Interfaces
What do these things have in common?

What is different about them?
Differences

- Rotary phone
  - Sends number as pulses over wire
  - Sends analog voice signal
- Touch-tone phone
  - Sends number as DTMF tones over wire
  - Sends analog voice signal
- Mobile phone
  - Communicates with tower via radio waves
  - Sends compressed digital voice signal
What are the similarities?
Common interfaces make it easy to deal with different objects.
If you know that you need to dial a number, talk into a mouthpiece and listen to an earpiece...

...it doesn’t matter how the phone is implemented!
Imagine a SortedArray class

- Operations: add(), get(int), and remove(int)
- Every operation keeps the array in sorted order
  - So you’ll have to compare elements
- What is the element type?
For maximum usefulness, we’d like an element type of Object.
But how can you compare Objects?
Interfaces

• *Interfaces* provide a way to specify that a class will support certain methods.

• You can then write code that deals with interfaces *without knowing what classes it will deal with!*

• **How can this improve our designs?**
public interface Comparable
{
    int compareTo(Object o);
}

public class Fred implements Comparable {
    /* ... */
    public int compareTo(Object o) {
        /* ... */
    }
}

Comparable is thus a wider type than Fred -- you can give a reference to Fred to a Comparable variable.
Comparable is thus a wider type than Fred -- you can give a reference to Fred to a Comparable variable.

Comparable c = new Fred();
Any questions?
Interfaces recap

• Common real-world interfaces mean that you don’t have to relearn how to use a phone when you get a new one

• As programmers, we’re interested in reusing code that could deal with a number of similar things

• e.g. one method to sort an array of Strings, Integers, or any objects that understand a compareTo message
Interfaces recap

• Interface types are like classes except:
  • Only public methods
  • No method bodies
  • No instance fields
  • Can’t instantiate these

• Instead, you deal with a class that implements an interface
Polymorphism means that the method that executes depends on the type of the base object.
c.compareTo(o);
Which method is invoked?
Comparable c1 = new Integer(5);
Comparable c2 = “Hello!”
Sometimes you want to convert from an interface type to a class type.
Casting

Comparable c;

/* ... */

String s = (String)c;
You can declare interfaces, each in its own file.
public interface Comparable
{
    int compareTo(Object o);
}

You can implement one or several interfaces in your classes, if you provide implementations for the interface methods.
public class Clicker
    implements Comparable {
    /* ... */
    public int compareTo(Object o) {
        // what if o is not a Clicker?
        Clicker c = (Clicker)o;
        return this.clicks - c.clicks;
    }
}
Any questions?
Self-check

- Assume \( c \) is a Comparable. What does the following statement do? 
  \[
  \text{String } s = (\text{String})c;
  \]
Self-check

• What will Java do with the following code?

```java
Comparable c = new Integer(5);
String s = (String)c;
```
SortedArray exercise

- public void add(Comparable c)
- public Comparable get(int i)
- public void remove(int i)
- public int size()
So, how about sorting an existing array?
Problem: modify code that sorts an int[] so that it sorts a Comparable[]
public class Sorter {
    public static void merge(int[] array, int start, int mid, int end) {
        /* ... */
    }

    public static void sort(int[] array, int begin, int end) {
        /* ... */
    }
}
public class Sorter {
    public static void merge(int[] array,
                              int start, int mid, int end) {
        /* ... */
    }

    public static void sort(int[] array,
                             int begin, int end) {
        /* ... */
    }
}

What do we need to change here?
public class Sorter {
    public static void merge(int[] array,
                              int start, int mid, int end) {
        /* ... */
    }

