... picking up where we stopped

- We were discussing the class hierarchy
- We had been focused on extending a class by creating a new child class
  - We looked at "overloading" methods
  - Allows us to have multiple methods with the same name but with different type signatures
  - Used when some arguments have default values. The "short" versions just call the "ultimate" one with default values for any unspecified parameters

Interfaces

A class has only one parent but can implement many interfaces. Decide on class hierarchy and what interfaces to support as part of process of developing clean, elegant code

Example: Overriding "toString"

- Class `Object` defines `toString`, so every object of every class contains `toString`.
  - `toString` in `Object`: prints `@Address`
  - Most classes override `toString()`
  - `toString()` in an object usually returns a string that contains values of the fields of the object, printed in a nice way.

```java
@Override // An "attribute": tells Eclipse what we intend
public string toString() {
    return this.name + ":" + this.value;
}
```

Example: Overriding "toString"

- Class `Object` defines `toString`, so every object of every class contains `toString`.
  - `toString` in `Object`: prints `@Address`
  - Most classes override `toString()`
  - `toString()` in an object usually returns a string that contains values of the fields of the object, printed in a nice way.

```java
// Putting it right into the declaration can increase clarity
public @Override string toString() {
    return this.name + ":" + this.value;
}
```
Example: Overriding “toString”

- Class `Object` defines `toString`, so every object of every class contains `toString`.
- `toString` in `Object`: prints `name@Address`
- Most classes override `toString()` in part.
- Method `toString` should override some inherited method.
- It contains values of the fields of the object, printed in a nice way.

// If you make a mistake, now Eclipse will notice & warn you
public @Override String ToString() { // Mistake: To, not To
    return this.name + ":" + this.value;
}

Is toString() the only use for override?

- This the most common use!

- But there is one other very common case
  - Java has many pre-defined classes for making lists or other kinds of collections
  - It can search and sort within them
  - These need a way to compare elements
  - Again, there are default comparison rules but they don’t often do exactly what you would want

Example: Overloading “compareTo”

- Interface `Comparable` has three methods:
  - `a.equals(b)`: returns true/false
  - `a.compareTo(b)`: returns -/+0
  - `a.hashCode()`: returns a number (ideally unique and randomized) representing object `a`. Usually return `data.hashCode()` for some data object in a that represents `a`’s “value” (perhaps a string or a number)
- Warning: Override one method? Must override all.
- Otherwise, get mix of inherited and override versions, and Java utilities that depend on them malfunction

Accessing Overridden Methods

Suppose a class overrides method `m` like `toString()` in the examples we just saw.
- Sometimes it is useful to be able to call the parent version.
- E.g., maybe you still want to print the Name@Address using `Object.toString()`.
- In subclass, call overridden method using `super.m()`.

Example:

```java
public @Override String toString() {
    return super.toString() + "\n      " + name + ", price=\"" + price;
}
```

Drill down: Initializing an object

- Questions to ask about initialization in Java:
  - When do things have default values? What are those?
  - What happens if you touch an uninitialized object?
  - What if you need a more complicated initialization that requires executing some code?
  - Who gets initialized first in an parent/subclass situation?
  - Who gets initialized first if two different classes have initializers that each refer to the other?

Shifting gears

- We’ve focused on the type hierarchy.
- Now let’s look at a different but related question: how things get initialized in Java.
  - For a single object
  - ... for static variables, and then for instance variables
  - ... then for objects in a subclass of a parent class
  - ... and then for two classes that refer to one another
Constructors

- Called to create new instances of a class.
- A class can define multiple constructors.
- Default constructor initializes all fields to default values (0, false, null...).

```java
class Thing {
    int val;
    Thing(int val) {
        this.val = val;
    }
    Thing() {
        this(3);
    }
}

Thing one = new Thing(1);
Thing two = new Thing(2);
Thing three = new Thing();
System.out.println("Thing two = " + two);
```

Java automatically calls two.toString(). It works: class Thing inherits Object.toString().

Won't print value in field val. Why not?

Constructors in class hierarchy

- Principle: initialize superclass fields first.
- Implementation: First statement of constructor must be call on constructor in this class or superclass.
  Java syntax or is:
  ```java
  this(arguments);
  ```
  or
  ```java
  super(arguments);
  ```

  If you don't do this, Java inserts call `super();`

```java
public class Hotel extends Lodging { … }
```

What are local variables?

