Week 2 - Friday

# COMP 2000

# Last time

- What did we talk about last time?
- Interfaces
- Implementing interfaces

# Questions?

# Project 1

# Defining Interfaces

# Abstract methods

- Primarily, interfaces contain **abstract methods**
- Abstract methods are methods that must be implemented by any class that implements an interface
  - Unless that class is *also* abstract, which we'll talk about next week
- Whether in interfaces or abstract classes, abstract methods are ones that you **have** to have (even if what they do is up to you)

```
public interface Pokeable {
    boolean poke(); // Abstract method
}
```

# No constructors!

- In Java, it's not possible to specify a constructor in an interface
- In other words, you can't say how an object is created
- Abstract methods are always regular methods, never constructors or static methods

# Constants

- In addition to abstract methods, constants are commonly found in interfaces
- These constants should be values that are useful in the context of the interface
- Sometimes, the **only** purpose of an interface is to hold constants, such as the interface **`WindowConstants`**, which holds named **`int`** values describing what happens when a windows closes
- These constants are always implicitly **`public`**, **`final`**, and **`static`**
  - You don't have to mark them that way
  - You **can't** mark them as **`private`** or **`protected`**

```
public interface Dialable {
    int NUMBER_LENGTH = 10;
    void dial(String number);
}
```

# Accessing constants

- To refer to a constant from an interface, you always say the name of the interface, followed by a dot, followed by the name of the constant

```
int value = WindowConstants.DISPOSE_ON_CLOSE;
```

- Since they're constants, you (obviously) can't change them with an assignment

# Default methods

- As of Java 8, interfaces can also have default methods
- The interface expects you to implement these methods, but if you don't, a default implementation is provided

```
public interface Punchable {
    default boolean wantsPunch() { // Default
        return false;
    }
    void getPunched(Punch punch); // Abstract
}
```

# Static methods

- Before Java 8, you couldn't put static methods in interfaces at all
- Now, you **can** put static methods in interfaces, but they aren't abstract
- In other words, static methods in interfaces do not require a class that implements the interface to make a corresponding method
- Instead, a static method merely does some useful task related to the interface
- Note that static variables are **not** allowed in an interface, so a static method can only interact with its parameters

# Static method in interface example

- Static methods can be used as a utility method for an interface
- Here, for example, we provide a method that determines the area of a regular polygon

```
public interface RegularPolygon {
     double getLength();        // Length of each side
     int getSides();            // Number of sides

     static double getArea(RegularPolygon shape) {
          return 0.25 * shape.getSides() *
          shape.getLength() * shape.getLength() /
          Math.tan(Math.PI/shape.getSides());
     }
}
```

# Interfaces inside of interfaces?

- Yes!
- It's possible to put an interface inside of another interface
- Doing so simply treats the outer interface like a name-space for the inner interface
- You don't want to do this unless the inner interface is only needed in the context of the outer interface
- One example is the **`Map`** interface which contains an **`Entry`** interface
  - Maps (also called dictionaries) store (*key*, *value*) pairs
  - Classes that implement the **`Entry`** interface are able to return both the key and the value of a particular entry in the map

# Weird stuff

- It's also possible to put classes inside of interfaces
- You could make the argument that doing so makes sense for classes that are deeply tied to how the interface functions
  - But this is done very rarely
- You can define exceptions inside of interfaces
- You can also put enums inside of interfaces
  - Like inner interfaces, it uses the interface like a name-space
  - It might make sense to put an enum inside an interface if the interface requires constants of the enum type

# Extending Interfaces

# Interfaces can extend other interfaces

- Like classes, you can use inheritance to extend an interface
- When you do so, the child interface gets all of the required methods from the parent interface
- It can also reference the constants and static methods within the parent interface
- Consider the following interface:

```
public interface Defender {
    boolean blockWithShield(Attack attack);
}
```

# Child interface

- We can make a child interface from **`Defender`** using the **`extends`** keyword

```
public interface NinjaDefender extends Defender {
    boolean parryWithKatana(Attack attack);
}
```

- This interface contains the **`blockWithShield()`** abstract method as well as the **`parryWithKatana()`** abstract method
- A class that implements this interface must have both

