LOYOLA
UNIVERSITY
CHICAGO

COMP 413: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

Projects 1 & 2

- **Android Studio / Bitbucket Issues**
  - **What issues are you still encountering?**
- **Brief Project 2 Review**

Some Thought Problems in Java

What is the output of the following Java code when `main()` is run?

```
class Changers {
    public static void changeMe1(int value) {
        value = 2*value;
    }
    public static void changeMe2(int[] value) {
        value = new int[value.length];
    }
    public static void changeMe3(int[] value) {
        value[0] = 2*value[0];
    }
    public static void main(String[] args) {
      int x = 23;
      int[] y = { 42 };    // what is this??
      int[] z = { 37 };
      changeMe1(x);
      System.out.println(x);    // Output: ___ #1
      changeMe2(y);
      System.out.println(y[0]);    // Output: _____ #2
      changeMe3(z);
      System.out.println(z[0]); // Output: _____ #3
    }
}
```

Some Thought Problems in Java

What is the output of the following Java code fragment? Briefly justify your answer.
(Recall that `x instanceof C` tells you whether the dynamic type of `x` is either `C` or a subtype of `C`. Also recall that `boolean` values print as `true` or `false`.)

```
class Coin {
   private double value;
   public void setValue(double value) { this.value = value; }
   public double getValue() { return value; }
   public Coin(double value) { setValue(value); }
   public boolean equals(Object that) {
       return (this == that) || (that instanceof Coin) &&
               (this.getValue() == ((Coin) that).getValue());
   }
}
//...
   Coin nickel1 = new Coin(0.05);
   Coin nickel2 = nickel1;
   System.out.println(nickel1.equals(nickel2) + " " + (nickel1 == nickel2));
   nickel2.setValue(0.10);
   System.out.println(nickel1.equals(nickel2) + " " + (nickel1 == nickel2));
```

Some Thought Problems in Java

What is the output of the following Java code fragment, assuming the same definition of class `Coin` from the previous subproblem? Briefly justify your answer.

```
class Coin {
    private double value;
    public void setValue(double value) { this.value = value; }
    public double getValue() { return value; }
    public Coin(double value) { setValue(value); }
    public boolean equals(Object that) {
        return (this == that) || (that instanceof Coin) &&
                (this.getValue() == ((Coin) that).getValue());
    }
}
//...
   Coin nickel1 = new Coin(0.05);
   Coin nickel2 = new Coin(0.05);
   System.out.println(nickel1.equals(nickel2) + " " + (nickel1 == nickel2));
   nickel2.setValue(0.10);
   System.out.println(nickel1.equals(nickel2) + " " + (nickel1 == nickel2));
```

Week 3 Topics

- **OO/Java review topics**
  - Inheritance and Composition
  - Interfaces
  - Abstract Classes
- More on Test-Driven Development (TDD) & JUnit

Inheritance & Polymorphism

Like mother, like daughter. *All men are mortal.*
—Common saying *Socrates is a man.*
Therefore Socrates is mortal.
—Typical Syllogism

- *Inheritance* is one of the key concepts in OO programming, and one that is needed in order to use many of the Java libraries.
- Inheritance will allow you to use an existing class to help you define new classes, or subtypes, making it easier to reuse software.
  - A related concept in object-oriented programming is *polymorphism*, which is a way to use inheritance so different kinds of objects use/provide different definitions (different actions) for the same method name.

Source: Walter Savitch, Inheritance in Java
http://www.informit.com/articles/article.aspx?p=26430

Inheritance

- **Inheritance allows you to define a very general class, and then later define more specialized classes by simply adding some new details to the original, more general class definition.** This saves work, because the more specialized class inherits all the properties of the general class and you only need to program the new features.

- For example, you might define a class for *vehicles* that has instance variables to record the vehicle's number of wheels and maximum number of occupants. You might then define a class for *automobiles*, and let the automobile class inherit all the instance variables and methods of the class for vehicles.

- You would then have to describe any added instance variables and added methods for *automobiles*, but if you use Java's inheritance mechanism, you would get the instance variables and methods from the *vehicle* class automatically.

Simple Inheritance Example

Base Class/Superclass Derived Class/Subclass

```
public class Person
{
 private String name;
 /// private instance variable - inhe
 public Person() {
  name = "No name yet.";
 } /// Person no-argument construct
 public Person(String initialName) {
  setName(initialName);
 } /// Person single-argument constr
 public void setName(String newName) {
  name = newName;
 } /// standard Person mutator - inh
 public String getName() {
  return name;
 } /// standard Person accessor - in
 public void writeOutput() {
  System.out.println("Name: " + name);
 } /// another Person method - inherited
 public boolean sameName(Person otherPerson) {
  return (getName().equalsIgnoreCase(otherPerson.getName()));
 } /// yet another Person method - inherited
 /// Note: constructors are not inherited
}
```

```
public class Student extends Person
{
 private int studentNumber; // name is inherited from Person (but it is not
      visible).
 public Student() {
  studentNumber = 0; // indicating no student number assigned yet …
 }
 public Student(String initialName, int initialStudentNumber) {
  super(initialName); // invoke the superclass constructor to initialize name.
  setStudentNumber(initialStudentNumber); // invoke our class method
 }
 public Student(String initialName) {
  this(initialName, 0); // this invokes another current class constructor.
 }
 public void reset(String newName, int newStudentNumber) {
  setName(newName); // uses base class setName() method.
  setStudentNumber(newStudentNumber);
 }
 public int getStudentNumber() {
  return studentNumber;
 }
 public void setStudentNumber(int newStudentNumber) {
  studentNumber = newStudentNumber;
 }
 public void writeOutput() { // overrides the base class writeOutput()
      method.
  System.out.println("Name: " + getName()); // use base class getName() method.
  System.out.println("Student Number : " + getStudentNumber());
 }
 public boolean equals(Student otherStudent) {
  return (sameName(otherStudent) // uses base class sameName() method
    && (getStudentNumber() == otherStudent.getStudentNumber())); // plus #.
 }
}
```

Java automatically inserts a `super();` call right here if not explicit. This is a call to the no-argument superclass or base class constructor.

