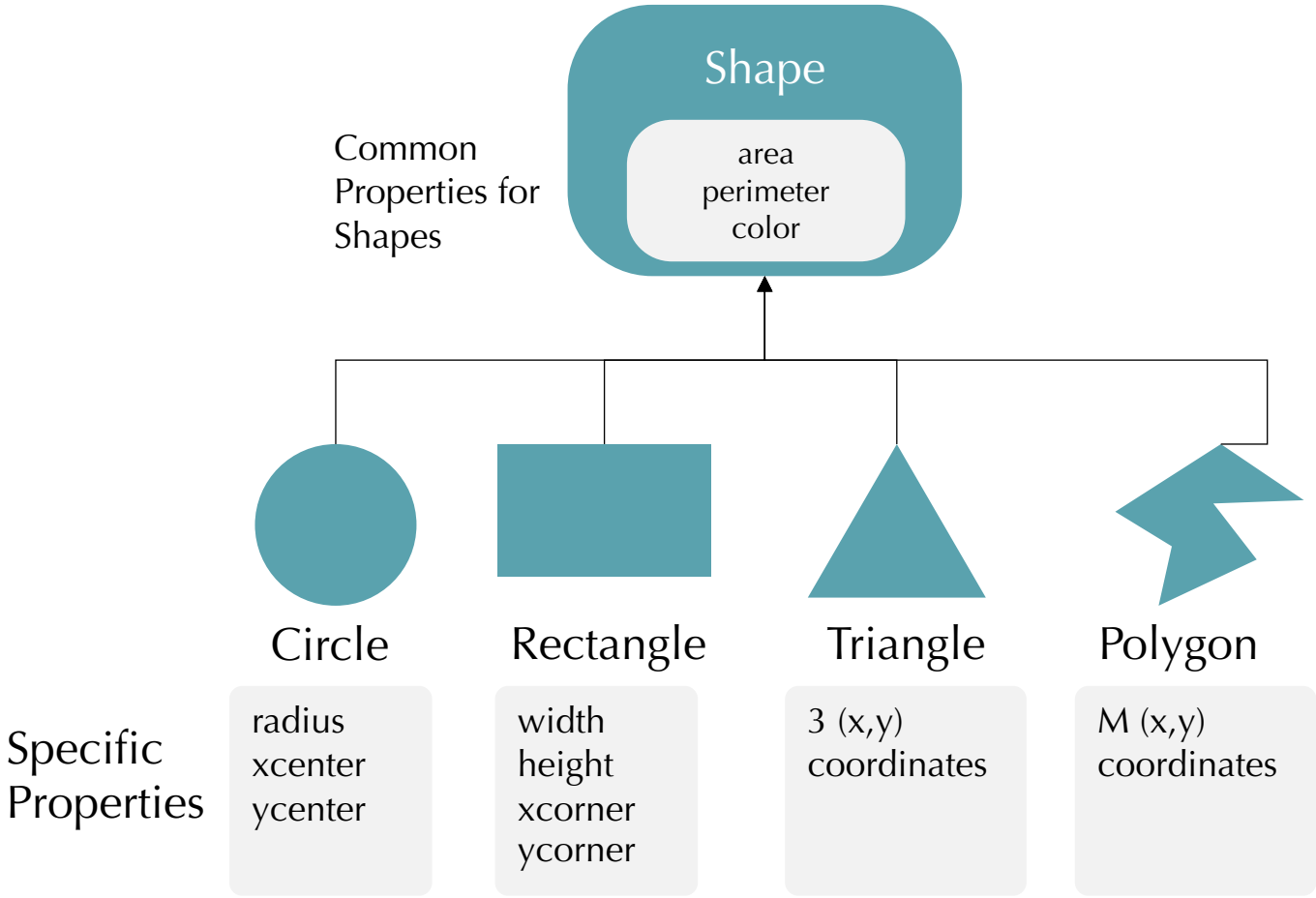
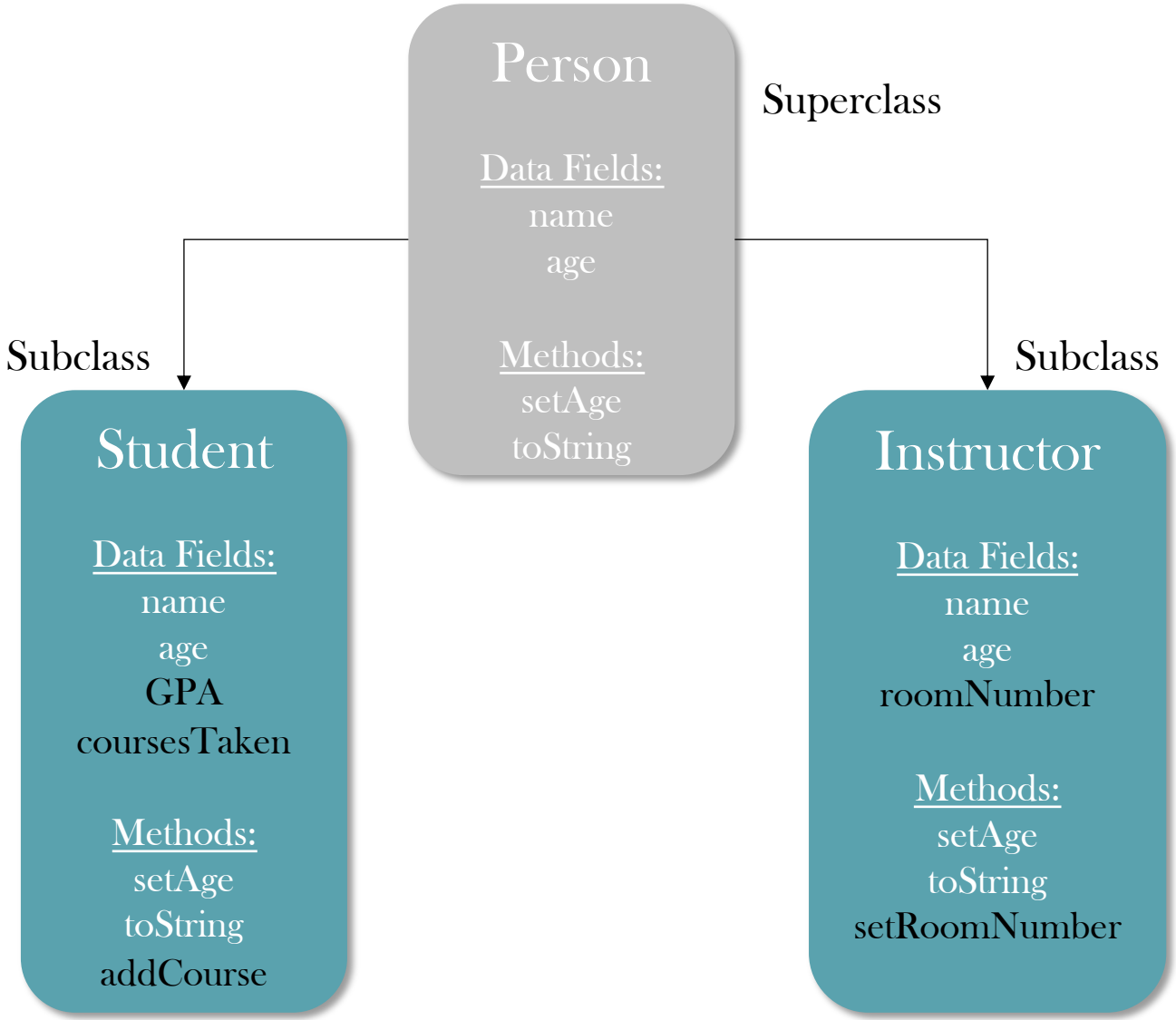