    public static void sort(int[] array,
                             int begin, int end) {
        /* ... */
    }
}
We aren’t sorting an array of ints.

```java
public class Sorter {
    public static void merge(int[] array,
                              int start, int mid, int end) {
        /* ... */
    }

    public static void sort(int[] array,
                             int begin, int end) {
        /* ... */
    }
}
```
We aren’t sorting an array of ints.

```java
public class Sorter {
    public static void merge(Comparable[] array,
                              int start, int mid, int end) {
        /* ... */
    }

    public static void sort(int[] array,
                             int begin, int end) {
        /* ... */
    }
}
```
We aren’t sorting an array of ints.

```java
public class Sorter {
    public static void merge(Comparable[] array,
                              int start, int mid, int end) {
        /* ... */
    }

    public static void sort(Comparable[] array,
                             int begin, int end) {
        /* ... */
    }
}
```
We aren’t sorting an array of ints.

```java
public class Sorter {
    public static void merge(Comparable[] array, 
                              int start, int mid, int end) {
        /* ... */
    }

    public static void sort(Comparable[] array, 
                            int begin, int end) {
        /* ... */
    }
}
```

Should any of the other ints change?
public static void sort(Comparable[] array, int begin, int end) {
    int mid;
    if (end - begin <= 1) return;
    mid = (begin + end) / 2;
    sort(array, begin, mid);
    sort(array, mid, end);
    merge(array, begin, mid, end);
}
Do we need to make any changes to the body of `sort`?

```java
public static void sort(Comparable[] array, int begin, int end) {
    int mid;
    if (end - begin <= 1) return;
    mid = (begin + end) / 2;
    sort(array, begin, mid);
    sort(array, mid, end);
    merge(array, begin, mid, end);
}
```
Do we need to make any changes to the body of `sort`?

```java
public static void sort(Comparable[] array, int begin, int end) {
    int mid;
    if (end - begin <= 1) return;
    mid = (begin + end) / 2;
    sort(array, begin, mid);
    sort(array, mid, end);
    merge(array, begin, mid, end);
}
```

No. Why not?
public static void merge(Comparable[] array, int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    int[] temp = new int[end - start];
    while ((i < mid) && (j < end))
        if (array[i] <= array[j])
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    while (i < mid)
        temp[k++] = array[i++];
    while (j < end)
        temp[k++] = array[j++];
    for (i = start; i < end; i++)
        array[i] = temp[i - start];
}
public static void merge(Comparable[] array,
        int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    int[] temp = new int[end - start];
    while ((i < mid) && (j < end))
        if (array[i] <= array[j])
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    while (i < mid)
        temp[k++] = array[i++];
    while (j < end)
        temp[k++] = array[j++];
    for (i = start; i < end; i++)
        array[i] = temp[i - start];
}
public static void merge(Comparable[] array, int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    int[] temp = new int[end - start];
    while ((i < mid) && (j < end))
        if (array[i] <= array[j])
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    while (i < mid)
        temp[k++] = array[i++];
    while (j < end)
        temp[k++] = array[j++];
    for (i = start; i < end; i++)
        array[i] = temp[i - start];
}

