Interfaces

Introduction

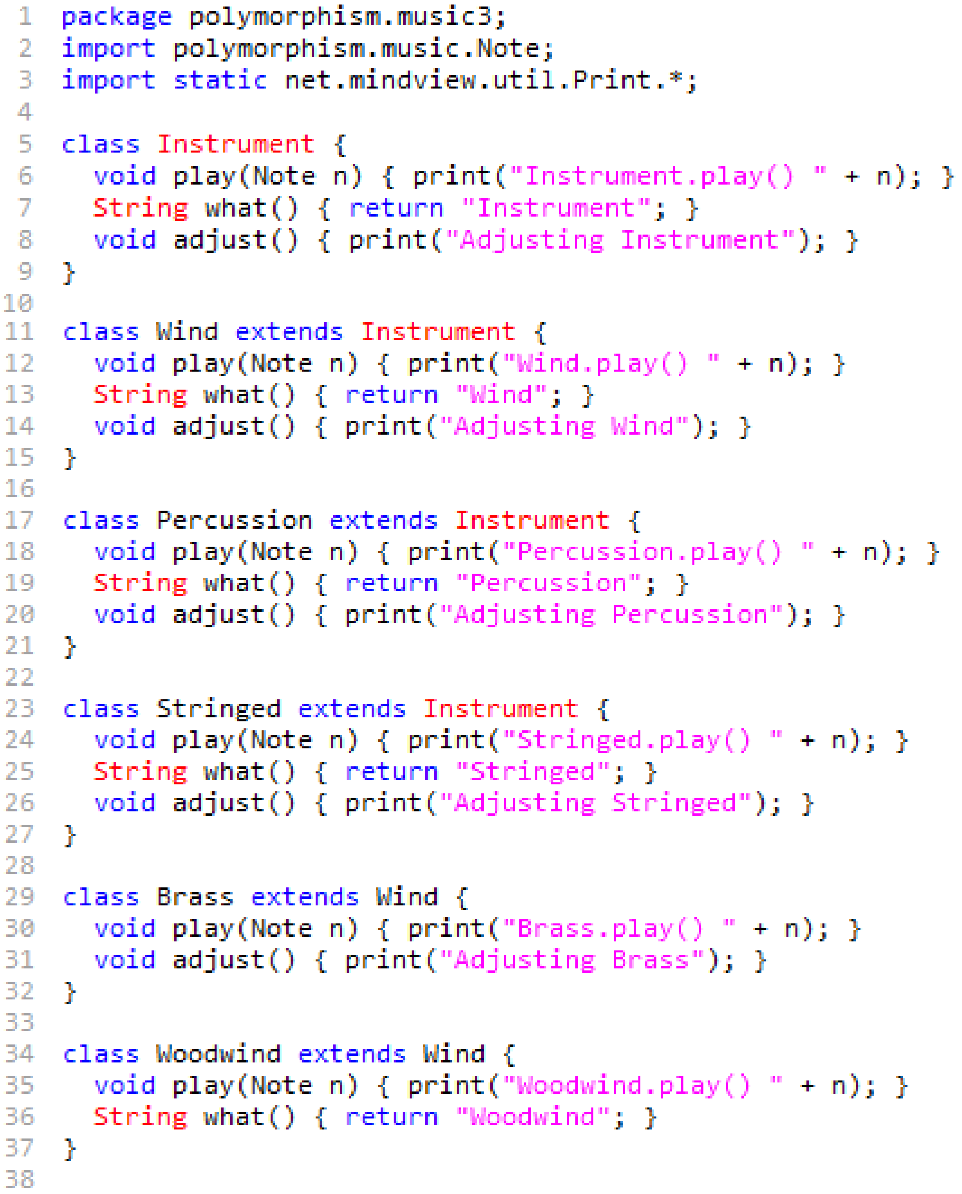
- ***Interfaces*** **and** ***abstract*** **classes provide more structured way to separate interface from implementation**

- **Such mechanisms are not that common in programming languages**
  - **C++, for example, only has indirect support for these concepts**

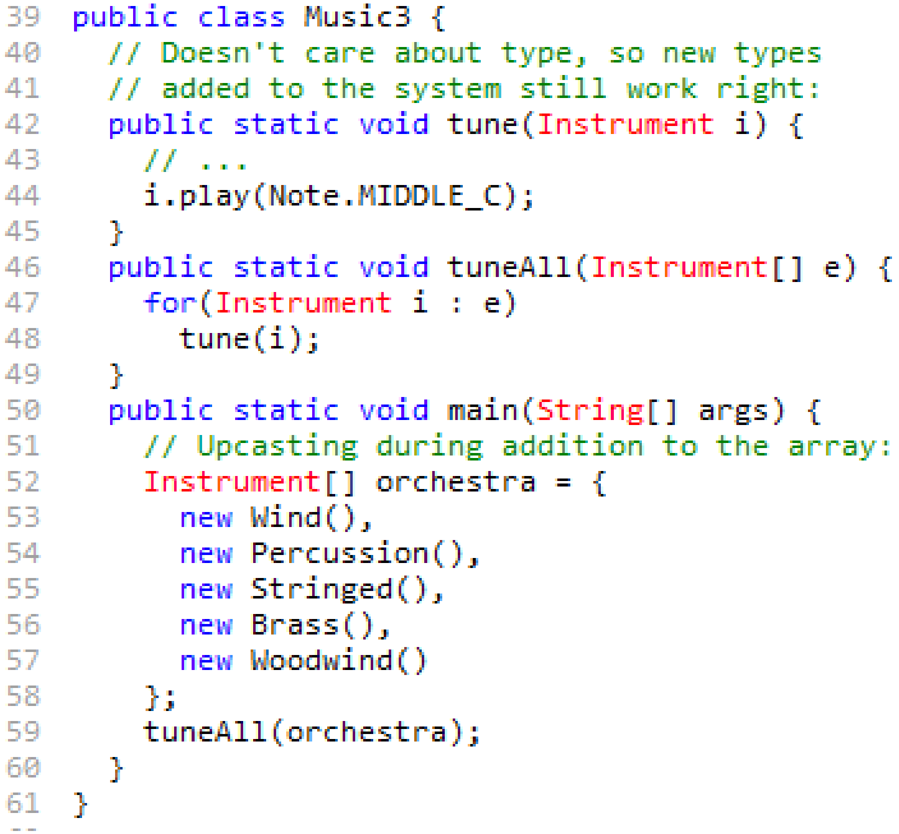
- **Java provides direct support**

- **First, we'll look at the *abstract* class**
  - **A kind of midway step between an ordinary class and an interface**

Extensibility



```java
package polymorphism.music3;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

class Instrument {
  void play(Note n) { print("Instrument.play() " + n); }
  String what() { return "Instrument"; }
  void adjust() { print("Adjusting Instrument"); }
}

class Wind extends Instrument {
  void play(Note n) { print("Wind.play() " + n); }
  String what() { return "Wind"; }
  void adjust() { print("Adjusting Wind"); }
}

class Percussion extends Instrument {
  void play(Note n) { print("Percussion.play() " + n); }
  String what() { return "Percussion"; }
  void adjust() { print("Adjusting Percussion"); }
}

class Stringed extends Instrument {
  void play(Note n) { print("Stringed.play() " + n); }
  String what() { return "Stringed"; }
  void adjust() { print("Adjusting Stringed"); }
}

class Brass extends Wind {
  void play(Note n) { print("Brass.play() " + n); }
  void adjust() { print("Adjusting Brass"); }
}

class Woodwind extends Wind {
  void play(Note n) { print("Woodwind.play() " + n); }
  String what() { return "Woodwind"; }
}

```



```java
public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
}
```

Abstract classes and methods

- The methods in the base class *Instrument* were always “dummy” methods
  - The intent of *Instrument* is to create a *common interface* for all the classes derived from it

- It establishes a basic form, so that you can say what’s common for all the derived classes

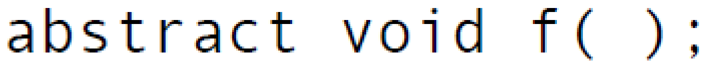
- Another way of saying this is to call Instrument an *abstract base class*, or simply an *abstract class*

- Create an *abstract* class when you want to manipulate a set of classes through its common interface

Abstract classes and methods (Cont.)

- ***Instrument*** **is meant to express only the *interface*, and not a particular *implementation***
  - **Create an *Instrument* object makes no sense**
  - **You’ll probably want to prevent the user from doing it**

- **Java provides a mechanism for doing this called the *abstract method***



```
abstract void f( );
```

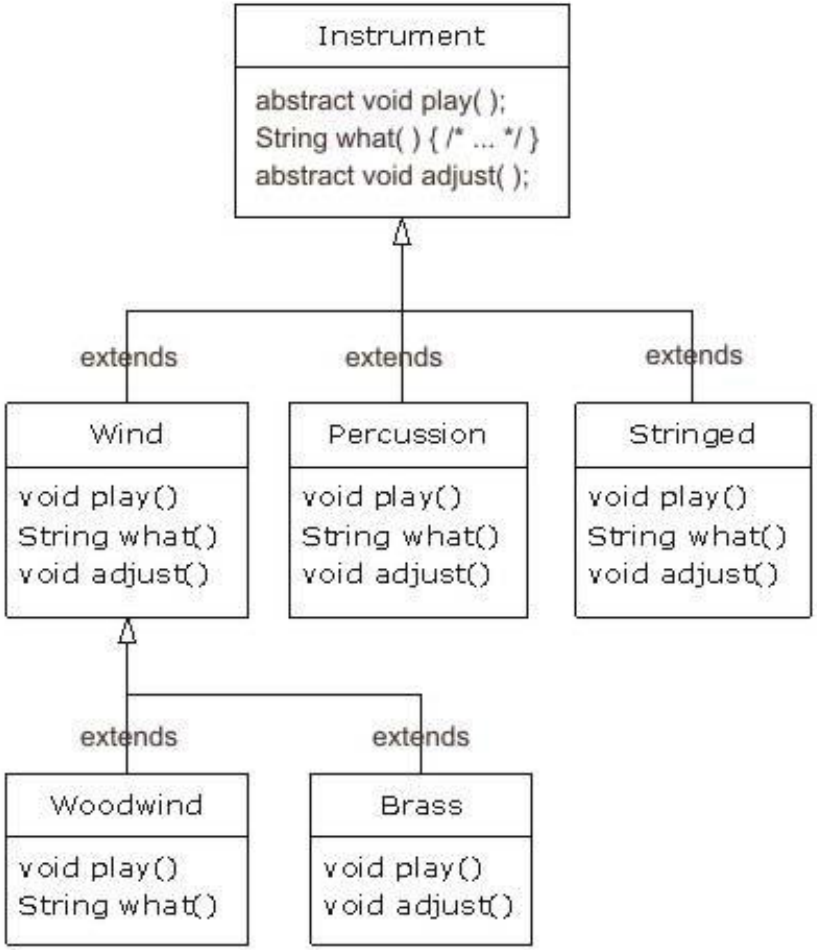
- **A class containing *abstract methods* is called an *abstract class***
  - **If a class contains one or more *abstract methods*, the class itself must be qualified as *abstract***

Abstract classes and methods (Cont.)

- **If an abstract class is incomplete, what is the compiler supposed to do when someone tries to make an object of that class?**
  - **It cannot safely create an object of an abstract class, so you get an error message from the compiler**
  - **Don’t need to worry about misusing it**
- **If you inherit from an abstract class and you want to make objects of the new type**
- ***Must* provide method definitions for all the abstract methods in the base class**
- **It’s possible to make a class *abstract* without including any *abstract* methods**
  - **prevent any instances of that class**

Abstract classes and methods (Cont.)

