Inheritance and Polymorphism

2020 Spring: AP Computer Science A

January 22nd, 2020

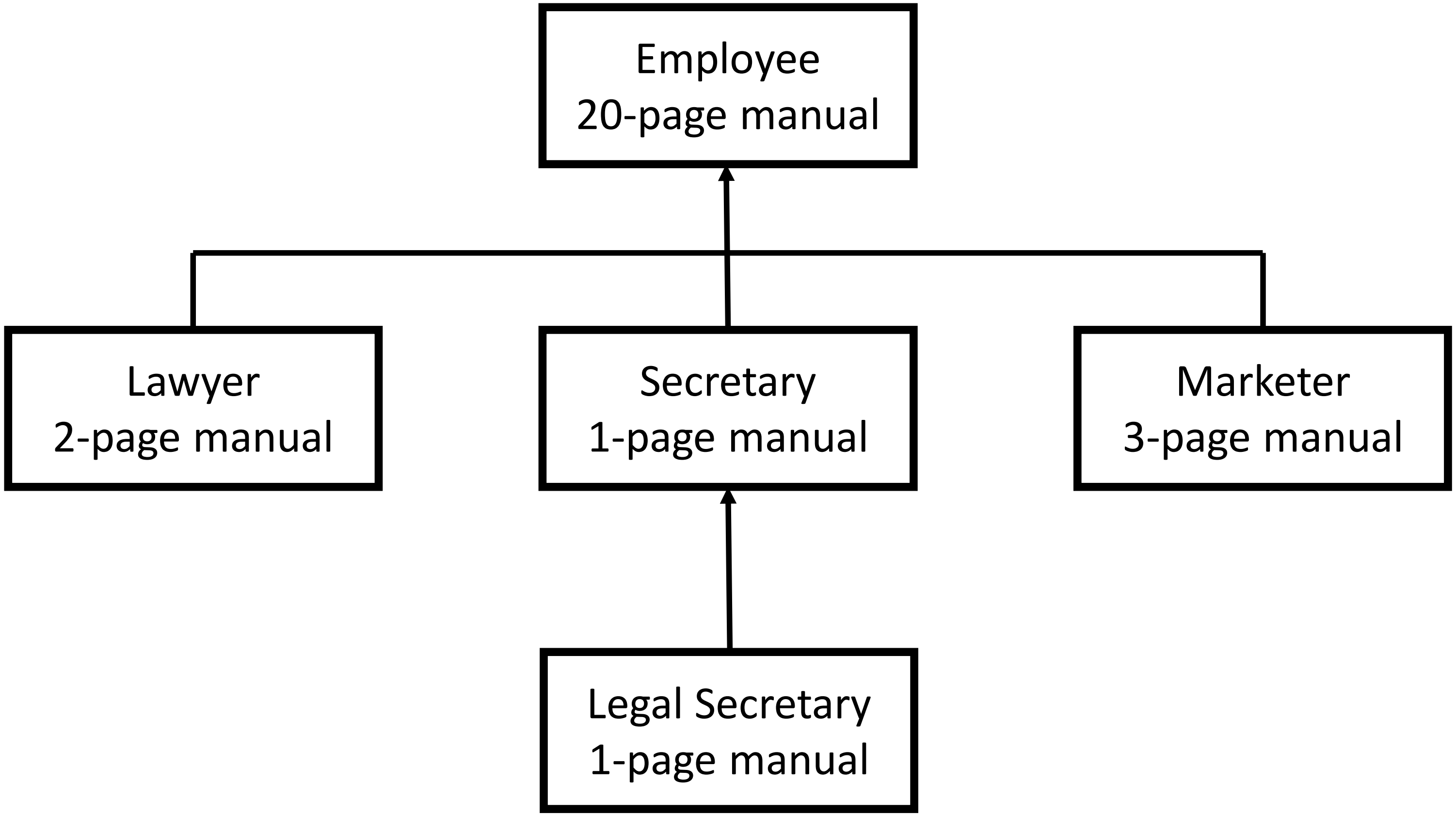
Today

- Inheritance
- Interacting with the superclass
- Class Object
- Comparing objects
- Polymorphism
- Abstract class
- Interface