Using Inheritance

```
public class InheritanceDemo
{
 public static void main(String[] args){
   Student s = new Student();
   s.setName("Warren Peace"); // setName() is inherited from the Person class.

   s.setStudentNumber(2001); // setStudentNumber() is defined in the Student class.
   s.writeOutput(); // the overridden version of writeOutput from the Student class is used.
 }
}
```

Screen Output

```
Name: Warren Peace
Student Number: 2001
```

Constraints on Inheritance

Use of private Base Class Instance Variables/Methods

Use of the *final* keyword

```
public final void specialMethod() {
    .
    .
    .
}
```

- Prevents specialMethod() from being overridden
- May be used on an entire class
  - In this case the class cannot be used as a base class for extension
  - If a class is declared **final**, then all of its methods become **final**

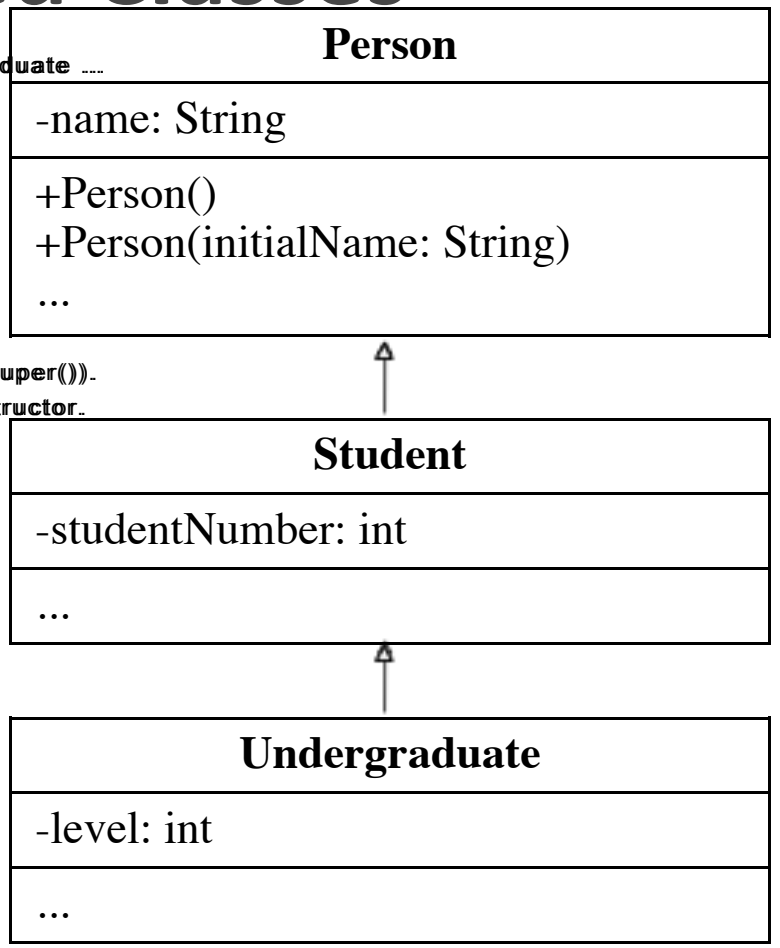
```
public void reset(String newName,
    int newStudentNumber) {
      name = newName; // ILLEGAL!
       // name is private, can't be
    seen
       // correct:
    setName(newName);
      studentNumber =
     newStudentNumber;
} // revised reset method from slide 6
```

- Since name is a private variable in the base class, derived classes like Student cannot access it directly, even though they inherit it – they must use accessors & mutators
- Similarly, private methods are not

Inheritance Example – A Chain of Multiple Derived Classes

```
public class Undergraduate extends Student // some other class could then extend Undergraduate ….
{
 private int level; // "level" of the Undergraduate student – perhaps 1 = Freshman, 2 = Sophomore, …

 public Undergraduate() {
  super(); // redundant – will be inserted by the compiler if not provided.
  level = 1;
 }
 public Undergraduate(String initialName, int initialStudentNumber, int initialLevel) {
  super(initialName, initialStudentNumber); // must be specified (otherwise get default super()).
  setLevel(initialLevel); // Note: super() or this() must occur first in every derived class constructor.
 }
 public void reset(String newName, int newStudentNumber, int newLevel) {
  // overloading of the Student reset() method (with more arguments)
  reset(newName, newStudentNumber); // this invokes the (correct) Student version of reset().
  setLevel(newLevel);
 }
 public int getLevel() {
  return level;
 }
 public void setLevel(int newLevel) {
  level = newLevel;
 }
 public void writeOutput() { // overriding of the writeOutput() method in the superclass (Student).
  super.writeOutput(); // invoke writeOutput() method in the superclass.
  System.out.println("Student Level: " + level);
 }
 public boolean equals(Undergraduate otherUndergraduate) { // overloading of Student equals() method (different argument types).
  return (super.equals(otherUndergraduate) // invoke equals() method in the superclass via an upcast.
     && (getLevel() == otherUndergraduate.getLevel())); // Note: chains of super's (i.e. multiple super's) are not allowed in method calls.
 }
}
```



UML Class Diagram

An Object of a Derived Class Has More than One Type

- Because an object of a derived class has the types of *all* of its ancestor classes (as well as its "own" type), you can assign an object of a class to a variable of any ancestor type (upcasting), but not the other way around.
- For example, if Student is a derived class of Person, and Undergraduate is a derived class of Student, then the following are all legal:

```
Person p1, p2;
p1 = new Student(); // only Person methods can be used with objects p1 & p2*
p2 = new Undergraduate(); // p1 accesses Student's writeOutput() method, but
                          // p2 accesses Undergraduate's writeOutput()*
```

- However, the following are all illegal:

```
Student s = new Person();           // ILLEGAL! (downcast)
Undergraduate ug = new Person();    // ILLEGAL! (downcast)
Undergraduate ug2 = new Student();  // ILLEGAL! (downcast)
```

*** The variable type determines what method names can be used, but the object determines which definition of each method name will be used (polymorphism).**

The Liskov Substitution Principle (LSP)

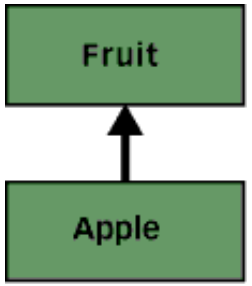
- A subtype should be indistinguishable in behavior from the type from which it is derived (via class inheritance or implementing an interface)
- Methods in a base class should be applicable to (available in) all of their derived classes, or in other words every derived class should obey the contract of the base class (i.e., have the same public interface, with the same general behavior of all methods).
- **Thus any object of a derived class is substitutable as an object of the base class.**

Inheritance vs. Composition – Two Fundamental Ways to Relate Classes

- One of the fundamental activities of an object-oriented design is establishing relationships between classes. Two fundamental ways to relate classes are inheritance and composition. Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition. Here's a simple example of inheritance:

```
class Fruit {
    //...
}

class Apple extends Fruit {
    //...
}
```



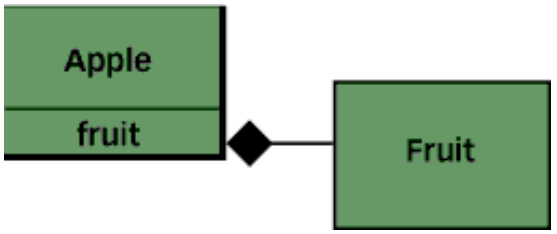
- In this example, class Apple is related to class Fruit by inheritance, because Apple extends Fruit – Fruit is the superclass, Apple the subclass.