- **Only some of the methods will be abstract**
  - **Making a class abstract doesn't force you to make all the methods abstract**



Instrument

abstract void play( );
String what( ) { /* ... */ }
abstract void adjust( );

extends

extends

extends

Wind

void play()
String what()
void adjust()

Percussion

void play()
String what()
void adjust()

Stringed

void play()
String what()
void adjust()

extends

extends

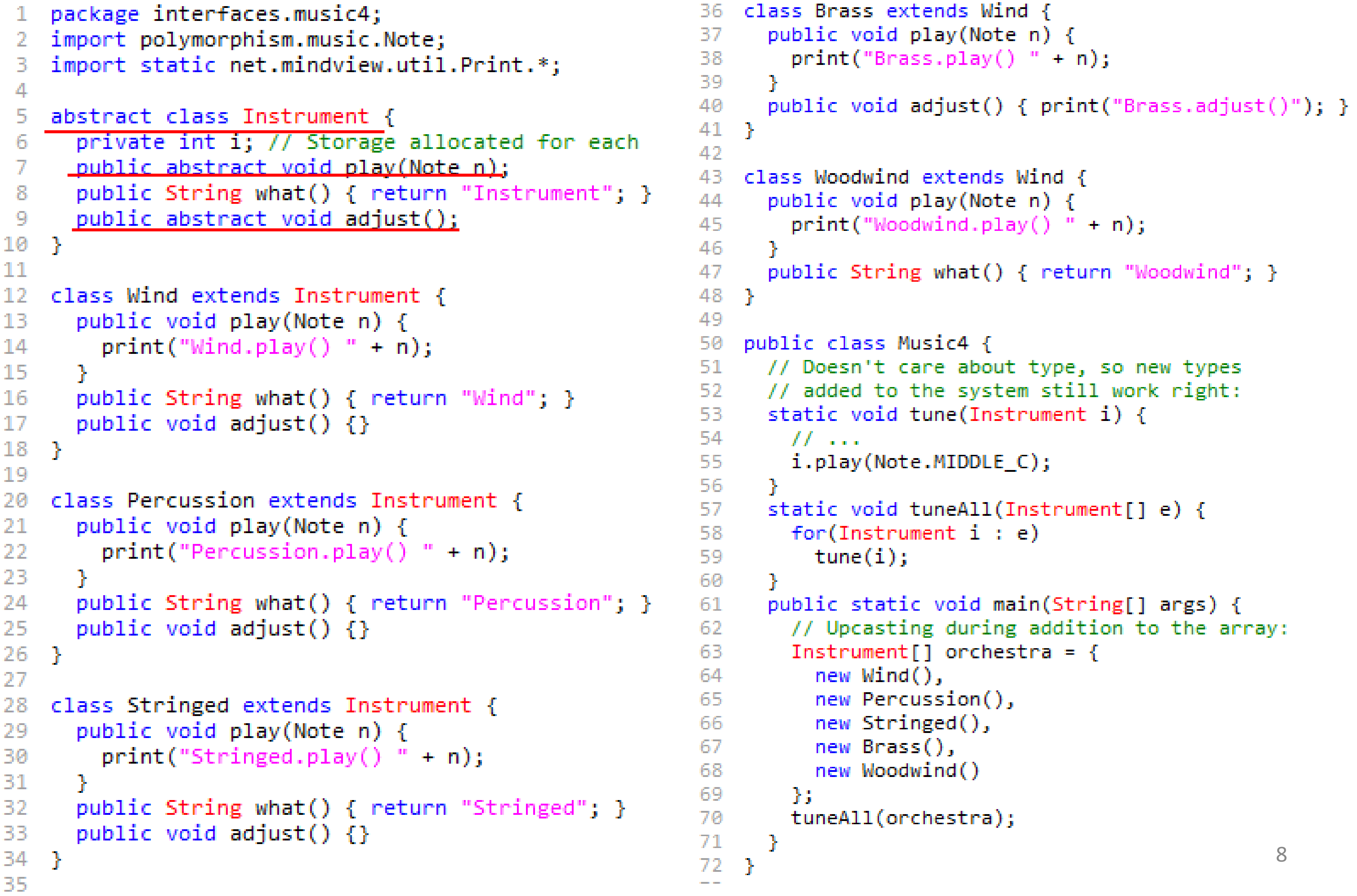
Woodwind

void play()
String what()

Brass

void play()
void adjust()

Abstract classes and methods (Cont.)



```java
package interfaces.music4;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

abstract class Instrument {
  private int i; // Storage allocated for each
  public abstract void play(Note n);
  public String what() { return "Instrument"; }
  public abstract void adjust();
}

class Wind extends Instrument {
  public void play(Note n) {
    print("Wind.play() " + n);
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument {
  public void play(Note n) {
    print("Percussion.play() " + n);
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play(Note n) {
    print("Stringed.play() " + n);
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play(Note n) {
    print("Brass.play() " + n);
  }
  public void adjust() { print("Brass.adjust()"); }
}

class Woodwind extends Wind {
  public void play(Note n) {
    print("Woodwind.play() " + n);
  }
  public String what() { return "Woodwind"; }
}

public class Music4 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
}
```

Interfaces

- The *interface* keyword takes the concept of abstractness one step further

- The *abstract* keyword allows you to create one or more undefined methods in a class
  - Provide part of the interface without providing a corresponding implementation
  - The implementation is provided by inheritors

- The *interface* keyword produces a completely *abstract* class
  - Provide no implementation at all
  - Allow the creator to determine method names, argument lists, and return types, but no method bodies
  - An interface provides only a form, but no implementation

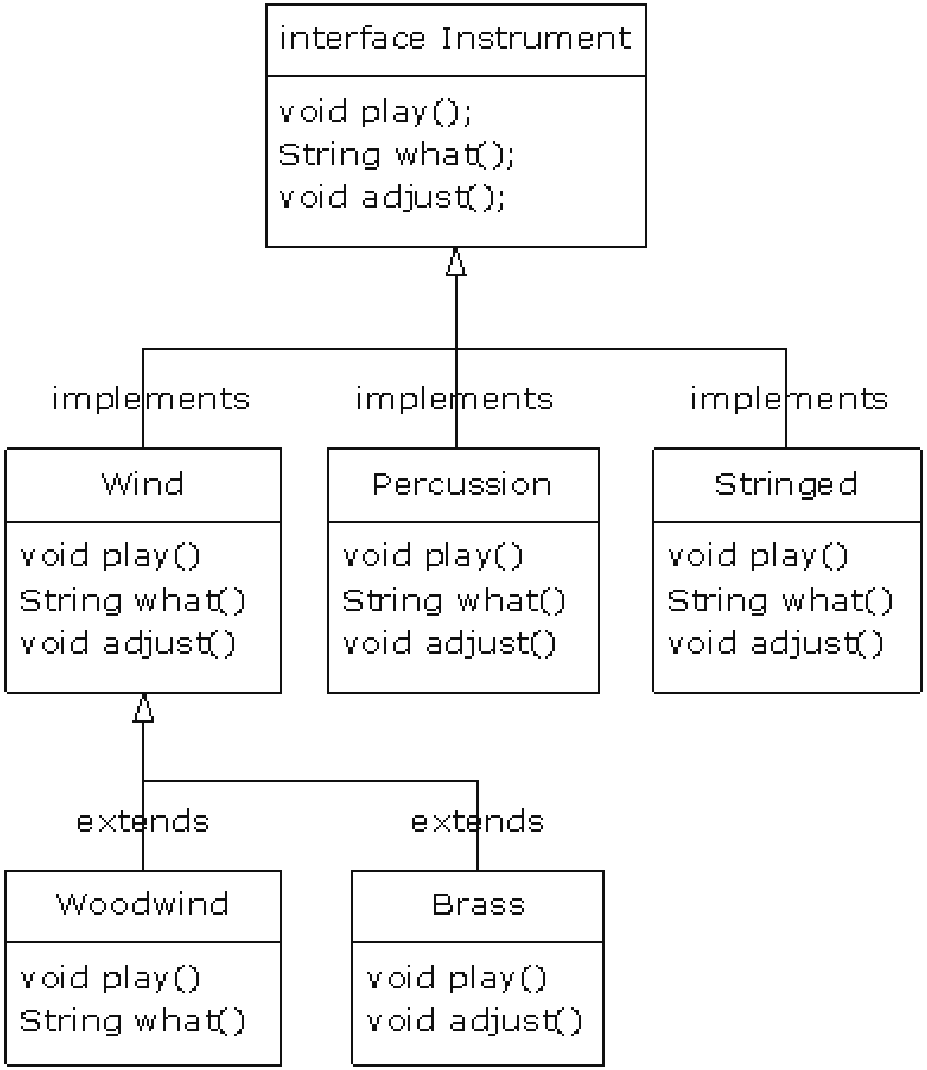
Interfaces (Cont.)

- **An interface says, "All classes that implement this particular interface will look like this. "**
  - **The interface is used to establish a "protocol" between classes**

- **Allow you to perform a variation of "multiple inheritance" by creating a class that can be upcast to more than one base type**

- **To create an interface, use the *interface* keyword instead of the *class* keyword**
  - **Can add the *public* keyword before the *interface* keyword**
  - **An interface can contain *field*s, but these are implicitly *static* and *final***

Interfaces (Cont.)

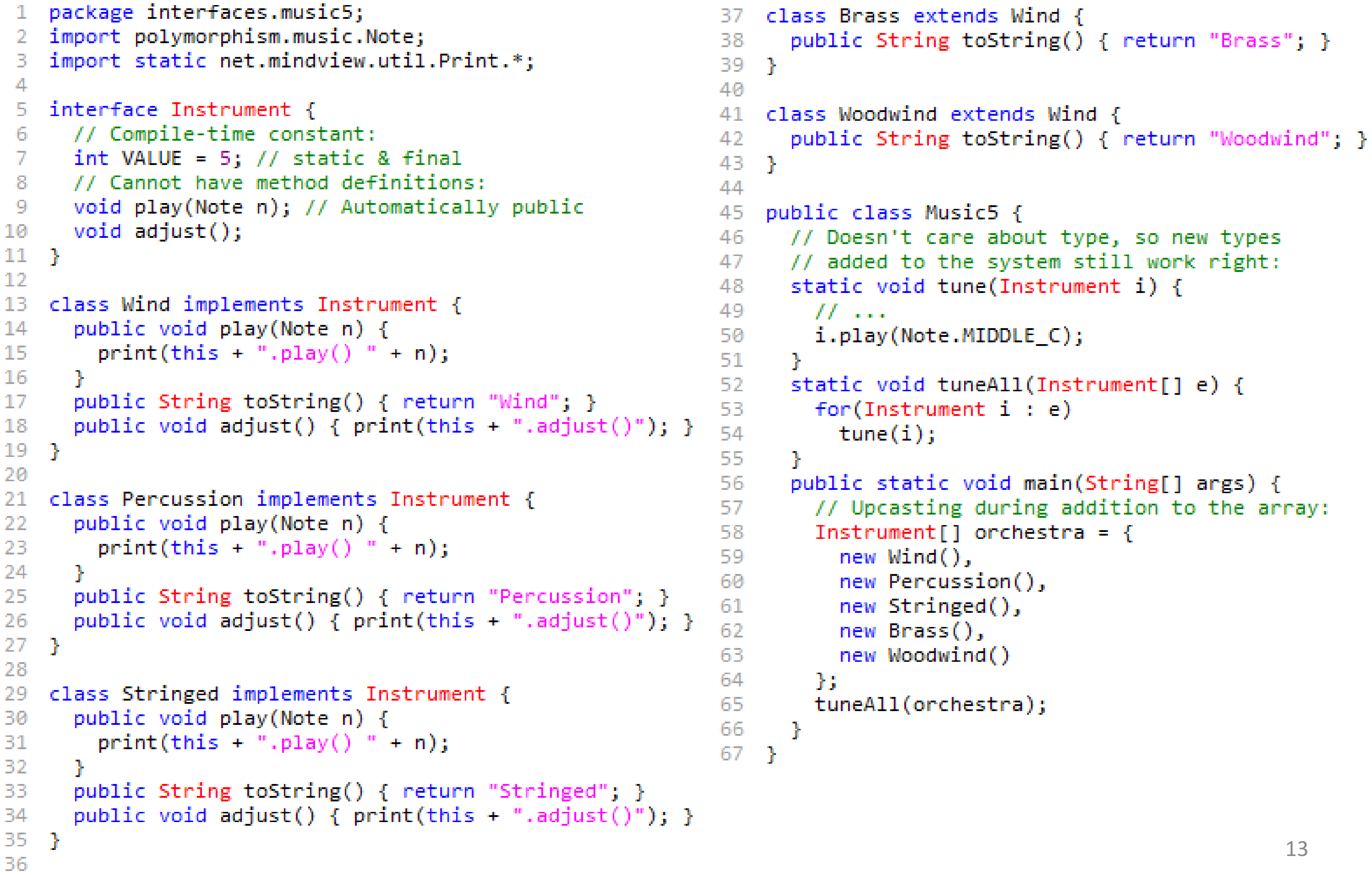


- Use the *implements* keyword says, “The interface is what it looks like, but now I’m going to say how it works”
- It looks like inheritance
- Once you implemented an interface, that implementation becomes an ordinary class that can be extended in the regular way

Interfaces (Cont.)