Law Firm Employee Analogy

- **Consider employees in a law firm**
- **Common rules: Work hours, vacation days, benefits, regulations ...**
  - All employees attend a common orientation to learn company rules
  - Each employee receives a 20-page manual of common rules
- **Each subdivision also has specific rules**
  - Employee receives a smaller (1-3 page) manual of these rules
  - Smaller manual adds some new rules and changes some rules from the large (20-page) manual
- ***You want to design a software for managing the employees***

Law Firm Employee Analogy



Law Firm Employee Analogy

- *Why not just have a 22-page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?*

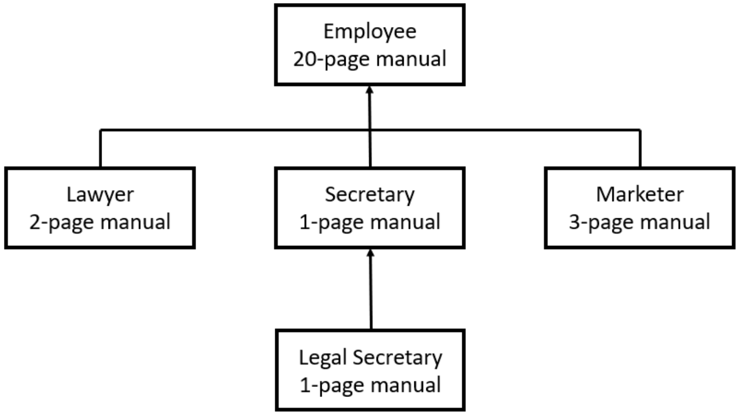
- **Some advantages of the separate manuals**
  - Maintenance: Only one update is needed when a common rule changes
  - Locality: Quick discovery of all rules specific to lawyers

- **Key ideas from this example**
  - General rules are useful (the 20-page manual)
  - Specific rules that my override general ones are also useful

Is-A Relationships, Hierarchies

- ***is-a relationship***: A hierarchical connection where one category can be treated as a specialized version of another
  - Every marketer ***is an*** employee
  - Every legal secretary ***is a*** secretary

- **Inheritance hierarchy:** A set of classes connected by is-a relationships that can share common code



Employee Regulations

- **Consider the following regulations for the employees**
  - Work 40 hours / week
  - Salaries
    - Employees make $40k / year
    - Legal secretaries make $5k extra / year
    - Marketers make $10k extra / year
  - Vacations
    - Employees get 2 weeks of paid vacation leave / year
    - Lawyers get an extra week / year
  - Vacation application form
    - Employees use a yellow form to apply for leave
    - Lawyers use a pink form to apply for leave

Employee Behavior

- **Each type of employee has some unique behavior**
  - Lawyers know how to sue
  - Marketers know how to advertise
  - Secretaries know how to take dictation
  - Legal secretaries know how to prepare legal documents

Employee Class

- **Simple Employee class**

```
public class Employee {
     public int getHours() {
          return 40;
     }

     public double getSalary() {
          return 40000.0;
     }

     public int getVacationDays() {
          return 10;
     }

     public String getVacationForm() {
          return "yellow";
     }
}
```

- **Exercise: Implement the Secretary class, based on the previous employee regulations and unique behavior (dictation)**

Secretary Class (Redundant)

```java
public class Secretary {
    public int getHours() {
        return 40;
    }

    public double getSalary() {
        return 40000.0;
    }

    public int getVacationDays() {
        return 10;
    }

    public String getVacationForm() {
        return "yellow";
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Code Sharing

- `takeDictation` is the only unique behavior in `Secretary`

- We would like to be able to say

```
public class Secretary {
    copy all the contents from the Employee class

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Inheritance

- ***inheritance*: A way to form new classes based on existing classes, taking on their data/states and behavior**
  - A way to *group* related classes
  - A way to *share code* between two or more classes

