Mahidol University
Faculty of Information
and Communication Technology

LECTURE 08
Midterm Review

ITCS123 Object Oriented Programming

Dr. Siripen Pongpaichet
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

Topic

- Variables and Data Types in OOPs
- Decision and Iteration
- Class and Object
- Array and ArrayList
- Inheritance
- Polymorphism

Variables and Data Types in OOPs

- Variables
- Primitive Types – Declaration, Casting
- Reference Types – Declaration, Parsing

Variables

- A **variables** is a storage location in a computer program. We can use a variable to store *values* or *objects* to be used later.

How to declare variable in JAVA?

```
// Syntax for variable declaration:

DataType variableName = value;

// or

DataType VariableName;
```

- **Data Type** is an attribute to describe data contained in a declared variable.
- Every variable in JAVA **must** specify its data type at declaration.
- This type **never** changes!

```
// Example
int age = 10;      // The variable age has type 'int' and the initial value is 10;
age = 22;          // The value 22 is an integer, so it can be stored in variable age
age = "Java"       // ERROR! The value Java is not an integer
```

How to choose DataType?

- In practice, we will mostly use:

`boolean`	to represent logic	**primitive data type.**
`int`, `long` and `double`	to represent numbers	**primitive data type.**
`String` (chains of `char`)	to represent text	**Reference data type.**
`ObjectName`	to represent object	**Reference data type.**

e.g., `String text = "Welcome to Java";`

Data Types

- In Java, every variable either
  - a reference to an **object** (class-types)

    e.g. Car myCar = new Car()

    String text = "String is an object";

  - belongs to one of **8 primitive types**

    ```
    e.g. int number = 25;
         boolean check = true;
    ```

How to notice:
1. Type starts with a capital letter:
   Car
2. Normal font style in Eclipse

How to notice:
1. Type starts with a lowercase letter:
   int, ***b***oolean
2. **Purple-bold** font style in Eclipse

Variable name	Value in memory
myCar	0x110
text	0x111
number	25
check	true

0x111

"String is an object"

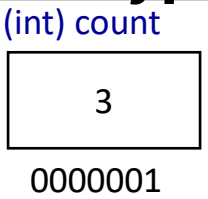
Data Types

Primitive data types

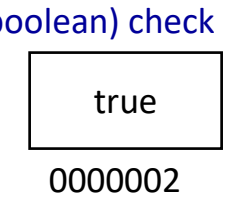
- **Primitive Data Type** is a basic data types to describe a variable. Variable declared by primitive data type will store a value inside variable.

- Example of **primitive data type** declaration:

```
int count = 3;
```




```
boolean check = true;
```



Note that: Keyword is store value inside variable.

Eight Primitive Data Types

Table 1 Primitive Types

Type	Description	Size
`int`	The integer type, with range –2,147,483,648 (`Integer.MIN_VALUE`) . . . 2,147,483,647 (`Integer.MAX_VALUE`, about 2.14 billion)	4 bytes
`byte`	The type describing a single byte, with range –128 . . . 127	1 byte
`short`	The short integer type, with range –32,768 . . . 32,767	2 bytes
`long`	The long integer type, with range –9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807	8 bytes
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
`char`	The character type, representing code units in the Unicode encoding scheme (see Computing & Society 4.2 on page 163)	2 bytes
`boolean`	The type with the two truth values `false` and `true` (see Chapter 5)	1 bit

Data Types

Primitive data types

- **Converting one primitive data types to** the other**..**
  - In JAVA you cannot assign data type **double** to **int**

e.g.

```
double balance = total + tax;
int dollars = balance;
//Error: Cannot assign double to int.
```

To converting primitive datatype, the **cast** operator **(int)** is used:

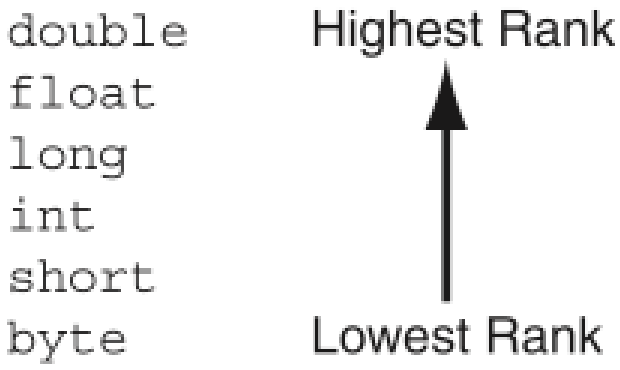
```
double balance = total + tax;
int dollars = (int) balance;
```

****In this case**: if balance is 13.75, then dollars is set to 13.

Casts

Widening Conversion

(automatically convert)

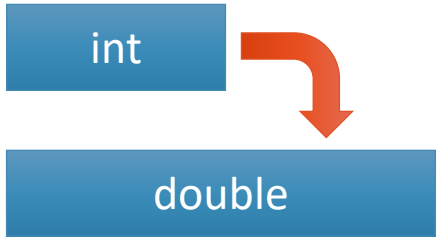


Narrowing Conversion

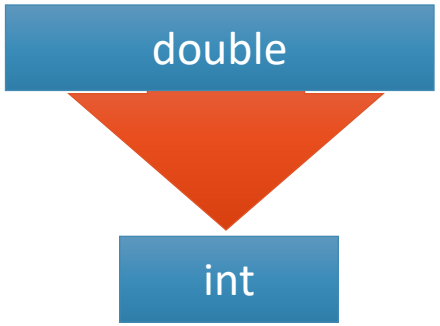
Requires CAST operator

Primitive Data Type Ranking

```
int  y = 10;
double x;
x = y;
```




```
double x = 72.456
int y;
y = (int) x;   // 72
```



Casts

How about casting number types to String class type?

- Used to convert a value or an expression to a different type
- Syntax `(typeName) expression;`

```
double balance = 13.75;
int dollars = (int) balance;          // OK, dollars = 13
int amountX = (int) balance * 100;    // amount = 1300
int amountY = (int) (balance * 100);   // amount = 1375
```

Cast discards fractional part

- `Math.round` converts a floating-point number to nearest integer

```
long rounded = Math.round(balance);
  // if balance is 13.75, then rounded is set to 14
  // if balance is 13.5, then rounded is set to 14
```

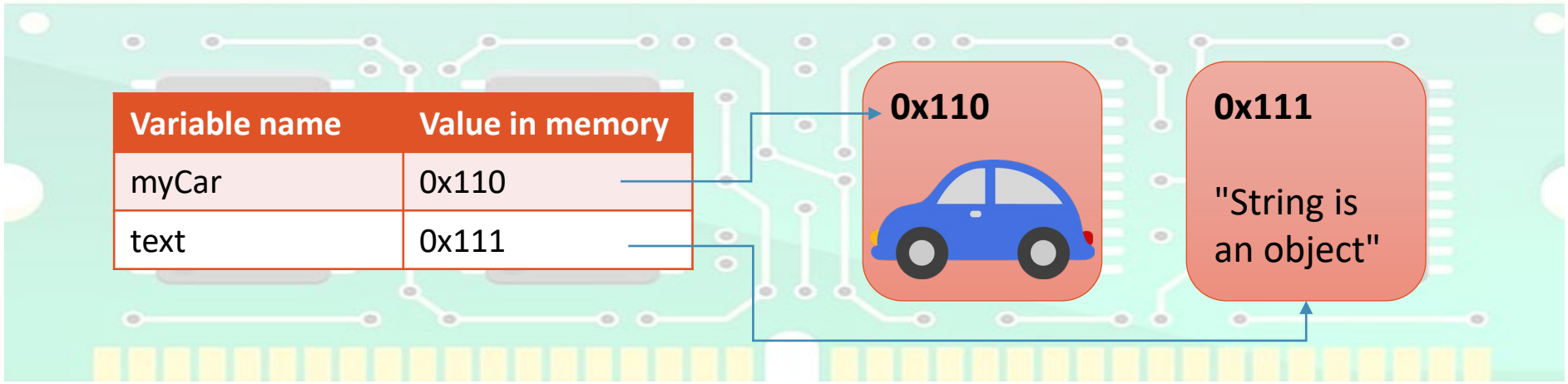
Reference Data Type

- **Reference Data Type** is a data types of a Class in which the value of variable contain reference to the actual value.

Example of reference data type declaration:

```
Car car1 = new Car();
String text = "String is an obj";
```

*** Variable stores an address to the actual value ***



Reference Data Type

- There are two types of class
  - **Pre-defined Class**
    - Wrapper Classes (from 8 premitive types)
      - Integer, Byte, Short, Long, Double, Float, Character, Boolean
    - Other Classes
      - For example: String, ArrayList, …
  - **User-defined Class**
    - For example: Car, Dog, Letter, …

Wrapper Classes

```
Integer i = 3;

Character c = 'C';

Double d = 1.2;
```

- Why do we need them?
  - Used with Collection object such as ArrayList<Object>
    - `ArrayList<Integer>` is correct but `ArrayList<int>` is error
  - Easily convert their value into String by calling `toString()` method

Data Types

Reference data types

- Converting one **Reference data types** to **Primitive data type…**

- To convert reference variable to primitive variable, the **parse** method of the (converted data type) wrapper class is used:

For example: Converting “**String**” to “**int**”,

```
String speed = "0202";
int speed_int = Integer.parseInt(speed);
```

Strings (1/3)

- `String` is a sequence of characters.

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

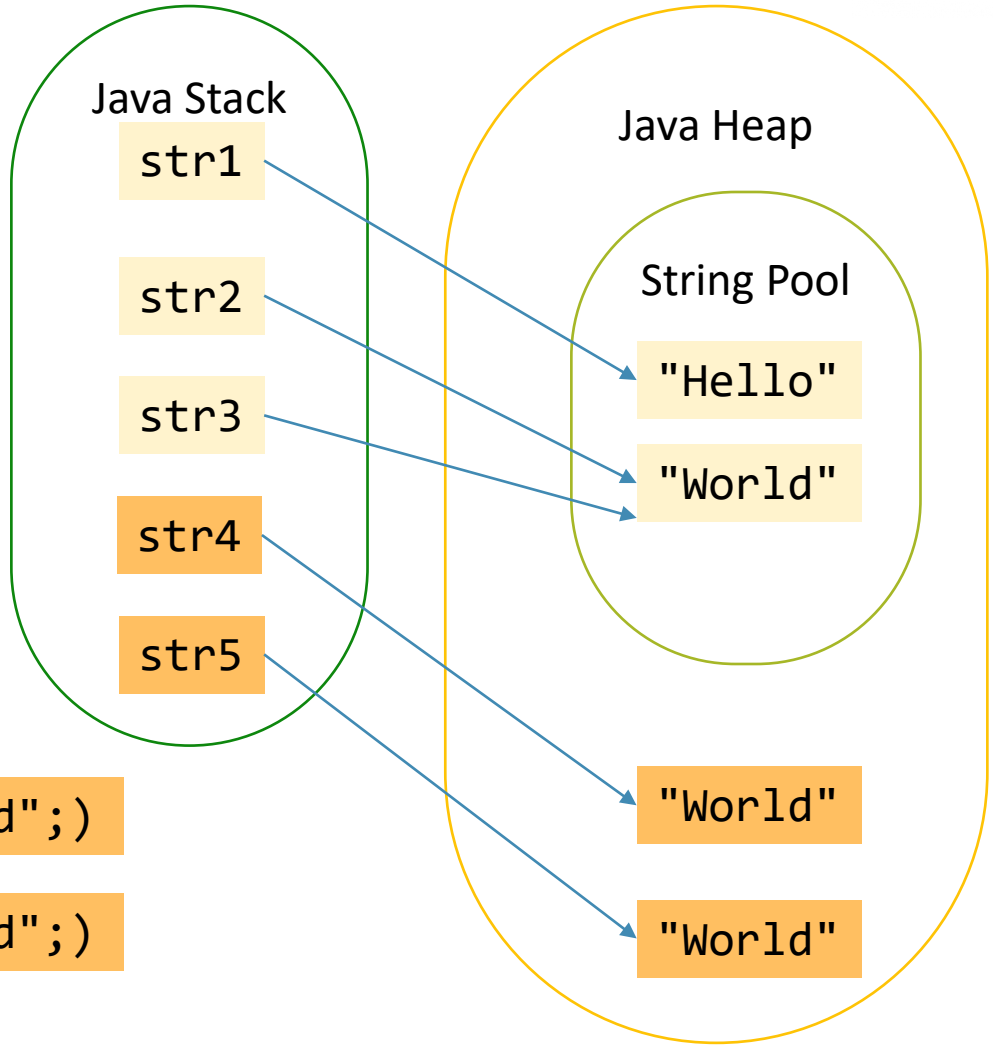
- Strings, though not of a primitive type, are frequently used
  - A variable created from the `String` class also count as an object reference.
- However, String is special. It can be created directly as primitive data type and also create by using new keyword
- Declaring String:

```
String message1 = "ABC";
String message2 = new String ("XYZ");
```

- Initializing string with empty string:

```
String message3 = ""; OR
String message4 = new String();
```

```
String str1 = "Hello";
String str2 = "World";
String str3 = "World";
```




```
String str4 = new String("World";)
String str5 = new String("World";)
```

NOTE

```
str2 == str3        // true
str4 == str5        // false
```

Strings (2/3)

- Since `String` is a class, it contains several attributes/fields, constructors and methods

  → You may search for String API for more detail

Statement	Result	Comment
`string str = "Ja";` `str = str + "va";`	`str` is set to `"Java"`	When applied to strings, + denotes concatenation.
`System.out.println("Please"` `+ " enter your name: ");`	Prints `Please enter your name:`	Use concatenation to break up strings that don't fit into one line.
`team = 49 + "ers"`	`team` is set to `"49ers"`	Because `"ers"` is a string, 49 is converted to a string.

