Chapter 2

A Trip to Objectville

- If you put all of your code in main(), that's not object-oriented.
- What if I write some functions and call them in main()?
- Well, that improves readability but still not OO.
- Get the heck out of main(), and start making some **objects** of our own
- Let's start with a story between two engineers…

- Once upon a time in a software company, two programmers were given the same spec and told to “build it”
- Whoever delivers first gets one of those cool Aeron™ chairs

大材貨 Herman Miller Aeron 2.0人體工學椅 經典再進化(全功能)- A SIZE

◆提供更順暢的前傾功能(比以往多1.8度)
◆可調節的SL腰部支撐墊，貼近人體身型
◆全新網布，比以往更具彈性
◆美國OA第一品牌
◆重新組裝，表現更加出色
◆支稱多種位置及更廣泛的姿勢範圍
◆傾仰範圍內收放自如
◆減輕脊椎承受壓力
◆保持體感舒適與透氣
◆符合人體曲線，適應多種姿態
◆榮獲多項設計大獎

The Best Chair for Programmers... (as an ex-Google tech lead) | Aeron vs Embody, Steelcase, Hyken...

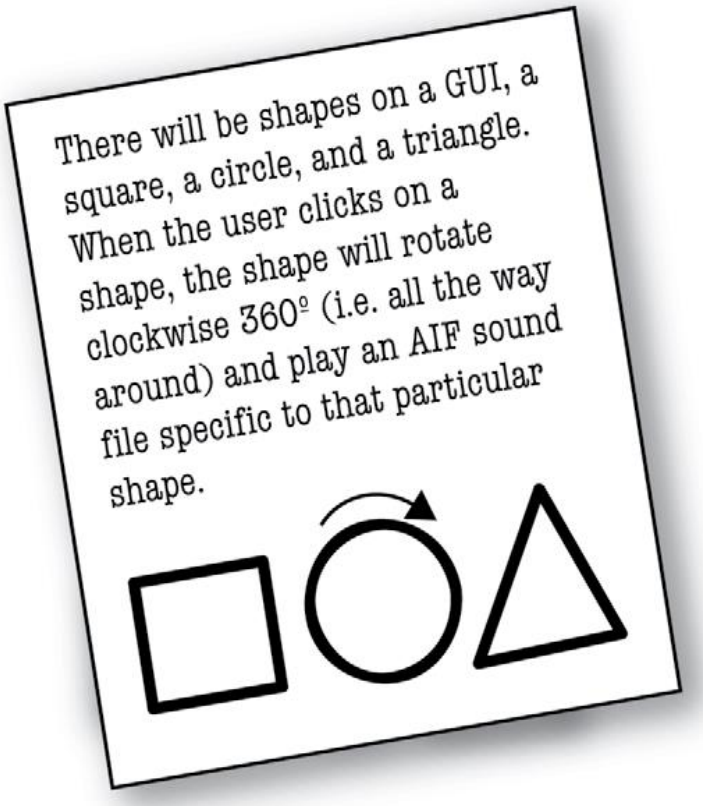
觀看次數：92萬次・1 年前

TechLead

Disclosure: Some links are affiliate links to products. I may receive a small commission for purchases made ...

10:51 ... the world famous **herman miller aeron** chair so why don't we take a look at what's going on with th...

- Larry, the procedural programmer, and Brad, the OO guy, both knew this would be a piece of cake.



- Larry's thought
  - What are the things this program has to **do**? What **procedures** do we need?
- Brad's thought
  - What are the **things** in this program… who are the key players?

- Suddenly, they receive a **spec change**.
- There will be an amoeba shape on the screen, with the others. When the user clicks on the amoeba, it will rotate like the others, but play another sound - a .hif sound file.

