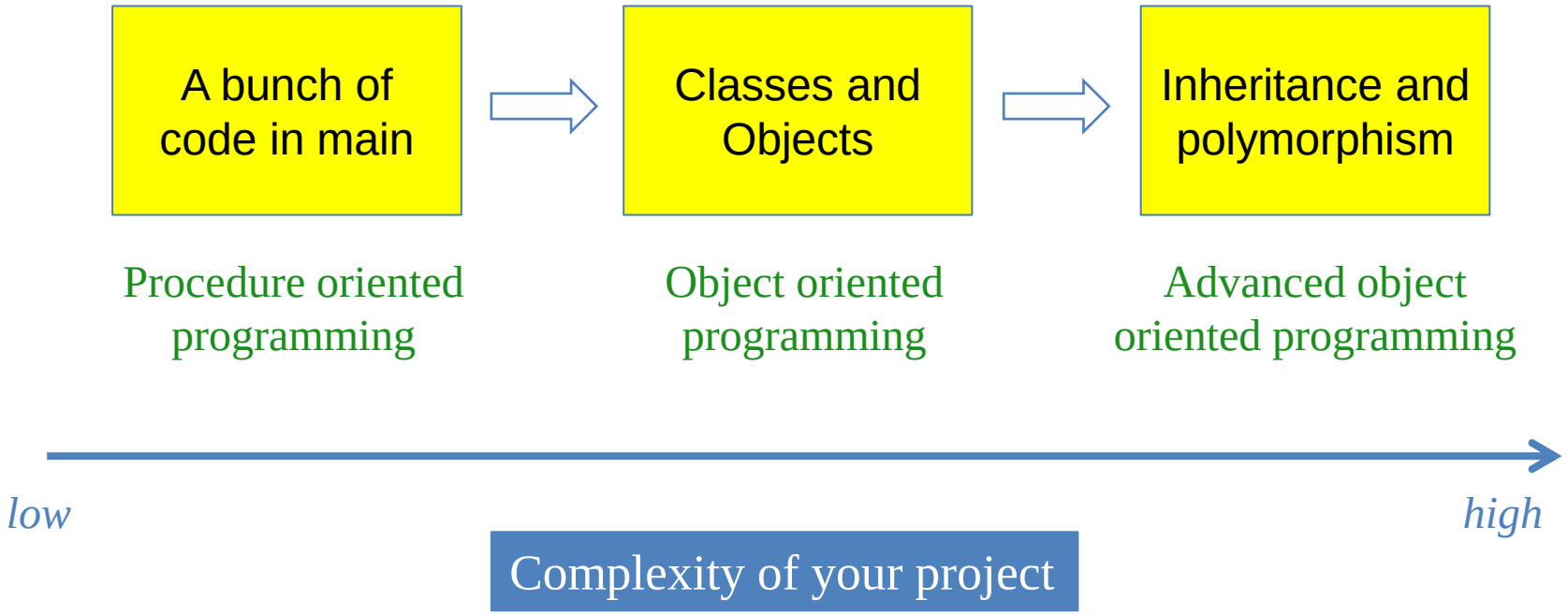
Lecture 3
Inheritance and Polymorphism

Zonghua Gu (based on Jianchen Shan)
Department of Computer Science
Hofstra University

Lecture Goals

- Explain the value of inheritance
- Use UML Diagrams to display class hierarchies
- Explain an "is-a" relationship between classes
- Understand that object construction occurs from the inside out
- Explain the purpose and implementation of polymorphism
- Create methods which override from a superclass
- Use casting of objects to aid the compiler
- Describe abstract classes and interfaces and decide which one to use

General Motivation



Motivation for Inheritance

Fully written Person class

```
public class Person {
        private String name;
        // more code here
}
```

Potential Problem

Now needs to handle:

1. Students
2. Faculty

they behave differently

Potential Solution 1

```
public class Person
{
        private String name;
        private boolean student;
        public person(boolean s)
        {
                this.student = s;
        }
}
```

Now in every method, I can just do this:

```
if (student)
        // code for students
      else
        // code for faculty
```

Motivation for Inheritance (Contd.)

Fully written Person class

```
public class Person {
        private String name;
        // more code here
}
```

Potential Problem

Now needs to handle:

1. Students
2. Faculty

they behave differently

Potential Solution 1 - Problems

```
public class Person
{
        private String name;
        private boolean student;
        private boolean graduate;
        private boolean fulltime;
        // more code here
}
```

different students behave differently

Each method becomes:

```
if (student)
        if (graduate && fulltime)
                // some code
        else if (!graduate)
                // more code
```

Motivation for Inheritance (Contd.)

