Announcements/Follow-ups

• Midterm #2 this Friday
• Projects:
  – Code reviews on submit server: feedback on style
  – P4 due today
  – P5 posted today
  – Study set 5 answers posted today
• Game of bugs/Bugs of life
• Arrays and mutability
Built-in array support

- **Java.lang.System**
  ```java
  public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) {...}
  ```

- **Java.util.Arrays**
  - Copying
  - Sorting
  - Searching
  - toString
Immutable Classes

• Immutable objects can never be changed once they are constructed
  – Strings (vs StringBuffer)
  – Integers
  – IntLists and CharLists?

• Advantages: Primitive-like.
  – Simple, bug-resistant, tamper-resistant
  – Never need to be copied

• Immutable classes vs enums?
Immutable design

• Classes are immutable *by design*

• Designing immutable classes:
  – Declare the class with the final keyword (more on this later)
  – Fields must be private or final
  – No setter methods
  – All mutable reference fields must be copied
    • When initialized
    • When returned by getters
  – Multi-threading considerations
Immutability examples

• Position/IntVector example revisited
• Privacy leaks: Primate/Dog example
  – If not designed carefully, private variables might still be accessible.
• No deep copies for immutable objects: StringArray example
StringBuffer

• Mutable version of String
  – Can replace characters
  – Can insert characters
  – Can increase length
• StringBuffer example
public class Square {
    double s;
    public Square(double s) {
        this.s = s;
    }
    public void dilate(double scale) {
        s *= scale;
    }
    public double area() {
        return s*s;
    }
    public String toString() {
        return "A Square with side "+s;
    }
}
public class Circle {
    double r;
    public Circle(double r) {
        this.r = r;
    }
    public void dilate(double scale) {
        r *= scale;
    }
    public double area() {
        return Math.PI*r*r;
    }
    public String toString() {
        return "A Circle with radius "+r;
    }
}
public class Triangle {
    double a, b, c;
    public Triangle(double a, double b, double c) {
        this.a = a; this.b = b; this.c = c;
    }
    public void dilate(double scale) {
        a *= scale; b *= scale; c *= scale;
    }
    public double area() {
        double s = (a+b+c)/2; return Math.sqrt(s*(s-a)*(s-b)*(s-c));
    }
    public String toString() {
        return "A Triangle with sides "+a", "+b", and "+c;"
    }
}
Interfaces

public interface Shape {

    public void dilate(double scale);

    public double area();

    public String toString();

}
Interfaces

• An interface is like a source-code version of an API
  – (Static, final) constants are initialized
  – Methods are declared but not implemented
• Particular classes “implement” an interface by implementing its methods
  – The class must implement all methods in the interface
  – The class can contain other additional methods as well
    • E.g. Circle may have an additional circumference() method
public interface MyInterface {

    public int ONE_CONSTANT = 1;
    public int ANOTHER_CONSTANT = 1;

    public void oneMethod();
    public void anotherMethod();

}
public interface MyInterface {
    public int ONE_CONSTANT = 1;
    public int ANOTHER_CONSTANT = 1;

    public void oneMethod();
    public void anotherMethod();
}

Interface syntax
Interface keyword instead of class
public interface MyInterface {

    public int ONE_CONSTANT = 1;
    public int ANOTHER_CONSTANT = 1;

    public void oneMethod();
    public void anotherMethod();

}
public class MyClass implements MyInterface, YourInterface {

    public void oneMethod() {
        System.out.println("one");
    }

    public void anotherMethod() {
        System.out.println("another");
    }

    public void anotherMethod() {
        System.out.println("yours");
    }

}
public class MyClass implements MyInterface, YourInterface {

    public void oneMethod() {
        System.out.println("one");
    }
    public void anotherMethod() {
        System.out.println("another");
    }
    public void anotherMethod() {
        System.out.println("yours");
    }
}

Implements keyword
public class MyClass implements MyInterface, YourInterface {

    public void oneMethod() {
        System.out.println("one");
    }

    public void anotherMethod() {
        System.out.println("another");
    }

    public void anotherMethod() {
        System.out.println("yours");
    }
}

Can implement multiple interfaces
Interface syntax

```java
public class MyClass implements MyInterface, YourInterface {
    public void oneMethod() {
        System.out.println("one");
    }
    public void anotherMethod() {
        System.out.println("another");
    }
    public void anotherMethod() {
        System.out.println("yours");
    }
}
```

