

# Outline

- Lecture 1 Review
- More types
- Methods
- Conditionals

# Types

Kinds of values that can be stored and manipulated.

**boolean:** Truth value (**true** or **false**).

**int:** Integer (0, 1, -47).

**double:** Real number (3.14, 1.0, -2.1).

**String:** Text (“hello”, “example”).

# Variables

Named location that stores a value

**Example:**

```
String a = "a";
```

```
String b = "letter b";
```

```
a = "letter a";
```

```
String c = a + " and " + b;
```

# Operators

Symbols that perform simple computations

Assignment: =

Addition: +

Subtraction: -

Multiplication: \*

Division: /

```
class GravityCalculator {  
    public static void main(String[] args) {  
        double gravity = -9.81;  
        double initialVelocity = 0.0;  
        double fallingTime = 10.0;  
        double initialPosition = 0.0;  
        double finalPosition = .5 * gravity * fallingTime *  
                                fallingTime;  
        finalPosition = finalPosition +  
                                initialVelocity * fallingTime;  
        finalPosition = finalPosition + initialPosition;  
        System.out.println("An object's position after " +  
                            fallingTime + " seconds is " +  
                            finalPosition + " m.");  
    }  
}
```

**finalPosition = finalPosition +  
initialVelocity \* fallingTime;  
finalPosition = finalPosition + initialPosition;**

**OR**

**finalPosition += initialVelocity \* fallingTime;  
finalPosition += initialPosition;**

# Questions from last lecture?

# Outline

- Lecture 1 Review
- **More types**
- Methods
- Conditionals



# Division

Division (“/”) operates differently on integers and on doubles!

Example:

```
double a = 5.0/2.0; // a = 2.5
```

```
int b = 4/2; // b = 2
```

```
int c = 5/2; // c = 2
```

```
double d = 5/2; // d = 2.0
```

# Order of Operations

Precedence like math, left to right

Right hand side of = evaluated first

Parenthesis increase precedence

```
double x = 3 / 2 + 1; // x = 2.0
```

```
double y = 3 / (2 + 1); // y = 1.0
```

# Mismatched Types

Java verifies that types always match:

```
String five = 5; // ERROR!
```

test.java.2: incompatible types

found: int

required: java.lang.String

```
String five = 5;
```

# Conversion by casting

```
int a = 2;      // a = 2
```

```
double a = 2;   // a = 2.0 (Implicit)
```

```
int a = 18.7;   // ERROR
```

```
int a = (int)18.7; // a = 18
```

```
double a = 2/3; // a = 0.0
```

```
double a = (double)2/3; // a = 0.6666...
```

# Outline

- Lecture 1 Review
- More types
- **Methods**
- Conditionals

# Methods

```
public static void main(String[] arguments)
```

```
{
```

```
    System.out.println("hi");
```

```
}
```

# Adding Methods

```
public static void NAME() {  
    STATEMENTS  
}
```

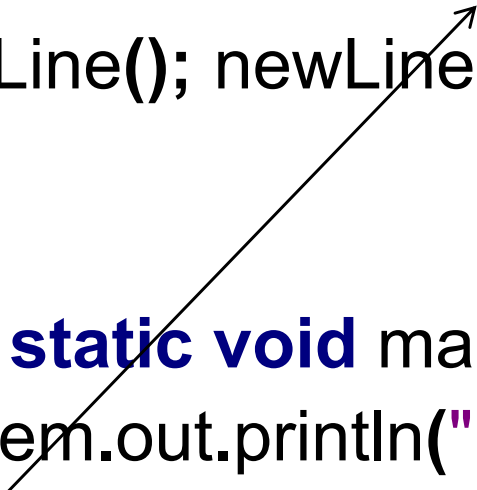
To call a method:

```
NAME ( ) ;
```

```
class NewLine {  
    public static void newLine() {  
        System.out.println("");  
    }  
  
    public static void threeLines() {  
        newLine(); newLine(); newLine();  
    }  
  
    public static void main(String[] arguments) {  
        System.out.println("Line 1"); ←  
        threeLines();  
        System.out.println("Line 2");  
    }  
}
```



```
class NewLine {  
    public static void newLine() {  
        System.out.println("");  
    }  
  
    public static void threeLines() {  
        newLine(); newLine(); newLine();  
    }  
  
    public static void main(String[] arguments) {  
        System.out.println("Line 1");  
        threeLines();  
        System.out.println("Line 2");  
    }  
}
```



```
class NewLine {  
    public static void newLine() {  
        System.out.println("");  
    }  
  
    public static void threeLines() {  
        newLine(); newLine(); newLine();  
    }  
  
    public static void main(String[] arguments) {  
        System.out.println("Line 1");  
        threeLines();  
        System.out.println("Line 2");  
    }  
}
```

The diagram illustrates the flow of method calls in the provided Java code. Three arrows originate from the `newLine()` calls within the `threeLines()` method and point to the `newLine()` method definition. A fourth arrow originates from the `threeLines()` method call within the `main()` method and points to the `threeLines()` method definition.

# Parameters

```
public static void NAME(TYPE NAME) {  
    STATEMENTS  
}
```

To call:

```
NAME ( EXPRESSION ) ;
```

```
class Square {  
    public static void printSquare(int x) {  
        System.out.println(x*x);  
    }  
  
    public static void main(String[] arguments) {  
        int value = 2;  
        printSquare(value);  
        printSquare(3);  
        printSquare(value*2);  
    }  
}
```

```
class Square2 {  
    public static void printSquare(int x) {  
        System.out.println(x*x);  
    }  
  
    public static void main(String[] arguments) {  
        printSquare("hello");  
        printSquare(5.5);  
    }  
}
```

What's wrong here?

```
class Square3 {  
    public static void printSquare(double x) {  
        System.out.println(x*x);  
    }  
  
    public static void main(String[] arguments) {  
        printSquare(5);  
    }  
}
```

What's wrong here?

# Multiple Parameters

```
[...] NAME(TYPE NAME, TYPE NAME) {  
    STATEMENTS  
}
```

To call:

```
NAME (arg1, arg2) ;
```

```
class Multiply {  
    public static void times (double a, double b) {  
        System.out.println(a * b);  
    }  
  
    public static void main(String[] arguments) {  
        times (2, 2);  
        times (3, 4);  
    }  
}
```



# Return Values

```
public static TYPE NAME() {  
    STATEMENTS  
    return EXPRESSION;  
}
```

`void` means “no type”

```
class Square3 {  
    public static void printSquare(double x) {  
        System.out.println(x*x);  
    }  
  
    public static void main(String[] arguments) {  
        printSquare(5);  
    }  
}
```

```
class Square4 {  
    public static double square(double x) {  
        return x*x;  
    }  
  
    public static void main(String[] arguments) {  
        System.out.println(square(5));  
        System.out.println(square(2));  
    }  
}
```

# Variable Scope

Variables live in the block ({} ) where they are defined (**scope**)

Method parameters are like defining a new variable in the method

```
class SquareChange {  
    public static void printSquare(int x) {  
        System.out.println("printSquare x = " + x);  
        x = x * x;  
        System.out.println("printSquare x = " + x);  
    }  
  
    public static void main(String[] arguments) {  
        int x = 5;  
        System.out.println("main x = " + x);  
        printSquare(x);  
        System.out.println("main x = " + x);  
    }  
}
```

```
class Scope {  
    public static void main(String[] arguments) {  
        int x = 5;  
        if (x == 5) {  
            int x = 6;  
            int y = 72;  
            System.out.println("x = " + x + " y = " + y);  
        }  
        System.out.println("x = " + x + " y = " + y);  
    }  
}
```

# Methods: Building Blocks

- Big programs are built out of small methods
- Methods can be individually developed, tested and reused
- User of method does not need to know how it works
- In Computer Science, this is called “*abstraction*”