Back in Larry's cube

At Brad's laptop at the beach

- Then they receive **ANOTHER** spec change…
- The amoeba shape was supposed to rotate around a point on one end, like a clock hand
- *Do you honestly think the spec won't change again?*

Back in Larry's cube

At Brad's laptop at the beach

- Larry found a flaw in Brad's approach:

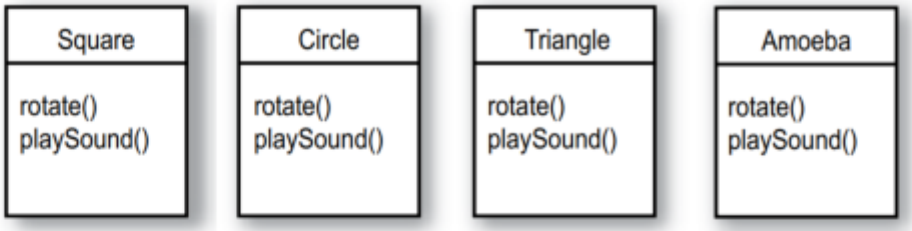
LARRY: You've got **duplicated** code! The *rotate* *procedure* is in all four Shape *things*.

BRAD: It's a **method**, not a procedure. And they're **classes**, not things.

LARRY: Whatever. It's a stupid design. You have to maintain **FOUR** different rotate "methods". How can that ever be good?

BRAD: Oh, I guess you didn't see the final design. Let me show you how OO **inheritance*** works, Larry

*Note: we'll introduce how to implement inheritance in chapter 7.

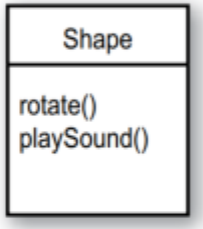


1

I looked at what all four classes have in common.

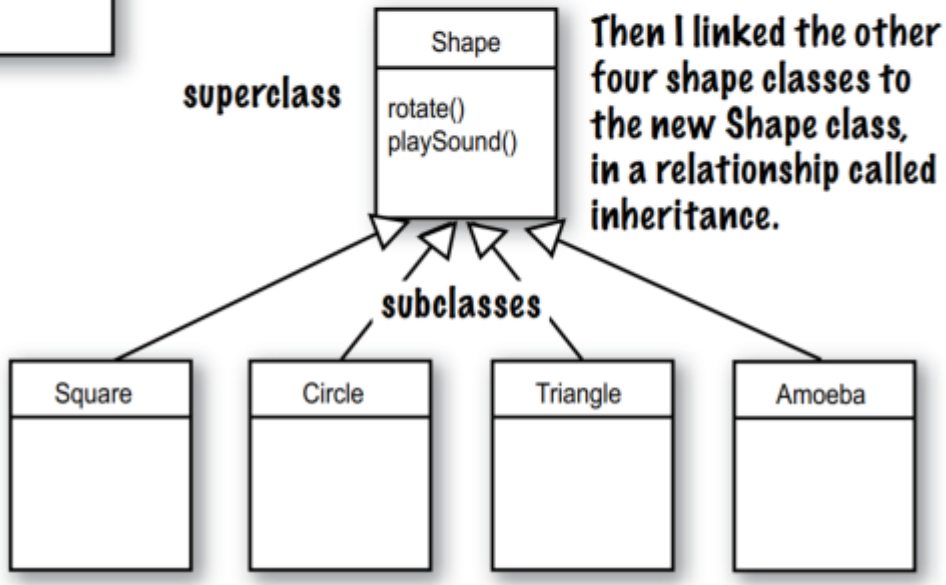
2

They're Shapes, and they all rotate and playSound. So I abstracted out the common features and put them into a new class called Shape.