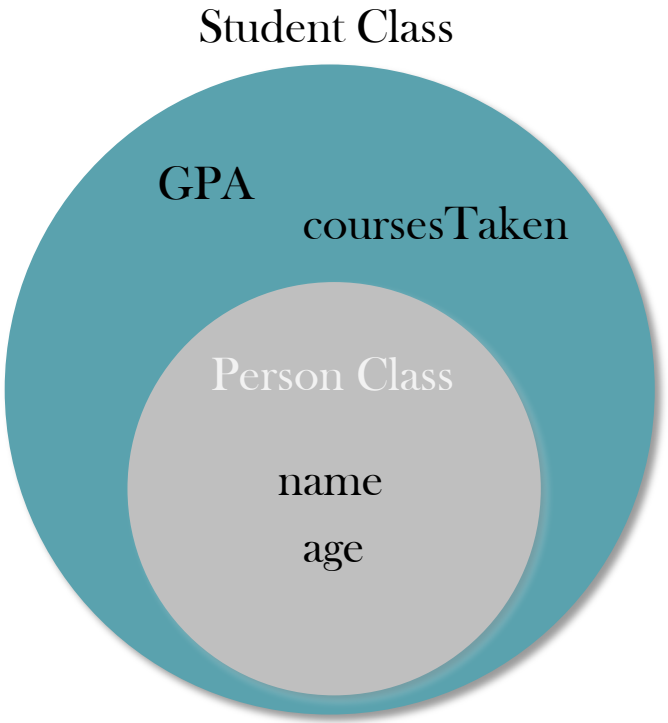
# Inheritance

