DEFINING CLASSES AND METHODS

OOP

Lecture 5

OBJECTIVES

- ❑ Describe concepts of class, class object
- ❑ Create class objects
- ❑ Define a Java class, its methods
- ❑ Describe use of parameters in a method
- ❑ Use modifiers public, private
- ❑ Define accessor, mutatorclass methods
- ❑ Describe information hiding, encapsulation
- ❑ Write method pre-and postconditions

CLASS AND METHOD DEFINITIONS

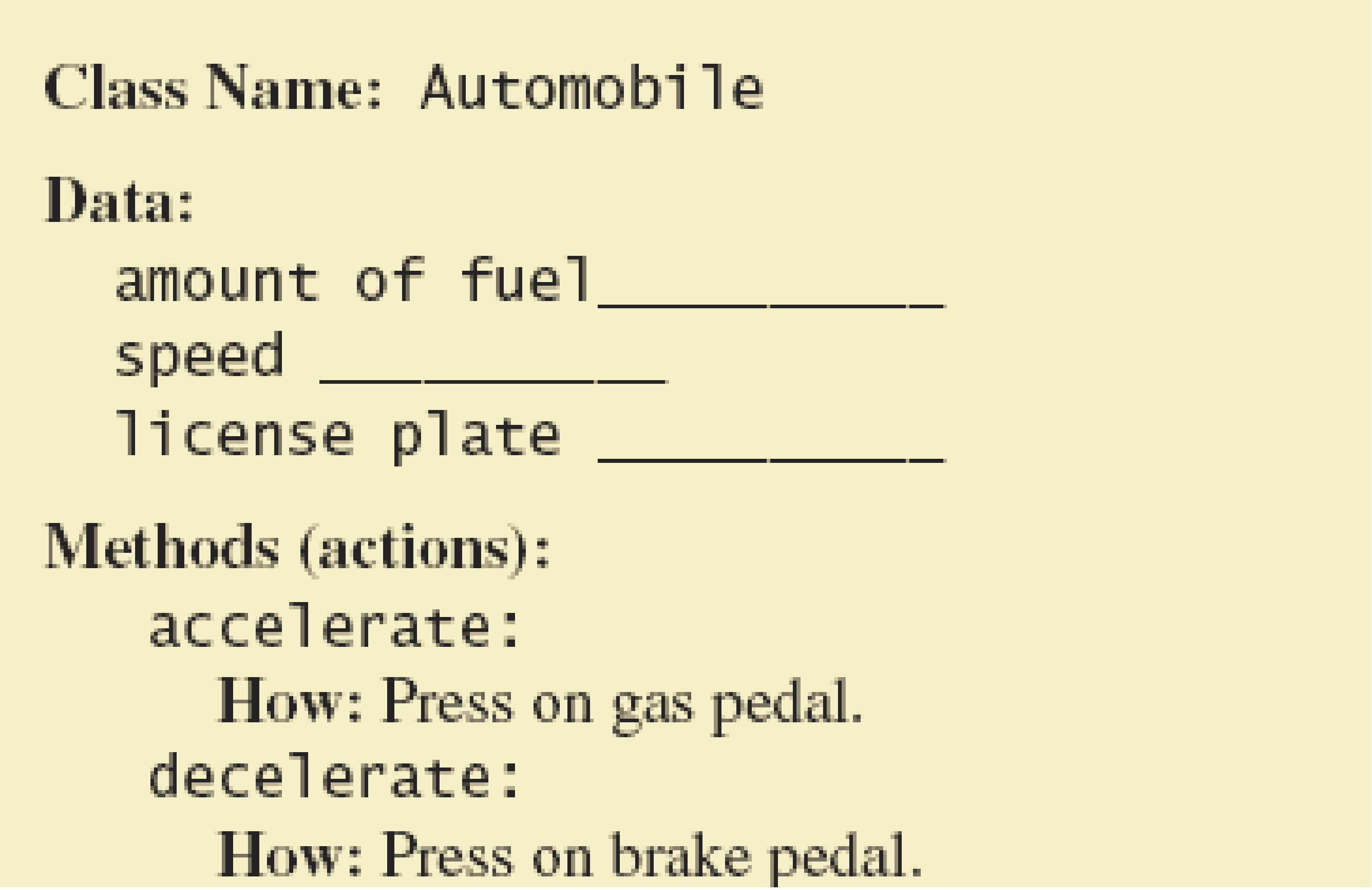
Java program consists of objects

- Objects of class types
- Objects that interact with one another

Program objects can represent

- Objects in real world
- Abstractions

CLASS AND METHOD DEFINITIONS



Class Name: Automobile

Data:

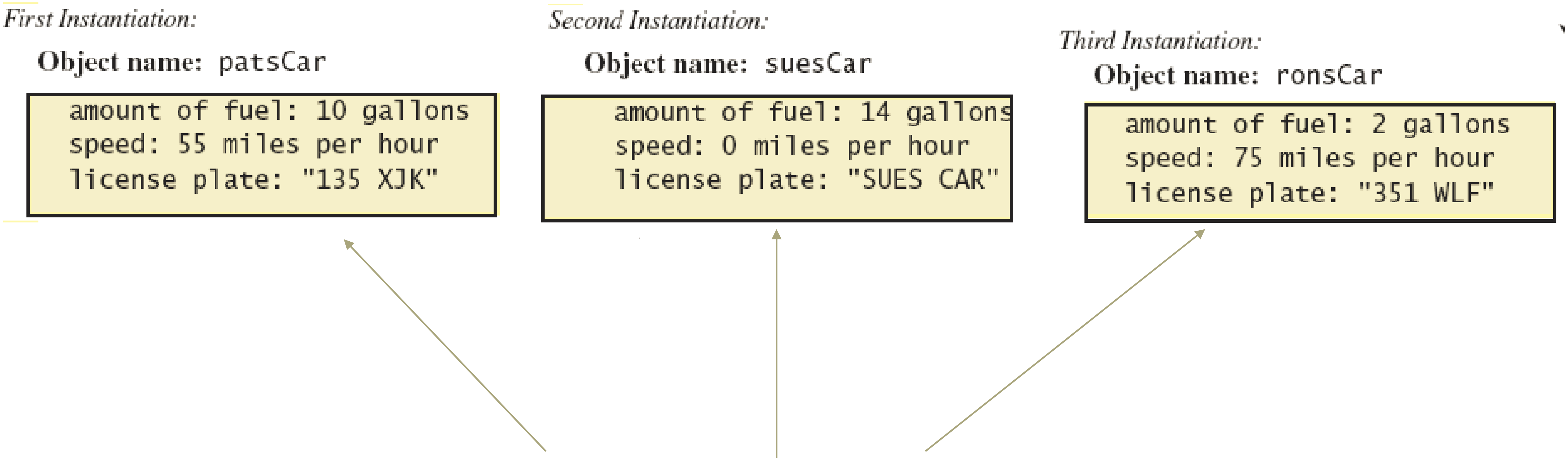
amount of fuel__________
speed __________
license plate __________

Methods (actions):

accelerate:
How: Press on gas pedal.
decelerate:
How: Press on brake pedal.

Class Description

CLASS AND METHOD DEFINITIONS



First Instantiation:

Object name: `patsCar`

```
amount of fuel: 10 gallons
speed: 55 miles per hour
license plate: "135 XJK"
```

Second Instantiation:

Object name: `suesCar`

```
amount of fuel: 14 gallons
speed: 0 miles per hour
license plate: "SUES CAR"
```

Third Instantiation:

