Objects

Lecture 2

Lecture objectives - be able to answer:

1. What are properties and actions (methods) of a software object
2. What is the difference between object class and object instance
3. How to create objects in java
4. How to make an object to perform action in java

What are Software Objects?

- Building blocks of software systems
  - a program is a collection of interacting objects
  - objects cooperate to complete a task
  - to do this, they communicate by sending “messages” to each other (and themselves!)

What can we model as objects

- Objects model *tangible* things
  - a school
  - a car
  - ______________________________
  - ______________________________

- Objects model *conceptual things*
  - a meeting
  - a date

- Objects model *processes*
  - finding a path through a maze
  - sorting a deck of cards

Software objects have

- ***capabilities***: what they can do, how they behave
- ***properties***: features that describe them

Object Capabilities: Methods

- Objects' *capabilities* allow them to perform specific actions
  - objects are smart—they "know" how to do things
  - an object gets something done only if it is told to use one of its capabilities -- by another object or itself
- *Capabilities* are also called *behaviours,* or *methods*

Capabilities can be:

- constructors: establish initial state of object’s properties
- commands: change object’s properties
- queries: provide responses based on object’s properties

For example: *trash cans* are capable of initiating, and responding to, certain actions

- constructor: be created
- commands: add trash, empty yourself
- queries: reply whether lid is open or closed, or how much trash is in can

Object Properties

Properties make an object unique

- affect way objects perform actions
- some properties are constant, others variable
- properties themselves are objects — they also can receive messages; e.g., trash can’s lid

Properties can be:

- attributes: things that help describe an object
- components: things that are "part of" an object
- associations: things an object knows about, but are not part of that object

Example: properties of *trash cans*

- attributes: color, size, material
- components: lid, trash bag, trash
- associations: a *trash can* can be associated with the room it's in

Object Properties: State

State: collection of all of an object’s properties; changes if any property changes

- some properties don’t change, e.g., the year a car was made
- others do, e.g., car’s color

$\longrightarrow$

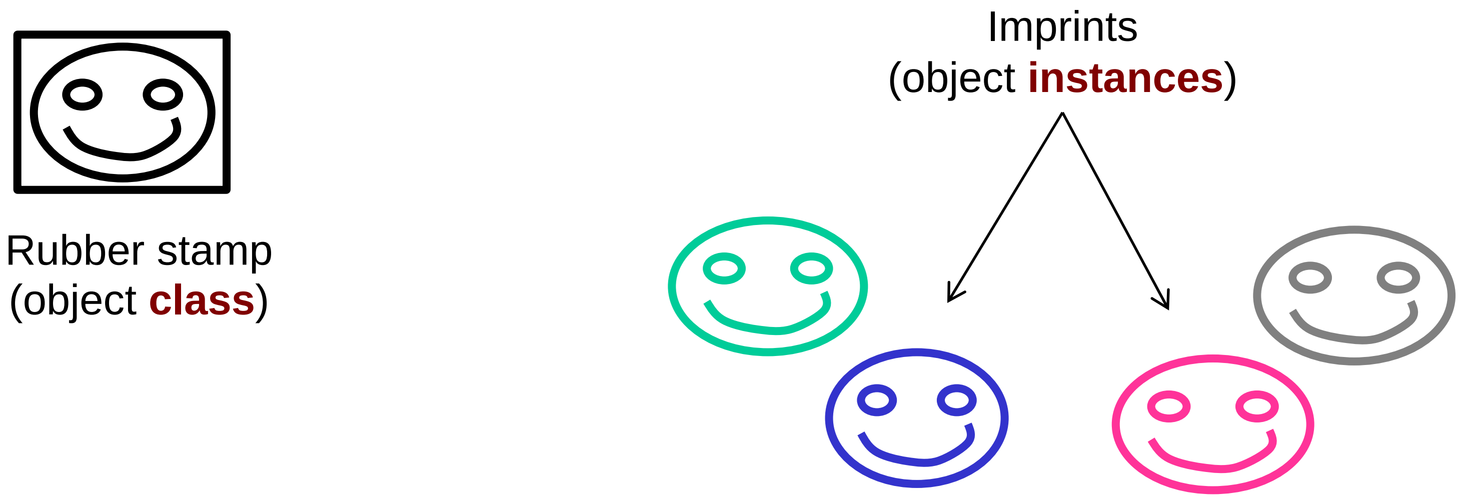
Object types

- Our current concept: each object corresponds directly to a *particular* real-life object, e.g., a specific atom or automobile

- Disadvantage: it’s much too impractical to work with objects this way
  - there may be arbitrarily many objects (e.g., modeling all atoms in the universe)
  - may not want to describe each individual object separately; they may have much in common – belong to the same *type*

Classes and Instances

- Classifying objects factors out commonality among sets of similar objects
  - describe what is common just once
  - then “stamp out” any number of copies later

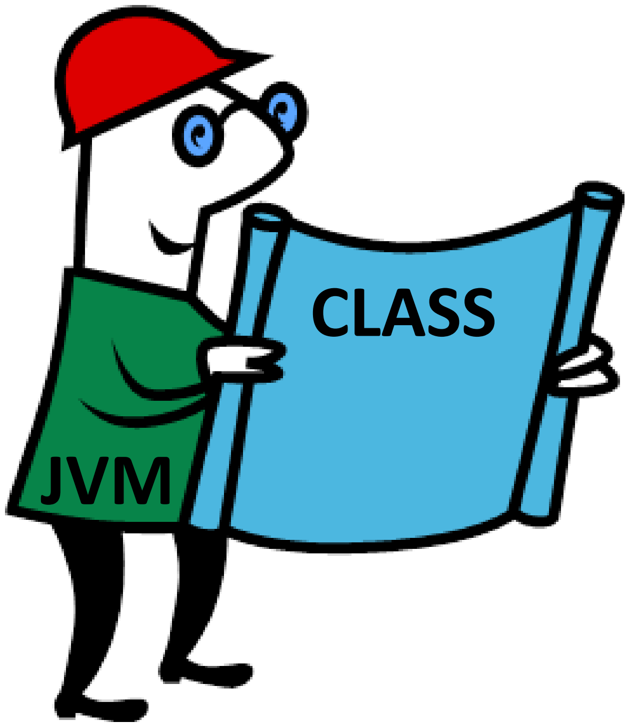


Object: class and instances

Object Class

- a class is a **type** of object

- defines **capabilities** and
  properties <u>common</u>
  among a set of individual objects
  - all *trash cans* can open, close,
  empty their trash



- defines **template** for making *object instances*
  - particular *trash can* instances may have a metal casing, be blue, be a certain size, etc.