- Local variable: variable declared in method body.
- Not initialized, you need to do it yourself.
- Eclipse should detect these mistakes and tell you.

```java
public class CSuper {  
    public CSuper() {
        System.out.println("CSuper constructor called.");
    }
}

public class A extends CSuper {  
    public A() {
        super();
        System.out.println("Constructor in A running.");
    }
    public static void main(String[] str) {
        ClassA obj = new ClassA();
    }
}
```

What happens here?

- If you access an object using a reference that has a `null` in it, Java throws a `NullPointerException`.
- Thought problem: what did developer intend?
  - Probably thinks `myFriend` points to an existence of class `RoomMate`.
  - `RoomMate` object created only with new-expression.

```java
class Thing {
    RoomMate myFriend;
    Thing(int val) {
        myFriend.value = val;
    }
}
```

Static Initializers

- An initializer for a static field runs once, when the class is loaded.
- Used to initialize static objects.

```java
class StaticInit {
    static Set<String> courses = new HashSet<String>();
    static {
        courses.add("CS 2110");
        courses.add("CS 2111");
    }
    static {
        // Glimpse of a "generic"
    }
}
```
Reminder: Static vs Instance Example

```java
class Widget {
    static int nextSerialNumber = 10000;
    int serialNumber;

    Widget() { serialNumber = nextSerialNumber++; }

    public static void main(String[] args) {
        Widget a = new Widget();
        Widget b = new Widget();
        Widget c = new Widget();

        System.out.println(a.serialNumber);
        System.out.println(b.serialNumber);
        System.out.println(c.serialNumber);
    }
}
```

Accessing static versus instance fields

- If name is unique and in scope, Java knows what you are referring to. In scope: in this object and accessible. Just use the (unqualified) name:
  - serialNumber
  - nextSerialNumber
- Refer to static fields/methods in another class using name of class:
  - Widget.nextSerialNumber
- Refer to instance fields/methods of another object using name of object:
  - a.serialNumber

Hair-raising initialization

- Suppose `a` is of type `A` and `b` is of type `B`
  - ... and `A` has static field `myAVal`,
  - ... and `B` has field `myBVal`
- Suppose we have static initializers:
  - `public static int myAVal = B.myBVal+1;`
  - `public static int myBVal = A.myAVal+1;`

Some Java « issues »

- An overriding method cannot have more restricted access than the method it overrides

```java
class A {
    public int m() {...}
}
class B extends A {
    private @Override int m() {...} //Illegal!
}
```

Can we override a field?

- ... Yes, Java allows this. There are some situations where it might even be necessary.
- We call the technique “shadowing”
- But it isn’t normally a good idea.
... a nasty example

```java
class A {
    int i = 1;
    int f() { return i; }
}
class B extends A {
    int i = 2; // Shadows variable i in class A.
    int @Override f() { return -i; } // Overrides method f in class A.
}
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i); // Refers to B.i; prints 2.
        System.out.println(b.f()); // Still refers to B.f(); prints -2.
        A a = (A) b; // Cast b to an instance of class A.
        System.out.println(a.i); // Now refers to A.i; prints 1;
        System.out.println(a.f()); // Still refers to B.f(); prints -2;
    }
}
```
Not a perfect, pretty picture

- It is often easy to take the first step but not the second one
- Large abstractions come naturally. But details often work better from the ground up
- Many developers work by building something small, testing it, then extending it
  - It helps to not be afraid of needing to recode things

Top-Down vs. Bottom-Up

- Is one way better? Not really!
  - It’s sometimes good to alternative
  - By coming to a problem from multiple angles you might notice something you had previously overlooked
  - Not the only ways to go about it
- Top-Down: harder to test early because parts needed may not have been designed yet
- Bottom-Up: may end up needing things different from how you built them

Software Process

- For simple programs, a simple process…
  - “Waterfall”
- But to use this process, you need to be sure that the requirements are fixed and well understood!
  - Many software problems are not like that
  - Often customer refines requirements when you try to deliver the initial solution!