Fully written Person class

```
public class Person {
        private String name;
        // more code here
}
```

Potential Problem

Now needs to handle:

1. Students
2. Faculty

they behave differently

cannot just copy

tedious

potential mistake

Potential Solution 2 - Problems

```
public class Student
{
        private String name;
        private String firstname;
        private String lastname;
}
```

```
public class Faculty
{
        private String name;
}
```

hard to keep common code consistent

```
// in main
Person persons[];
Student students[];
Faculty faculty[];
```

cannot use this anymore

no clean way single array of everyone for thing like sorting by join date

Motivation for Inheritance (Contd.)

- What do we want then?

1. Keep common behavior in one class
2. Split different behavior into separate classes
3. Keep all of the objects in a single data structure

The answer is Inheritance

Details of Inheritance: Extend Keyword

common code

```
public class Person {
        private String name;
        public getName() { return name; }
        // more code here
}
```

base/super class

diverging code

```
public class Student   extend Person  {
        private String name;
        // more code here
}
```

derived/sub class

"extend" means "inherit from"

diverging code

```
public class Faculty   extend Person  {
        private String name;
        // more code here
}
```

derived/sub class

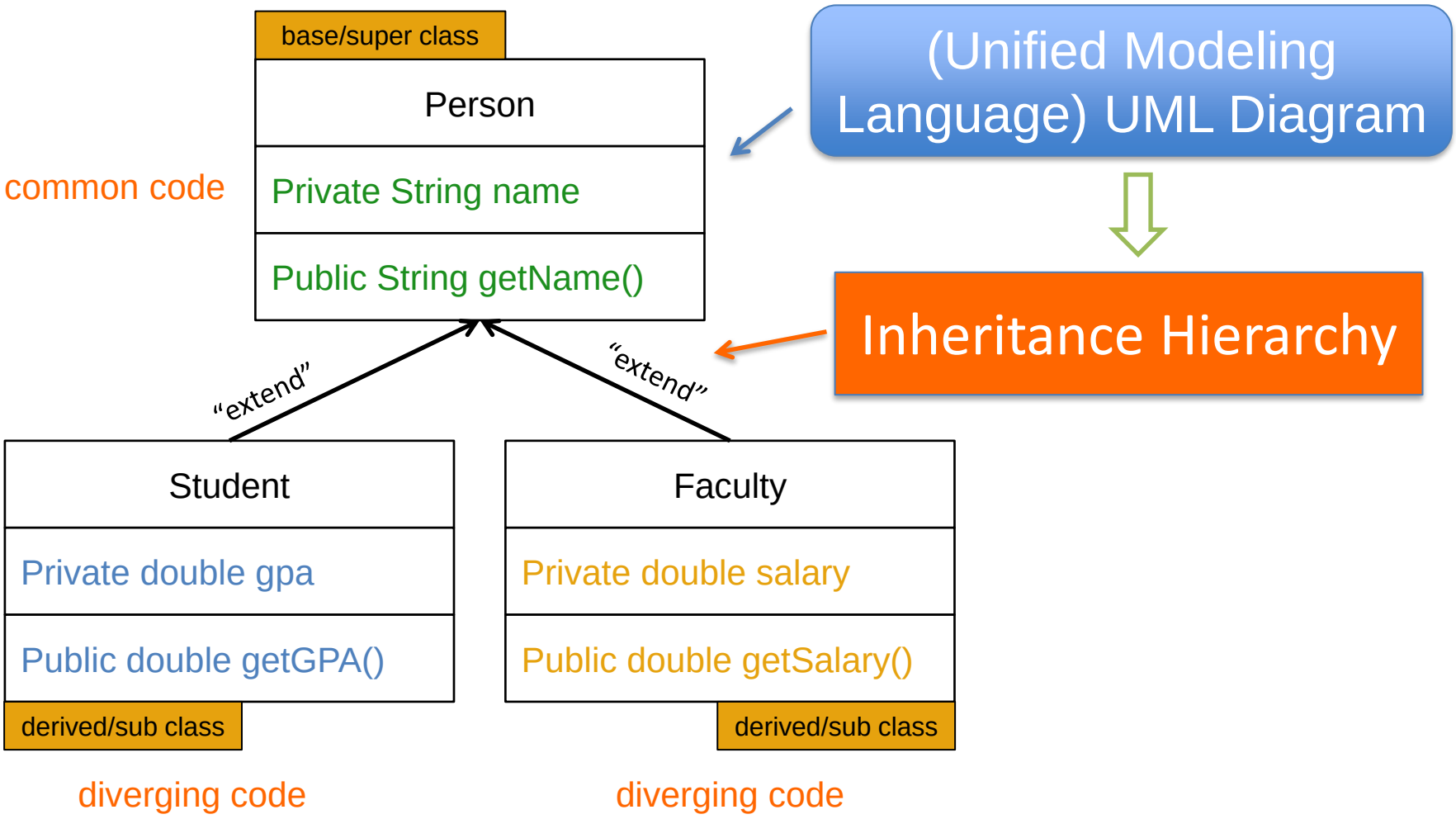
What is inherited?