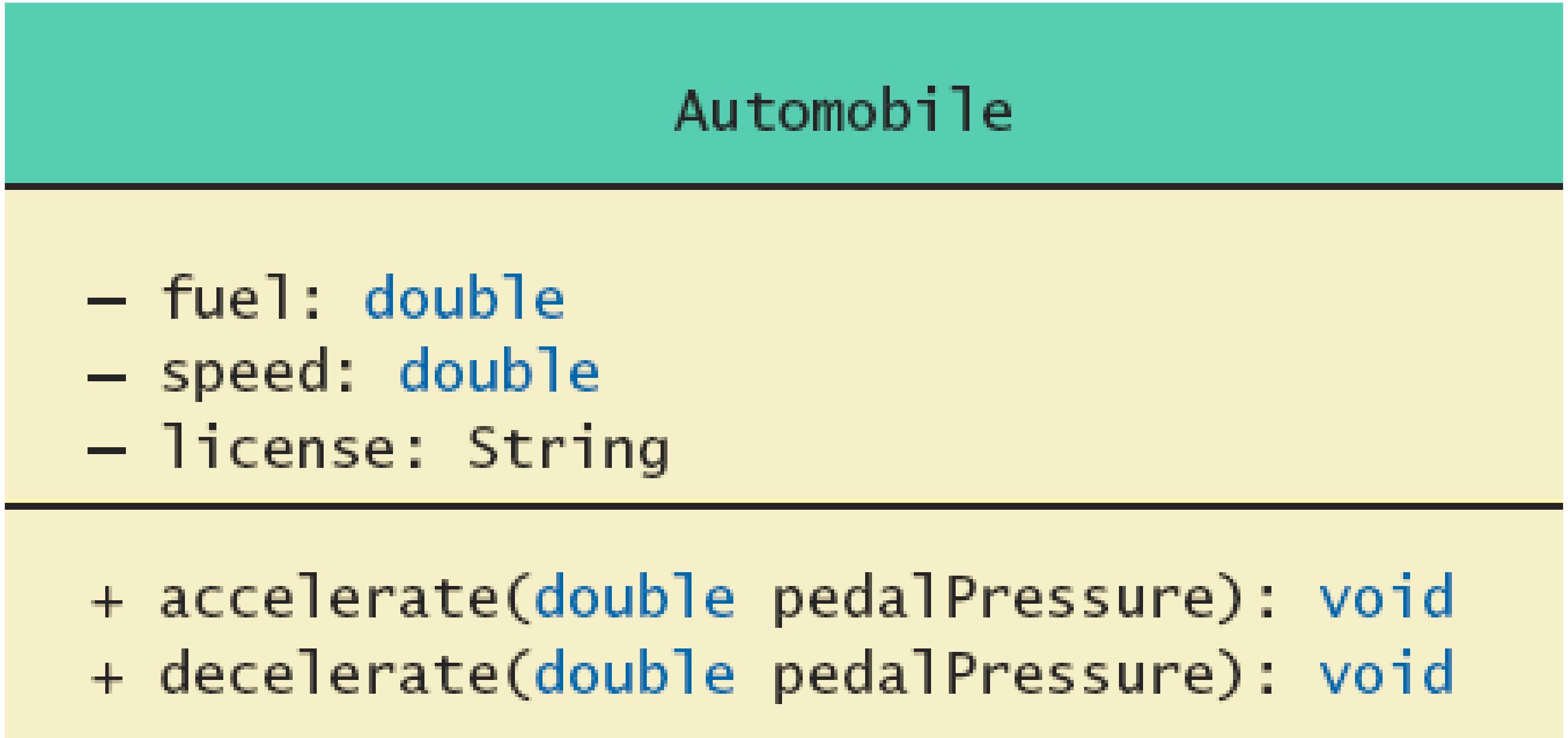
Object name: `ronsCar`

```
amount of fuel: 2 gallons
speed: 75 miles per hour
license plate: "351 WLF"
```

Objects that are instantiations of the class **`Automobile`**

CLASS AND METHOD DEFINITIONS

A class outline as a UML (Universal Modeling Language) class diagram



Automobile

```
– fuel: double
– speed: double
– license: String
```

```
+ accelerate(double pedalPressure): void
+ decelerate(double pedalPressure): void
```