Source: Bill Venners, JavaWorld, 11/98

Composition: Using Instance Variables that are References to Other Objects

- Here's a simple example of composition:

```
class Fruit {
    //...
}
class Apple {
    private Fruit fruit = new Fruit();
   //...
}
```



- In this example, class Apple is related to class Fruit by composition, because Apple has an instance variable that holds a reference to a Fruit object. Each newly created Apple object also creates a new Fruit object.
- In this example, Apple is the front-end class and Fruit is the back-end class. In a composition relationship, the front-end class holds a reference in one of its instance variables to a back-end class.
- Often the front-end class will forward method calls to the back-end class.

Dynamic Binding, Polymorphism, and Change

- When you establish an inheritance relationship between two classes, you get to take advantage of dynamic binding and polymorphism. Dynamic binding means the JVM will decide at runtime which method implementation to invoke based on the class of the object. Polymorphism means you can use a variable of a superclass type to hold a reference to an object whose class is the superclass or any of its subclasses.
- One of the prime benefits of dynamic binding and polymorphism is that they can help make code easier to change … in some ways.
  - As we saw, if you have a fragment of code that uses a variable of a superclass type, such as Fruit, you could later create a brand new subclass, such as Banana, and the old code fragment will work without change with instances of the new subclass.
  - If Banana overrides any of Fruit's methods that are invoked by the code fragment, dynamic binding will ensure that Banana's implementation of those methods gets executed. This will be true even though class Banana didn't exist when the code fragment was written and compiled.
- Thus, inheritance helps make code easier to change if the needed change involves adding a new subclass. This, however, is not the only kind of change you may need to make.

Changing the Superclass Interface

- In an inheritance relationship, superclasses are often said to be "fragile," because one little change to a superclass can ripple out and require changes in many other places in the application's code.
  - To be more specific, what is actually fragile about a superclass is its interface. If the superclass is well-designed, with a clean separation of interface and implementation in the object-oriented style, any changes to the superclass implementation shouldn't ripple at all.
  - Changes to the superclass interface (eg, its public method signatures), however, can ripple out and break any code that uses the superclass or any of its subclasses. What's more, a change in the superclass interface can break the code that defines any of its subclasses.

- Let’s look at an example: if you change the return type of a public method in class Fruit (a part of Fruit's interface), you can break the code that invokes that method on any reference of type Fruit or any subclass of Fruit.
  - In addition, you will break any code that defines a subclass of Fruit that overrides the method. Such subclasses won't compile until you change the return type of the overridden method to match the changed method in superclass Fruit.

Inheriting a Superclass Method and Interface

```
class Fruit {
    // Return int number of pieces of peel that resulted from the peeling activity.
    public int peel() {
        System.out.println("Peeling is appealing."); return 1;
    }
}

  class Apple extends Fruit { // empty class Apple simply reuses Fruit's peel() method
}
class Example1 {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

- When you run the Example1 application, it will print out "Peeling is appealing."
- If you decide in the future to change the return value of peel() to new type Peel, you will break the code for Example1. Your change to the Fruit superclass will break Example1's code even though Example1 uses Apple directly and never explicitly mentions Fruit.

Revising a Superclass Interface

- If Fruit is changed so that the peel() method returns type Peel, the Apple class still compiles and works fine, but the old implementation of Example1 is broken and won't compile:

```
class Peel {                          class Apple extends Fruit {
    private int peelCount;                    }
    public Peel(int peelCount) {
       this.peelCount = peelCount;       class Example1 { // this is now broken!
    }                              public static void main(String[] args) {
    public int getPeelCount() {          Apple apple = new Apple();
       return peelCount;             int pieces = apple.peel(); // wrong
    }                              }
    //...                 }
}
class Fruit {
    // Return a Peel object that
    // results from the peeling activity.
    public Peel peel() {
       System.out.println(
       "Peeling is appealing.");
       return new Peel(1);
    }
}
```

Reusing a Class Method via Composition Instead of Inheritance

- In this example, Apple holds a reference to a Fruit instance and defines its own peel() method that simply invokes peel() on the Fruit instance; Apple will be the front-end class, and the former superclass Fruit is the back-end class. The explicit invocation of the composed object's peel() method is sometimes called *forwarding* or *delegating* the method call to the back-end object.

```
class Fruit {
    // Return int number of pieces of peel that resulted from the peeling activity.
   public int peel() {
       System.out.println("Peeling is appealing."); return 1;
   }
}
  class Apple { // front-end class; no longer a subclass of Fruit
    private Fruit fruit = new Fruit(); // back-end class
    public int peel() { // Apple's implementation of the peel() method
        return fruit.peel(); // explicitly invokes the back-end class' peel() method (forwarding)
    }
}
  class Example2 {
    public static void main(String[] args) {
       Apple apple = new Apple();
      int pieces = apple.peel();
   }
}
```

The Advantage: If Fruit's peel() Method Changes, Apple Can Maintain its Interface

- If Fruit is changed so that the peel() method returns type Peel, the Apple class can accommodate the change and still provide the same interface to the outside world:

```
class Peel {                          class Apple {
    private int peelCount;                        private Fruit fruit = new Fruit();
    public Peel(int peelCount) {          public int peel() { // only this changes!
      this.peelCount = peelCount;         Peel peel = fruit.peel();
   }                                      return peel.getPeelCount();
    public int getPeelCount() {           }
       return peelCount;
   }              // Now the old implementation of Example2
   //...              // continues to work fine …
}                 class Example2 { // no longer broken
                          public static void main(String[] args) {
class Fruit {                 Apple apple = new Apple();
    // Return a Peel object that          int pieces = apple.peel();
   // results from the peeling activity.                }
   public Peel peel() {          }
       System.out.println(
      "Peeling is appealing.");      // Essentially Apple has insulated its users/clients
      return new Peel(1);       // from changes to Fruit's interface through
   }              // composition and forwarding.
}
```

Note: if the composed class interface change is large enough, even composition may not be able to accommodate the change …

Comparing Composition and Inheritance

- It's easier to change the interface of a back-end class (composition) than that of a superclass (inheritance). Superclass interface changes ripple down the inheritance hierarchy to subclasses and out to code that just uses those subclasses' interfaces.
- It's easier to change the interface of a front-end class (composition) than that of a subclass (inheritance). A change to a subclass interface must be compatible with the interfaces of its supertypes. For example, you can't add a method to a subclass that has the same method signature but a different return type from that of a method inherited from a superclass. You can do this in a front-end class.

Comparing Composition and Inheritance

- Composition allows you to delay the creation of back-end objects until (and unless) they are needed, as well as changing the back-end objects dynamically throughout the lifetime of the front-end object. With inheritance, you get the image of the superclass in your subclass object image as soon as the subclass is created, and it remains part of the subclass object throughout the lifetime of the subclass.
- On the other hand, it's easier to add new subclasses (inheritance) than it is to add new front-end classes (composition), because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition, unless you use composition with interfaces (see upcoming Interface slides).
- The explicit method-invocation forwarding (delegation) approach of composition will have a small performance cost compared to inheritance's single invocation of an inherited superclass method implementation. This performance cost may or may not be significant, depending on the application and JVM implementation, and it's getting smaller all the time.

Choosing Between Composition and Inheritance

- **Make sure inheritance models the *is-a* relationship.** Inheritance should be used only when a subclass *is-a (is-a-kind-of, can-be-used-as-a)* superclass. In the examples above, an Apple likely *is-a* Fruit, so you might be inclined to use inheritance.
- However, an important question to ask yourself when you think you have an *is-a* relationship is whether that *is-a* relationship will be constant throughout the lifetime of the application (and, with luck, the lifecycle of the code).
  - For example, you might think that an Employee *is-a* Person, when really Employee represents a role that a Person plays part of the time.
    What if the person becomes unemployed?
    What if the person is both an Employee and a Supervisor?
  - Impermanent *is-a* relationships should usually be modeled with composition.
- **Don't use inheritance just to get code reuse.** If all you really want is to reuse code and there is no *is-a* relationship in sight, use composition.
- **Don't use inheritance just to get polymorphism.** If all you really want is polymorphism, but there is no natural is-a relationship, use composition with interfaces.

Week 3 Topics

- **OO/Java review topics**
  - Inheritance and Composition
  - Interfaces
  - Abstract Classes
- More on Test-Driven Development (TDD) & JUnit

Design with Interfaces

- Separation of interface and implementation is central to Java's spirit, and the Java interface construct enables you to achieve this separation in your designs.

Source: Bill Venners, JavaWorld, 12/98

Java Interfaces

- An **interface** in the Java programming language is an abstract type that is used to specify an interface (in the generic sense) that classes must implement. An interface acts as a contract between the provider of some behavior and its users.

- **Interfaces are implicitly *abstract***, and cannot be directly instantiated except when instantiated by a concrete class which *implements* the interface. That class must implement (provide method bodies for) all abstract methods defined in the interface or else it must also be declared *abstract*, in which case its subclasses must implement the remaining methods.

- Classes may implement multiple interfaces, similar to multiple inheritance, and interfaces may extend other interfaces.

Primary source: Wikipedia

Example of Polymorphism Flexibility

- In the following (1), polymorphism allows a variable of type Animal to hold a reference to a Dog object (2):

```
abstract class Animal {

    abstract void talk();
}

class Dog extends Animal {

    void talk() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {

    void talk() {
        System.out.println("Meow.");
    }
}
```

1

abstract method signature

A class (3) that accepts Animal or any subclass of Animal to invoke talk() …

3

method implementation

```
class Interrogator {

    static void makeItTalk(Animal subject) {
        subject.talk();
    }
}
```

including new future subclasses (4):

4

method implementation

```
class Bird extends Animal {

    void talk() {
        System.out.println("Tweet, tweet!");
    }
}
```

2

```
Animal animal = new Cat();
```

(can be added later)

Subclass acting as superclass – upcasting (Liskov Substitution Principle)

Source: Bill Venners, JavaWorld, 12/98

“More Polymorphism” via Interfaces

- Given: add a new
t family
l pass its
tTalk():

```
interface Talkative {

    void talk();
}

abstract class Animal implements Talkative {

    abstract public void talk();
}

class Dog extends Animal {

    public void talk() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {

     public void talk() {
        System.out.println("Meow.");
    }
}

class Interrogator {

    static void makeItTalk(Talkative subject) {
        subject.talk();
    } // accepts any class that implements
      Talkative
}
```

all interface methods are *public abstract* by default

```
class Clock {
}

class CuckooClock implements Talkative {

    public void talk() {
        System.out.println("Cuckoo, cuckoo!");
    }
}
```

must implement

```
class Example4 {

    public static void main(String[] args) {
        CuckooClock cc = new CuckooClock();
        Interrogator.makeItTalk(cc);
    }
}
```

ng again

Even More Flexibility via Interfaces (Composition with Interfaces)

- Given: Apple reuses Fruit's peel(). You can later add a new class like this and still pass its instances to peelAnItem():

```
interface Peelable {
    int peel();
}

class Fruit {
    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public int peel() {

        System.out.println("Peeling is appealing.");
        return 1;
    }
}

class Apple implements Peelable {
    private Fruit fruit = new Fruit();

    public int peel() {
        return fruit.peel();
    }
}

class FoodProcessor {
    static void peelAnItem(Peelable item) {
        item.peel();
    }
}
```

"bridge" class

"composition"

delegation of *peel()* to the Fruit class (aka "forwarding")

```
class Banana implements Peelable {

    private Fruit fruit = new Fruit();

    public int peel() {
        return fruit.peel();
    }
}
```

composition

Using an interface design allows the flexibility of both composition (a "has-a" relationship) and polymorphism.

Example of When to Use an Interface

- ArrayList (java.util package) is handy, useful, used a lot, and implements the List interface (in the same package).
- Imagine you want to write a class that retrieves some data and returns it in the form of an ArrayList:

Obvious approach

