CS62 Class 4: Memory Management, Inheritance

Java Fundamentals

Lecture 4 agenda

- Memory management: stack vs heap & garbage collection
- Inheritance
- Polymorphism

Memory management in Java

What happens to our Java code

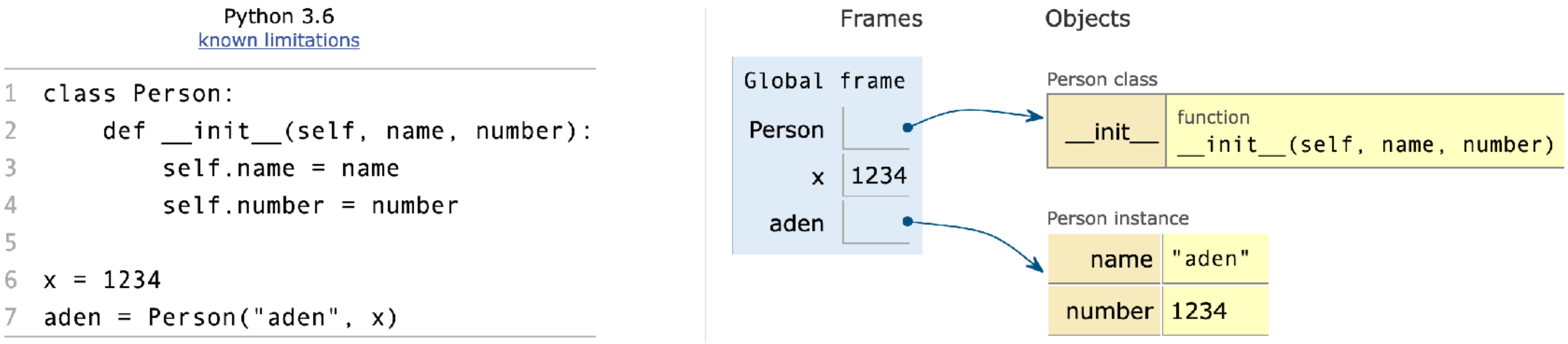
- We write our source code in .java files
- The javac Java compiler compiles the source code into bytecode.
  - This will result in .class files that match the source code file names.
  - This is compile time.
- The JVM Java Virtual Machine will translate bytecode into native machine code.
  - WORA is one of the main powers of Java: Write Once, Run Anywhere (or Away, depending on whom you ask).
  - This is runtime.

Typical structure of a Java project

- `src` - source files (.java), might be organized within packages
- `bin` - bytecode files (.class)
- `lib` - libraries and other dependencies

Stack vs heap (review in Python)

- Recall using Python Tutor to step through your code, recall drawing stack frames in CS51P



```python
class Person:
    def __init__(self, name, number):
        self.name = name
        self.number = number

x = 1234
aden = Person("aden", x)
```

Stack frames are the **stack**

Static memory allocation, contains method calls and primitives (like x = 1234)

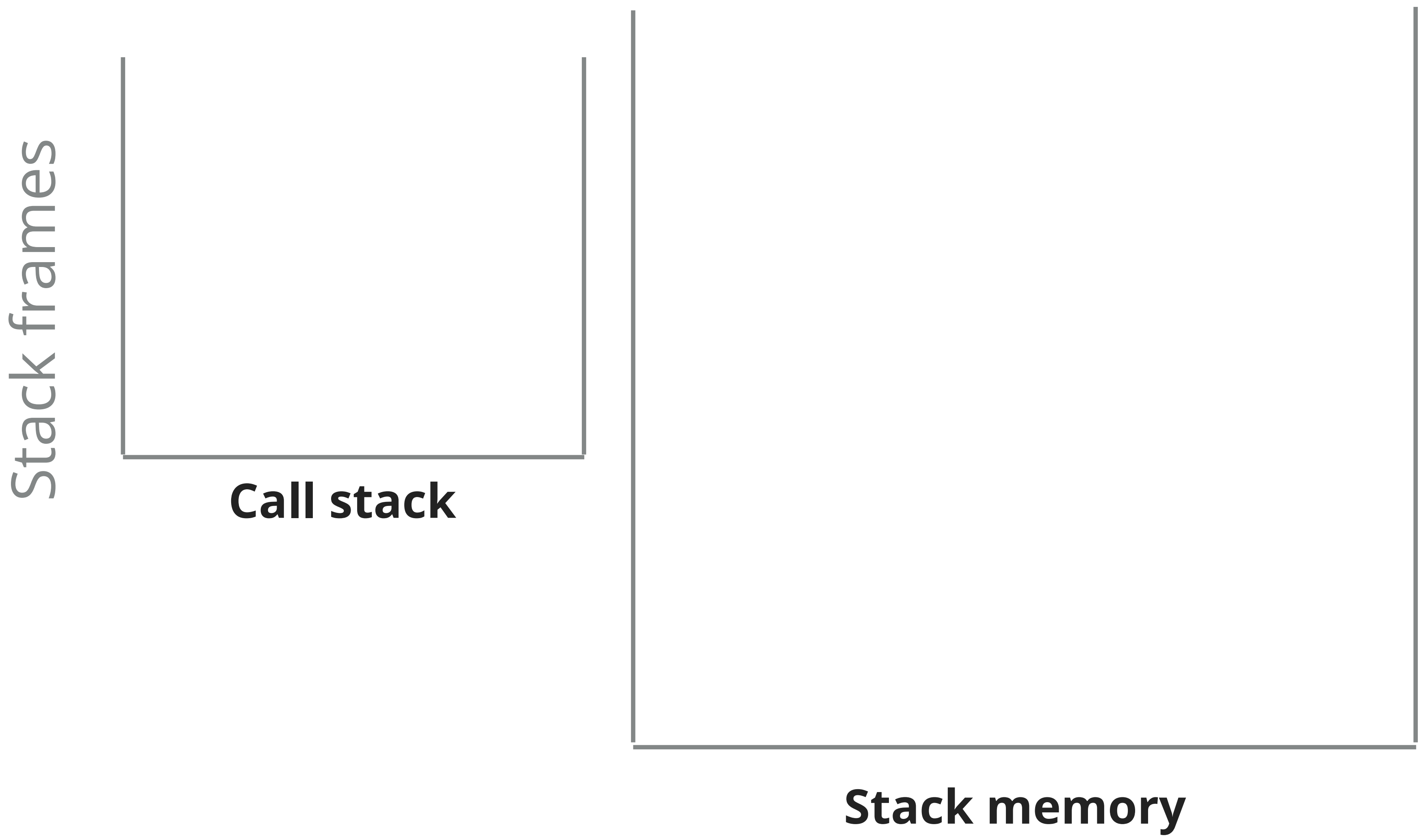
Fast, follows a last-in-first-out order

Objects are stored in the **heap**

Dynamic memory allocation (since we don’t know how big objects are at compile time)