`String first = in.next();`
`String last = in.next();`
`(User input: Harry Morgan)`
	`first` contains `"Harry"`
	`last` contains `"Morgan"`
		The `next` method places the next word into the string variable.

`String greeting = "H & S";` `int n = greeting.length();`	`n` is set to 5	Each space counts as one character.

Strings (3/3)

Statement	Result	Comment
`String str = "Sally";` `char ch = str.charAt(1);`	ch is set to 'a'	This is a char value, not a String. Note that the initial position is 0.
`String str = "Sally";` `String str2 = str.substring(1, 4);`	str2 is set to "all"	Extracts the substring starting at position 1 and ending before position 4.
`String str = "Sally";` `String str2 = str.substring(1);`	str2 is set to "ally"	If you omit the end position, all characters from the position until the end of the string are included.
`String str = "Sally";` `String str2 = str.substring(1, 2);`	str2 is set to "a"	Extracts a String of length 1; contrast with str.charAt(1).
`String last = str.substring(` `  str.length() - 1);`	last is set to the string containing the last character in str	The last character has position str.length() - 1.

Example

```
String message = "Java is Great Fun!";

String upper = message.toUpperCase();    // JAVA IS GREAT FUN!

char letter = message.charAt(2);         // v

String sub2 = message.substring(5, 10);     // is Gr

String sub3 = message.substring(10, 5);     // ERROR!

String lastTwoChar = message.substring (message.length()-2);  // n!

String sub4 = message.substring (8, message.length()-2); //Great Fu
```

J	a	v	a		i	s		G	r	e	a	t		F	u	n	!
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Index start with 0

Strings Concatenation

- Use the + operator:

```
String name = "Dave";
String message = "Hello, " + name;
                          // message is "Hello, Dave"
```

- If one of the arguments of the + operator is a string, the other is converted to a string

```
String a = "Agent";
int n = 7;
String bond = a + n;  // bond is "Agent7"
```

- Reduce print statements

```
System.out.print("The total is ");
System.out.println(total);
```

```
System.out.print("The total is " + total);
```

Strings Conversion

- Convert to number:

```
String str = "19";
int n = Integer.parseInt(str);


String str2 = "3.95";
double x = Double.parseDouble(str2);
```

- Convert to string:

```
String str = "" + n;

       -------OR-------

String str = Integer.toString(n);
```

Decision and Iteration

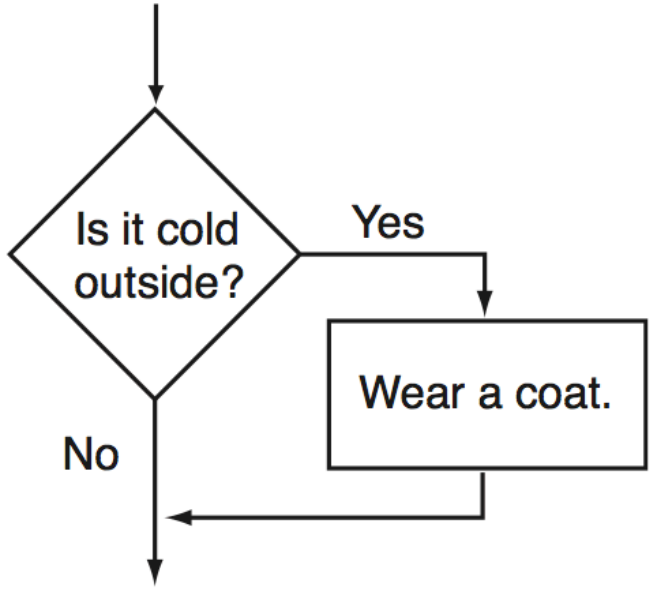
- **`if/else`** statement
- Multiple Alternatives (**`if/elseif`**)
- **`switch`** statement

- **`while`** loop
- **`for`, `for-each`** loop
- Nested loop
- **`do-while`** loop

`if` Statement

- The if statement is used to create a decision structure, which allows a program to have more than one path of execution.
- The if statement causes *one or more* statements to execute only when a `boolean expression` is `true`.

Simple decision structure logic




```
if (BooleanExpression)
    statement;
```

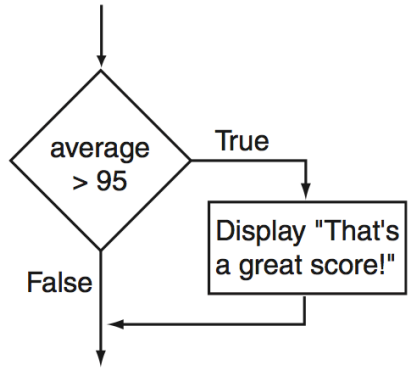
This action is ***conditionally executed*** because it is performed only when a certain condition (cold outside) exists.

Example 1: Single Statement

No semicolon (;) here

```
if (average > 95){
    System.out.println("That's a great score!");
}
```

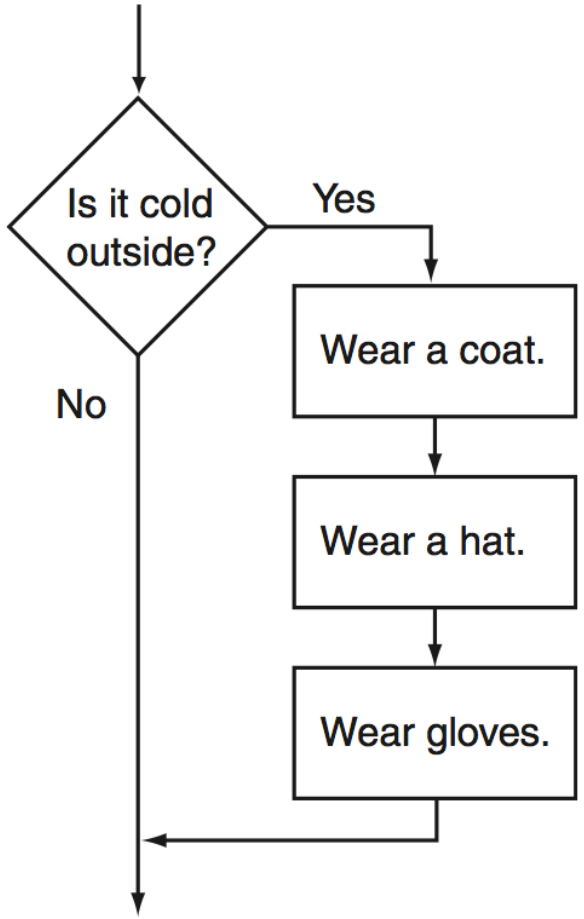
Note: If there is only 1 conditional executed statement, you may ignore the { }



Statement	Outcome
`if (hours > 40)` `  overTime = true;`	If `hours` is greater than 40, assigns `true` to the `boolean` variable `overTime`.
`if (value < 32)` `  System.out.println("Invalid number");`	If `value` is less than 32, displays the message "Invalid number"

`if` Statement

Three-action decision structure logic




```
if (BooleanExpression){
    statement_1;
    statement_2;
    statement_3;
        …
    statement_n;
}
```

Example 2: Block of Statements

Enclosing a group of statements inside braces { } creates a *block* of statements.

```
if (sales > 50000) {
    bonus = 500.0;
    commissionRate = 0.12;
    daysOff += 1;
}
```

An If statement missing its braces

```
if (sales > 50000)
  bonus = 500.0;
commissionRate = 0.12;
daysOff += 1;
```

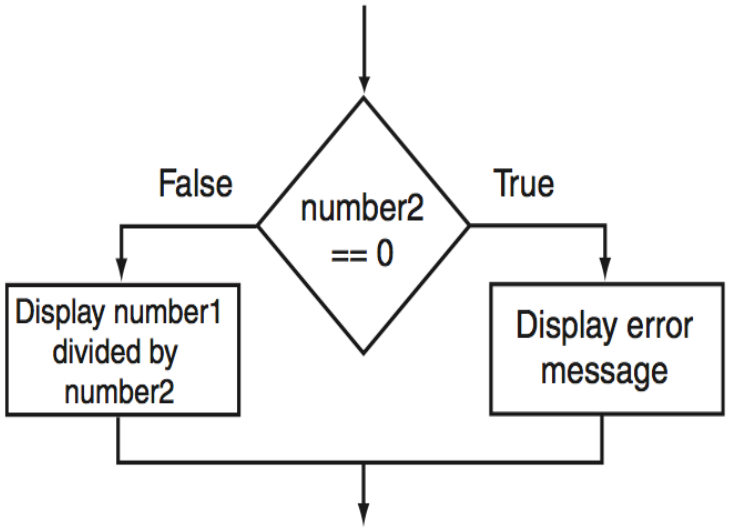
Only this statement is conditionally executed.

These statements are *always* executed.

if-else statement

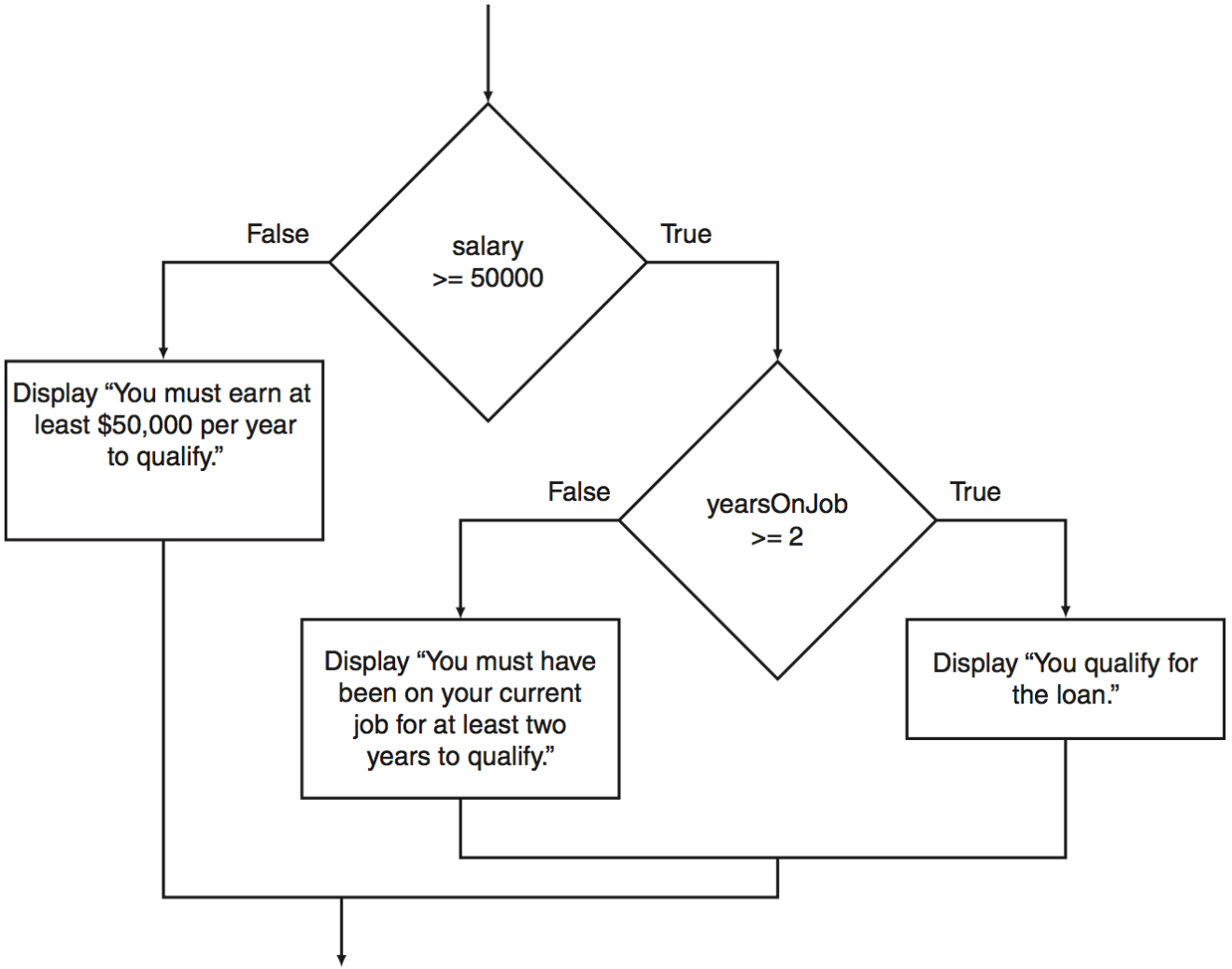
- The `if-else` statement is an expansion of `if` statement
- It will execute one group of statements if its boolean expression is `true`, or another group if its boolean expression is `false`.

```
if (BooleanExpression)
    statement or block;
else
    statement or block;
```



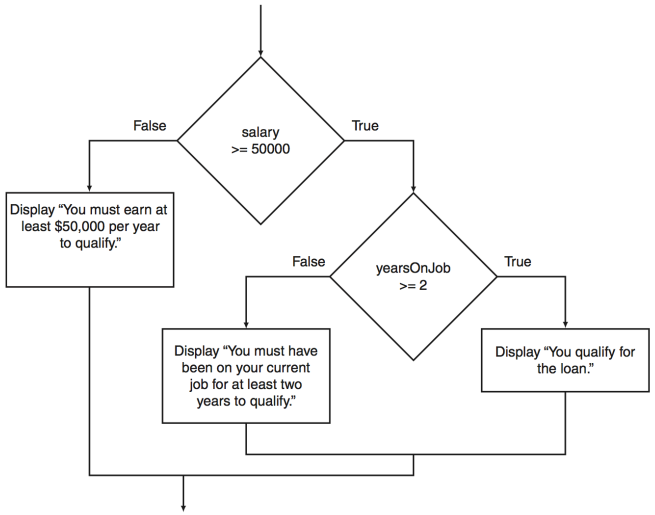

```
if (number2 == 0){
    System.out.println("Number1 cannot
            be divided by 0.");
} else{
    System.out.println(number1/number2);
}
```

Nested `if-else` Statement




```
if (booleanExpression1) {
    if(booleanExpression2)
        statement or block;
    else
        statement or block;
}
else
    statement or block;
```

Nested `if-else` Statement



This `if` and `else` go together.