```
public ArrayList getStuff(int id) {
    ArrayList rval = new ArrayList();
    // magical code that gets the stuff
    return rval;
}
```

Better: using an interface

```
public List getStuff(int id) {
    List rval = new ArrayList();
    // magical code that gets the stuff
    return rval;
} // in this case List is the interface
```

- Now if you decide you want to use, say, a LinkedList instead of an ArrayList because it's more efficient, you only need to change getStuff() vs. it *and all its callers*. This approach is used in Project 2!

Design Using Interfaces

- Interfaces are the preferred means of communicating with the parts of your program that represent abstractions that may have several implementations.
- Any class can provide an implementation of an interface.
  - As long as you don't change the interface, you can make changes to implementing classes, or plug in new classes, without impacting code that depends only on the interface.
  - In Java 8 you can add interface methods without any impact!
- **Interfaces help you decouple the parts of your system** from each other and generate code that is more flexible: more easily changed, extended, and customized.
- **http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html**

Some Thought Problems in Java

You are given the following classes and interfaces in Java:

```
interface I { int f(); }
interface J { int g(); }
class K implements I, J {
        public int f() { return 2; }
        public int g() { return 4; }
}
```

For each of the following code fragments, indicate what the resulting output is or whether there is an error in the fragment; if there is an error, indicate why it occurs, and if it occurs at compile time or at runtime.

a) `I x = new I(); x.f(); // Output/error?`

b) `I x = new K(); x.f(); // Output/error?`

c) `I x = new K(); x.g(); // Output/error?`

d) `I x = new K(); ((K) x).g(); // Output/error?`

Some Thought Problems in Java

Consider the following Java code:

```
interface I { int f(); int g(); }
abstract class C implements I { public int f() { return
  1; } }
class D extends C {
      public int f() { return 3; }
      public int g() { return f() + 2 * super.f(); }
}
class E extends D { public int f() { return 9; } }
//...
I x = new D(); System.out.println(x.g());
I y = new E(); System.out.println(y.g());
```

a) What does this code print (the last 2 lines)? Stated another way, which methods do these references invoke in those 2 lines: `f()` and `super.f()`?

b) Why must class C be declared as abstract?

Week 3 Topics

- **OO/Java review topics**
  - Inheritance and Composition
  - Interfaces
  - Abstract Classes
- More on Test-Driven Development (TDD) & JUnit

Abstract Classes and Methods

- An *abstract class* is a class that is declared abstract — it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed/inherited.
  - **An abstract class provides a contract between a service provider and its clients.**
- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY); // the abstract modifier is required, unlike in C#
```

- If a class includes *any* abstract methods, the class itself *must* be declared abstract, as in:

```
abstract class GraphicObject {  // a class can be declared abstract even with all concrete methods!
  // declare fields
  // declare non-abstract methods
  abstract void draw();
}
```

- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract. Eventually a non-abstract or concrete class must implement the remainder of all of the abstract methods, and only then can that class be instantiated.

Source: The Java Tutorials
http://java.sun.com/docs/books/tutorial/java/IandI/abstract.html

Abstract Classes vs. Interfaces

- Unlike interfaces, abstract classes can contain fields that are not public static final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains *only* abstract method declarations and static final fields, it should probably be declared as an interface instead.
- Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way. Think of Comparable or Cloneable, for example (possible future topics).
- By comparison, abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).
  Note: interfaces in Java 8 can provide default method implementations.
- An abstract class may have static fields and static methods. You can invoke such abstract class static members via a class reference — e.g., AbstractClass.staticMethod() — just as you would with any other class. An abstract class can also declare explicit constructors; an interface cannot.

A Simple Abstract Class Example

```
public abstract class DefaultPrinter { // there's no such thing as a DefaultPrinter …
    public String toString() {
      return "Use this to print documents."; // fully implemented concrete method
    }

    public abstract void print(Object document); // abstract method to be filled in
  }
```

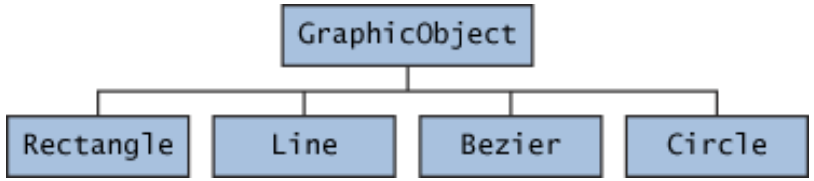
- The toString method has an implementation in DefaultPrinter (it overrides the one inherited from Object), so you do not need to override this method.
- The print method is declared abstract and does not have a body (no braces).

Here's an example of an implementation via extension in the MyPrinter class:

```
public class MyPrinter extends DefaultPrinter {
    public void print(Object document) { // overridden/implemented abstract method
      System.out.println("Printing document");
      // some code here
    }
  }
```

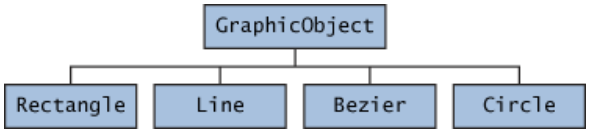
Another Abstract Class Example

- In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common.
- Some of these states and behaviors are the same for all graphic objects—for example: position, fill color, and moveTo. Others require different implementations—for example, resize or draw. All GraphicObjects must know how to draw or resize themselves; they just differ in how they do it.
- This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for example, GraphicObject, as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle inherit from GraphicObject

Abstract Class Example – 2



- The GraphicObject class can look something like this:

```
abstract class GraphicObject {
   private int x, y;
   ...
   void moveTo(int newX, int newY) {
       ... // concrete method
   }
   abstract void draw();
   abstract void resize();
}
```

- Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods

```
class Circle extends GraphicObject {
   void draw() {
       ...
   }
   void resize() {
       ...
   }
}
```

```
class Rectangle extends GraphicObject {
   void draw() {
       ...
   }
   void resize() {
       ...
   }
}
```

Some Thought Problems in Java

What is the correct answer for each of these 3 questions?

a) A <u>concrete</u> class has a constructor (select exactly one):

1. Always
2. Sometimes
3. Never

b) An <u>abstract</u> class has a constructor (select exactly one):

4. Always
5. Sometimes
6. Never

c) An <u>interface</u> has a constructor (select exactly one):