Slower

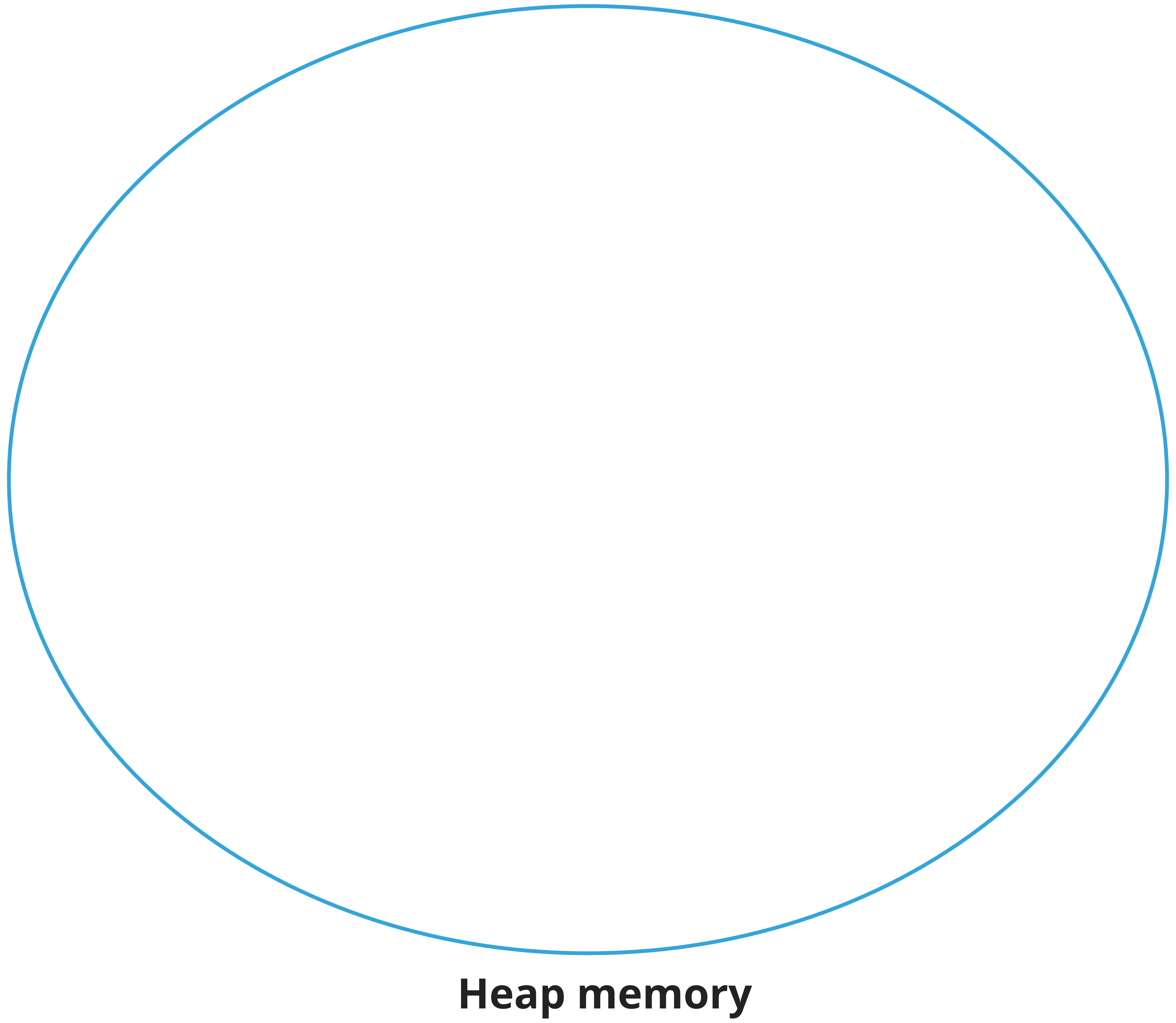
Stack vs heap in Java (walkthrough)