- **You can choose to explicitly declare the methods in an interface as *public*, but they are *public* even if you don’t say it**
  - **Otherwise, they would default to package access**
  - **Reduce the accessibility of a method during inheritance, which is not allowed by the Java compiler**

Interfaces (Cont.)



```java
package interfaces.music5;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

interface Instrument {
  // Compile-time constant:
  int VALUE = 5; // static & final
  // Cannot have method definitions:
  void play(Note n); // Automatically public
  void adjust();
}

class Wind implements Instrument {
  public void play(Note n) {
    print(this + ".play() " + n);
  }
  public String toString() { return "Wind"; }
  public void adjust() { print(this + ".adjust()"); }
}

class Percussion implements Instrument {
  public void play(Note n) {
    print(this + ".play() " + n);
  }
  public String toString() { return "Percussion"; }
  public void adjust() { print(this + ".adjust()"); }
}

class Stringed implements Instrument {
  public void play(Note n) {
    print(this + ".play() " + n);
  }
  public String toString() { return "Stringed"; }
  public void adjust() { print(this + ".adjust()"); }
}

class Brass extends Wind {
  public String toString() { return "Brass"; }
}

class Woodwind extends Wind {
  public String toString() { return "Woodwind"; }
}

public class Music5 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
}
```

Complete decoupling

- **Whenever a method works with a *class* instead of an *interface*, you are limited to using that class or its subclasses**

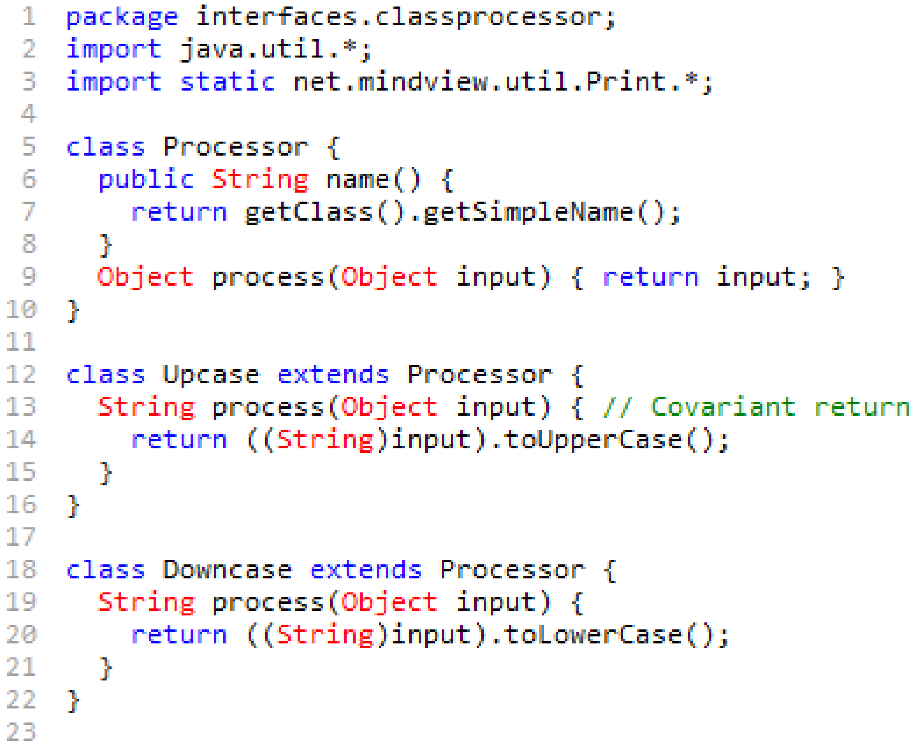
- **If you would like to apply the method to a class that isn't in that hierarchy, you're out of luck**

- **An *interface* relaxes this constraint considerably**

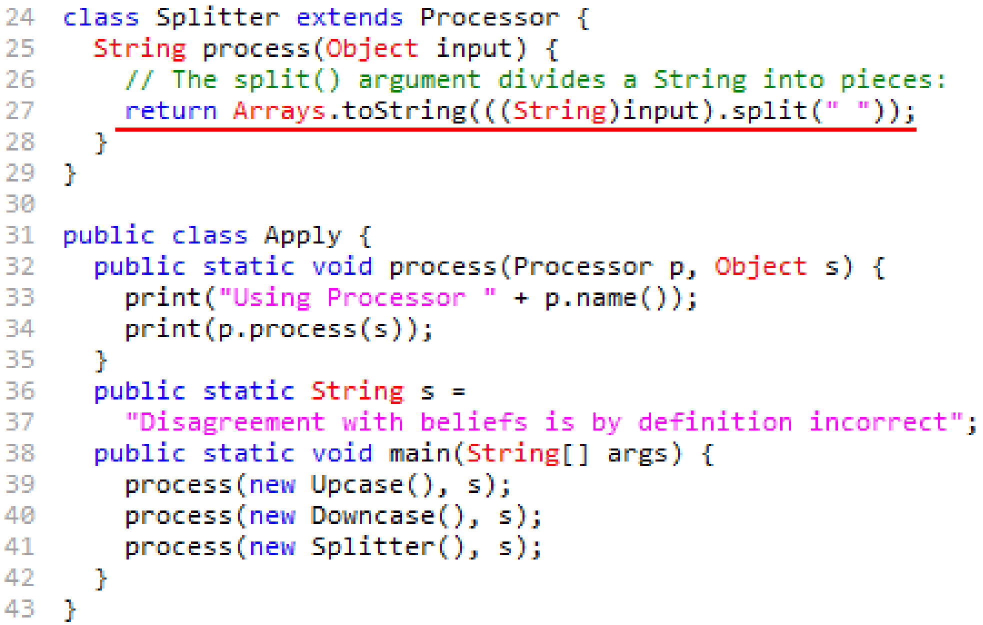
- **As a result, it allows you to write more reusable code**

Complete decoupling (Cont.)

- Creating a method that behaves differently depending on the argument object that you pass it is called the ***Strategy design pattern***
- The ***split( )*** method is a shorter way of creating an array of ***String***



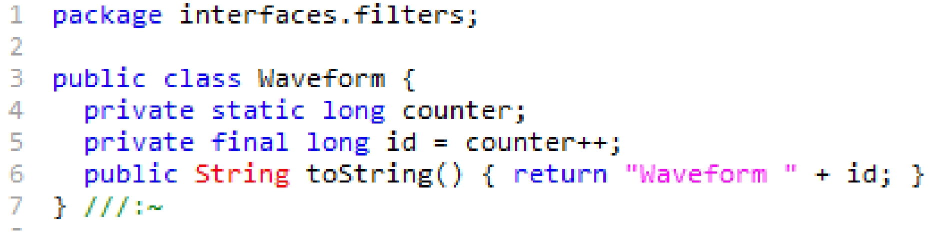
```java
package interfaces.classprocessor;
import java.util.*;
import static net.mindview.util.Print.*;

class Processor {
  public String name() {
    return getClass().getSimpleName();
  }
  Object process(Object input) { return input; }
}

class Upcase extends Processor {
  String process(Object input) { // Covariant return
    return ((String)input).toUpperCase();
  }
}

class Downcase extends Processor {
  String process(Object input) {
    return ((String)input).toLowerCase();
  }
}

```



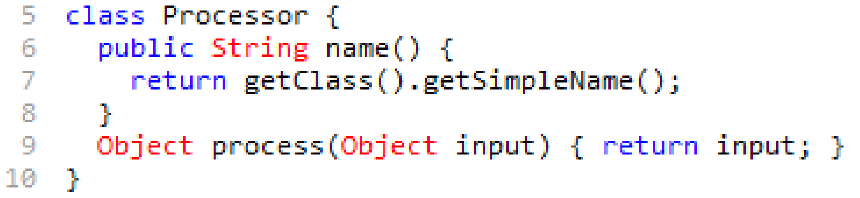
```java
class Splitter extends Processor {
  String process(Object input) {
    // The split() argument divides a String into pieces:
    return Arrays.toString(((String)input).split(" "));
  }
}

public class Apply {
  public static void process(Processor p, Object s) {
    print("Using Processor " + p.name());
    print(p.process(s));
  }
  public static String s =
    "Disagreement with beliefs is by definition incorrect";
  public static void main(String[] args) {
    process(new Upcase(), s);
    process(new Downcase(), s);
    process(new Splitter(), s);
  }
}
```

Complete decoupling (Cont.)

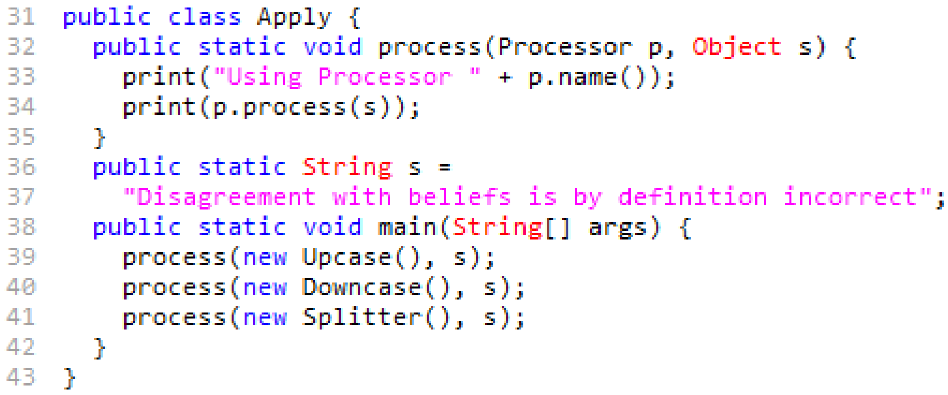
- ***Filter*** **has the same interface elements as** ***Processor*****, but because it isn't inherited from** ***Processor***
- **You can't use a** ***Filter*** **with the** ***Apply.process( )*** **method**



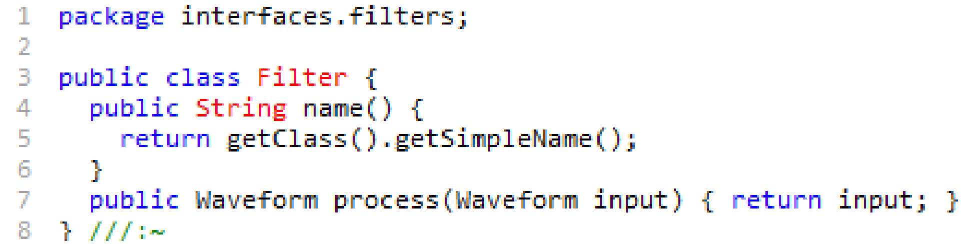
```java
package interfaces.filters;

public class Waveform {
  private static long counter;
  private final long id = counter++;
  public String toString() { return "Waveform " + id; }
} ///:~
```



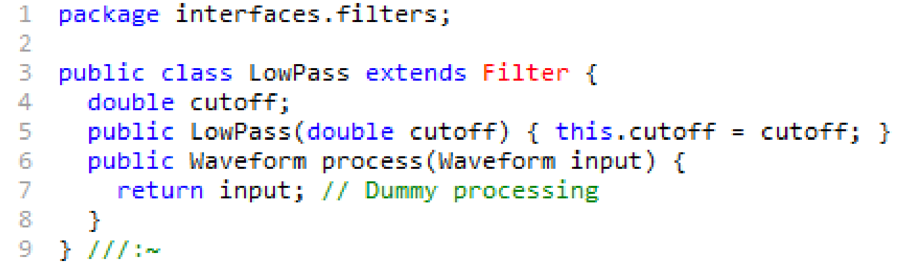
```java
class Processor {
  public String name() {
    return getClass().getSimpleName();
  }
  Object process(Object input) { return input; }
}
```



