CREATING SOFTWARE FROM SCRATCH

• determining the primary objects of the application
  – think of objects as *active* agents
    *anthropomorphize* (give objects intentionality)
    objects have responsibilities (= methods)
    objects interacting with other objects
    *collaborations*
    sharing information
    need-to-know access
    pushing vs. pulling information
    *events* that trigger interactions
    cascading events
    events as objects too
    changes of *state* within some objects, etc.

• how to start?
CREATING SOFTWARE FROM SCRATCH

• **Use Cases**
  – deciding on what you want to achieve
    software requirements (specifications)
    case-by-case (scenario-by-scenario)
  – start with most important requirements
    (not error cases)
  – for each case or scenario:
    what objects perform what roles?
    with what other objects?
  – noun phrases = objects
  – adjectives = subclasses/superclasses of objects
  – verbs = methods

• Design and analyze before programming
  – *diagram* to make your ideas concrete
    UML *class diagrams* (this *isa* that and *hasa* that)
    UML *sequence diagrams* (this *calls* that *in* that then that *calls* ...)


PERILS OF WRITING CODE EVOLUTIONARILY
(or: writing without thinking enough)

class Ctr {
    private int n;
    public Ctr() { n = 0; }
    public void increment() { n++; }
    public int getCount() { return n; }
}

class Test {
    public static void main (String[] args) {
        Ctr c = new Ctr();
        for (int i = 0; i < 3; i++) {
            c.increment();
            System.err.println("counter = " + c.getCount());
        }
    }
}

>> java Test
counter = 1
counter = 2
counter = 3...
Then maybe we decide we want a new feature: incrementing by a specified step size (not just by 1)

class Ctr {
    private int n, inc;

    public Ctr(int stepSize) {
        n   = 0;
        inc = stepSize;
    }

    public Ctr() { this(1); }

    public void increment() { n += inc; }
    public int getCount() { return n; }
}
class Test {
    public static void main (String[] args) {
        Ctr c = new Ctr(2);
        for (int i = 0; i < 3; i++) {
            c.increment();
            System.err.println("counter = " + c.getCount());
        }
    }
}

>> counter = 2, 4, 6, ...
class Ctr {
    protected int n, inc;
    public Ctr() { this(1); }
    public Ctr(int stepSize) {
        n = 0;
        inc = stepSize;
    }
    public void increment() { n += inc; }
    public int getCount() { return n; }
}

defining the following count-by-two class
class Ctr2 extends Ctr {
    public Ctr2() { super(2); }
    public void decrement() { n -= inc; }
}

note we slipped in a new method called decrement
and adding even more functionality, e.g. a way to report the number of increments and decrements:

class Ctr2 extends Ctr {
    protected int numIncs, numDecs;
    public Ctr2() { super(2); }
    public void decrement() {
        n -= inc;
        numDecs++;
    }
    public int numIncrements() { return numIncs; }
    public int numDecrements() { return numDecs; }
}

has any bug been introduced?
Refactoring (moving methods around)

class Ctr {
    private int n = 0;
    private int numIncs = 0;
    private int numDecs = 0;
    private int inc;

    public Ctr(int stepSize) { inc = stepSize; }
    public Ctr() { this(1); }

    public void increment() { n += inc; numIncs++; }
    public void decrement() { n -= inc; numDecs++; }
    public int numIncrements() { return numIncs; }
    public int numDecrements() { return numDecs; }
    public int getCount() { return n; }
    public void reset() { n = 0; numIncs = 0; numDecs = 0; }
}

class Ctr2 extends Ctr {
    public Ctr2() { super(2); }
}

THE LESSONS

- incremental evolution of designs practically inevitable
- look for symmetries in your design; be cautious of breaking symmetries
- be ready to refactor to enhance symmetry, regularity, elegance
- test thoroughly each change (don't presume)
- analyze rather than hack
- don't be in a hurry to jump into programming
- design for extensibility, testability, ... other ity's
- design/program with tight adherence to OO principles
- attend to your programming style, too!
interfaces would require a lot of redundant coding
and putting it all together
public abstract class Duck {
    private FlyBehavior flyBehavior;
    private QuackBehavior quackBehavior;

    public Duck(){}
    public void setFlyBehavior(FlyBehavior fb) { flyBehavior = fb; }
    public void setQuackBehavior(QuackBehavior qb) { quackBehavior = qb; }