Object Class

Classes implement capabilities as ***methods***

- A ***method*** is a sequence of statements in Java
- The sender sends a message to the receiver by calling one of the receiver's methods
  - a car object may send a message to an engine object to tell it to turn on

Classes implement properties as ***instance variables***

- slot of memory allocated to the object that can hold a potentially changeable value
  - e.g., GPA of a particular student

Object Instances

- *Object instances* are individual objects
  - made from class template
  - can have many *object instances* of the same class
  - all will have the same attributes, but may have different values for them
  - e.g. two *object instances* of Car will both have a color, but one may be red and one may be blue
  - the process of creating an *object instance* is called ***instantiating*** an object

- Different instances of the **`TrashCan`** class may have:
  - different color and position
  - different types of trash inside

- So their *instance variables* have different values
  - However, each is still a **`TrashCan`**

Instance identity

- Individual instances have individual ***identities***
  - this allows other objects to send messages to given

Variable must have a **type**

  - each instance is unique because of the values given to its properties, even though they all have the same capab

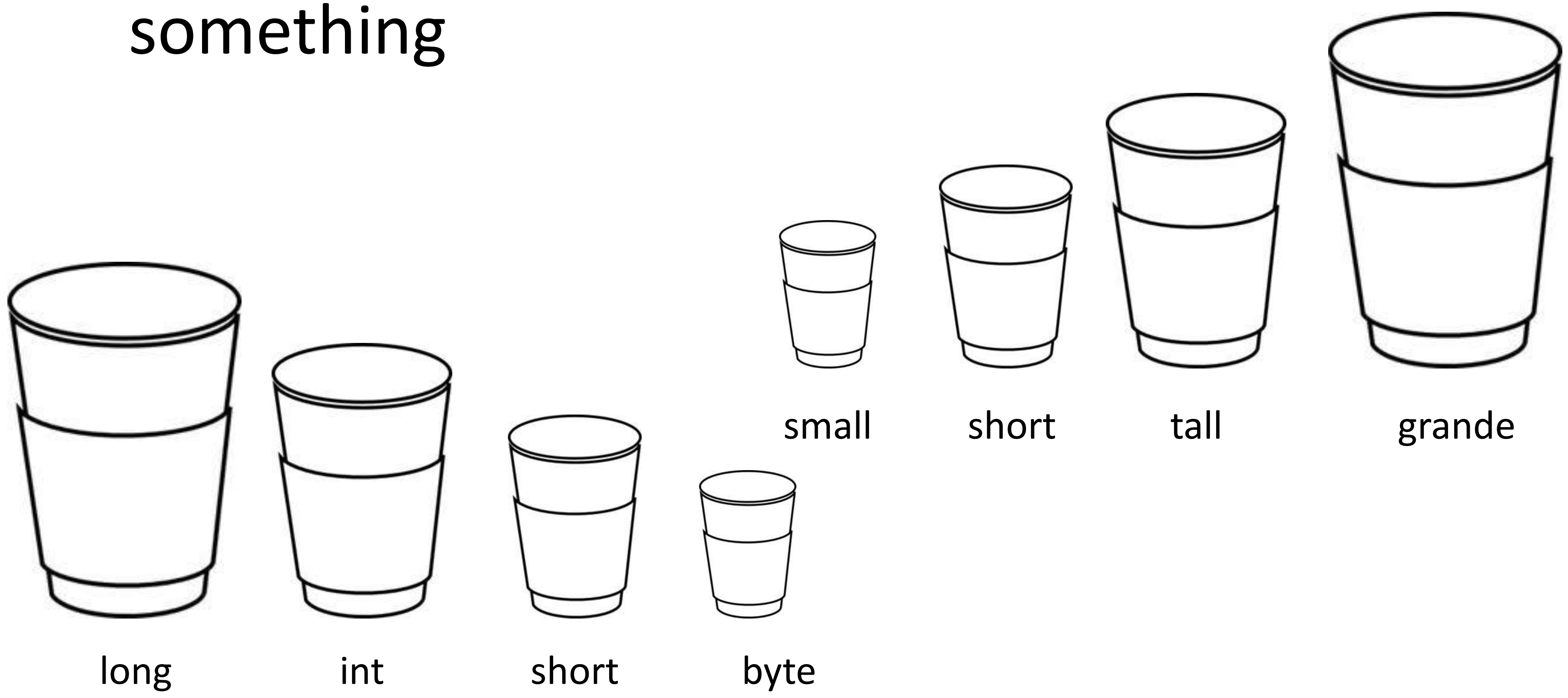
Variable must have a **name**

  - think of the CSCI-331 student instances in our class

- We will assign an instance to a ***variable***. Java is a typed language, so we need to declare the variable of a specific **type**
- Classes that we create just extend existing type system with new types