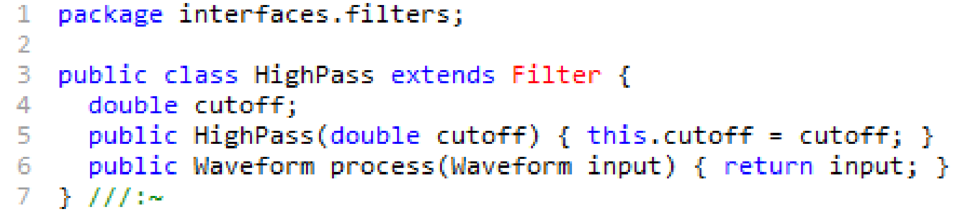
```java
public class Apply {
  public static void process(Processor p, Object s) {
    print("Using Processor " + p.name());
    print(p.process(s));
  }
  public static String s =
    "Disagreement with beliefs is by definition incorrect";
  public static void main(String[] args) {
    process(new Upcase(), s);
    process(new Downcase(), s);
    process(new Splitter(), s);
  }
}
```



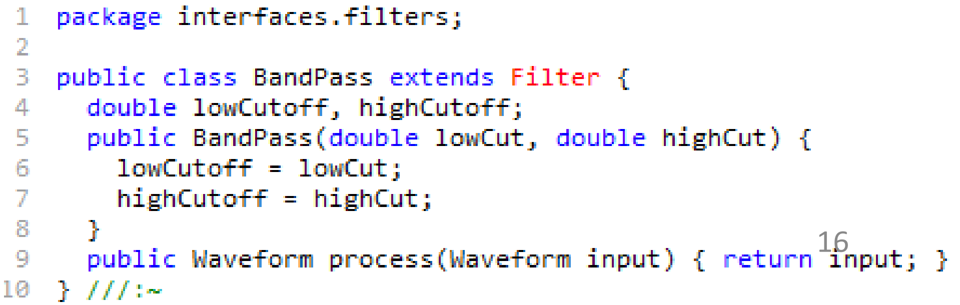
```java
package interfaces.filters;

public class Filter {
  public String name() {
    return getClass().getSimpleName();
  }
  public Waveform process(Waveform input) { return input; }
} ///:~
```



```java
package interfaces.filters;

public class LowPass extends Filter {
  double cutoff;
  public LowPass(double cutoff) { this.cutoff = cutoff; }
  public Waveform process(Waveform input) {
    return input; // Dummy processing
  }
} ///:~
```



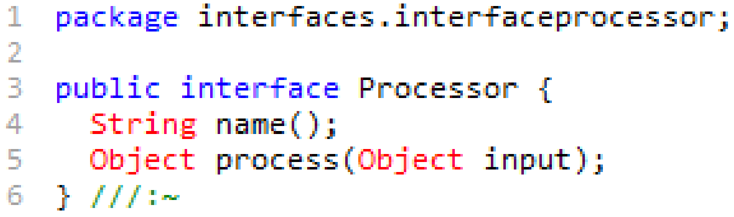
```java
package interfaces.filters;

public class HighPass extends Filter {
  double cutoff;
  public HighPass(double cutoff) { this.cutoff = cutoff; }
  public Waveform process(Waveform input) { return input; }
} ///:~
```



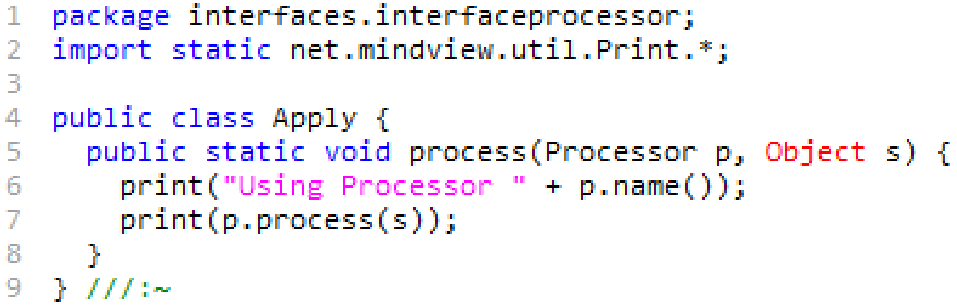
```java
package interfaces.filters;

public class BandPass extends Filter {
  double lowCutoff, highCutoff;
  public BandPass(double lowCut, double highCut) {
    lowCutoff = lowCut;
    highCutoff = highCut;
  }
  public Waveform process(Waveform input) { return input; }
} ///:~
```

Complete decoupling (Cont.)

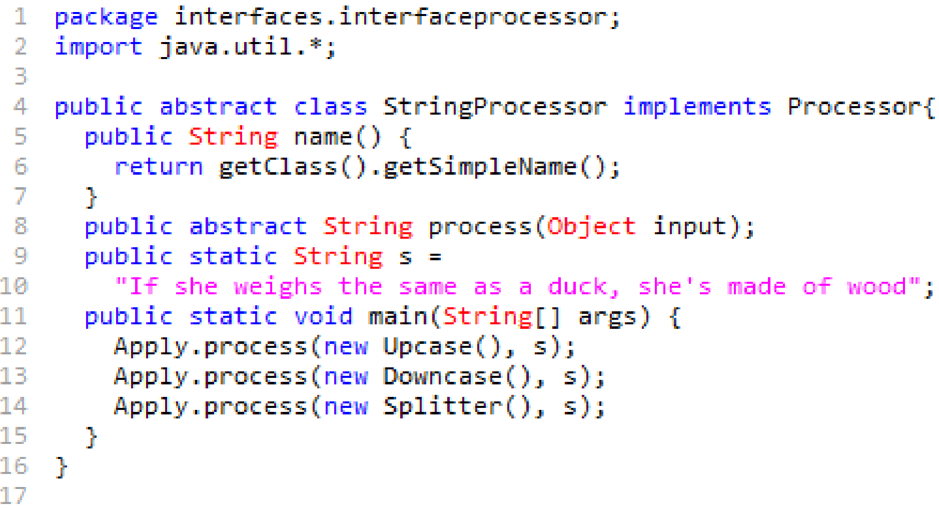
- If ***Processor*** is an interface, however, the constraints are loosened enough that you can reuse an ***Apply.process( )*** that takes that interface.



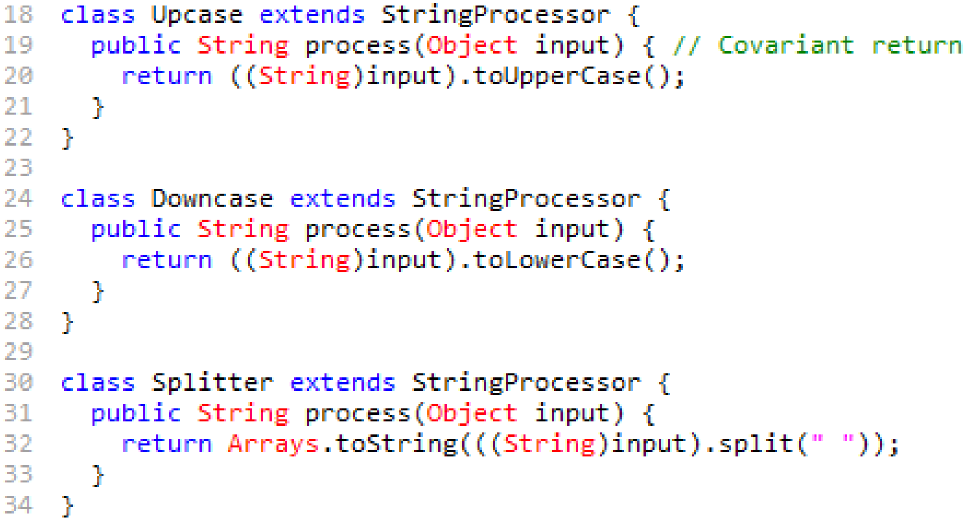
```java
package interfaces.interfaceprocessor;

public interface Processor {
  String name();
  Object process(Object input);
} ///:~
```



```java
package interfaces.interfaceprocessor;
import static net.mindview.util.Print.*;

public class Apply {
  public static void process(Processor p, Object s) {
    print("Using Processor " + p.name());
    print(p.process(s));
  }
} ///:~
```



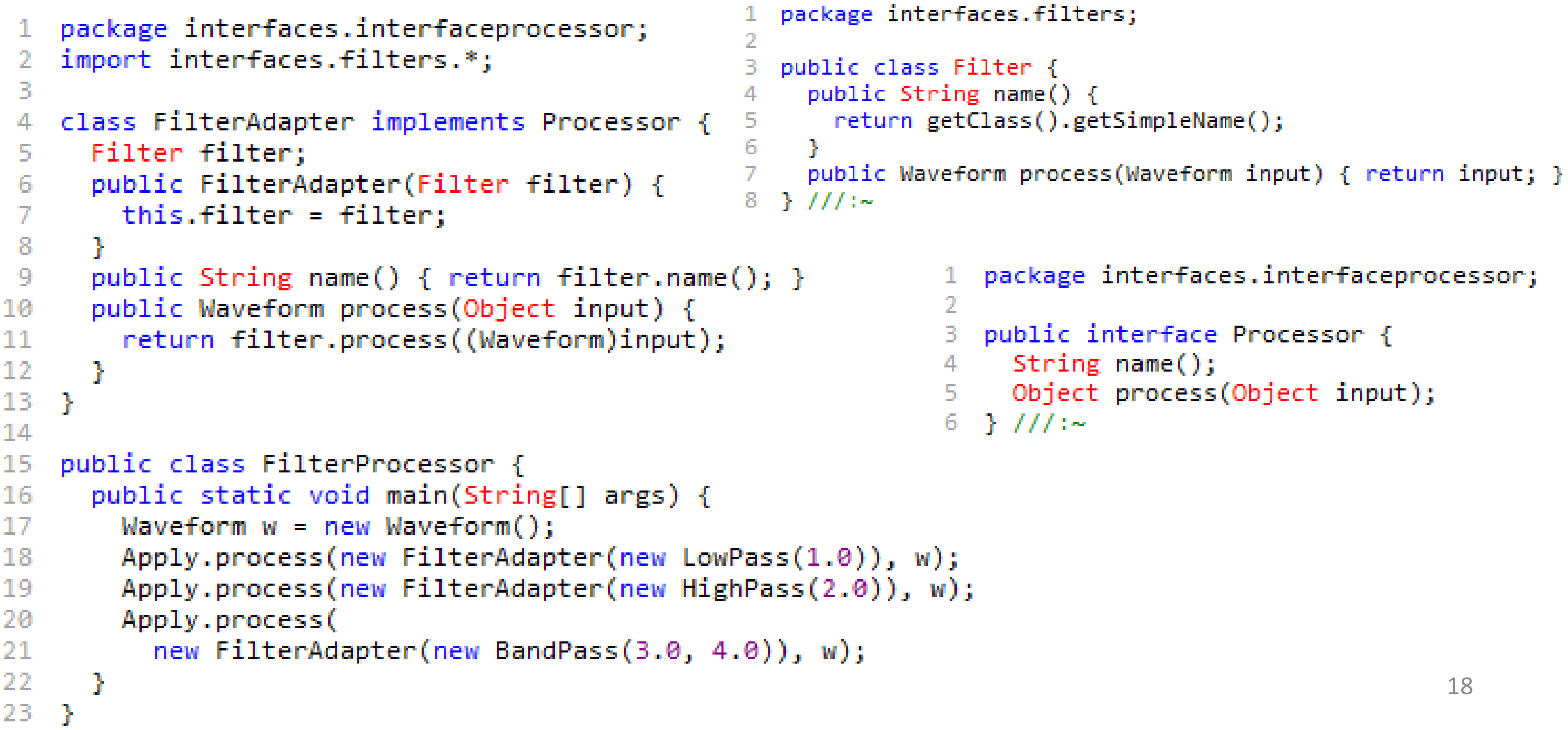
```java
package interfaces.interfaceprocessor;
import java.util.*;

public abstract class StringProcessor implements Processor{
  public String name() {
    return getClass().getSimpleName();
  }
  public abstract String process(Object input);
  public static String s =
    "If she weighs the same as a duck, she's made of wood";
  public static void main(String[] args) {
    Apply.process(new Upcase(), s);
    Apply.process(new Downcase(), s);
    Apply.process(new Splitter(), s);
  }
}
```



```java
class Upcase extends StringProcessor {
  public String process(Object input) { // Covariant return
    return ((String)input).toUpperCase();
  }
}

class Downcase extends StringProcessor {
  public String process(Object input) {
    return ((String)input).toLowerCase();
  }
}

class Splitter extends StringProcessor {
  public String process(Object input) {
    return Arrays.toString(((String)input).split(" "));
  }
}
```

Complete decoupling (Cont.)