    abstract void display();

    public void performFly() { flyBehavior.fly(); }
    public void performQuack() { quackBehavior.quack(); }
    public void swim() { System.out.println("All ducks float, even decoys!"); }
}

public class MallardDuck extends Duck {
    public MallardDuck() {
        QuackBehavior q = new Quack();
        setQuackBehavior(q);
        setFlyBehavior(new FlyWithWings());

        public void display() { System.out.println("I'm a real Mallard duck"); }
    }
}
code for the Strategy Pattern

```java
public class MallardDuck extends Duck {
    public MallardDuck() {
        QuackBehavior q = new Quack();
        setQuackBehavior(q);
        setFlyBehavior(new FlyWithWings());

    public void display() { System.out.println("I'm a real Mallard duck"); }
}

public interface QuackBehavior { public void quack(); }

public class Quack implements QuackBehavior {
    public void quack() { System.out.println("Quack"); }
}

public class Squeak implements QuackBehavior {
    public void quack() { System.out.println("Squeak"); }
}
```
public abstract class Duck {
    private FlyBehavior flyBehavior;
    private QuackBehavior quackBehavior;

    public Duck() {}
    public void setFlyBehavior(FlyBehavior fb) { flyBehavior = fb; }
    public void setQuackBehavior(QuackBehavior qb) { quackBehavior = qb; }

    abstract void display();

    public void performFly() { flyBehavior.fly(); }
    public void performQuack() { quackBehavior.quack(); }
    public void swim() { System.out.println("All ducks float, even decoys!"); }
}

public class MallardDuck extends Duck {
    public MallardDuck() {
        QuackBehavior q = new Quack();
        setQuackBehavior(q);
        setFlyBehavior(new FlyWithWings());

        public void display() { System.out.println("I'm a real Mallard duck"); }
    }
}
The Open-Closed Principle (OCP)

• **Open For Extension**
  – This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

• **Closed for Modification**
  – The source code of such a module is inviolate. No one is allowed to make source code changes to it.

• The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes.

• Define a family of algorithms, encapsulate each one, and make them interchangeable. It lets the algorithm vary independently from clients that use it.
CREATING A HIERARCHY OUT OF THIN AIR

• Idea:
two mutual antagonists (Warrior). Their Swords and Armor can have different hardneses. The Sword must be made of sufficiently hard metal in order to puncture the Armor or Shield of the enemy. If it is, the opponent’s life energy is reduced, otherwise the opponent simply scoffs at the individual, and blah blah...

• Use cases:
case 1: Individual Enemy has a sword made of sterner stuff than its enemy's armor. Reduce opponent’s life energy. Follow on with consequences of reduced life energy (e.g., die).

• classes:
   Sword, Shield, Armor, Warrior, ....

• methods:
   stab(), ...
CREATING A HIERARCHY OUT OF THIN AIR

• This is definitely shoot-from-the-hip programming (also called shoot-yourself-in-the-foot programming)

```java
class MetalThing {
    public int hardness;
    MetalThing(int hardness) { this.hardness = hardness; }
    public String toString() { return ("Hardness = " + hardness + ","); }
}

class Shield extends MetalThing {
    Shield(int hardness) { super(hardness); }
}

class Armor extends MetalThing {
    Armor(int hardness) { super(hardness); }
}
```

• Shield and Armor thought of first... then superclass MetalThing to encapsulate the idea of hardness that was starting to be seen as a recurring theme.

• the toString method came to mind only when trying to debug it
then swords came to mind, which I knew would involve instances of the Enemy class

```java
class Sword extends MetalThing {
    protected int length;

    Sword(int length, int hardness) {
        super(hardness);
        this.length = length;
    }

    public void stab(Warrior w) { w.beStabbed(this); }
    public String toString() { return super.toString() + "Length = " + length + " "; }
}

class BattleSword extends Sword {
    protected int width;

    BattleSword(int l, int bladeWidth, int hard) {
        super(l, hard);
        width = bladeWidth;
    }
    public String toString() { return super.toString() + "Width = " + width + " "; }
}
```

CREATING A HIERARCHY OUT OF THIN AIR
class Warrior {
    final int FULL_OF_LIFE = 100;
    final int MINIMUM_LIFE = 10;
    final int SWORD_LENGTH = 2;