Variables

- Variable is just a cup. A container. It holds something

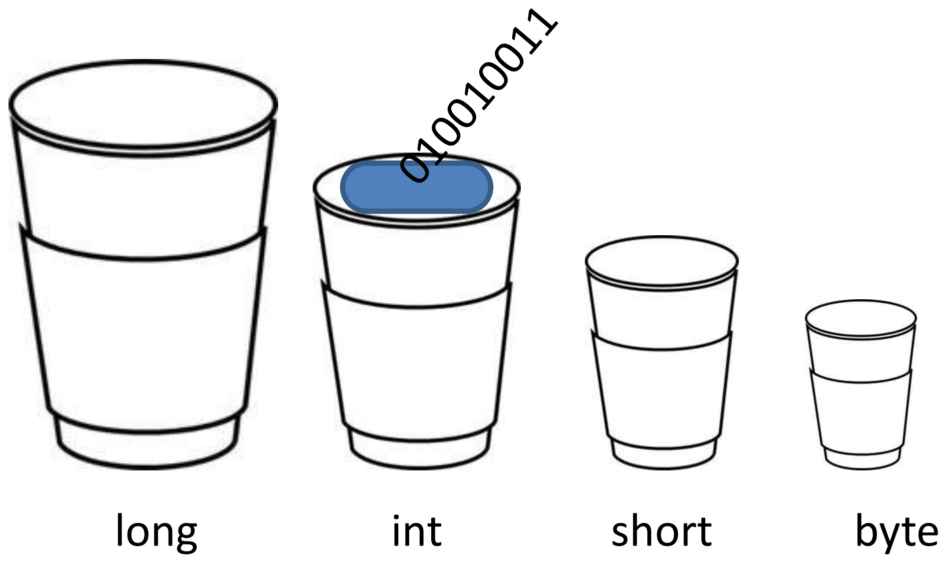


Java integer primitive data types

Variables for primitive data types

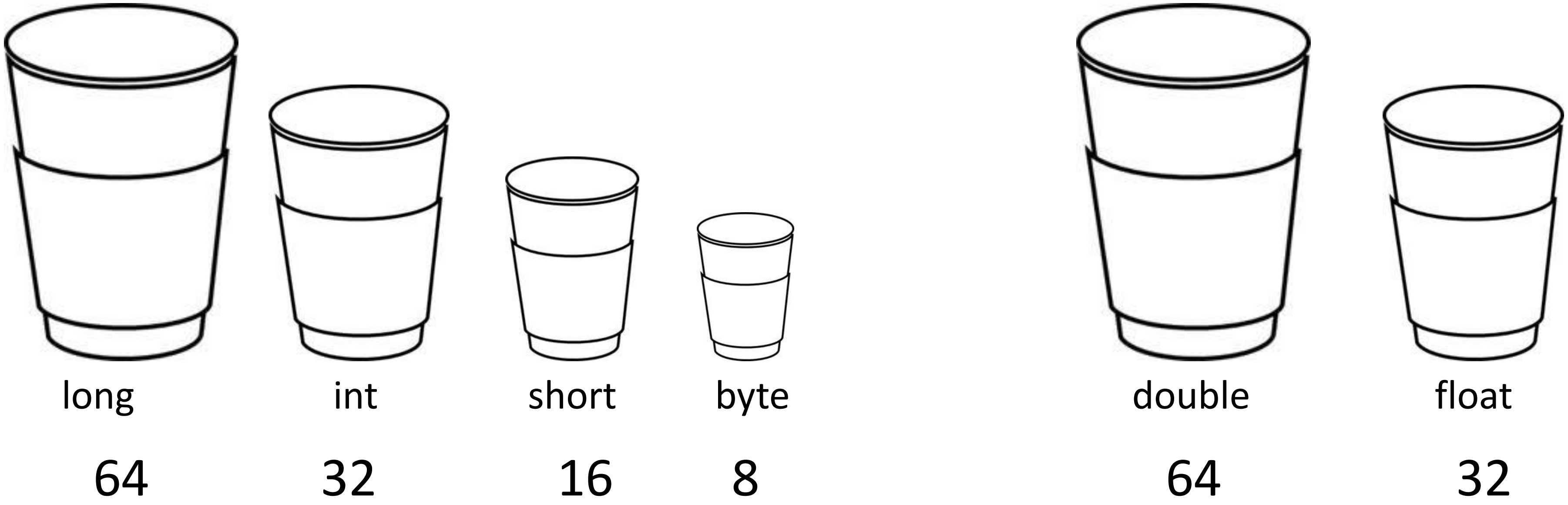
- You say to the compiler: “I’d like an int variable please with the value 90, and name the variable *height*”

int height=90;



Java integer primitive data types

Size of numeric variables



Java numeric primitive data types – size in bits

Compiler prevents spilling

```
1.   int x = 34.5;
2.   boolean boo = x;
3.   int g = 17;
4.   int y = g;
5.   y = y + 10;
6.   short s;
7.   s = y;
8.   byte b = 3;
9.   byte v = b;
10.  short n = 12;
11.  v = n;
12.  byte k = 128;
```

What statements are legal when written in this order?

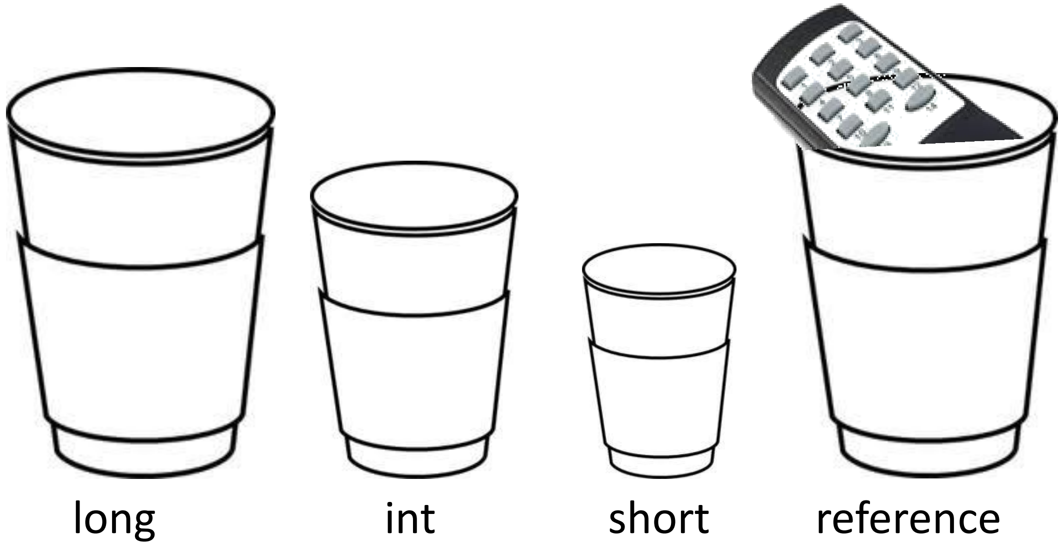
Variables to store objects

- There is actually no such thing as an object variable. You can't stuff objects of different sizes into a cup
- There's only an object reference variable. An object reference variable holds bits that represent a way to access an object.
- It doesn't hold the object itself, but it holds something like a pointer. Or an address. Except, in Java we don't really know what is inside a reference variable. We do know that whatever it is, it represents one and only one object. And the JVM knows how to use the reference to get to the object.

Reference variable

Dog myDog;

– reference variable of type dog. Does not reference any instance, has value *null*, and cannot call any methods of Dog class



WAG
BARK
EAT
PLAY
FETCH
DOG
Think of a Dog reference variable as a Dog remote control. You use it to get the object to do something (invoke methods).

When the instance is created

```
Dog myDog;
myDog=new Dog();
myDog.bark();
```

Now we can call the methods of class Dog

An instance of class Dog is created (somewhere),

and myDog is a reference to this instance

Where the instance is created

- Unlike C++, in Java all objects are created with a keyword *new*. This means that all objects live on the heap, not in the stack.

Object is created in the stack

C++

```
Dog myDog;
Dog * myDog=new Dog();
```

Java

```
Dog myDog;
Dog myDog=new Dog();
```

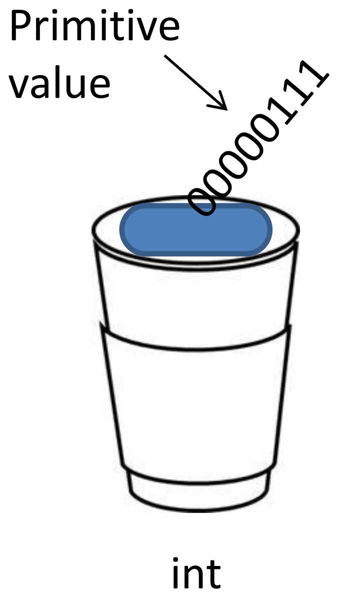
In Java, all primitive variables and the reference variables live in the stack, but all objects live on the heap

Reference and value

- An object reference is just another variable value.
- Something that goes into the cap.

