CS18000: Problem Solving and Object-Oriented Programming

Interfaces and Inheritance

Video 1
Interface Concepts

Interfaces

Interfaces
Encapsulation

Interface Concepts

- Interface:
  - A point where two systems interact
  - Typically asymmetric: one system “defines” the interface, the other system “uses” it
- Examples:
  - Graphical User Interface (GUI): user -> computer
  - Application Programming Interface (API): application program -> library of related methods

Java Class

- A Java class provides one form of interface
- Public members (methods, mainly) define the interface to “clients” (users) of that class
- Class interface consists of
  - Public method signatures (what the method expects)
  - Method return types (what the method returns)

- The Java language abstracts this idea one step further…

Java Interface

- Defines a “contract” between
  - A class that defines the interface, and
  - A class that implements (uses) the interface
- Any class that *implements* the interface must provide implementations for all the method bodies given in the interface *definition* (except default methods)

Interface Syntax

- A class-like declaration
  - `interface Doable { … }`
  - Exists in own file
  - Includes method declarations
- But…
  - No method *bodies* (except default methods)
  - No fields (other than constants)

- An interface is like a class in which you forgot to declare the fields and left out the method bodies

Default Methods

- A default method is an instance method defined in an interface whose method header begins with the default keyword
- It also provides a code body
- Every class that implements the interface inherits the interface's default methods but can override them

```
public interface Addressable
{
   String getStreet();
   String getCity();

   default String getFullAddress()
   {
      return getStreet() + ", " + getCity();
   }
}
```

Video 2
Implementing Interfaces

Implementing an Interface

- Classes may declare that they "`implement`" an interface
- Given interface `Doable` a class `Henway` can implement it…

```
public class Henway implements Doable { … }
```

- All the methods declared in `Doable` must appear in `Henway` (and other methods may appear, too)

Example: Doable

```
interface Doable {
    int compute(int x);
    void doit(int y);
}

class Henway implements Doable {
    public int compute(int x) {
       return x + 1;
    }
    public void doit(int y) {
        System.out.println(y);
    }
}
```

Fields in Interfaces

- Interfaces may include fields
- Fields are implicitly declared
  - public,
  - final, and
  - static
- That is, fields in interfaces are constants, and so must be declared with an initializer (=)
- Allows easy use of shared constants
- Methods are implicitly declared public

Example: Constants

```
interface Constants {
    double X = 1234.56;
    int Y = -1;
    String Z = "hello there";
}

public class Booyah implements Constants {
    public static void main(String[] args) {
        System.out.println(X);
        System.out.println(Y);
        System.out.println(Z);
    }
}
```

Implementing Multiple Interfaces

- A class can implement multiple interfaces
- The methods implemented are the union of the methods specified in the interfaces
- Examples:

```
class SoapOpera implements Cryable { … }
class SitCom implements Laughable { … }
class Movie implements Laughable, Cryable { … }
```

Example: Rideable

- Rideable defines an interface to something you ride:

```
void mount();
void dismount();
void move(boolean forward);
void turn(int direction);
void setSpeed(double mph);
```

- Implementations:

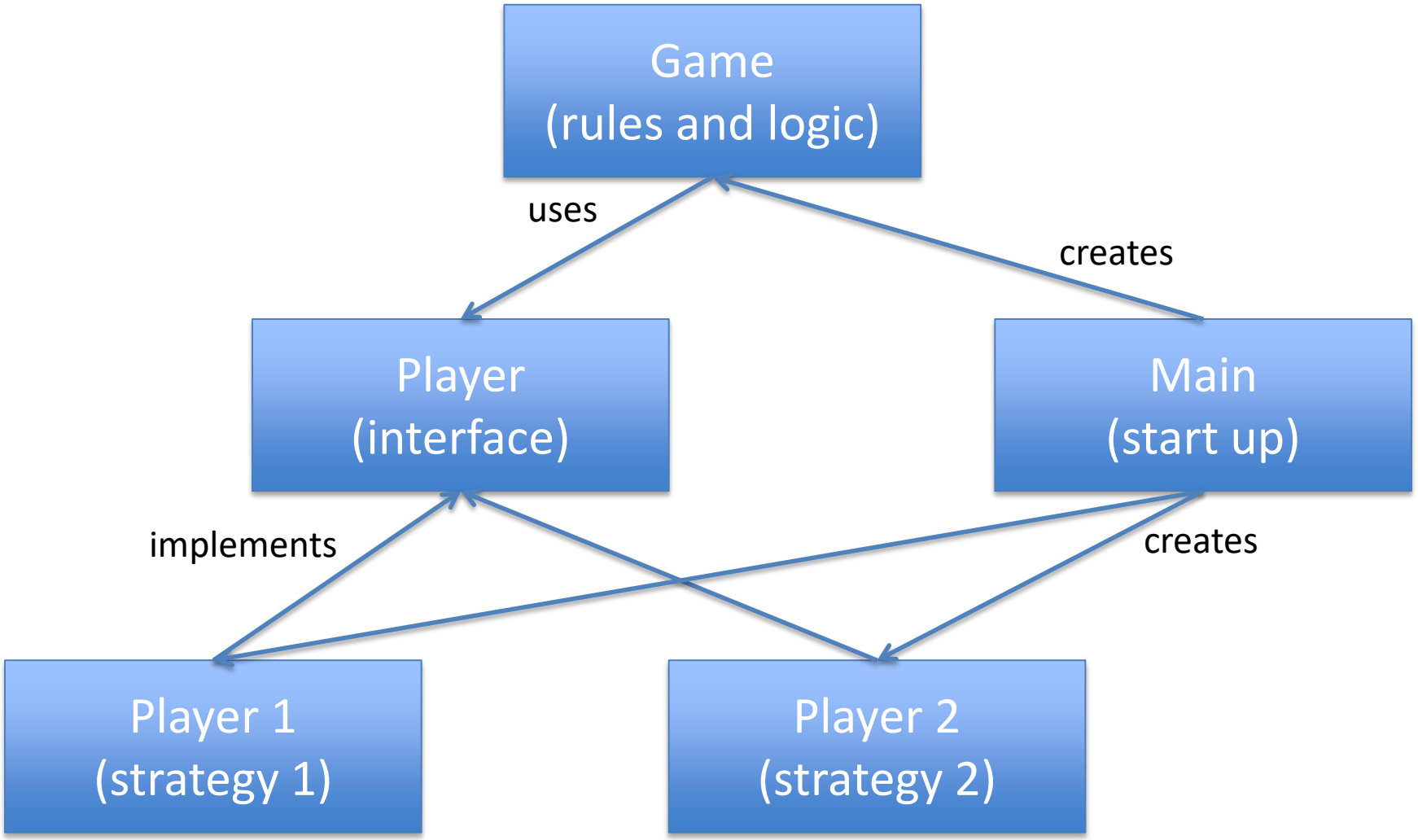
```
class Motorcycle implements Rideable { … }
class Horse implements Rideable, Trainable {
… }
class Bicycle implements Rideable { … }
```

Video 3
Building a Game

Example: Building a Game

- Problem: Implement a turn-based game in which players can pick up valuable objects
- Multiple players, each with own strategy
- Rules enforced by game controller
- Use of Java interface:
  - Each player class implements Player interface
  - Game controller expects parameters of type Player
- Main program:
  - Creates player objects from classes
  - Creates game controller with player objects
  - Starts game controller
  - Prints results

Game Program Class Diagram



Player Interface

```
interface Player {
    void makeMove();
    void getItems();
}
```

Dragon Class

```
public class Dragon implements Player {
    public void makeMove() {…};
    public void getItems() {…};
    …other methods…
    }
}
```

Butterfly Class

```
public class Butterfly implements Player
{
    public void makeMove() {…};
    public void getItems() {…};
    …other methods…
    }
}
```

Main Class

```
public class Main {
    public static void main(String[] args) {
        Dragon bob = new Dragon();
        Butterfly ann = new Butterfly();
        Game game = new Game(bob, ann);
        game.play();
        System.out.println("game over");
    }
}
```

Game Class

```
public class Game {
    private Player p1;
    private Player p2;

    Game(Player p1, Player p2) {
        this.p1 = p1;
        this.p2 = p2;
    }

    void play() {
        p1.makeMove(); …
        p2.makeMove(); …
        p1.getItems(); …
        p2.getItems(); …
    }
}
```