3

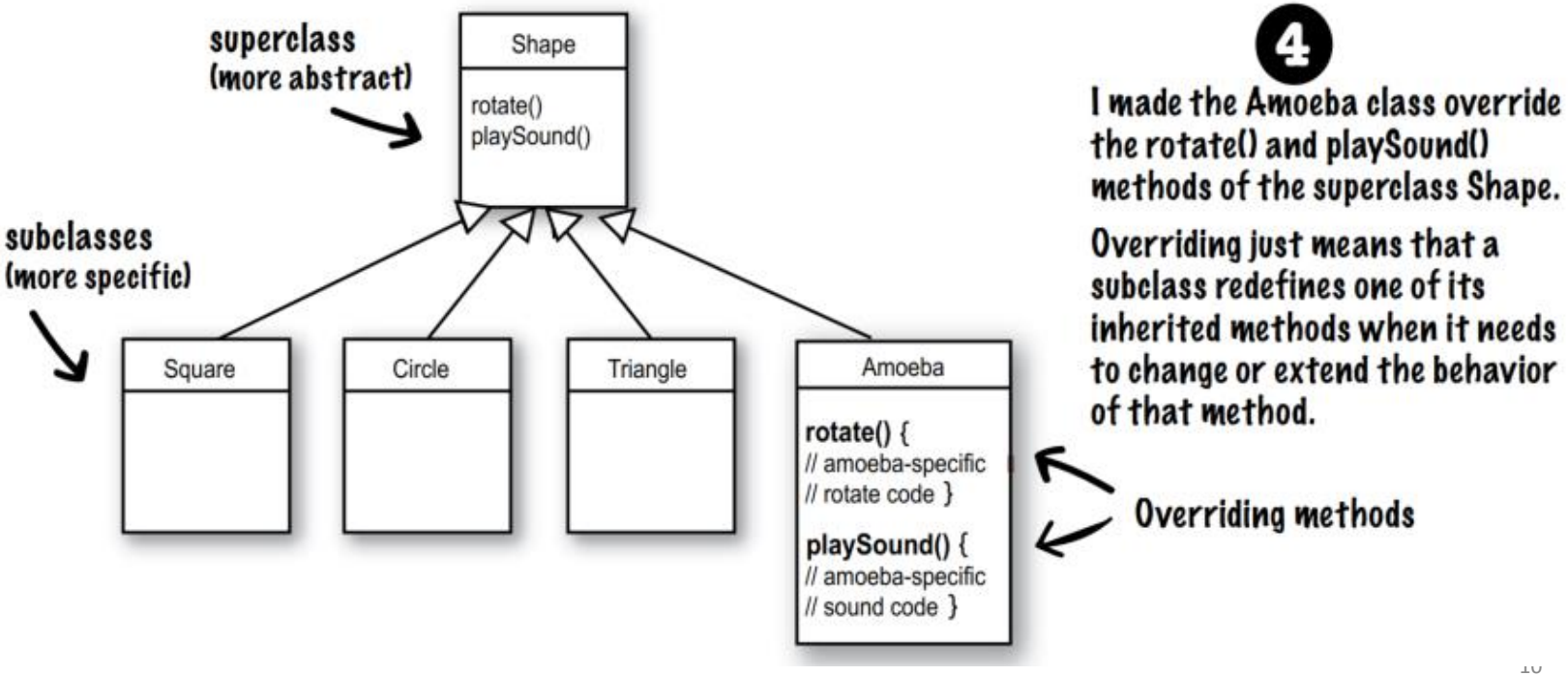
Then I linked the other four shape classes to the new Shape class, in a relationship called inheritance.