Stack frames

Call stack

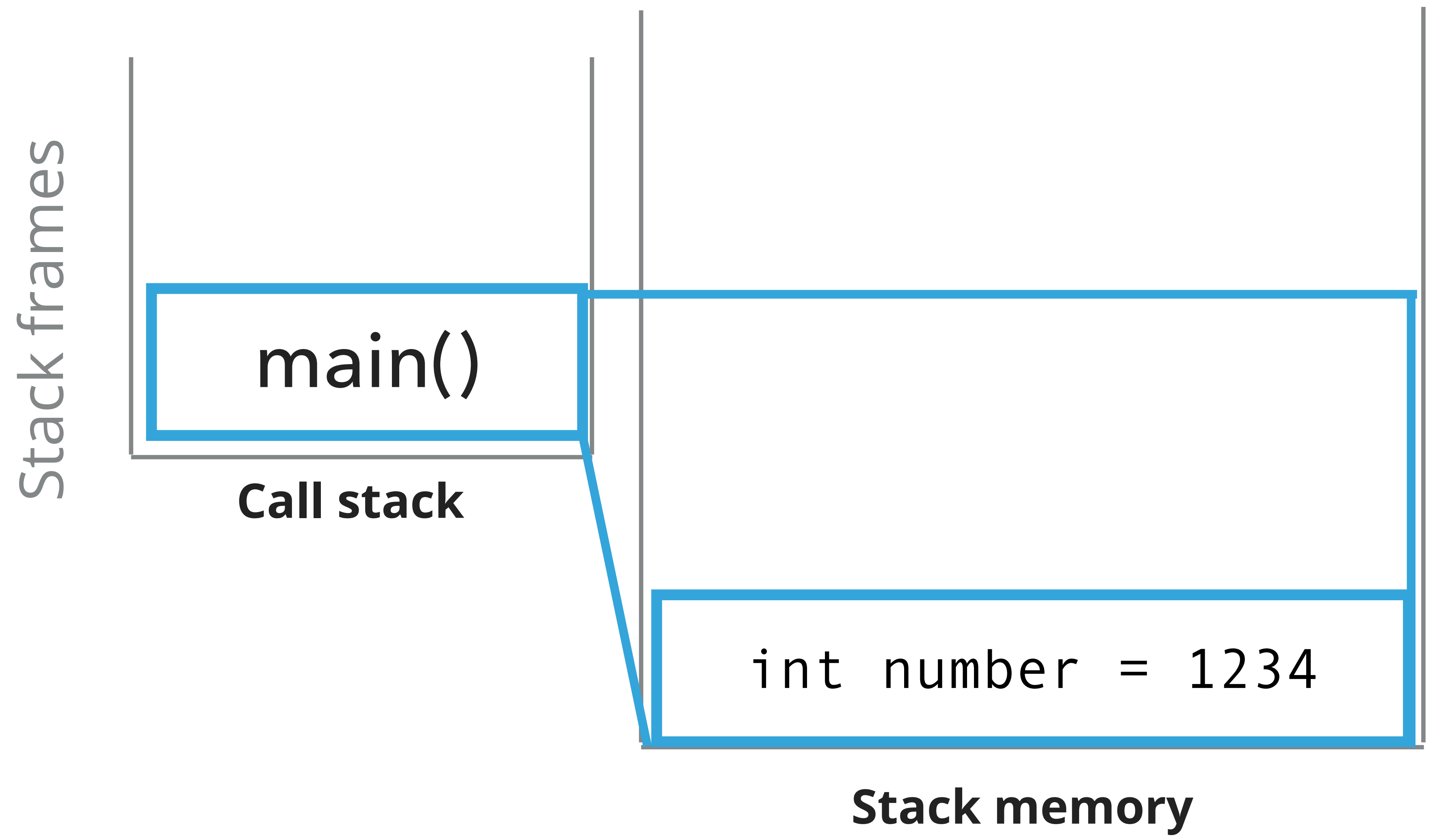
Stack memory

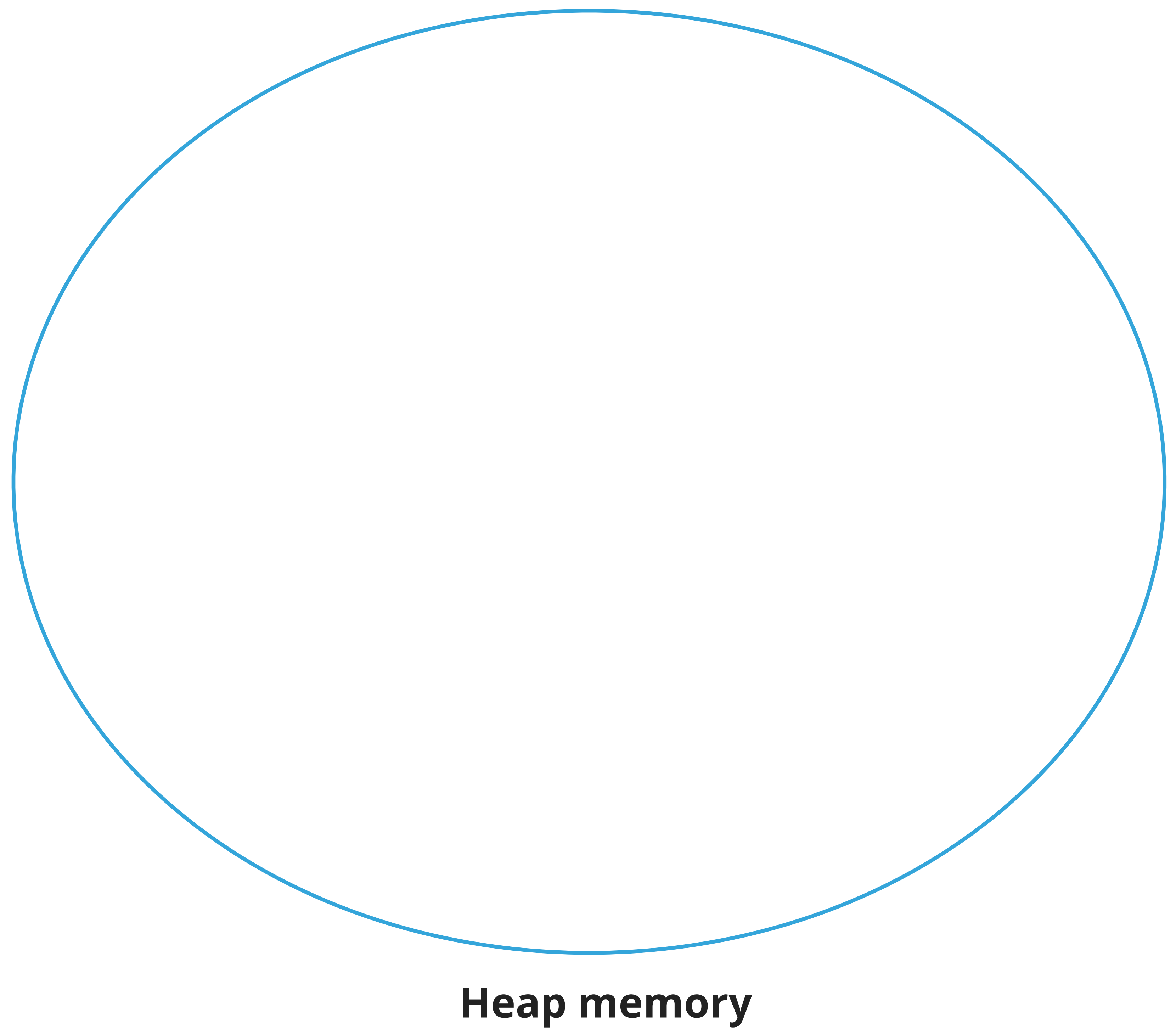


Heap memory

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }
   public static void main(String args[]) {
       int number = 1234;
       String name = “Aden”;
       Person aden = null;
       aden = new Person(name, number)
  }
```

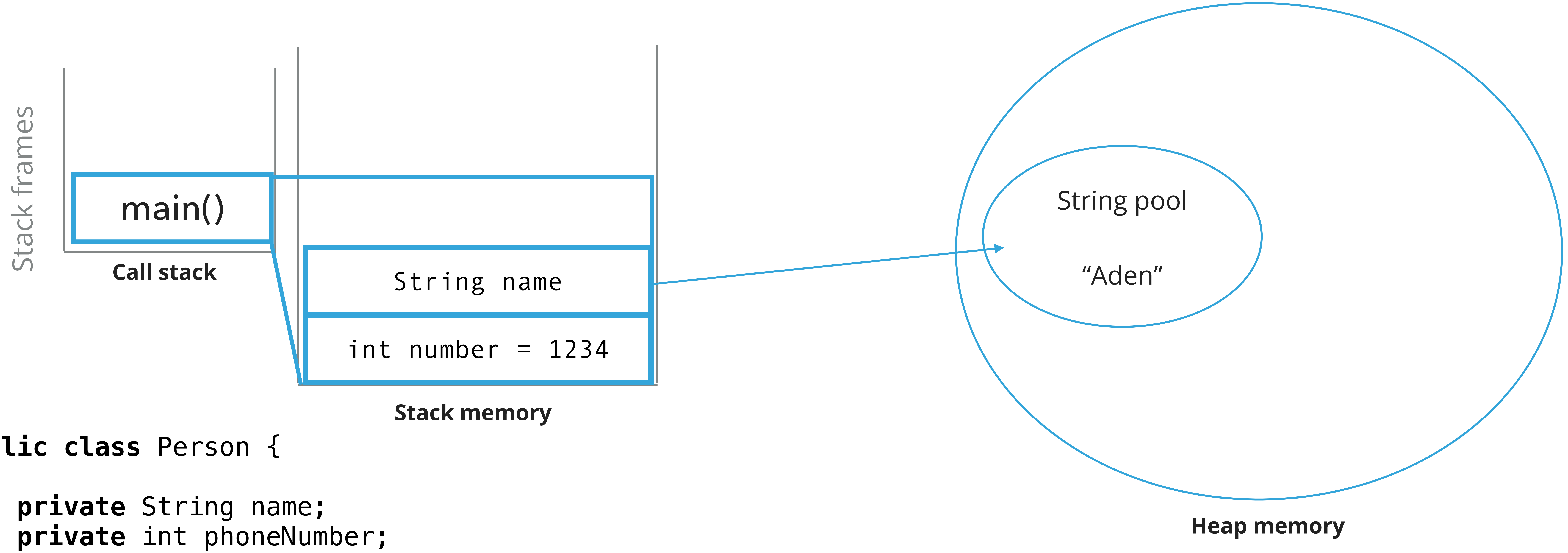
Stack vs heap in Java (walkthrough)





```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
       int number = 1234;
       String name = “Aden”;
       Person aden = null;
       aden = new Person(name, number)
  }
```

Stack vs heap in Java (walkthrough)

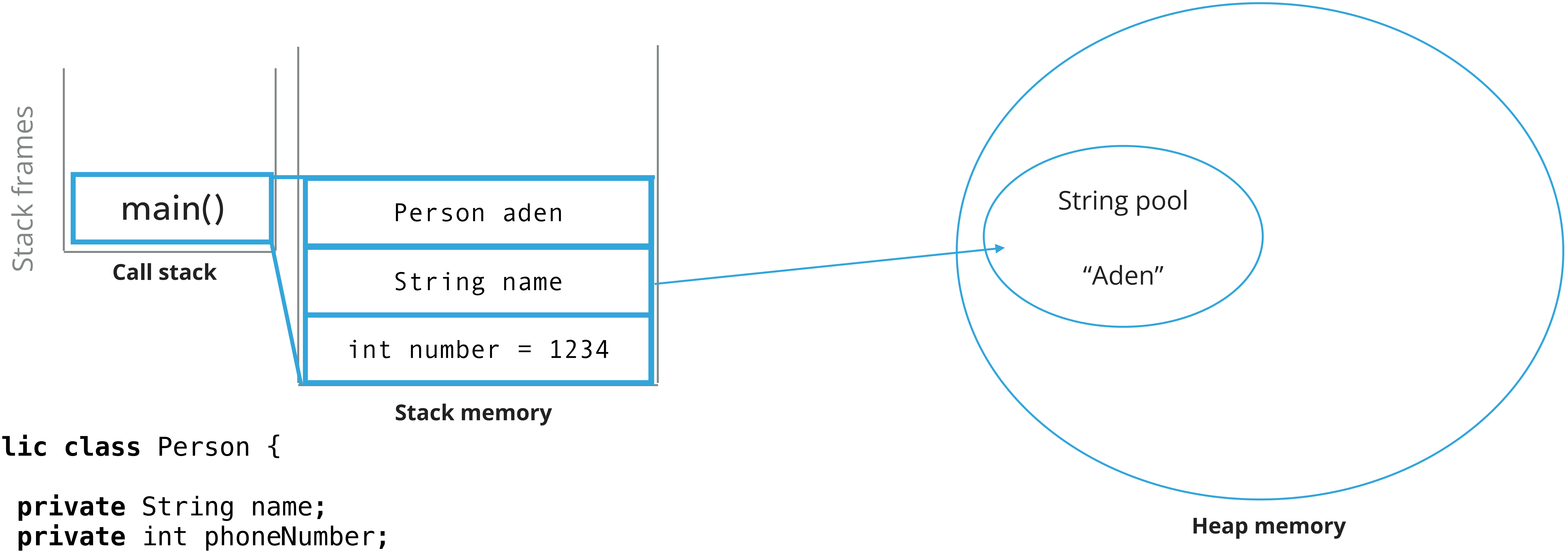


