IAT 265

Objects

Outline

- User defined methods
- Class & Object
  - Why object?
  - Object components
  - Ladybug class
  - Primitive types and Object References
- Method signature & overloading
- Object-oriented Programming
  - Encapsulation: information hiding

User defined methods

- With user defined methods:
  - Make your code reusable (by calling a method more than once)
  - your code becomes easier to write, understand, and debug

Example: Ladybug

- Rather than mess up draw() method too much, define a method:

```
void drawBug(){
  pushMatrix();
  translate(bugX, bugY);
   ...
}
```

- Then call it within draw():

```
void draw() {
  ...
  drawBug();
}
```

Class & Object

Why objects?

- We live in a world full of objects
  - Images, cars, remote controls, televisions, employees, students, ladybugs, fishes, ...
- The older languages are procedural
- OOP languages have the added capability to encapsulate objects' properties and functions into one container – ***class***
  - Instances of a class are called ***objects***

Object Oriented vs. Procedural Languages

Procedural (e.g. C)

- We create some data representing a fish
- We write a *procedure* that can accept the data and draw the fish

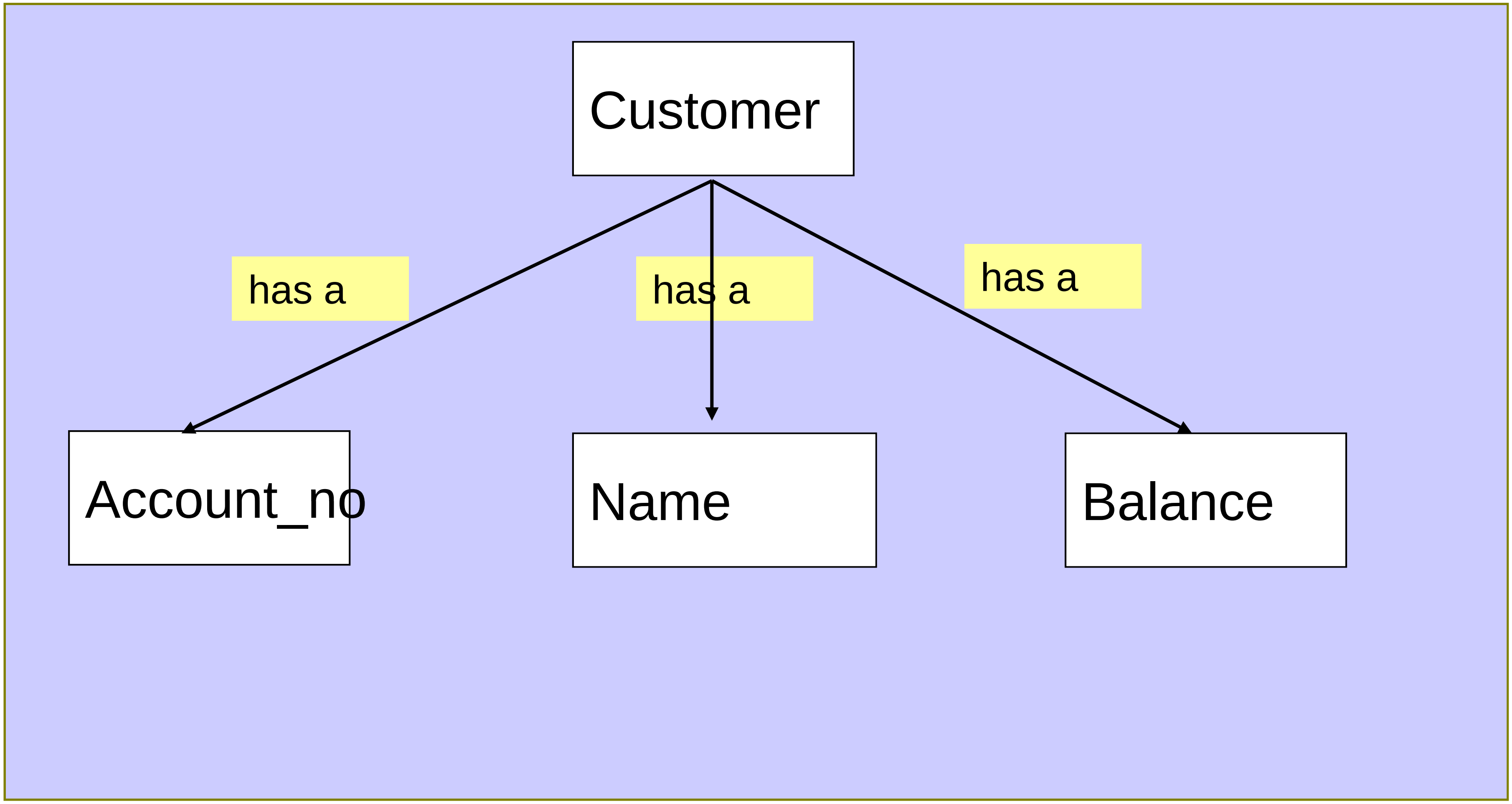
Object Oriented (e.g. Java)

- We create a *class* that contains fish data AND a procedure to draw it
- The data and the procedure (ability to draw) are all in ONE "container"

Object-Oriented Programming

- So what?
- Think about this:
  - When you go to bank reception or school registration, why the attendant can find you with a couple of key stroke?

A Customer object Encapsulates all its properties into one



What an object can offer?

- About Who you are:
  - Relevant properties/states (e.g. Fish: sizes, location, alive ...)
- About What you can do:
  - Behaviors of an object (e.g. Fish: move, collide, dodge, ...)