- One class can ***extend*** another, absorbing its data and behavior
  - ***superclass***: The parent class that is being extended
  - ***subclass***: The child class that extends the superclass and inherits its behavior
    - Subclass gets a copy of every field and method from the superclass

Inheritance Syntax

```
public class ClassName extends SuperClass {
    ...
}
```

- **Example**

```
public class Secretary extends Employee {
    ...
}
```

- **By *extending* `Employee` class, each `Secretary` object now**
  - Receives *all the methods* from the `Employee` class automatically
  - Can be *treated as* an `Employee` by client code (later in slide)

Improved Secretary Class

```
public class Secretary extends Employee {
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

- Now we only have to write the parts ***unique to each class***
  - Secretary will ***inherit*** these methods from `Employee`
    - `getHours, getSalary, getVacationDays, getVacationForm`
  - Secretary adds the `takeDictation` method

Implementing Lawyer Class

- **Recall the regulations for lawyer**
  - Extra week of paid vacation (total of 3)
  - Use pink form when applying for vacation leave
  - Unique behavior: Know how to sue

- **Problem:** *We want lawyers to **inherit most of the behavior** from employee, but we **want to replace some parts** with new behavior!*

Overriding Methods

- ***override*: To write a new version of a method in a subclass that *replaces* the superclass's version**
  - No special syntax is required to override a superclass method
  - *Just write a new version of it in the subclass*

```
public class Lawyer extends Employee {
    // overrides getVacationForm method in Employee class
    public String getVacationForm() {
        return "pink";
    }
}
```

- **Exercise:** Complete the Lawyer class
  - 3 weeks vacation, pink vacation form, can sue

Lawyer Class

```
public class Lawyer extends Employee {
    // overrides getVacationDays
    public int getVacationDays() {
        return 15;
    }

    // overrides getVacationForm
    public String getVacationForm() {
        return "pink";
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

- **Exercise:** Complete the `Marketer` class.
  - Marketers make $10k extra (total $50k) and know how to advertise

Marketer Class

```java
public class Marketer extends Employee {
    public double getSalary() {
        return 50000.0;
    }

    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
}
```

Levels of Inheritance

- ***Multiple levels of inheritance* in a class hierarchy are allowed**
  - A legal secretary is the same as a regular secretary but
    - Makes more money ($45k total)
    - Can file legal briefs

```
public class LegalSecretary extends Secretary {
    ...
}
```

- **Exercise:** Complete the `LegalSecretary` class

LegalSecretary Class

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 45000.0;
    }

    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }
}
```

Changes to Common Behavior

- **Imagine a company-wide change affecting *all employees***

- Everyone is given a $10k raise!
  - The base employee salary is now $50k
  - Legal secretaries now make $55k
  - Marketers now make $60k

- **Implementations should be modified to reflect this policy change**

Modifying the Superclass

```
public class Employee {
    ...

    public double getSalary() {
        return 50000.0;
    }

    ...
}
```

- Are we done?

- *The **subclasses** of* `Employee` ***are still incorrect***
  - They have overridden `getSalary` to return other values

Unsatisfactory Solution

```
public class Marketer extends Employee {
    public double getSalary() {
        return 60000.0;
    }

    ...
}

public class LegalSecretary extends Secretary {
    public double getSalary() {
    }   return 55000.0;


    ...
}
```

- **Problem:** The subclasses' salaries are ***based on the*** `Employee` ***salary,*** but the `getSalary` ***code does not reflect this***

Calling Overridden Methods

- **Subclasses *can call overridden methods* with super**

```
super.method(parameters)
```

- **Example:**

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return super.getSalary() + 5000.0;
    }
    ...
}
```

- **Exercise:** Modify `Lawyer` and `Marketer` to use `super`
  - `getVacationDays` should also be modified