CLASS FILES AND SEPARATE COMPILATION

Each Java class definition usually in a file by itself

- File begins with name of the class
- Ends with .java

Class can be compiled separately

- Helpful to keep all class files used by a program in the same directory

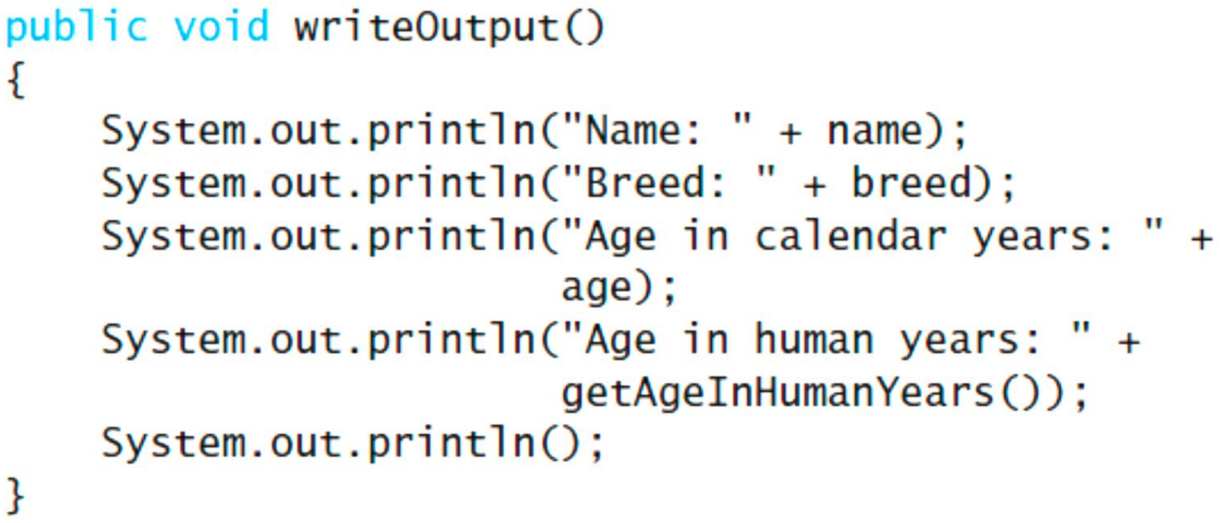
DOG CLASS AND INSTANCE VARIABLES

class Dog

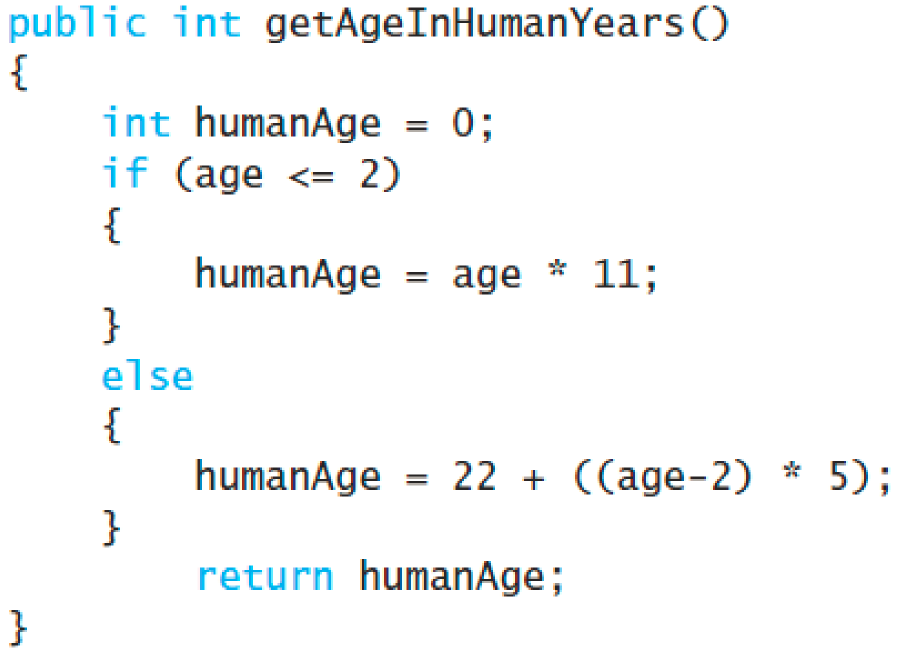
- Dog class has
    - Three pieces of data (instance variables)
    - Two behaviors
- Each instance of this type has its own copies of the data items
- Use of public
    - No restrictions on how variables used

CLASS EXAMPLE

```
public class Dog
{
    public String name;
    public String breed;
    public int age;
```



```
public void writeOutput()
{
    System.out.println("Name:  " + name);
    System.out.println("Breed:  " + breed);
    System.out.println("Age in calendar years: " +
                       age);
    System.out.println("Age in human years: " +
                       getAgeInHumanYears());
    System.out.println();
}
```



```
public int getAgeInHumanYears()
{
    int humanAge = 0;
    if (age <= 2)
    {
        humanAge = age * 11;
    }
    else
    {
        humanAge = 22 + ((age-2) * 5);
    }
        return humanAge;
}
```