- Public instance variables
- Public methods
- Private instance variables

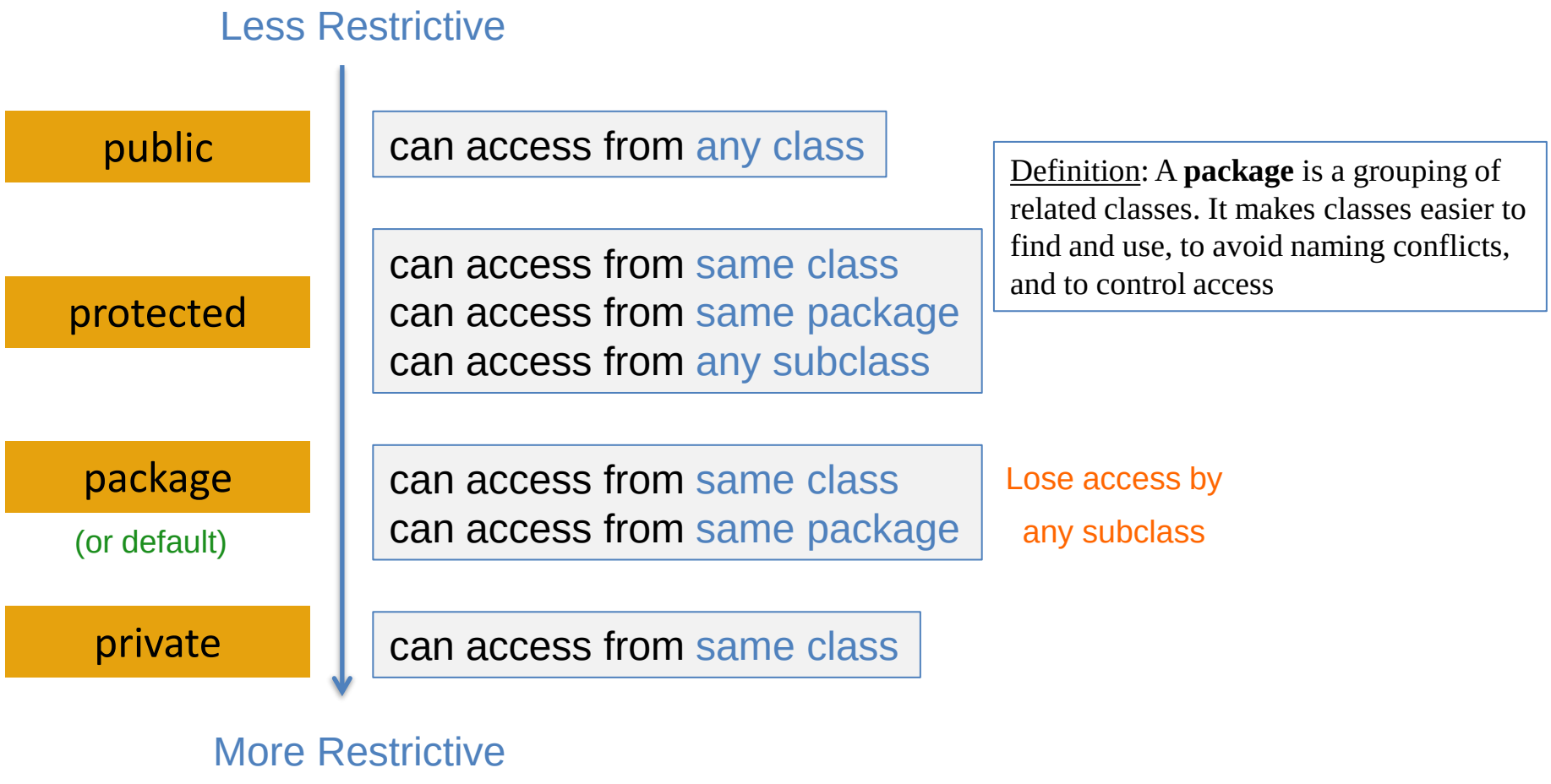
Private variables can be accessed only through public methods!

Private methods cannot be inherited!

Illustrate Inheritance Hierarchy with UML Diagrams



Definitions of Visibility Modifiers



Definition: A **package** is a grouping of related classes. It makes classes easier to find and use, to avoid naming conflicts, and to control access