Improved Subclasses

```
public class Marketer extends Employee {
    public double getSalary() {
        return super.getSalary() + 10000.0;
    }

    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
}


public class Lawyer extends Employee {
    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

    public String getVacationForm() {
        return "pink";
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

Inheritance and Constructors

- **Suppose we want to give employees more vacation days**
  - For each year worked, we'll award 2 additional vacation days
  - When an `Employee` object is constructed, we'll *pass in the **number of years** the person has been with the company*
  - We must *add some new states and behaviors* to the `Employee` class
- **Exercise:** Make necessary modifications to the `Employee` class

Modified Employee Class

```java
public class Employee {
    private int years;

    public Employee(int initYears) {
        years = initYears;
    }

    public int getHours() {
        return 40;
    }

    public double getSalary() {
        return 40000.0;
    }

    public int getVacationDays() {
        return 10 + 2 * years;
    }

    public String getVacationForm() {
        return "yellow";
    }
}
```

Problem with Constructors

- Suddenly, *the **subclasses do not compile***
  - Once we write a constructor (that requires parameters) in the superclass, ***we must now write constructors for subclasses***

- **Constructors are *not inherited*!**
  - Subclasses originally receives a default constructor that contains

```
public Lawyer() {
    super();     // calls Employee() constructor
}
```

  - But the constructor `Employee(int)` ***replaces the default constructor***
    - The subclasses' default constructors are *now trying to call a non-existent default constructor* in `Employee`

Calling Superclass Constructor

- **Call with super**

```
super(parameters);
```

- **Example**

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
    }
}
```

  - The super call ***must be the first statement in the constructor***

- **Exercise:** Make a similar modification to the `Marketer` class

Modified Marketer Class

```
public class Marketer extends Employee {
    public Marketer(int years) {
        super(years);
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }

    public void advertise() {
        System.out.println("Act now while supplies
    last!");
    }
}
```

- **Exercise:** Also modify the `Secretary` class
  - Secretaries' years of employment are *not tracked*
  - They do not earn extra vacation for years worked

Modified Secretary Class

```
public class Secretary extends Employee {
    public Secretary() {
        super(0);
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " +
    text);
    }
}
```

- Since `Secretary` ***doesn't require any parameters*** to its constructor, `LegalSecretary` ***compiles without a constructor***
  - `LegalSecretary`'s default constructor calls the `Secretary()` constructor

Inheritance and Fields

- **Suppose we give $5k raise for each year at the company**

```
public class Lawyer extends Employee {
    ...
    public double getSalary() {
        return super.getSalary() + 5000 * years;
    }
    ...
}
```

- **Does not work!**

- ***Private fields cannot be directly accessed from subclasses***
  - Because ***subclassing shouldn't break encapsulation***
  - How to solve this problem?
    - Add an *accessor method* for any field needed by the subclass, since methods will be inherited

Class `Object`

- **All classes have a superclass named `Object`**
  - Every class ***implicitly extends*** `Object`

- **The `Object` class defines several methods**
  - `public String toString()`
    - Returns a text representation of the object, often so that it can be printed
    - Automatically called in `System.out.println`
  - `public boolean equals(Object other)`
    - Compare the object to any other for equality, returns `true` if the objects have equal state

Object Variables

- **You can store any object in a variable of type `Object`**

```
Object o1 = new Point(5, -3);
Object o2 = "Hello, Java!";
Object o3 = new Scanner(System.in);
```

- **An `Object` variable only knows how to do *general* things**

```
String s = o1.toString();          // OK
int len = o2.length();             // error
String line = o3.nextLine();       // error
```

- **Methods can be written with `Object` parameters**

```
public void checkNull(Object o) {
    if(o == null) {
        throw new IllegalArgumentException();
    }
}
```

Comparing Objects

- **The == operator compares *reference to objects*, not their state**

- **The `equals` method *compares the state of objects***

```
if (str1.equals(str2)) {
    // equal
}
```

- **But if you *don't override* the `equals` method, it behaves like ==**
  - This is the *behavior we inherit from class* `Object`
  - Java doesn't understand how to compare user-created classes by default