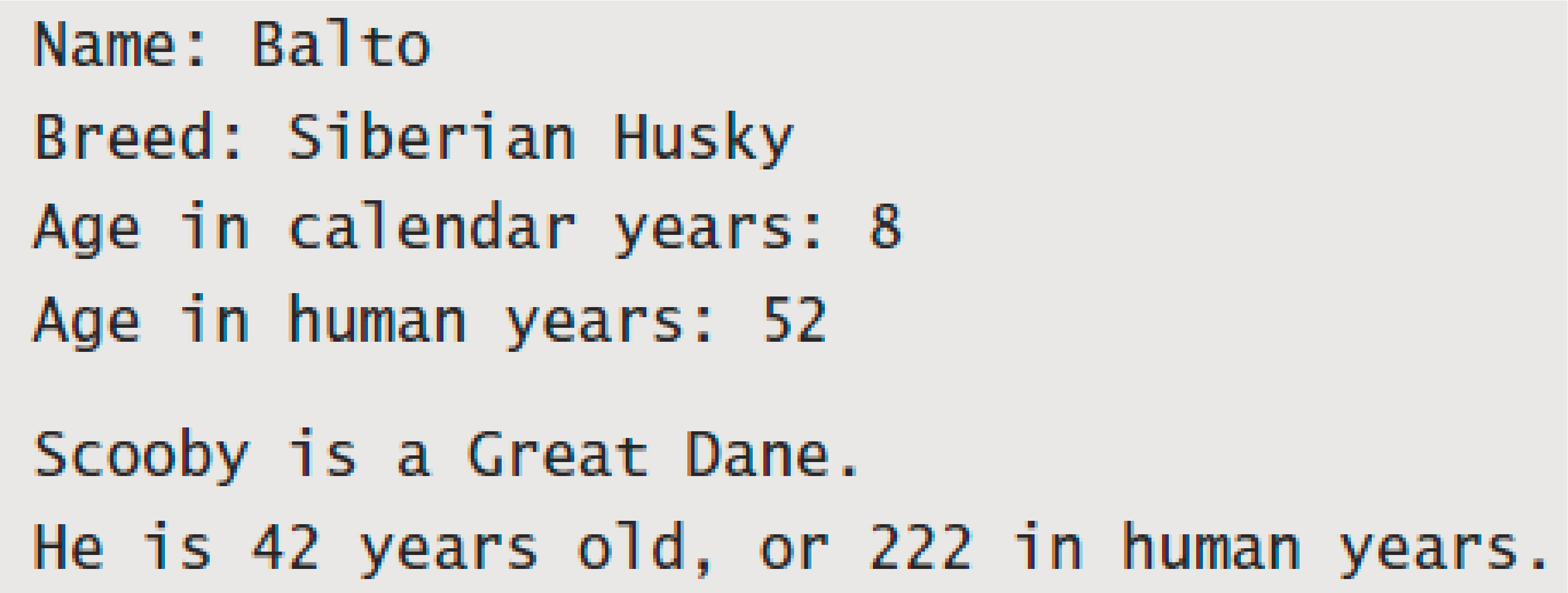
```
}
```

USING DOG CLASS AND ITS METHODS

```
public class DogDemo{
public static void main(String[] args){
    Dog balto = new Dog();
    balto.name = "Balto";
    balto.age = 8;
    balto.breed = "Siberian Husky";
    balto.writeOutput();

    Dog scooby = new Dog();
    scooby.name = "Scooby";
    scooby.age = 42;
    scooby.breed = "Great Dane";
    System.out.println(scooby.name + " is a "
    +scooby.breed + ".");
    System.out.print("He is " + scooby.age +
    " years old, or ");
    int humanYears =
    scooby.getAgeInHumanYears();
    System.out.println(humanYears + " in
    human years.");
  }
  }
```

USING DOG CLASS AND ITS METHODS



```
Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52

Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
```