# Inheritance Hierarchy: Example



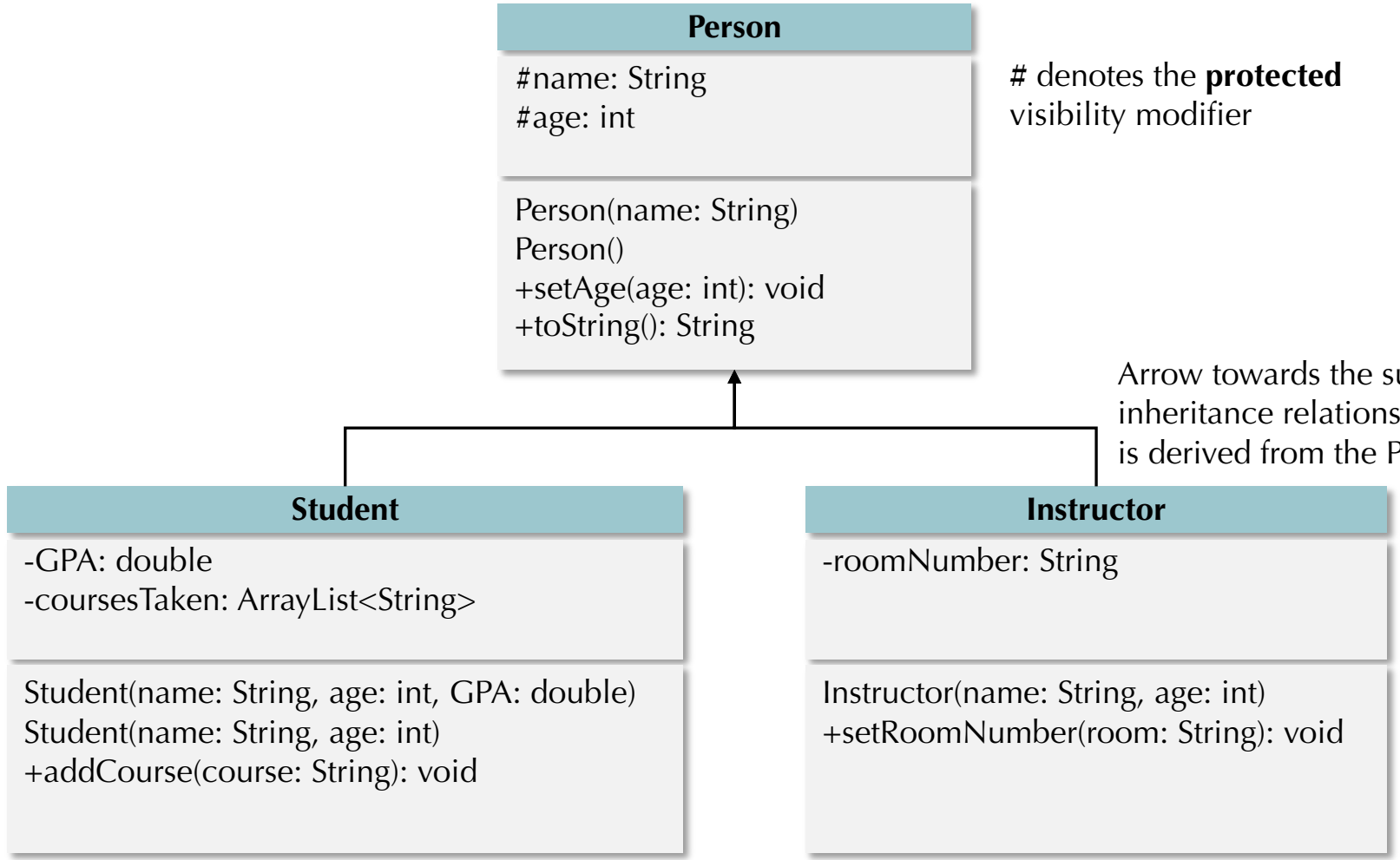
# Inheritance Hierarchy: Example




Student class is an extension of the Person class: It covers Person attributes/methods and adds new attributes/methods, i.e., subclasses extends superclasses

# UML Class Diagrams for Inheritance




# denotes the **protected** visibility modifier

Arrow towards the super class denotes the inheritance relationship, e.g., Instructor class is derived from the Person class

You do not need to write superclass's data fields and methods in the subclass. They are implicitly defined.

# Inheritance Hierarchy: Example

```
// super class
public class Person {

    protected String name;
    protected int age;

    Person(){} // nor-arg constructor

    Person(String name) {
        this.name = name;
        setAge(0);
    }

    protected void setAge(int age) {
        if (age >= 0)
            this.age = age;
        else
            System.out.println("Age should be positive.");
    }

    public String toString() {
      return "Name: " + name + ", Age: " + age;
    }
}
```

# Inheritance Hierarchy: Example

```
// extends keyword implies inheritance: Student class is derived from the Person class.
public class Student extends Person {
    private double GPA;
    private ArrayList<String> coursesTaken;

    Student(String name, int age, double GPA){
        this.name = name;
        this.age = age;
        this.GPA = GPA;
        coursesTaken = new ArrayList<>();
    }

    Student(String name, int age){
        super(name);     // super calls super class constructor
        setAge(age);     // You can use super class methods (if not private)
        GPA = 1.00;      // you can access super class data fields (if not private)
        coursesTaken = new ArrayList<>();
    }

    public void addCourse(String newCourse) { coursesTaken.add(newCourse); }

    public String toString() { return "Student: " + super.toString() + ",GPA: " + GPA + ",Courses: " + coursesTaken; }
}
```

# Inheritance Hierarchy: Example

```
public class Instructor extends Person {

    private String roomNumber;

    Instructor(String name, int age){
        super(name); // call super class constructor
        setAge(age);
    }

    public void setRoomNumber(String room) {
        roomNumber = room;
    }

    public String toString() {
        return "Instructor: " + super.toString() + ", Room: " + roomNumber;
    }

}
```

# Inheritance Hierarchy: Client Code

```
public class App {
    public static void main(String[] args) {

        Student s = new Student("John", 19, 3.10);
        s.addCourse("COMP101");
        s.addCourse("COMP202");
        System.out.println(s);

        Student p = new Student("Alice",21);
        p.addCourse("EE303");
        System.out.println(p);

        Instructor t = new Instructor("Robert",30);
        t.setRoomNumber("A540");
        System.out.println(t);
    }
}
```

# Inheritance

- Object-oriented programming allows you to define new classes from existing classes. This is called inheritance
- Suppose you need to define classes to model circles, rectangles, and triangles. These classes have many common features
- What is the best way to design these classes to avoid redundancy and make the system easy to comprehend and easy to maintain?
- Use inheritance

# Superclass and Subclass

- Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses)
- Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes
- The specialized classes inherit the properties and methods from the general class

# Superclass and Subclass

- A class Circle extended from Shape class is called a subclass. Also referred to as a child class, an extended class, or a derived class
- Shape class is called a superclass. Also referred to as a parent class or a base class
- A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods
- Visibility of data fields and methods
  - Subclass cannot access superclass's private members
  - If you want to have a data fields or method that can be accessed from subclasses, but not accessible from the other external classes, define them as protected

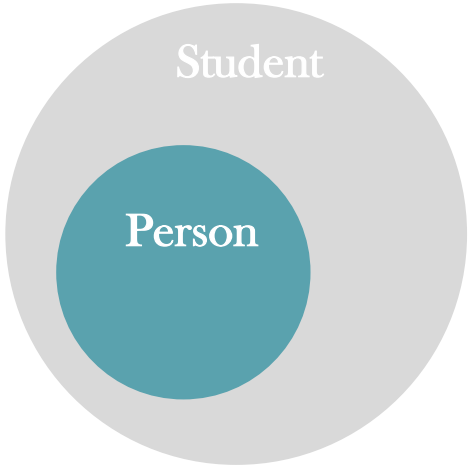
# Syntax for Subclass Declaration

```
public class SubClassName extends SuperClassName {

    // instance variables
    // methods

}
```