Video 4
Fibonacci Generator

Example: Fibonacci Generator

- Write a class to generate the Fibonacci sequence
- Each value is sum of two previous values
- 1, 1, 2, 3, 5, 8, 13, 21, …
- Constructor takes an `int` n that specifies the (finite) number of values to generate
- Fibonacci object provides `hasNext()` and `next()` methods to generate the n values

Two Standard Java Interfaces (simplified)

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}

interface Iterable {
    Iterator iterator();
}
```

Java for-each Loop

- Uses `Iterable` interface

```
for (Tree t : list) { … }
```

- The `list` must implement the `Iterable` interface
- That is, it must have a method that returns an `Iterator` over elements of the collection

Fibonacci (1)

```
import java.util.Iterator;
Import java.lang.Iterable;

public class Fibonacci implements Iterator, Iterable {
    private int n; // how many Fibonacci numbers
    private int i; // how many so far
    private int f1, f2; // last two Fibonacci numbers generated

    public Fibonacci(int n) {
        this.n = n;
        i = 0;
        f1 = f2 = 1;
    }

    // method required by Iterable interface...
    public Iterator iterator() {
        return this;
    }
```

Fibonacci variables

```
private int n; // how many Fibonacci numbers
private int i; // how many so far
private int f1, f2; // last two Fibonacci numbers generated


n =
i =
f1 =
f2 =
t =
```

Fibonacci (2)

```
// method required by Iterator interface...
public boolean hasNext() {
    return i < n;
}

// method required by Iterator interface...
public Integer next() {
    if (i == 0 || i == 1) {
        i++;
        return 1;
    }
    int t = f1 + f2;
    f1 = f2;
    f2 = t;
    i++;
    return t;
}
```

Fibonacci (3)

```
    // method required by Iterator interface...
    public void remove() {
    }

    public static void main(String[] args) {
        Iterator i1 = new Fibonacci(25);
        while (i1.hasNext())
            System.out.printf("%d ", i1.next());
        System.out.printf("\n");