METHODS

- When you use a method you "invoke" or "call" it
- Two kinds of Java methods
  - Return a single item
  - Perform some other action –a **void** method
- The method **main** is a **void** method
  - Invoked by the system
  - Not by the application program

METHODS

Calling a method that returns a quantity

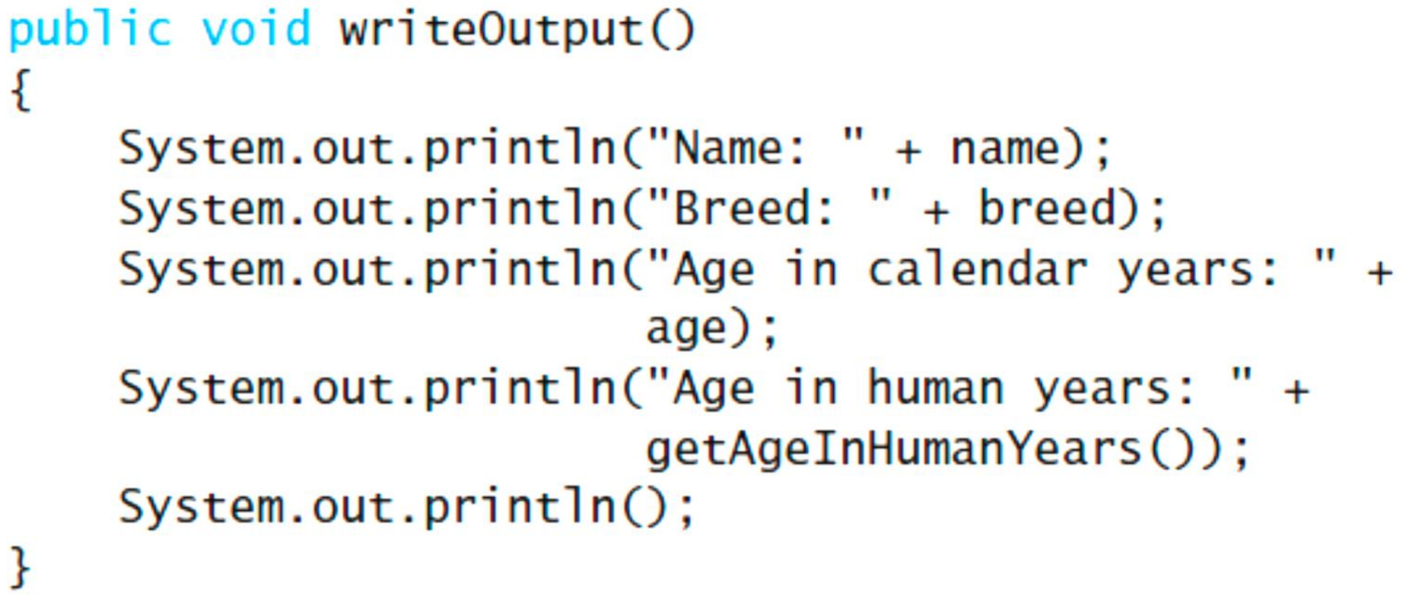
- Use anywhere a value can be used

Calling a void method

- Write the invocation followed by a semicolon
- Resulting statement performs the action defined by the method

DEFINING VOID METHODS

Consider method **`writeOutput()`** from



```
public void writeOutput()
{
    System.out.println("Name: " + name);
    System.out.println("Breed: " + breed);
    System.out.println("Age in calendar years: " +
                       age);
    System.out.println("Age in human years: " +
                       getAgeInHumanYears());
    System.out.println();
}
```

- Method definitions appear inside class definition
- Can be used only with objects of that class