- You are often in the situation of not being able to modify the classes that you want to use
- Use the ***Adapter design pattern***
- Write code to take the interface that you have and produce the interface that you need



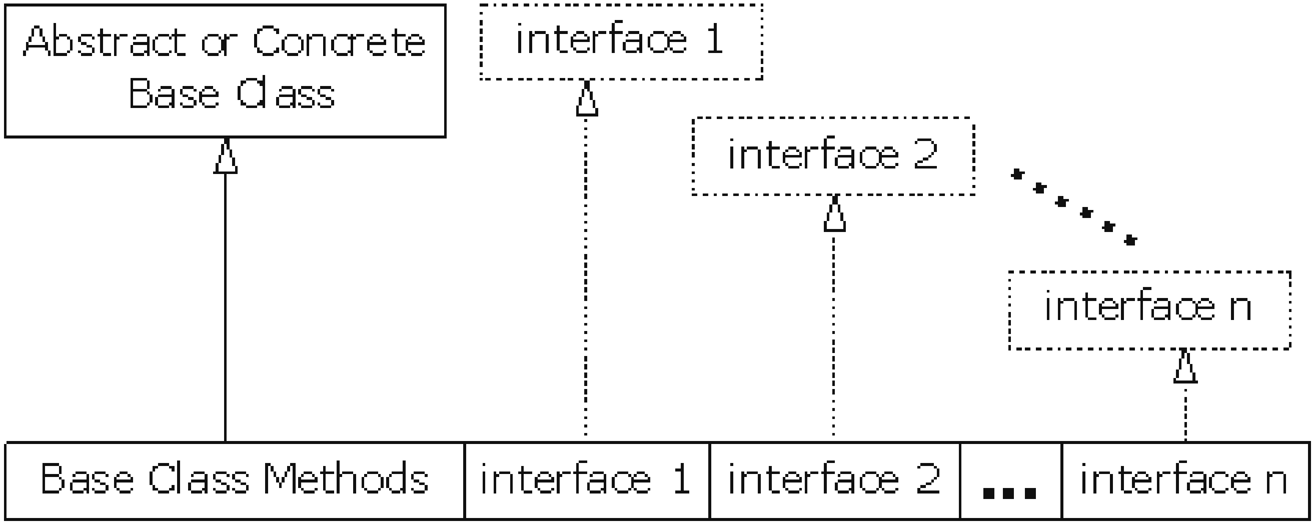
```java
package interfaces.interfaceprocessor;
import interfaces.filters.*;

class FilterAdapter implements Processor {
  Filter filter;
  public FilterAdapter(Filter filter) {
    this.filter = filter;
  }
  public String name() { return filter.name(); }
  public Waveform process(Object input) {
    return filter.process((Waveform)input);
  }
}

public class FilterProcessor {
  public static void main(String[] args) {
    Waveform w = new Waveform();
    Apply.process(new FilterAdapter(new LowPass(1.0)), w);
    Apply.process(new FilterAdapter(new HighPass(2.0)), w);
    Apply.process(
      new FilterAdapter(new BandPass(3.0, 4.0)), w);
  }
}
```

```java
package interfaces.filters;

public class Filter {
  public String name() {
    return getClass().getSimpleName();
  }
  public Waveform process(Waveform input) { return input; }
} ///:~
```

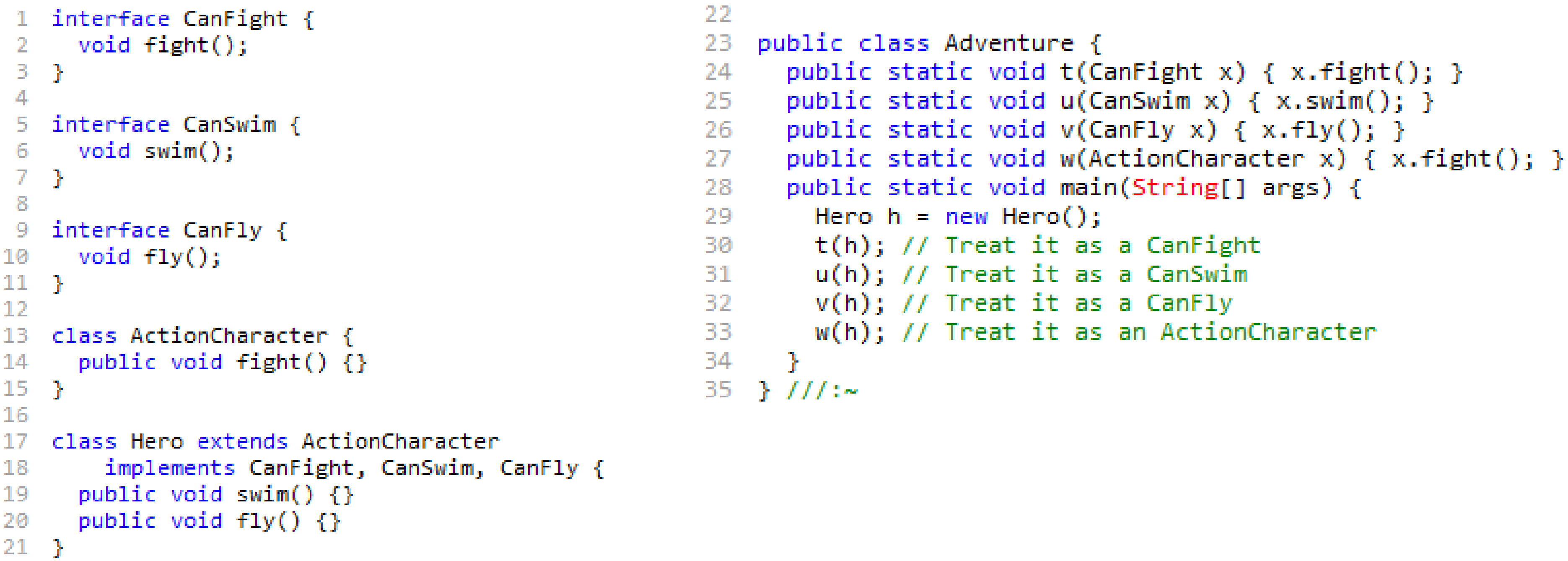
```java
package interfaces.interfaceprocessor;

public interface Processor {
  String name();
  Object process(Object input);
} ///:~
```

“Multiple inheritance” in Java

- **Many interfaces can be combined**
  - **"An x is an a and a b and a c."**
- **You can inherit from only one *base class***
- **All the rest of the base elements must be *interfaces***
- **Place all the interface names after the *implements* keyword and separate them with commas**
- **Upcast to each interface**



“Multiple inheritance” in Java

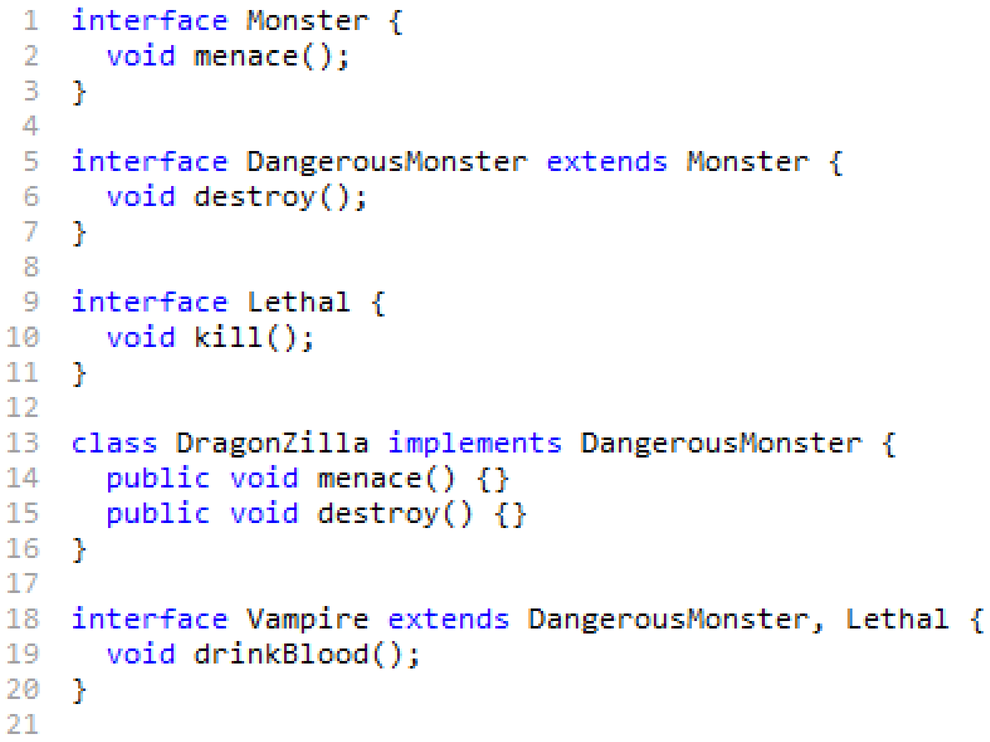


```java
interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}

public class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
} ///:~
```

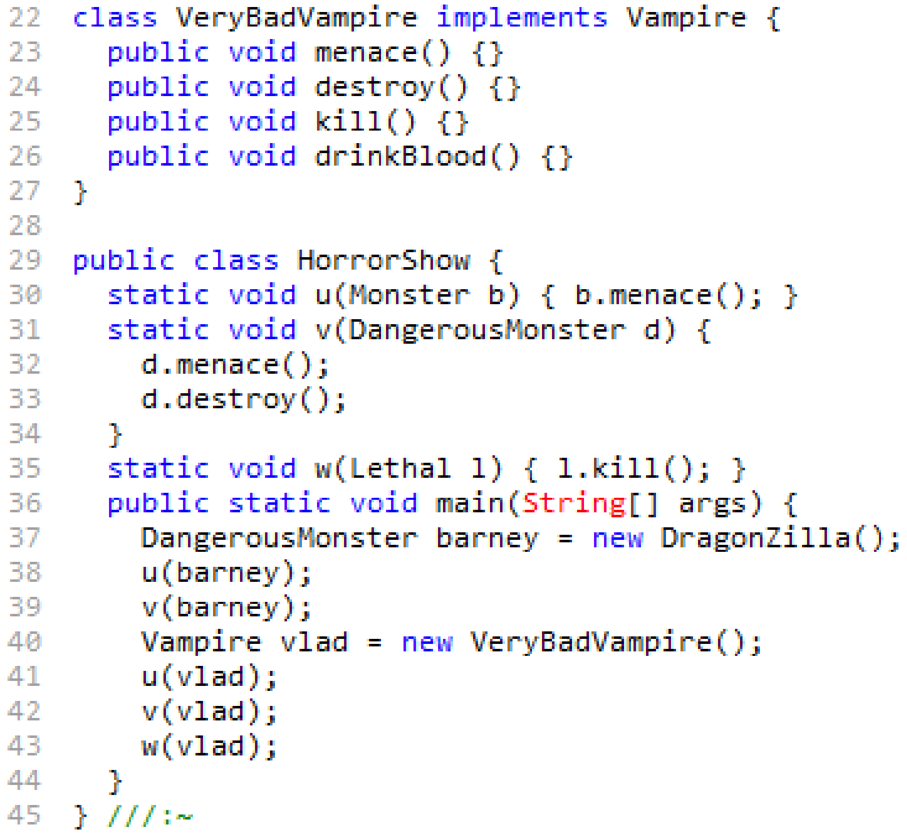
- **One of the core reasons for interfaces**
  - **Upcast to more than one base type**
  - **Prevent the client programmer from making an object of this class and to establish that it is only an interface**
- **Should you use an *interface* or an *abstract* class?**
  - **If it’s possible to create your base class without any method definitions or member variables, you should always prefer interfaces to abstract classes**

Extending an interface with inheritance

- ❑ Add new method declarations to an interface by using inheritance
- ❑ Combine several interfaces into a new interface with inheritance