Parts of a class

- As blueprint for objects, a class consists of:
  - fields (member variables): hold objects' properties/states
  - methods (functions): hold objects' behaviors

Classes vs Objects

- A Class is a blueprint for fish
- An Object is a fish
- Many fishes, one blueprint

Parts of a class in detail

- Classes define *fields*, *constructors* and *methods*

- Fields are the variables that will appear inside every instance of the class
  - Each instance has its own values

- Constructors are special methods that define how to build instances (generally, how to set the initial values of fields)
  - Special: a) share the same name as the class; b) no return type

- Methods are how you *do things* to instances

Defining the Ladybug class

```
class Ladybug
{
   // fields
   int bugX;
   int bugY;
   int bugW;
   int bugH;
   int changeX;

   // constructor
   Ladybug(int x, int y,  int w,
        int h, int chgX) {
    bugX= x;
    bugY=y;
    bugW=w;
    bugH=h;
    changeX = chgX;
  }
```

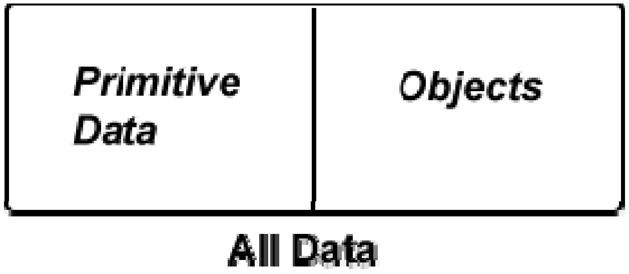
```
// methods
void drawBug() {
 //make the bug rotate
 if(changeX<0) {
   rotateY(PI);
 }
 //change moving direction
 if((bugX+bugW+9) > (gardenX+gardenW)
     ||(bugX-bugW-9) < gardenX) {
   changeX = changeX * -1;
 }

 //Move the bug at speed changeX
 bugX = bugX+changeX;

 //Draw bug body
 …
 //draw the four dots
 …
 //draw the head as an arc
 …
 //draw its body line and antenna }}
```

Using the class to create instances

- Classes define a *type*
- You can now *declare* variables of this type and *initialize* them using the constructor
- Like arrays, the keyword *new* is used to tell Java to create a new object

```
Ladybug b1, b2 ;
void setup() {
   …
   b1 = new Ladybug(gardenX+50, 200, 34, 30, 1);
   b2 = new Ladybug(gardenX+gardenW-50, 200, 34, 30,
   -1);
}
void draw() {
   …
  b1.drawBug();
  b2.drawBug();
}
```

Classification of Data Types

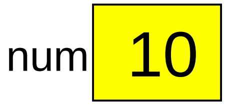


- Primitive Data Types: - primitive data
  - integer (byte, short, int, and long)
  - float (float and double)
  - char (E.g. a, b, c, A, B, C, &, `*`, etc)
  - boolean (`true` or `false`)

- Reference Data Types - objects
  - Class, String and Array

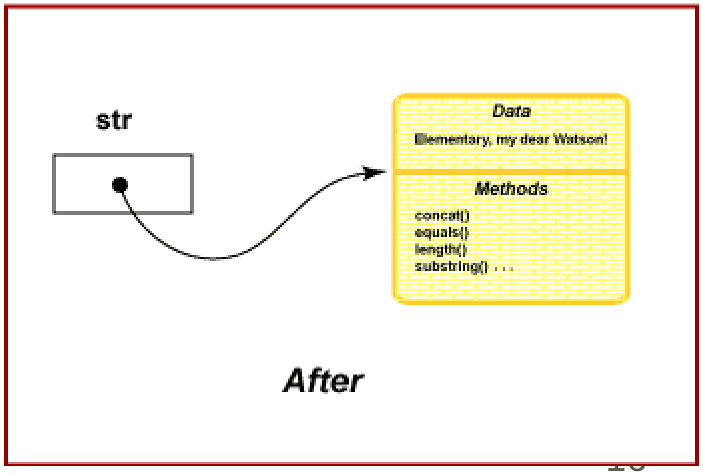
Difference between variables of Primitive and Reference types

- A primitive type variable is an identifier for a value
  - E.g. int num = 10;

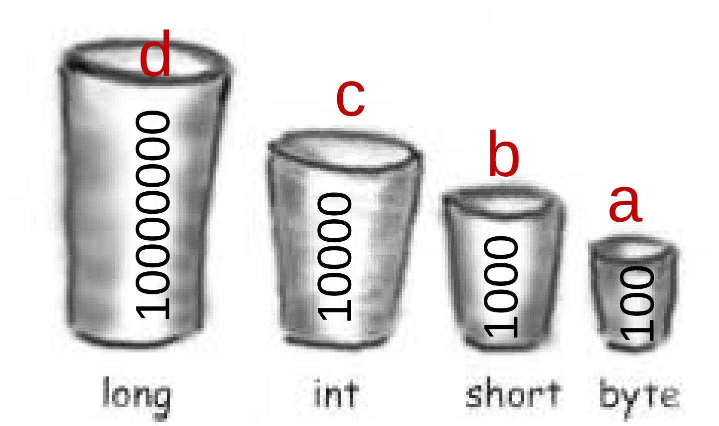


- A reference type variable is a reference to an object's memory location (its address rather than a value):
  - E.g.