DEFINING VOID METHODS

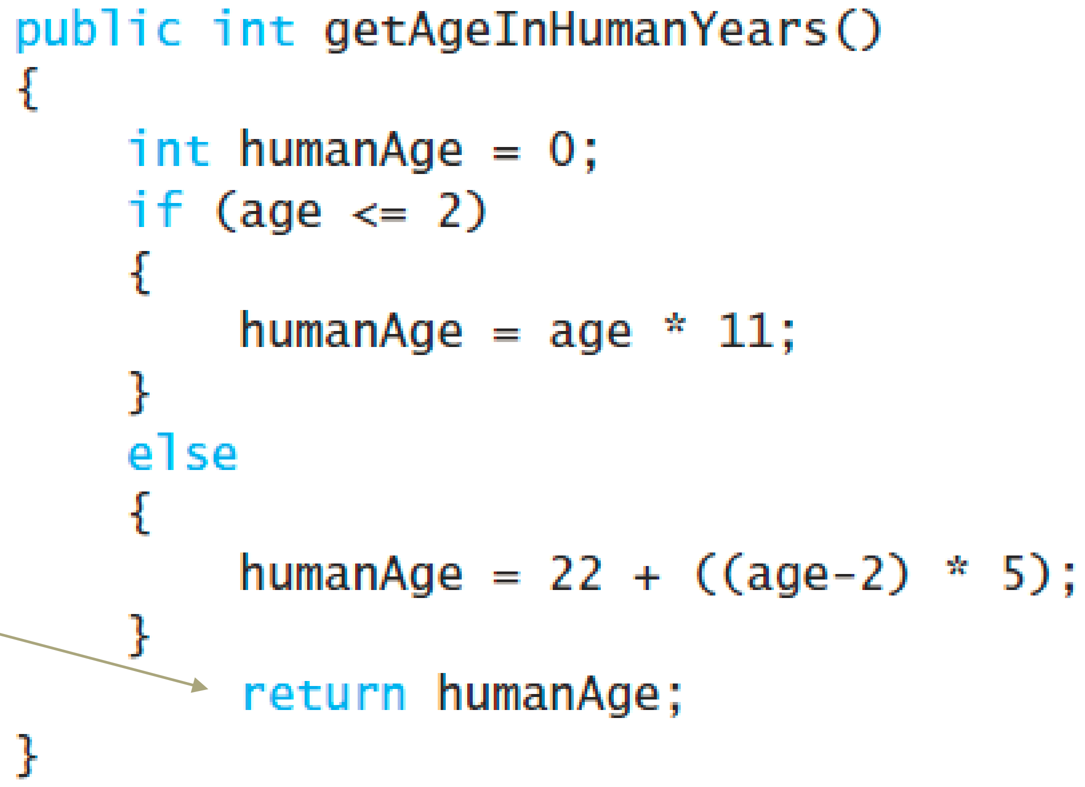
- Most method definitions we will see as public
- Method does not return a value
- Specified as a **void** method
- Heading includes parameters
- Body enclosed in braces **{ }**
- Think of method as defining an action to be taken

METHODS THAT RETURN A VALUE

Consider method **getAgeInHumanYears( )**

Heading declares type of value to be returned

Last statement executed is return



```
public int getAgeInHumanYears()
{
    int humanAge = 0;
    if (age <= 2)
    {
        humanAge = age * 11;
    }
    else
    {
        humanAge = 22 + ((age-2) * 5);
    }
        return humanAge;
}
```

THE KEYWORD THIS

Referring to instance variables outside the class –must use

- Name of an object of the class
- Followed by a dot
- Name of instance variable

Inside the class,

- Use name of variable alone
- The object (unnamed) is understood to be there

THE KEYWORD THIS

- Inside the class the unnamed object can be referred to with the name this
- Example
  - **this.name = keyboard.nextLine();**
- The keyword **this** stands for the receiving object
- We will seem some situations later that require the this

LOCAL VARIABLES

Variables declared inside a method are called *local* variables

- May be used only inside the method
- All variables declared in method **main** are local to **main**

Local variables having the same name and declared in different methods are different variables

BLOCKS

- Recall compound statements
  - Enclosed in braces {}
- When you declare a variable within a compound statement
  - The compound statement is called a *block*
  - The scope of the variable is from its declaration to the end of the block
- Variable declared outside the block usable both outside and inside the block