Flaws on `equals` Method

- **We can change the default behavior by overriding!**
  - The method should compare the states of two objects and return true if they have the same state

- **What's wrong with this?** (`Point.java`)

```
public boolean equals(Point other) {
    return x == other.x && y == other.y;
}
```

Flaws on equals Method

- **It should be legal to compare a `Point` to any other object**

```
Point p = new Point(1, 2);
if (p.equals("hello")) {     // false
    // ...
}
```

  - `equals` should always return `false` if a non-`Point` is passed

- Parameter to `equals` ***must be of type*** `Object`
- `Object` is a ***general type*** that can match any `Object`
- Having an `Object` parameter means ***any object can be passed***
  - If we don't know what type it is, *how can we compare it?*
    - Don't know the passed object's fields beforehand

Class Casting

- **Solution:** ***Type-cast*** the `Object` parameter to a `Point`

```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- ***Casting objects is different compared to casting primitive types***
  - Really ***casting an*** `Object` ***reference into a*** `Point` ***reference***
  - ***Doesn't actually change the object*** that was passed
  - Tells the compiler to ***assume*** that `o` refers to a `Point` object

instanceof

- ***What if the assumption is not true?***
- Java *won't be able to cast* Object o into a Point
  - A **ClassCastException** is thrown

```
if (variable instanceof type) {
    statement(s);
}
```

- **instanceof asks *if a variable refers to an object of given type***
  - Used as a boolean test

instanceof

```
String s = "hello";
Point p = new Point();

if(s instanceof Point);      // false
if(s instanceof String);     // true
if(p instanceof Point);      // true
if(p instanceof String);     // false
if(p instanceof Object);     // true
if(s instanceof Object);     // true
if(null instanceof String);  // false
if(null instanceof Object);  // false
```

Correct equals Method

```
public boolean equals(Object o) {
    if (o instanceof Point) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
    }
    return false;
}
```

- Always check the type first
- If the type is same, cast and compare the states
- If the type is different, the two objects are not equal, so return `false`

Polymorphism

- ***polymorphism*: Ability for the same code to be used with different types of objects and behave differently with each**
  - Selecting the *appropriate method for a particular object* in a class hierarchy
  - ***Only applies to overridden methods in subclasses***

- A variable of type T ***can hold an object of any subclass of T***

```
Employee ed = new Lawyer(1);
```

  - You can call any methods from the `Employee` class on ed

- **When a method is called on ed, it behaves as a Lawyer**

```
System.out.println(ed.getSalary());           // Lawyer salary
System.out.println(ed.getVacationForm());     // pink
```

Polymorphism and Parameters

- **You can pass any subtype of a parameter's type**

```
public static void main(String[] args) {
    Lawyer lisa = new Lawyer(0);
    Secretary steve = new Secretary();
    printInfo(lisa);
    printInfo(steve);
}

public static void printInfo(Employee empl) {
    System.out.println(empl.getSalary());
    System.out.println(empl.getVacationDays());
    System.out.println(empl.getVacationForm());
}
```

- ***dynamic binding*: Making a run-time decision about which instance method to call**
  - The methods in `printInfo` will depend on the type of the actual object empl refers to

Polymorphism and Arrays

- **Array of superclass types can store any subtype as elements**

```
Employee[] e = { new Lawyer(0), new Secretary(),
                 new Marketer(0), new LegalSecretary() };

for (int i = 0; i < e.length; ++i) {
    System.out.println(e[i].getSalary());
    System.out.println(e[i].getVacationDays());
}
```

Casting References

- **A variable can only call that type's methods, not a subtype's**

```
Employee ed = new Lawyer(1);
int hours = ed.getHours();        // OK
ed.sue();                         // compile error
```

  - Compiler: *ed could store any kind of an employee but not all kinds know how to sue, I cannot compile this!*

- **To use Lawyer methods on ed, we should *type-cast* it**

```
Lawyer realEd = (Lawyer) ed;
realEd.sue();

