TOPICS

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?
SCOPE and DEFINITION of SHADOWING

```c
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
```
SHADOWING IN SUBCLASSES

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?
CASTING TO A SUPERCLASS

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("\ta.getHardness = " + a.getHardness());

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

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

A GOOD OO DESIGN, con't.
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

```java
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 ??

```java
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("squirter!");
    }
}
```
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.

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

```java
interface Animate extends Movable {}  
interface Inanimate extends Movable {}  
```

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

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

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

an Animal can implement various interfaces simultaneously:

```java
class Animal implements Animate, ThingWithMass {}

class Vertebrate extends Animal{}

class Human extends Mammal
    implements largerThanBreadboxThing {}
```

that's a lot of expressive power