Primitive variable:
int x=7;

The bits representing 7 go into the cap



Reference variable:
Dog d=new Dog();

The bits representing a way to get to the Dog object go into the cap

Dog object
reference

Reference and value

- An object reference is just another variable value.

Reference variable:

Dog d=new Dog();

With primitive variables, the value of the variable is... the *value* (5, -26.7, ‘a’).

With reference variables, the value of the variable is... *bits representing a way to get to a specific object.*

You don’t know (or care) how any particular JVM implements object references. Even if you *know*, you still can’t use the bits for anything other than accessing an object.

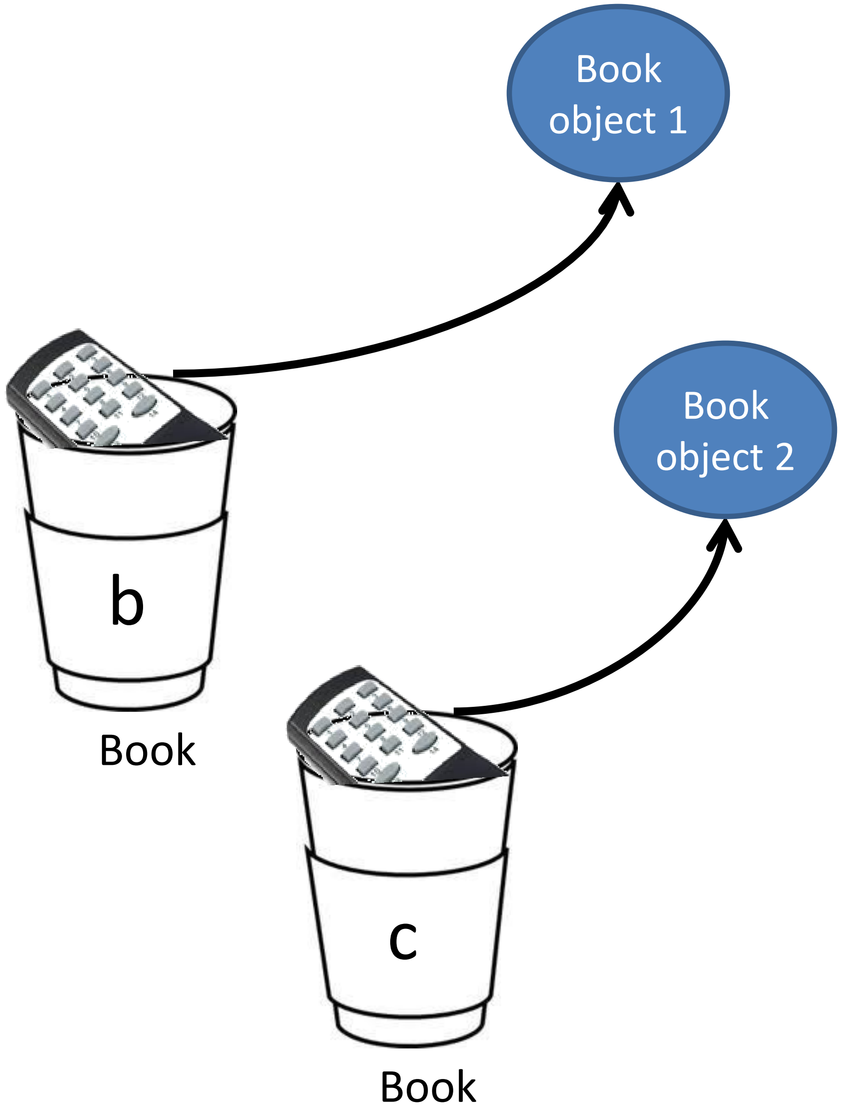
Dog object
reference

Assigning references I

- Book b=new Book();
- Book c=new Book();

References: 2

Objects: 2

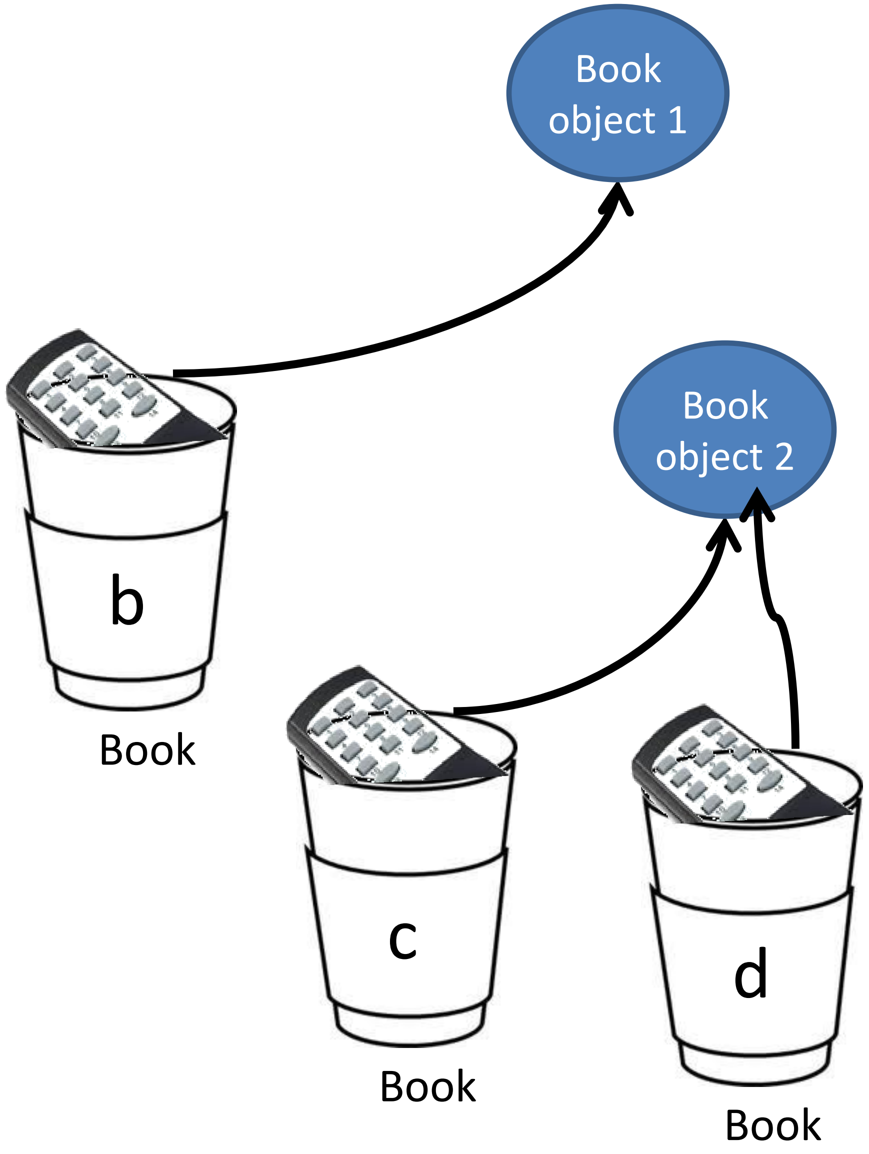


Assigning references II

- Book b=new Book();
- Book c=new Book();
- Book d=c;