# Mathematical Functions

`Math.sin(x)`

`Math.cos(Math.PI / 2)`

`Math.pow(2, 3)`

`Math.log(Math.log(x + y))`



# Outline

- Lecture 1 Review
- More types
- Methods
- **Conditionals**

if statement

```
if (CONDITION) {  
    STATEMENTS  
}
```

```
public static void test(int x) {  
    if (x > 5) {  
        System.out.println(x + " is > 5");  
    }  
}
```

```
public static void main(String[] arguments) {  
    test(6);  
    test(5);  
    test(4);  
}
```

# Comparison operators

$x > y$ :  $x$  is greater than  $y$

$x < y$ :  $x$  is less than  $y$

$x \geq y$ :  $x$  is greater than or equal to  $x$

$x \leq y$ :  $x$  is less than or equal to  $y$

$x == y$ :  $x$  equals  $y$

( equality:  $==$ , assignment:  $=$  )

# Boolean operators

&&: logical AND

||: logical OR

if (x > 6) { if (x < 9) { ... } }	→	if ( x > 6 && x < 9) { ... }
---	---	------------------------------------

else

```
if (CONDITION) {  
    STATEMENTS  
} else {  
    STATEMENTS  
}
```

```
public static void test(int x) {  
    if (x > 5) {  
        System.out.println(x + " is > 5");  
    } else {  
        System.out.println(x + " is not > 5");  
    }  
}
```

```
public static void main(String[] arguments) {  
    test(6);  
    test(5);  
    test(4);  
}
```

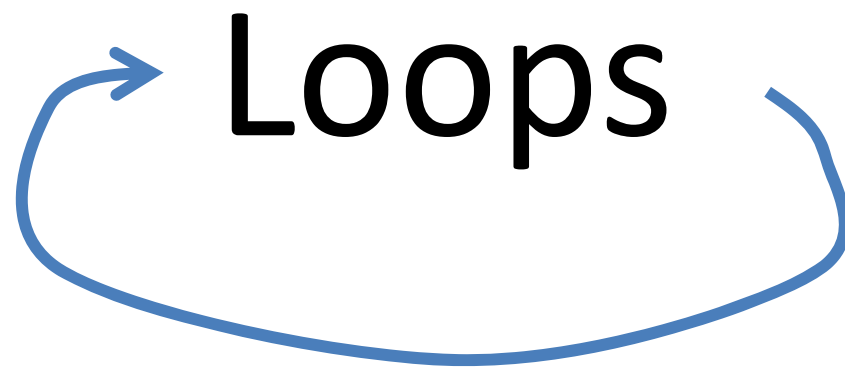
else if

```
if (CONDITION) {  
    STATEMENTS  
} else if (CONDITION) {  
    STATEMENTS  
} else if (CONDITION) {  
    STATEMENTS  
} else {  
    STATEMENTS  
}
```



```
public static void test(int x) {  
    if (x > 5) {  
        System.out.println(x + " is > 5");  
    } else if (x == 5) {  
        System.out.println(x + " equals 5");  
    } else {  
        System.out.println(x + " is < 5");  
    }  
}
```

```
public static void main(String[] arguments) {  
    test(6);  
    test(5);  
    test(4);  
}
```



# Loops

```
static void main (String[] arguments) {  
    System.out.println("Rule #1");  
    System.out.println("Rule #2");  
    System.out.println("Rule #3");  
}
```

What if you want to do it for 200 Rules?

# Loops

Loop operators allow to loop through a block of code.

There are several loop operators in Java.

# The *while* operator

```
while (condition) {  
    statements  
}
```

# The *while* operator

```
int i = 0;
while (i < 3) {
    System.out.println("Rule #" + i);
    i = i+1;
}
```

Count carefully

Make sure that your loop has a chance to finish.

# The *for* operator

```
for (initialization; condition; update) {  
    statements  
}
```

# The *for* operator

```
for (int i = 0; i < 3; i=i+1) {  
    System.out.println("Rule #" + i);  
}
```

Note: `i = i+1` may be replaced by `i++`



# Branching Statements

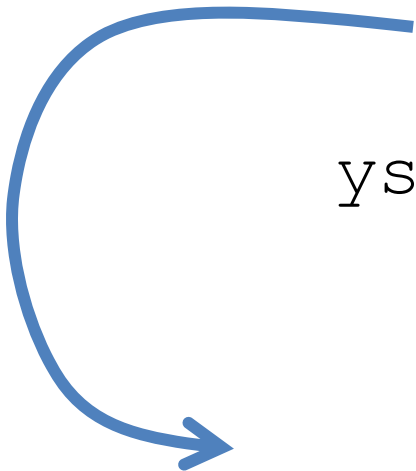
*break* terminates a *for* or *while* loop

```
for (int i=0; i<100; i++) {
```

```
    if (i == 50)
```

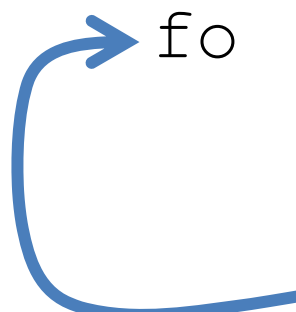
```
        break;
```

```
    system.out.println("Rule #" + i);
```



# Branching Statements

*continue* skips the current iteration of a loop and proceeds directly to the next iteration



```
for (int i=0; i<100; i++) {  
    if (i == 50)  
        continue;  
    System.out.println("Rule #" + i);  
}
```

# Embedded loops

```
for (int i = 0; i < 3; i++) {  
    for (int j = 2; j < 4; j++) {  
        System.out.println (i + " " + j);  
    }  
}
```

Scope of the variable defined in the initialization:  
respective *for* block

# Arrays

# Arrays

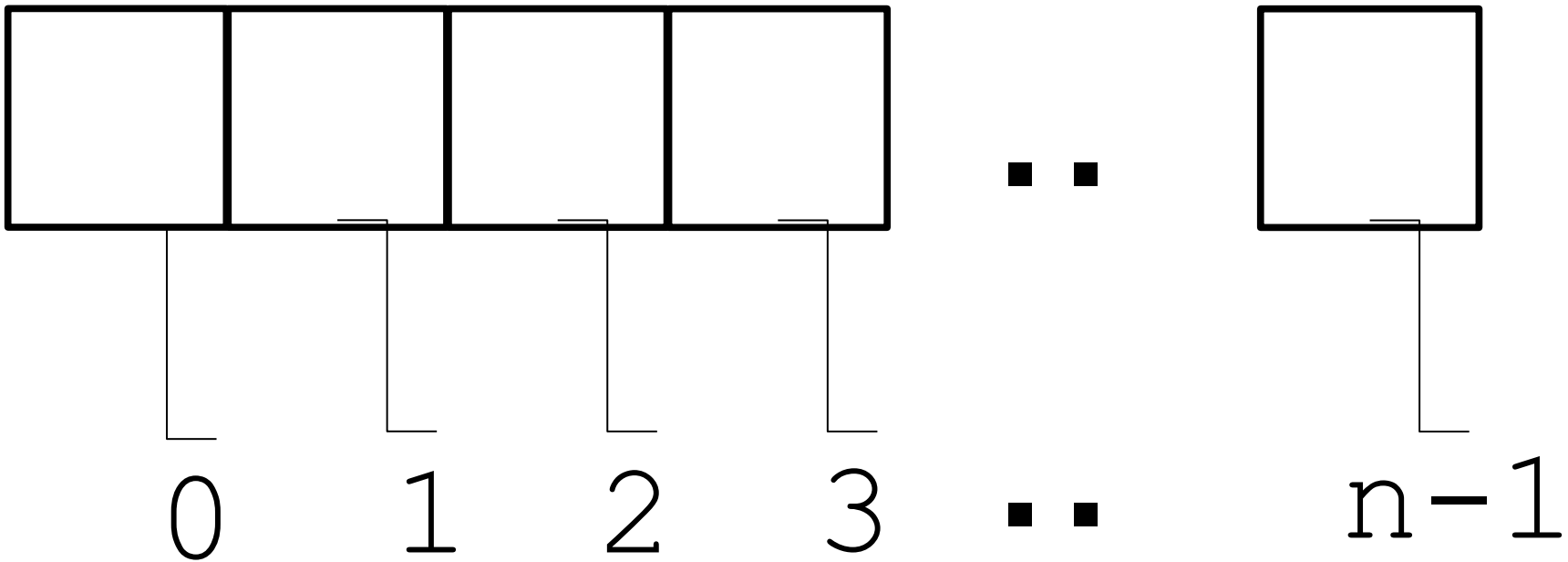
An array is an indexed list of values.