```
public class Person {

  private String name;
  private int phoneNumber;

  public Person(String name, int phoneNumber) {
    this.name = name;
    this.phoneNumber = phoneNumber;
  }

  public static void main(String args[]) {
    int number = 1234;
    String name = “Aden”;
    Person aden = null;
    aden = new Person(name, number)
 }
```

The reference to the String is stored in the stack

The actual String object is in the heap in Java’s “String pool”

Stack vs heap in Java (walkthrough)



Stack frames

main()

Call stack

Person aden

String name

int number = 1234

Stack memory

String pool

“Aden”

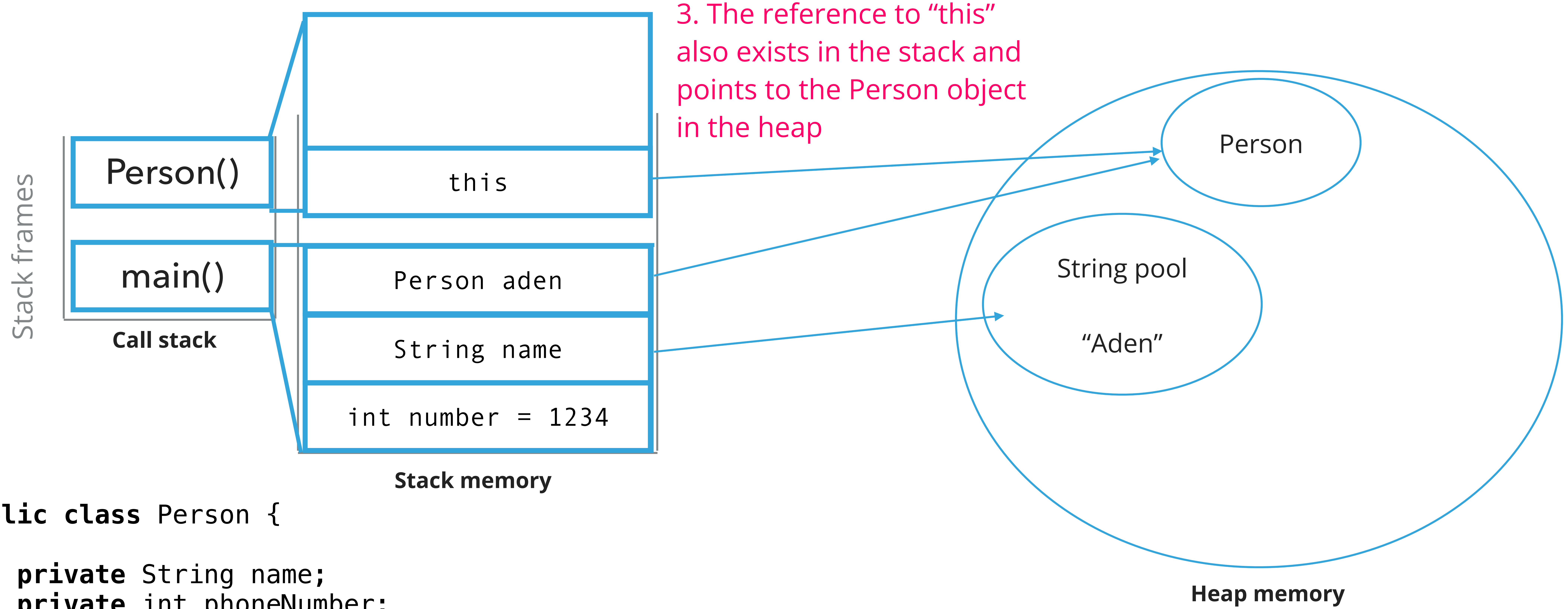
Heap memory

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
       int number = 1234;
       String name = “Aden”;
       Person aden = null;
       aden = new Person(name, number)
  }
```

The reference to the Person is stored in the stack

It doesn’t exist in the heap yet since we set it to null originally

Stack vs heap in Java (walkthrough)



3. The reference to “this” also exists in the stack and points to the Person object in the heap

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
       int number = 1234;
       String name = “Aden”;
       Person aden = null;
       aden = new Person(name, number)
  }
```

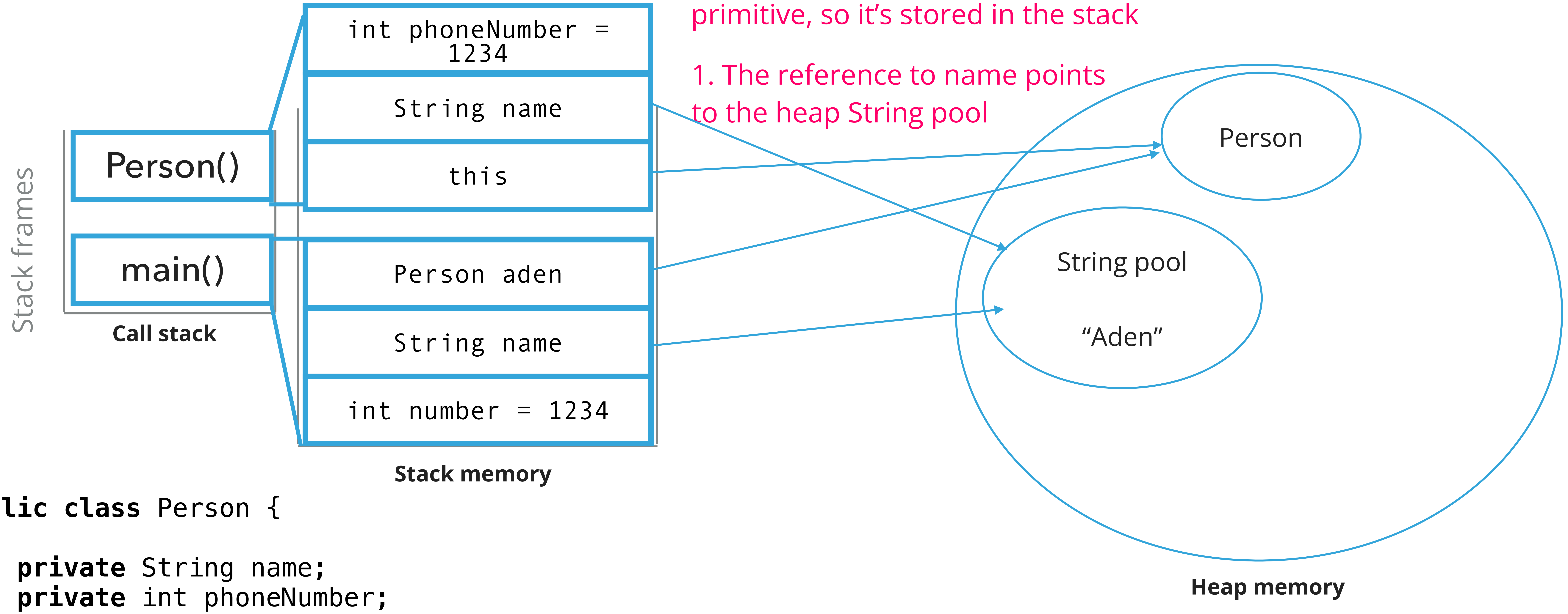
2. It creates a Person object in the heap

1. The call to the Person constructor method goes in the stack

Stack vs heap in Java (walkthrough)

2. But the phoneNumber is a primitive, so it’s stored in the stack

1. The reference to name points to the heap String pool



Stack frames

Person()

main()

Call stack

int phoneNumber = 1234

String name

this

Person aden

String name

int number = 1234

Stack memory

Person

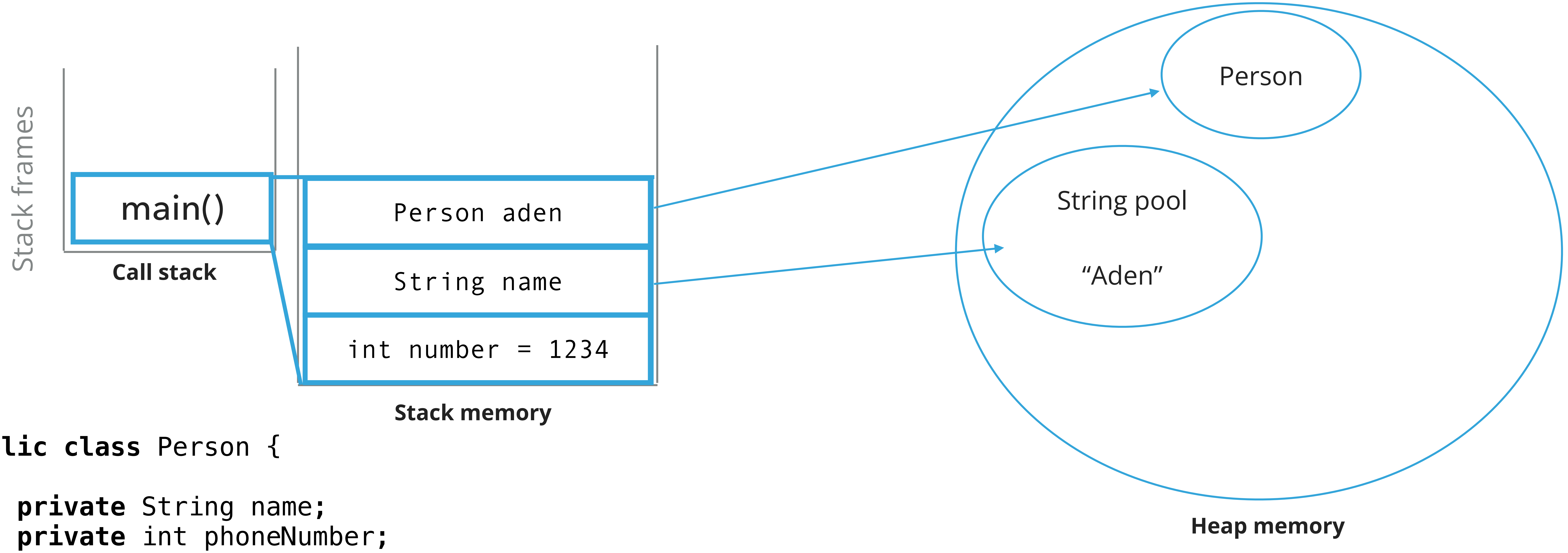
String pool

“Aden”

Heap memory

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
       int number = 1234;
       String name = “Aden”;
       Person aden = null;
       aden = new Person(name, number)
  }
```