This `if` and `else` go together.

```
if (salary >= 50000)
{
    if (yearsOnJob >= 2)
    {
        System.out.println("You qualify for the loan.");
    }
    else
    {
        System.out.println("You must have been on your "
                         + "current job for at least "
                         + "two years to qualify.");
    }
}
else
{
    System.out.println("You must earn at least "
                     + "$50,000 per year to qualify.");
}
```

if-else-if Statement

The `if-else-if` statement tests a series of conditions.

It is often simpler to test a series of conditions with the `if-else-if` statement than with a set of nested `if-else` statements.

```
if (expression_1)
{
  statement
  statement
  etc.
}
```

If `expression_1` is true these statements are executed, and the rest of the structure is ignored.

```
else if (expression_2)
{
  statement
  statement
  etc.
}
```

Otherwise, if `eexpression_2` is true these statements are executed, and the rest of the structure is ignored.

Insert as many `else if` clauses as necessary

```
else
{
  statement
  statement
  etc.
}
```

Do not omit '`else`'

These statements are executed if none of the expressions above are true.

Nested If-Else vs. If-Else-If Statement

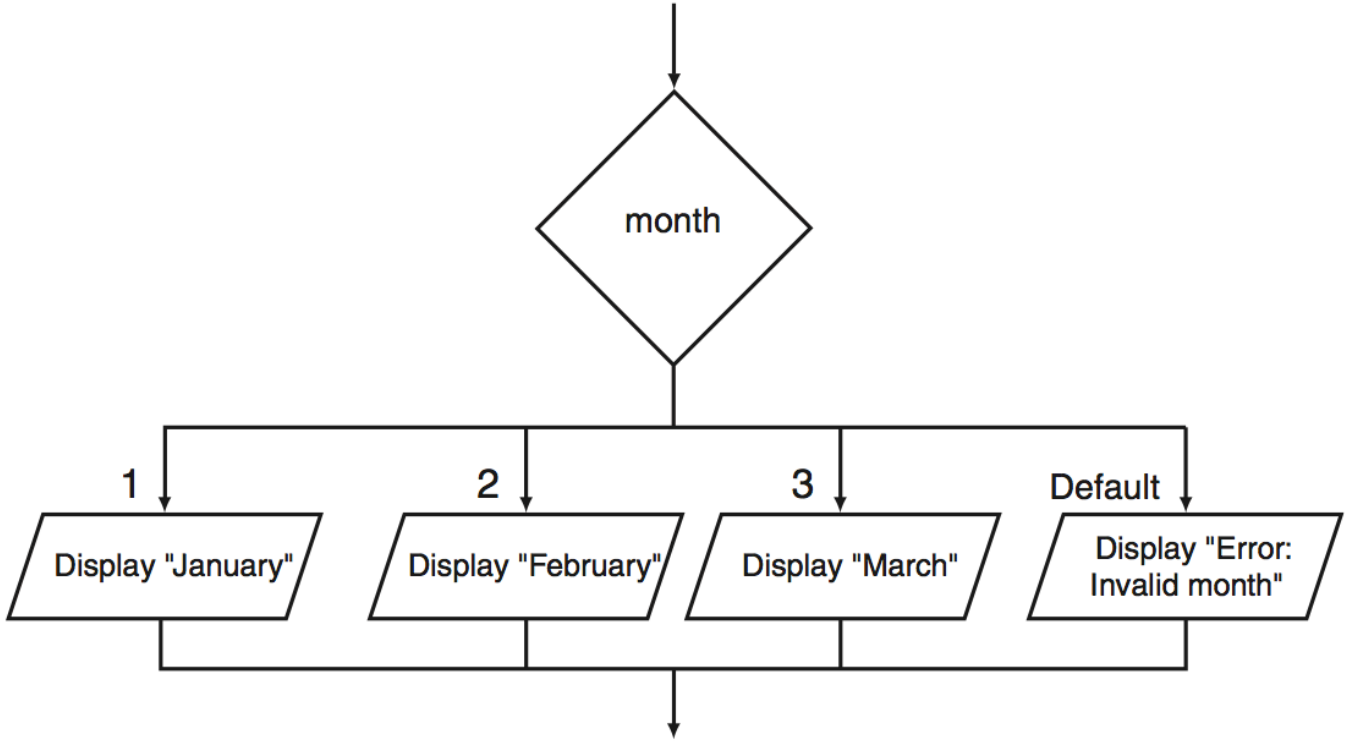
```
char grade;

if (score < 60)
{
   grade = 'F';
}
else
{
   if (score < 70)
   {
      grade = 'D';
   }
   else
   {
      if (score < 80)
      {
         grade = 'C';
      }
      else
      {
         if (score < 90)
         {
            grade = 'B';
         }
         else
         {
            grade = 'A';
         }
      }
   }
}
```

If the score is less than 60, then the grade is F.
Otherwise, if the score is less than 70, then the grade is D.
Otherwise, if the score is less than 80, then the grade is C.
Otherwise, if the score is less than 90, then the grade is B.
Otherwise, the grade is A.

```
char grade;

if (score < 60)
   grade = 'F';
else if (score < 70)
   grade = 'D';
else if (score < 80)
   grade = 'C';
else if (score < 90)
   grade = 'B';
else
   grade = 'A';
```

`switch` Statement

- It also use to create *multiple alternative decision structure* by allowing a variable to be tested for equality against a list of values.
- Each value is called a case, and the variable being switched on is checked for each case.



The `testExpression` is a variable or expression.
that gives a char, byte, short, int, or string value. (NOT BOOLEAN)

```
switch (testExpression)
{
    case value_1:
        statement;
        statement;
        etc.
        break;

    case value_2:
        statement;
        statement;
        etc.
        break;

    Insert as many case sections as necessary.

    case value_N:
        statement;
        statement;
        etc.
        break;

    default:
        statement;
        statement;
        etc.
        break;
}
```

These statements are executed if the `testExpression` is equal to `value_1`.

These statements are executed if the `testExpression` is equal to `value_2`.

These statements are executed if the `testExpression` is equal to `value_N`.

(Optional)

These statements are executed if the `testExpression` is not equal to any of the `case` values.

```
switch (month) {
  case 1:
    System.out.println("Jan");
    break;
  case 2:
    System.out.println("Feb");
    break;
  case 3:
    System.out.println("Mar");
    break;
  default:
    System.out.println("---");
    break;
}
```

```
if (month == 1)
    System.out.println("Jan");
else if (month == 2)
    System.out.println("Feb");
else if (month == 3)
    System.out.println("Mar");
else
    System.out.println("---");
```

switch Statement (without break;)

```
input = keyboard.nextLine();
feedGrade = input.charAt(0);  // Get the first char.

// Determine the grade that was entered.
switch(feedGrade)
{
   case 'a':
   case 'A':
      System.out.println("30 cents per lb.");
      break;
   case 'b':
   case 'B':
      System.out.println("20 cents per lb.");
      break;
   case 'c':
   case 'C':
      System.out.println("15 cents per lb.");
      break;
   default:
      System.out.println("Invalid choice.");
}
```

When the user enters 'a' the corresponding case has no statements associated with it, so the program ***falls through*** to the next case, which corresponds with 'A'. Same technique for 'b' and 'c'.

So, if the input for a user is

A [Enter]
Output: 30 cents per lb.

B [Enter]
Output: 20 cents per lb.

b [Enter]
Output: 20 cents per lb.

c [Enter]
Output: 15 cents per lb.

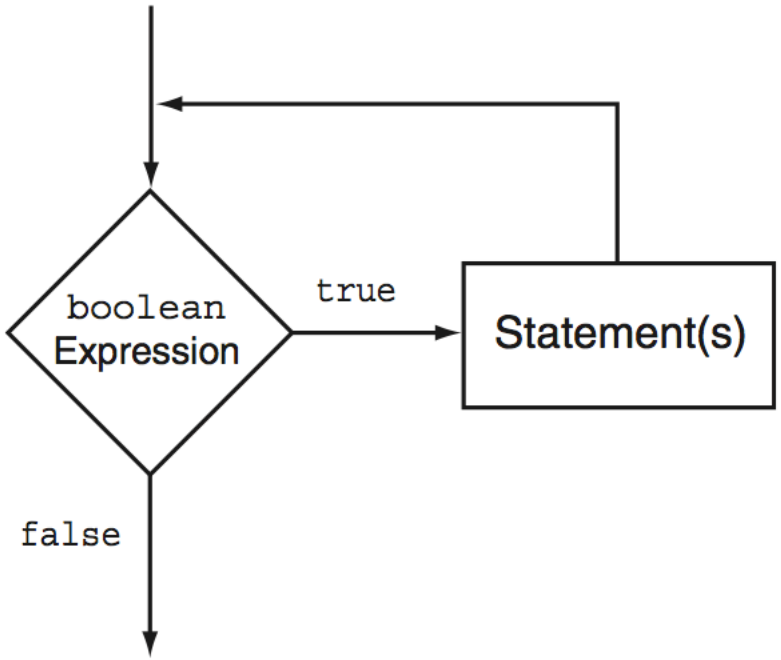
d [Enter]
Output: Invalid choice.

Iteration

- While Loop
- For Loop
- Do Loop

while Loop (1/2)

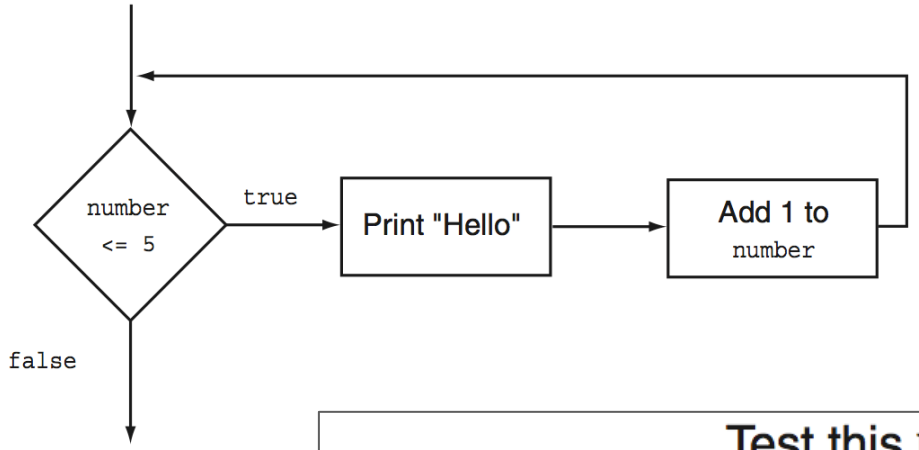
- While loop: a statement for repeatedly executes a body statement as long as a given condition is true
- Pretest loop




```
while (BooleanExpression)
    statement;
```

```
while (BooleanExpression) {
    statement;
    statement;
    // Place as many statements
    // here as necessary.
}
```

while Loop (2/2)



Test this `boolean` expression.

```
while (number <= 5)
{
      System.out.println("Hello");
      number++;
}
```

If the `boolean` expression is `true`, perform these statements.

After executing the body of the loop, start over.