You can make an array of any type

int, double, String, etc..

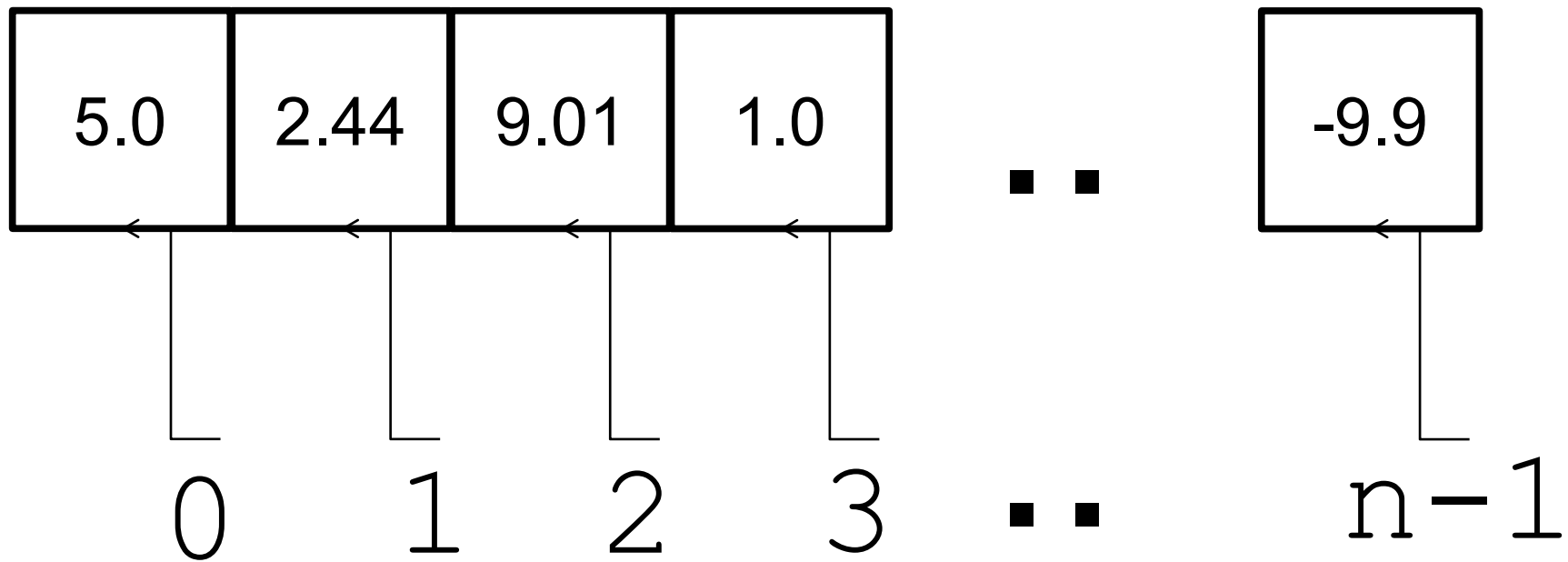
All elements of an array must have the same type.

# Arrays



# Arrays

Example: double [ ]



# Arrays

The index starts at zero and ends at length-1.

Example:

```
int[] values = new int[5];  
values[0] = 12; // CORRECT  
values[4] = 12; // CORRECT  
values[5] = 12; // WRONG!! compiles but  
                // throws an Exception  
                // at run-time
```

Have a demo with runtime exception



# Arrays

An array is defined using TYPE `[]`.

Arrays are just another type.

```
int[] values;    // array of int
```

```
int[][] values;  // int[] is a type
```

# Arrays

To create an array of a given size, use the operator `new` :

```
int[] values = new int[5];
```

or you may use a variable to specify the size:

```
int size = 12;
```

```
int[] values = new int[size];
```

# Array Initialization

Curly braces can be used to initialize an array.

It can **ONLY** be used when you declare the variable.

```
int[] values = { 12, 24, -23, 47 };
```

# Quiz time!

Is there an error in this code?

```
int[] values = {1, 2.5, 3, 3.5, 4};
```

# Accessing Arrays

To access the elements of an array, use the `[]` operator:

```
values[index]
```

Example:

```
int[] values = { 12, 24, -23, 47 };  
values[3] = 18;           // {12, 24, -23, 18}  
int x = values[1] + 3;    // {12, 24, -23, 18}
```

# The *length* variable

Each array has a `length` variable built-in that contains the length of the array.

```
int[] values = new int[12];  
int size = values.length; // 12
```

```
int[] values2 = {1, 2, 3, 4, 5}  
int size2 = values2.length; // 5
```

# String arrays

A side note

```
public static void main (String[] arguments) {  
    System.out.println (arguments.length) ;  
    System.out.println (arguments[0]) ;  
    System.out.println (arguments[1]) ;  
}
```

# Combining Loops and Arrays



# Looping through an array

Example 1:

```
int[] values = new int[5];  
  
for (int i=0; i<values.length; i++) {  
    values[i] = i;  
    int y = values[i] * values[i];  
    System.out.println(y);  
}
```

# Looping through an array

Example 2:

```
int[] values = new int[5];  
int i = 0;  
while (i < values.length) {  
    values[i] = i;  
    int y = values[i] * values[i];  
    System.out.println(y);  
    i++;  
}
```

# Popular Issues 1

- Array **Index** vs Array **Value**

```
int[] values = {99, 100, 101};  
System.out.println(values[0] );    // 99
```

Values	99	100	101
Indexes	0	1	2

# Popular Issues 2

- Curly braces { ... } after `if/else`, `for/while`


```
for (int i = 0; i < 5; i++)  
    System.out.println("Hi");  
    System.out.println("Bye");
```

- What does this print?