7. Always
8. Sometimes
9. Never

Concrete Class Constructors

- How do we know that a concrete class always has a constructor?
- An example Java program to prove this:

```
public class Concrete {
   Concrete c = new Concrete(); // compiles successfully
}
```

- Another example proof (two separate .java files):

```
public class Concrete {
    public Concrete(int x) {}
}

public class SubConcrete extends Concrete {} // does not compile!

                // why not???
```

Concrete Class Constructors

- This class:

```
public class SubConcrete extends Concrete {}
```

- … is expanded by the Java compiler as follows:

```
public class SubConcrete extends Concrete {
    public SubConcrete() { // automatic constructor
        super(); // call to superclass no-arg constructor
    } // the super() call causes the compilation error
}
```

Abstract Class & Interface Constructors

- How do we know that an abstract class always has a constructor?
- An example proof (again, two separate .java files):

```
public class Concrete {
    public Concrete(int x) {}
}

public abstract class Abstract extends Concrete {}

        // Similar to the above, this does not compile
```

- Did you know that an Abstract class can extend a concrete class?
- By definition, an Interface cannot have a constructor or state

Potential Future Review Topics

- Generic Types
- Annotations
- Collections
- Other topics of interest
  (what are some of your favorites?)

Week 3 Topics

- OO/Java review topics
  - Inheritance and Composition
  - Interfaces
  - Abstract Classes
- **More on Test-Driven Development (TDD) & JUnit**

Why Do Test-Driven Development?

The challenge: solving the right problem right

The function of software development is to support the operations and business of an organization. Our focus as professional software developers should be on delivering systems that help our organizations improve their effectiveness and throughput, that lower the operational costs, and so forth.

Even after several decades of advancements in the software industry, the quality of the software produced remains a problem. Considering the recent years' focus on time to market, the growth in the sheer volume of software being developed, and the stream of new technologies to absorb, it is no surprise that software development organizations have continued to face quality problems.

There are two sides to these quality problems: high defect rates and lack of maintainability.

Why Do Test-Driven Development?

Solution: being test-driven

Just like the problem we're facing has two parts to it—poorly written code and failure to meet actual needs—the solution we're going to explore in the coming chapters is two-pronged as well. On one hand, we need to learn how to build the thing right. On the other, we need to learn how to build the right thing. The solution I'm describing in this book—being test-driven—is largely the same for both hands. The slight difference between the two parts to the solution is in *how* we take advantage of tests in helping us to create maintainable, working software that meets the customer's actual, present needs.

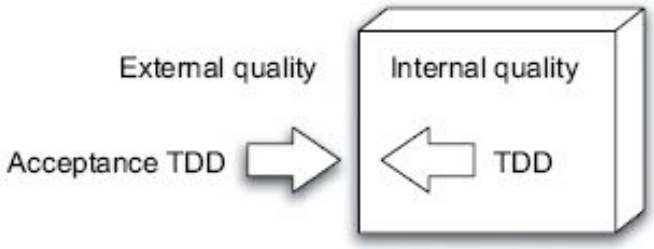


Figure 1.1
TDD is a technique for improving the software's internal quality, whereas acceptance TDD helps us keep our product's external quality on track by giving it the correct features and functionality.

What's in it for me?

- I rarely get a support call or end up in a long debugging session.
- I feel confident in the quality of my work.
- I have more time to develop as a professional.

TDD and API Design

When we write a test in the first step of the TDD cycle, we're really doing more than just writing a test. We're making design decisions. We're designing the API—the interface for accessing the functionality we're testing. By writing the test before the code it's testing, we are forcing ourselves to think hard about how we want the code to be used. It's a bit like putting together a jigsaw puzzle. As illustrated by figure 1.5, it's difficult to get the piece you need if you don't know the pieces with which it should connect.

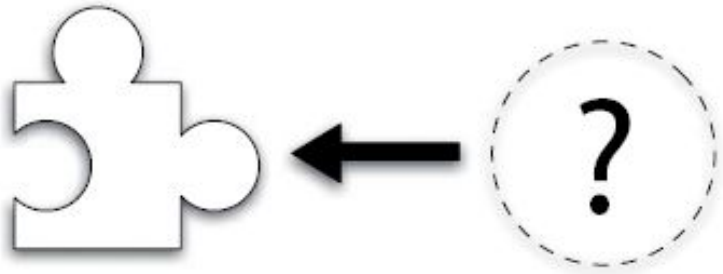
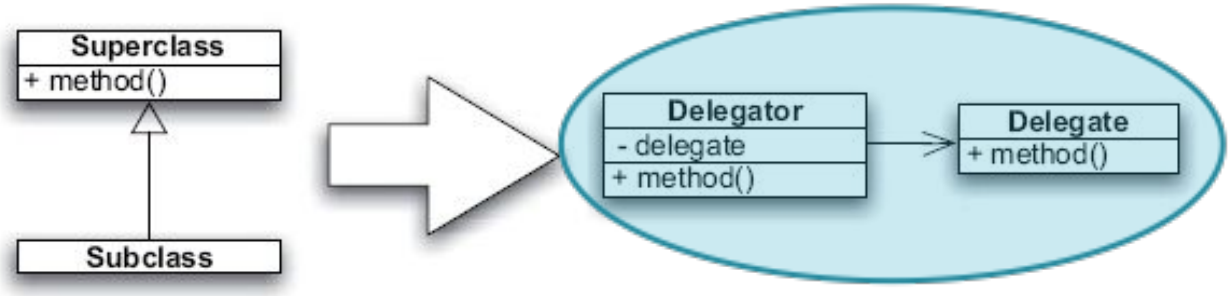


Figure 1.5
How do we know what our interface should be like if we don't try to use it? We don't. Writing the test before the code makes us think about our design from the code user's (the developer's) perspective, leading to a usable API.

It's not easy to create simple-to-use APIs. That's why we need all the help we can get. As it turns out, driving our design with tests is extremely effective and produces modular, testable code. Because we're writing the test first, we have no choice but to make the code testable. By definition, the code we write *is* testable—otherwise it wouldn't exist!

A Note About Refactoring: Think of It as Program Transformation

- Example: replacing inheritance with delegation
  - Why? As we saw, a subclass may be inheriting more than it really needs to or should reuse, both fields and methods



This is forwarding that we discussed in earlier slides.