Implementations of each method from each implemented interface
Restrictions on interfaces

- All fields must be constant (static and final)
  - Static and final keywords are assumed if omitted
- All methods must be instance
  - No static or constructor
- All members (methods and constants) must be public
  - As an API, these are the members available to users of the interface. Inner-workings (e.g. private members) should not be specified.
  - Public keyword can be omitted
- The interface itself must be public or package-private
  - No modifier indicates package-private
  - Package-private interfaces are only accessible from the same package
Built-in interfaces

• CharSequence
  – Declares charAt(...), length(), and more
  – Implemented by String, StringBuffer, and more

• Comparable
  – Declares compareTo()
  – Implemented by Integer, Boolean, String, and many more
  – Any array whose elements implement Comparable can be automatically sorted with Arrays.sort()
Built-in interfaces

- List
  - Declares get(), set(), indexOf(), and many more
  - Implemented by many “Collections” (more later)
  - “Optional” methods?? (e.g. set())
    - Must be implemented, but may throw “UnsupportedOperationException”
    - The other alternative would be no optional methods, but many more built-in interfaces
    - “Optional” methods not advised in your own interfaces
Purposes of Interfaces

• Organization
  – Enforces encapsulation at compile-time
  – Provides a contract for users (other programmers)
  – Interfaces should be designed carefully
    • Many classes may eventually implement the interface
    • Changes to the interface requires changes to all classes that implement it

• Polymorphism
Polymorphism

• “Poly-morphism” $\leftrightarrow$ “Many-forms”
• Informally: The ability to type-cast non-primitive data
  
  \[
  \text{int } i = 3; \\
  \text{float } f = (\text{float}) i;
  \]

  \[
  \text{Circle } c = \textbf{new} \text{ Circle}(3); \\
  \text{Shape } s = (\text{Shape}) c;
  \]

• Inheritance also facilitates polymorphism (more later)
Explicit/implicit casting

The rules of type-casting depend on the “is-a” relationship

- $\mathbb{Z} \subset \mathbb{Q}$:
  - An integer **IS A** rational number
  - A rational number is not necessarily an integer

- Circles $\subset$ Shapes
  - A circle **IS A** shape
  - A shape is not necessarily a circle

- A String is never a shape
Explicit/implicit casting

Which type-casts are necessary?

• `int i = (int) 3.0;`
• `float f = (float) 3;`
• `Shape s = (Shape) new Circle(3);`
• `Circle c = (Circle) s;`
• `String str = (String) c;`
Explicit/implicit casting

Which type-casts are necessary?

- `int i = (int) 3.0; //Yes`
- `float f = (float) 3; //No`
- `Shape s = (Shape) new Circle(3); //No`
- `Circle c = (Circle) s; //Yes`
- `String str = (String) c; //Error`

“is-a” relationship gives the correct answer (“hiding/destroying data” does not)
Polymorphic references

• As long as the underlying object implements Shape, it can be referenced by a Shape variable.

  //Casts are not necessary here
  Shape s = (Shape) new Circle(3);
  s = (Shape) new Triangle(3);
  s = (Shape) new Square(3);
Polymorphic references

- A reference variable is a *memory address*
- An object is a data structure on the heap

**Stack**
- Circle c
- Triangle t
- Shape s

**Heap**
- Circle object
double r 3
- Triangle object
double a 3
double b 3
double c 3
Polymorphic references

- A reference variable is a *memory address*
- An object is a data structure on the heap

Stack
- Circle c
- Triangle t
- Shape s

Heap
- Circle object
  - double r   3
- Triangle object
  - double a 3
  - double b 3
  - double c 3
Polymorphic references

• The Shape variable can call all of (and only) the methods in the Shape interface
  s.area();  //Valid
  s.dilate(3.0);  //Valid
  s.toString();  //Valid
  s.circumference();  //Invalid
• But the underlying object determines which method gets called:
  s.area();  //Circle or triangle area?  //Depends on the underlying object
Dynamic binding

• Different classes can have methods with the same name, so method calls must be “bound” to objects before they are executed
  – “Early”/ “Static” binding: at compile time
  – “Late”/ “Dynamic” binding: at run time

• Java uses late binding
  – With interfaces and polymorphic references, the class of the underlying object is not necessarily known until run-time
  – Late binding allows this sort of polymorphism, but at the cost of efficiency