# Accessing superclass's data

- You can access super class's public (and protected) data fields/methods from subclasses
- However, you can not access super class's private data fields and methods from subclasses
  - They can be accessed through public getters/setters (accessors/mutators ) if defined in the superclass

- A subclass contains more information and methods than its superclass
  - Superclass is more general. Subclass is more specific



# super Keyword

# Using super keyword

- The keyword super refers to the superclass and can be used to invoke the superclass's methods and constructors
- It can be used in two ways: 1) Call a superclass constructor 2) Call a superclass method

Calling superclass constructor

```
public class Student extends Person {

 public double GPA;

 Student() {
  super();
  GPA = 1.0;
 }

 Student(String name, int age, double GPA) {
  super(name, age);
  GPA = 1.0;
 }
}
```

Calling superclass method

```
public class Student extends Person {

 @Override
 public String toString() {
  return super.toString() + ", GPA: " + GPA;
 }

}
```

# Using super keyword

- You do not need to use super.methodName() syntax to call superclass method
- Use of super.methodName() is only required if the subclass contains a method with the same name, and you need to call superclass method

Calling superclass methods: With or without the super keyword

```
public class Student extends Person {

  public void increaseAge() {
    setAge(age+1); // calling superclass method setAge without the super keyword
  }

  @Override
  public String toString() {
    return super.toString() + ", GPA: " + GPA;
    // here super keyword is needed since both subclass and superclass has toString method
  }

}
```

# Constructor Chaining

# Constructor chaining

- When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor
- This process continues until the last constructor along the inheritance hierarchy is called. This is called constructor chaining

```
public class Student extends Person {

 public double GPA;

 Student(String name, int age, double GPA) {
  super(name, age);
  GPA = 1.0;
 }

}
```

# Constructor chaining

- If a subclass constructor does not call any constructor, the compiler automatically puts super() as the first statement

```
public class Student extends Person {

    Student(String name, int age, double GPA){
        this.name = name;
        this.age = age;
    }


}
```



```
public class Student extends Person {

    Student(String name, int age, double GPA){
        super();
        this.name = name;
        this.age = age;
    }

}
```

# Constructor Chaining

- Student constructor automatically calls super() first

```
public class Person {

 public String name;
 public int age;

 Person() {
  System.out.println("No-arg Constructor: Person");
 }
}
```

```
public class Student extends Person {

 public double GPA;

 Student() {
  System.out.println("Constructor: Student");
  GPA = 1.0;
 }
}
```

Client code

```
public static void main(String[] args) {
  Student student = new Student();
}
```

Output

```
No-arg Constructor: Person
Constructor: Student
```

# Always Write No-arg Constructors

- Subclass constructors automatically invokes superclass's nor-arg constructor, if not written explicitly
- In that case, if the superclass does not have a no-arg constructor, compile error occurs
- To prevent this error, always write no-arg constructors to classes if the class is to be extended

```
public class Person {

 public String name;
 public int age;

 public Person(String name, int age) {
  this.name = name;
  this.age = age;
 }
}
```

```
public class Student extends Person {

 public double GPA;

 Student() {
   // gives compile error: super() is not written in Person
   GPA = 1.0;
 }

}
```

# Public, Private, and Protected

Visibility modifiers

# Protected Visibility Modifier

- Visibility (or accessibility) modifiers such as public, private and protected specify how classes and class members are accessed
  - Public members can be accessed from any other classes
  - Private members can be accessed only from inside of the class
- A protected member of a class can be accessed from a subclass
  - Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow non-subclasses to access these data fields and methods

# Visibility Modifiers

- Use the private modifier to hide the members of the class completely so that they cannot be accessed directly from outside the class
- Use the protected modifier to enable the members of the class to be accessed by the subclasses in any package or classes in the same package
- Use the public to enable the members of the class to be accessed by any class

# Overriding

# Method Overriding

- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass: This is referred to as method overriding
- To override a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass
- An instance method can be overridden only if it is public or protected: A private method cannot be overridden, because it is not accessible outside its own class

```
public class Person {

 public String toString() {
  return name + ", Age: " + age;
 }

}
```

```
public class Student extends Person {

 @Override
 public String toString() {
  return super.toString() + ", GPA: " + GPA;
 }

}
```

# Overriding vs Overloading

- Overloading: Define multiple methods with the same name but different signatures
- Overriding: Provide a new implementation for a method in the subclass

### Overriding

```
public class Person {
 public void increaseAge() {
  age = age + 1;
 }
}
```

```
public class Student extends Person {
 @Override
 public void increaseAge() {
  age = age + 2;
 }
}
```

### Overloading

```
public class Person {
 public void increaseAge() {
  age = age + 1;
 }
}
```

```
public class Student extends Person {
 public void increaseAge(int value) {
  age = age + value;
 }
}
```

# final keyword

## Prevent Extending and Overriding

# Prevent Extending Classes

- You may want to prevent classes from being extended: Use the final modifier to indicate that a class is final and cannot be extended

- In the example given below, Person class is final and Student class cannot be created using inheritance

```
public final class Person {

 public String name;
 public int age;

 Person() {
  System.out.println("No-arg Constructor: Person");
 }
}
```

# Prevent Overriding Methods

- You also can define a method to be final: Final methods cannot be overridden

Overriding is not possible for final methods

```
public class Person {
 public final void increaseAge() {
  age = age + 1;
 }
}
```

```
public class Student extends Person {
 // Compile error
 public void increaseAge() {
  age = age + 2;
 }
}
```

# Object Class

# Object class

- Every class in Java is descended from the Object class
  - If no inheritance is specified, superclass of a class is Object class by default