        Iterable i2 = new Fibonacci(30);
        for (Object i : i2)
            System.out.printf("%d ", (Integer) i);
        System.out.printf("\n");
    }
}
```

Video 1
Inheritance

Inheritance

Problem

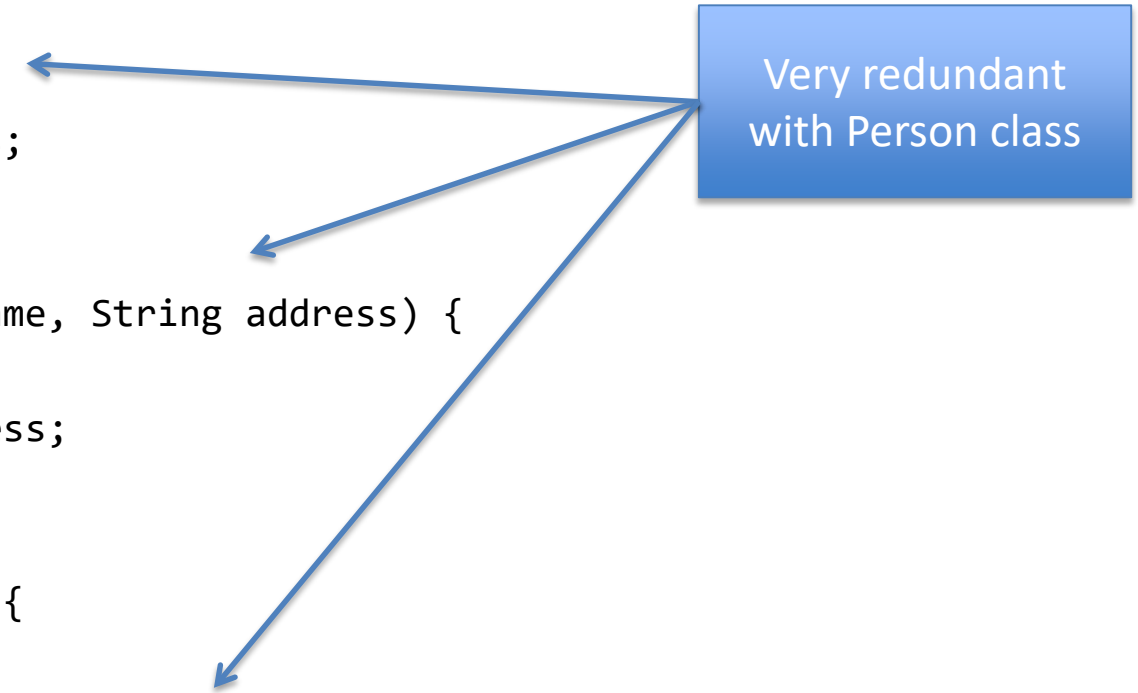
- Sometimes classes have related or overlapping functionality
- Consider a program for keeping track of personnel at the university
- Need a Person class to keep information
- But also might want special classes for
  - Student: to include grades or classes taken
  - Professor: to include salary and rank

Person Class

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

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

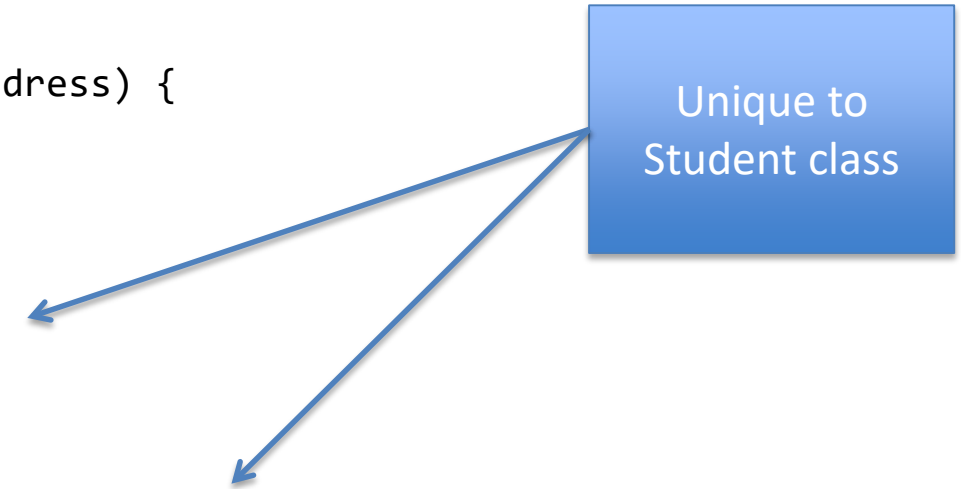
Student Class (1)

```
public class Student {
    private String name;
    private String address;
    private String[] classes;
    private String[] grades;

    public Student(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }
// continued…
```



Student Class (2)