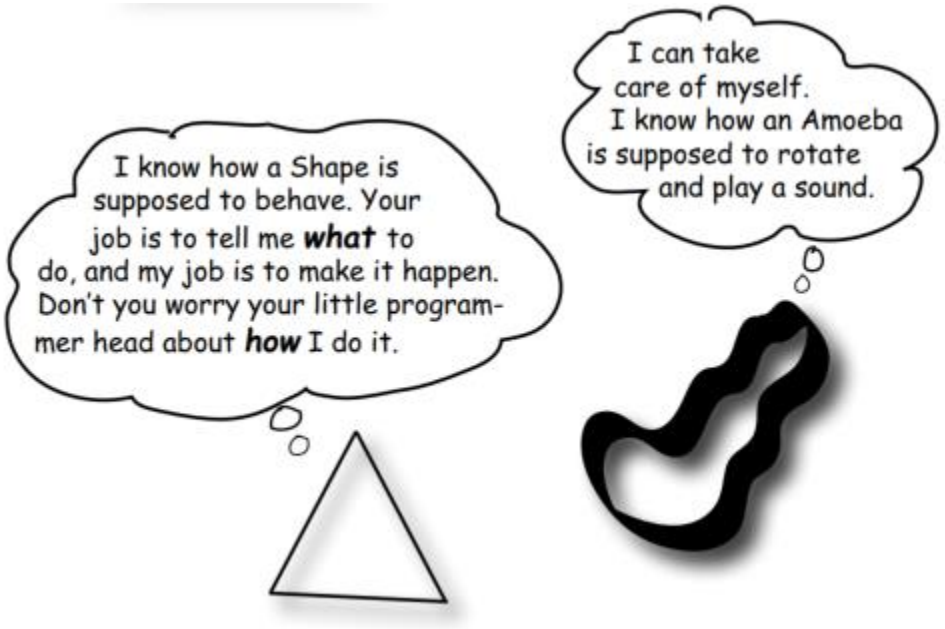
You can read this as, **"Square inherits from Shape"**, **"Circle inherits from Shape"**, and so on. I removed rotate() and playSound() from the other shapes, so now there's only one copy to maintain.

The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

- What about the Amoeba rotate() and playSound()?
- How can amoeba do something different if it “inherits” its functionality from the Shape class?
- The Amoeba class **overrides** the methods of the Shape class. Then at runtime, the JVM knows exactly which rotate() method to run when someone tells the Amoeba to rotate

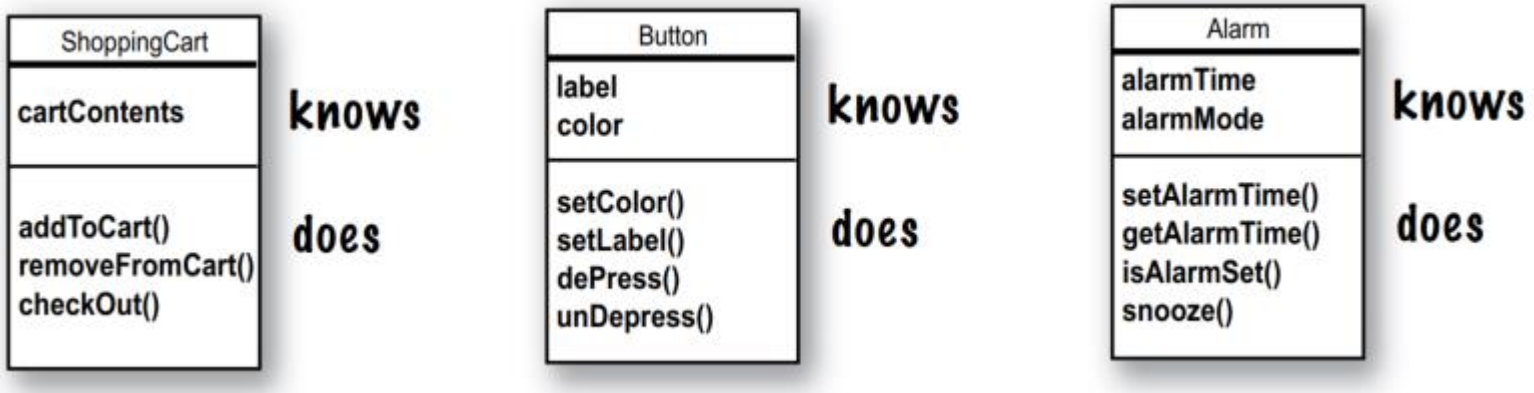


- When it's time for, say, the triangle to rotate, the program code calls the rotate() method on the triangle object. The rest of the program really doesn't know or care how the triangle does it.
- When you need to add something new to the program, you just write a new class for the new object type, so the **new objects will have their own behavior.**