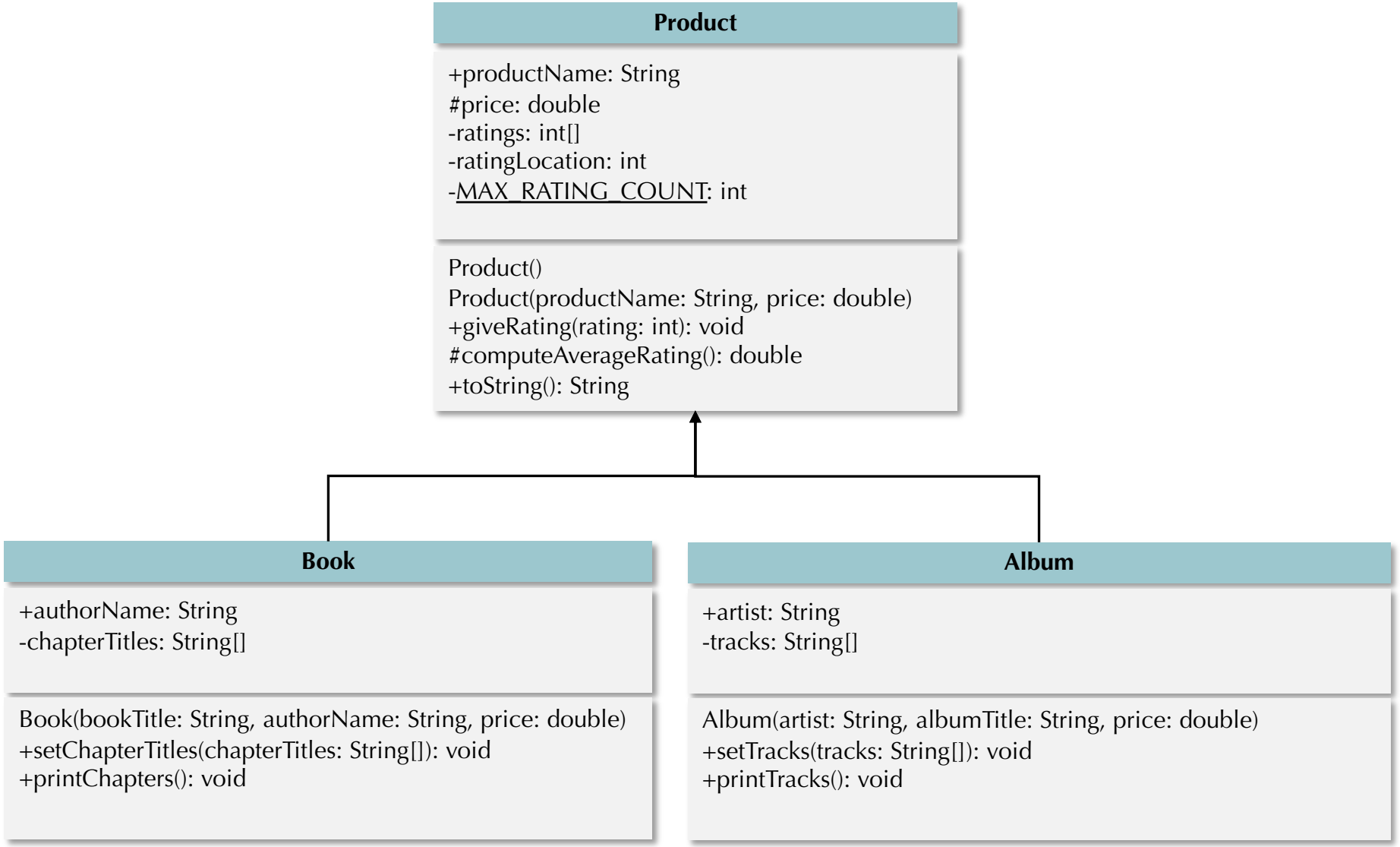
- Object class has toString() method which returns a string that describes the object
  - By default, it returns a string consisting of a class name and the object's memory address
  - Usually, you should override the toString method so that it returns a string representation of the object

```
public class Person {

 public String toString() {
  return name + ", Age: " + age;
 }

}
```

# Inheritance Example

Products: Books and Albums

# UML Diagrams for Inheritance Hierarchy



# Product Superclass

```
public class Product {

  // data fields
  public String productName;
  protected double price;
  private int[] ratings;

  // Maximum rating limit
  private final static int MAX_RATING_COUNT = 10;
  private int ratingLocation = 0; // location to a rating

  Product(){} // no-arg constructor

  public Product(String productName, double price) {
    super();
    this.productName = productName;
    this.price = price;
    ratings = new int[MAX_RATING_COUNT];
  }

  // adds a rating to product
  public void giveRating(int rating) {
    if (ratingLocation < MAX_RATING_COUNT)
      ratings[ratingLocation++] = rating;
    else
      System.out.println("Maximum rating limit reached!");
  }
```

```
  // computes average rating
  protected double computeAverageRating() {
    double sum = 0.0;
    for (int i = 0; i < ratingLocation; i++)
      sum += ratings[i];
    if (ratingLocation == 0)
      return 0.0; // if there is no rating, return zero
    else
      return sum / ratingLocation;
  }

  @Override
  public String toString() {
    return "\nName=" + productName + ", Price="
              + price + "\nRatings=" + Arrays.toString(ratings);
  }
```

# Book Subclass

```
public class Book extends Product {
  // data fields
  public String authorName;
  private String[] chapterTitles;

  public Book(String bookTitle, String authorName, double price) {
    super(bookTitle, price);         // call superclass constructor
    this.authorName = authorName;    // set author name
  }

  // set chapter titles
  public void setChapterTitles(String[] chapterTitles) {
    this.chapterTitles = chapterTitles;
  }

  public String toString() { // calls superclass methods: toString and computeAverageRating
    return super.toString() + "\nAuthor: " + authorName + "\nAverage Rating: " + computeAverageRating();
  }

  // print chapter titles
  public void printChapters() {
    for (String chapter : chapterTitles)
      System.out.println(chapter);
  }
}
```