Incremental & Iterative

- Deliver versions of system in several small cycles
- Recognizes that for some settings, software development is like gardening
  - You plant seeds… see what does well… then replace the plants that did poorly

Information Hiding

- What “information” do classes hide?
  - “Internal” design decisions.

```java
public class Set {
    public void add(Object o) ...
    public boolean contains(Object o) ...
    public int size() ...
}
```

- Classes' interface: everything in it that is externally accessible

Encapsulation

- By hiding code and data behind its interface, a class encapsulates its “inner workings”
- Why is that good?
  - Can change implementation later without invalidating the code that uses the class

```java
class LineSegment {
    private Point2D p1, p2;
    public double length() {
        return p1.distance(p2);
    }
}
```
Degenerate Interfaces

- Public fields are usually a Bad Thing:

```java
class Set {
    public int count = 0;
    public void add(Object o) ...
    public boolean contains(Object o) ...
    public int size() ...
}
```

- Anybody can change them; the class has no control

Use of interfaces?

- When team builds a solution, interfaces can be valuable!
  - Rebecca agrees to implement the code to extract genetic data from files
  - Tom will implement the logic to compare DNA
  - Willy is responsible for the GUI

- By agreeing on the interfaces between their respective modules, they can all work on the program simultaneously

Principle of Least Astonishment

- Interface should “hint” at its behavior

```java
Bad:
p
```public int product(int a, int b) {
    return a*b > 0 ? a*b : -a*b;
}
```

```java
Better:
/** Return absolute value of a * b */
p
```public int absProduct(int a, int b) {
    return a*b > 0 ? a*b : -a*b;

Names and comments matter!

Outsmarting yourself

- A useful shorthand... Instead of

```java
something = something * 2;
```

... use

```java
something *= 2;
```

- All such operators:

```java
+= -= *= /= %= ^=
```

Principle of Least Astonishment

- Unexpected side effects are a Bad Thing

```java
class MyInteger {
    private int value;

    public MyInteger times(int factor) {
        value *= factor;
        return new MyInteger(value);
    }
}
```

```java
MyInteger i = new MyInteger(100);
MyInteger j = i.times(10);
```

Developer trying to be clever. But what does code do to i?

Duplication

- It is common to find some chunk of working code, make a replica, then edit the replica

- But this makes your software fragile: later, when code you copied needs to be revised, either
  - The person doing that changes all instances, or
  - Some become inconsistent

- Duplication can arise in many ways:
  - Constants (repeated “magic numbers”)
  - Code vs. comment
  - Within an object’s state
  - ...
“DRY” Principle

- Don’t Repeat Yourself
- Nice goal: have each piece of knowledge live in one place
- But don’t go crazy over it
  - DRYing up at any cost can increase dependencies between code
  - “3 strikes and you refactor” (i.e. clean up)

Refactoring

- Refactor: improve code’s internal structure without changing its external behavior
- Most of the time we’re modifying existing software
- “Improving the design after it has been written”
- Refactoring steps can be very simple:
  
  ```java
  public double weight(double mass) {
      return mass * GRAVITY;
  }
  ```

- Other examples: renaming variables, methods, classes

Why is refactoring good?

- If your application later gets used as part of a Nasa mission to Mars, it won’t make mistakes
- Every place that the gravitational constant shows up in your program a reader will realize that this is what they are looking at
- The compiler may actually produce better code

Common refactorings

- Rename something
  - Eclipse will do it all through your code
  - Warning: Eclipse doesn’t automatically fix comments!

- Take a chunk of your code and turn it into a method
  - Anytime your “instinct” is to copy lines of code from one place in your program to another and then modify, consider trying this refactoring approach instead...
  - ... even if you have to modify this new method, there will be just one “version” to debug and maintain!

Extract Method

- A comment explaining what is being done usually indicates the need to extract a method
  ```java
  public double totalArea() {
      // add the circle area = PI * pow(radius,2);
      ...
  }
  ```

- One of most common refactorings

```java
public double totalArea() {  
    area += circleArea(radius);  
}  
```
Refactoring & Tests

- Eclipse supports various refactorings
- You can refactor manually
  - Automated tests are essential to ensure external behavior doesn’t change
  - Don’t refactor manually without retesting to make sure you didn’t break the code you were “improving”!
- More about tests and how to drive development with tests next week

Summary

- We’ve seen that Java offers ways to build general classes and then to create specialized versions of them
  - In fact, we saw several ways to do this
- Our challenge is to use this power to build clean, elegant software that doesn’t duplicate functionality in confusing ways
- The developer’s job is to find abstractions and use their insight to design better code!