    String name;
    Shield shield;
    Armor armor;
    Sword sword;
    int lifeEnergy = FULL_OF_LIFE;

    Warrior(int hardness, String name) {
        this.name = name;
        shield      = new Shield(hardness);
        armor       = new Armor(hardness);
        sword       = new Sword(SWORD_LENGTH, hardness);
        System.err.println(name + " created with: ");
        System.err.println("\tshield:" + shield.toString());
        System.err.println("\tsword:" + sword.toString());
        System.err.println("\tarmor:" + armor.toString());
    }
}
public void stab(Warrior w) { w.beStabbed(sword); }

public void beStabbed(Sword s) {
    if (armor.hardness < s.hardness) {
        System.out.println(name + ": stabbed by a " + s.length + "-foot long sword!");
        lifeEnergy /= 10;
        if (lifeEnergy < MINIMUM_LIFE)
            System.out.println(name + ": farewell! <sniff> ");
    }
    else
        System.out.println(name + ": Ha! Can't puncture me!");
}
CREATING A HIERARCHY OUT OF THIN AIR

- and a simple driver

```java
class Test10 {
    public static void main (String[] args) {
        final int SOFT = 20;
        final int HARD = 100;

        Warrior Gardoz = new Warrior(SOFT, "Gardoz");
        Warrior Zormox = new Warrior(HARD, "Zormox");

        Gardoz.stab(Zormox);
        Zormox.stab(Gardoz);
        Zormox.stab(Gardoz);
    }
}
```
>>java Test
Gardoz created with:
shield: Hardness = 20
sword: Hardness = 20 Length = 2
armor: Hardness = 20
Zormox created with:
shield: Hardness = 100
sword: Hardness = 100 Length = 2
armor: Hardness = 100
Zormox: Ha! Can't puncture me!
Gardoz: stabbed by a 2-foot long sword!
Gardoz: stabbed by a 2-foot long sword!
Gardoz: farewell! <sniff>
SOME REVIEW OF SCOPE OF VARIABLES

• scope of variables ... visibility of a given variable

• variables as pronouns, and what they may refer to
  at compile time
  at run time

• setters and getters, and strict encapsulation of object attributes

• how can a class be deriving abilities (methods) without inheritance?
void Blah {
    int x;

    if ... {
        int x; // this x shadows outer x
        ...
    }
}

void Blah {
    int x;

    if ... {
        x = 0; // this x same as outer x
        ...
    }
}

- looking outward to increasingly larger contexts
class A {
    public int x = 1;
    public void y() { ...println("A") }
}

class B extends A {
    public int x = 2;
    public void y() { ...println("B") }
}

in constructing a B, think of first constructing an instance of A to lay foundation then extensions of B overlaying any counterparts in A

x and y in B overlay (shadow) original versions in A ... but caution (stay tuned)

incidentally, from a design point of view:
why would you use same name y() in class B?
why would you use same name x in class B?
class A {
    public int x = 1;
    public void y() { ...println("A") }
}

class B extends A {
    public int x = 2;
    public void y() { ...println("B") }
}

A a = new A();
B b = new B();
A c = new B();

a.x == ? what does a.y() print?
b.x == ? what does b.y() print?
c.x == ? what does c.y() print?

a.x == 1   a.y() prints A
b.x == 2   b.y() prints B
c.x == 1   c.y() prints B

why???
class MetalThing {
    public int hardness = 1;

    MetalThing() {}
    public int getHardness() { return hardness; }
}

class Armor extends MetalThing {
    public int hardness = 2;

    Armor() {}
    public int getHardness() { return hardness; }
}
class Test {
    public static void main (String[] args) {
        MetalThing a = new MetalThing();
        Armor b = new Armor();
        MetalThing c = b;

        System.out.print("a.hardness = " + a.hardness);
        System.out.println("a.getHardness = " + a.getHardness());

        System.out.print("b.hardness = " + b.hardness);
        System.out.println("b.getHardness = " + b.getHardness());

        System.out.print("c.hardness = " + c.hardness);
        System.out.println("c.getHardness = " + c.getHardness());
    }
}

>>java Test
a.hardness = 1  a.getHardness() = 1
b.hardness = 2  b.getHardness() = 2
c.hardness = 1  c.getHardness() = 2
abstract class MetalThing {
    private final int hardness;