# Album Subclass

```
public class Album extends Product {

  // data fields
  public String artist;
  public String[] tracks;

  public Album(String artist, String albumTitle, double price) {
    super(albumTitle, price);
    this.artist = artist;
  }

  public void setTracks(String[] tracks) {
    this.tracks = tracks;
  }

  public String toString() {
    return super.toString() + "\nArtist: " + artist + "\nAverage Rating: " + computeAverageRating();
  }

  public void printTracks() {
    for (String track: tracks)
      System.out.println(track);
  }

}
```

# Client Code for Testing

```
public static void main(String[] args) {

   Product p = new Product("Nike", 100);
   p.giveRating(3); p.giveRating(2); p.giveRating(1); p.giveRating(4);

   Book b = new Book("Book Title 1", "Author Name 1", 25);
   b.giveRating(2); b.giveRating(2); b.giveRating(2);
   String[] chapterTitles = {"Chapter 1", "Chapter 2", "Chapter 3"};
   b.setChapterTitles(chapterTitles);

   Album a = new Album("Artist Name 1", "Album Name 1", 60);
   a.giveRating(4); a.giveRating(5); a.giveRating(5);
   String[] tracks = {"Track 1", "Track 2"};
   a.setTracks( tracks );

   System.out.println(p);

   System.out.println(b);
   b.printChapters();

   System.out.println(a);
   a.printTracks();
}
```

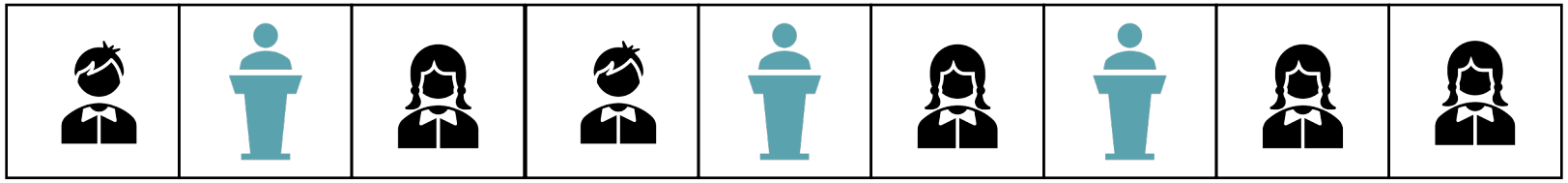
Output

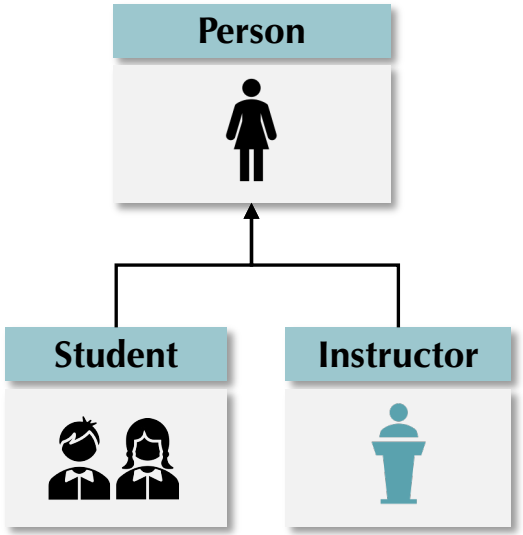
```
Name=Nike, Price=100.0
Ratings=[3, 2, 1, 4, 0, 0, 0, 0, 0, 0]

Name=Book Title 1, Price=25.0
Ratings=[2, 2, 2, 0, 0, 0, 0, 0, 0, 0]
Author: Author Name 1
Average Rating: 2.0
Chapter 1
Chapter 2
Chapter 3

Name=Album Name 1, Price=60.0
Ratings=[4, 5, 5, 0, 0, 0, 0, 0, 0, 0]
Artist: Artist Name 1
Average Rating: 4.666666666666667
Track 1
Track 2
```

# Polymorphism

# How to Store Students and Instructors?

- Assume that there are many students and three instructors in a course: How can we store all members in a single array?
- Student and instructor classes are derived from the Person superclass






- Answer: Define a Person array
- Person array can store both students and instructors since they are both inherited from the Person class

# Storing Subclass Objects in a Single Array

- Person array can store both students and instructors

```
public static void main(String[] args) {

 Person[] courseArray = new Person[1000];

 courseArray[0] = new Student("John",19);
 courseArray[1] = new Student("Mary",20);
 courseArray[2] = new Instructor("Robert", 30, 505);
 courseArray[3] = new Instructor("Sarah", 32, 508);

 for (Person courseMember : courseArray)
  if (courseMember != null)
   System.out.println(courseMember.name);

}
```