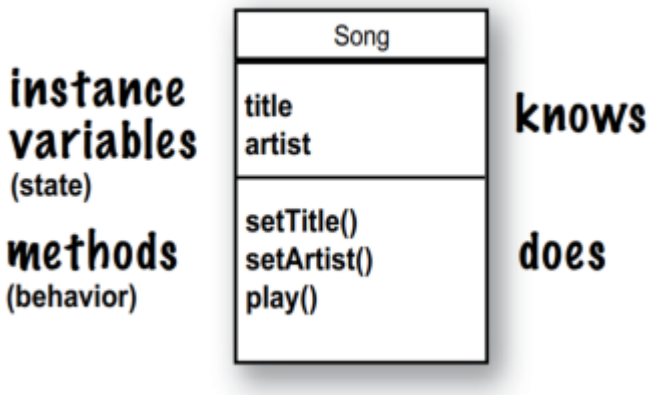


OOP spirit:
Extend your program ***without*** having to touch previously working code.

- When you design a class, think about the objects that will be created from that class type. Think about
- Things the object **knows**
- Things the object **does**



- Things an object **knows** about itself are called instance variables
- Things an object **can do** are called methods
- Instance is somewhat like another way of saying **object**
- When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data.



Sharpen your pencil

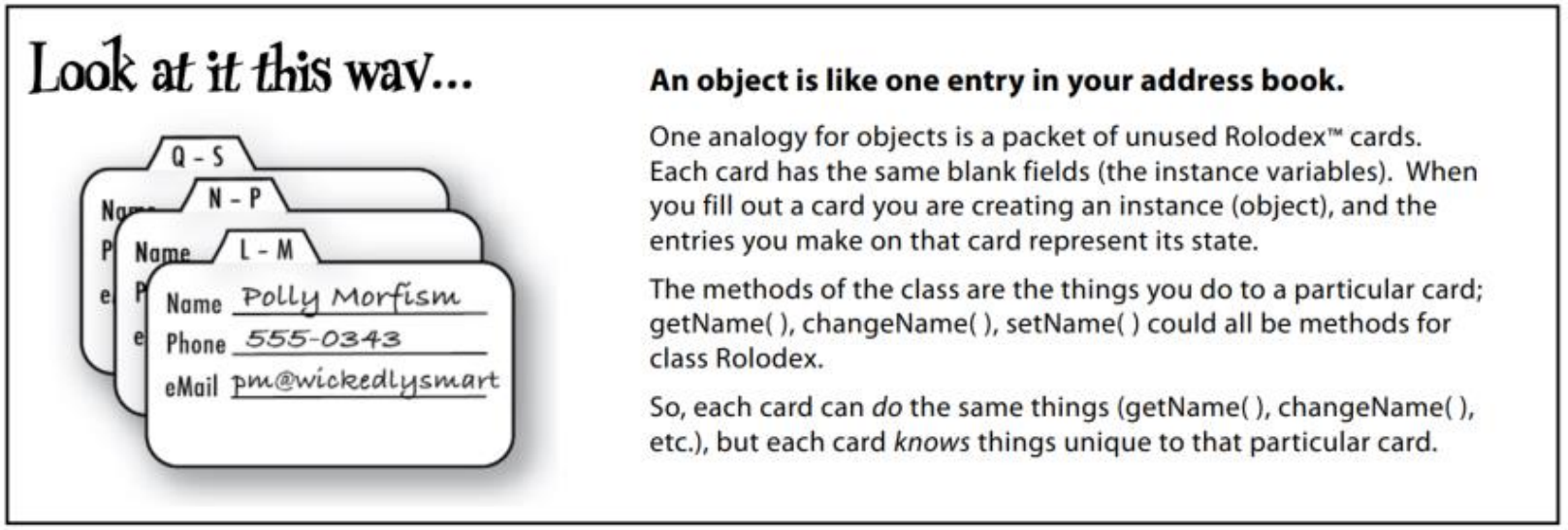
Fill in what a television object might need to know and do.

