Announcements

• Project 1 spec and code is available on the course website. Due July 15th. Start early!
• Midterm I is next Wed in class from 11:00 – 1:00p.m. Open book: lecture notes, lab notes, but no laptop!
Today

• Constants
• Preventing Inheritance
• Abstract Classes
• Interfaces
Static Fields and Methods

class Parent {
    int x = 0;
    static int y = 1;
    static void f() {
        System.out.printf "Ahem!\n";
    }
    static int f(int x) {
        return x+1;
    }
}

class Child extends Parent {
    String x = "no";
    static String y = "way";
    static void f() {
        System.out.printf "I wanna!\n";
    }
}

Child dave = new Child();
Parent pDave = dave;

Child dave = new Child();
Parent pDave = dave;

dave.x => no
pDave.x => 0
dave.y => way
pDave.y => 1
dave.f() => ?
pDave.f() = ?
dave.f(1) => ?
pDave.f(1) => 2

Fields hide inherited fields of same name; static methods hide methods of the same signature.
Constants

• Adding `final` in front of a variable means that the value of the variable can never be changed.
  – You must initialize the value of the `final` variable when it is declared.
  – Use `final` variables to create constants.

```java
class Math {
    public static final double PI = 3.141592653…;
}
```

```java
public class System {
    public static final PrintStream out = … ;
}
```
Constants (cont.)

• Numerical values that are used repeatedly should be turned into a final constant.

```java
if (month == 3) { . . . } //BAD

public static final int MARCH = 3;
if (month == MARCH) { . . . }
```

• Not all constants need to be static.
  – For an array x, x.length is a final field.
Preventing Inheritance

- A class declared to be `final` cannot be extended by others

```java
final class Executive extends Manager{
}
```

- A method declared `final` cannot be overridden by subclasses
  - All methods in a final class are automatically `final`

```java
class Employee
{
    ...
    public final String getName()
    {
        return name;
    }
    ...
}
```
Abstract Classes

• As we move up the inheritance hierarchy, classes become more general or more *abstract*.

• At the some point, the ancestor class becomes so general that it becomes a basis for other classes rather than a class with specific instances that you want to create.

• Such ancestor classes are declared to be *abstract*. Their sole purpose is to be extended.
• Some attributes and methods make sense for every human, and these can be factored out into the Human class.

```java
public class Human {
    // ... 
    private int age;
    private String name;

    public int getAge() { return age};
    public String getName() { return name};
    // ... 
}
```
Example (cont.)

public class Human {
    //...,
    public String introduce() { //how should we code this?
        return "";
    }
}

Student david = new Student( . . . );
david.introduce(); //"I’m David and I major in Computer Science"

Employee wendy = new Employee( . . . );
wendy.introduce(); //"I’m Wendy I work for Google"

• The Human class knows nothing about the person except the name and age.
• We can try:
    public String introduce() {
        return "";
    }
• A better way:
    public abstract String introduce(); //no implementation required
Abstract Class

```java
public abstract class Human {
    private String name;

    public Human(String n);

    public String getName() { return name; }

    public abstract String introduce();
}
```

- Any class with one or more abstract methods must be declared as abstract.
- abstract classes can have concrete data and methods.
- abstract methods act as placeholders for methods that are implemented in subclasses.
public abstract class Human {
    protected String name;

    ... 

    public abstract String introduce();
}

public class Student extends Human{
    private String major;

    public Student(String name, String major) {...}

    public String introduce() {
        return "I'm " + name + "and I major in" + major;
    }
}

public class Employee extends Human{
    private String company;

    public Employee(String name, String company) {...}

    public String introduce() {
        return "I'm " + name + "and I work for " + company;
    }
}
**Rules for abstract Classes**

- **The concrete classes** Student and Employee **must implement the abstract method** `introduce()`.
  - The compiler will generate an error if you don’t.
- **If you extend an abstract class without implementing an abstract method, the subclass must be declared as abstract.**
- **You can create a variable of an abstract class:**
  ```java
  Human student;  //OK
  ```
- **You cannot create an object of an abstract class:**
  ```java
  //Compile time ERROR!
  Human student = new Human(“David”);
  
  //OK!
  Human student = new Student(“David”);
  ```
  `student.introduce();`
Example

```java
Human[] human = new Human[2];
human[0] = new Employee( ... );
human[1] = new Student( ... );

for (int i = 0; i < human.length; i++) {
    human[i].introduce();
}
```

• The call `human[i].introduce();` is well defined because `human[i]` never refers to an object of the abstract `human` class, it must refer to a concrete subclass.
What are Interfaces?