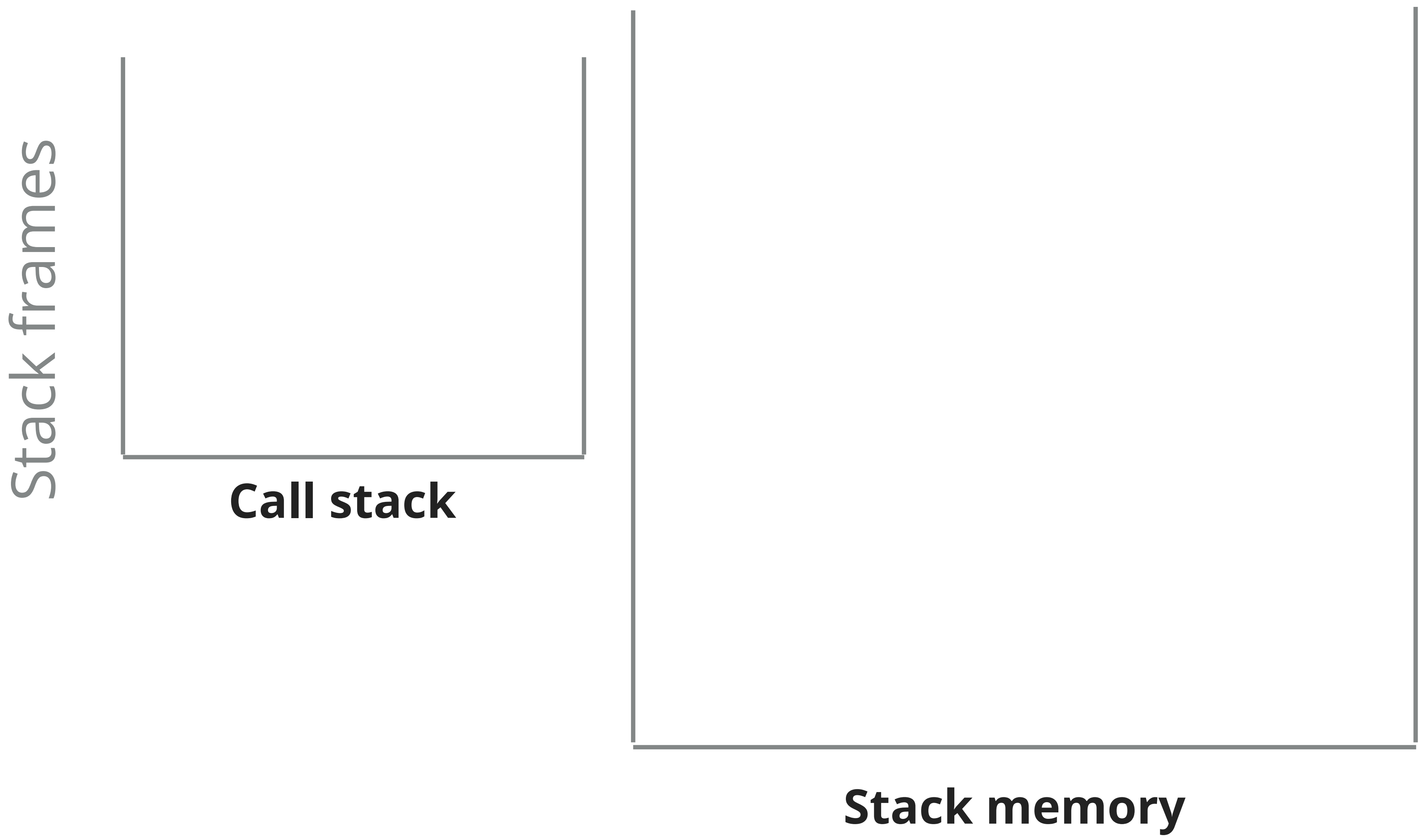
Stack vs heap in Java (walkthrough)