Television

instance variables

methods

- What's the difference between a **class** and an **object**?
- A class is a **blueprint** for an object
- It tells the virtual machine how to make an object of that particular type.
- Each object made from that class can have its own values for the instance variables of that class.
- An **instance** is an object created from a class.



- How to create and use an object?
- You need two classes. One class for the type of object you want to use (Dog, AlarmClock, Television, etc.) and another class to test your new class. The tester class is where you put the main method, and in that main() method you create and access objects of your new class type.

1

```
class Dog {

  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Ruff! Ruff!");
  }
}
```

instance variables

a method

2

```
class DogTestDrive {
  public static void main (String[] args) {
    Dog d = new Dog();
    d.size = 40;
    d.bark();
  }
}
```

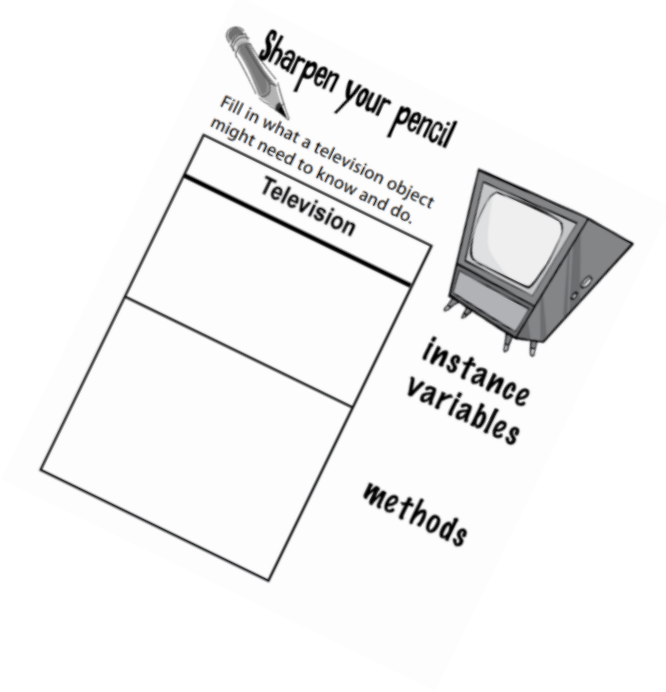
make a Dog object

use the dot operator (.) to set the size of the Dog

dot operator

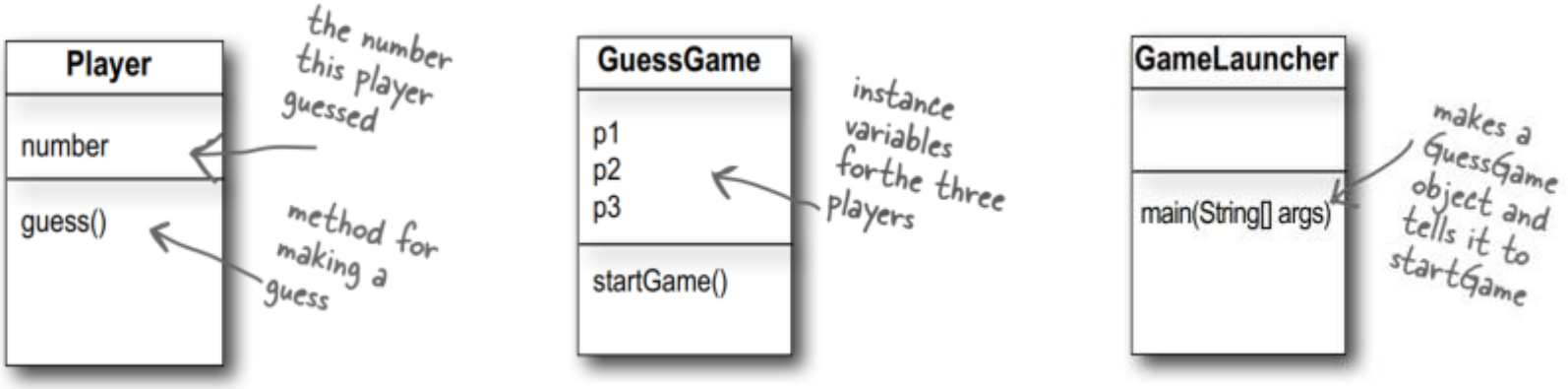
and to call its bark() method

- Exercise
- Write a Television class which is designed by your own in the previous few pages. You don't have to write any logic or initialize your variables at this stage.



- In a true OO application, you need objects **talking** to other objects, as opposed to a static main() method creating and testing objects.
- A real Java application is nothing but objects **talking** to other objects. In this case, **talking** means objects calling methods on one another.
- So, what's the usage of main()?
  - To **launch** your application
  - To **test** your program

- Example: design a guessing game
- The game generates a random number between 0 and 9, and **three** players try to guess it.
- How will you design this program?



- Java takes out the Garbage
- Each time an object is created in Java, it goes into an area of memory known as *The Heap*. All objects — no matter when, where, or how they're created — live on the heap.
- But it's not just any old memory heap; the Java heap is actually called the Garbage-Collectible Heap.

- When you create an object, Java allocates memory space on the heap according to how much that particular object needs. An object with, say, 15 instance variables, will probably need more space than an object with only two instance variables.
- What happens when you need to reclaim that space? Java manages that memory for you!
- When the JVM can ‘see’ that an object can never be used again, that object becomes eligible for garbage collection. And if you’re running low on memory, the Garbage Collector will run, throw out the unreachable objects, and free up the space, so that the space can be reused.