- Describing what classes *should* do, without specifying *how* they would do it.
- Think of it as a contract for a set of classes.
  “If your class conforms to the requirements set in this contract (or this interface), then I’ll perform these services for the objects of your class”
Example

• The sort method of the Array class promises to sort an array of objects, but under one condition: the objects in the array must implement the Comparable interface:

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

Like an abstract method, no implementation is provided
Example

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

public class Employee implements Comparable {
    public int compareTo(Object other) {
        ...
    }
}

The Employee class must implement the compareTo method. i.e., define a method named compareTo that takes an Object and returns an int.
compareTo

- **When we call** `x.compareTo(y)`, the method returns an indication whether `x` or `y` is larger:
  - return a negative number if `y` is larger
  - a positive number if `x` is larger,
  - zero otherwise.

- **Suppose we want to sort by salary:**

```java
public class Employee implements Comparable {
    public int compareTo(Object other) {
        Employee otherEmployee = (Employee) other;
        if (salary < otherEmployee.salary) return -1;
        if (salary > otherEmployee.salary) return 1;
        return 0;
    }
}
```
Example
• If the objects are not equal, it does not matter what negative or positive value your return.

• For instance, to sort by employee ID number:

  //negative if the first ID is less than the other,
  //0 if they are the same;
  //some positive value otherwise.
  return id - other.id;

• The subtraction trick works only for integers:

  salary - other.salary can round to 0 if the salaries are close enough.
Comparable

• An interface to describe Objects that have a natural order on them, such as String and Integer.

• Might use in a general-purpose max function:

```java
/** The largest value in array A, or null if A empty. */
public static Comparable max (Comparable[] A)
{
    if (A.length == 0) return null;
    Comparable result = A[0];
    for (int i = 1; i < A.length; i += 1)
        if (result.compareTo(A[i]) < 0) result = A[i];
    return result;
}
```

• Now max(S) will return maximum value in S if S is an array of Strings, or any other kind of Object that implements Comparable.
Properties of Interfaces

• Interfaces are not classes, you can’t construct objects:
  
  ```java
  x = new Comparable( ... ); //ERROR
  ```

• You can declare interface variables:
  
  ```java
  Comparable x; //OK
  ```

• An interface variable must reference an object of a class that implements the interface.
  
  ```java
  x = new Employee(...); //OK
  ```
Properties of Interfaces

• All methods in an interface are automatically public.
  - `int compareTo(Object other);`
  was not explicitly declared to be public. The implementation in Employee class must be
  `public int compareTo(Object other);`
  otherwise the compiler will complain.

• All fields in are automatically public static final.
  - Interfaces can contain only constants, never instance variables.
Extending Interfaces

• You can extend interfaces to go from greater degree of generality to a greater degree of specialization.

• Example: Java Collections Framework

```java
public interface Set extends Collection{
  .  .  .}
```
Interfaces and Abstract Classes

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

```java
public class Employee extends Comparable {
    int compareTo(Object other) {
        ...
    }
}
```

```java
public class Employee extends Human, Comparable //ERROR

public class Employee extends Human implements Comparable //OK
```

- A class can have one and only one superclass.
Implement Multiple Interfaces

```java
interface Readable {
    Object get();
}

interface Writeable {
    void put(Object x);
}

class Source implements Readable {
    Object get();
}

class Sink implements Writeable {
    void put(Object x);
}

class Variable implements Readable, Writeable {
    Object get();
    Object put();
}

void copy(Readable r, Writeable w) {
    w.put(r.get());
}
```
Using Interfaces and Abstract Class

- Example
The Java Collections Framework
Next Lecture

• Readings:
  – Head First Java, Chapter 7