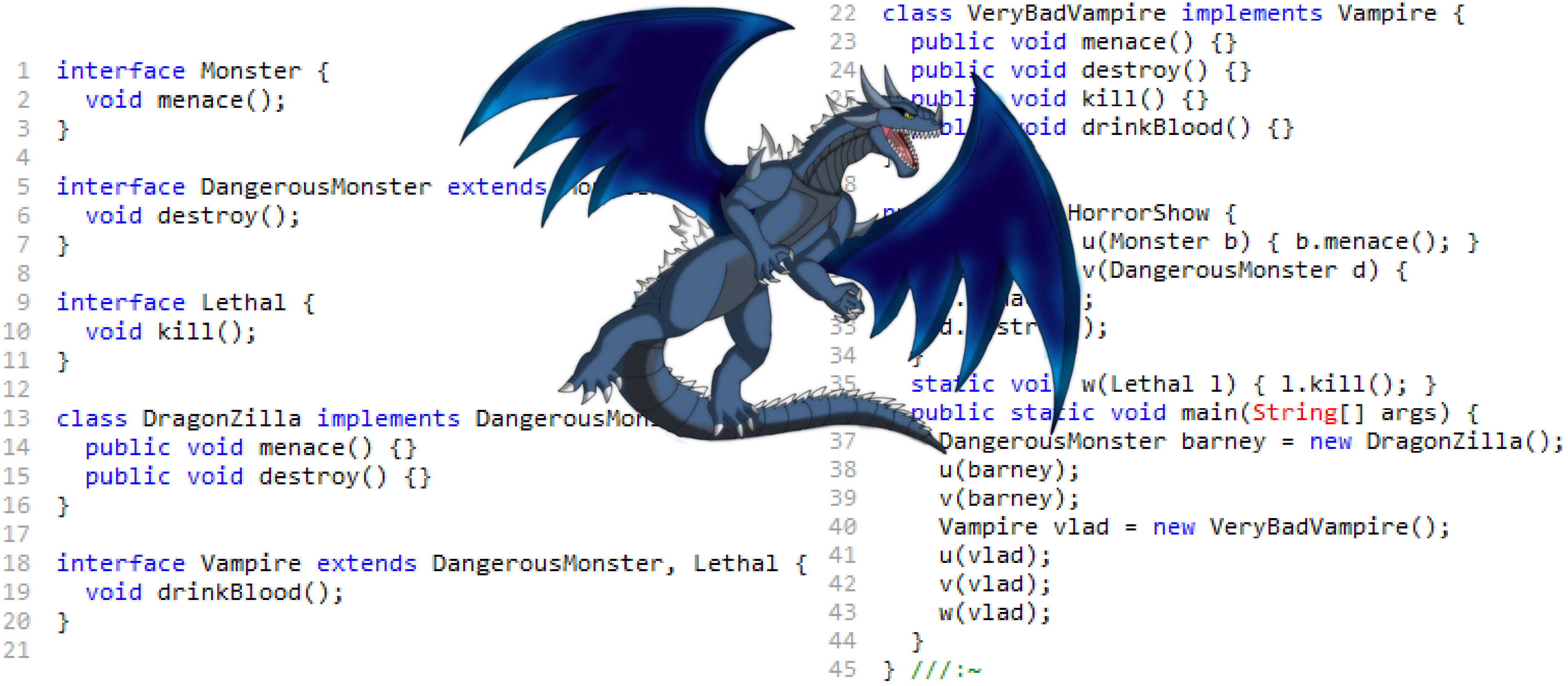
```java
interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

```



```java
class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}

public class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
} ///:~
```

Extending an interface with inheritance

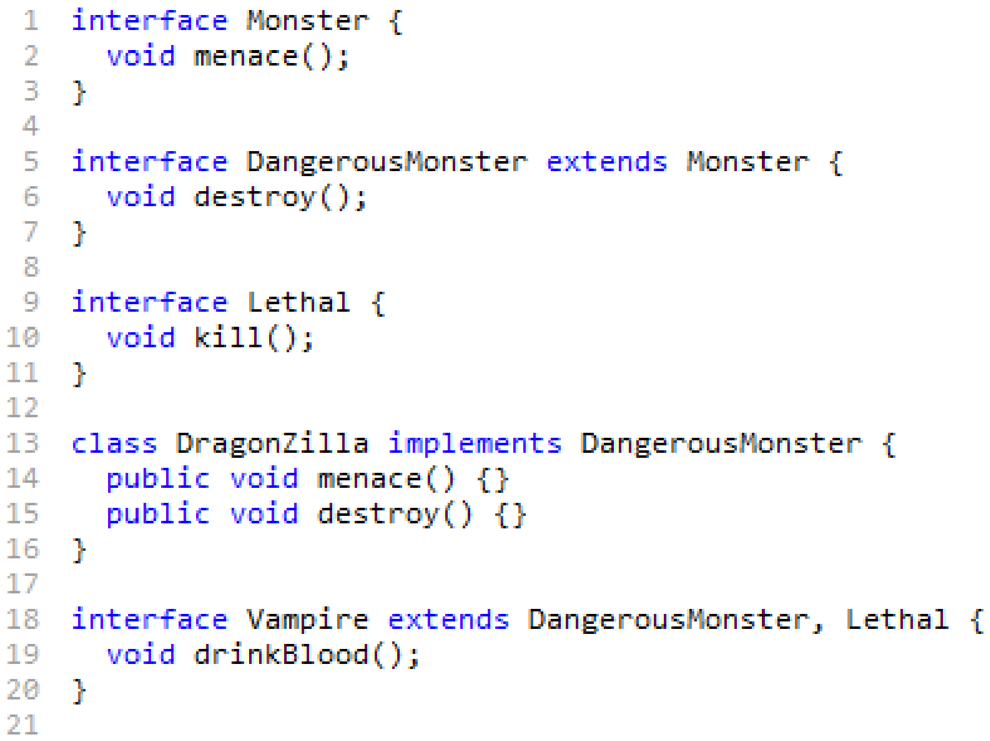
- Add new method declarations to an interface by using inheritance
- Combine several interfaces into a new interface with inheritance



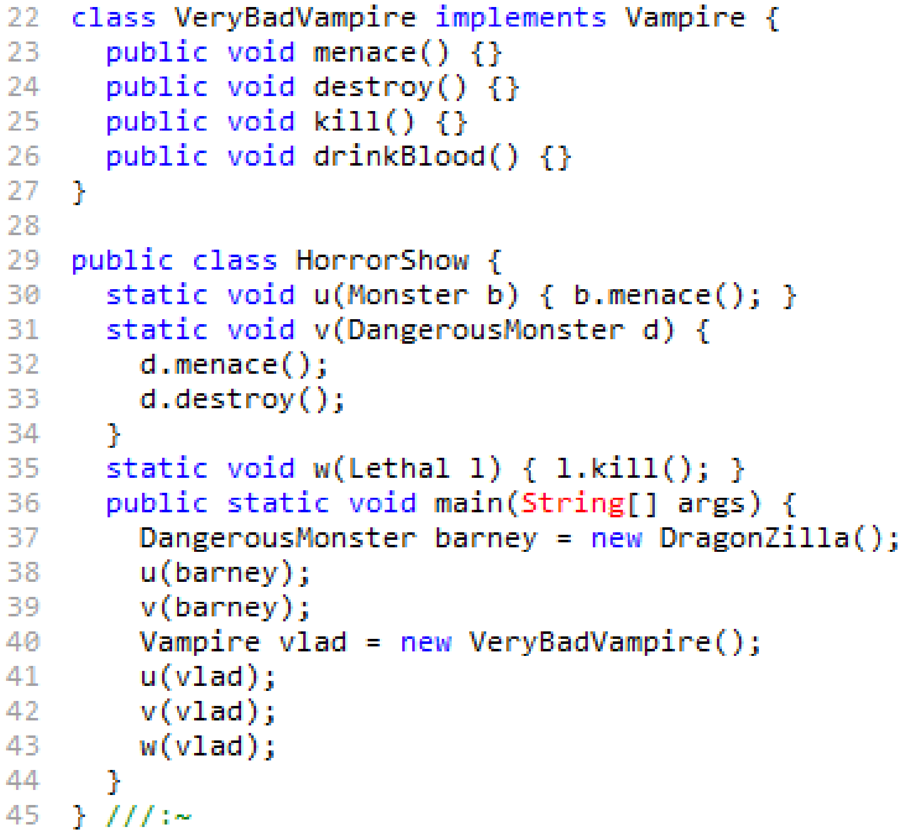
```java
interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}

public class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
} ///:~
```

Extending an interface with inheritance

- ❑ **Add new method declarations to an interface by using inheritance**
- ❑ **Combine several interfaces into a new interface with inheritance**



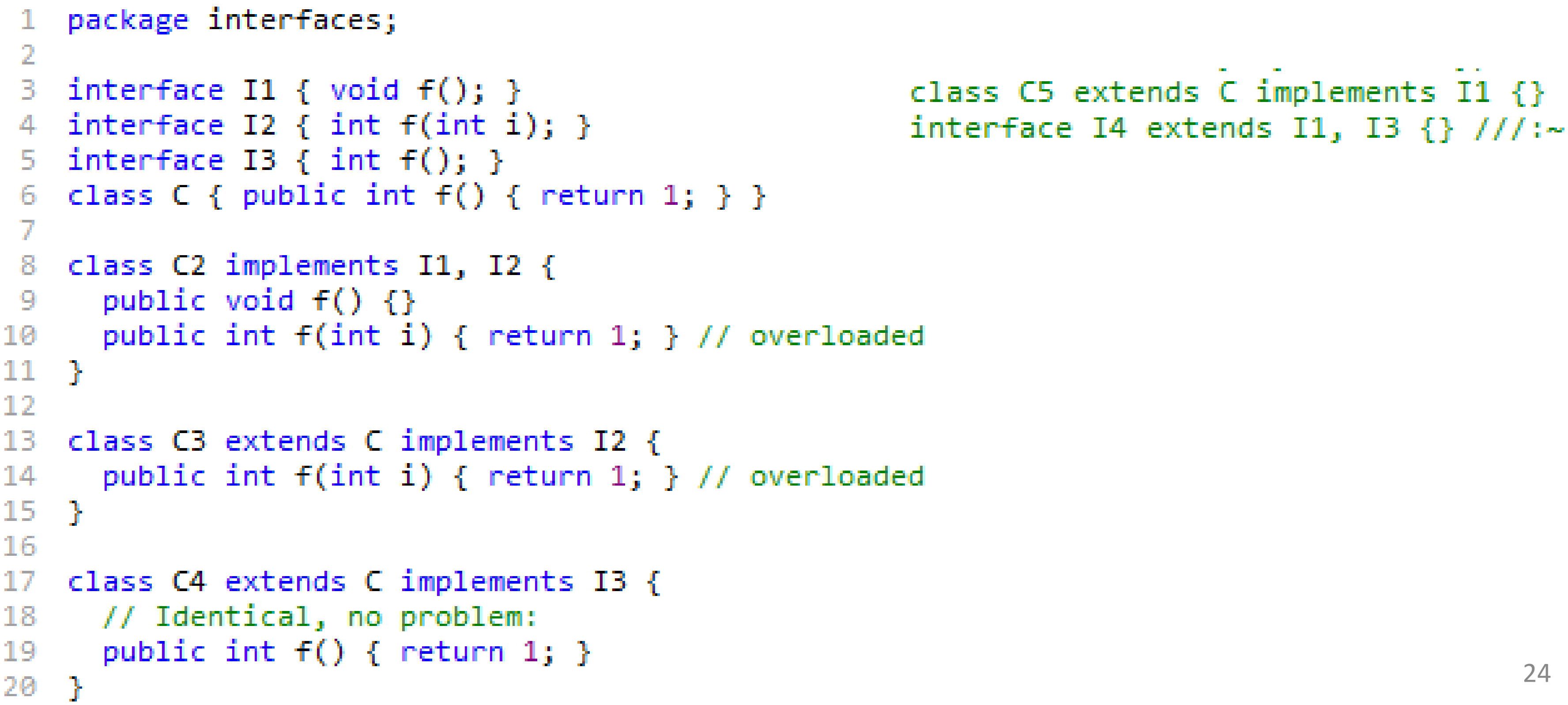
```java
interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

```



```java
class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}

public class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
} ///:~
```

Name collisions when combining Interfaces

- The difficulty occurs because overriding, implementation, and overloading get unpleasantly mixed together
- Using the same method names in different interfaces causes confusion in the readability of the code



```java
package interfaces;

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; } // overloaded
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // overloaded
}

class C4 extends C implements I3 {
  // Identical, no problem:
  public int f() { return 1; }
}
```

```java
class C5 extends C implements I1 {}
interface I4 extends I1, I3 {} ///:~
```