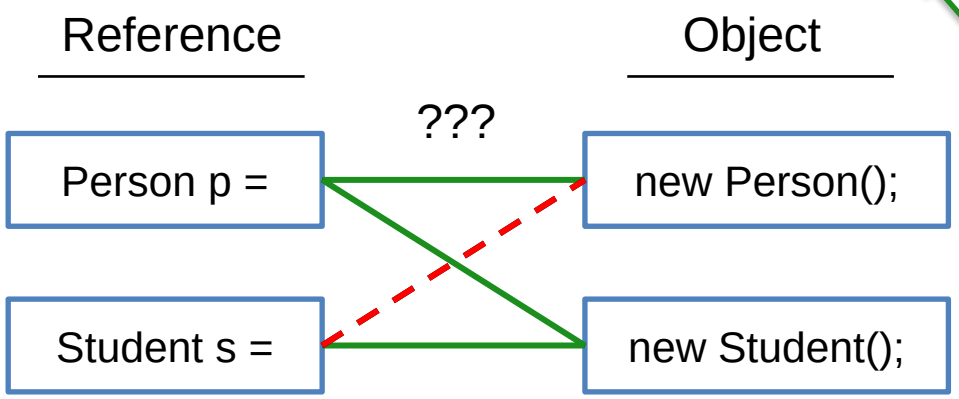
Lose access by any subclass

Rule of thumb: Make member variables private (and methods either public or private)

"Is-a" Relationship Between Reference and Object Type

- What do we want then?

1. Keep common behavior in one class
2. Split different behavior into separate classes
3. Keep all of the objects in a single data structure



the code compiles and works just fine

```
// in main
Person[] p = new Person[3];
p[0] = new Person();
p[1] = new Student();
p[2] = new Faculty();
```

A Person array CAN store Student and Faculty objects

```
Person p = new Person();
Student s = new Student();
Person p = new Student();
Student s = new Person();
```

✓ A Person "is-a" Person

✓ A Student "is-a" Student

✓ A Student "is-a" Person

✗

Why? Well, not all the features of a Student are necessarily within a Person.

Some Practices

```
public class Person {
        private String name;
        public String getName() {return name;}
}
```

```
public class Student extends Person {
        private int id;
        public int getID() {return id;}
}
```

```
public class Faculty extends Person {
        private String id;
        public String getID() {return id;}
}
```

```
Student s = new Student();
Person p = new Person();
Person q = new Person();
Faculty f = new Faculty();
Object o = new Faculty();
```

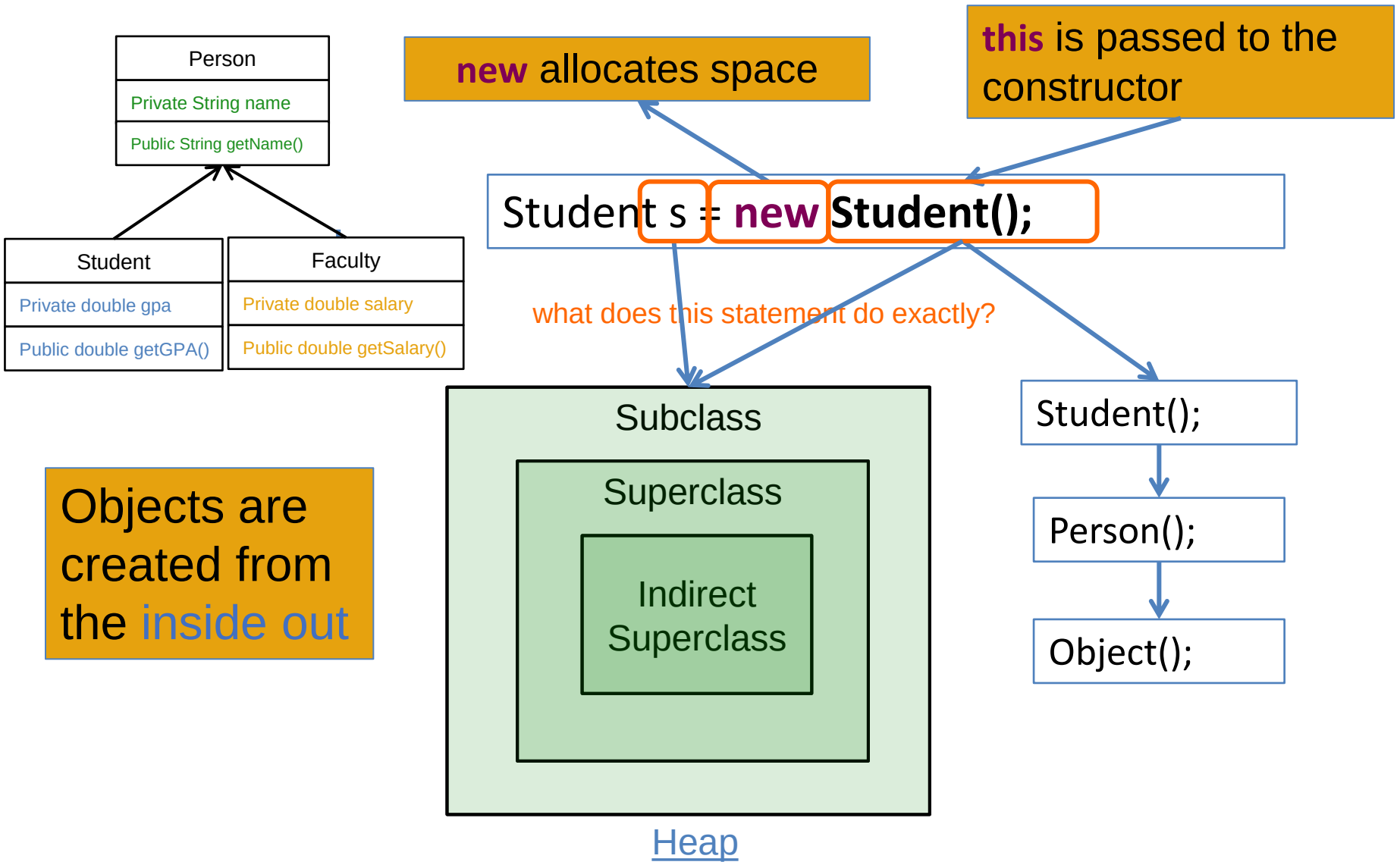
Which of the following lines of code, when executed in sequence, will cause an error?

