Recap: Inheritance

- **Inheritance**: A way to allow a software developer to reuse classes by deriving a new class from an existing one, e.g.,
  - Secretary extends Employee
  - InstaPic extends Picture

- When constructing an object, Java makes sure that the constructor of the parent is first called
  - Insertion of super() (if no super…) as first statement in a constructor
  - slight complexity: Java adds a default constructor for each class, if the class has no constructor

- **Overriding** methods allow a child class to revise the behaviors of the parent class

- Design principle: when overriding parent’s behavior, derive behavior (e.g., 1.2 times base pay), not the final outcome (e.g., $60,000)

Example Extension

- **The boss wants to give employees more vacation days the longer they’ve been with the firm:**
  - For each year worked, 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 firm.
  - Exercise: Make necessary modifications to the Employee class.

Solution: Add Served Years to Employee

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

    public Employee(String name, int years) {
        this.name = name;
        this.years = years;
    }

    public Employee(String name) {
        this.name = name;
        years = 0; // 0 year service as default
    }

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

Outline

- Admin and recap
- Class inheritance
  - why and how?
  - inheritance and object construction
  - overriding and using overridden methods
  - inheritance and field access
Inheritance and Fields

**Setting:** To retain the best lawyers, the firm will pay a lawyer the base and $5000 for each year in the firm.

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

**Problem**

- Fields declared `private` cannot be accessed from subclasses
  - Reason: so that subclassing cannot break encapsulation
  - Q: how to get around this limitation?

**Solution 1**

- Java provides a third visibility modifier to denote fields/methods to be accessible by only child classes: `protected`

```java
public class Employee {
    private String name;
    protected int years;
    public Employee(String name, int years) {
        this.name = name;
        this.years = years;
    }
    ...
}
```

**Solution 2**

- Add an accessor for any field needed by the subclass

```java
public class Employee {
    private String name;
    private int years;
    public Employee(String name, int initialYears) {
        this.name = name;
        years = initialYears;
    }
    ...
    public int getYears() {
        return years;
    }
    ...
}
```

**Discussion**

- Which design do you like?
  - Design 1: make `year` `protected`
  - Design 2: Add `public getYear()`

- Settings where `protected` is more appropriate than adding a public accessor?
  - Adding `public getYear()` makes it available to not only child class, but also all other classes

**Outline**

- Admin and recap
- Class inheritance
  - why and how?
  - inheritance and object construction
  - overriding and using overridden methods
  - inheritance and field access
  - inheritance hierarchy
Levels of inheritance

- Multiple levels of inheritance in a hierarchy are allowed.
  - Example: A legal secretary is the same as a regular secretary but makes more money (10% more) and can file legal briefs.
  - Exercise: Implement the LegalSecretary class.

LegalSecretary class

```java
// A class to represent legal secretaries.
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }
    public double pay() {
        return super.pay() * 1.1;
    }
}
```

Partner class

- Partner is a senior lawyer that can get bonus. Thus it supports:
  - awardBonus(double bonus)

```java
// A class to represent partner.
public class Partner extends Lawyer {
    private double bonus;
    public void awardBonus(double bonus) {
        this.bonus = bonus;
    }
    public double pay() {
        return super.pay() + bonus;
    }
}
```

Class Hierarchies

- Many large-scale software systems define class hierarchies, where the root defines the common behaviors

```
Employee
  \-- Secretary
    \-- LegalSecretary
  \-- Lawyer
  \-- Marketer
  \-- Partner

Animal
  \-- Reptile
    \-- Snake
    \-- Lizard
  \-- Bird
    \-- Parrot
  \-- Mammal
    \-- Horse
    \-- Bat
```
A class called `Object` is defined in the `java.lang` package of the Java standard class library.

All classes in Java are derived from the `Object` class:
- Even if a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class.
- The `Object` class is therefore the ultimate root of all class hierarchies.

The `Object` class contains a few most basic methods, which are inherited by all classes:
- `toString()`
- `equals()`
- `clone()`

The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class and a hash value.

Every time we have defined `toString`, we have actually been overriding it.

Shortcut: If the parameter to the `println` method is an object, the method will invoke the `toString` method.

**Outline**
- Admin and recap
- Class inheritance
  - Why and how?
  - Inheritance and object construction
  - Overriding and using overridden methods
  - Inheritance and field access
  - Inheritance hierarchy
  - Inheritance and method binding

**Example Setting**
- It turns out that the vacation bonus policy does not apply to secretaries: they get fixed 10 days vacation, not \((10 + 2 \times \text{years})\).