# As many as you want!

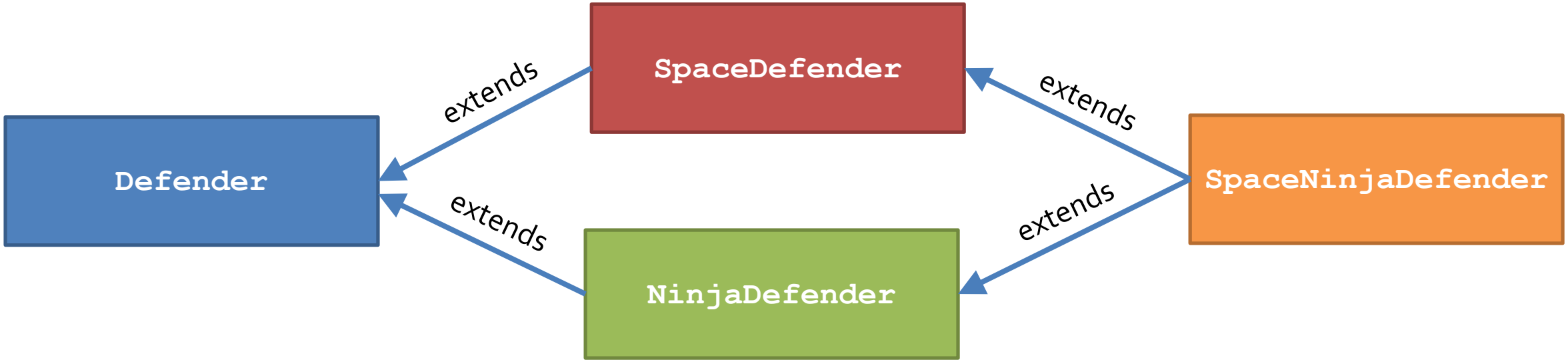
- Child classes can only have a single parent, but child interfaces can extend an unlimited number of parents

```
public interface PunchableNinjaDefender extends
    NinjaDefender, Punchable {
    void hateLife();
}
```

- The child interface gets the union of all the abstract methods and constants from all the parent interfaces