References: 3

Objects: 2

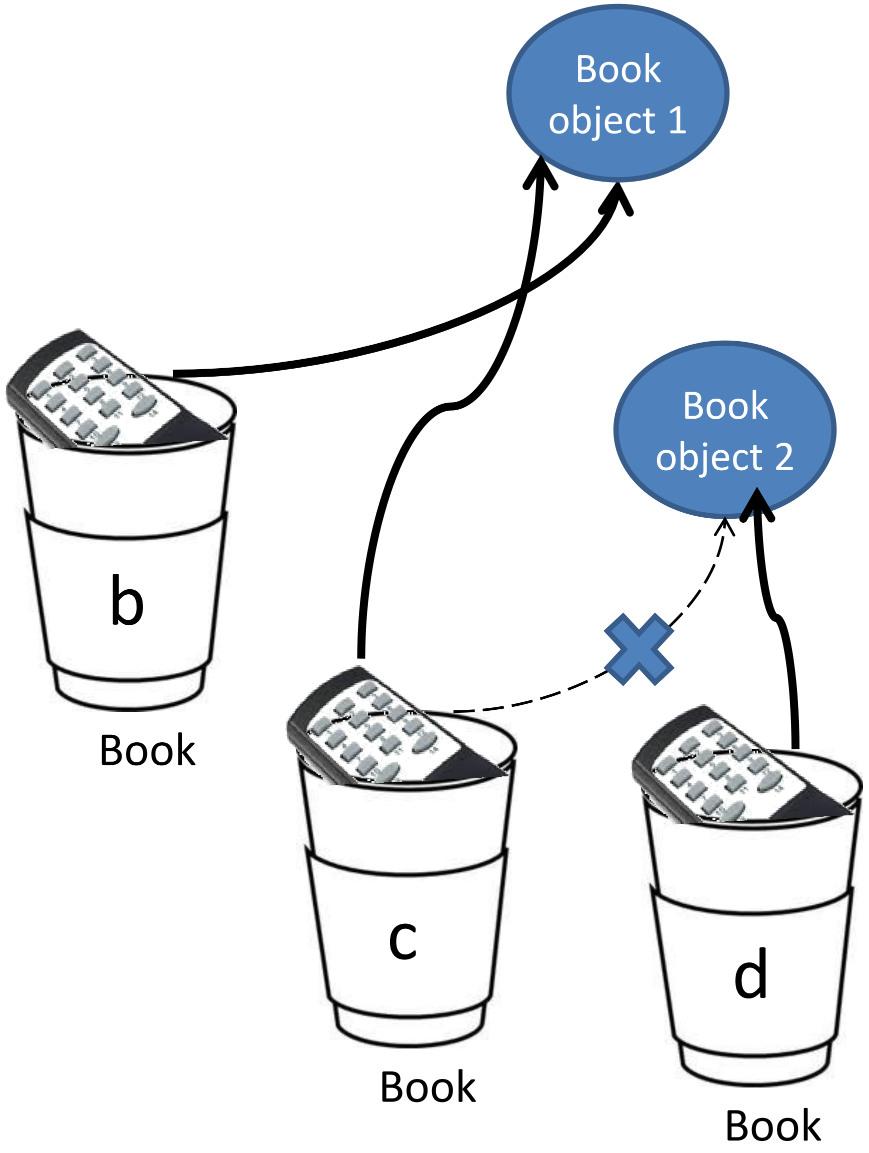


Assigning references III

- Book b=new Book();
- Book c=new Book();
- Book d=c;
- c=b;

References: 3

Objects: 2



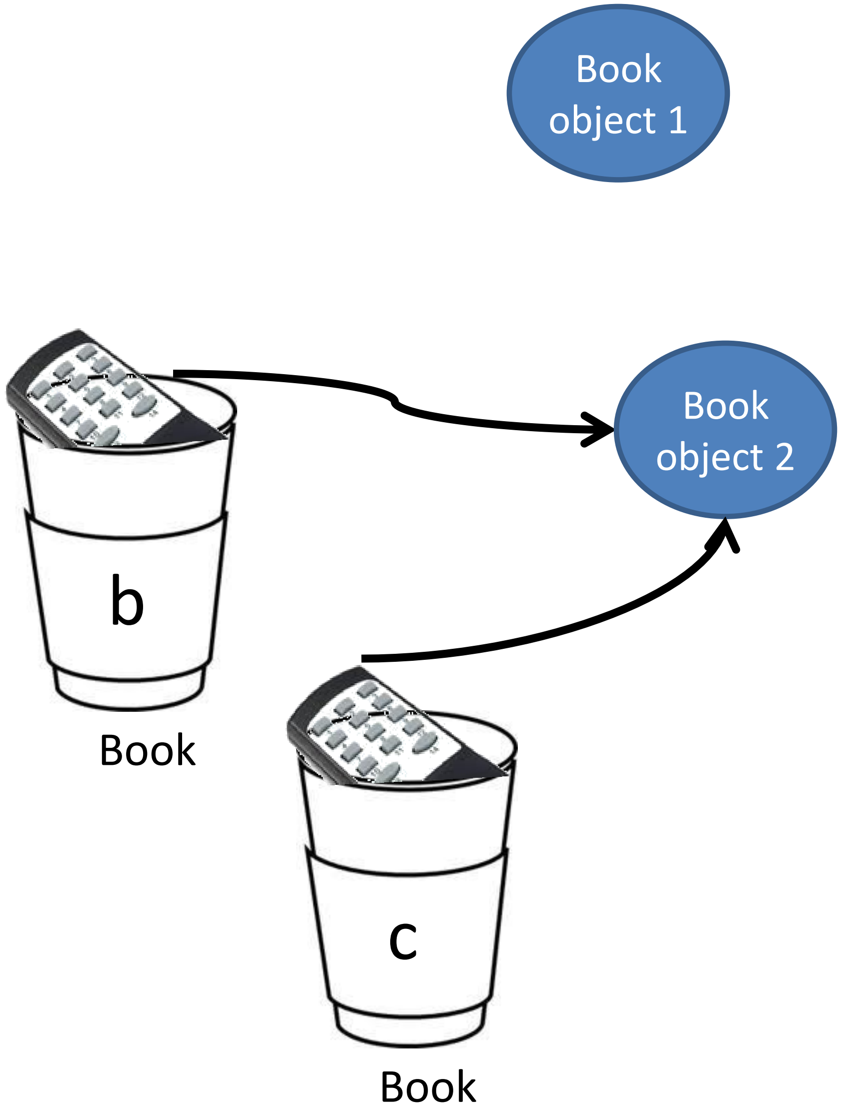
Assigning references IV

- Book b=new Book();
- Book c=new Book();
- b=c;