```
    // continued…

    public void setAddress(String address) {
        this.address = address;
    }

    public String[] getClasses() {
        return classes;
    }

    public void setClasses(String[] classes) {
        this.classes = classes;
    }

    // and more…
}
```



Inheritance

- Rather than duplicating members (fields and methods) among these classes, Java allows classes to share member definitions in a hierarchical fashion
- One class can “extend” another “inheriting” fields and methods from it
- Terminology: the “subclass” *inherits* from the “superclass”

Example

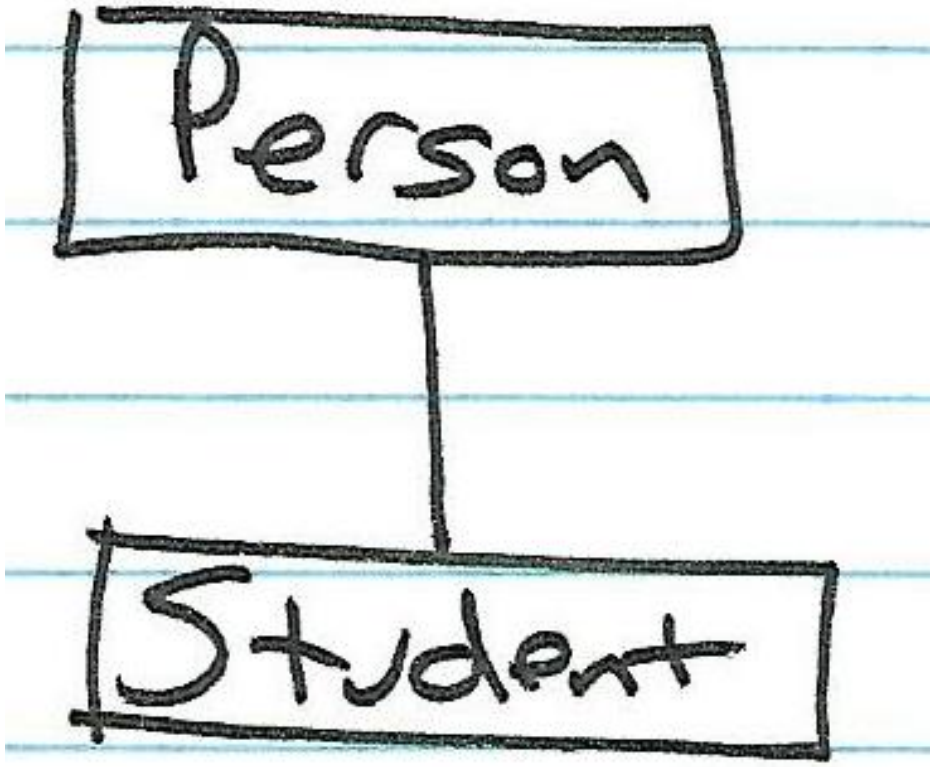
- Class Person has fields name, address, as well as accessors and mutators
- Class Student “extends” Person
  - Inherits the fields and methods from Person
  - Adds classes and grades (and more accessors and mutators)
- Class Professor “extends” Person
  - Inherits the fields and methods from Person
  - Adds rank and salary (and more accessors and mutators)
- Common fields and methods go in Person, and are inherited by its subclasses
- Class-specific fields and methods go in their respective class

Student Subclass

Subclass only contains the differences

```
public class Student extends Person {
    private String[] classes;
    private String[] grades;

    public Student(String name, String address, String[] classes,
                                                    String[] grades) {
        super(name, address);
        this.classes = classes;
        this.grades = grades;
    }

    public String[] getClasses() {
        return classes;
    }

    public void setClasses(String[] classes) {
        this.classes = classes;
    }
}
```

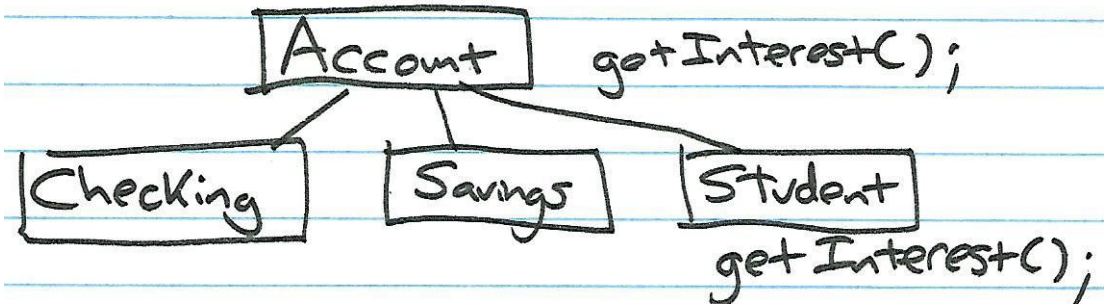
Student Subclass



Classes and Subclasses

```
Student s = new Student (…);
String[] classes = s.getClasses();
String name = s.getName();
double gpa = s.getGPA();