    String str = new String("Elementary, my dear Watson!");



Another metaphor: Primitive types

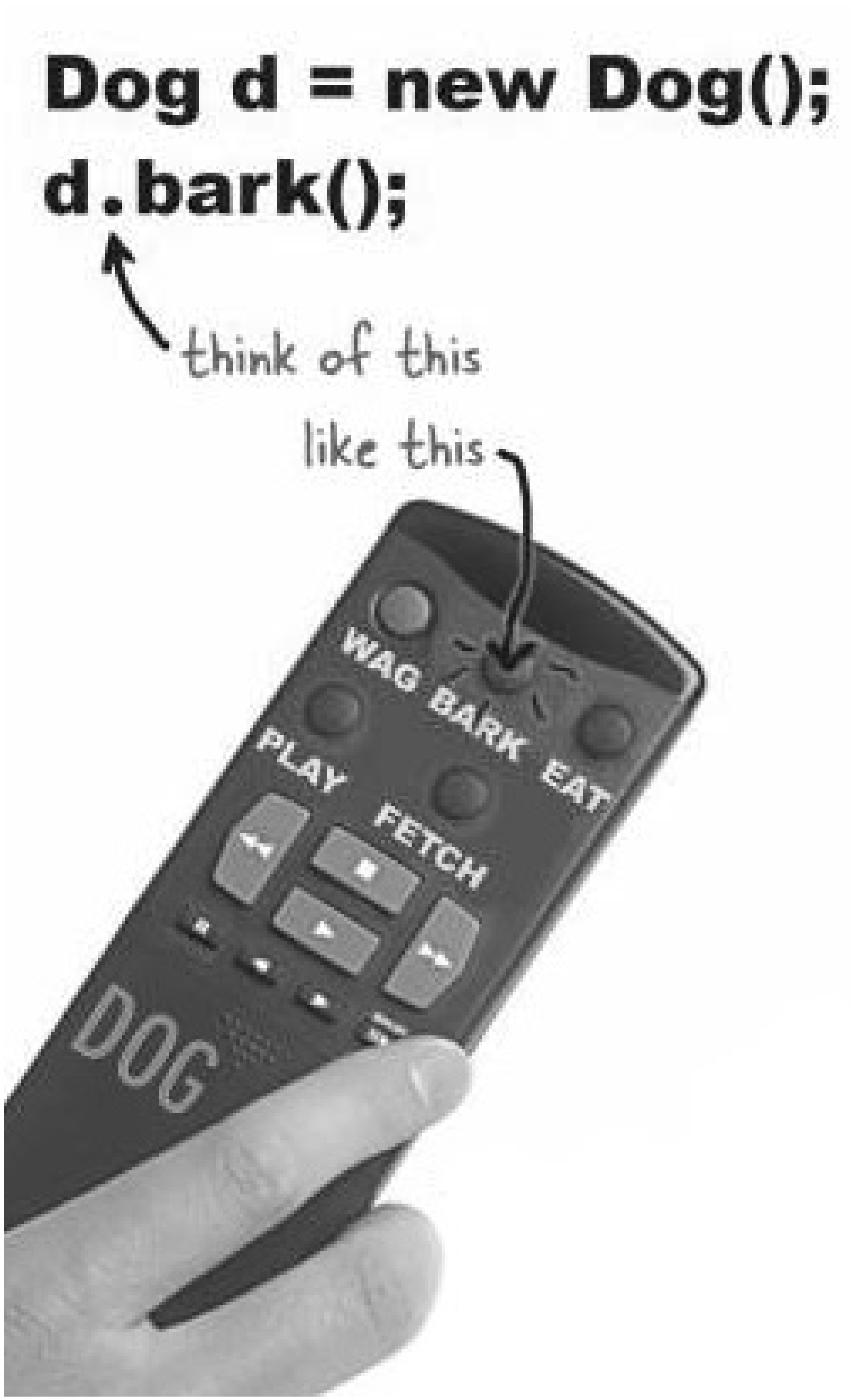


- A Primitive type variable is a bucket that holds values

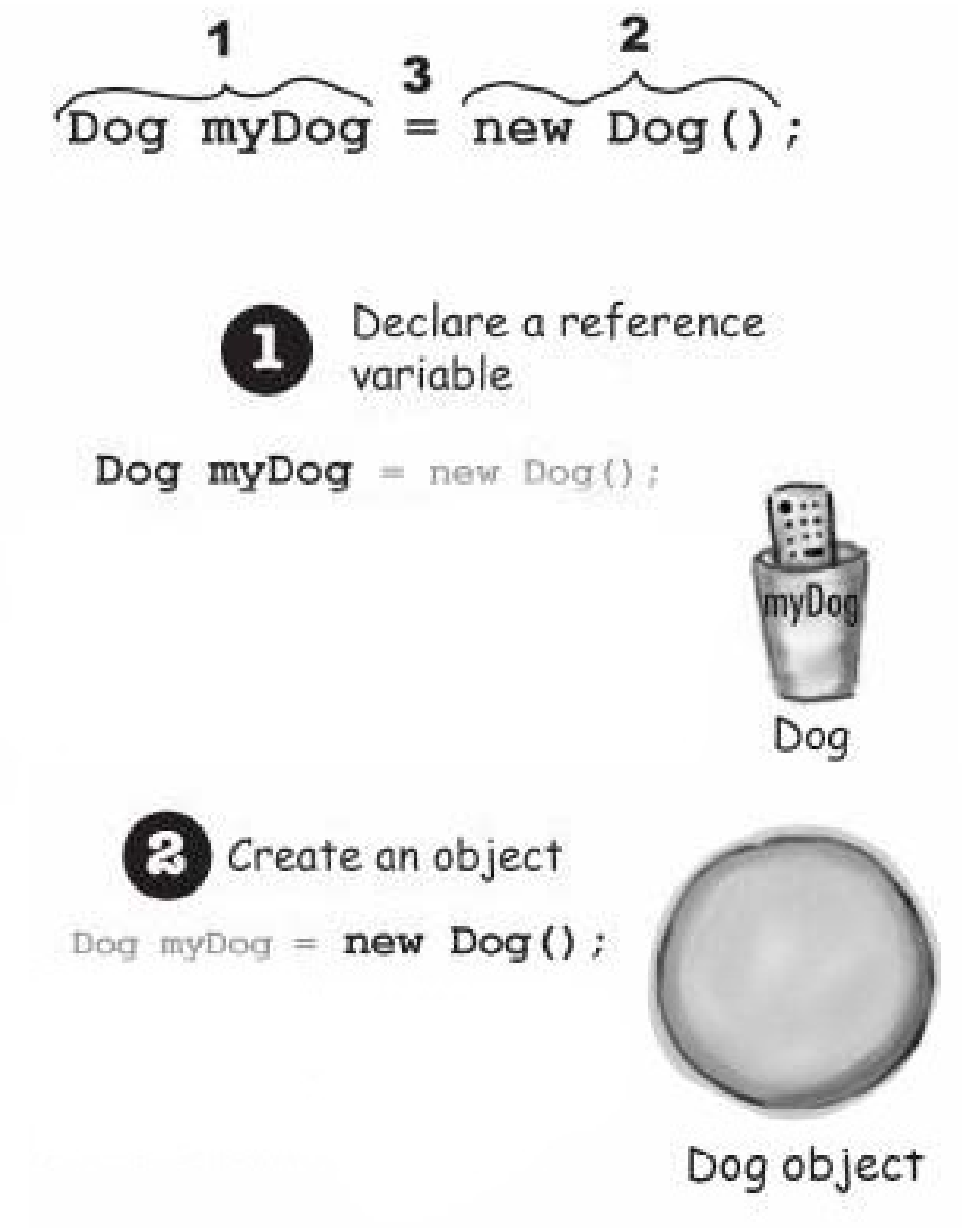
byte:	8bits	e.g. byte a = 100;
short:	16bits	e.g. short b = 1000;
int:	32bits	e.g. int c = 10000;
long:	64bits	e.g. long d = 10000000;

Reference

- Like a remote control
- a reference is a primitive thing that points at objects
- the assignment operator causes the reference to point at a new instance of the class



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



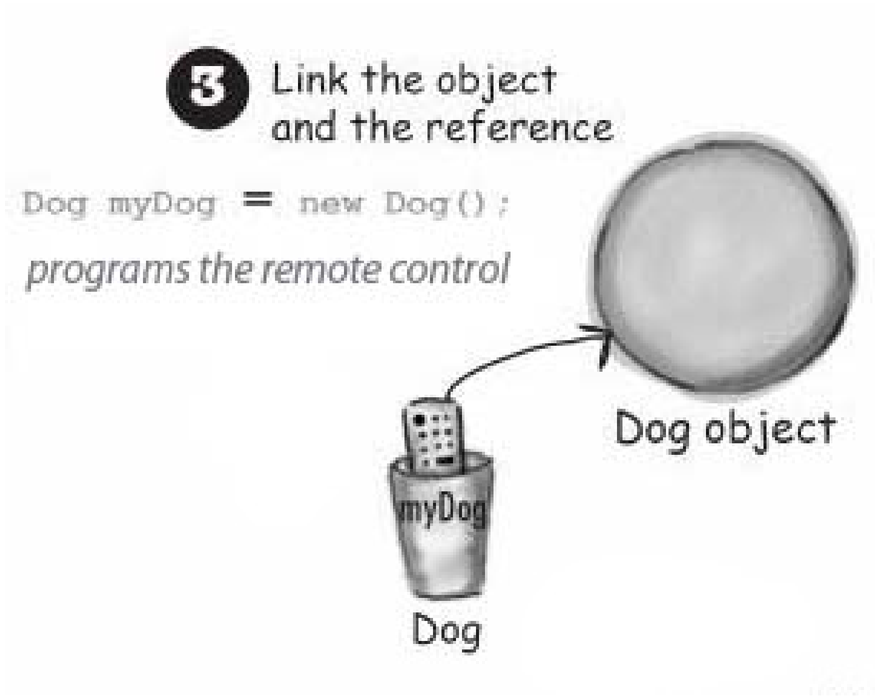
```
$\overbrace{\text{Dog myDog}}^{1}$ $\overset{3}{=}$ $\overbrace{\text{new Dog();}}^{2}$
```

1 Declare a reference variable

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

2 Create an object

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



3 Link the object and the reference

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

programs the remote control

Add a method for collision detection between bugs

```
//Method to detect collision between the current bug and another bug
 boolean detectCollision(Ladybug otherBug) {
   if ( abs(bugX-otherBug.bugX)<(bugW/2+otherBug.bugW/2) &&
          abs(bugY-otherBug.bugY)< (bugH/2+otherBug.bugH/2) ) {
     return true;
   }
   return false;
 }
```

Method Signature

- *Signature* is a term that means
  - The full specification of the method name
- signature = return type + method name+ (parameters if any)
- Signature is important to programmers as you can learn from it how to call a method correctly:
  - What and how many arguments it demands
  - What type of value it returns to you
- Signature is also important to the system:
  - System differentiates methods based on signatures rather than names