# You can have the same ancestor multiple ways

- We can even imagine that you have the same (great)grandparent in multiple ways




- We'll use UML class diagrams to show these and other inheritance relationships

# Interface Examples

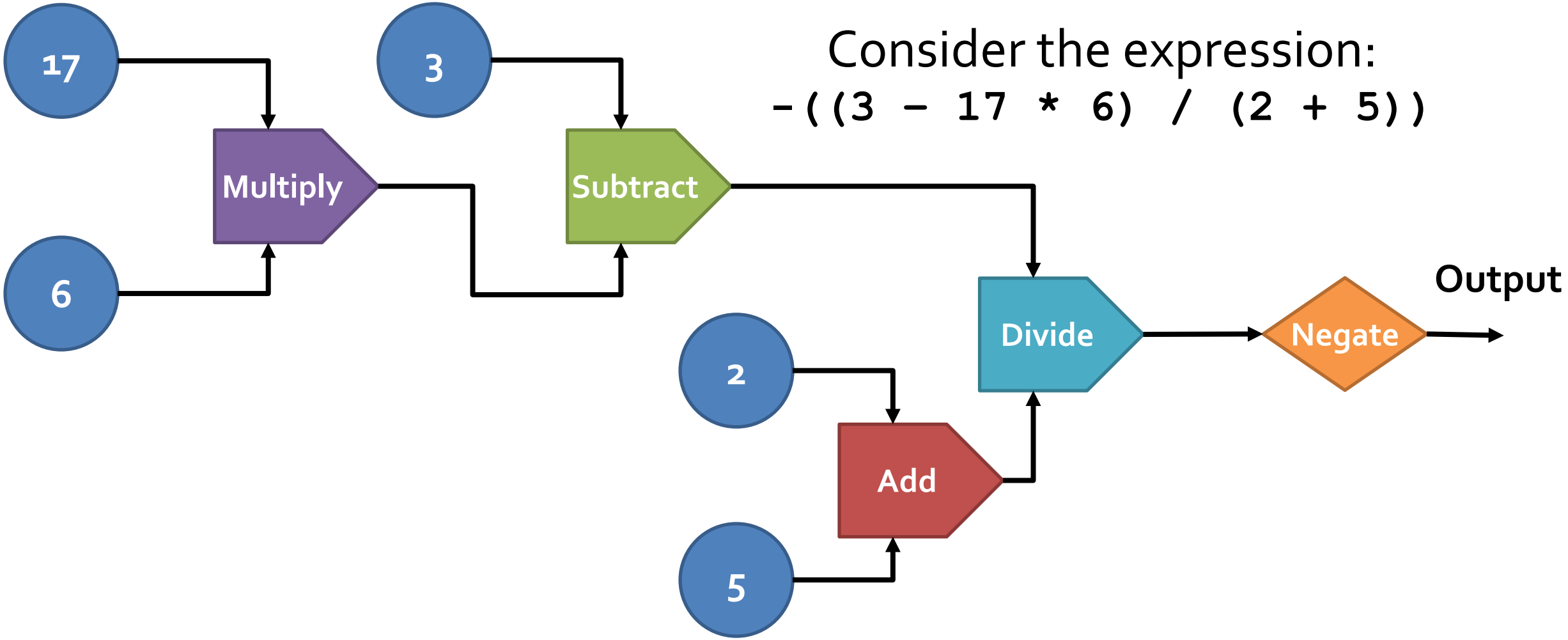
# Operations

- We can build a tree of operations that models an algebraic expression
- For example, a we could have operations like negate, add, subtract, multiply, and divide, with constant values that are **`double`** values
- Any algebraic expression could look like a tree of such operations and values

# Example tree

Consider the expression:

```
-((3 - 17 * 6) / (2 + 5))
```



# What interfaces would be useful?

- Every object in the expression has a value
- We can make an interface that they all implement that gives that value

```
public interface Value {
    double getValue();
}
```

# Numbers are just about that simple

- Concrete values could be represented by the **`Number`** class, which holds a constant value

```
public class Number implements Value {
     private double number;

     public Number(double number) {
          this.number = number;
     }

     public double getValue() {
          return number;
     }
}
```

# Binary operations

- Add, subtract, multiply, and divide are **binary operations**
- In this case, "binary" just means that they take two operands and has nothing to do with binary numbers
- They can be represented with an interface that extends the **`Value`** interface
- It might be useful to be able to retrieve the individual operands from any binary operation

```
public interface BinaryOperation extends Value {
    Value getOperand1();
    Value getOperand2();
}
```

# Example Add class

```
public class Add implements BinaryOperation {
	private Value operand1;
	private Value operand2;

	public Add(Value operand1, Value operand2) {
		this.operand1 = operand1;
		this.operand2 = operand2;
	}
	public double getValue() {
		return operand1.getValue() + operand2.getValue();
	}
	public Value getOperand1() {
		return operand1;
	}
	public Value getOperand2() {
		return operand2;
	}
}
```

# Unary operations

- Negate is the only unary operation that we have, but it's wise to plan for more
- Unary operations can be represented with an interface similar to **`BinaryOperation`**

```
public interface UnaryOperation extends Value {
    Value getOperand();
}
```

# Example Negate class

```
public class Negate implements UnaryOperation {
      private Value operand;

      public Negate(Value operand) {
            this.operand = operand;
      }

      public double getValue() {
            return -operand.getValue();
      }

      public Value getOperand() {
            return operand;
      }

}
```

# More classes

- It's easy to add **`Subtract`**, **`Multiply`**, **`Divide`** classes that implement the **`BinaryOperation`** interface
  - We could even add a **`Modulus`** class or a **`Power`** class
- Likewise, the **`UnaryOperation`** interface could be implemented with a **`BitwiseComplement`** class or others
- Note that **`Add`**, **`Subtract`**, **`Multiply`**, and **`Divide`** differ only by the operation they do in **`getValue()`**
  - They all have to declare **`operand1`** and **`operand2`**
  - It might make more sense for **`BinaryOperation`** to be an abstract class instead of an interface
  - Abstract classes are like interfaces except that they can contain methods and data and can be inherited from
  - Serious designers think a lot about how to make the right trade-offs

# Final usage of all the new classes

- The original tree could be modeled with the following code:

```
public class Math {

	public static void main(String[] args) {
		Multiply multiply = new Multiply(new Number(17), new Number(6));
		Subtract subtract = new Subtract(new Number(3), multiply);
		Add add = new Add(new Number(2), new Number(5));
		Divide divide = new Divide(subtract, add);
		Negate negate = new Negate(divide);

		System.out.println("Answer: " + negate.getValue());
	}

}
```

# Upcoming

# Next time…

- On Monday, we'll talk about class inheritance

# Reminders

- Read Chapter 11
- Keep working on Project 1