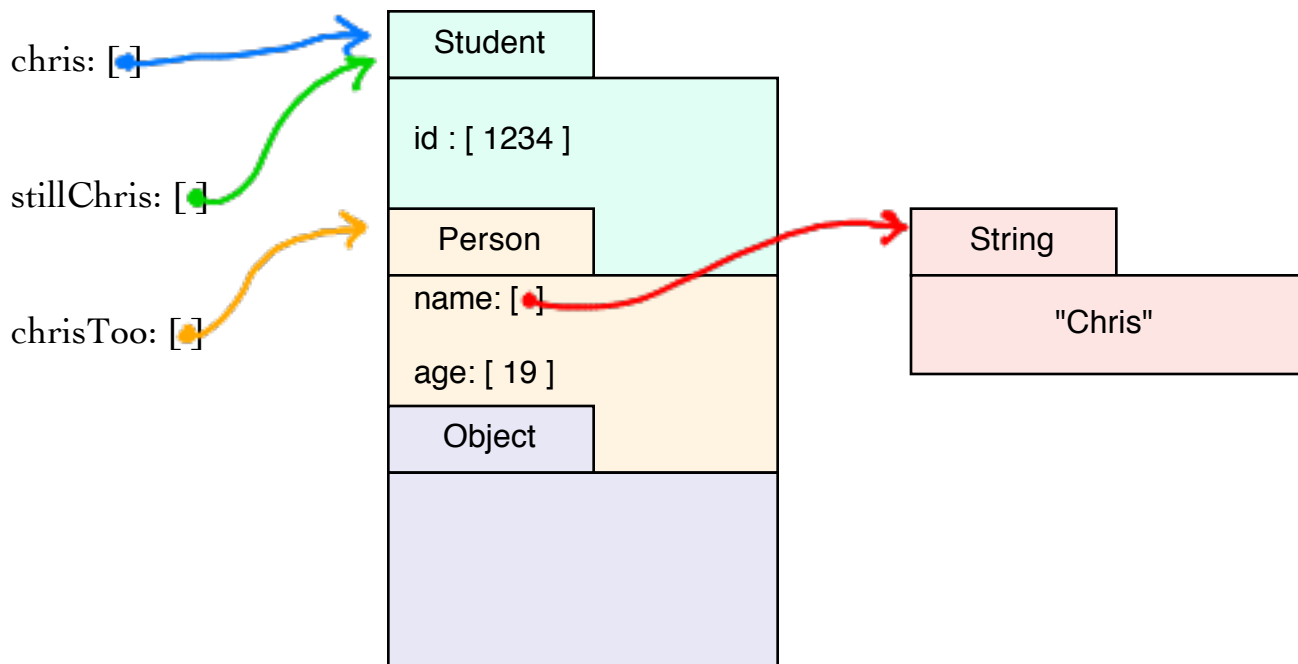
#### DaRN

```

Student chris = new Student( "Chris", 19, 1234 );
Student stillChris = chris;
Person chrisToo = chris;

```

when *executed*, produces...



From the *compiler's* perspective, it doesn't know that at *runtime* **chrisToo** will really be pointing at a **Student** instance. We can, however, tell the compiler that the variable is pointing to something of a *more specific type* [e.g., type **Student** is a more

specific type than a **Person**]. This is accomplished by *casting*

```
chris = (Student) chrisToo; ✓
```

"Even though **chrisToo**'s declared type is **Person**, we the programmer are telling the compiler we know that it's really a *special kind of Person*, specifically, a **Student**"

```
System.out.println( (Student) chrisToo).getID() ); ✓
```

↙ parentheses required ↘

the *member selection operator* (aka, "dot operator") has a higher precedence than the cast, thus

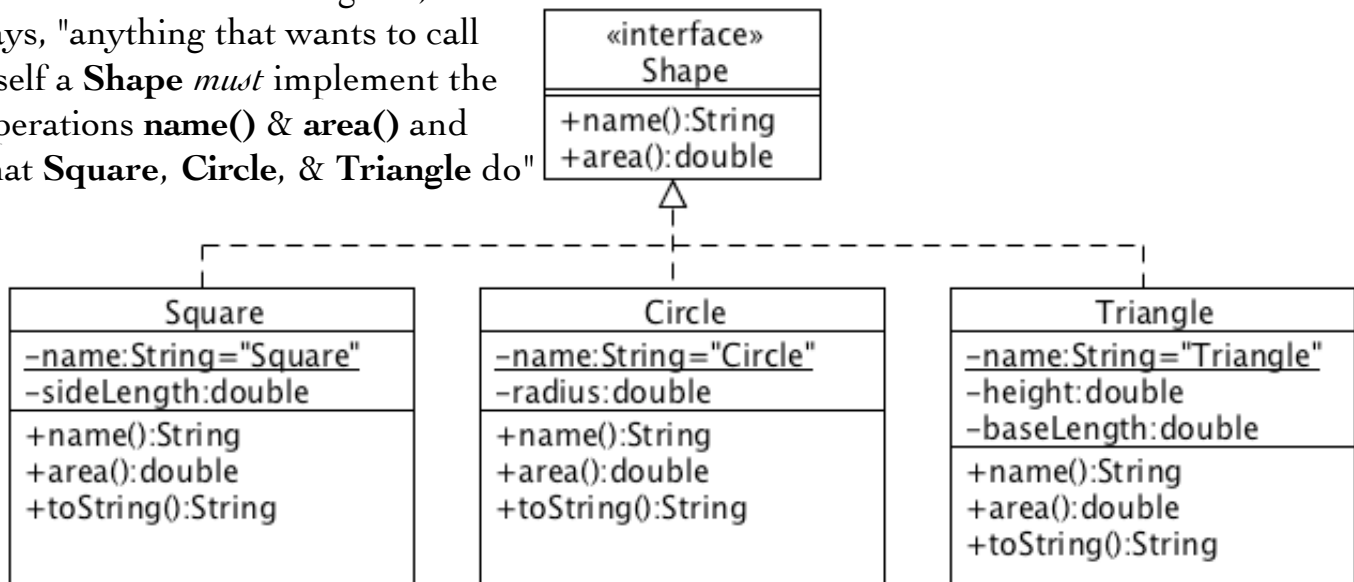
```
System.out.println( (Student) chrisToo.getID() );
```

error

means, "cast the *return value* of calling `getID()` on the object referenced by **chrisToo**" (which we know won't compile).

## Interface inheritance II

Consider this UML diagram, which says, "anything that wants to call itself a **Shape** *must* implement the operations `name()` & `area()` and that **Square**, **Circle**, & **Triangle** do"



UML notes

- \* class members are underlined (they are declared **static** in Java)
- \* interfaces are tagged with the stereotype «**interface**»
- \* *interface inheritance* is denoted by a dashed line, with a hollow triangle arrowhead