Call detectCollision() method based on its signature

```
void draw(){
    ...
    //Signature: boolean detectCollision(Ladybug otherBug)
  boolean colliding = b1.detectCollision(b2);
    if(colliding ) {
      b1.changeX *= -1;
      b2.changeX *= -1;
     }
   }
}
```

Same name, different signature – method overloading

- You may have more than one method with the **same name**
  - **No** more than one method with the **same signature** though!!
- Overloading - build variants of the same method name with different parameters:

```
Ladybug() {
     bugX = random(gardenW);
     bugY = random(gardenH);
     …
}
// another constructor!!
Ladybug(int x, int y,  int w, int h, int chgX) {
     bugX= x;
     bugY=y;
     …
}
```

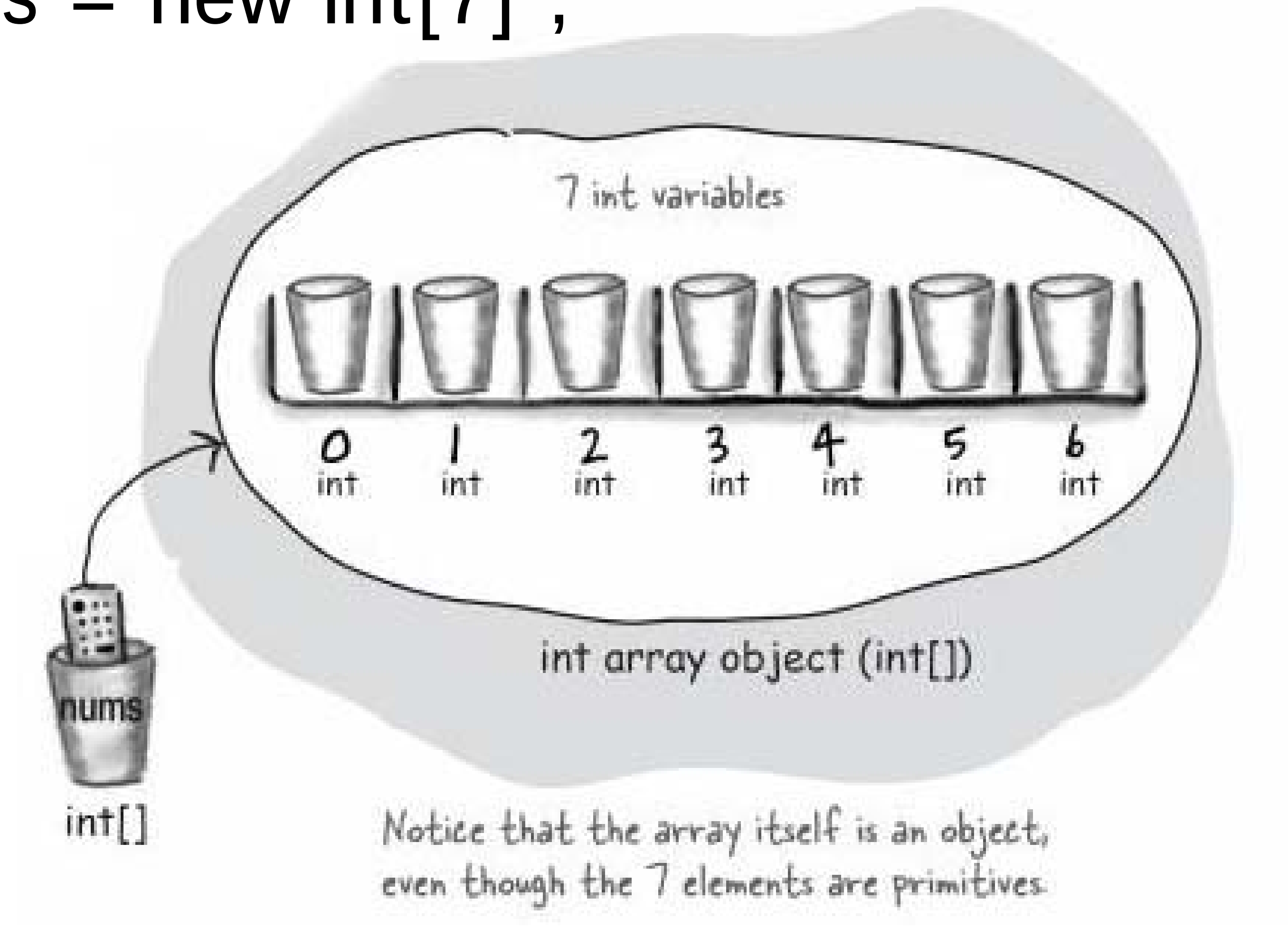
Method overloading

- Another example, with print() method:

```
int i = 1  ;
float f = 3.14 ;
String s = "Hello";
void print(int i) → print( i );
void print(float f) → print( f );
void print(String s) → print(s);
```