((Lawyer) ed).sue();              // short version
```

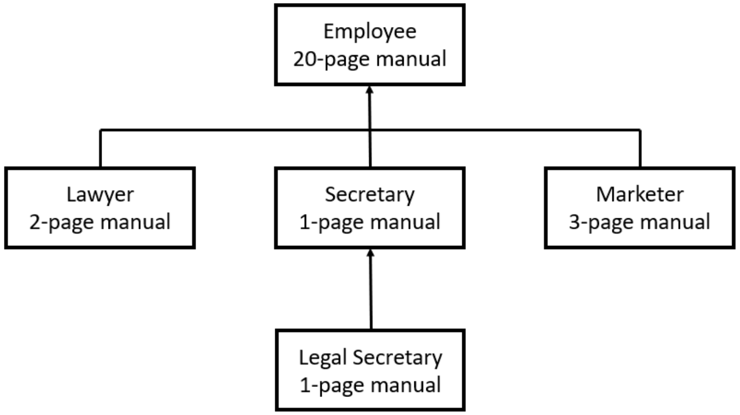
Casting References

- **The code crashes if you cast an object too far down the hierarchy**

```
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi");
((LegalSecretary) eric).fileLegalBriefs();
// ClassCastException – Secretary is not a LegalSecretary
```

- **You can only cast up and down the tree, not sideways**

```
Lawyer linda = new Lawyer(0);
((Secretary) linda).takeDictation("Hi");
// ClassCastException – Lawyer is not a Secretary
```



Polymorphism Problem

```
class Ham {
    public void a() {
        System.out.print("Ham a ");
        b();
    }

    public void b() {
        System.out.print("Ham b ");
    }

    public String toString() {
        return "Ham";
    }
}

class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b ");
    }
}
```

```
class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a ");
        super.a();
    }

    public String toString() {
        return "Yam";
    }
}

class Spam extends Yam {
    public void b() {
        System.out.print("Spam b ");
    }
}
```

Polymorphism Problem

- **What is the output?**

```
public static void main(String[] args) {
    Ham[] food = { new Lamb(), new Ham(), new Spam(), new Yam() };
    for (int i = 0; i < food.length; ++i) {
        System.out.println(food[i]);
        food[i].a();
        System.out.println();
        food[i].b();
        System.out.println();
        System.out.println();
    }
}
```

Polymorphism at Work

- ***Lamb's a inherits Ham's a. a calls b. But Lamb overrides b...***
    - How would Lamb's a method be executed?
    - `Ham a Lamb b`

```
class Ham {
    public void a() {
        System.out.print("Ham a ");
        b();
    }

    public void b() {
        System.out.print("Ham b ");
    }

    public String toString() {
        return "Ham";
    }
}

class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b ");
    }
}
```

Polymorphism at Work

- **Exercise:** Try an create a class table
  - Write the output of the method corresponding to the class

method	Ham	Lamb	Yam	Spam
a		Ham a b()		
b				
toString				

Polymorphism at Work

- **Exercise:** Try an create a class table
  - Write the output of the method corresponding to the class

method	Ham	Lamb	Yam	Spam
a	Ham a **b()**	Ham a **b()**	Yam a Ham a **b()**	Yam a Ham a **b()**
b	Ham b	Lamb b	Lamb b	Spam b
toString	Ham	Ham	Yam	Yam

Polymorphism at Work

- `b()` will correspond to each class's `b()` method

method	Ham	Lamb	Yam	Spam
a	Ham a **Ham b**	Ham a **Lamb b**	Yam a Ham a **Lamb b**	Yam a Ham a **Spam b**
b	Ham b	Lamb b	Lamb b	Spam b
toString	Ham	Ham	Yam	Yam

Polymorphism at Work

- **Final output will look like this**