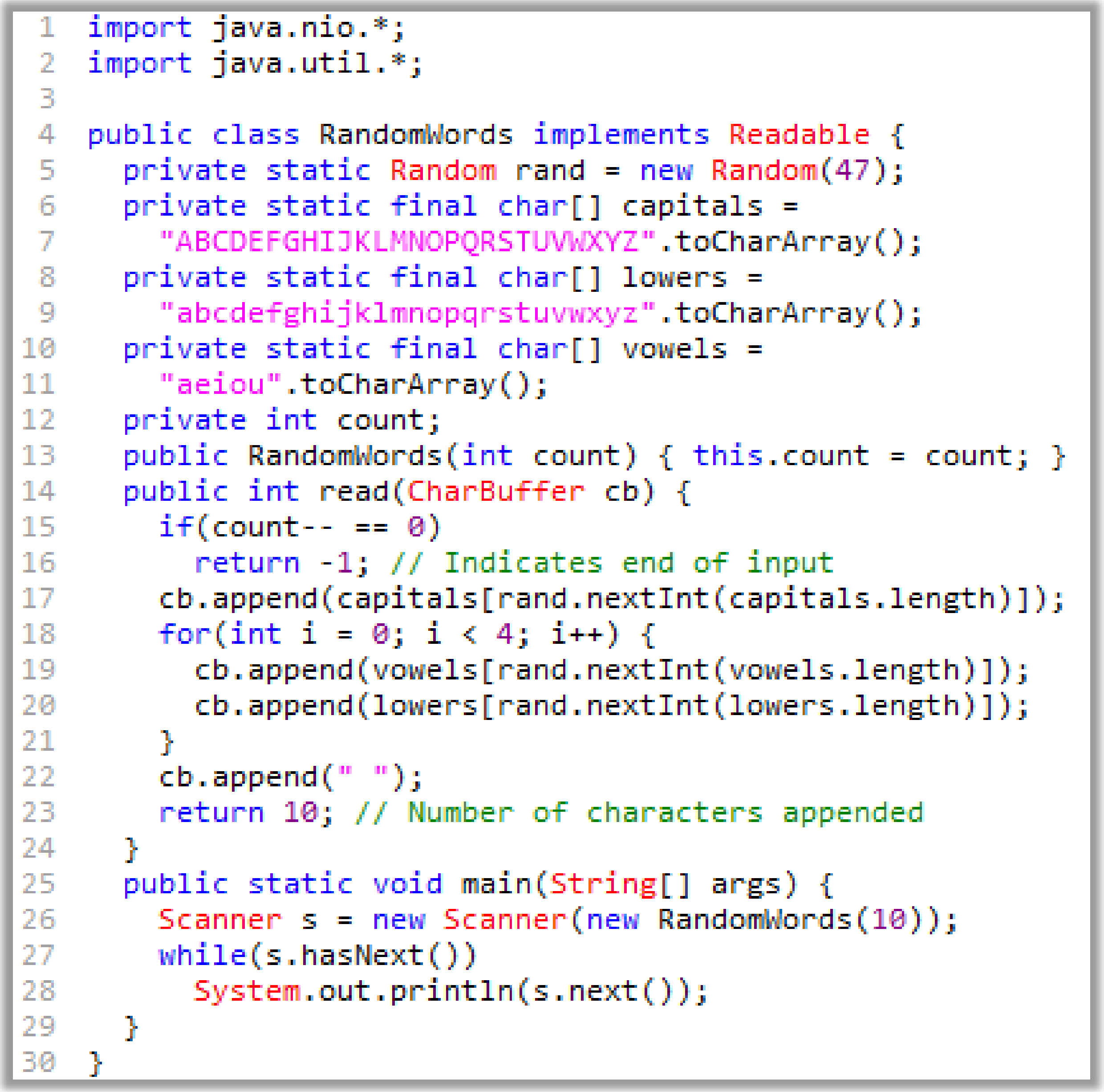
Adapting to an interface

- **A common use for interfaces is the *Strategy design pattern***
  - **Write a method that performs certain operations, and that method takes an interface that you also specify**
  - **You can use my method with any object you like, as long as your object conforms to my interface**

- **This makes your method more flexible, general and reusable**

- **For example**
  - **The constructor for the Java SE5 *Scanner* class takes a *Readable* interface**
  - ***Readable* is not an argument for any other method in the Java standard library**
  - ***Scanner* doesn’t have to constrain its argument to be a particular class**

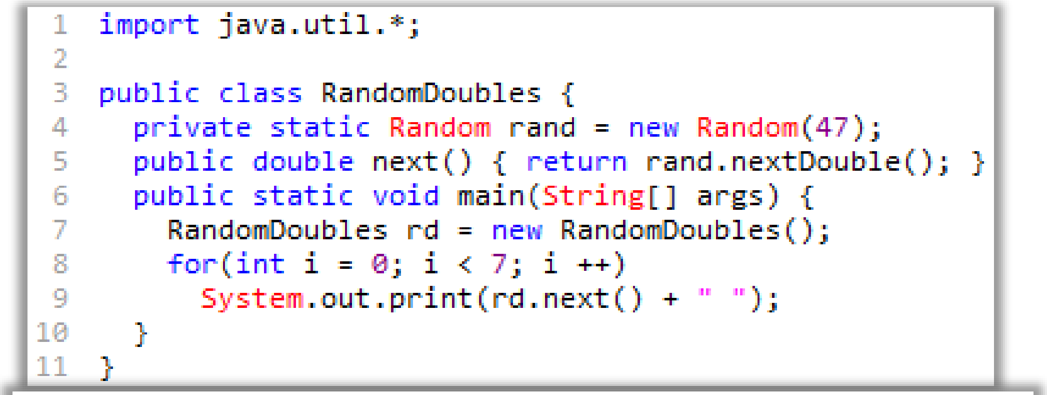
Adapting to an interface (Cont.)



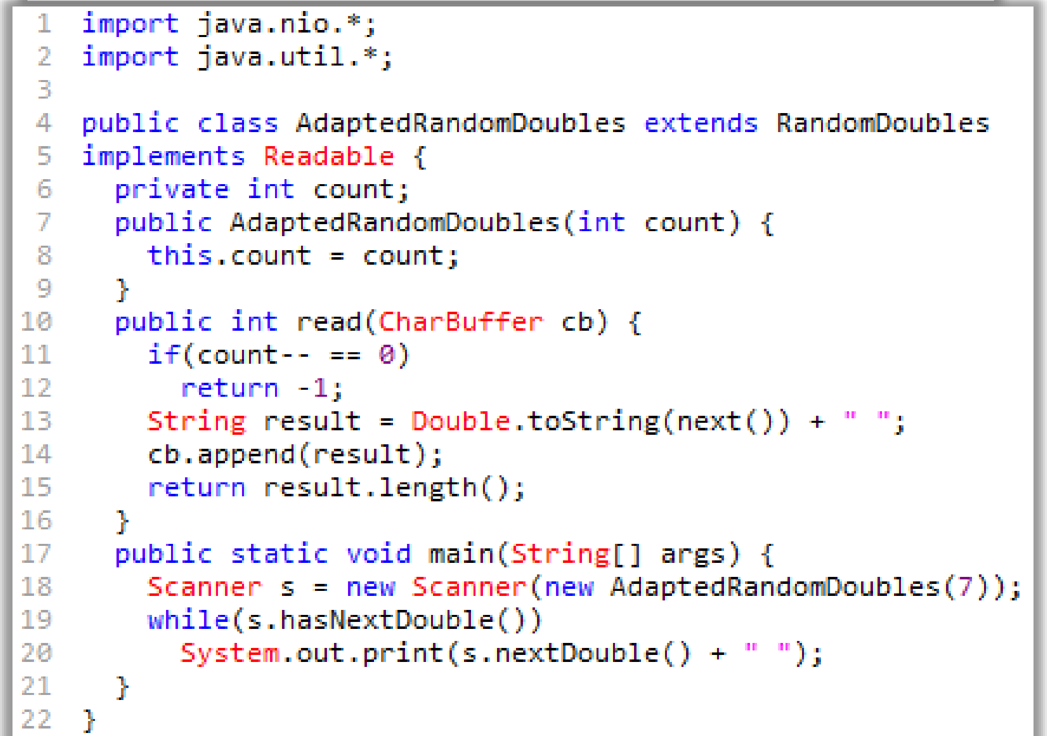
```java
import java.nio.*;
import java.util.*;

public class RandomWords implements Readable {
  private static Random rand = new Random(47);
  private static final char[] capitals =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
  private static final char[] lowers =
    "abcdefghijklmnopqrstuvwxyz".toCharArray();
  private static final char[] vowels =
    "aeiou".toCharArray();
  private int count;
  public RandomWords(int count) { this.count = count; }
  public int read(CharBuffer cb) {
    if(count-- == 0)
      return -1; // Indicates end of input
    cb.append(capitals[rand.nextInt(capitals.length)]);
    for(int i = 0; i < 4; i++) {
      cb.append(vowels[rand.nextInt(vowels.length)]);
      cb.append(lowers[rand.nextInt(lowers.length)]);
    }
    cb.append(" ");
    return 10; // Number of characters appended
  }
  public static void main(String[] args) {
    Scanner s = new Scanner(new RandomWords(10));
    while(s.hasNext())
      System.out.println(s.next());
  }
}
```

Adapting to an interface (Cont.)

- Suppose you have a class that does not already implement *Readable*—how do you make it work with *Scanner*?



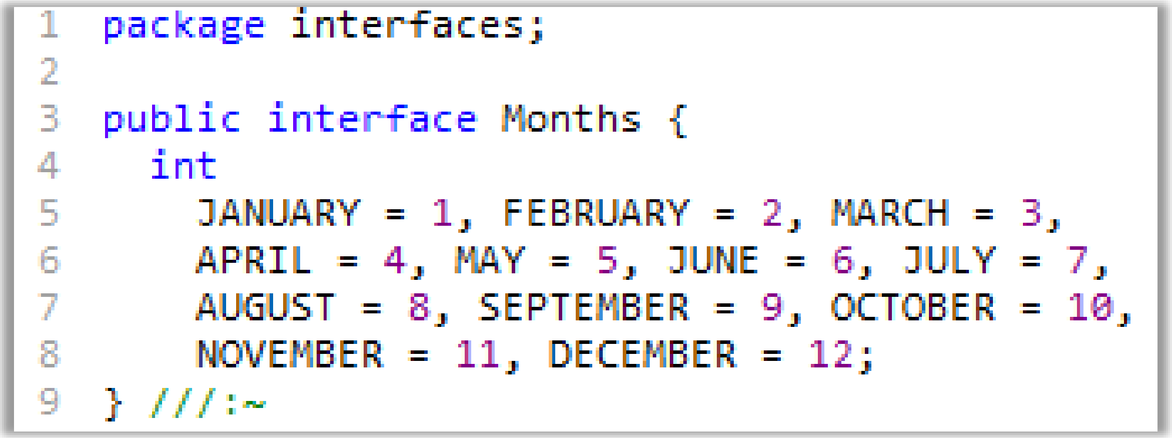
```java
import java.util.*;

public class RandomDoubles {
  private static Random rand = new Random(47);
  public double next() { return rand.nextDouble(); }
  public static void main(String[] args) {
    RandomDoubles rd = new RandomDoubles();
    for(int i = 0; i < 7; i ++)
      System.out.print(rd.next() + " ");
  }
}
```



```java
import java.nio.*;
import java.util.*;

public class AdaptedRandomDoubles extends RandomDoubles
implements Readable {
  private int count;
  public AdaptedRandomDoubles(int count) {
    this.count = count;
  }
  public int read(CharBuffer cb) {
    if(count-- == 0)
      return -1;
    String result = Double.toString(next()) + " ";
    cb.append(result);
    return result.length();
  }
  public static void main(String[] args) {
    Scanner s = new Scanner(new AdaptedRandomDoubles(7));
    while(s.hasNextDouble())
      System.out.print(s.nextDouble() + " ");
  }
}
```