Arrays

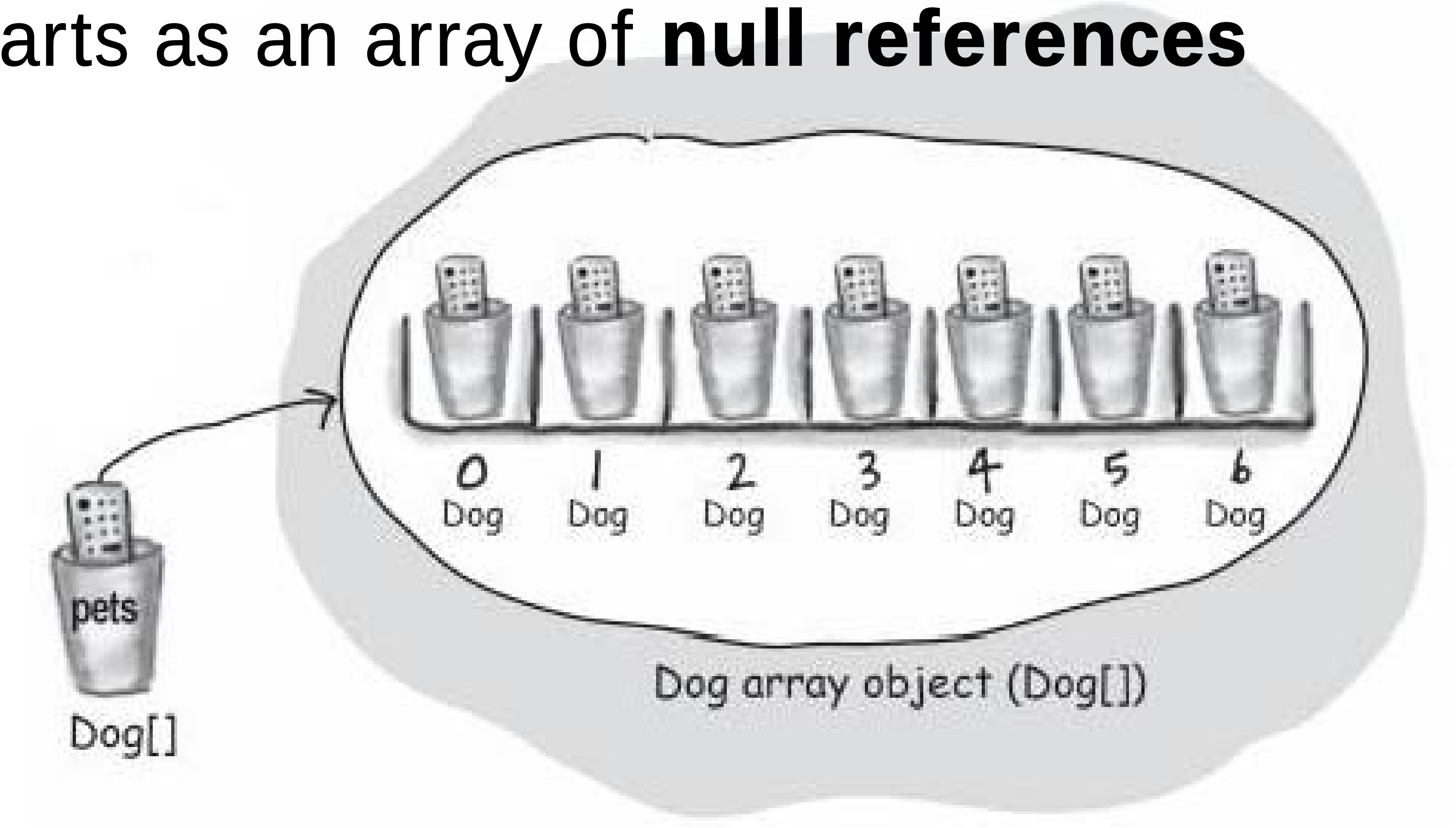
- int[] nums = new int[7] ;



Notice that the array itself is an object, even though the 7 elements are primitives.