# Popular Issues 3

- Variable initialization

```
int getMinValue(int[] vals) {  
    int min = 0;  
    for (int i = 0; i < vals.length; i++) {  
        if (vals[i] < min) {  
            min = vals[i]  
        }  
    }  
}
```

- What if `vals = {1, 2, 3}`?  Problem?
- Set `min = Integer.MAX_VALUE` or `vals[0]`

# Popular Issues 4

- Variable Initialization – secondMinIndex

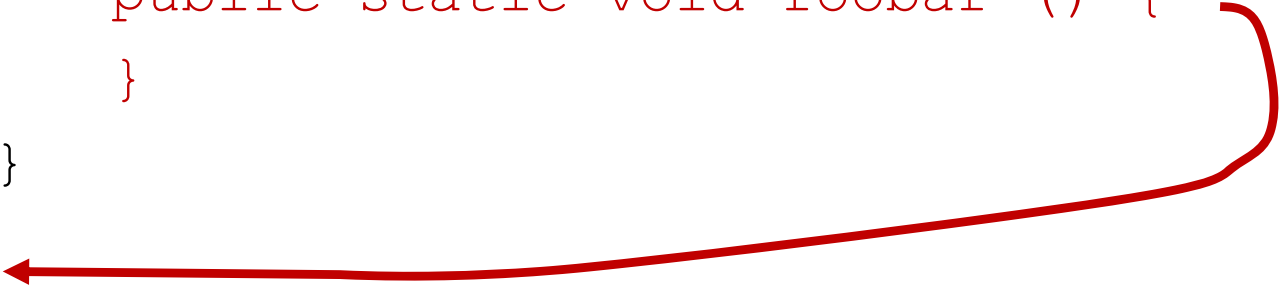
```
int minIdx = getMin(vals)
int secondIdx = 0;
for (int i = 0; i < vals.length; i++) {
    if (i == minIdx) continue;
    if (vals[i] < vals[secondIdx])
        secondIdx = i;
}
```

- What if vals = {0, 1, 2}?
- See solutions

# Popular Issues 5

## Defining a method inside a method

```
public static void main(String[] arguments) {  
    public static void foobar () {  
    }  
}
```



# Debugging Notes 1

- Use `System.out.println` throughout your code to see what it's doing

```
for ( int i=0; i< vals.length; i++) {  
    if ( vals[i] < minVal) {  
        System.out.println("cur min: " + minVal);  
        System.out.println("new min: " + vals[i]);  
        minVal = vals[i];  
    }  
}
```



# Debugging Notes 2

- Formatting
- Ctrl-shift-f is your friend

```
for (int i = 0; i < vals.length; i++) {  
    if (vals[i] < vals[minIdx]) {  
minIdx=i;}  
return minIdx;}
```

- Is there a bug? Who knows! Hard to read

# Today's Topics

Object oriented programming

Defining Classes

Using Classes

References vs Values

Static types and methods

# Today's Topics

Object oriented programming

Defining Classes

Using Classes

References vs Values

Static types and methods

**Whew!**  
That's a lot!

# Object oriented programming

- Represent the real world

Baby

# Object oriented programming

- Represent the real world

Baby

Name

Sex

Weight

Decibels

# poops so far

# Object Oriented Programming

- Objects group together
  - Primitives (int, double, char, etc..)
  - Objects (String, etc...)

## Baby

```
String name  
boolean isMale  
double weight  
double decibels  
int numPoops
```

# Why use **classes**?

- Why not just primitives?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
```

# Why use **classes**?

- Why not just primitives?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
// little baby david
String nameDavid2;
double weightDavid2;
```



David2?  
Terrible 😞



# Why use **classes**?

- Why not just primitives?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
// little baby david
String nameDavid2;
double weightDavid2;
```



David2?  
Terrible 😞

500 Babies? That Sucks!

# Why use **classes**?



Baby1

# Why use **classes**?



Baby1



Baby2



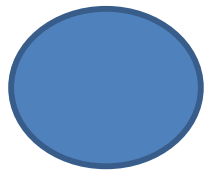
Baby3



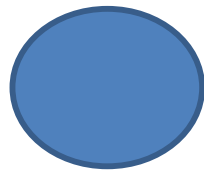
Baby4

496  
more  
Babies  
...

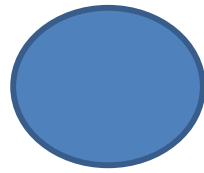
# Why use **classes**?



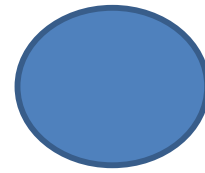
Baby1



Baby2



Baby3

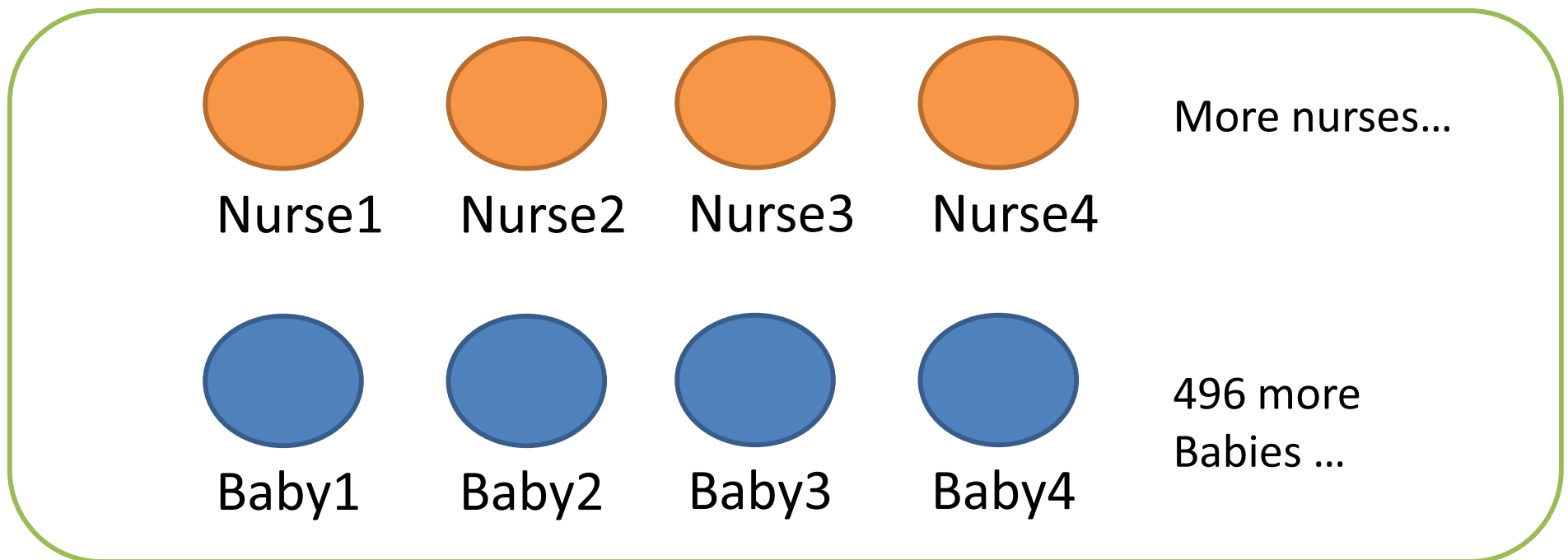


Baby4

496 more  
Babies ...

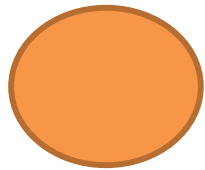
Nursery

# Why use **classes**?



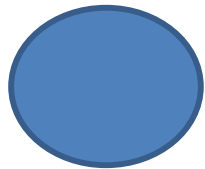
Nursery

# Why use **classes**?



[ ]

Nurse

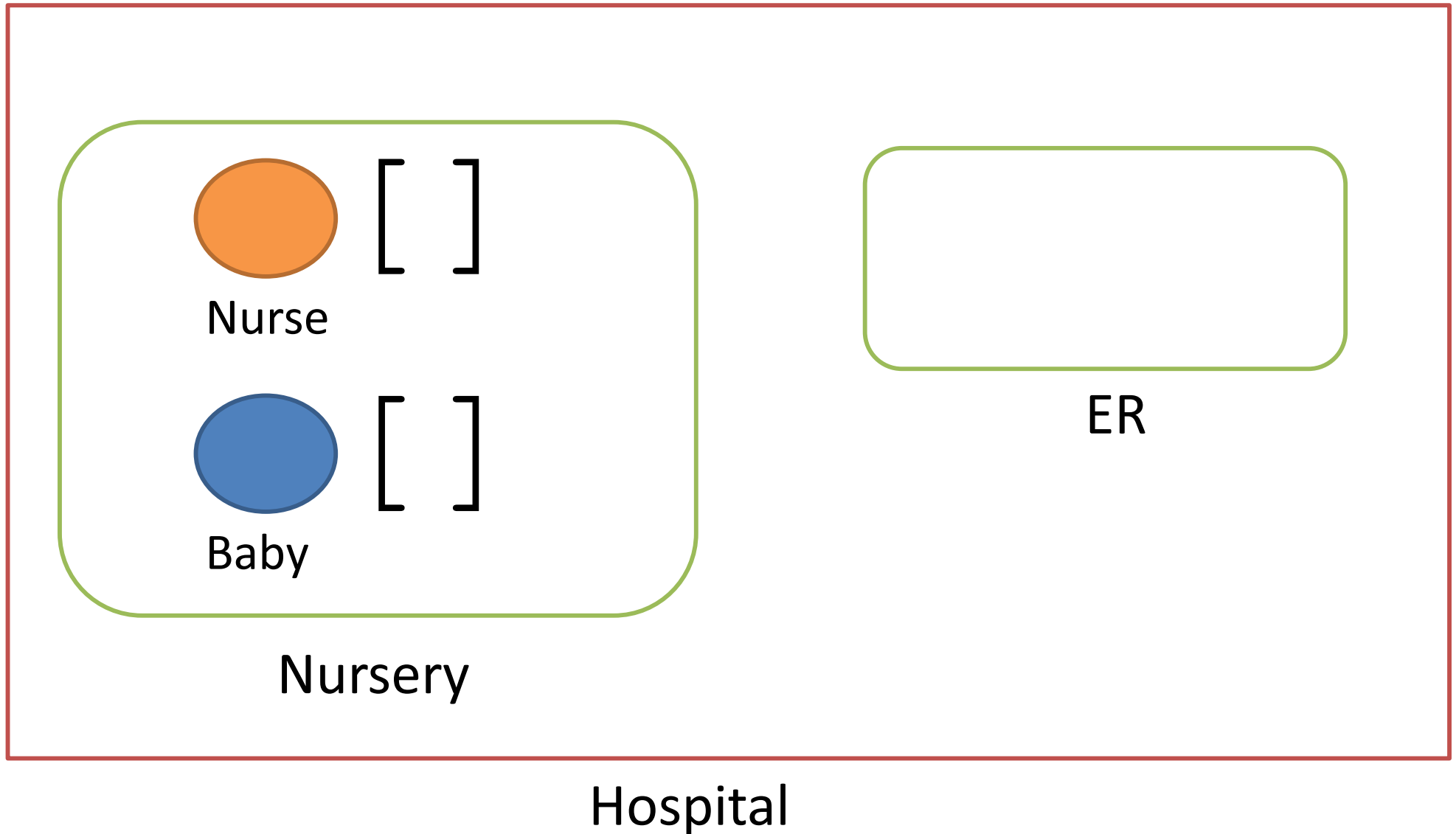


[ ]

Baby

Nursery

# Why use **classes**?



# Defining classes



# Class - overview