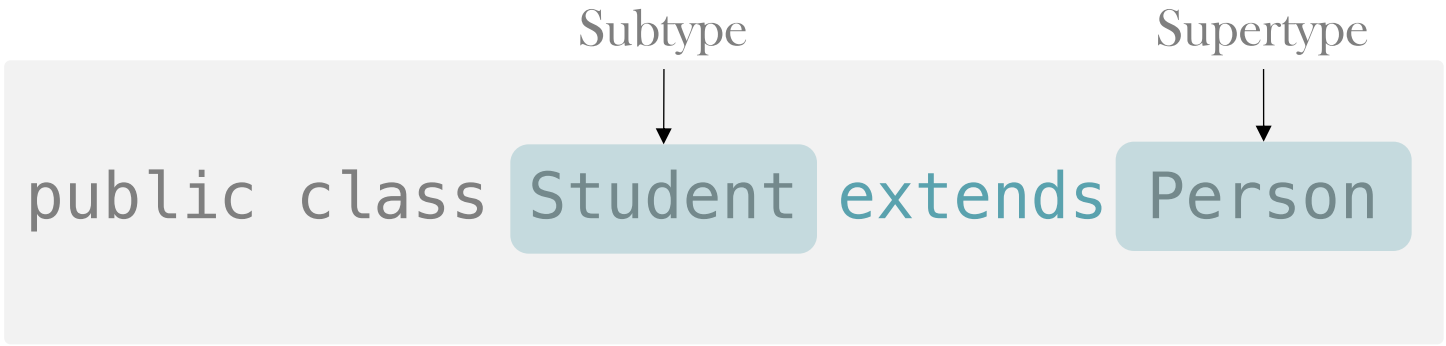
# Superclass Variable Can Store Subclass Object

- In general, a superclass variable can store subclass objects
- For example, a Person variable can store a student or an instructor object

```
public static void main(String[] args) {

  Person person1 = new Student("John",21);
  Person person2 = new Instructor("Sarah", 32, 509);

  System.out.println(person1.age);
  System.out.println(person2.name);

}
```

# Supertype vs Subtype

- Polymorphism means that a variable of a supertype can refer to a subtype object
  - Superclass defines supertype and Subclass defines subtype

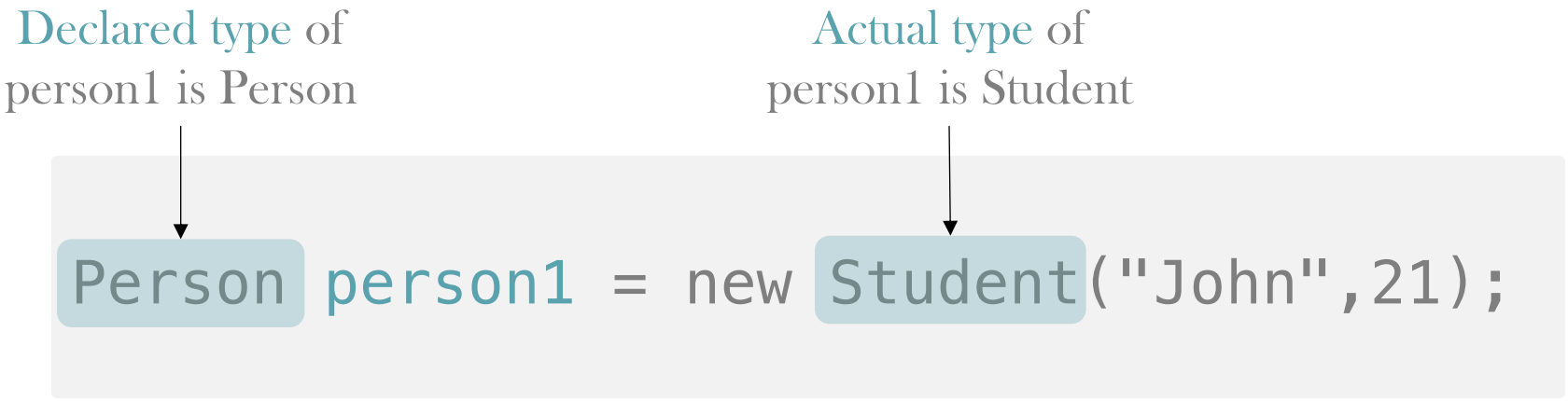



- Examples

```
public static void main(String[] args) {
  Person person1 = new Student("John",21); // supertype variable person1 refers to subtype Student
  Person person2 = new Instructor("Sarah", 32, 509);
}
```

# Declared Type vs Actual Type

- With polymorphism, declared type can be different from the actual type




- Type that is used in the declaration is called the declared type
  - Person is the declared type in Person p;
- The actual type is the actual class for the object referenced by the variable
  - Instructor is the actual type in p = new Instructor()

# Dynamic Binding

- Which method is invoked by the variable is determined by the actual type
- This is known as dynamic binding
- Java decides which method is invoked at runtime

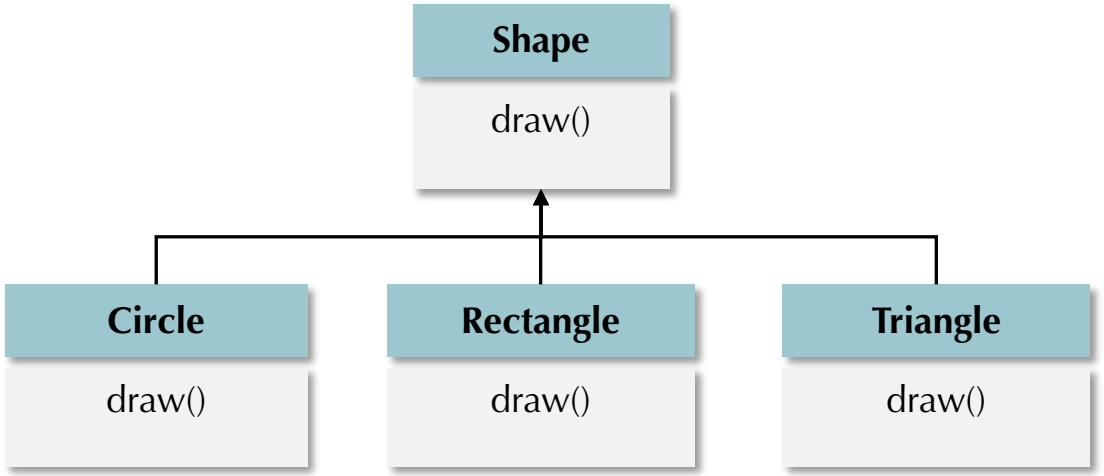
```
Person person1 = new Student("John",21);
Person person2 = new Instructor("Sarah", 32, 509);

System.out.println(person1); // Actual type is Student. Student's toString method is invoked.
System.out.println(person2); // Actual type is Instructor. Instructor's toString method is invoked.
```

Output

```
Student: John, Age: 21, GPA: 0.0
Instructor: Sarah, Age: 32, Room Number: 509, Courses Given: null
```

# Dynamic Binding Example

- Suppose that Circle, Rectangle and Triangle classes are inherited from the Shape class and each subclass overrides draw() method in the Shape class
  - draw() method in the Circle class draws a circle, etc.