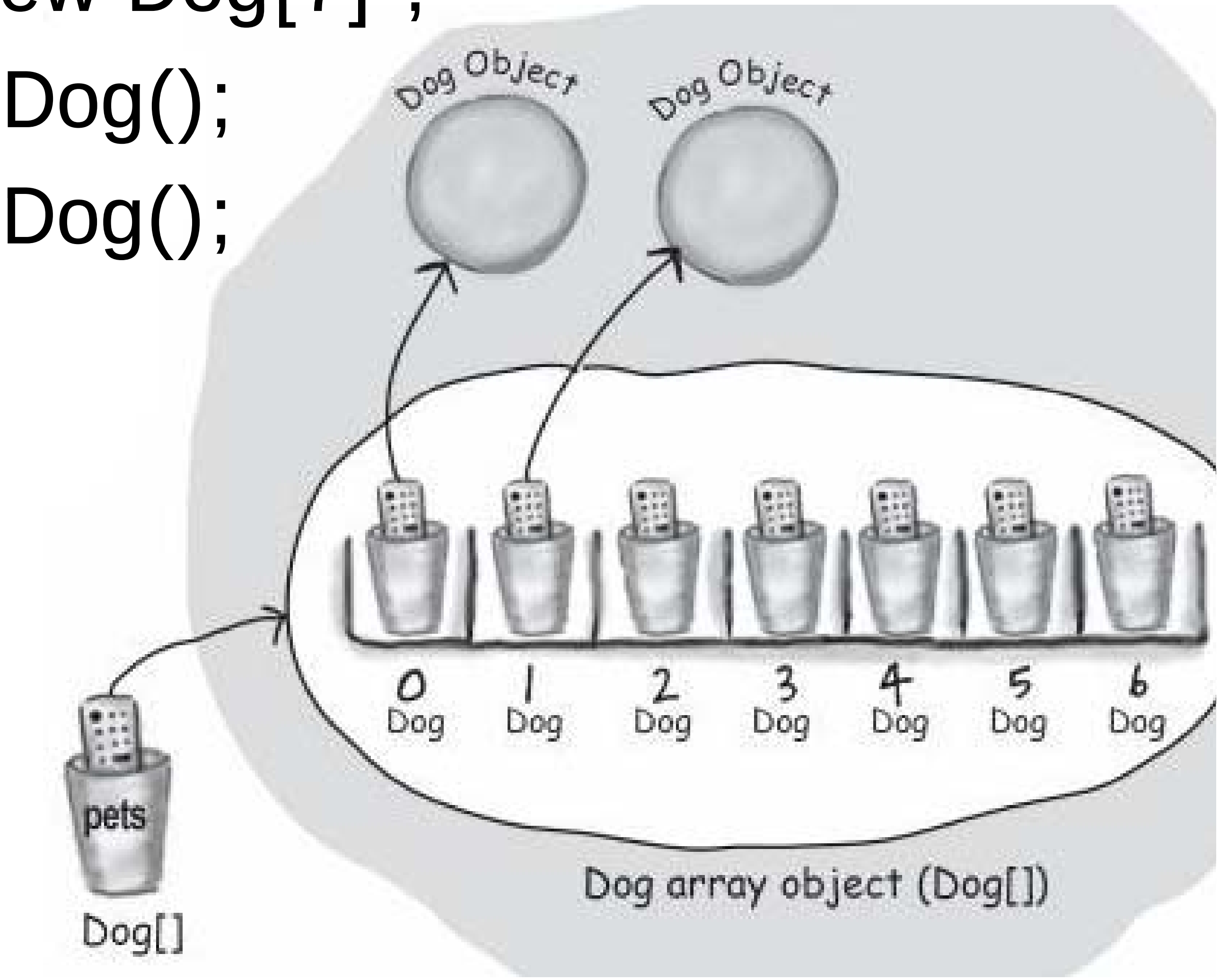
Array of objects

- Dog[] pets = new Dog[7];
- It starts as an array of **null references**



Array of objects

```
Dog[] pets = new Dog[7] ;
pets[0] = new Dog();
pets[1] = new Dog();
```



June 13, 2011

Example: Ladybug array

```
//create the Ladybug array
Ladybug[] bugs = new Ladybug[count];

//in setup(), fill the bugs array with Ladybug objects
...
for(int i=0; i<count; i++) {
    bugs[i] = new Ladybug (random(gardenW),
          random(gardenH),  random(-1,1), random(-1,1),
          random(12,36));
  }
//in draw(), with loops
```

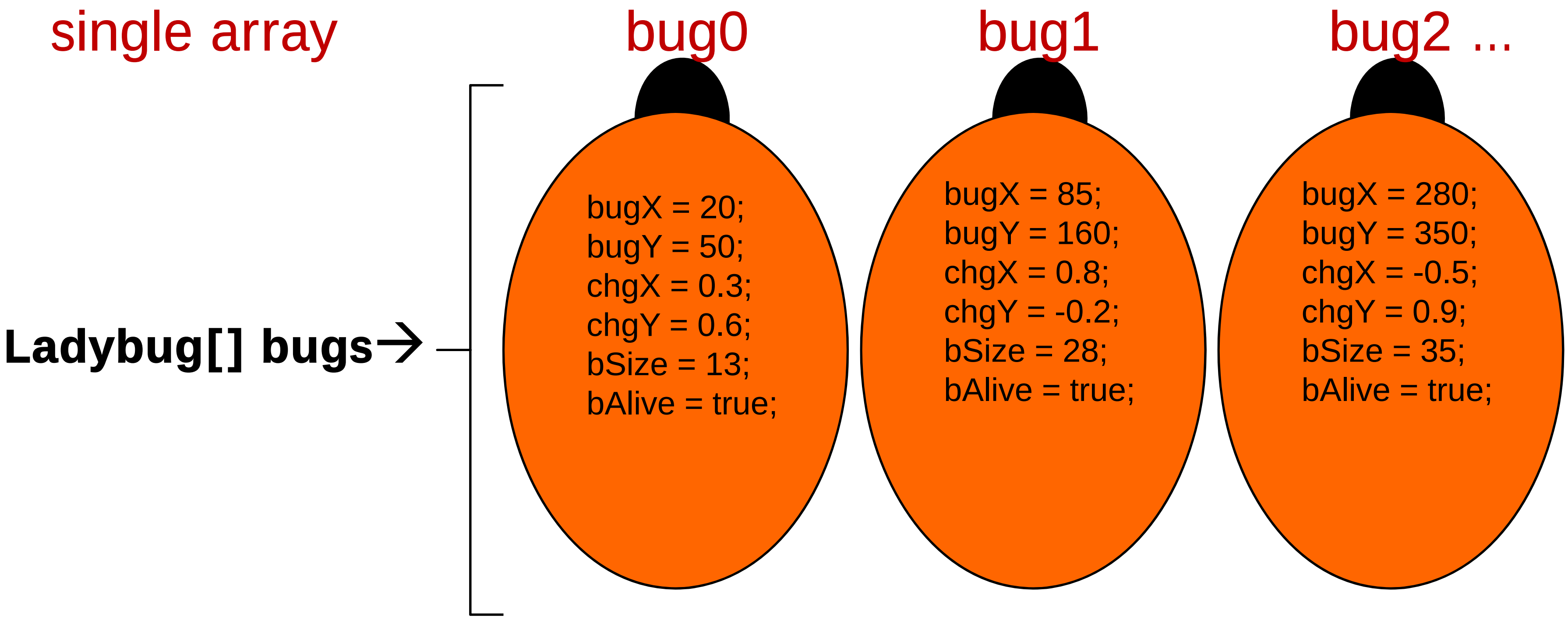
The Advantage of using Objects

- Before using objects

6 arrays			bug0	bug1	bug2 ...
float[] bugX	→	{	20,	85,	280, ...}
float[] bugY	→	{	50,	160,	350, ...}
float[] changeX	→	{	0.3,	0.8,	-0.5, ...}
float[] changeY	→	{	0.6,	-0.2,	0.9, ...}
float[] bSize	→	{	13,	28,	35, ...}
boolean[] bugAlive	→	{	true,	true,	true, ...}