String n = s.getName();	✓
p = s;	✓
int m = p.getID();	✗
f = q;	✗
o = s;	✓

```
int m = ((Student)p).getID();
```

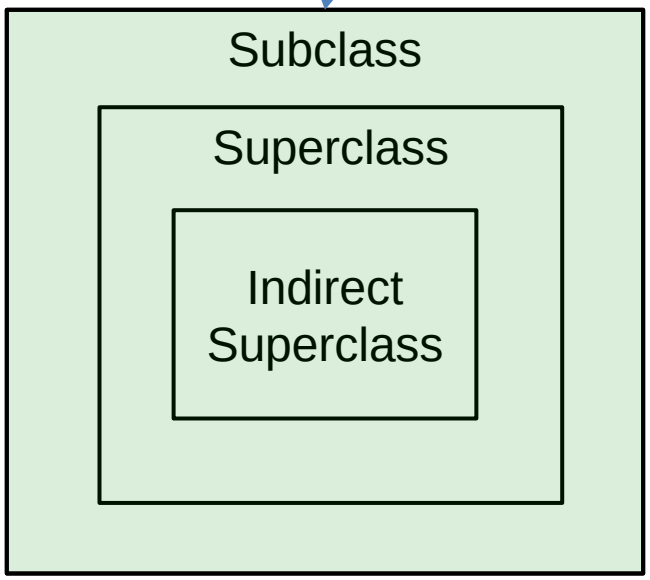
do casting and compiler would trust you

Revisit Object Construction with Inheritance



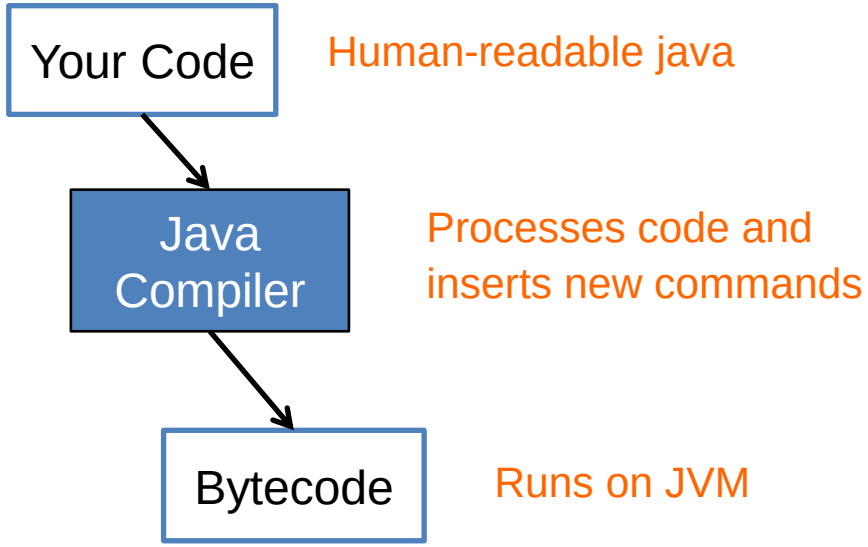
Object Construction with Compiler Support

```
Student s = new Student();
```



Wait, I don't remember extending Object...

compiler did that for you!



Compiler's Rules

```
public class Person {
        private String name;
}
```

```
public class Person extends object {
        private String name;
}
```

```
public class Person extends object
{
        private String name;
        public Person() {

        }
}
```

Added by compiler

```
public class Person extends object
{
        private String name;
        public Person() {
                super();
        }
}
```