```
public class Baby {  
    String name;  
    boolean isMale;  
    double weight;  
    double decibels;  
    int numPoops = 0;  
  
    void poop() {  
        numPoops += 1;  
        System.out.println("Dear mother, "+  
            "I have pooped.  Ready the diaper.");  
    }  
}
```

Class  
Definition

# Class - overview

```
Baby myBaby = new Baby();
```

Class

Instance

# Let's declare a baby!

```
public class Baby {
```

```
}
```

# Let's declare a baby!

```
public class Baby {
```



fields



methods

```
}
```

# Note

- Class names are Capitalized
- 1 Class = 1 file
- Having a `main` method means the class can be run

# Baby fields

```
public class Baby {  
  
    TYPE var_name;  
    TYPE var_name = some_value;  
  
}
```

# Baby fields

```
public class Baby {  
    String name;  
    double weight = 5.0;  
    boolean isMale;  
    int numPoops = 0;  
  
}
```

# Baby Siblings?

```
public class Baby {  
    String name;  
    double weight = 5.0;  
    boolean isMale;  
    int numPoops = 0;  
    XXXXXX    YYYYYY;  
  
}
```



# Baby Siblings?

```
public class Baby {  
    String name;  
    double weight = 5.0;  
    boolean isMale;  
    int numPoops = 0;  
    Baby[] siblings;  
  
}
```

Ok, let's make this baby!

```
Baby ourBaby = new Baby();
```

But what about it's name? it's sex?

# Constructors

```
public class CLASSNAME {  
    CLASSNAME ( ) {  
        }  
  
    CLASSNAME ( [ARGUMENTS] ) {  
        }  
}
```

```
CLASSNAME obj1 = new CLASSNAME ( ) ;  
CLASSNAME obj2 = new CLASSNAME ( [ARGUMENTS] )
```

# Constructors

- Constructor name == the class name
- No return type – never returns anything
- Usually initialize fields
- All classes need at least one constructor
  - If you don't write one, defaults to

```
CLASSNAME  ()  {  
    }  
}
```

# Baby constructor

```
public class Baby {  
    String name;  
    boolean isMale;  
    Baby(String myname, boolean maleBaby) {  
        name = myname;  
        isMale = maleBaby;  
    }  
}
```

# Baby methods

```
public class Baby {  
    String name = "Slim Shady";  
    ...  
    void sayHi() {  
        System.out.println(  
            "Hi, my name is.. " + name);  
    }  
}
```

# Baby methods

```
public class Baby {  
    String weight = 5.0;  
  
    void eat(double foodWeight) {  
        if (foodWeight >= 0 &&  
            foodWeight < weight) {  
            weight = weight + foodWeight;  
        }  
    }  
}
```

# Baby class

```
public class Baby {  
    String name;  
    double weight = 5.0;  
    boolean isMale;  
    int numPoops = 0;  
    Baby[] siblings;  
  
    void sayHi() {...}  
    void eat(double foodWeight) {...}  
}
```



Using classes

# Classes and Instances

```
// class Definition  
public class Baby {...}
```

```
// class Instances  
Baby shiloh = new Baby("Shiloh Jolie-Pitt", true);  
Baby knox   = new Baby("Knox Jolie-Pitt",   true);
```

# Accessing fields

- Object.FIELDNAME

```
Baby shiloh = new Baby("Shiloh Jolie-Pitt",  
                        true)  
System.out.println(shiloh.name);  
System.out.println(shiloh.numPoops);
```

# Calling Methods

- Object.METHODNAME([ARGUMENTS])