The Advantage of using Objects

- After using objects



single array

Ladybug[] bugs→

bug0

```
bugX = 20;
bugY = 50;
chgX = 0.3;
chgY = 0.6;
bSize = 13;
bAlive = true;
```

bug1

```
bugX = 85;
bugY = 160;
chgX = 0.8;
chgY = -0.2;
bSize = 28;
bAlive = true;
```

bug2 ...

```
bugX = 280;
bugY = 350;
chgX = -0.5;
chgY = 0.9;
bSize = 35;
bAlive = true;
```

Example: Ladybug array

```
//in draw(), within loops, beyond other things:
   …
   //Here we will check to see if our bug "i" hits the wall by calling
   //method with signature void detectBound()
   bugs[i].detectBound();
   …
   //detect collision among bugs by calling method with signature
   // boolean detectCollision(Ladybug otherBug)
   if(bugs[i].alive && bugs[k].alive && i != k) {
      if(bugs[i].detectCollision(bugs[k])) {
         …
      }
    }
   …
   bugs[i].drawBug();
```

Three Principles of OOP

- **Encapsulation**
- Inheritance
- Polymorphism

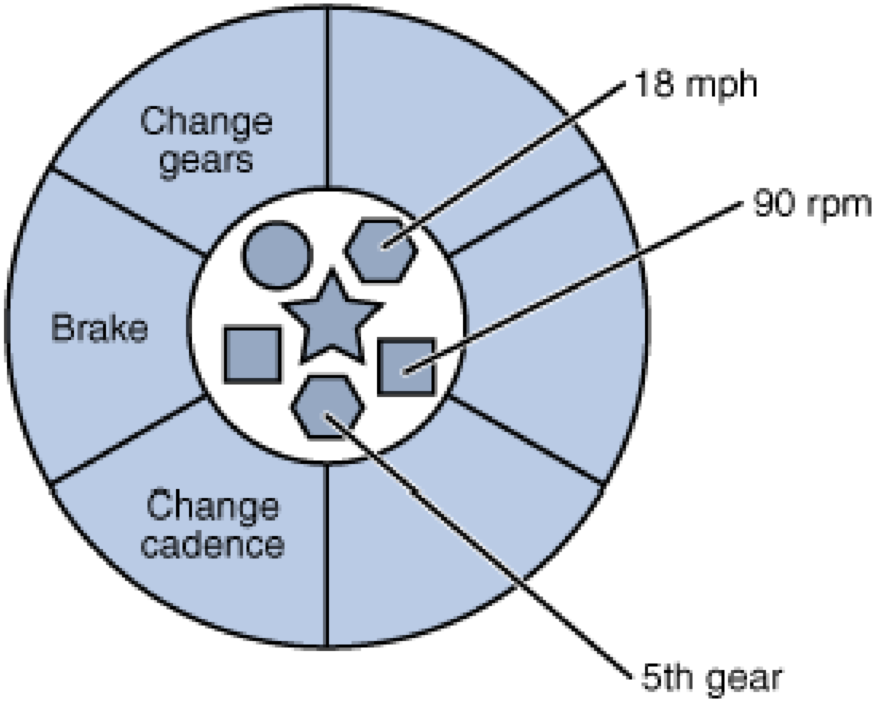
Data Encapsulation

- Hiding internal states with
  - private fields

- Performing all interaction through an object's
  - public methods

Encapsulation for Bicycle object

- State
  - int gear ;
  - float speed ;
  - float cadence ;
- Behavior
  - changeGears(int g);
  - brake(float level);
  - changeCadence(float c);
  - int getGear();
  - float getSpeed();   ...



Encapsulation for Bicycle object

- An object’s private fields can’t be accessed by any objects/methods external to it

```
class Bicycle
{
    private int cadence = 0;
    private int speed = 0;
    private int gear = 1;
    //Constructor
    Bicycle () { }
} //end of Bicycle

//Tried to access private from an external method
void someMethodOutsideBicycle () {
   Bicycle bike = new Bicycle ();
   bike.gear = 5;
   print(bike.gear);
}
```

Illegal !!
You can’t do these in Java

Walk around via Setter and Getter methods

- What can you do with private data?
  - to set it: setVarName(varType newValue)
  - to get it: varType getVarName()

Example of Setter & Getter

```
class Bicycle
{
      …
       private int gear = 1;
       …
      void setGear( int g) { gear = g; }
      int getGear () { return gear; }
}
/ /Tried to access private from an external method
void someMethodOutsideBicycle () {
    Bicycle bike = new Bicycle ();
    bike. setGear(5);
    print(bike.getGear());
}
```

Why Hide information?

- Controls access
  - By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world
- Ensures correctness. For instance:

```
void setGear( int g) {
   if(g > 10) {
     print (“Wrong data”);
   } else {
     gear = g;
   }
}
```

Principle: Define the Interface

- Define the interface:
  - public methods with defined operations
- The interface is the thing that other people use
- In heritance, if you have the same interface in both parent class and child class
  - **You can plug in a better implementation in the child's version!**

Summary of principles

- User defined methods
- Class & Object
  - Why object?
  - Object components
  - Ladybug class
  - Primitive types and Object References
- Method signature & overloading
- Object-oriented Programming
  - Encapsulation: information hiding