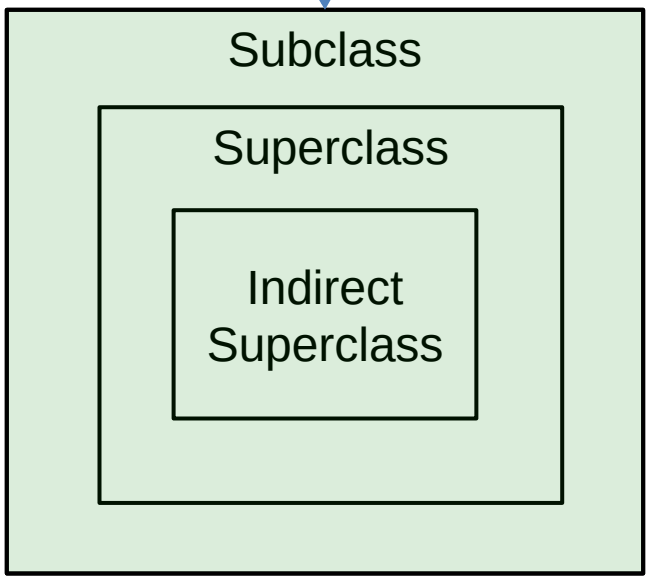
Rule #1 - No superclass?
Compiler inserts: extends Object

Rule #2 - No constructor?
Java gives you one for you.

Rule #3 - 1st Line must be:
this($args_{opt}$) Same class constructor call
or
super($args_{opt}$) Base class constructor call
Otherwise, Java inserts:
"super();"

Object Construction with Compiler Support (Contd.)

```
Student s = new Student();
```



Has super class:
1st rule doesn't apply

```
public class Student extends Person
{
}
```

```
public class Student extends Person
{
        public Student() {
                super();
        }
}
```

Has no constructor:
2nd rule DOES apply

Needs to call super's default constructor:
3rd rule DOES apply

But how do we initialize name ?

Compiler ensures object construction occurs from the inside out

Variable Initialization in a Class Hierarchy

```
public class Person extends Object {
        private String name;
        public Person() {
                super();
        }
}
```

Initialize `name` variable in `Person`

```
public class Person extends Object {
        private String name;
        public Person(String n) {
                this.name = n;
                super();
        }
}
```

ERROR! `super()` has to be the first line!

```
public class Person extends Object {
        private String name;
        public Person(String n) {
                super();
                this.name = n;
        }
}
```

```
public class Student extends Person
{
        public Student() {
                super();
        }
}
```

Initialize `name` variable in `Student`

```
public class Student extends Person
{
        public Student(String n) {
                super();
                this.name = n;
        }
}
```

but no getters and setters

ERROR! `name` is private

```
public class Student extends Person
{
        public Student(String n) {
                super(n);
        }
}
```

initialize without public setters

Variable Initialization in a Class Hierarchy (Contd.)

```
public class Student extends Person
{
        public Student(String n) {
                super(n);
        }
}
```

Add a no-arg constructor

```
public class Student extends Person
{
        public Student(String n) {
                super(n);
        }
        public Student() {
                super("Student");
        }
}
```

should not jump to the super class if there is same class constructor

Use super class constructor

```
public class Student extends Person
{
        public Student(String n) {
                super(n);
        }
        public Student() {
                this("Student");
        }
}
```

Use our same class constructor

Some Practices

```
public class Person {
        private String name;
        public Person(String n) {
                this.name = n;
                System.out.print("#1 ");
        }
}
```

```
public class Student extends Person {
        public Student() {
                this("Student");
                System.out.print("#2 ");
        }
        public Student(String n) {
                super(n);
                System.out.print("#3 ");
        }
}
```

Suppose you call:

Student s = new Student();

What is the order of statements printed?

A. #1 #2 #3

B. #1 #3 #2

C. #3 #2 #1

D. #3 #1 #2

E. None of the above

#1 #3 #2

Some Practices (Contd.)

```
public class Person {
        private String name;
        public Person(String n) {
                super();
                this.name = n;
        }
        public void setName(String n) {
                this.name = n;
        }
}
```

```
public class Student extends Person {
        public Student() {
                this.setName("Student");
        }
}
```

Super()

Suppose you call:

Student s = new Student();

What will be the name variable for this object?

A. "student"

B. "Undefined"

C. null

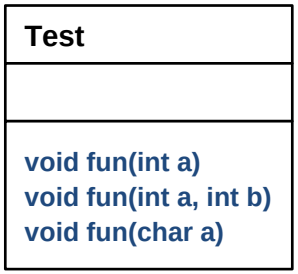
D. Compile Error

E. Runtime Error

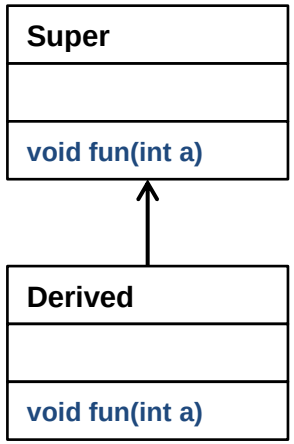
ERROR: Implicit super constructor Person() is undefined. Must explicitly invoke another constructor

Method Overriding

- **Overloading**: **Same class** has same method name with **different** parameters
- **Overriding**: **Subclass** has same method name with the **same parameters** as the superclass



Overloading



Overriding

- What do we want then?