```
Baby shiloh = new Baby("Shiloh Jolie-Pitt",  
                        true)  
shiloh.sayHi();      // "Hi, my name is ..."  
shiloh.eat(1);
```

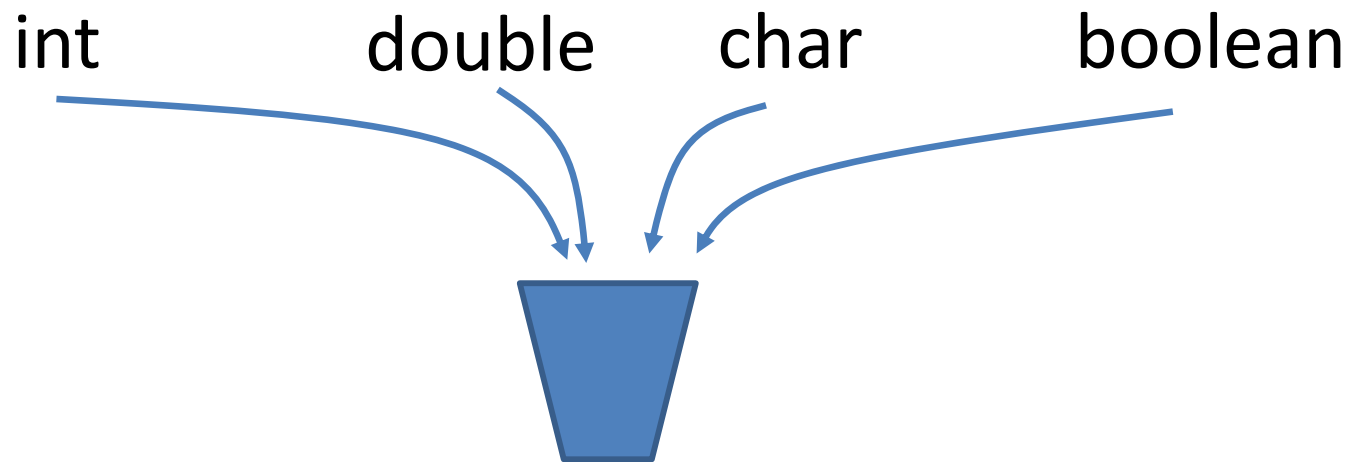
# References vs Values

# Primitives vs References

- **Primitive** types are basic java types
  - int, long, double, boolean, char, short, byte, float
  - The actual **values** are stored in the variable
- **Reference** types are arrays and objects
  - String, int[], Baby, ...

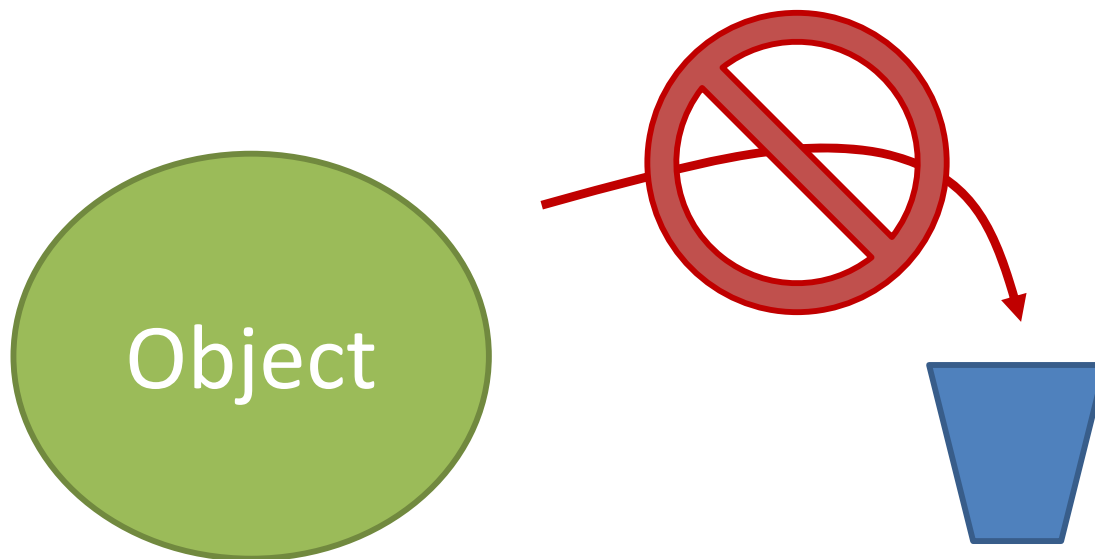
# How java stores **primitives**

- Variables are like fixed size cups
- Primitives are small enough that they just fit into the cup



# How java stores **objects**

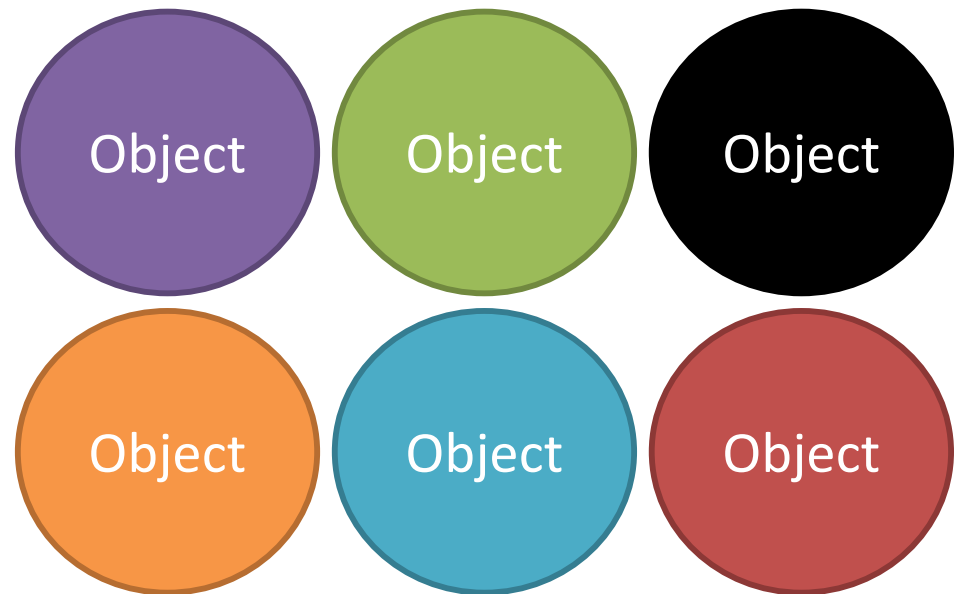
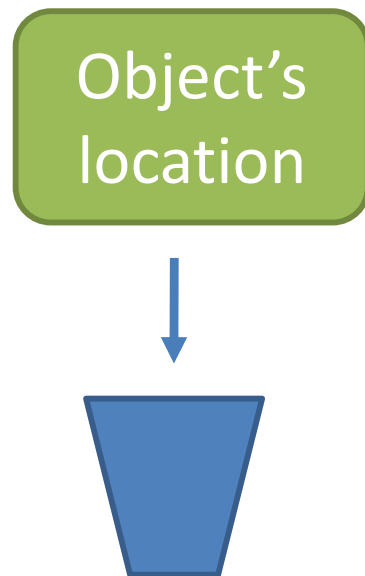
- Objects are too big to fit in a variable
  - Stored somewhere else
  - Variable stores a number that locates the object





# How java stores **objects**

- Objects are too big to fit in a variable
  - Stored somewhere else
  - Variable stores a number that locates the object



# References

- The object's location is called a **reference**
- **==** compares the references

```
Baby shiloh1 = new Baby("shiloh");
```

```
Baby shiloh2 = new Baby("shiloh");
```

**Does** `shiloh1 == shiloh2`?

# References

- The object's location is called a **reference**
- **==** compares the references