You can’t put a square peg in a round hole.
...and you can’t put a Comparable reference in an int variable!
public static void merge(Comparable[] array, 
   int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    Comparable[] temp = 
      new Comparable[end - start];
    while ((i < mid) && (j < end))
      if (array[i] <= array[j])
        temp[k++] = array[i++];
      else
        temp[k++] = array[j++];
    while (i < mid)
      temp[k++] = array[i++];
    while (j < end)
      temp[k++] = array[j++];
    for (i = start; i < end; i++)
      array[i] = temp[i - start];
}
public static void merge(Comparable[] array, int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    Comparable[] temp =
        new Comparable[end - start];
    while ((i < mid) && (j < end))
        if (array[i] <= array[j])
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    while (i < mid)
        temp[k++] = array[i++];
    while (j < end)
        temp[k++] = array[j++];
    for (i = start; i < end; i++)
        array[i] = temp[i - start];
}
public static void merge(Comparable[] array, int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    Comparable[] temp =
        new Comparable[end - start];
    while ((i < mid) && (j < end))
        if (array[i] <= array[j])
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    while (i < mid)
        temp[k++] = array[i++];
    while (j < end)
        temp[k++] = array[j++];
    for (i = start; i < end; i++)
        array[i] = temp[i - start];
}
public static void merge(Comparable[] array, int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    Comparable[] temp =
        new Comparable[end - start];
    while ((i < mid) && (j < end))
        if (array[i] <= array[j])
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    while (i < mid)
        temp[k++] = array[i++];
    while (j < end)
        temp[k++] = array[j++];
    for (i = start; i < end; i++)
        array[i] = temp[i - start];
}
public static void merge(Comparable[] array, int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    Comparable[] temp =
        new Comparable[end - start];
    while ((i < mid) && (j < end))
        if (array[i] <= array[j])
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    while (i < mid)
        temp[k++] = array[i++];
    while (j < end)
        temp[k++] = array[j++];
    for (i = start; i < end; i++)
        array[i] = temp[i - start];
}
public static void merge(Comparable[] array, int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = 0;
    Comparable[] temp =
    new Comparable[end - start];
    while ((i < mid) && (j < end))
        if (array[i].compareTo(array[j]) <= 0)
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    while (i < mid)
        temp[k++] = array[i++];
    while (j < end)
        temp[k++] = array[j++];
    for (i = start; i < end; i++)
        array[i] = temp[i - start];
}
public static void merge(Comparable[] array, 
   int start, int mid, int end) {
   int i = start;
   int j = mid;
   int k = 0;
   Comparable[] temp =
       new Comparable[end - start];
   while ((i < mid) && (j < end))
      if (array[i].compareTo(array[j]) <= 0)
         temp[k++] = array[i++];
      else
         temp[k++] = array[j++];
   while (i < mid)
      temp[k++] = array[i++];
   while (j < end)
      temp[k++] = array[j++];
   for (i = start; i < end; i++)
      array[i] = temp[i - start];
}

Any other changes?
Summary of changes to Sorter

- type of arrays:
  - was int[]
  - should be Comparable[]

- Comparisons:
  - was array[i] <= array[j]
  - should be array[i].compareTo(array[j]) <= 0
Interfaces cheat sheet

• Big picture:
  • *interfaces* describe behaviors that classes can provide
  • interfaces allow programmers to write generalized code that deals with any class that implements the interface
  • interfaces limit coupling and allow code reuse
Interfaces cheat sheet

• Details:
  • declare an interface as you would a class, but with the `interface` keyword
  • interface declarations contain constants and public methods
  • declare a class that implements an interface (or interfaces) with the `implements` keyword
Interfaces cheat sheet

• More details:
  • If $C$ implements $I$, you can give a reference to a $C$ to a variable of type $I$ (vice versa requires a cast)
  • invoking an interface method on some reference $o$ will pick the right method for the kind of object $o$ refers to
  • (we call this property polymorphism)
Any questions?
Example: calculator

• We’d like to evaluate arithmetic expressions, like $1 + 2$

• How would we write a program to do this?
Example: calculator

- Idea: evaluate the expression
  - first, evaluate left-hand operand (1f), storing float result in a variable
  - next, evaluate right-hand operand (2f), storing float result in a variable
  - finally, combine the results using the operator (+), returning a float result (3f)
Example: calculator

• We’ll want to support the standard binary operations: addition, division, etc; as well as unary negation

• We’ll call operands Terms

• What behaviors do terms need to support?

• What behaviors do operations need to support?
We’d like to be able to evaluate a `Term` as well as turn it into a `String` (to print it out).

```java
public interface Term {
    float eval();
    String toString();
}
```
We’d like to be able to evaluate operations, set their operands, and turn them into Strings.

```java
public interface UnaryOp {
    float eval();
    void setOp(Term val);
    String toString();
}

public interface BinaryOp {
    float eval();
    void setLeftOp(Term val);
    void setRightOp(Term val);
    String toString();
}
```
What are the terms in $1 + 2$?
What are the terms in $1 + 2$?
What are the terms in $1 + 2$?

1  2
What are the terms in $1 + 2$?