…

Student t = s;
Student t = s.clone();
```

Classes and Subclasses



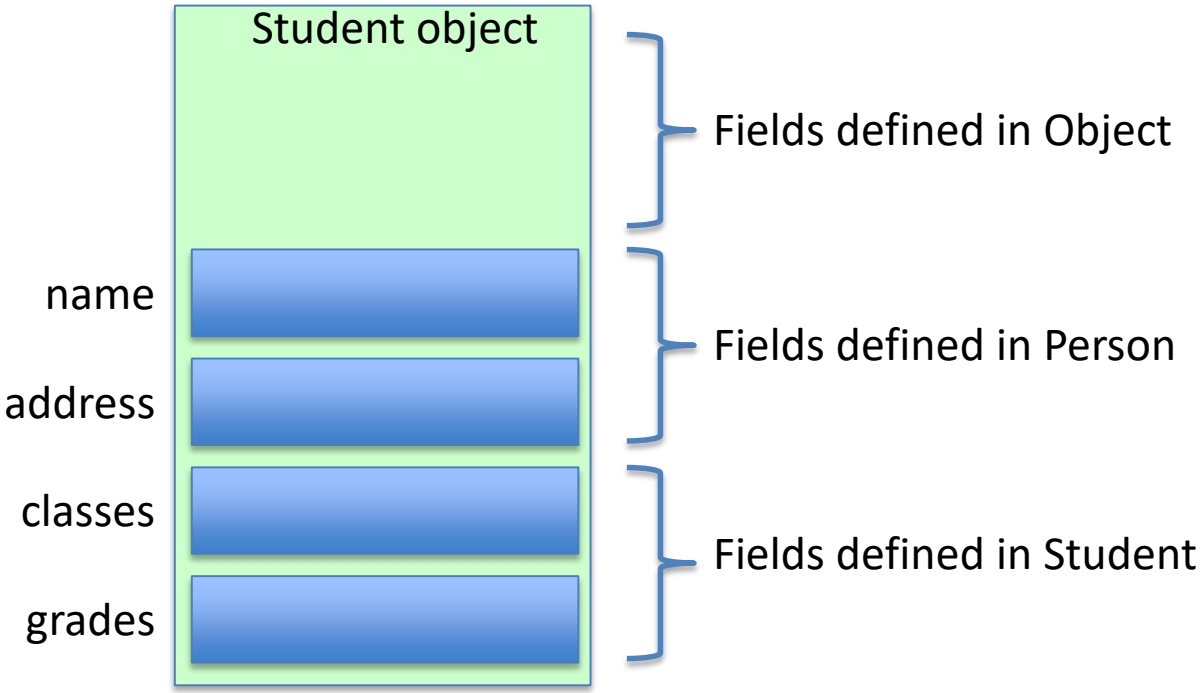

```
Person fred = new Student (…);
String name = fred.getName();

Account ch = new Checking (…);
```

Object Class

- One designated class in Java—Object—is the root of all classes
- Any class that doesn't extend another class implicitly extends the Object class
- A class can only extend one other class (but can implement multiple interfaces)
- Java is a "single inheritance" system
- C++ is a "multiple inheritance" system

Subclass Object

- Contains its fields as well as all the fields defined in its superclasses



Object Class Methods

- The Object class has a small number of public methods. Samples…
  - clone() – makes a copy of the object
  - equals(Object e) – compares for equality
  - toString() – returns a String representation
- The toString() method is very handy:
  - It is called by printf and similar methods when a String is needed (e.g., for printing)
  - You can override it in your classes to get something more descriptive

Video 2
Constructor Chaining

Constructor Chaining

- When constructing an object of a class, it is important that all the constructors up the inheritance chain have an opportunity to initialize the object under construction
- Called *constructor chaining*
- Java enforces constructor chaining by inserting implicit calls to superclass constructors
- You can override this behavior by inserting your own calls

Constructor Rules

- Every class must have at least one constructor

- The first line of every constructor must be a call to another constructor.

Default Constructors

- If you don't provide any constructors in a class, Java provides one for you:

```
public ClassName() {
    super();
}
```

- The statement "super();" calls the 0-argument constructor in the superclass

Default Chaining

If you *do* provide a constructor…

- by default Java inserts the statement

```
super();
```

- at the beginning to enforce chaining

Explicit Chaining

- You can explicitly call a superclass constructor yourself
- Useful for passing arguments “up the line” to initialize the object using superclass constructors
- See the Student example earlier
  - Calls super(name, address)
  - Invokes constructor in Person to initialize these fields

Explicit Chaining

- The first step in each constructor is to either
  - Call another constructor in the current class, or
  - Call a superclass constructor
- To call another constructor, use this(…)
- To call a superclass constructor, use super(…)
- You can do one or the other but not both
- In either case, the argument types are matched with the class constructors to find a match
- If no explicit this(…) or super(…) is provided in a constructor, Java automatically calls super() (the superclass constructor with no arguments)

Constructor Complications

- If the base class does not have a parameterless constructor, the derived class constructor must make an explicit call, with super(…), to an available constructor in the base class

super() and this()

- Recall that this(…) can be used to call another constructor in the current class
- If you call this(…), Java does not call super()
- OK, since, the constructor you call must either call this(…) or super(…), so super(…) will eventually be called
- If specified explicitly, calls to super(…) or this(…) must be the *first* statement in a constructor—ensures proper initialization by superclass constructors before subclass constructors continue

Wheel Example (1)

