final

(the keyword, not the exam)
Motivation

- Suppose we’ve defined an Employee class, and we don’t want someone to come along and muck it up
  - E.g., we don’t want a CEO subclass that gives itself a raise

- The `final` keyword indicates that some definition (of a class, method, or field) cannot be changed or overridden by a subclass.
// A class to represent employees
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public final double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }
}

No subclass is allowed to change the definition of getSalary()!
final fields, methods, and classes

The final keyword can be applied to fields (as we’ve seen before):

```java
// no code may change the value of salary,
// including a subclass’s code
public final double salary = 40000.0;
```

Or to methods:

```java
// subclasses cannot override the getSalary method
public final double getSalary() { return salary; }
```

Or even to classes:

```java
// the Employee class cannot be extended
// It can’t have any subclasses at all!
public final class Employee { ... }
```
Exercise: Update the Employee class

- Add final declarations to the Employee class below so that different subclasses can have different salaries, but once their salary is set, they can’t be changed.

```java
public class Employee {
    private double salary;
    public Employee(int sal) {
        salary = sal;
    }
    public double getSalary() {
        return salary;
    }
}
```
abstract
Opposite of final

- The final keyword prevents subclasses from changing (overriding) code.

- Sometimes, you want to do the opposite:

  Force another programmer or piece of code to finish parts of a class.
Example: Employee salary

Let’s say you want every subclass of Employee to have a salary, but you want the subclass to decide what the salary should be.

We can define an “abstract” getSalary() method:

```java
public abstract double getSalary();
```

Note: no method definition!

- Abstract method declarations don’t provide definitions, just signatures.
- They are there to force subclasses to provide the definitions.
Abstract Rules (1)

1. **If a class has an abstract method**, or it inherits an abstract method that it doesn’t override, then the **class must be declared abstract**.

```java
public abstract class Employee {
    public abstract double getSalary();

    // you can mix abstract and non-abstract methods
    // in an abstract class
    public int getHours() { // Note: not abstract!
        return 40;
    }
}
```
Abstract Rules (2)

2. If a class is abstract, it can’t have a constructor.
   → No Employee object can be constructed
   → But you can declare Employee references.

```java
public abstract class Employee {
   public abstract double getSalary();

   public static void main(String [] args) {
      Employee e;  // NO ERROR: reference is fine
      e = new Employee(); // ERROR! No constructor
   }
}
```
Extending an abstract class

```java
public class Lawyer extends Employee {
    // since Employee declares an abstract getSalary,
    // Lawyer must define getSalary by overriding it
    // or else Lawyer must be an abstract class
    public double getSalary() {
        return 45000.0;
    }
}

public static void main(String [] args) {
    Employee e;  // Fine, no problem
    e = new Lawyer(); // Also fine (polymorphism)
    e = new Employee(); // ERROR!  No constructor!
}
```
Abstract classes: what’s the point?

- If you can’t construct objects for a class, what’s the point of the class? How can we use it?
  - Short Answer: polymorphism.
  - We can use references of type Employee as a place to store Lawyers, Secretaries, CEOs, etc.
  - Because getSalary() is declared in Employee, e.getSalary() is legal syntax, even though getSalary() is not defined in Employee.
Exercise

- Create an abstract ClosedShape class with an abstract getArea() method
- Include a non-abstract (aka, concrete) method in ClosedShape that prints the area of the shape to the screen.
- Write non-abstract (aka, concrete) subclasses Rectangle and Circle
- Write main methods for each that construct an object and print its area.
Interfaces

Completely abstract.
Going full abstract

What if our abstract class had no non-abstract methods?

```java
public abstract class Employee {
    public abstract double getSalary();
    public abstract int getHours();
    public abstract String getVacationForm();
}
```

Each subclass would have different definitions. They share only the names of their methods.

Java has an alternative way to do this: **interfaces**
Interfaces

Let's say you have the following two related classes:

```java
public class Scientist {
    public void discover() {
        System.out.println("Eureka! I have found it!");
    }

    public void publish() {
        System.out.println("My research is better than yours.");
    }
}

public class Engineer {
    public void discover() {
        System.out.println("Cool, what did I just do?"陛下
    }

    public void publish() {
        System.out.println("I don't know how this happened, but it works.");
    }
}
```

Neither of their methods do the same thing.
But they're still similar – they both discover and publish. Can we get code reuse?

interface Researcher {
    void discover();
    void publish();
}

But why? What does that accomplish?
    Now we can create Researcher references
Using Interface Objects

```java
public static void researchCycle(Researcher r) {
    r.discover();
    r.publish();
}

public static void main(String [] args) {
    Researcher researcher1 = new Scientist();
    Researcher researcher2 = new Engineer();
    // Interfaces have no constructors
    // They can only be used as types for references
    researcher2 = new Researcher(); // ERROR!

    researchCycle(researcher1);
    researchCycle(researcher2);
}
```
Using Interfaces

- Interfaces are a way of specifying what objects are capable of, without saying how.

- Interface variables can execute any of the methods listed in the interface, but the behavior depends on the class of the object.
  - That is, interface variables are **polymorphic**.

- There are no constructors for interfaces. They are not classes, and no objects of that run-time type are created. They are compile-time types for references.
Implementing Interfaces

public class Scientist implements Researcher {

    public void discover() {
        System.out.println("Eureka! I have found it!");
    }

    public void publish() {
        System.out.println("My research is better than yours.");
    }

}
public class Engineer implements Researcher {
    public void discover() {
        System.out.println("Whoa, what did I just do?\n");
    }

    public void publish() {
        System.out.println("I don't know how this happened, but it works.\n");
    }
}
Exercise: Comparable interface

- Define the Comparable interface

```java
public interface Comparable {
    public int compareTo(Object other);
}

public interface Comparable<T> {
    public int compareTo(T other);
}
```
Exercise: Dataset with Interfaces

- Below is a dataset class that finds the maximum and average of a set of ints.

- Modify the dataset class so that it works for any class which implements the right interface (that you define).

- Create a BankAccount class that implements your interface, and use Dataset to find the BankAccount with the maximum balance.
public class Dataset
{
    private int count = 0;
    private int maximum = Integer.MIN_INT;
    private double sum = 0;
    public void add(int x) {
        sum += x;
        count++;
        if(x > maximum) { maximum = x; }
    }
    public int getMaximum() { return maximum; }
    public double getAvg() { return sum / count; }
}
public class BankAccount {
    private double balance = 0;

    public BankAccount(double initBalance) {
        balance = initBalance;
    }
    public double getBalance() { return balance; }

    public static BankAccount getMaxAcct(
            BankAccount [] accounts)
    {
        // fill this in!
    }
}
public class BankAccount implements Measurable {
    private int balance = 0;
    public BankAccount(int initBalance) {
        balance = initBalance;
    }
    public int getBalance() { return balance; }
    public int getMeasure() {
        return getBalance();
    }
    public BankAccount getMaxAcct(BankAccount [] accounts) {
        Dataset d = new Dataset();
        for(int i=0; i<accounts.length; i++) {
            d.add(accounts[i]);
        }
        return (BankAccount)d.getMaximum();
    }
}