```
Baby shiloh1 = new Baby("shiloh");
```

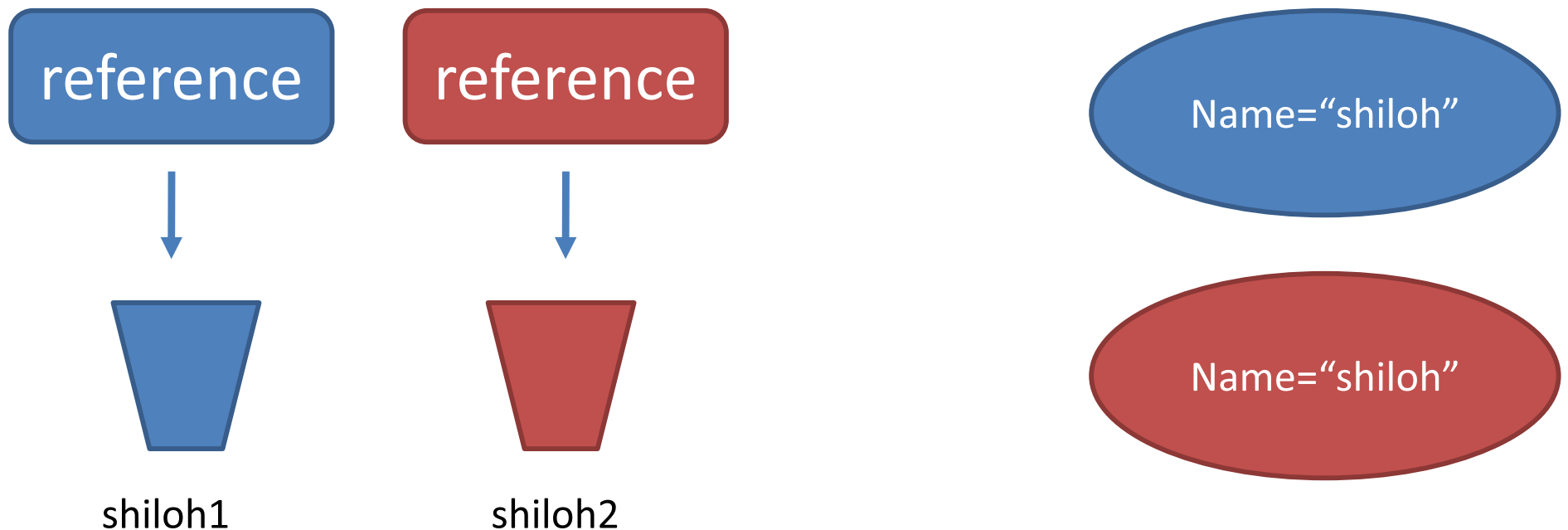
```
Baby shiloh2 = new Baby("shiloh");
```

Does `shiloh1 == shiloh2`?

**no**

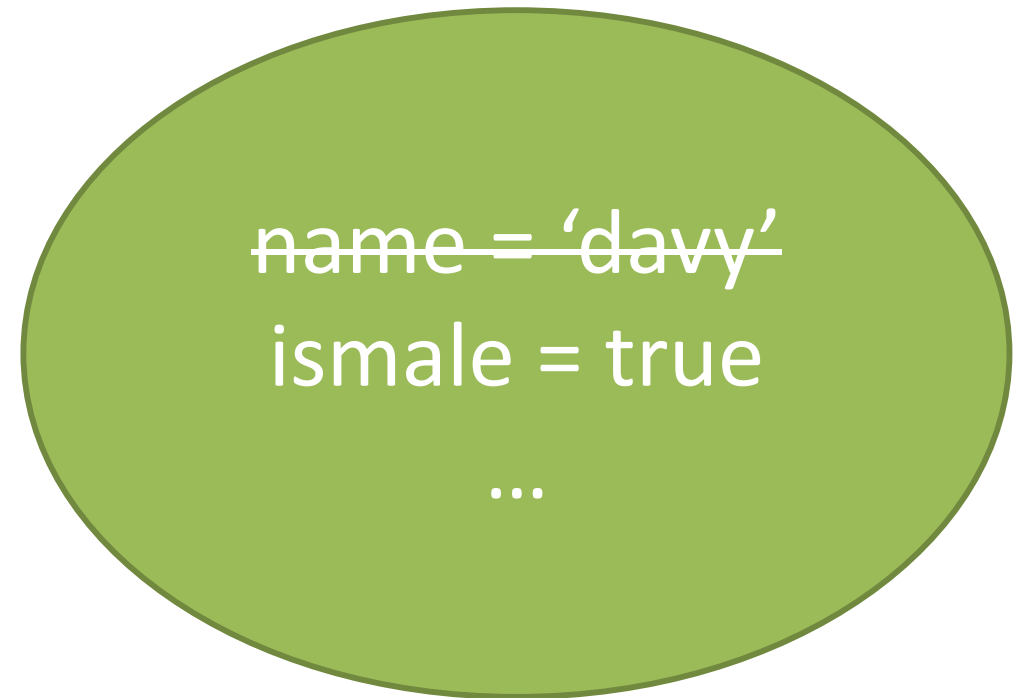
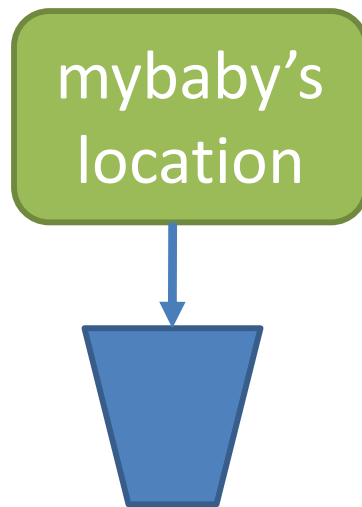
# References

```
Baby shiloh1 = new Baby("shiloh");  
Baby shiloh2 = new Baby("shiloh");
```



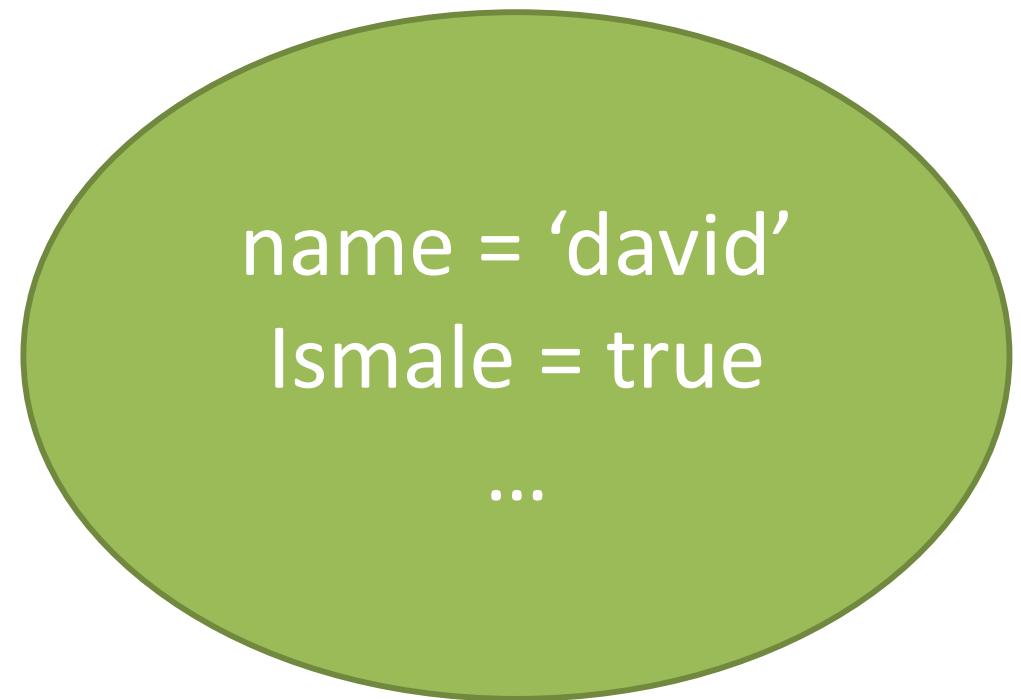
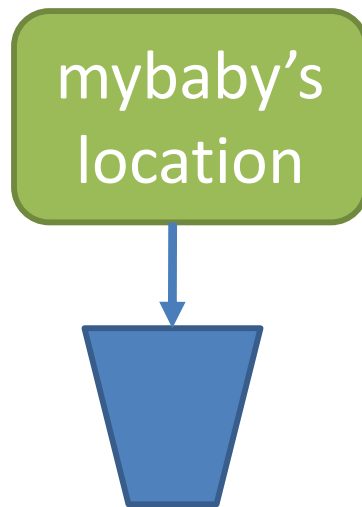
# References