References: 2

Reachable Objects: 1

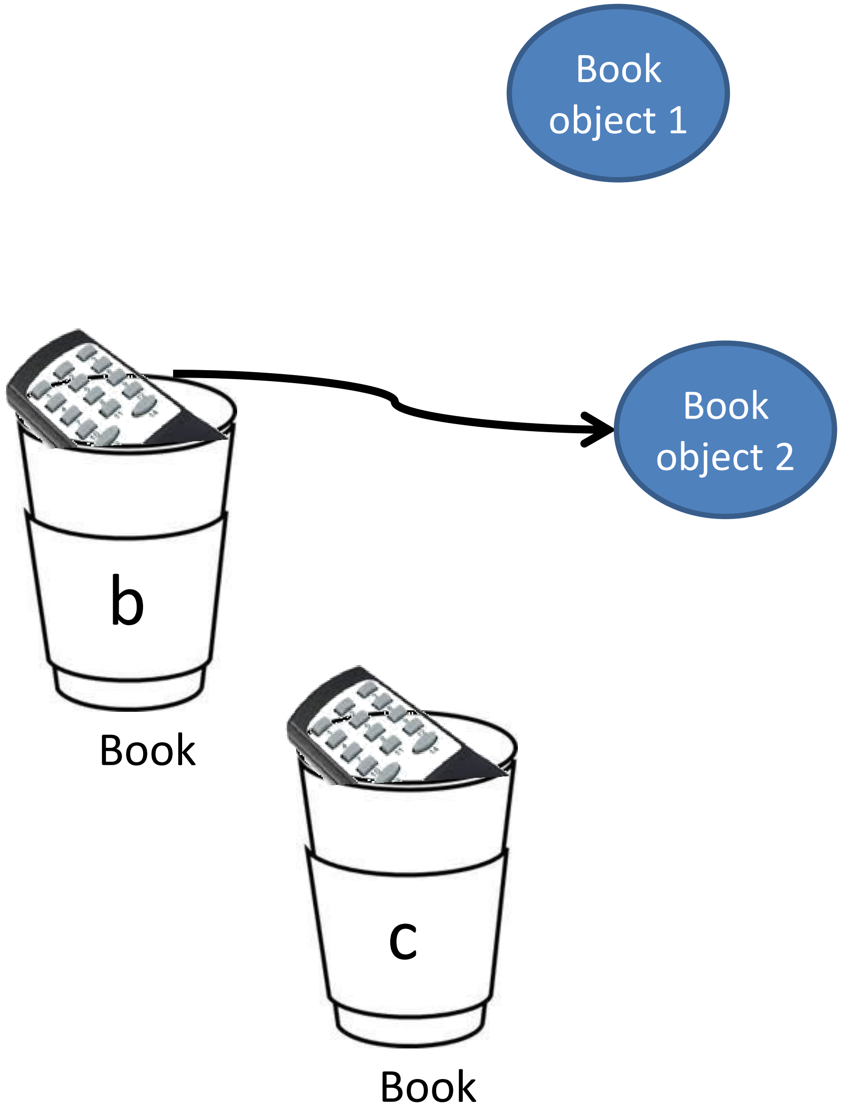
Abandoned objects: 1



Assigning references V

- Book b=new Book();
- Book c=new Book();
- b=c;
- c=null;

Active References: 1
Null references: 1
Reachable Objects: 1
Abandoned objects: 1



Digression: Memory

- **Registers**. Fast, inside the processor. Very small, Out of our control.
- **The stack**. RAM, but direct support from the processor via its *stack pointer*. An extremely fast and efficient way to allocate storage. While some Java storage exists on the stack—in particular, object references—Java objects themselves are not placed on the stack.
- **The heap.** RAM, a general-purpose pool of memory where all Java objects live. Flexibility – since is allocated in a run time. It takes more time to allocate heap storage than it does to allocate stack storage (if you even *could* create objects on the stack in Java, as you can in C++).
- **Static storage**. "Static" is used here in the sense of "in a fixed location".
- **Constant storage**. Constant values are often placed directly in the program code.
- **Non-RAM storage**. The examples are *streamed objects,* and *persistent objects*

Object allocation on stack and heap: C++

- C++ takes the approach that control of efficiency is the most important issue, so it gives the programmer a choice:
  - For maximum run-time speed, the storage and lifetime can be determined while the program is being written, by placing the objects on the stack (these are sometimes called *automatic* or *scoped* variables).
  - This places a priority on the speed of storage allocation
  - However, you sacrifice flexibility because you must know the exact quantity, lifetime, and type of objects while you're writing the program.
  - If you are trying to solve a more general problem such as computer-aided design, warehouse management, or air-traffic control, this is too restrictive.

Object allocation on stack and heap: Java

- Java creates objects dynamically in a pool of memory called the heap.
  - In this approach, you don't know until run time how many objects you need, what their lifetime is, or what their exact type is. Those are determined at the spur of the moment while the program is running.
  - If you need a new object, you simply make it on the heap at the point that you need it.
  - Because the storage is managed dynamically, at run time, the amount of time required to allocate storage on the heap can be noticeably longer than the time to create storage on the stack.
  - The dynamic approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object.
  - In addition, the greater flexibility is essential to solve the general programming problem

The issue: lifetime of an object

- With languages that allow objects to be created on the stack, the compiler determines how long the object lasts and can automatically destroy it.
- However, if you create it on the heap the compiler has no knowledge of its lifetime.
- In a language like C++, you must determine programmatically when to destroy the object, which can lead to memory leaks if you don’t do it correctly.
- Java provides a feature called a *garbage collector* that automatically discovers when an object is no longer in use and destroys it.
- A garbage collector is much more convenient because it reduces the number of issues that you must track and the code you must write.
- More important, the garbage collector provides a much higher level of insurance against the insidious problem of memory leaks.

How expensive it is to allocate objects on the heap

- You can think of the C++ heap as a yard where each object needs to find a free place and then stake out its own piece of turf. This real estate can become abandoned sometime later and must be reused.
- In some JVMs, the Java heap is quite different; it’s more like a conveyor belt that moves forward every time you allocate a new object. This means that object storage allocation is remarkably rapid. The “heap pointer” is simply moved forward into virgin territory, so it’s effectively the same as C++’s stack allocation. (Of course, there’s a little extra overhead for bookkeeping, but it’s nothing like searching for storage.)