Figure 1.10 Refactorings are transformations between two functionally identical states or structures of the code. Here we see a transformation from using inheritance hierarchy to using a delegate to provide the same functionality while improving the design's fitness for our current needs. These transformations are not absolute improvements—they're simply disciplined transitions from one possible design to another. In fact, for many refactorings there exists a reverse refactoring, making the same transformation in the opposite direction.

  - One *ns*
  - We'll cover design patterns in more detail later …

Test Driven Decomposition

- Task- vs. test-driven decomposition of requirements: an example mail-template system (filling in mail fields)

Mail template subsystem decomposed into a set of tasks	Mail template subsystem decomposed into a set of test(s)
Write a regular expression for identifying variables from the template.	Template without any variables renders as is.
Implement a template parser that uses the regular expression.	Template with one variable is rendered with the variable replaced with its value.
Implement a template engine that provides a public API and uses the template parser internally.	Template with multiple variables is rendered with the appropriate placeholders replaced by the associated values.
...	...

- Tasks tell us what we should do, whereas tests tell us how to know when we're done

Tool Support for TDD in Java – JUnit

Unit-testing with xUnit

A number of years ago, Kent Beck created a unit-testing framework for SmallTalk called SUnit (http://sunit.sf.net). That framework has since given birth to a real movement within the software development community with practically every programming language in active use having gotten a port of its own.[12] For a Java developer, the de facto standard tool—also based on the design of SUnit—is called JUnit, available at http://www.junit.org/. The family of unit-testing frameworks that are based on the same patterns found in SUnit and JUnit is often referred to as *x*Unit because of their similarities: If you know one, it's not much of a leap to start using another (provided that you're familiar with the programming language itself).

What exactly does a *unit-testing framework* mean in the context of xUnit? It means that the library provides supporting code for writing unit tests, running them, and reporting the test results. In the case of JUnit, this support is realized by providing a set of base classes for the developer to extend, a set of classes and interfaces to perform common tasks such as making assertions, and so forth. For running the unit tests written with JUnit, the project provides different *test runners*—classes that know how to collect a set of JUnit tests, run them, collect the test results, and present them to the developer either graphically or as a plain-text summary.

JUnit 4 Overview

```
import static org.junit.Assert.*;
import org.junit.*;
import java.io.*;

public class TestConsoleLogger {

    private static final String EOL =
        System.getProperty("line.separator");
    private ConsoleLogger logger;
    private PrintStream originalSysOut, originalSysErr;
    private ByteArrayOutputStream sysOut, sysErr;

    @Before public void createFixture() {
        logger = new ConsoleLogger();
        originalSysOut = System.out;
        originalSysErr = System.err;
        sysOut = new ByteArrayOutputStream();
        sysErr = new ByteArrayOutputStream();
        System.setOut(new PrintStream(sysOut));
        System.setErr(new PrintStream(sysErr));
    }

    @After public void resetStandardStreams() {
        System.setOut(originalSysOut);
        System.setErr(originalSysErr);
    }

    @Test public void infoLevelGoesToSysOut()
        throws Exception {
        logger.info("msg");
        streamShouldContain("[INFO] msg" + EOL, sysOut.toString());
    }
```

1. Import static Assert methods from JUnit
2. By convention, test class names start with Test; this one will be used to test a ConsoleLogger
3. Common "fixture" for test methods
4. @Before: create known state
5. @After: test cleanup
6. Public void methods annotated with @Test are considered JUnit test cases
7. JUnit catches all exceptions

JUnit 4 Overview (continued)

```
@Test(timeout = 100)                                    ❽ @Test annotation allows attributes
public void errorLevelGoesToSysErr() throws Exception {
    logger.error("Houston...");
    streamShouldContain("[ERROR] Houston..."
        + EOL, sysErr.toString());
}

private void streamShouldContain(String expected,       ❾ Declaration of non-test helper method
    String actual) {
    assertEquals("Wrong stream content.",               ❶ Use of Assert class method
        expected, actual);
}
}
```

❶ We get assertion methods from JUnit's `Assert` class through a static import.

❷ The name of the class should indicate that it's a test—for example, have the class name start with `Test`.

❸ The instance variables set up in the `@Before` method represent the common fixture for the test methods.

❹ We can prepare a known state for the test by tagging a public method with the `@Before` annotation.

❺ We can clean up after our test by tagging a public method with the `@After` annotation.

❻ All public void methods tagged with `@Test` are considered test cases by the JUnit 4 TestRunner.

❼ Test methods can declare any exceptions—JUnit catches them.

❽ The `@Test` annotation also allows for timing tests, testing for expected exceptions, and so on.

❾ We can declare any number of helper methods as long as they don't look like test methods.

JUnit 4 Annotations

JUnit 4 Annotation	Description
@Test public void method()	Annotation @Test identifies that this method is a test method.
@Before public void method()	Will perform the method() before each test. This method can prepare the test environment, e.g. read input data, initialize the class
@After public void method()	Will perform the method() after each test.
@BeforeClass public void method()	Will perform the method before the start of all tests. This can be used to perform time intensive activities for example be used to connect to a database
@AfterClass public void method()	Will perform the method after all tests have finished. This can be used to perform clean-up activities for example be used to disconnect to a database
@Ignore	Will ignore the test method, e.g. useful if the test has not yet been implemented, the underlying code has been changed and the test has not yet been adapted, or if the runtime of this test is just to long to be included
@Test(expected=IllegalArgumentException.class)	Tests if the method throws the named exception
@Test(timeout=100)	Fails if the method takes longer then 100 milliseconds

JUnit 4 Test Methods (Asserts)

Statement	Description
fail(String)	Let the method fail, might be usable to check that a certain part of the code is not reached
assertTrue(true);	TRUE
assertsEquals([String message], expected, actual)	Test if the values are the same. Note: for arrays the reference is checked not the content of the arrays
assertsEquals([String message], expected, actual, tolerance)	Usage for float and double; the tolerance is the number of decimal places which must be the same
assertNull([message], object)	Checks if the object is null
assertNotNull([message], object)	Check if the object is not null
assertSame([String], expected, actual)	Check if both variables refer to the same object
assertNotSame([String], expected, actual)	Check that both variables refer not to the same object
assertTrue([message], boolean condition)	Check if the boolean condition is true
try {a.shouldThroughException(); fail("Failed")} catch (RuntimeException e) {asserttrue(true);}	Alternative way for checking for exceptions

Note: the table's assertsEquals method names should be **assert**Equals.