```
Baby mybaby = new Baby("davy", true)  
mybaby.name = "david"
```



# References

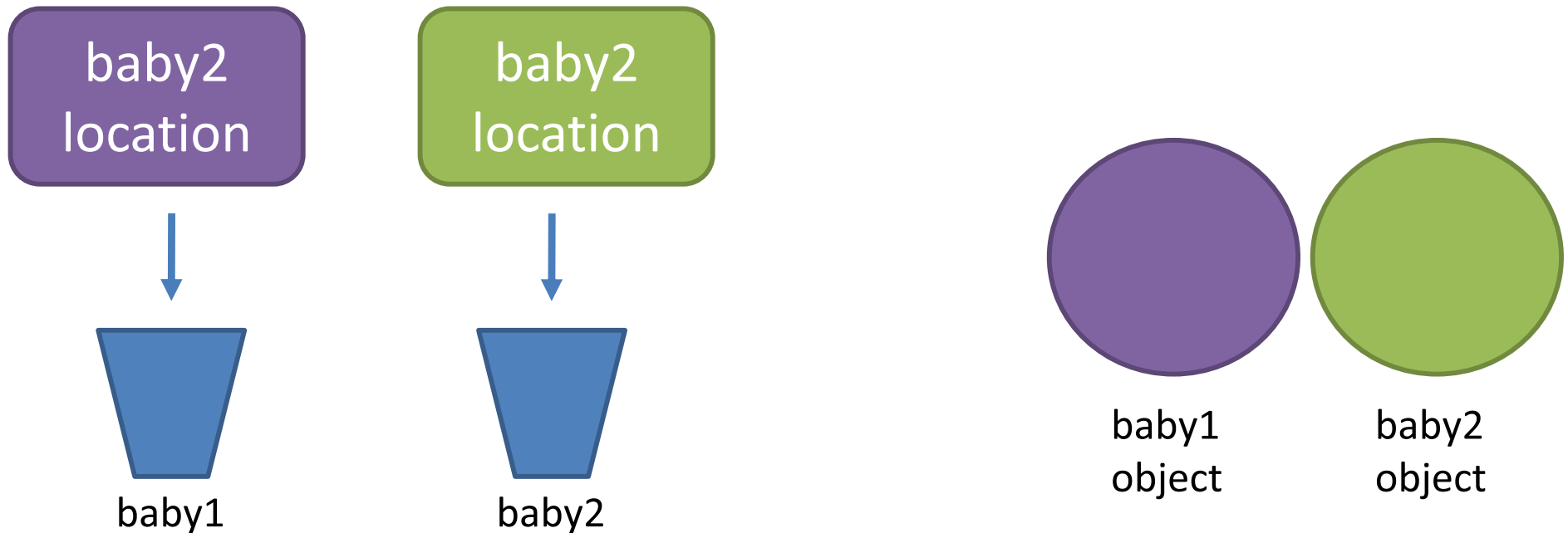
```
Baby mybaby = new Baby( 'davy', true)  
mybaby.name = 'david'
```



# References

- Using = updates the reference.

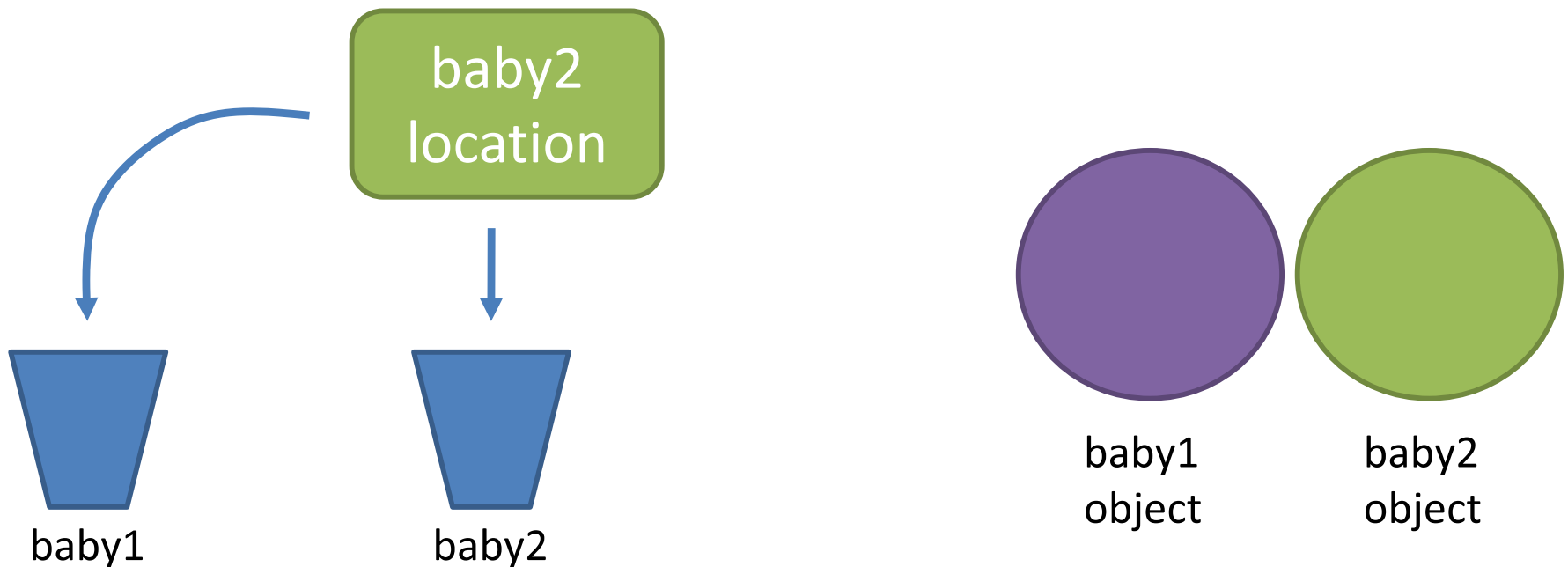
```
baby1 = baby2
```



# References

- Using = updates the reference.

```
baby1 = baby2
```





# References

- using [ ] or •
  - Follows the reference to the object
  - May modify the object, but never the reference
- Imagine
  - Following directions to a house
  - Moving the furniture around
- Analogous to
  - Following the reference to an object
  - Changing fields in the object

# Methods and references

```
void doSomething(int x, int[] ys, Baby b) {  
    x = 99;  
    ys[0] = 99;  
    b.name = "99";  
}
```

...

```
int i = 0;  
int[] j = {0};  
Baby k = new Baby("50", true);  
doSomething(i, j, k);
```

i=? j=? k=?

static types and methods

# static

- Applies to fields and methods
- Means the field/method
  - Is defined for the class declaration,
  - Is not unique for each instance

# static

```
public class Baby {  
    static int numBabiesMade = 0;  
}  
Baby.numBabiesMade = 100;  
Baby b1 = new Baby();  
Baby b2 = new Baby();  
Baby.numBabiesMade = 2;
```

What is

b1.numBabiesMade?

b2.numBabiesMade?

# static example

- Keep track of the number of babies that have been made.

```
public class Baby {  
    int numBabiesMade = 0;  
    Baby() {  
        numBabiesMade += 1;  
    }  
}
```

# static field

- Keep track of the number of babies that have been made.

```
public class Baby {  
    static int numBabiesMade = 0;  
    Baby() {  
        numBabiesMade += 1;  
    }  
}
```

# static method

```
public class Baby {  
    static void cry(Baby thebaby) {  
        System.out.println(thebaby.name + "cries");  
    }  
}
```

Or

```
public class Baby {  
    void cry() {  
        System.out.println(name + "cries");  
    }  
}
```



# static notes

- Non-static methods can reference static methods, but not the other way around
  - Why?

```
public class Baby {  
    String name = "DMX";  
    static void whoami() {  
        System.out.println(name);  
    }  
}
```