**Solution 1**
- We set all Secretaries to 0 years of service because they do not have vacation bonus.

```java
public class Secretary extends Employee {
    public Secretary(String name, int years) {
        super(name, 0);
    }

    public void prepareDoc(String text) {
        System.out.println("Working on doc " + text);
    }
}
```

**Problem**
- Solution 1 is not good:
  - If we call `getYears()` on a Secretary object, we get 0.
  - What if we wanted to give other rewards to all employees based on years of service?
Idea: Separation

Separate base days and bonus days to allow adaptation

Improved Employee code

Separate the standard 10 vacation days from those that are awarded based on seniority.

```java
public class Employee {
    private String name;
    private int years;
    public int vacationDays () {
        return 10 + 2 * years;
    }
    ...
}
```

Improved Secretary code

Secretary can override getSeniorityVacation.

```java
public class Secretary extends Employee {
    public Secretary(String name, int years) {
        super(name, years);
    }
    public int getSeniorityVacation() {
        return 0;
    }
    public void prepareDoc(String text) {
        System.out.println("Working on text: " + text);
    }
}
```

Example Client

```java
public class Firm {
    public static void main(String args) {
        Lawyer larry = new Lawyer("Larry", 10);
        int vacDays = larry.getVacationDays();
    }
}
```

Summary

The method invoked is always determined by the object, not the class.

Hence when a method in a base class invokes the name of another method defined in the base class, the real method invoked can be defined either in the base class, or a child class.

This is called dynamic binding.
What is Polymorphism?

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

Recap: Reference Variables

- Interaction with an object occurs through object reference variables
- An object reference variable holds the reference (address, the location) of an object

```java
ChessPiece bishop1 = new ChessPiece();
```

Recap: Object Reference Variable

- Object reference variable assignment copies address, creating aliases

```java
bishop2 = bishop1;
```

Polymorphic Reference through Inheritance

- A variable of type T can hold an object of class T or descendant of T, e.g.,
  ```java
  Employee emp = new Employee("Ed");
  emp = new Lawyer("Larry");
  emp = new LegalSecretary("Lisa");
  ```
- When we use a variable v of type T to refer to objects both of base type T and descent of T, we say that it is a polymorphic reference through inheritance

Polymorphic Reference and Method

- You can call any methods defined in the based class T (e.g., Employee) class on polymorphic reference of type T (e.g., emp)
- When you invoke a method through a polymorphic reference variable, it is the type of the object being referenced, not the reference type, that determines which method is invoked.
- Careful use of polymorphic references can lead to elegant, robust software designs
Polymorphic Reference through Inheritance

Employee ed
Reference variable type
Object type: Lawyer
ed.vacationDays() // 15
Object type: Secretary
ed.vacationDays() // 10

Polymorphic Reference: Example

Employee emp;
emp = new Lawyer("Larry");
System.out.println ( emp.vacationDays() ); // OUTPUT: 15
System.out.println ( emp.vacationForm() ); // OUTPUT: pink

emp = new LegalSecretary("Lisa");
System.out.println ( emp.vacationDays() ); // OUTPUT: 10
System.out.println ( emp.vacationForm() ); // OUTPUT: yellow

Polymorphic Method Parameter

- Define a method that takes a reference to a base type and apply to all derived types.

This is how `print` in `PrintStream` is defined:
```java
void print(Object obj) {
    // all objects have the toString() method
    // convert to string and then output
}
```

Polymorphic Method Parameter: Example

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

    public static void printInfo(Employee emp) {
        System.out.println("salary: " + emp.pay());
        System.out.println("v.days: " + emp.vacationDays());
        System.out.println("v.form: " + emp.vacationForm());
    }
}
```

OUTPUT:
```
salary: 50000.0
v.days: 15
v.form: pink
```
```
salary: 50000.0
v.days: 10
v.form: yellow
```

Polymorphism and Arrays

- A common usage of polymorphism is to define an array of a base type, but different entries refer to different types of objects.
  - To handle a heterogeneous population of objects with uniformity

Polymorphism and Arrays: Example

```java
public class Staff {
    private Employee[] staffList;
    public Staff() {
        staffList = new Employee[4];
        staffList[0] = new Lawyer("Lisa");
        staffList[1] = new Secretary("Sally");
        staffList[2] = new Marketer("Mike");
        staffList[3] = new LegalSecretary("Lynne");
    }

    public void payday() {
        for (int count = 0; count < staffList.length; count++) {
            System.out.printf("%-10s: $%.2f
", staffList[count].name(), staffList[count].pay());
        }
    }
}
```

Works on any mix of Employee objects.
Include a new type of secretary who are paid by hours.