Simple JUnit Example: Subscription

```
public class Subscription {            // class to be tested, with some defects

      private int price;              // subscription total price in cents
      private int length;              // length of subscription in months

      public Subscription(int p, int n) {
            price = p;
            length = n;
      }

     /**
      * Calculate the monthly subscription price in $,
      * rounded up to the nearest cent.
      */
      public double pricePerMonth() {
            double r = (double) price / (double) length;// defect: returns price in cents
   vs $
            return r;                  // defect: doesn't round the result
      }

     /**
      * Call this to cancel/nullify this subscription.
      */
      public void cancel() {
             length = 0;
      }
}
```

JUnit Test for the Subscription Class

```
 import org.junit.*;
import static org.junit.Assert.*;

public class TestSubscription {

    @Test
    public void testReturnsDollars() {
        System.out.println("Test if pricePerMonth returns dollars per month...");
        Subscription s = new Subscription(200,2);
        assertTrue(s.pricePerMonth() == 1.0);
    }

    @Test
    public void testRoundsUp() {
        System.out.println("Test if pricePerMonth rounds up correctly...");
        Subscription s = new Subscription(200,3);
        assertTrue(s.pricePerMonth() == 0.67);
    }
}
```

Running the Subscription JUnit Test

Compiling and running the JUnit test in a Windows cmd shell (not in an IDE)
→TestSubscription.bat file:

cd <full path to Subscription.java and TestSubscription.java> ← **put both in the same Windows folder***

javac -cp . Subscription.java ← **-cp defines the Java classpath – set it to . (the current directory)**

javac -cp .;<full path to JUnit.jar> TestSubscription.java ← **set classpath to . plus the Junit.jar file path****

java -cp .;<full path to JUnit.jar> org.junit.runner.JUnitCore TestSubscription

Output:

```
…
Time: 0
  There were 2 failures:
  1) testReturnsDollars(TestSubscription)
   java.lang.AssertionError:
        ...
          at TestSubscription.testReturnsDollars(TestSubscription.java:10) ← these lines are buried deep ...
          ...
  2) testRoundsUp(TestSubscription)
   java.lang.AssertionError:
          ...
          at TestSubscription.testRoundsUp(TestSubscription.java:17) ← these lines are buried deep ...
          ...
  FAILURES!!!
  Tests run: 2, Failures: 2
```

*** Note: put any full Windows pathname in quotes if it includes spaces**
**** The JUnit .jar path must include the full .jar filename**

End of Week 3

- OO/Java review topics
  - Inheritance and Composition
  - Interfaces
  - Abstract Classes
- More on Test-Driven Development (TDD) & JUnit

Backup: Interfaces vs. Abstract Classes – 1

http://mindprod.com/jgloss/interfacevsabstract.html

feature	interface	abstract class
multiple inheritance	A class may implement several interfaces.	A class may extend only one `abstract` class.
default implementation	An `interface` cannot provide any code at all, much less default code. **Changed in Java 8.**	An `abstract` class can provide complete code, default code, and/or just stubs that have to be overridden.
constants	Static final constants only, can use them without qualification in classes that implement the `interface`. On the other paw, these unqualified names pollute the namespace. You can use them and it is not obvious where they are coming from since the qualification is optional.	Both instance and static constants are possible. Both static and instance intialiser code are also possible to compute the constants.
third party convenience	An `interface` implementation may be added to any existing third party class.	A third party class must be rewritten to extend only from the `abstract` class.
is-a vs -able or can-do	Interfaces are often used to describe the peripheral abilities of a class, not its central identity, e.g. an `Automobile` class might implement the `Recyclable` `interface`, which could apply to many otherwise totally unrelated objects.	An `abstract` class defines the core identity of its descendants. If you defined a Dog `abstract` class then Dalmatian descendants **are** Dogs, they are not merely dogable. Implemented interfaces enumerate the general things a class can do, not the things a class is. In a Java context, users should typically implement the Runnable `interface` rather than extending `Thread`, because they're not really interested in providing some new `Thread` functionality, they normally just want some code to have the capability of running independently. They want to create something that can be run in a thread, not a new kind of thread.The similar is-a vs has-a debate comes up when you decide to inherit or delegate. multiple inheritance for further discussion of is-a vs has-a
plug-in	You can write a new replacement module for an `interface` that contains not one stick of code in common with the existing implementations. When you implement the interface, you start from scratch without any default implementation. You have to obtain your tools from other classes; nothing comes with the `interface` other than a few constants. This gives you freedom to implement a radically different internal design.	You must use the `abstract` class as-is for the code base, with all its attendant baggage, good or bad. The `abstract` class author has imposed structure on you. Depending on the cleverness of the author of the `abstract` class, this may be good or bad.

Backup: Interfaces vs. Abstract Classes – 2

http://mindprod.com/jgloss/interfacevsabstract.html

feature	interface	abstract class
homogeneity	If all the various implementations share is the method signatures, then an `interface` works best.	If the various implementations are all of a kind and share a common status and behaviour, usually an `abstract` class works best. Another issue that's important is what I call "heterogeneous vs. homogeneous." If implementors/subclasses are homogeneous, tend towards an `abstract` base class. If they are heterogeneous, use an `interface`. (Now all I have to do is come up with a good definition of hetero/homo-geneous in this context.) If the various objects are all of-a-kind, and share a common state and behavior, then tend towards a common base class. If all they share is a set of method signatures, then tend towards an `interface`.
maintenance	If your client code talks only in terms of an `interface`, you can easily change the concrete implementation behind it, using a factory method.	Just like an `interface`, if your client code talks only in terms of an `abstract` class, you can easily change the concrete implementation behind it, using a factory method.
speed	Slow, requires extra indirection to find the corresponding method in the actual class. Modern JVMs are discovering ways to reduce this speed penalty.	Fast
terseness	The constant declarations in an `interface` are all presumed `public static final`, so you may leave that part out. You can't call any methods to compute the initial values of your constants. You need not declare individual methods of an `interface` `abstract`. They are all presumed so.	You can put shared code into an `abstract` class, where you cannot into an `interface`. If interfaces want to share code, you will have to write other bubblegum to arrange that. You may use methods to compute the initial values of your constants and variables, both instance and static. You must declare all the individual methods of an `abstract` class `abstract`.
adding functionality Changed in Java 8!	If you add a new method to an `interface`, you must track down all implementations of that `interface` in the universe and provide them with a concrete implementation of that method.	If you add a new method to an `abstract` class, you have the option of providing a default implementation of it. Then all existing code will continue to work without change.