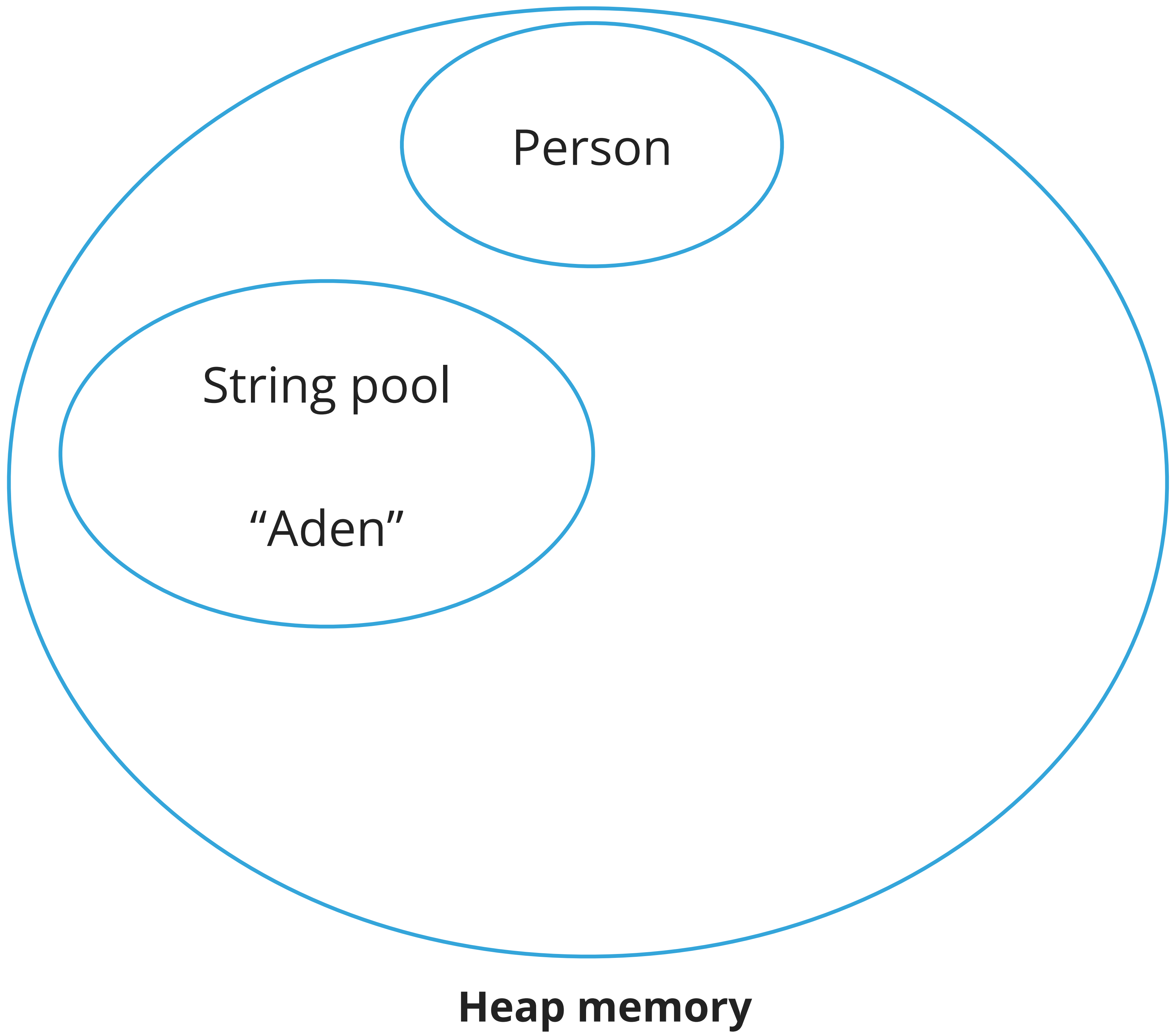


```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
       int number = 1234;
       String name = “Aden”;
       Person aden = null;
       aden = new Person(name, number)
  }
```

Once the constructor call ends, it’s wiped from the stack

Stack vs heap in Java (walkthrough)