```
public class Wheel {
    private double radius;

    public Wheel(double radius) {
        this.radius = radius;
    }
}

public class Tire extends Wheel {
    private double width;

    public Tire(double radius, double width) {
        // super(radius);
        this.width = width;
    }
}
```

No “extends”, so implicitly extends Object class

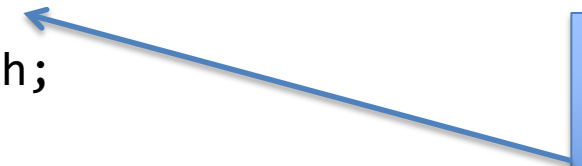
Since constructor provided, no default (0 argument) constructor provided or available.

Since no call to super(…) or this(…), Java inserts call to super(), Object constructor

If no call to super(…), Java inserts call to super(), which doesn’t exist. Result -> syntax error

Wheel Example (1)

```
public class Wheel {
    private double radius;

    public Wheel(double radius) {
        this.radius = radius;
    }
}

public class Tire extends Wheel {
    private double width;

    public Tire(double radius, double width) {
        super(radius);
        this.width = width;
    }
}
```



With call to super(…), superclass constructor called with specified argument

Terminology

- Student `extends` Person
- Student is a *subclass* of Person
- Person is a *superclass* of Student
- Person is the *parent* class, Student is the *child* class
- Person is the *base* class, Student is the *derived* class

- Superclass/subclass may be counterintuitive since the subclass has more "stuff" than the superclass
- Instead, think "superset/subset". Objects in class Student are a subset of objects in class Person

Video 3
Subclass Access and Overriding

More Inheritance

Access Restrictions and Visibility

Overriding and Hiding

instanceof

Reminder: Java Access Modifiers

- Can apply to members: fields and methods
- Modifiers control access to members from methods in other classes
- This list is from least to most restrictive:

Keyword	**Restriction**
`public`	None (any other method can access)
`protected`	Only methods in the class, subclasses, or in classes in the same package can access
[none]	Only methods in the class or in classes in the same package can access (called “package private”)
`private`	Only methods in the class can access

Subclass Access

- Subclasses cannot access private fields in their superclasses
- Two options:
  - Leave as is; provide accessors and/or mutators
  - Change private to protected
- Protected allows subclass access to superclass fields (even if the subclass is in a different package)
- General advice: use accessors and mutators

Overloading vs Overriding

- Overloading – In the same class, two methods with the same name, but different signatures

- Overriding – In a superclass and subclass, two methods with the same name, same signature

Overriding Methods

- A subclass method with the same signature as a superclass method *overrides* the superclass method
- The subclass method is executed *instead* of the superclass method
- Useful to change the behavior of a method when applied to a subclass object
- A method that is not overridden is inherited by (available to) the subclass

Accessing Overridden Methods

- Overridden methods can also be accessed using super: super.method(…)

Overriding Methods

```
public class Person {
    public void display() {
        System.out.println(name,address);
    }
}

public class Student extends Person {
    public void display() {
        System.out.println(getName(),getAddress(),classes,grades);
    }
}