JVM tricks

- Now you might observe that the heap isn’t in fact a conveyor belt, and if you treat it that way, you’ll eventually start paging memory a lot (which is a big performance hit) and later run out.
- The trick is that the garbage collector steps in, and while it collects the garbage it compacts all the objects in the heap so that you’ve effectively moved the “heap pointer” closer to the beginning of the conveyor belt and farther away from a page fault.
- The garbage collector rearranges things and makes it possible for the high-speed, infinite-free-heap model to be used while allocating storage.

How does garbage collector work I

- A simple but slow garbage collection technique is called *reference counting*.
- The garbage collector moves through the entire list of objects, and when it finds one with a reference count of zero it releases that storage.
- The one drawback is that if objects circularly refer to each other they can have nonzero reference counts while still being garbage.
- It doesn’t seem to be used in any JVM implementation.

How does garbage collector work II

- Garbage collection is not based on reference counting.
- It is based on the idea that any nondead object must ultimately be traceable back to a reference that lives either on the stack or in static storage.
- Thus, if you start in the stack and walk through all the references, you'll find all the live objects.
- For each reference that you find, you must trace into the object that it points to and then follow all the references in *that* object, tracing into the objects they point to, etc., until you've moved through the entire web that originated with the reference on the stack.
- Note that there is no problem with detached self-referential groups—these are simply not found, and are therefore automatically garbage.

Garbage collector is not working in the background

- The garbage collection is *not* done in the background; instead, the program is stopped while the garbage collection occurs.
- In the Sun literature you'll find many references to garbage collection as a low-priority background process, but it turns out that the garbage collection was not implemented that way.
- Instead, the Sun garbage collector runs when memory got low. In addition, it requires that the program be stopped.
- If you want your object to be destroyed right away, call **object.gc();**

Summary: Life and death of an object

- Objects in Java are allocated only on the heap using keyword *new*
- Each variable of a non-primitive type is a reference variable, it is allocated in the stack (during compilation) and it stores the path to a real object which will be created in a run time.
- Abandoned (unreferenced) objects are cleared automatically by Java garbage collector
- Garbage collection is not guaranteed until the memory is running low

Messages for Object Communication

- No instance is an island — it must communicate with others to accomplish tasks
  - associations allow them to know about other objects

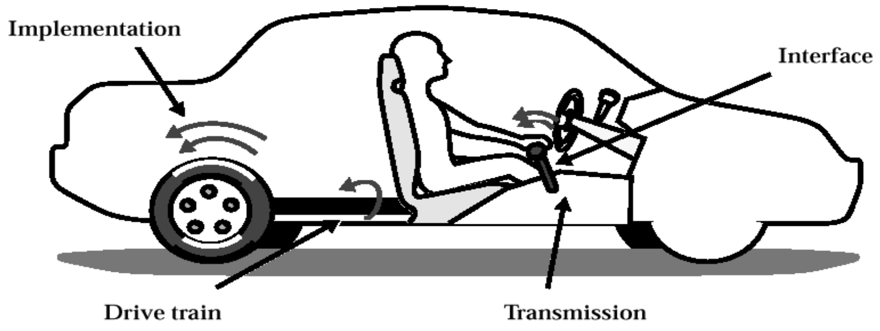
- Instances send messages to one another to invoke capabilities (i.e., to execute a task)
  - messages are sent via methods -- code that executes a task
  - each message invokes the corresponding method in the receiving object
  - when a human object sends a message to a car object to tell it to move, we say the human object *calls a method* on the car object

- Each message requires:
  - *sender*: object initiating action
  - *receiver*: instance whose method is being called
  - *message name*: name of method being called
  - optional *parameters*: extra info needed by method to operate (Parameters are also called arguments)
    - we'll discuss parameters in detail in a few lectures

- Receiver can (but does not need to) reply
  - we'll discuss *return types* in detail in the next lecture

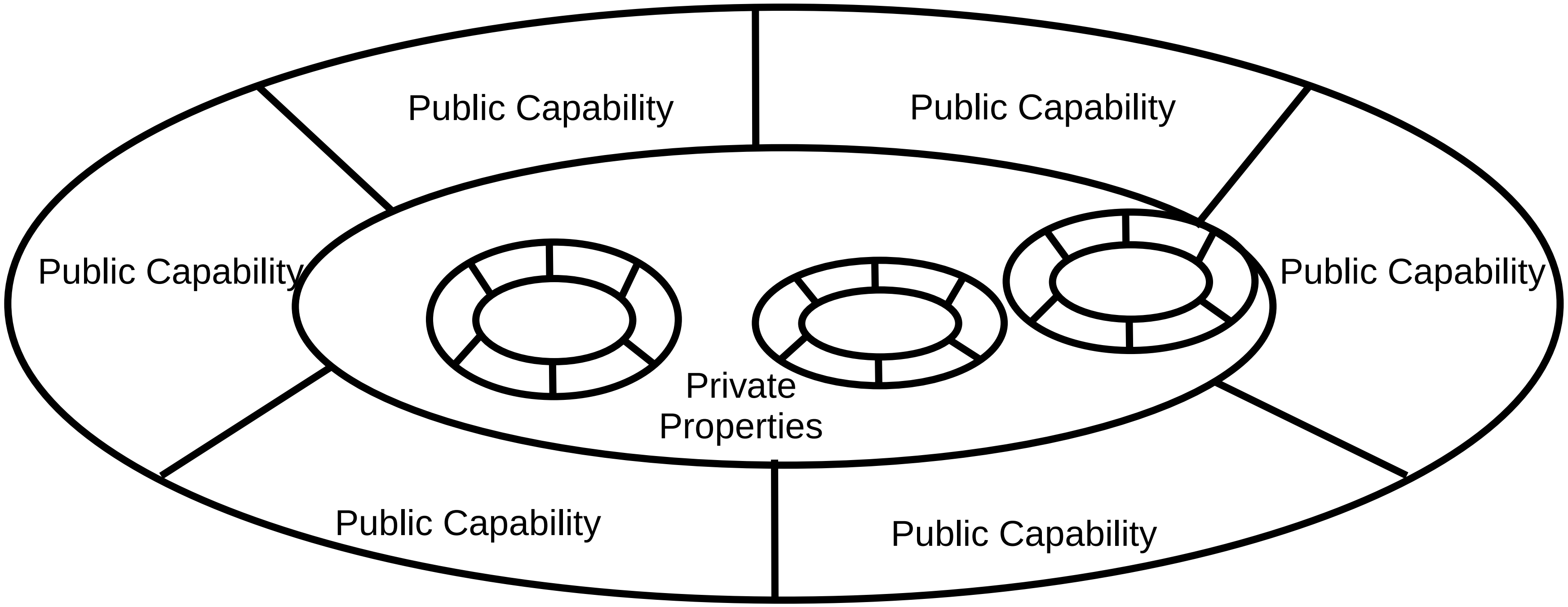
Encapsulation

- A car *encapsulates* lots of information
  - quite literally, under hood and behind dashboard
- So, you do not need to know how a car works in order to use it
  - steering wheel and gear shift are the interface
  - engine, transmission, drive train, wheels, . . . , are the (hidden) implementation



