

# Interfaces

Look ma, no code!

# Levels of Abstraction

- A concrete class is completely defined
- An abstract class has some functionality that is declared but not defined
- An interface is a bunch of declared functions with no definitions!

# Contracts

- An interface is like a contract
  - When you agree to a contract, there are certain things that you agree to do
  - When you implement an interface, there are certain methods you agree to do

# Example Interface

```
public interface Drawable
{
    public abstract void draw(Graphics g);
}
```

- No class
- No concrete methods
- No variables
- No constructors

# Example Interface

```
public interface Drawable
{
    void draw(Graphics g);
}
```

- The **public abstract** is implied (optional)
- You can never have private abstract methods (how could you implement them if you don't know about them?)

# Implementing Interfaces

```
public class Rectangle extends Shape implements Drawable  
{  
}
```

- You **implement** an interface, not extend!
- This class will not compile because it is not fulfilling the contract with Drawable.

# Implementing Interfaces

```
public class Rectangle extends Shape implements Drawable
{
    public void draw(Graphics g) {
        //drawing code
    }
}
```

- This class will compile because it implements the abstract method

# Implementing Interfaces

## v.2

```
public abstract class Oval extends Shape implements Drawable  
{  
  
}
```

- This class will compile
  - It has unimplemented methods
  - But it was also marked abstract (so okay)

# Implement Many

```
public class Bug implements Drawable, Movable, Actable
{
    //methods to implements
}
```

- A class may implement many interfaces!
- Still can't extend many classes =(

# Types

```
public class Rectangle extends Shape implements Drawable {  
}
```

- You can use any of the super types!
  - Rectangle r = new Rectangle();
  - Shape s = new Rectangle();
  - Drawable d = new Rectangle();
  - but now you can only use Drawable methods =(

# Aside

- Interfaces can have a specific type of variable: **public static final**
- These are constants that are accessible through the interface name.

```
public interface Mathable
{
    public static final double PI = 3.1415...;
    public static final double E = 2.718...;
}
```

# Aside

- The **public static final** part is implied
- `System.out.println(Mathable.PI);`
- Implementors can use variables directly!

```
public interface Mathable
{
    double PI = 3.1415....;
    double E = 2.718....;
}
```

# Aside #2

- Technically an interface can extend one (or many) superinterfaces (different than classes!)

```
public interface Readable
{
    String read();
}
```

```
public interface Writable
{
    void write(String);
}
```

```
public interface ReadWritable extends Readable, Writable
{
}
```

# Recap

- Interfaces...
  - have all methods automatically public abstract
  - have no constructors or fields
  - can have final constants technically
- Think of them as contracts

# Java Interface

```
public interface List<E>
{
    int size();
    boolean add(E obj);
    void add(int index, E obj);
    E get(int index);
    E set(int index, E obj);
    E remove(int index);
}
```

Why do these  
look familiar?

# ArrayList<E>

- ArrayList<E> implements List<E> (shock!)

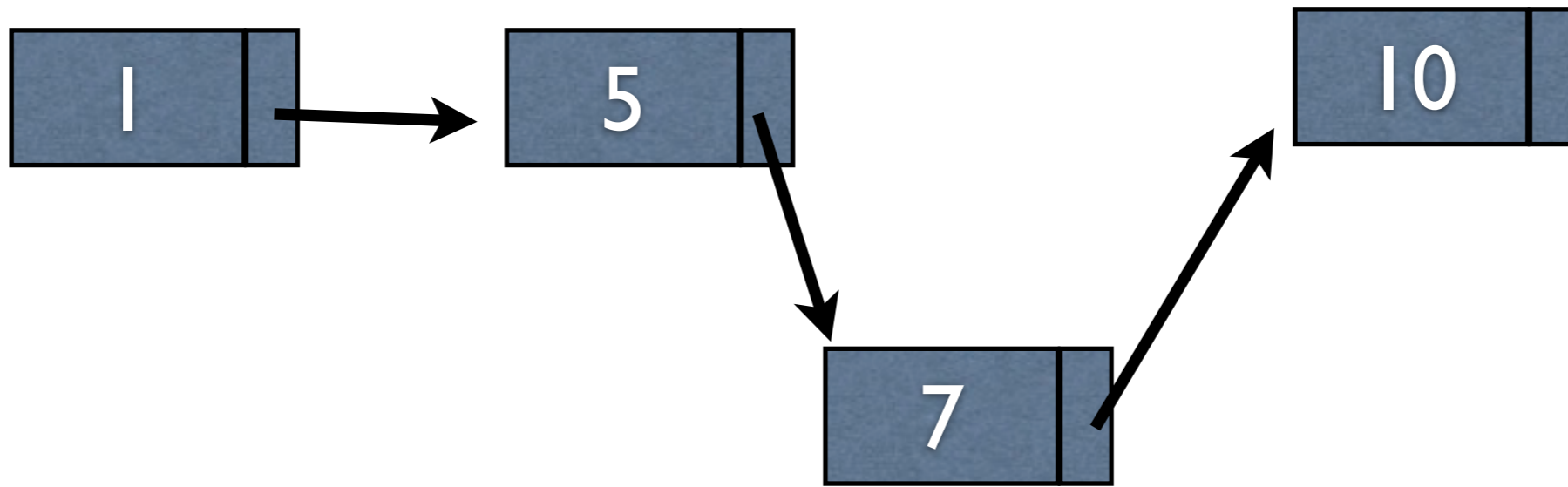
```
public class ArrayList<E> implement List<E>
{
    public int size() {
        return size;
    }

    //etc
}
```

# What's the point of List?

- Defines the general idea of a list
- ArrayList is a specific implementation of a list
- LinkedList is a different implementation (uses a bunch of connected objects instead of an array - no on AP test)

# LinkedList - Not on AP



- LinkedList is non contiguous memory
- Easy to insert! (no shifting)
- Hard to jump to spot...

# To Know

- When you see a `List<E>` variable on the AP test, treat it like an `ArrayList<E>`
- You **CANNOT** create a `List<E>`

```
List<String> strs = new ArrayList<String>();  
int s = strs.size();
```