# Dynamic Binding Example

- Following code draws a circle, a rectangle and a triangle in the for-each loop

```
Shape[] shapes = new Shape[100]; // store different shapes in a single array
shapes[0] = new Circle(x,y,r);
shapes[1] = new Rectangle(x,y,w,h);
shapes[2] = new Triangle(x1,y1,x2,y2,x3,y3);

for (Shape currentShape : shapes)
  currentShape.draw(); // invokes different draw methods based on actual type
```

- Superclass should have the draw method and subclasses should override it
  - Otherwise, currentShape.draw() gives a compile error
  - Java requires that the declared type has the method/instance variable

# Dynamic Binding

- If the declared type does not have the methods or instance variables, compile error occurs

- Example: Person superclass has name and age instance variables but does not have GPA
  - The statement person1.GPA gives compile error since Person superclass does not have GPA data field

```
public static void main(String[] args) {

  Person person1 = new Student("John",21);
  Person person2 = new Instructor("Sarah", 32, 509);

  System.out.println(person1.age);   // correct
  System.out.println(person2.name);  // correct
  System.out.println(person1.GPA);  // compile error: declared type does not know the GPA variable

}
```

# Use of Polymorphism with Method Parameters

- A method with a supertype input argument can be invoked with a subtype object
- Example: Write a method which prints age of a student or an instructor

```
public class App {
 public static void main(String[] args) {

  Student s = new Student("Alice",21);
  printInformation(s);
  Person p = new Instructor("Bob",29,507);
  printInformation(p);
 }

  // method can accept Person, Student or Instructor
 public static void printInformation(Person input) {
  System.out.println("Age is: " + input.age);
 }
}
```

# Polymorphism

- You cannot assign a supertype object to a subtype variable
- Declared type should always be more general
  - Person is general
  - Student is more specific

```
public class App {
 public static void main(String[] args) {

   Person p = new Instructor("Bob",29,507); // correct
   Student s = new Person(); // wrong: not all Persons are Student

 }
}
```

# Type Casting Objects

- One object can be typecast into another object. This is called casting an object

- Example: You cannot assign Person to a Student

```
Person person = new Student("Alice",22);
Student student =  person; // you cannot assign a Person object to Student type (compile error)
```

- If you know that actual class of variable person is Student, you can typecast it to Student:

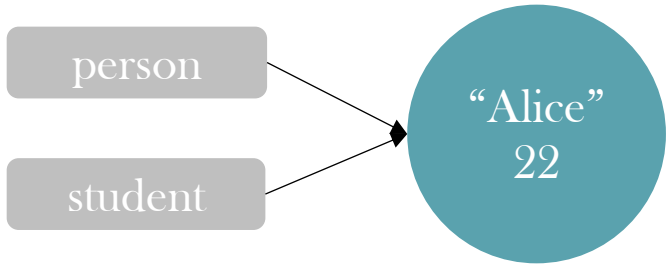
```
Person person = new Student("Alice",22);
Student student =  (Student) person; // first cast person to Student since actual type is Student
```

# Type Casting Objects

- Casting objects does not create a new object

```
Person person = new Student("Alice",22);
Student student =  (Student) person;
```

- Here, both person and student variables point to the same object



# instanceof Operator

- Write a method which accepts a Person argument and displays
  - GPA of a student, if the input is a Student
  - Room number of an instructor, if the input is an Instructor

```
public static void main(String[] args) {

 Instructor a = new Instructor("Bob", 29, 509);
 printDetails(a);

 Student b = new Student("Alice", 22);
 b.GPA = 3.1;
 printDetails(b);

}
```

# instanceof + Type Casting

- Write a method which accepts a Person argument and displays
  - GPA of a student, if the input is a Student
  - Room number of an instructor, if the input is an Instructor

```
public static void printDetails(Person input) {

 if (input instanceof Student) {
   // if input is student, cast it to Student and display GPA
  Student s = (Student) input;
  System.out.println("Student GPA: " + s.GPA);
 }
 else if (input instanceof Instructor) {
   // if input is Instructor, cast it to Instructor and display room number
  Instructor m = (Instructor) input;
  System.out.println("Instructor Room Number: " + m.roomNumber);
 }

}
```

# instanceof + Type Casting: Shorter Syntax

- You can use ((Student) input).GPA to perform type casting in a single statement
  - Parentheses are required
- However, it is more readable to first create a Student object s and access GPA using s
  - Student s = (Student) inputObject;
    s.GPA

```
public static void printDetails(Person input) {

 if (input instanceof Student) {
   // compact notation for type casting
  System.out.println("Student GPA: " + ((Student) input).GPA);
 }
 else if (input instanceof Instructor) {

  System.out.println("Instructor Room Number: " + ((Instructor) input).roomNumber);
 }
}
```

# instanceof Operator

- instanceof operator is used to determine the type of an object
- Example: If the object's type is Student, the statement inputObject instanceof Student returns true

```
if (inputObject instanceof Student) {
  // inputObject's type is Student
}
```

- If the object is Student, following statement also returns true since Student is derived from Person

```
if (inputObject instanceof Person) {
  // inputObject's type is also a Person
}
```