public class Student extends Person {
    public void display() {
        super.display();
        System.out.println(classes,grades);
    }
}
```

The `instanceof` Operator

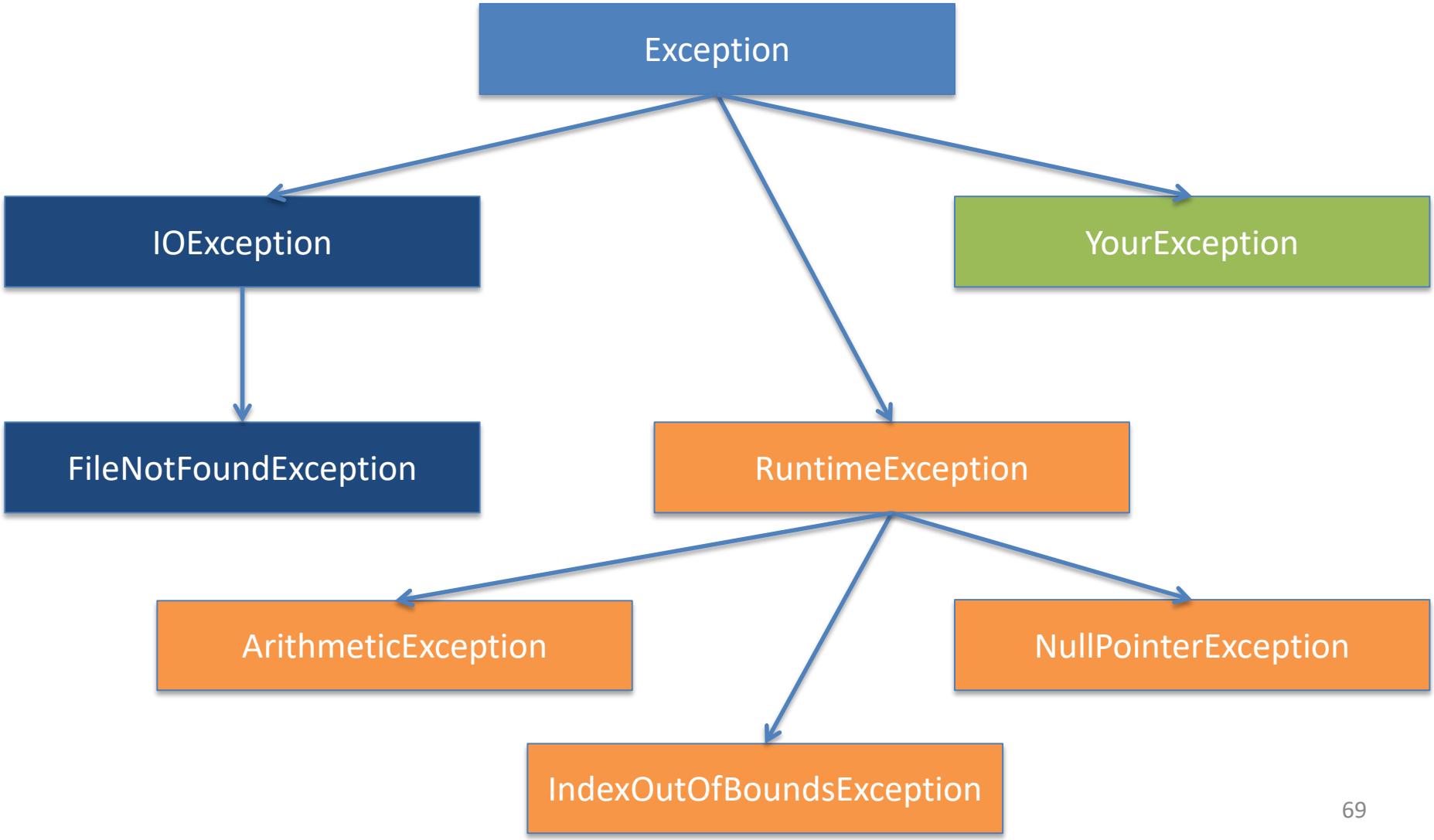
- It is possible to determine if an object is of a particular class (or subclass)
- The expression…

  `(objectA instanceof ClassB)`
- …evaluates `true` if the object referenced by `objectA` is an instance of the class `ClassB`
- `Person pers = (Person) ois.readObject();`
- `(pers instanceof Student)` is true if `pers` is an object of the subclass `Student`

Example: Object I/O (2)

```
class Tree implements Serializable {
    long circumference;
    String species;

    Tree(long circumference, String species) {
        this.circumference = circumference;
        this.species = species;
    }

    public String toString() {
        return String.format("%x: circumference = %d,  species = %s",
                hashCode(), circumference, species);
    }
}
```

Exception Class Hierarchy



Making Your Own Exception Class

```
public class StudentNotFoundException extends Exception {
    public StudentNotFoundException (String message) {
        super (message);
    }
}

public class FindStudent {
    public Student search (int student) throws
                StudentNotFoundException {
    if (...) {
        throw new StudentNotFoundException
            (Integer.toString(student));
        }
    }
}
```