    MetalThing(int hardness) { this.hardness = hardness; }

    public int getHardness() { return hardness; }

    public boolean harderThan(MetalThing otherMetalThing) {
        return hardness > otherMetalThing.getHardness();
    }
}

abstract class Armor extends MetalThing {
    protected boolean ornate;

    Armor(int hardness, boolean ornate) {
        super(hardness);
        this.ornate = ornate;
    }

    public boolean canPenetrate(MetalThing target) {
        return this.harderThan(target);
    }
}
class Helmet extends Armor {
    private int size;
    private boolean visorUp;

    Helmet(int hatSize, int hardness, boolean ornate) {
        super(hardness, ornate);
        size = hatSize;
        setVisorUp(true);
    }

    public void setVisorUp(boolean up) {
        visorUp = up;
    }

    public boolean getVisorUp() {
        return visorUp;
    }

    public int getSize() {
        return size;
    }

    public boolean getOrnate() {
        return ornate;
    }

    public String toString() {
        return (super.toString() + "size = " + getSize() + " " +
                "ornate = " + getOrnate() + " " + "visor up = " + getVisorUp());
    }
}
ENCAPSULATION AND SETTERS/GETTERS

class Sword extends MetalThing {
    private int length;

    Sword(int length, int hardness) {
        super(hardness);
        this.length = length;
    }

    public int getLength() { return length; }
    public void stab(Warrior w) { w.beStabbed(this); }
    public String toString() { return super.toString() + "Length = " + getLength() + ", " ; }
}

note how Sword has the ability to call a method in some Warrior

in CRC lingo, Warrior is a collaborator of Sword (some collaborator, huh?)
how about being able to stab anything that is stabbable?

Warrior, this, and maybe other kinds of player.

one solution:
A) make a new abstract superclass Player and have Warrior extend Player.
B) have Player class contain abstract beStabbed method
C) have Sword's stab method apply to any subclass of Player

public void stab(Player victim) { victim.beStabbed(this);}

and leave what happens up to the victim's specific class
but ...
but what about being able to have Sword stab a piece of fruit, the ground, empty air ??

Sword sword = new Sword(...);
Fruit apple = new Fruit("apple");
sword.stab(apple);

those classes don't extend Player

maybe make a new class Stabbable and have Player, apple, etc. extend that class ??

nope you can only play the inheritance game once per class
Java does not allow "multiple inheritance", so can't say:
Warrior extends Player, Stabbable { //illegal

but we can do the following ...
INHERITANCE VERSUS INTERFACES

• define an interface that describes what methods need to be implemented to account for the notion

```java
public interface Stabbable {
    abstract public void beStabbed(Sword s);
}
```

and require that any class you want to be able to stab implement that interface.

```java
class Apple implements Stabbable {
    public void beStabbed(Sword s) {
        System.out.println("squirt!");
    }
}
```
public interface Stabbable {
    abstract public void beStabbed(Sword s);
}

So now recall Warrior already has

public void beStabbed(Sword s) {
    if ( ... and etc.

which already happens to satisfy that interface ... so just

class Warrior implements Stabbable {
...

and now, with no further modifications to Warrior, this allows:

zormox.stab(gardoz); // resulting in “stabbed by a 2-foot long sword!”
zormox.stab(apple); // resulting in “squirt!”
create multiple interfaces, e.g.

interface Movable {
    Point3D getPosition();
    Vector3D getVelocity();
    Vector3D getHeading();
    void setPosition(Point3D point);
    void setVelocity(Vector3D velocity);
    void setHeading(Point3D heading);
}

interface Animate extends Movable {}
interface Inanimate extends Movable {}

then every sort of Animal can have the properties and attributes of being an Animate

class Animal implements Animate{}
class Vertebrate extends Animal{}
class Mammal extends Vertebrate{}````
if we also write

interface ThingWithMass {
    final int PLANKS_CONSTANT ...
    void setMass(int mass);
    void getMass(int mass);
    ... and etc.
}

an Animal can implement various interfaces simultaneously:

    class Animal implements Animate, ThingWithMass {}
    class Vertebrate extends Animal{}
    class Human extends Mammal
        implements largerThanBreadboxThing {}

that's a lot of expressive power