Example 1

- Print event number from 0 to 8

```
int i = 0;
while (i < 5) {
  System.out.println(i * 2);
}
/* Output:
0
2
4
6
8
*/
```

Example 2

- If you deposit $10,000 with an interest of 5%, how many year until you have more than $20,000

```
double balance = 10000;
double targetBalance = 20000;
int years = 0;
while (balance < targetBalance) {
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
System.out.println(years);
/* Output:
15
*/
```

Tracing:

1. 1st iteration: `balance` : 10000 < `targetBalance` : 20000 ☑
   `years` : 1
   `balance` : 10500
2. 2nd iteration: `balance` : 10500 < `targetBalance` : 20000 ☑
   `years` : 2
   `balance` : 11025
3. ...
4. 16th iteration `balance` : 20789.28 < `targetBalance` : 20000 ⛔
   stops the loop

for Loop

- For Loop: an execution of a body statements in a specific number of times.
- It is useful when you know how many times a task is to be repeated.

```
for (initialization; condition; update)
    statement;
```

```
for (initialization; condition; update) {
    statement;
    statement;
    // Place as many statements here
    // as necessary.
}
```

for Loop

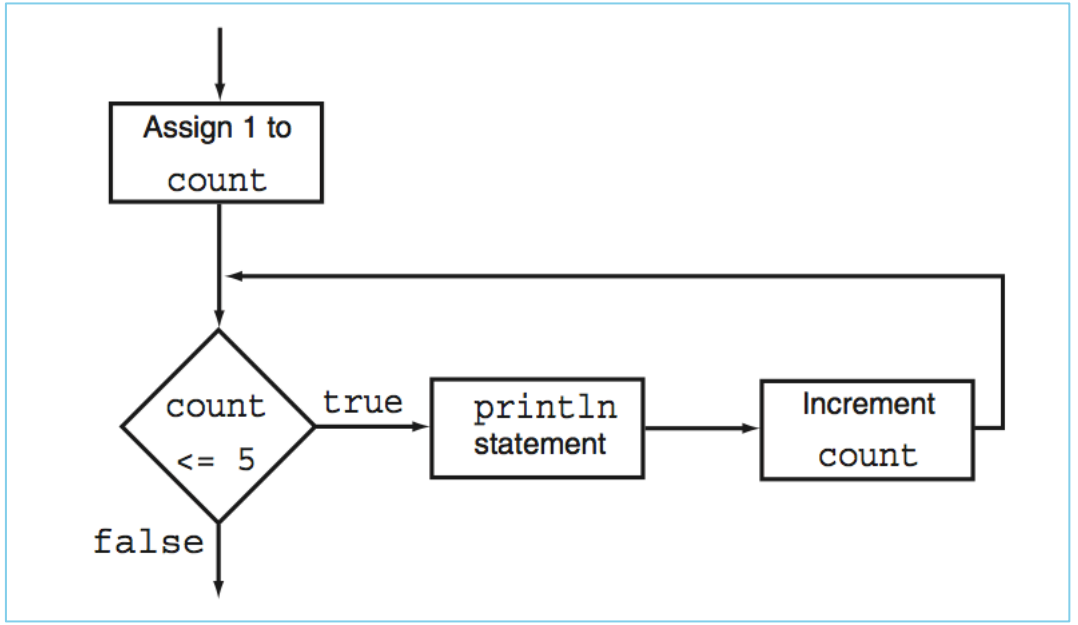
Step 1: Perform the initialization expression.

Step 2: Evaluate the test expression. If it is `true`, go to step 3. Otherwise, terminate the loop.

```
for (count = 1; count <= 5; count++)
   System.out.println("Hello");
```

Step 3: Execute the body of the loop.

Step 4: Perform the update expression, then go back to step 2.



Examples


```
int number;
for (number = 1; number <= 10; number++) {
    System.out.print(number + " ");
}
```

Output: 1 2 3 4 5 6 7 8 9 10

```
for (int number = 1; number <= 10; number++) {
    System.out.print(number + " ");
}
```

Output: 1 2 3 4 5 6 7 8 9 10

```
for (int number = 2; number <= 100; number += 2) {
    System.out.println(number);
}
```

Output: Display **even numbers** from 2 through 100

```
for (int number = 2; number <= 100; number += 2) {
    System.out.println(number);
}
System.out.println(number);
```

Output: Compilation Fails

Examples

- If you deposit $10,000 with an interest of 5%, how much money you will have after 15 years?

```
double balance = 10000;
int numberOfYears = 15;
for (int i = 0; i < numberOfYears; i++)
{
 double interest = balance * rate / 100;
 balance = balance + interest;
}
System.out.println(balance)
/* Output:
20789.28
*/
```

Tracing:

1. 1st iteration: `i` : 0 < `numberOfYears` : 15 ☑
   `balance` : 10500
   `i` : 1
2. ...
3. 16th iteration: `i` : 15 < `numberOfYears` : 15 🚫
   stops the loop

Multiple statements in Initialization and Update Expressions

```
int x, y;
for (x = 1, y = 1; x <= 5; x++) {
    System.out.println(x + " plus " + y
                       + " equals " + (x + y));
}
```

```
1 plus 1 equals 2
2 plus 1 equals 3
3 plus 1 equals 4
4 plus 1 equals 5
5 plus 1 equals 6
```

```
1 plus 1 equals 2
2 plus 2 equals 4
3 plus 3 equals 6
4 plus 4 equals 8
5 plus 5 equals 10
```

```
int x, y;
for (x = 1, y = 1; x <= 5; x++, y++) {
    System.out.println(x + " plus " + y
                       + " equals " + (x + y));
}
```

for-each Loop

- "**for-each**" loop is used exclusively to loop through elements in an **array**

```
for (type variableName : arrayName){
   // code block to be executed
}
```

Have to be the same type

```
String[] menu = {"Coffee", "Coco", "Tea" }
for (String m : menu){
   System.out.println(m);
}
```

Variable `m` can be used in side the code block

Output:
Coffee
Coco
Tea

for-each Loop [Object]

```java
public class Dog {
  private int age;
  private String color;

  public Dog(int age, String color){
    this.age = age;
    this.color = color;
  }

  public int getAge(){
    return age;
  }

  public String toString(){
    return "DOG["+age+","+color+"]";
  }
}
```

```java
Dog[] dogs = new Dog[3];
dogs[0] = new Dog(2, "white");
dogs[1] = new Dog(5, "brown");
dogs[2] = new Dog(1, "black");

for(Dog d: dogs){
  System.out.println(d.toString());
  if(d.getAge() < 4){
    System.out.println("Younger than 4 years old");
  }
}
```

OUTPUT
DOG[2,white]
Younger than 4 years old
DOG[5,brown] DOG[1,black]
Younger than 4 years old

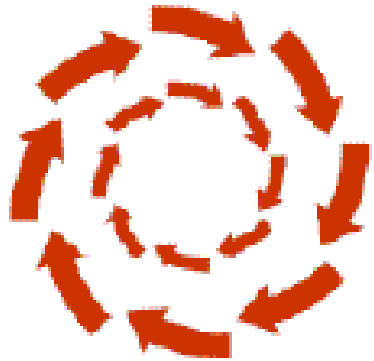
This for-each gets the same result of the for loop with index here

```java
for(int i = 0; i < 3; i++){
  System.out.println(dogs[i].toString());
  if(dogs[i].getAge() < 4){
    System.out.println("Younger than 4 years old");
  }
}
```

Nested Loop

- Nested Loop: a placing of one loop inside the body of another loop is called nesting.

```
for(initialization,; BooleanExpression; update){
  for(initialization2,; BooleanExpression2; update2){
      statement;
      statement;
      // Place as many statements here
      // as necessary.
  }
}
```

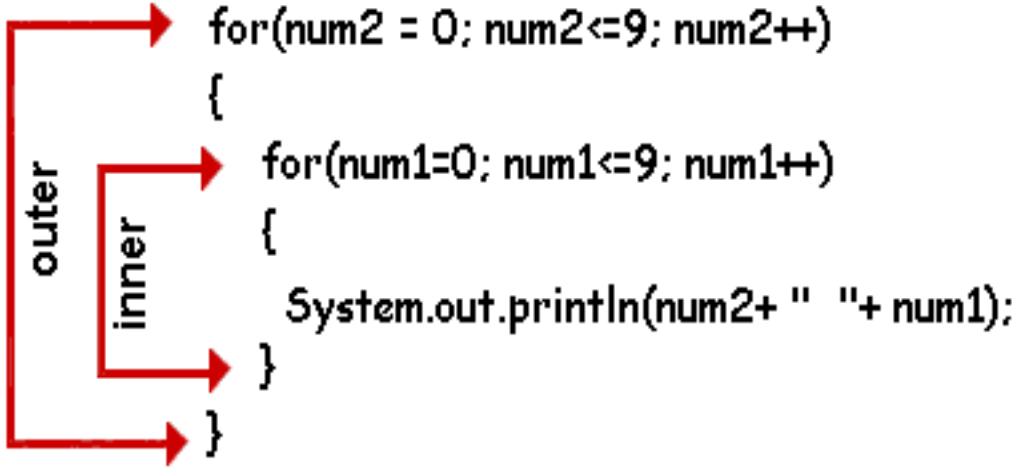


Nested Loop (Cont)

- **Example1:** webpage counter

0	0

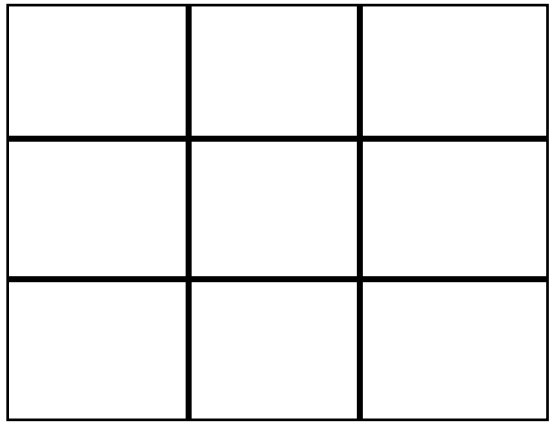
Considering 2 digits webpage counter...




```
for(num2 = 0; num2<=9; num2++)
{
  for(num1=0; num1<=9; num1++)
  {
    System.out.println(num2+ "  "+ num1);
  }
}
```

Nested Loop (Cont)

- **Example2:** Table position



```
for(int row=0; row<3; row++){
   for(int col=0; col<3; col++){
     System.out.println("table position=" + row + "," + col);
   }
}
```

do… while Loop

- do… while Loop: this is similar to while loop apart from that this loop is guaranteed to execute at least one time.

```
do{
    statement;
    // Place as many statements here
    // as necessary.
} while(booleanExpression);
```

Do…while Loop

Practice1…

```
int i = 0;
int sum = 9;
do{
   i++;
   sum = sum + i;
   System.out.println(i + ", " + sum);
} while (sum < 10);
```

1, 10

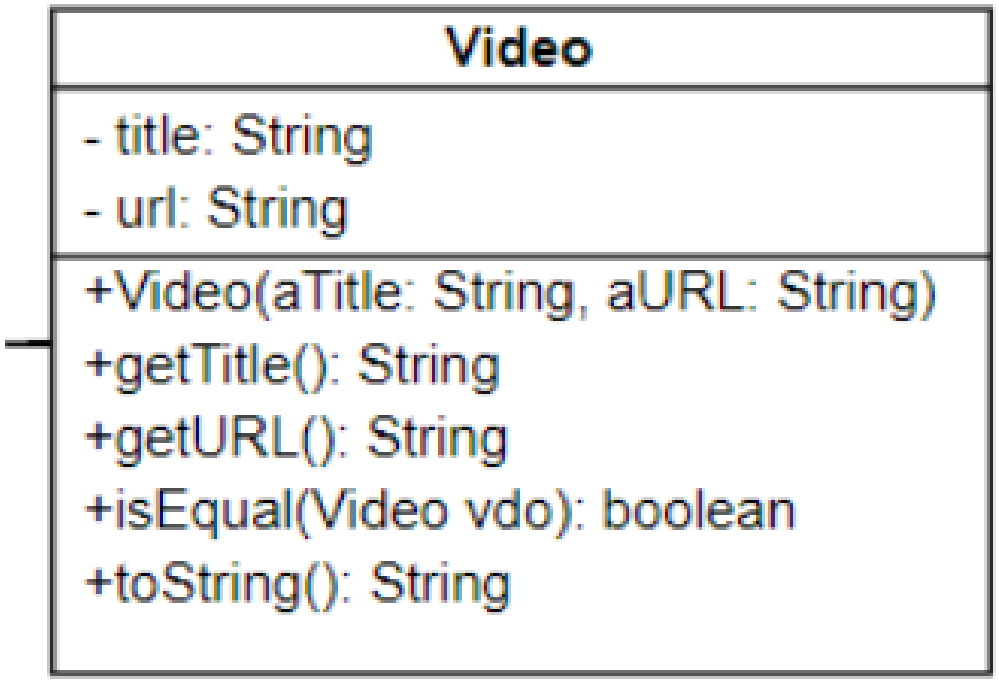
Do…while Loop

- **Practice2…**
- To accept input from users, and keep asking until they input a negative number

```
Scanner scan = new Scanner(System.in);
int val;
do{
    System.out.print("Please enter a negative number: ");
    val = scan.nextInt();
} while(val >= 0);

System.out.println("Input value is " + val);
```

Class and Object

Class and Object



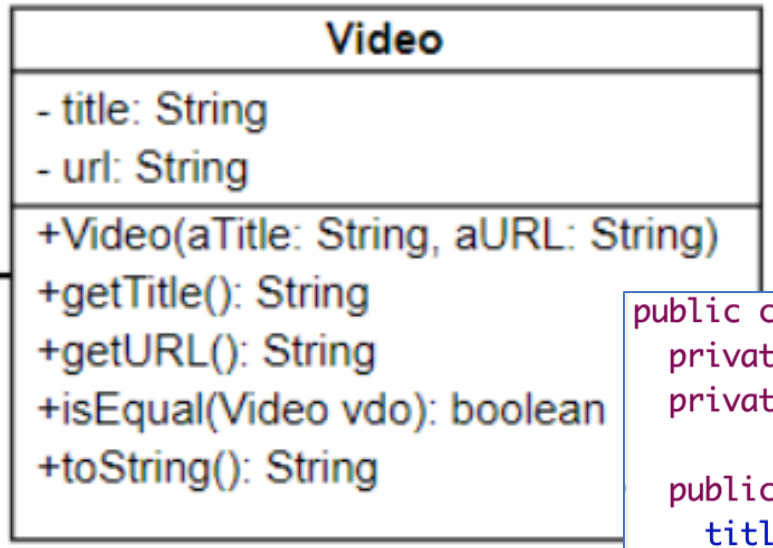
Instance fields (Class attributes)