PARAMETERS OF PRIMITIVE TYPE

Note the declaration **public int predictPopulation(intyears)**

The *formal* parameter is **years**

Calling the method **int futurePopulation=predictPopulation(10);**

The *actual* parameter is the integer 10

View sample program, **class SpeciesSecondClassDemo**

```java
public class SpeciesSecondTryDemo{
public static void main(String[] args){
SpeciesSecondTry speciesOfTheMonth = new SpeciesSecondTry();
System.out.println("Enter data on the Species of the " + "Month:");
speciesOfTheMonth.readInput();
speciesOfTheMonth.writeOutput();
int futurePopulation = speciesOfTheMonth.predictPopulation(10);
speciesOfTheMonth.name = "Klingon ox";
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
System.out.println("In ten years the population will be " +
speciesOfTheMonth.predictPopulation(10));}}
```

PARAMETERS OF PRIMITIVE TYPE

- Parameter names are local to the method
- When method invoked
  - Each parameter initialized to value in corresponding actual parameter
  - Primitive actual parameter cannot be altered by invocation of the method
- Automatic type conversion performed

byte -> short -> int-> long -> float -> double

INFORMATION HIDING, ENCAPSULATION: OUTLINE

- Information Hiding
- Pre-and Postcondition Comments
- The public and private Modifiers
- Methods Calling Methods
- Encapsulation
- Automatic Documentation with javadoc
- UML Class Diagrams

INFORMATION HIDING

- Programmer using a class method need not know details of implementation
  - Only needs to know *what* the method does
- Information hiding:
  - Designing a method so it can be used without knowing details
- Also referred to as *abstraction*
- Method design should separate *what* from *how*

PRE AND POST CONDITIONS

- Precondition comment
    - States conditions that must be true before method is invoked
- Example

```
/**
Precondition: The instance variables of the calling object have values.
Postcondition: The data stored in (the instance variables of) the
receiving object have been written to the screen.
*/
public void writeOutput()
```

PRE AND POST CONDITIONS

- Postcondition comment
- Tells what will be true after method executed
- Example

```
/**
Precondition: years is a nonnegative number.
Postcondition: Returns the projected population of the
receiving object after the specified number of years.
*/
public int predictPopulation(int years)
```

PUBLIC AND PRIVATE MODIFIERS

- Type specified as public
    - Any other class can directly access that object by name
- Classes generally specified as public
- Instance variables usually not **public**
    - Instead specify as private

```
public class Rectangle{
private int width;
private int height;
private int area;
public void setDimensions(int newWidth, int newHeight)
{
    width = newWidth;
    height = newHeight;
    area = width * height;
}
public int getArea()
{
        return area;
}
}
```

```
Rectangle box = new Rectangle( );
box.setDimensions(10, 5);
System.out.println("The area of our
rectangle is " + box.getArea());
```

EXAMPLE

- Demonstration of need for private variables
- Statement such as
    - **box.width= 6;**
- is illegalsince width is **private**
- Keeps remaining elements of the class consistent in this example

EXAMPLE

- Another implementation of a Rectangle class
- class Rectangle2
- Note setDimensions method
- This is the only way the **width** and **height** may be altered outside the class

ACCESSOR AND MUTATOR METHOD

- When instance variables are private must provide methods to access values stored there
  - Typically named getSomeValue
  - Referred to as an accessor method
- Must also provide methods to change the values of the private instance variable
  - Typically named setSomeValue
  - Referred to as a mutator method

```
import java.util.Scanner;
public class SpeciesFourthTry{
private String name;
private int population;
private double growthRate;

public void setSpecies(String newName, int newPopulation,
double newGrowthRate){
name = newName;
if (newPopulation >= 0)
population = newPopulation;
else{
System.out.println("ERROR: using a negative population.");
System.exit(0);
}
growthRate = newGrowthRate;
}
```

```
public String getName()
{
return name;
}
public int getPopulation()
{
return population;
}
public double getGrowthRate()
{
return growthRate;
}
}
```

EXAMPLE

```
public class Dog{
int dogAge;
public Dog(String name)
{
System.out.println(“Dog’s name is :" + name );
}
public void setAge( intage )
{
dogAge= age;
}
public intgetAge( )
{
System.out.println("Dog's age is :" + dogAge);
return dogAge;
}
```

```
public static void main(String []args){
Dog dogObj= new Dog( "Divine" );
dogObj.setAge( 3 );
dogObj.getAge( );
System.out.println("Dog's age is:" +
dogObj.dogAge);
}
}
```

Output:

```
Dog’s name is :Divine
Dog's age is :3
Dog's age is:3
```