1. Keep common behavior in one class
2. Split different behavior into separate classes
3. Keep all of the objects in a single data structure

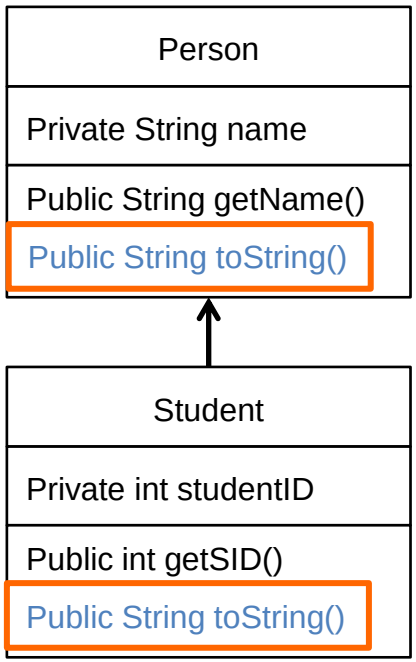
A `private` method cannot be overridden since it is not visible from any other class. When we use `final` specifier with a method, the method cannot be overridden in any of the inheriting classes. Since private methods are inaccessible, they are implicitly final in Java.

An Example: Object Class

String	toString() Returns a string representation of the object.

All java classes can override it

Override Object's `toString()` method for `Person` class



Override Object's `toString()` method for `Student` class

```
public class Person {
        private String name;
        // more code here
        public String toString() {
                return this.getName();
        }
        public static void main(String[] args) {
                Person p = new Person("Tim");
                System.out.println(p.[redacted]);
        }
}
```

```
$ Tim
```

`println` automatically calls `toString()`

```
public class Student extends Person{
        private int studentID;
        // more code here
        public String toString() {
                return this.getSID() + ": " +
                       [redacted];
        }
        public static void main(String[] args) {
                Student s = new Student("Cara", 1234);
                System.out.println(s);
        }
}
```

```
$ 1234: Cara
```

what if `Person` changes?

Introduce to Polymorphism

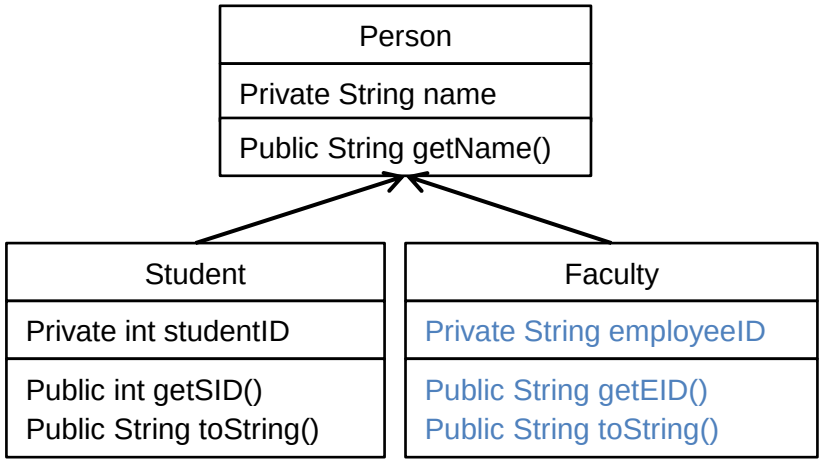
```
Person s = new Student("Cara", 1234);
System.out.println(s);
```

The dynamic (or actual) type of the object is Student, so its `toString()` method will be called.

```
$ 1234: Cara
```
✓

```
$ Cara
```
✗

For superclass reference to subclass object, the actually called method depends on the dynamic type. This is referred as **Polymorphism**.




```
Person p[] = new Person[3];
p[0] = new Person( "Tim" );
p[1] = new Student( "Cara", 1234 );
p[2] = new Faculty( "Mia", "ABCD" );
for(int i = 0; i < p.length; i++)
{
  System.out.println(p[i]);
}
```