Constructor method (one or more)

Setter/Getter method

Other method with and without return

From Diagram to Code

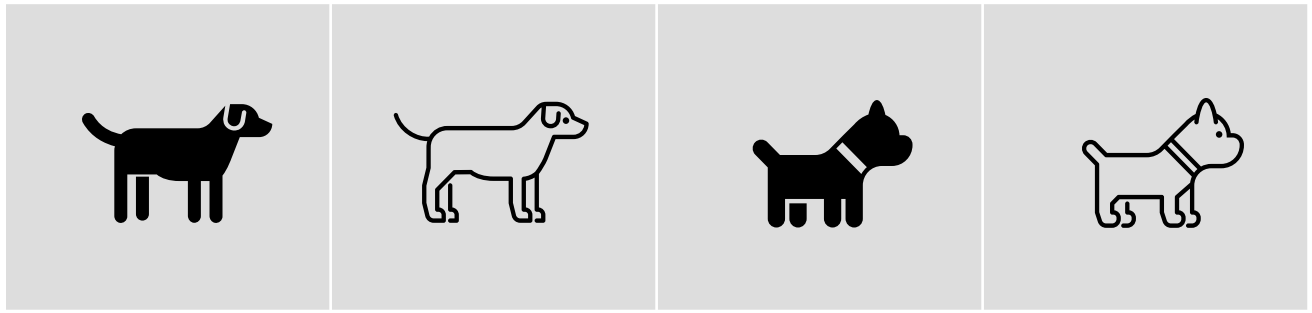



```java
public class Video{
  private String title;
  private String url;

  public Video(String aTitle, String aURL){
    title = aTitle;
    url = aURL;
  }

  /*
   * Return the title of this video
   */
  public String getTitle(){
    return this.title;
  }

  /*
   * Return the url of this video
   */
  public String getURL(){
    return this.url;
  }

  /*
   * Check whether the given video is equal to this video or not.
   * Two videos will be equal if their title are the same
   * and their url are the same as well.
   */
  public boolean isEqual(Video vdo){
    return (this.title.equals(vdo.getTitle()) && this.url == vdo.getURL());
  }


  /*
   * Return a string with video's name and url.
   */
  public String toString(){
    return "Title: " + this.title + " URL: "+this.url;
  }
}
```


Array
and
ArrayList

Array

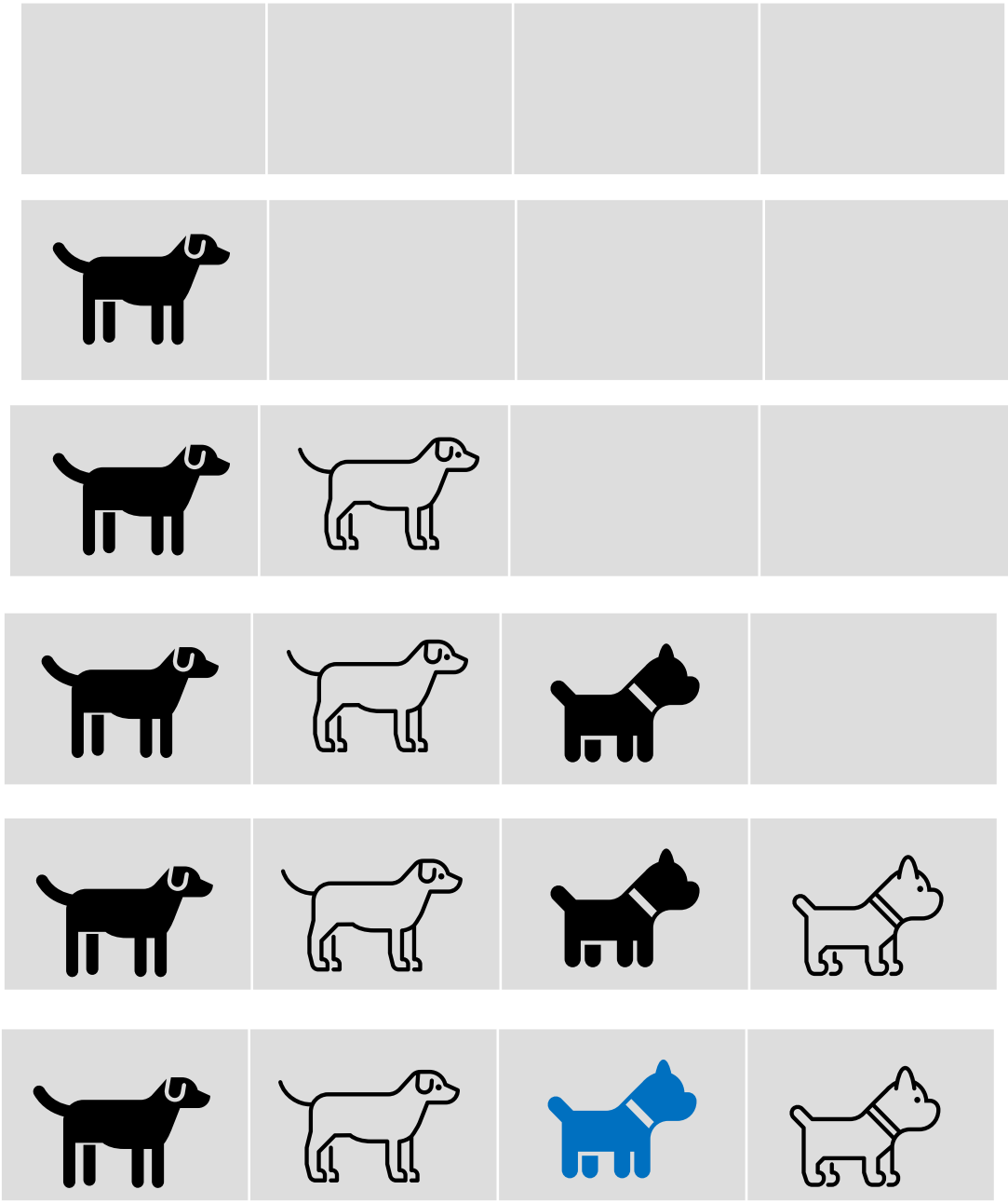
10	20	30	40



50

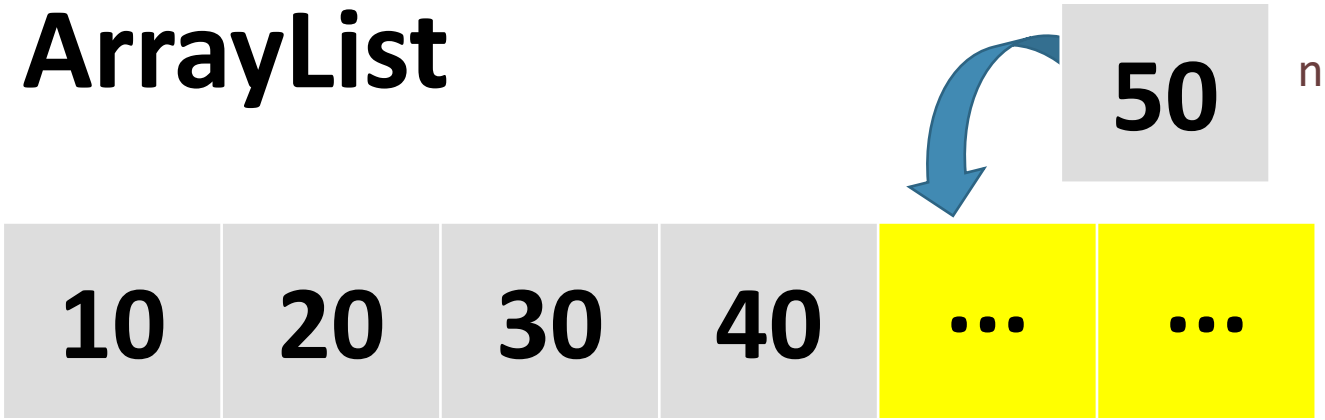
```
int[] nums = new int[4];
-- or --
int[] nums = {10, 20, 30, 40};

int x = nums[2]; // x = 30

nums[0] = 5;
```

```
Dog[] dogs = new Dog[4];

dogs[0] = new Dog(12, "black");

System.out.println(dogs[0]);
```



```
Dog[] dogs = new Dog[4];

dogs[0] = new Dog(12, "black");

dogs[1] = new Dog(12, "white");

dogs[2] = new Dog(3, "black");

dogs[3] = new Dog(3, "white");

// change color of a dog
dogs[2].color = "blue";

dogs[2].setColor("blue");
```

ArrayList



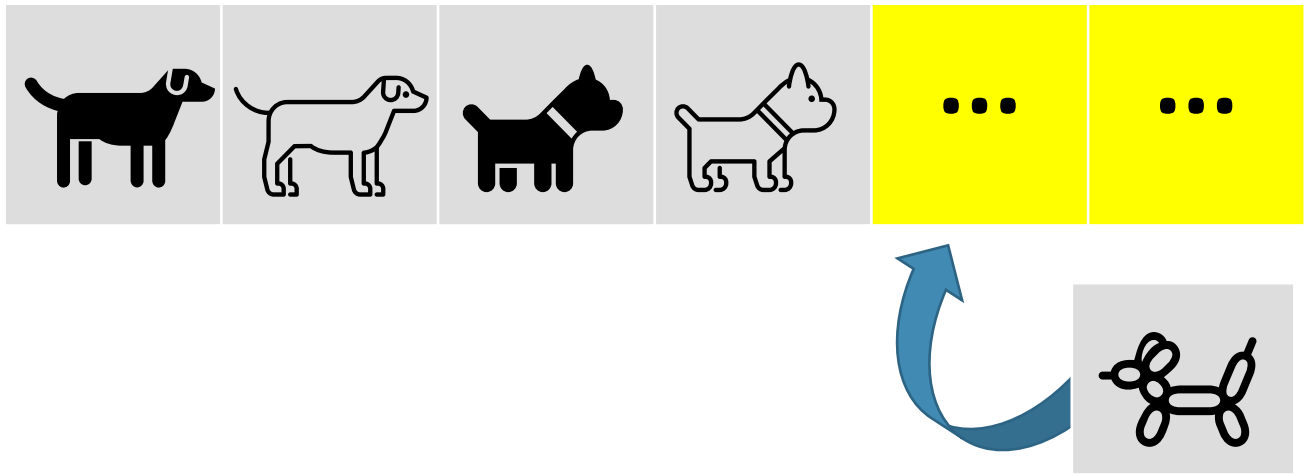

```
nums.add(50);
```

```
nums.add(10);
nums.add(20);
nums.add(30);
nums.add(40);
```

```
ArrayList<Integer> nums = new ArrayList<Integer>();
```

```
int x = nums.get(2); // x = 30
```

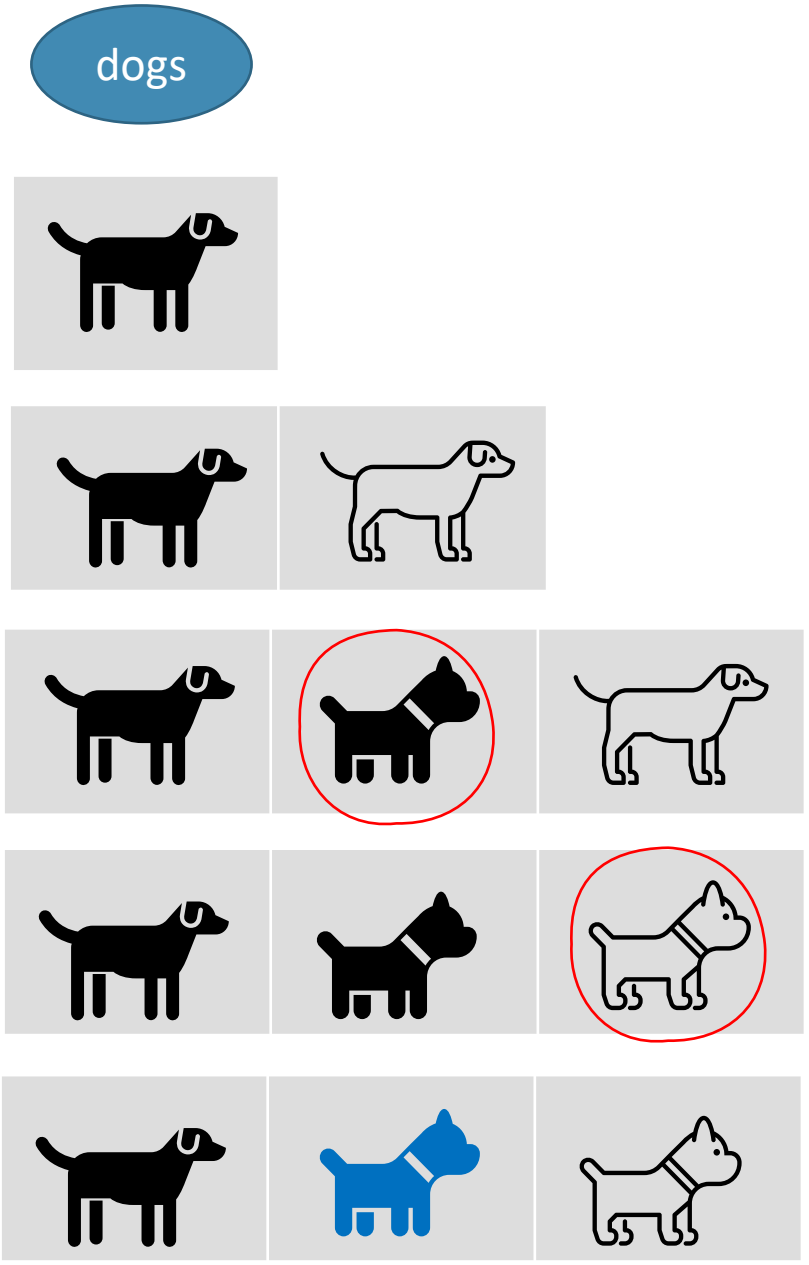
```
ArrayList<Dog> dogs = new ArrayList<Dog>();
```




```
dogs.add(new Dog(12, "black"));
dogs.add(new Dog(12, "white"));
dogs.add(new Dog(3, "black"));
dogs.add(new Dog(3, "white"));
```

```
Dog d = dogs.get(2); // [3, black]
```