- Likewise, you do not need to know how an object works to send messages to it
- But, you do need to know what messages it understands (i.e., what its capabilities are)
  - class of instance defines the messages that can be sent to it

Views of a Class



- Objects separate *interface*† from *implementation*
  - object is "black box," hiding internal workings and parts
  - interface protects implementation from misuse

†Interface in the generic sense, not the Java syntax sense

Views of a Class

- Interface: *public* view
  - allows instances to cooperate with one another without knowing too many details
  - like a written agreement: consists of a list of capabilities and documentation on how to use them

- Implementation: *private* view
  - properties that help capabilities complete their tasks
  - like the engine of a car

Always declare attributes as private

```
Dog d=new Dog();
d.height=25;
d.height=0;
…
d.bark();
```

Notes About Java Syntax

- Reserved words (keywords)
  - certain words in Java have a particular meaning and cannot be used for any other purpose
  - these are case-sensitive, and consist entirely of lower case letters

```
class    public    new    private    extends
```

- Identifiers
  - names assigned for classes, methods, and variables
  - first character must be a letter or underscore
  - the rest may be any combination of letters, numbers, and underscores — but no spaces

```
Professor          _5thStreet
aPrettyLongName    a_pretty_long_name
```

Making Code More Readable

- Naming conventions
  - conventions are things we suggest to make code more consistent and easier to understand
  - this is purely aesthetic; not enforced by the compiler

- We use capitalization to distinguish an identifier’s purpose
  - class names begin with upper case letters
  - method names begin with lower case
  - instance variables start with an underscore

Coding conventions

	Good Name	Poor Name
class:	`Professor`	`Thing` (no role, purpose)
method:	`teachClass`	`doStuff` (not specific)
instance:	`_csci331Professor`	`p` (too cryptic)

Use names indicative of functionality

A Complete Program

- Here is our first complete Java program, where we are going to use already written classes Ball and BallApp
- We will deconstruct this code

```
public class BallApp extends JFrame
{
        public BallApp(String title)
        {
        }

        public void addBall(Ball ball)
        {
        }

        public static void main(String[] argv)
        {
             BallApp app= new BallApp("Bouncing ball");
        }
}
```

Syntax: Declaring a Class

- *Class declaration* tells Java compiler that we are about to define a new class
  - i.e., we are “declaring” our intent to “define” a class that can be used as a template to instantiate object instances
  - a program must include at least one class definition

```
public class BallApp extends javax.swing.JFrame
```

- Reserved word **`public`** indicates that anyone can create an instance of this class

- Reserved word **`class`** indicates to Java that we are going to define a new class

- **`BallApp`** is the name of the class
  - chosen to indicate that it is an *application* (or program) with a bouncing ball

Syntax: Defining the Class

- *Class definition* following a declaration tells the Java compiler what it means to make an instance of this class and how that instance will respond to messages
  - thus, simply *declaring* a class is not enough
  - we must also *define* what a class does (i.e., how it will fulfill its purpose – its properties and capabilities)

- *Curly braces*, `{ }`, indicate beginning and end of a logical block of code or "code body:" in this case, a class definition
  - represent difference between declaration and definition
  - code written between curly braces is associated with class declared immediately before them

```
public class BallApp extends javax.swing.JFrame
{
}
```

  - this is an example of an empty code block. While this "nothing" or "null" code compiles because it is legal, i.e., *syntactically correct*, it does not do anything useful

- Java programs are composed of any number of class definitions
  - in this respect, Java code is like a dictionary: "declaration" of concept, followed by its definition
  - no code can appear outside of a class definition

Constructors

- Constructor: a special method that is called whenever a class is instantiated (created)
  - another object sends a message that calls a constructor
  - constructor is the first message an object receives and cannot be called subsequently on the instance
  - *establishes initial state of properties* for the object instance

- If you do not define any constructors for a class, Java writes one for you
  - called *default* constructor
  - default constructor will initialize each instance variable to its default value
  - If you write at least one non-default constructor, you cannot use the default constructor anymore

```
public class BallApp extends JFrame
{
        public BallApp(String title)
        {
        }
}
```

Object Instantiation

```
public class BallApp extends JFrame
{
      public BallApp(String title)
      {
      }

      public static void main(String [] args)
      {
         BallApp app=new BallApp (“Bouncing ball”);
      }
}
```

Main is the entry point of your program

The instance of BallApp is created

Calling object commands

```
public class BallApp extends JFrame
{
      public BallApp(String title)
      {
      }
      public void addBall(Ball ball)
      {
      }
      public static void main(String [] args)
      {
          BallApp app=new BallApp (“Bouncing ball”);
          Ball b=new Ball();
          app.addBall(b);
      }
}
```

We create an instance of a ball and add it to the BallApp, which is a JFrame – a window for holding other components

Note, that we were able to add the ball because BallApp has a corresponding method – understands our command

Making ball bounce

```
public class BallApp extends JFrame
{
        public BallApp(String title)
        {
        }
        public void addBall(Ball ball)
        {
        }
        public static void main(String [] args)
        {
            BallApp app=new BallApp ("Bouncing ball");
            Ball b=new Ball();
            app.addBall(b);
            b.startBouncing();
        }
}
```

How Java program is executed

- User starts program by typing `java BallApp` in a terminal
- JavaVM calls the method `main(…)`
- `BallApp` is instantiated by the `new` command
  - i.e., constructor `BallApp(String)` is called

- The body of the `BallApp()` constructor draws an empty window on the screen

- We create a new instance of a `Ball.` We call next `BallApp`’s method `addBall`. `BallApp` (and anything constructed in it) runs until program terminates.
- `Ball` knows how to bounce and doesn’t stop on its own, so it will keep going until `BallApp` is destroyed

Final notes about the Ball

- The interface for bouncing ball is extremely simple — instantiate it and watch it go
  - pro: it is easy (it does all the work!)
  - con: impossible to change anything about the ball
  - class interfaces can be this simple or much, much more complex
- We will spend much of our time exploring appropriate class interfaces to make collections of objects work together to do something useful

Client and Server programming

- Client programming – composing a useful program from existing components – objects
- Server programming – creating new useful components. The principle here: each component knows how to do something well, not too much.
- In the `BallApp` we were in a role of client programmers

Summary

- We use objects to abstract parts of reality
- We distinguish object types. The type of an object – or a class – defines properties (fields) and methods (what messages does the object understand)
- In java, we can create an instance of an object only on the heap
- We make an object do things by calling a method that is defined in an object class

Class is not an object