```
Ham
Ham a Lamb b
Lamb b

Ham
Ham a Ham b
Ham b

Yam
Yam a Ham a Spam b
Spam b

Yam
Yam a Ham a Lamb b
Lamb b
```

Abstract Class

- ***abstract class*: A superclass that represents an abstract concept**
  - Abstract classes ***cannot be instantiated***
    - Cannot use new to create an instance of an abstract class
  - Abstract classes control the state and behavior that will be inherited by subclasses
  - Abstract classes may contain ***abstract methods***

- ***abstract method*: A method that has *no implementation code*, but has a *header***
  - We declare a method abstract, if there is *no good default code for the method*
  - Abstract methods work as a *placeholder*
    - Declaration for ***generic behaviors*** that subclasses **must override**
  - If a class contains any abstract methods, the class *must be declared an abstract class*

Abstract Class

- **Syntax**

```
public abstract class ClassName {
    public abstract type name(parameters);
}
```

- **A class can be abstract even if it has no abstract methods**
- **You can create variables (but not objects) of the abstract type**

- **Exercise:** Create an abstract Shape class
  - Fields: name (string)
  - Methods: area (abstract double), perimeter (abstract double)

Shape Class Example

```
public abstract class Shape {
    private String name;

    public Shape(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract double area();

    public abstract double perimeter();
}
```

- **If a class were to *extend* the Shape class, it should have area and perimeter as its methods**
- **Abstract methods will be implemented by the subclass**

Extending an Abstract Class

```
public class Circle extends Shape {
    private double radius;

    public Circle(double radius, String name) {
        super(name);
        this.radius = radius;
    }

    public double area() {
        return Math.PI * radius * radius;
    }

    public double perimeter() {
        return 2 * Math.PI * radius;
    }
}
```

- **Use extends, just like extending normal classes**
  - Abstract methods must be implemented

Extending an Abstract Class

- **If you don't implement the abstract methods, the class should be declared abstract**

```
public abstract class Quadrilateral extends Shape {
    public Quadrilateral(String name) {
        super(name);
    }
}
```

Notes on Abstract Classes

- An abstract class ***can have both fields and concrete (non-abstract) methods***
- A non-abstract subclass of an abstract superclass ***must provide implementation code for all abstract methods of the superclass***
- An abstract class may or may not have constructors
- It is ***illegal to instantiate*** abstract classes

Interface

- ***interface*: A list of methods that a class can implement**
  - Usually a collection of related methods
  - Cannot contain fields

- **Syntax**

```
public interface Name {
    public type name(parameters);

    public default type name(parameters) {
        statement(s);
    }
}
```

  - Methods in an interface are `abstract` by default
  - Or a default implementation can be provided

Shape Interface

- **The interface describes the features *common* to all shapes**

```
public interface Shape {
    public double area();

    public double perimeter();
}
```

  - Note that an interface cannot contain fields
  - All classes using this interface should implement the methods listed in this interface

Implementing an Interface

- **A class can declare that it *implements* an interface**

```
public class ClassName implements InterfaceName {
    // ...
}
```

- **Example**

```
public class Circle implements Shape {
    private double radius;

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

    public double area() {
        return Math.PI * radius * radius;
    }

    public double perimeter() {
        return 2 * Math.PI * radius;
    }
}
```

Extends + Implements

```
public abstract class Quadrilateral {
    private String name;

    public Quadrilateral(String name) {
        this.name = name;
    }
}

public class Rectangle extends Quadrilateral implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height, String name) {
        super(name);
        this.width = width;
        this.height = height;
    }

    public double area() {
        return width * height;
    }

    public double perimeter() {
        return 2 * (width + height);
    }
}
```

Abstract Class vs. Interface

- **Inheritance gives you an is-a relationship and code sharing**
  - A Lawyer object can be treated as an Employee, and Lawyer inherits Employee's code
- **Interfaces/Abstract classes *give you an is-a relationship without code sharing***
- ***Polymorphism works*** for both abstract classes and interfaces
- An interface cannot contain instance variables, whereas an abstract class can

- ***A class***
  - ***Can extend only one superclass***
  - ***But, can implement many interfaces***