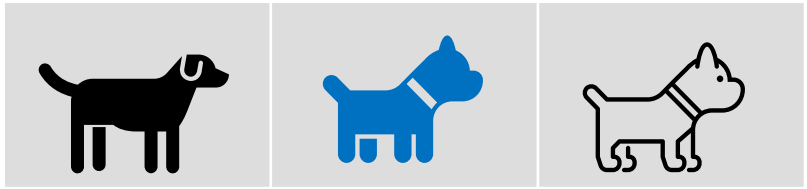
```
nums.add(new Dog(3, "white"));
```




```
ArrayList<Dog> dogs = new ArrayList<Dog>();

dogs.add(new Dog(12, "black"));

dogs.add(new Dog(12, "white"));

dogs.add(1, new Dog(3, "black"));

dogs.set(2, new Dog(3, "white"));

// change color of dog at index 1
dogs.get(1).color = "blue";
// -- or --
dogs.get(1).setColor("blue");
```

dogs

```
ArrayList<Dog> dogs = new ArrayList<Dog>();
```



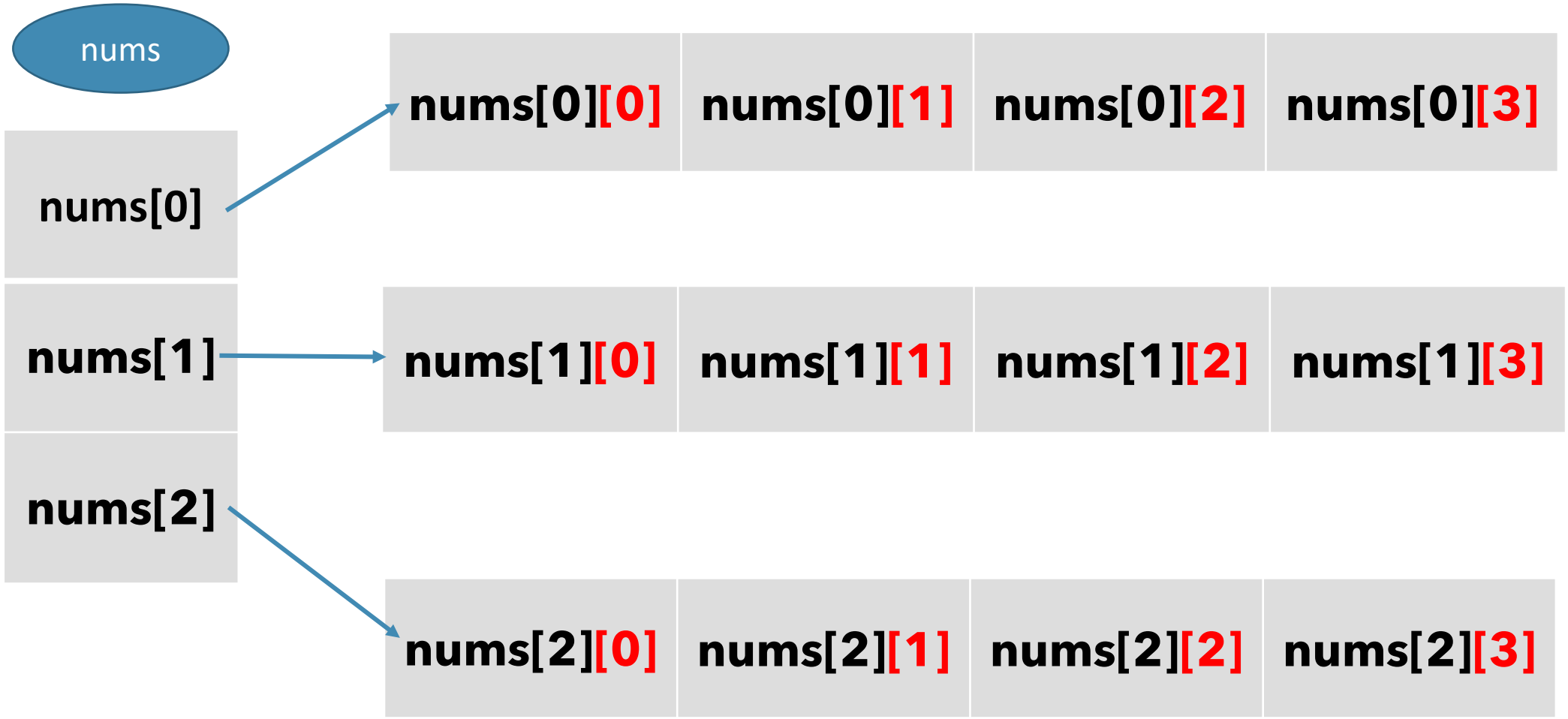
```
// count dog who are younger than 4 months
int count = 0;

for(Dog d : dogs){
    if(d.getAge() < 4)
        count++;
}

System.out.println(count);  // 2
```

```
// print dog with index
for(int i = 0; i < dogs.size(); i++){
    System.out.println("index:" + i + ", " + dogs.get(i));
}
```

```
index:0, DOG[12,black]
index:1, DOG[3,blue]
index:2, DOG[3,white]
```

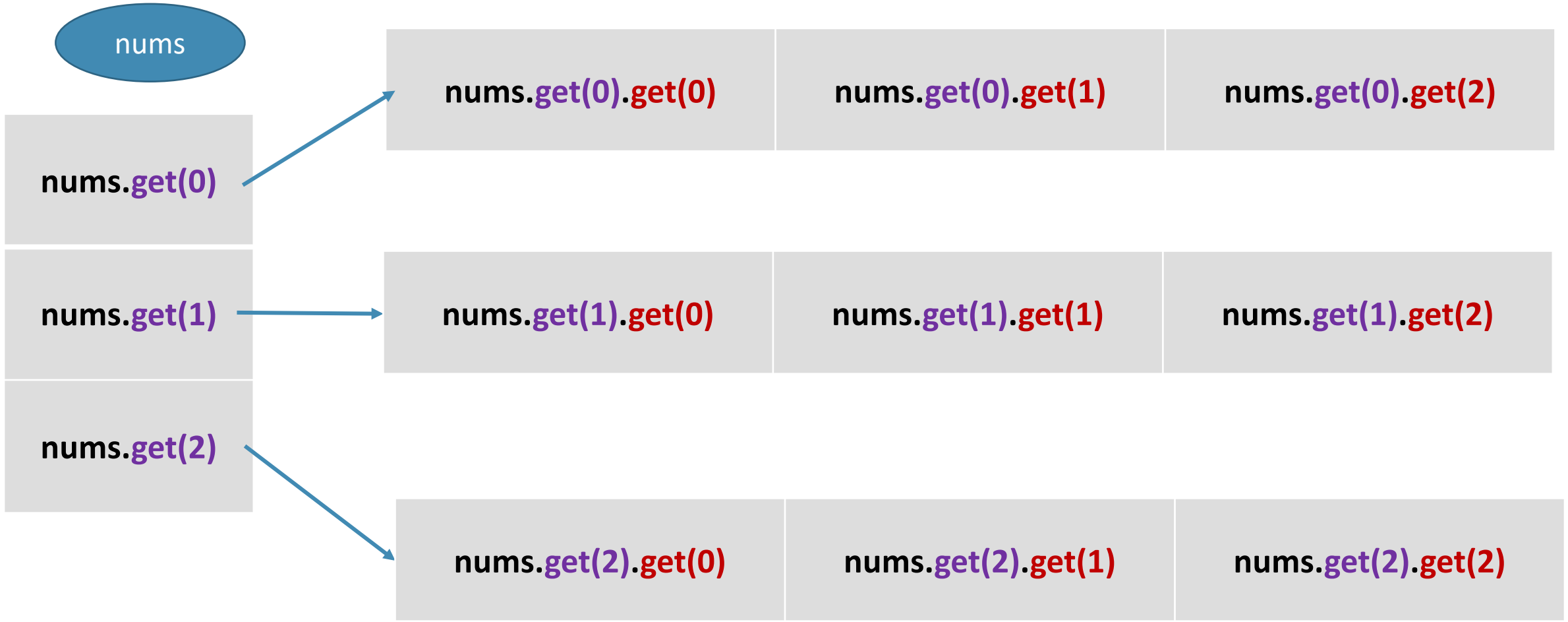
2D Array

```
int[][] nums = new int[3][4];
```



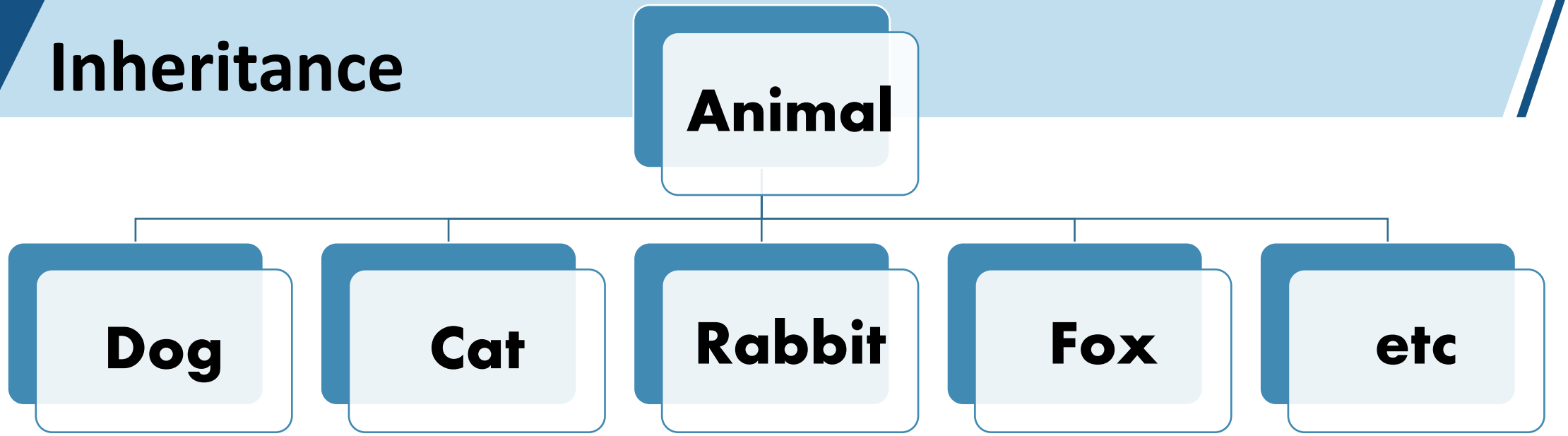
2D ArrayList

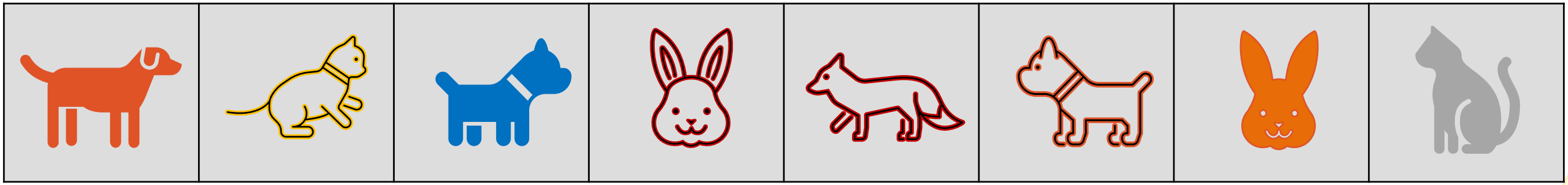
```
ArrayList<ArrayList<Integer>> nums = new ArrayList<ArrayList<Integer>>();
```