- Q: What if I need global variables and methods? How do I do that if everything has to go in a class?
- A: In later chapter, you'll learn marking a method as public and static makes it behave much like a 'global'
- Q: Then how is this object-oriented if you can still make global functions and global data?
- A: The static (global-like) things such as pi and random() in the math class are the exception rather than the rule in Java. They represent a very special case, where you don't have multiple instances/objects

- Q: What if I have a hundred classes? Or a thousand? Isn't that a big pain to deliver all those individual files? Can I bundle them into one Application Thing?
- A: You can put all of your application files into a Java Archive – a .jar file. In the jar file, you can include a simple text file formatted as something called a manifest, that defines which class in that jar holds the main() method that should run.

- Summary
- All java code is defined in a **class**
- A class describes how to make an object of that class type
- An object can take care of itself; you don't have to know or care how the object does it
- An object knows things and does things. Things an object knows about itself are called instance variables. Things an object does are called methods
- When you create a class, you should create a separate test class to test that class.

- Can these files be compiled?

A

```
class TapeDeck {

  boolean canRecord = false;

  void playTape() {
    System.out.println("tape playing");
  }

  void recordTape() {
    System.out.println("tape recording");
  }
}

class TapeDeckTestDrive {
  public static void main(String [] args) {

    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

B

```
class DVDPlayer {

  boolean canRecord = false;

  void recordDVD() {
    System.out.println("DVD recording");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {

    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();

    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```

Who am I?

A bunch of Java components, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one of them, choose all for whom that sentence can apply. Fill in the blanks next to the sentence with the names of one or more attendees. The first one's on us.

Tonight's attendees:

Class Method Object Instance variable

I am compiled from a .java file.	class
My instance variable values can be different from my buddy's values.	
I behave like a template.	
I like to do stuff.	
I can have many methods.	
I represent 'state'.	
I have behaviors.	
I am located in objects.	
I live on the heap.	
I am used to create object instances.	
My state can change.	
I declare methods.	
I can change at runtime.	