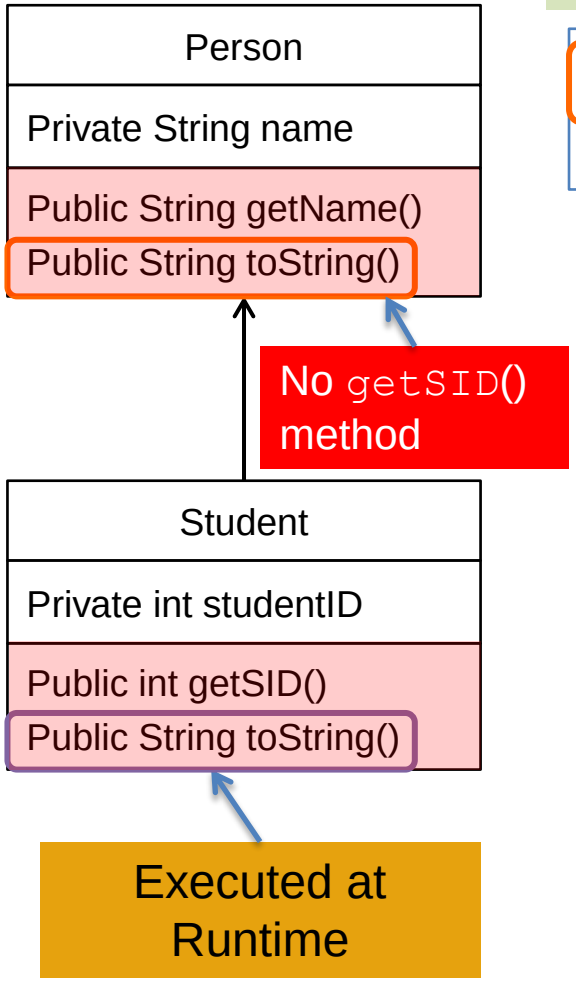
```
$ Tim
$ 1234: Cara
$ ABCD: Mia
```

Polymorphism allow us to keep all of our objects in one big collection, and then call appropriate methods on every element

Java Polymorphism Fully Explained In 7 Minutes
https://www.youtube.com/watch?v=jhDUxynEQRI

Polymorphism Implementation: Compile Time and Run Time Rules

Think like a compiler, act like a runtime environment.



1. compiler interprets the code → 2. the runtime environment executes the interpreted code

```
Person s = new Student("Cara", 1234);
s.toString();
```

String toString()

Method Signature

Compile Time Rules:

- Compiler ONLY knows reference type
- Can only look in reference type class for method
- Outputs a method signature

Run Time Rules:

- Follow exact runtime type of object to find method
- Must match compile time method signature to appropriate method in actual actual object's class

```
Person s = new Student("Cara", 1234);
s.getSID();
```

Compile Time Error!

needs explicit casting

Use Casting of Objects to Aid the Compiler

Two types of casting:

- Automatic type promotion (like `int` to `double`)

  Superclass superRef = new Subclass();

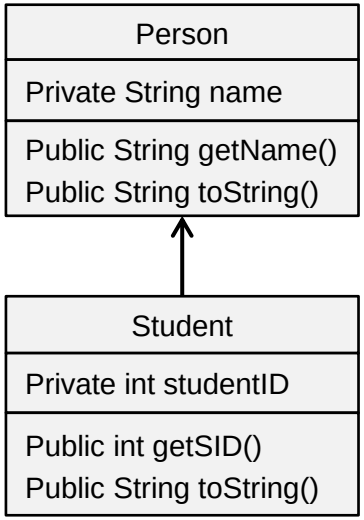
- Explicit casting (like `double` to `int`)

  Subclass ref = (Subclass)superRef;

Widening

Narrowing

BE CAREFUL:
Compiler trusts you




```
Person s = new Student("Cara", 1234);
s.getSID();
((Student)s).getSID();
```

This works

```
Person s = new Person(”Tim”);
((Student)s).getSID();
```

break the trust

Runtime Error!
java.lang.ClassCastException: From Person to Student

Runtime type check - `instanceof`

- Provides runtime check of **is-a** relationship

```
if(s instanceof Student )
{
        // only executes if s is-a
        // Student at runtime
        ((Student)s).getSID();
}
```

Abstract Classes and Interfaces

- `Person` - Campus Accounts
  - "Person" objects no longer make sense
  - Add method "monthlyStatement"
- How do we:
  - Force subclasses to have this method
  - Stop having actual Person objects
  - Keep having Person references
  - ~~Retain common Person code~~

Implementation vs. Interface

Abstract classes offer inheritance of both!

- **Implementation**: instance variables and methods which define common behavior
- **Interface**: method signatures which define required behaviors

What if we just want to inherit the Interface?

Abstract classes! Then use an Interface!

- Can make any class abstract with keyword:

```
public abstract class Person {
```

- Class **must** be abstract if any methods are:

```
public abstract void monthlyStatement(){
```

Interfaces only define required methods. Classes can inherit from multiple Interfaces

Abstract Classes and Methods in Java Explained in 7 Minutes
https://www.youtube.com/watch?v=HvPlEJ3LHgE

Abstract Classes and Interfaces (Contd.)

```
// Defined in java.lang.Comparable
package java.lang;
public interface Comparable<E> {
// Compare this object's name to o's name
// Return < 0, 0, > 0 if this object compares
//   less than, equal to, greater than o.
 public abstract int compareTo(E o);
}
```

```
public class Person implements Comparable<Person> {
        private String name;
        // more code here
        @Override
        public int compareTo(Person o) {
                return this.getName().compareTo(o.getName());
}
```

Abstract class or Interface?

- If you just want to define a required method:

  `Interface`

- If you want to define potentially required methods AND common behavior:

  `Abstract class`