Inheritance

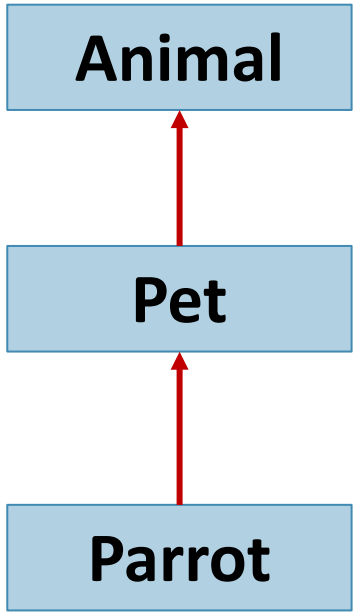
Inheritance

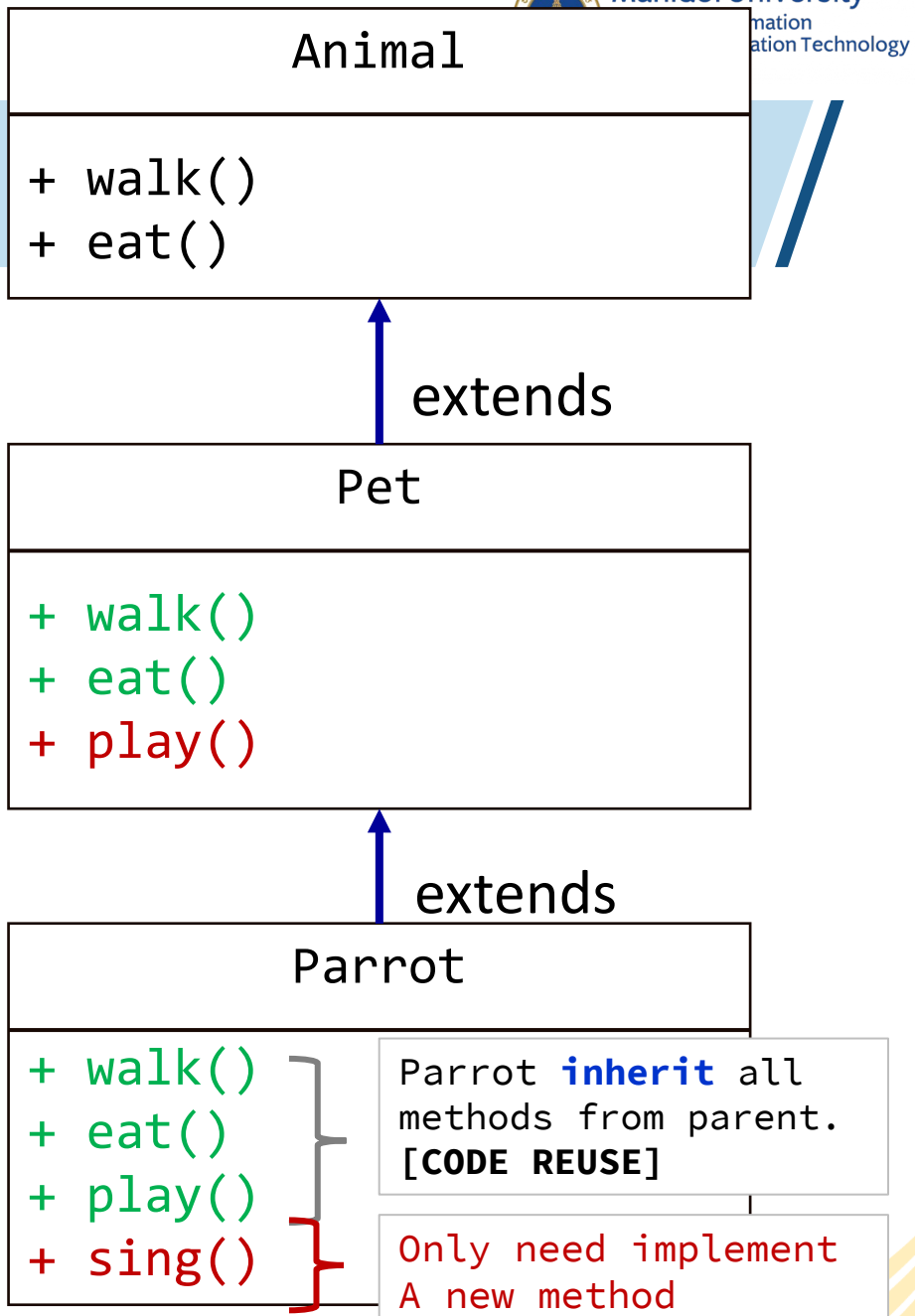




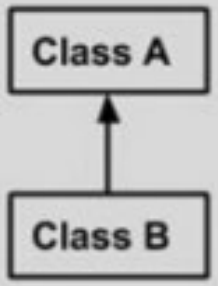
Inheritance (Example)

Inheritance can be done in multi-level.
As example in this case, we can create a new class named `Parrot` extends from the class `Pet`.



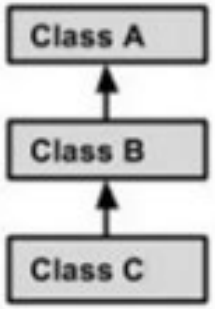


Single Inheritance



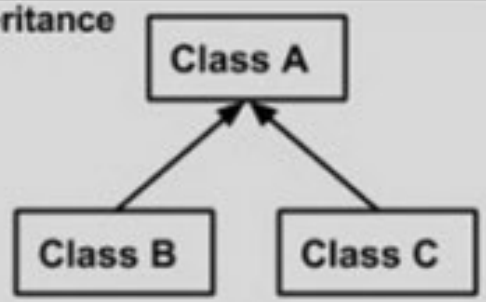

```
public class A {
   .......
}
public class B extends A {
   .........
}
```

Multi Level Inheritance



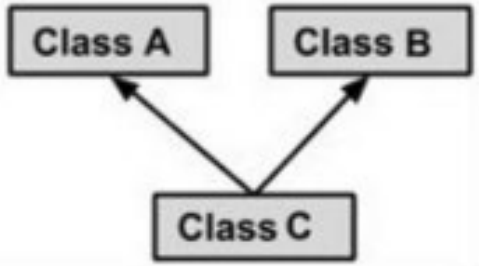

```
public class A { ..................}
public class B extends A {.....................}
public class C extends  B {..................... }
```

Hierarchical Inheritance




```
public class A { ..................}
public class B extends A {.....................}
public class C extends A {..................... }
```

Multiple Inheritance




```
public class A { ..................}
public class B {.....................}
public class C extends  A,B {
   ....................
} // Java does not support mutiple Inheritance
```

c1 =

object

```
name: Silver
color: gray
hungry: 10
---------------
greeting()
```

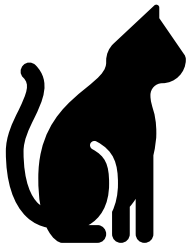
c2 =

```
name: Elsa
color: white
hungry: 5
---------------
greeting()
```

Instance fields/Attributes

Method

object



Cat

class

```
name:
color:
hungry:
----------------
greeting()
```

```
System.out.println("Meow! " + this.name);
```

1) Simplest Cat Class

```java
public class Cat {
    /*
     * Instance fields or Instance variables or Attributes
     */
    String name;
    String color;
    int hungry;

    /*
     * When there is no constructor method provided,
     * the class has a default constructor
     * with empty parameter and empty body.
     * For example, this Cat class has this constructor
     *
     * public Cat(){
     * }
     */

    /*
     * Instance method with no return (void)
     * To display the greeting message
     */
    void greeting(){
        System.out.println(
         "Meow! " + this.name);
    }
}
```

```java
public class CatTester {
    public static void main(String[] args){

        Cat c1 = new Cat(); // Create Cat object via Cat()
                            // default constructor method

        c1.name = "Silver"; // Assign value to its attribute
        c1.color = "gray";  // Assign value to its attribute
        c1.hungry = 10;     // Assign value to its attribute

        Cat c2 = new Cat(); // Create another Cat object
        c2.name = "Elsa";
        c2.color = "White";
        c2.hungry = 5;

        c1.greeting();      // call greeting() method by c1
        c2.greeting();      // call greeting() method by c2
    }
}
```

OUTPUT:

Meow! Silver
Meow! Elsa

2) Cat Class with Constructor Method

```java
public class Cat {
    String name;
    String color;
    int hungry;

    /*
     * Define a constructor method with 3 parameters
     */
    Cat(String name, String color, int hungry){
        this.name = name;       // "this." is required because
                                // the names of the parameter and
                                // the attribute are the same.

        this.color = color;     // "this." is required
        this.hungry = hungry;   // "this." is required
    }

    void greeting(){
        System.out.println(
         "Meow! " + this.name); // "this." is no required
                                // since no duplicate variable name
    }

}
```

```java
public class CatTester {
    public static void main(String[] args){

        // Create Cat object via defined constructor method
        Cat c1 = new Cat("Silver", "gray", 10);
        Cat c2 = new Cat("Elsa", "white", 5);

        /*
         * NOTE: If you already define your own constructor, then
         * the default constructor method is no longer valid.
         * Cat c = new Cat(); // error!!!
         */

        c1.greeting();
        c2.greeting();
    }
}
```

OUTPUT:

Meow! Silver
Meow! Elsa

3) Public vs Private

```java
public class Cat {
    String name;            // public by default
    public String color;    // public access modifier
    private int hungry;     // private access modifier

    /*
     * Constructor method
     */
    Cat(String name, String color, int hungry){
        this.name = name;
        this.color = color;
        this.hungry = hungry;
    }

    /*
     * Instance method
     */
    void greeting(){
        System.out.println(
         "Meow! " + this.name);
    }

    /*
     * Getter method with return type (int)
     */
    int getHungry(){
        return hungry;
    }
}
```

```java
public class CatTester {
    public static void main(String[] args){

        Cat c1 = new Cat("Silver", "gray", 10);

        // access "public" instance field
        // using objectname.instancefiledname
        // correct... yea!
        System.out.println(c1.name);
        System.out.println(c1.color);

        // access "private" instance field
        // error... boo!
        //System.out.println(c1.hungry);

        // have to access through the getter method
        // correct... yea!
        System.out.println(c1.getHungry());
    }
}
```

4) static variable/method & final

```java
public class Cat {
    public String name, color;
    private int hungry;

    // "static" keyword to indicate a variable of the class
    static int objCounter = 0;

    // "final" keyword to indicate an unchangeable variable
    final String TYPE = "cat";

    // static and final are commonly used together
    static final int LIVE = 9;

    Cat(String name, String color, int hungry){
        this.name = name;
        this.color = color;
        this.hungry = hungry;
        this.objCounter++;      // increase the counter when
                                // an object is created
    }

    // "static" keyword to indicate a method of class
    public static void begin(){
        System.out.println("Let's start");
    }

    void greeting(){
        System.out.println("Meow! " + this.name);
    }
}
```

```java
public class CatTester {
    public static void main(String[] args){

        // Access static variable using Cat class
        System.out.println(Cat.objCounter);     // OUTPUT: 0

        Cat c1 = new Cat("Silver", "gray", 10);
                // this constructor will increase objCounter
        System.out.println(Cat.objCounter);     // OUTPUT: 1

        Cat c2 = new Cat("Elsa", "white", 5);
        System.out.println(Cat.objCounter);     // OUTPUT: 2

        // Call static method using Cat class
        Cat.begin();        // OUTPUT: Let's start

        // Call instance method using specific object
        c1.greeting();      // OUTPUT: Meow! Silver
        c2.greeting();      // OUTPUT: Meow! Elsa

        // Access final variable
        System.out.println(c1.TYPE);    // OUTPUT: cat

        // Access static final variable
        System.out.println(Cat.LIVE);   // OUTPUT: 9

    }
}
```

5) Method with Parameters

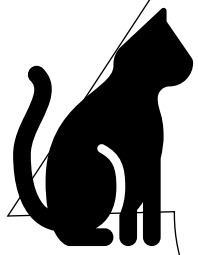
```java
public class Cat {
    public String name;
    public String color;
    private int hungry;

    Cat(String name, String color, int hungry){
        this.name = name;
        this.color = color;
        this.hungry = hungry;
    }

    /*
     * instance method with one parameter (primitive type)
     * This method changes the hungry level (instance field)
     */
    void feed(int level){
        hungry = hungry - level;
    }

    void greeting(){
        System.out.println("Meow! " + this.name);
    }

    int getHungry(){
        return this.hungry;
    }
}
```

```java
public class CatTester {
    public static void main(String[] args){

        Cat c1 = new Cat("Silver", "gray", 10);
        Cat c2 = new Cat("Elsa", "white", 5);

        // to display hungry level
        System.out.println(c1.getHungry());     // OUTPUT: 10

        // feed c1 object which only effects its hungry level
        c1.feed(2);

        // to display hungry level after feed(x)
        System.out.println(c1.getHungry());     // OUTPUT: 8

        // hungry level of c2 is not effected
        System.out.println(c2.getHungry());     // OUTPUT: 5
    }
}
```

common attributes: same variable name and type OR **common methods:** same method name and method body

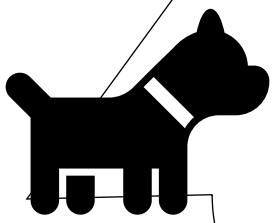
similar methods: same method name but different method body

unique attributes/method: only available in a specific class



Cat Class

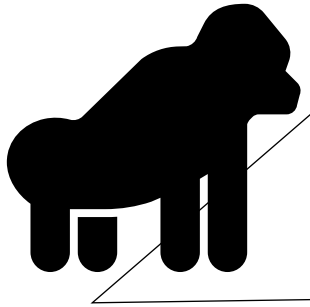
```
name:
color:
hungry:
energy:
---------------
Cat(nanme,color,hungry,energy)
getHungry()
getEnergy()
greeting() -> Meow! I'm + [name]
isAlive()
play()
sleep()
feedFood(food)
displayCat()
```



Dog Class

```
name:
color:
hungry:
energy:
sense: // sense of smell
---------------
Dog(nanme,color,hungry,energy,sense)
getHungry()
getEnergy()
getSense()
greeting() -> Bark! I'm + [name]
isAlive()
play()
sleep()
feedFood(food)
displayDog()
```

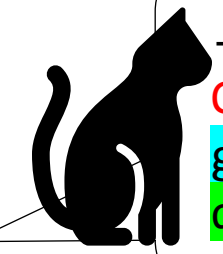
Superclass
(general class)



Animal Class

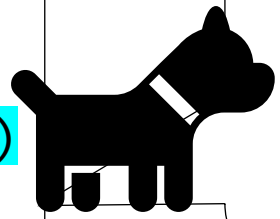
```
name:
color:
hungry:
energy:
---------------
Animal(nanme,color,hungry,energy)
getHungry()
getEnergy()
greeting() -> I'm + [name] // same on both animal
isAlive()
play()
sleep()
feedFood(food)
```

Subclass only have things that differ from superclass!!!