Fields in interfaces

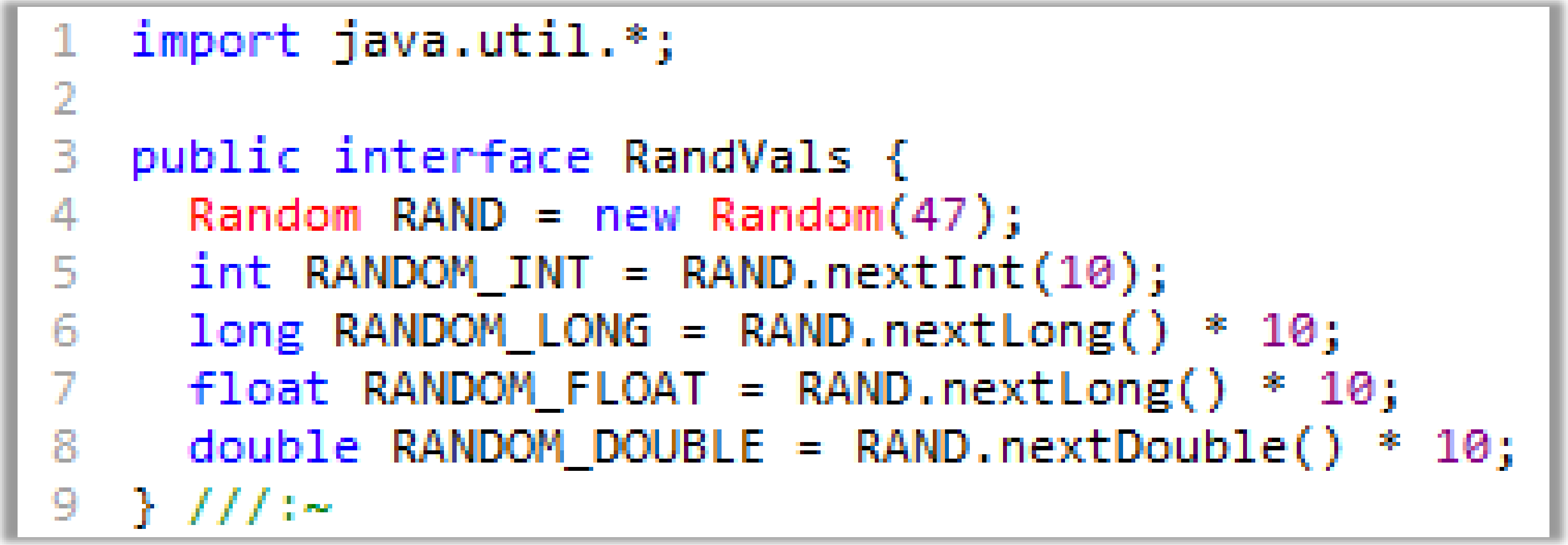
- Any fields you put into an interface are automatically *static* and *final*
  - Before Java SE5, this was the only way to produce the same effect as an enum in C or C++
- The fields in an interface are automatically *public*
- It rarely makes sense to use interfaces for constants anymore



```java
package interfaces;

public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

Initializing fields in interfaces

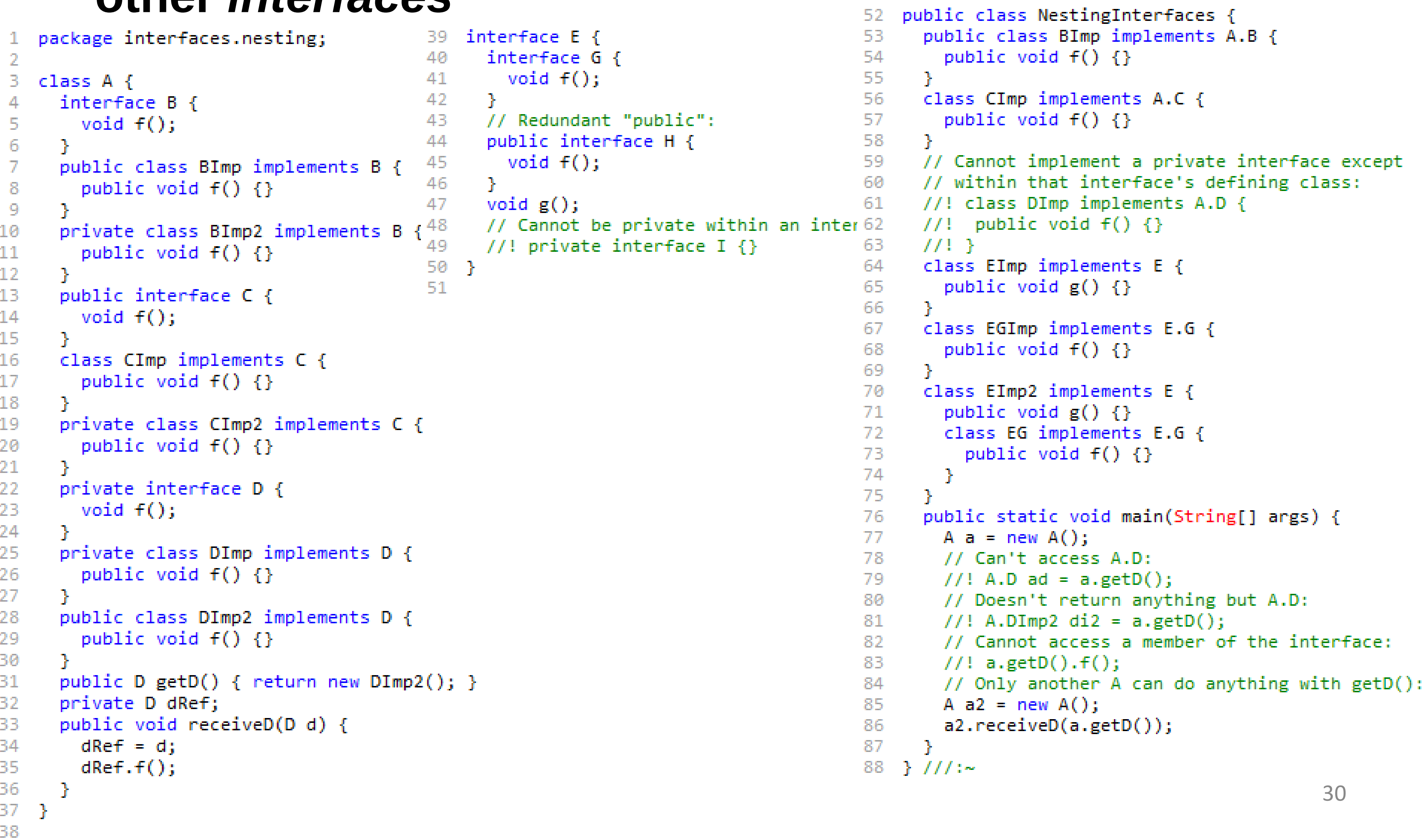
- Fields defined in interfaces cannot be "blank ***final***s"
  - Can be initialized with non-constant expressions



```java
import java.util.*;

public interface RandVals {
  Random RAND = new Random(47);
  int RANDOM_INT = RAND.nextInt(10);
  long RANDOM_LONG = RAND.nextLong() * 10;
  float RANDOM_FLOAT = RAND.nextLong() * 10;
  double RANDOM_DOUBLE = RAND.nextDouble() * 10;
} ///:~
```

Nesting interfaces

- Interfaces may be nested within *classes* and within other *interfaces*



```java
package interfaces.nesting;

class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void receiveD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
    void f();
  }
  // Redundant "public":
  public interface H {
    void f();
  }
  void g();
  // Cannot be private within an inter
  //! private interface I {}
}

public class NestingInterfaces {
  public class BImp implements A.B {
    public void f() {}
  }
  class CImp implements A.C {
    public void f() {}
  }
  // Cannot implement a private interface except
  // within that interface's defining class:
  //! class DImp implements A.D {
  //!   public void f() {}
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
    class EG implements E.G {
      public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // Can't access A.D:
    //! A.D ad = a.getD();
    // Doesn't return anything but A.D:
    //! A.DImp2 di2 = a.getD();
    // Cannot access a member of the interface:
    //! a.getD().f();
    // Only another A can do anything with getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
} ///:~
```

Interfaces and factories

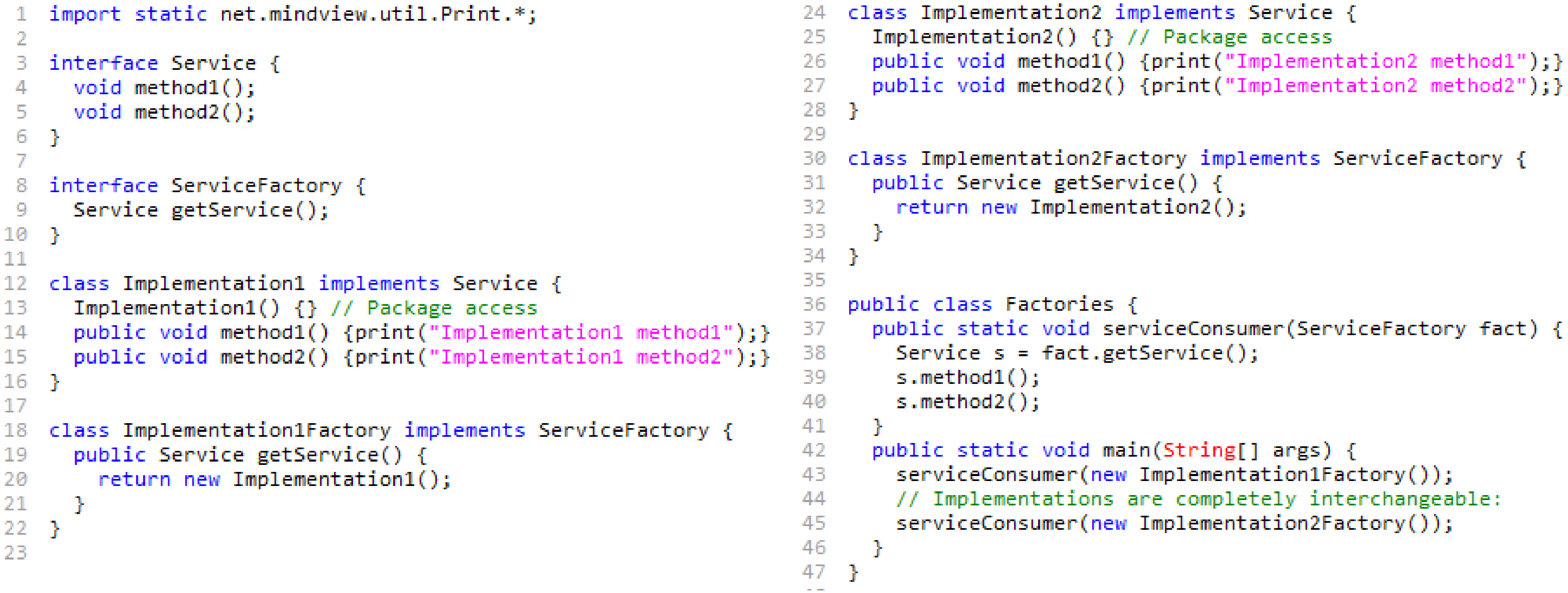
- An interface is intended to be a gateway to multiple implementations

- A typical way to produce objects that fit the interface is the ***Factory Method design pattern***

- Call a creation method on a factory object which produces an implementation of the interface
  - Your code is completely isolated from the implementation of the interface
  - Make it possible to transparently swap one implementation for another

Interfaces and factories (Cont.)

- Without the Factory Method, your code would somewhere have to specify the exact type of *Service* being created, so that it could call the appropriate constructor



```java
import static net.mindview.util.Print.*;

interface Service {
  void method1();
  void method2();
}

interface ServiceFactory {
  Service getService();
}

class Implementation1 implements Service {
  Implementation1() {} // Package access
  public void method1() {print("Implementation1 method1");}
  public void method2() {print("Implementation1 method2");}
}

class Implementation1Factory implements ServiceFactory {
  public Service getService() {
    return new Implementation1();
  }
}

class Implementation2 implements Service {
  Implementation2() {} // Package access
  public void method1() {print("Implementation2 method1");}
  public void method2() {print("Implementation2 method2");}
}

class Implementation2Factory implements ServiceFactory {
  public Service getService() {
    return new Implementation2();
  }
}

public class Factories {
  public static void serviceConsumer(ServiceFactory fact) {
    Service s = fact.getService();
    s.method1();
    s.method2();
  }
  public static void main(String[] args) {
    serviceConsumer(new Implementation1Factory());
    // Implementations are completely interchangeable:
    serviceConsumer(new Implementation2Factory());
  }
}
```

Thank you

zhenling@seu.edu.cn