(So we’ll need a Number class that implements Term.)
public class Number implements Term {
    private float f;
    public Number(float f) {
        this.f = f;
    }

    public float eval() {
        return f;
    }

    public String toString() {
        return String.valueOf(f);
    }
}

Calculating things

• We have a Number class that implements Term

• We’d like to evaluate a simple expression like $1 + 2$

• We just need an AdditionOp class that implements BinaryOp in order to evaluate $1 + 2$
Calculating things

- We have a `Number` class that implements `Term`.
- We'd like to evaluate a simple expression like `1 + 2`.
- What would code that uses `AdditionOp` look like?
- We just need an `AdditionOp` class that implements `BinaryOp` in order to evaluate `1 + 2`.
Number one = new Number(1.0f);
Number two = new Number(2.0f);
Number one = new Number(1.0f);
Number two = new Number(2.0f);

BinaryOp add = new AdditionOp();
Number one = new Number(1.0f);
Number two = new Number(2.0f);

BinaryOp add = new AdditionOp();
add.setLeftOp(one);
add.setRightOp(two);
Number one = new Number(1.0f);
Number two = new Number(2.0f);

BinaryOp add = new AdditionOp();
add.setLeftOp(one);
add.setRightOp(two);
float result = add.eval();
public class AdditionOp implements BinaryOp {
    private Term leftOp;
    private Term rightOp;

    /* ... some methods omitted ... */

    public float eval() {
        float left = leftOp.eval(),
                right = rightOp.eval();
        return left + right;
    }
}
What about expressions like \((1 + 2) + 3?\)
Well, what are the terms in \((1 + 2) + 3\)?
Well, what are the terms in \((1 + 2) + 3\)?
Well, what are the terms in \((1 + 2) + 3\)?
Well, what are the terms in \((1 + 2) + 3\)?

\[ 1 + 2 \quad 3 \]

If we want to support compound expressions, we need to support using *operations as terms*!
public class AdditionOp implements 
BinaryOp, Term {
    private Term leftOp;
    private Term rightOp;

    /* ... some methods omitted ... */

    public float eval() {
        float left = leftOp.eval(),
            right = rightOp.eval();
        return left + right;
    }
}
Number one = new Number(1.0f);
Number two = new Number(2.0f);
Number three = new Number(3.0f);
Number one = new Number(1.0f);
Number two = new Number(2.0f);
Number three = new Number(3.0f);

Term left = new AdditionOp();
left.setLeftOp(one);
left.setRightOp(two);
Number one = new Number(1.0f);
Number two = new Number(2.0f);
Number three = new Number(3.0f);

Term left = new AdditionOp();
left.setLeftOp(one);
left.setRightOp(two);

BinaryOp add = new AdditionOp();
Number one = new Number(1.0f);
Number two = new Number(2.0f);
Number three = new Number(3.0f);

Term left = new AdditionOp();
left.setLeftOp(one);
left.setRightOp(two);

BinaryOp add = new AdditionOp();
add.setLeftOp(left);
add.setRightOp(three);
Number one = new Number(1.0f);
Number two = new Number(2.0f);
Number three = new Number(3.0f);

Term left = new AdditionOp();
left.setLeftOp(one);
left.setRightOp(two);

BinaryOp add = new AdditionOp();
add.setLeftOp(left);
add.setRightOp(three);

float result = add.eval();
Notice that we didn’t change the `eval` method of `AdditionOp`!
Exercise: implement +, *, and - (both unary and binary) operations and a small tester main.

```java
public interface Term {
    float eval();
    String toString();
}

public interface UnaryOp {
    float eval();
    String toString();
    void setOp(Term val);
}

public interface BinaryOp {
    float eval();
    String toString();
    void setLeftOp(Term val);
    void setRightOp(Term val);
}
```
What’s wrong with our code?

- Most of the `eval` methods in the `BinaryOp` classes look roughly the same:
  - evaluate left operand, getting a float
  - evaluate right operand, getting a float
  - perform some operation on these two floats
What’s wrong with our code?

• Most of the eval methods in the BinaryOp classes look roughly the same:
  • evaluate left operand, getting a float
  • evaluate right operand, getting a float
  • perform some operation on these two floats
Note that two of these steps are the same for every operation!

- Most of the eval methods in the BinaryOp classes look roughly the same:
  - evaluate left operand, getting a float
  - evaluate right operand, getting a float
  - perform some operation on these two floats