Cat Class

```
---------------
Cat(nanme,color,hungry,energy)
greeting() -> Meow! + super.greeting()
displayCat()
```



Dog Class

```
sense: // sense of smell
---------------
Dog(nanme,color,hungry,energy,sense)
getSense()
greeting() -> Bark! + super.greeting()
displayDog()
```

Do you see any different between these two Cat classes?

```java
public class Cat {
    public String name, color;
    private int hungry, energy;

    Cat(String name, String color, int hungry, int energy){
        this.name = name;
        this.color = color;
        this.hungry = hungry;
        this.energy = energy;
    }

    /*
     * greeting method()
     */
    void greeting(){
        System.out.println("Meow! I'm " + this.name);
    }

    /*
     * show Cat's info
     */
    void displayCat(){
        System.out.println("----------------------");
        System.out.println("Name: " + this.name);
        System.out.println("Color: " + this.color);
        System.out.printf("Hungry (%d), Energy (%d)\n", hungry, energy);
    }

}
```

Extends, Attribute, Constructor, body of greeting method, how to access common attribute which are private?

```java
public class Cat extends Animal {
    Cat(String name, String color, int hungry, int energy){
        super(name, color, hungry, energy);
    }

    /*
     * greeting method()
     */
    void greeting(){
        System.out.println("Meow!");
        super.greeting();   // calling greeting method from Animal
    }

    /*
     * show Cat's info
     */
    void displayCat(){
        System.out.println("----------------------");
        System.out.println("Name: " + this.name);
        System.out.println("Color: " + this.color);
        System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
        // hungry and energy are private attributes in Animal,
        // so you have to use getter method to access them
    }
}
```

```java
/*
 * greeting method() in Animal class
 */
void greeting(){
    System.out.println("I'm " + this.name);
}
```

```java
public class Dog extends Animal {
    private int sense;         // Dog has a good sence of smell

    Dog(String name, String color, int hungry, int energy, int sense){
        super(name, color, hungry, energy);
        this.sense = sense;
    }

    int getSense(){
        return this.sense;
    }

    /*
     * greeting method() -> override greeting in Animal method
     */
    void greeting(){
        System.out.println("Bark!");
        super.greeting();
    }


    /*
     * show Dog's info
     */
    void displayDog(){
        System.out.println("------------------------");
        System.out.println("Name: " + this.name);
        System.out.println("Color: " + this.color);
        System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
        System.out.println("Sense of smell: " + sense);

    }
}
```

```java
/*
 * show Cat's info
 */
void displayCat(){
    System.out.println("-----------------------");
    System.out.println("Name: " + this.name);
    System.out.println("Color: " + this.color);
    System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
    // hungry and energy are private attributes in Animal,
    // so you have to use getter method to access them
}
```

```java
/*
 * show Dog's info
 */
void displayDog(){
    System.out.println("-----------------------");
    System.out.println("Name: " + this.name);
    System.out.println("Color: " + this.color);
    System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
    System.out.println("Sense of smell: " + sense);

}
```

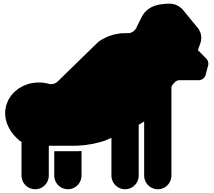
These two methods have different name yet have very similar behavior (method's body).

So we can simplify them and put the common code in superclass (Animal)

```java
/*
 * show general Animal's info (in Animal class)
 */
void display(){
    System.out.println("-----------------------");
    System.out.println("Name: " + this.name);
    System.out.println("Color: " + this.color);
    System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
}
```

```java
/*
 * show Dog's info
 */
void display(){
    super.display();
    System.out.println("Sense of smell: " + sense);

}
```

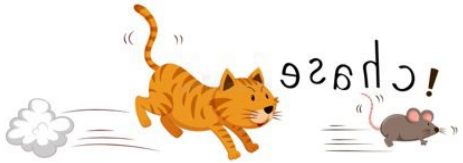
Superclass (general class)



```java
public class Animal {
    private String name;

    public Animal(String n){
        this.name = n;
    }

    public String getName(){
        return this.name;
    }

    public void greeting(){
        System.out.println("I'm " + name);
    }
}
```

Subclass (specialized class)

```java
public class Cat extends Animal{
    public Cat(String name){
        super(name);
    }

    @Override
    public void greeting(){
        System.out.print("Meow! ");
        super.greeting();
    }

    public void chasing(){
        System.out.println("Chasing mouse...");
    }
}
```




```java
public class Dog extends Animal{

    public Dog(String name) {
        super(name);
    }

    @Override
    public void greeting(){
        System.out.print("Bark! ");
        super.greeting();
    }

    public void catching(){
        System.out.println("Catching frisbee...");
    }
}
```

What will be the output of this?

```java
Animal a = new Animal("Olaf");
a.greeting();

Cat c = new Cat("Elsa");
c.greeting();

Dog d = new Dog("Pika");
d.greeting();
```

```java
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
a2.greeting();

Animal a3 = new Dog("Pika");
a3.greeting();
```

```java
Animal[] pets = new Animal[3];
pets[0] = new Animal("Olaf");
pets[1] = new Cat("Elsa");
pets[2] = new Dog("Pika");

for(Animal p: pets){
    p.greeting();
}
```

Polymorphism

Polymorphism

What is Polymorphism

- **Polymorphism** is an OOP concept where the ability of variables and methods to take on **Many Forms**.

- **Polymorphism consists of the following concept:**
  - **Shadowing** (variable)
  - **Overriding** (methods)
  - **Overloading** (methods)

Polymorphism

Shadowing (variable)

- **Shadowing** variable is a situation when we define a variable in a closure scope with a variable name that is the same as one for a variable we've already defined in an outer scope.

```
class Animal {
  String name = "Animal";

  public static void main(String args[]) {
    Animal animal = new Animal();
    Dog dog = new Dog();
    System.out.println(animal.name + " " + dog.name);
  }
}
```

```
public class Dog extends Animal {
   String name = "Dog";
}
```

▶ This is called shadowing—name in class Dog shadows name in class Animal

Polymorphism

Overloading

- Two or more methods with different signatures

Overriding

(Note that we already leared this last week!!)

- Replacing an inherited method with another method having the same signature

Signature in Java

`foo(int i)` and `foo(int i, int j)` are different

`foo(int i)` and `foo(int k)` are the same

`foo(int i, double d)` and `foo(double d, int i)` are different

```
public static void main(String args[]) {
    myPrint(5);          // call myPrint( int )
    myPrint(5.0);        // call myPrint( double )

    Animal a = new Animal("Olaf");
    a.greeting("Nice to meet you");

}

static void myPrint(int i) {
    System.out.println("int i = " + i);
}

static void myPrint(double d) {
    // Overload: same name, different parameters
    System.out.println("double d = " + d);
}
```

Overloading

Overload: in the same class, two methods can have the same name, provided they differ in their parameter types. These are different methods, each with its own implementation. The Java compiler considers them to be completely unrelated.

Overriding: where a subclass method provides an implementation of a method whose parameter variables have the same types.

[Note! If you mean to override a method but use a parameter variable with a different type, then you accidentally introduce an overloaded method.]

```java
public class Animal {
    private String name;

    public Animal(String n){
        this.name = n;
    }

    public String getName(){
        return this.name;
    }

    public void greeting(){
        System.out.println("I'm " + name);
    }

    /*
     * Overload method: same name but different signature
     */
    public void greeting(String msg){
        System.out.println("I'm " + name);
        System.out.println(msg);  // to print the given msg
    }
}
```

Override greeting() method

```java
public class Cat extends Animal{
    public Cat(String name){
        super(name);
    }

    /*
     * Override method: same name same signature
     * from the superclass (Animal)
     */
    @Override
    public void greeting(){
        System.out.print("Meow! ");
        super.greeting();
    }

    public void chasing(){
        System.out.println("Chasing mouse...");
```

Example overloading usage

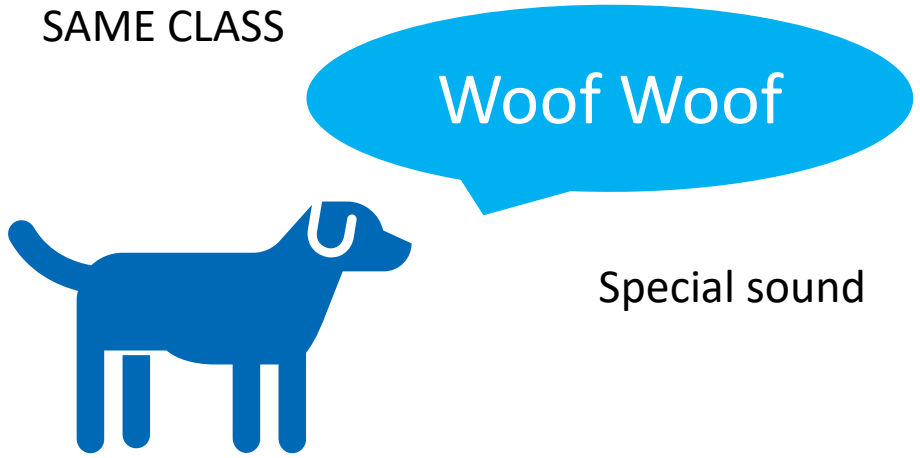
```java
/*
 * Use over load to minimize the code
 */
public void greeting(String msg){
    greeting();
    System.out.println(msg);  // to print additional msg
}
```

Override vs Overload

INHERITANCE

Default sound

SAME CLASS

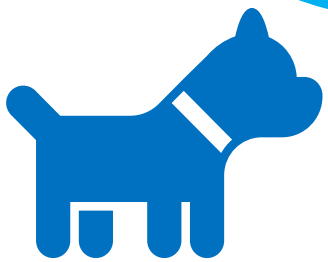


Special sound

Abstract Class & Abstract Method

```java
public abstract class Animal {
    private int age;
    public String color;

    public Animal(int age){
        this.age = age;
        this.color = "unknow";
    }

    public Animal(int age, String color){
        this.age = age;
        this.color = color;
    }

    public void setColor(String newColor){
        color = newColor;
    }

    public int getAge(){
        return age;
    }

    public String toString(){
        return age + "," + color;
    }

    public abstract void speak();
}
```

```java
public class Dog extends Animal {

    public Dog(int age){
        super(age);
    }

    public Dog(int age, String color){
        super(age, color);
    }

    @Override
    public String toString(){
        return "DOG["+ super.toString() +"]";
    }

    public void speak() {
        System.out.println("Bark Bark");
    }
}
```



Mock Exam Protocol

No score is collected

Mock Exam General Instruction - MyCourses

Type 11: Computer-based instructor upload content Exam

Allowed

1. A dictionary (ENG-to-ENG)
2. PDF files uploaded in Finex by instructors (This slide!!!)
3. Eclipse IDE ONLY

Not allowed

1. Other applications

Standard Rules (ICT Program)

1. No communication
2. No additional devices
3. No internet
4. No USB Drive

Practical Programming Exam Instruction

1. You MUST use a PC, mouse, and keyboard in the lab for taking the exam. Boot your machine using Windows. You should find every tool that you need to complete, and test your code such as Java JDK, and Eclipse.
2. You MUST write your name, student ID, and section in a comment section on top of every code file you submit.
3. You MUST download the starter code for each question. Once you finished, you MUST submit one or more .java files for each question separately. Any answer files that are submitted in the wrong question will be ignored.
4. There is NO extra time for submission, so please manage your time wisely. Questions with no answer will receive ZERO point.

Midterm Exam

35% of the total assessments

Midterm Exam General Instruction - FINEX

Type 11: Computer-based instructor upload content Exam

Allowed

1. A dictionary (ENG-to-ENG)
2. PDF files uploaded in Finex by instructors (This slide!!!)
3. Eclipse IDE ONLY

Not allowed

1. Other applications

Standard Rules (ICT Program)

1. No communication
2. No additional devices
3. No internet
4. No MyCourses
5. No USB Drive

Practical Programming Exam Instruction

1. You MUST use a PC, mouse, and keyboard **in the lab for taking the exam.** Boot your machine using Windows. You should find every tool that you need to complete, and test your code such as Java JDK, and Eclipse.
2. You MUST write your name, student ID, and section in a comment section on top of every code file you submit.
3. You MUST download the starter code for each question. Once you finished, you MUST submit **one or more .java files** for each question separately. Any answer files that are submitted in the wrong question will be ignored.
4. There is NO extra time for submission, so please manage your time wisely. Questions with no answer will receive ZERO point.

More Details on Midterm Exam

- Types of Questions
  - Quiz style e.g., multiple choices, short answers, matching answers, etc.
  - Coding questions: 3 Questions
- Each question may have one or more starter code files. And you may have to submit one or more answer files as well.
- The total score is 120 points. However, we only **collect 100 points** for this exam.
  - This means if you get 80 points, your score will be 80. But if you get 117 points, your score will be capped at 100.
- Try to do the questions that you are most confident first.