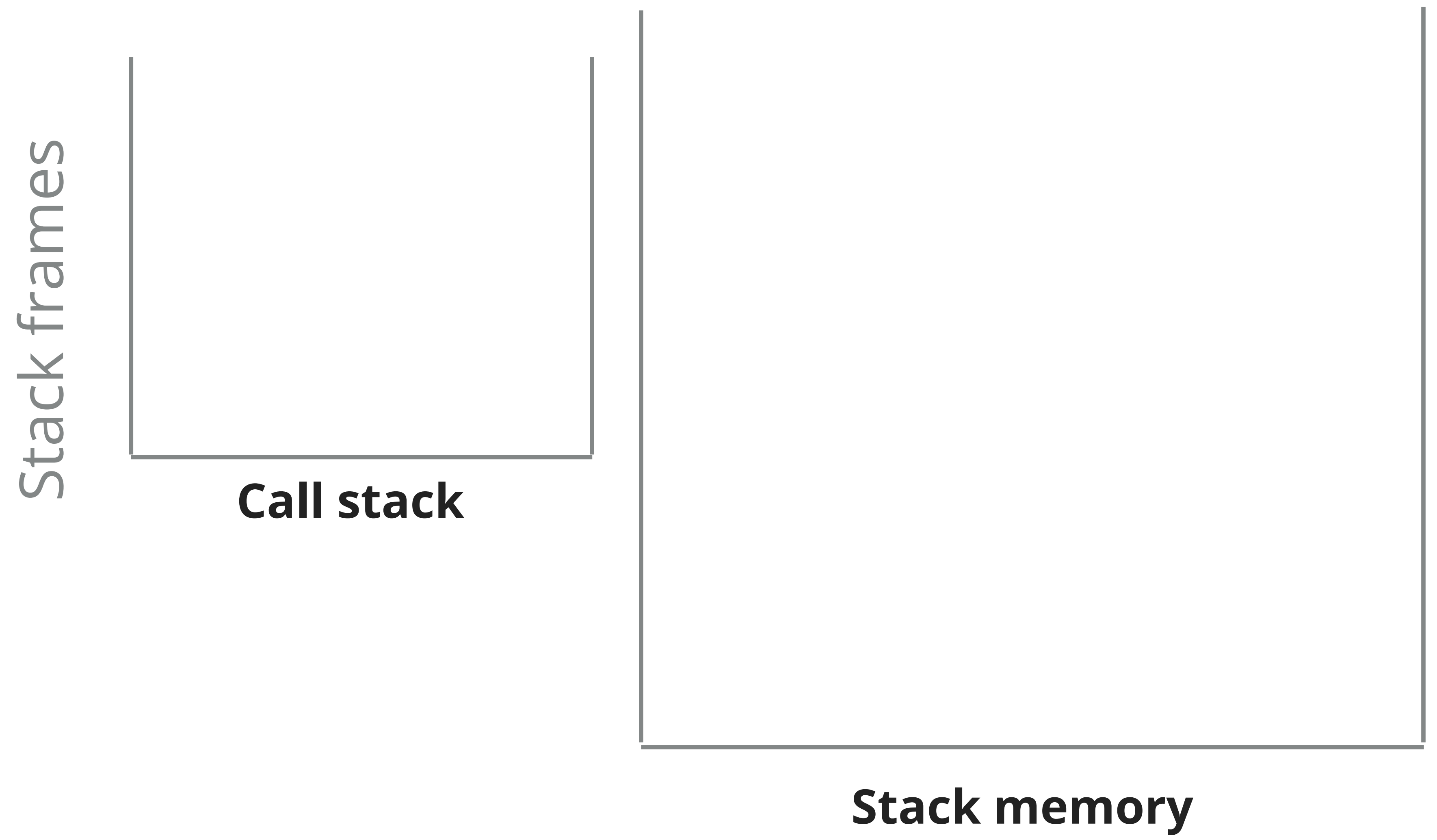


We’re just left with our heap objects with no references to them :’(

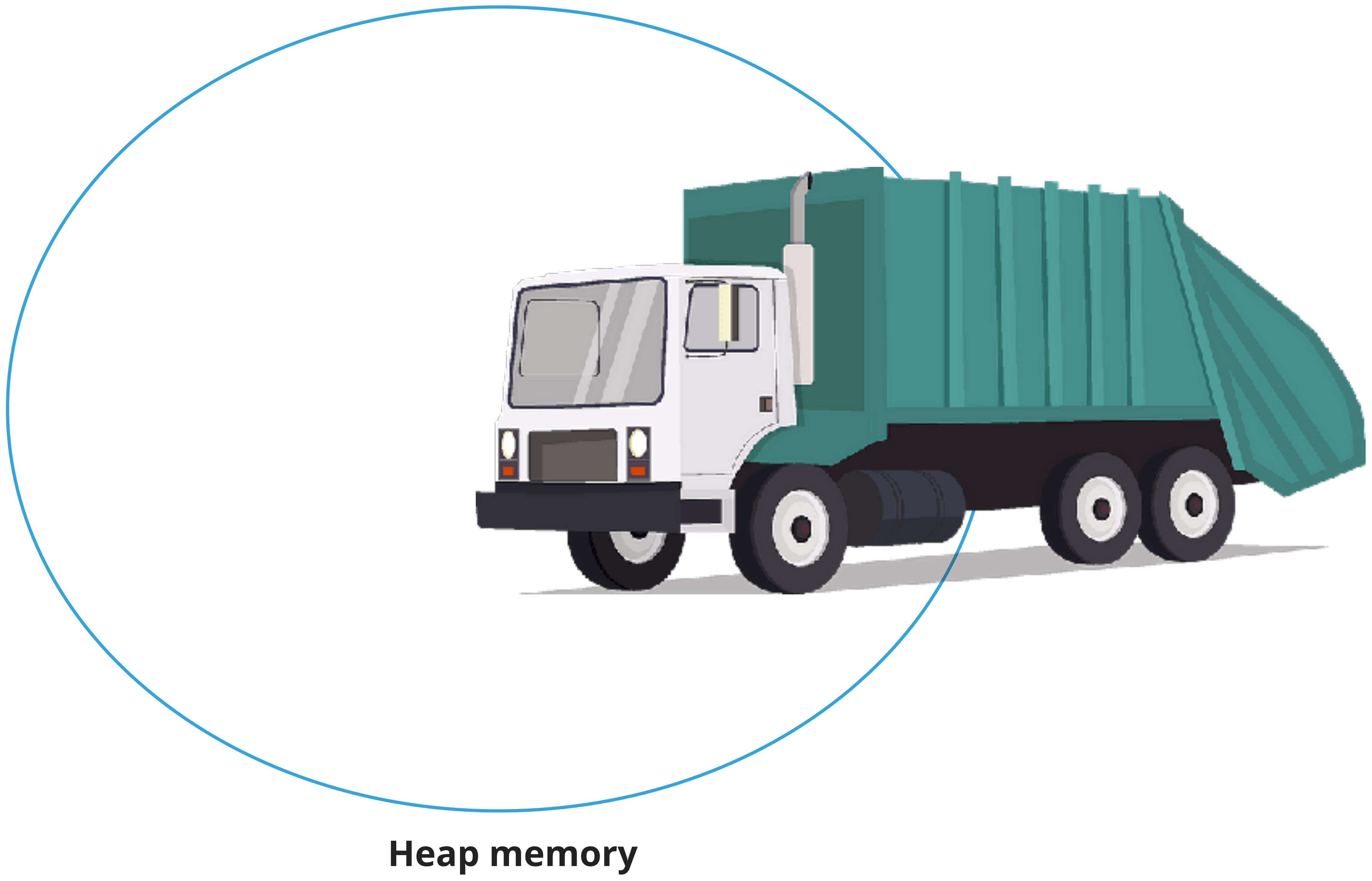
```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
       int number = 1234;
       String name = “Aden”;
       Person aden = null;
       aden = new Person(name, number)
  }
```

Once the main call ends, it’s wiped from the stack

Stack vs heap in Java (walkthrough)



```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
       int number = 1234;
       String name = “Aden”;
       Person aden = null;
       aden = new Person(name, number)
  }
```



Java automatically runs a garbage collector to get rid of heap objects that have been unreferenced and unused :D

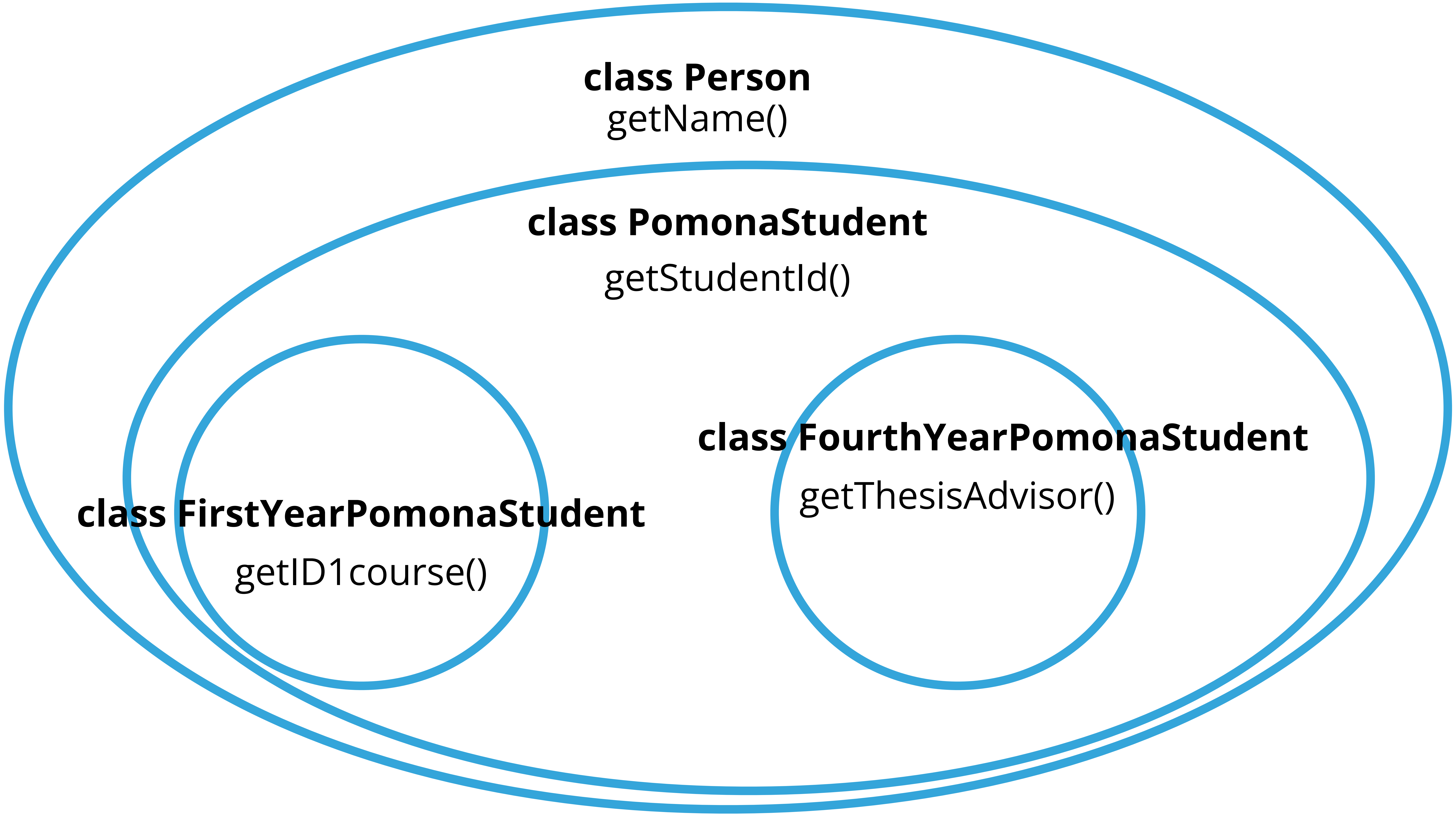
Summary

- The memory in the stack is fast and is for primitives & function calls
- The memory in the heap is slower and is for objects
- A garbage collector comes around and collects unused memory in the heap (as a programmer, you don’t have much control over this)

Inheritance

Inheritance conceptual overview

- Classes can be parent/child classes of each other (subclasses)



Changes to PomonaStudent class

```
package registrar;
class PomonaStudent {
    private String name;
    private String email;
    private int id;
    private String major;
    private static int studentCounter;

    protected PomonaStudent(String name, String email, int id){
        this.name = name;
        this.email = email;
        this.id = id;
        major = "Undeclared";
        studentCounter++;
   }
    //protected setters getters
    protected int getMaxCredits(){
        return 4;
    }
    public String toString(){
        return "Name: " + name + "\nemail: " + email + "\nid: " + id + "\n";
    }
}
```

Students across different years have some unique characteristics

- First-year students take ID1
- Fourth-year students write a thesis
- Second-year students and above can take 6 credits
- Transfer students take 1 PE class instead of 2, etc.
- But, they still are Pomona students so the basic information we would need about them doesn’t change.

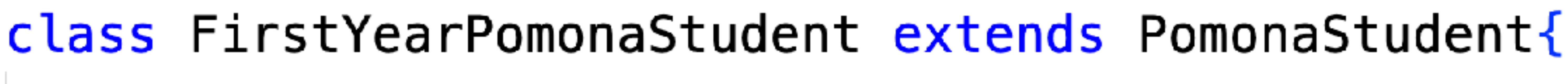
Inheritance

- When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can **reuse** the variables and methods of the existing class without having to write (and debug!) them.

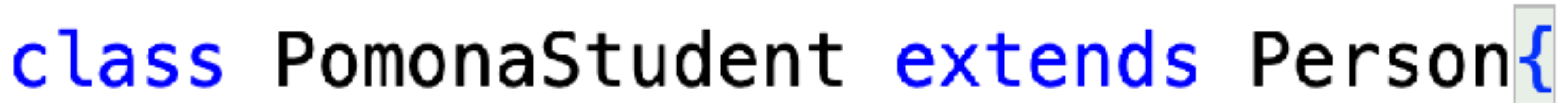
- A class that is derived from another is called a subclass or child class.

- The class from which the subclass is derived is called a superclass or parent class.

- Java allows multilevel inheritance: A class can extend a class which extends a class etc.



```
class FirstYearPomonaStudent extends PomonaStudent{
```



```
class PomonaStudent extends Person{
```

Inheritance

- The subclass inherits all the `public` and `protected` variables and methods.
  - Not the `private` ones, although it can access them with appropriate getters and setters.
- The inherited variables can be used directly, just like any other variables.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
  - We have already done that! (How?)

All classes inherit class `Object`

- Directly if they do not extend any other class, or indirectly as descendants.
- `Object` class has built-in methods that are inherited.
- `public String toString()`
  - Returns string representation of object – default is hexadecimal hash of memory location.
  - We've overrode this!
- `public boolean equals (Object other)`
  - Default behavior uses == returns true only if this and other are located in same memory location.
  - Works fine for primitives but not objects. We would need to override it (more later).
- `public int hashCode()`
  - Unique identifier defined so that if `a.equals(b)` then `a, b` have same hash code (more later).

use the `super` keyword to access the parent

- Refers to the direct parent of the subclass.
  - E.g., `super` in `FirstYearPomonaStudent` refers to `PomonaStudent`
- `super.instanceMethod()`: for overridden methods.
  - What is an overridden method? If FirstYearPomonaStudent has a method that's the same name as a method in PomonaStudent, but we want to call the PomonaStudent one instead, we need to use `super`.
- `super(args)`: to call the constructor of the super class. Should be called in the first line of the subclass's constructor.

Finally, some code

```
package registrar;
```

remember: for inheritance, you need the extends keyword

```
class FirstYearPomonaStudent extends PomonaStudent{
   private String id1;
```

A unique instance variable to first years: the ID1 course

```
   private static int firstYearCounter;
```

Keep track of freshmen

```
   protected FirstYearPomonaStudent(String name, String email, int id,
String id1){
        super(name, email, id);
```

calls PomonaStudent(name, email, id)

```
        this.id1 = id1;
        firstYearCounter++;
```

then fill in the extra, subclass specific stuff

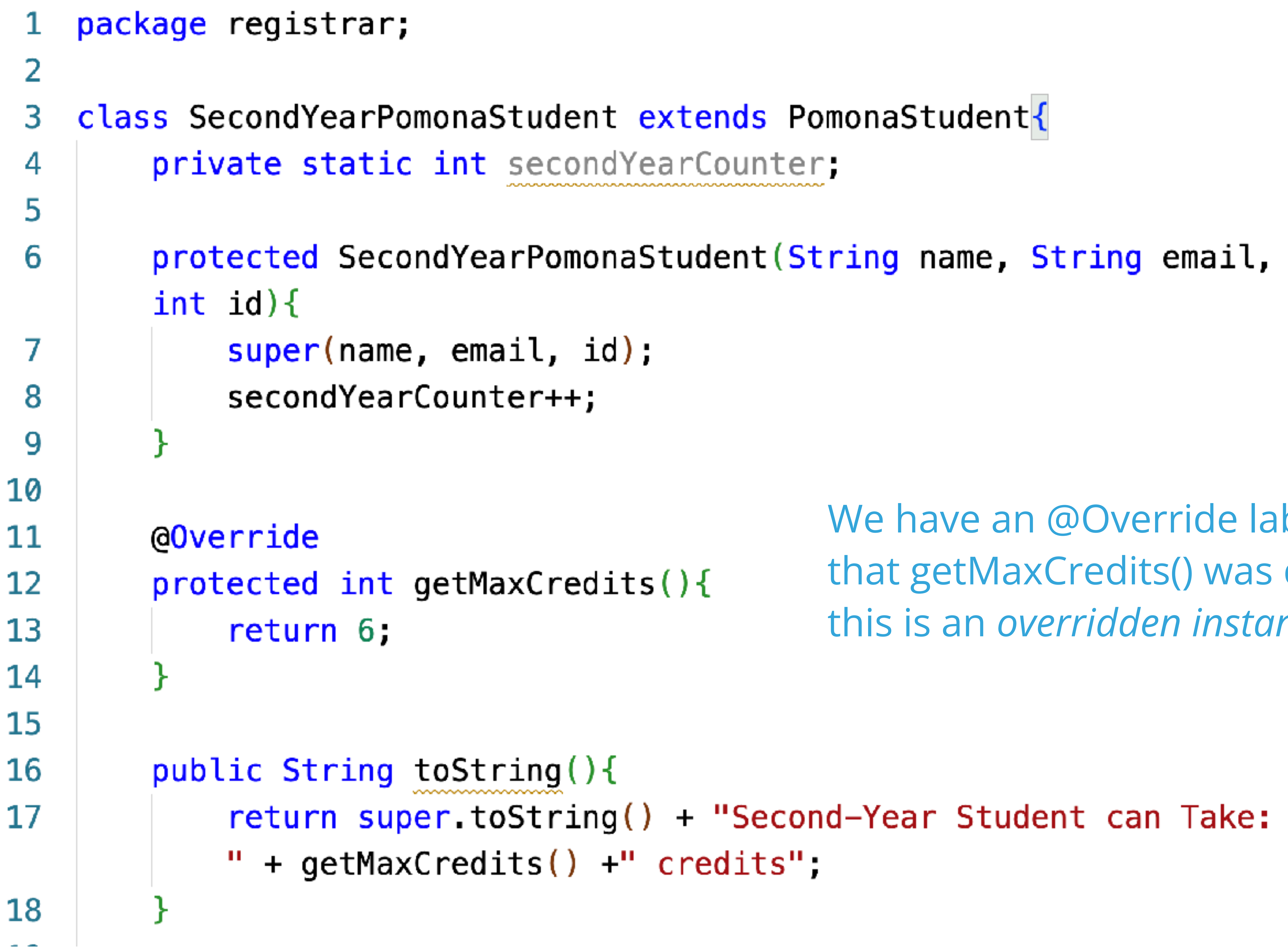
```
    }
   //getters and setters

  public String toString(){
        return super.toString() + "First-Year Student Attending ID1: " +
id1;
    }
}
```

calls PomonaStudent.toString()

Q: Why do we need super? Why can't we write PomonaStudent.toString() directly?

A: the class name syntax is reserved for static methods! Java will think you're trying to call a static method, instead of the instance method of the parent class.

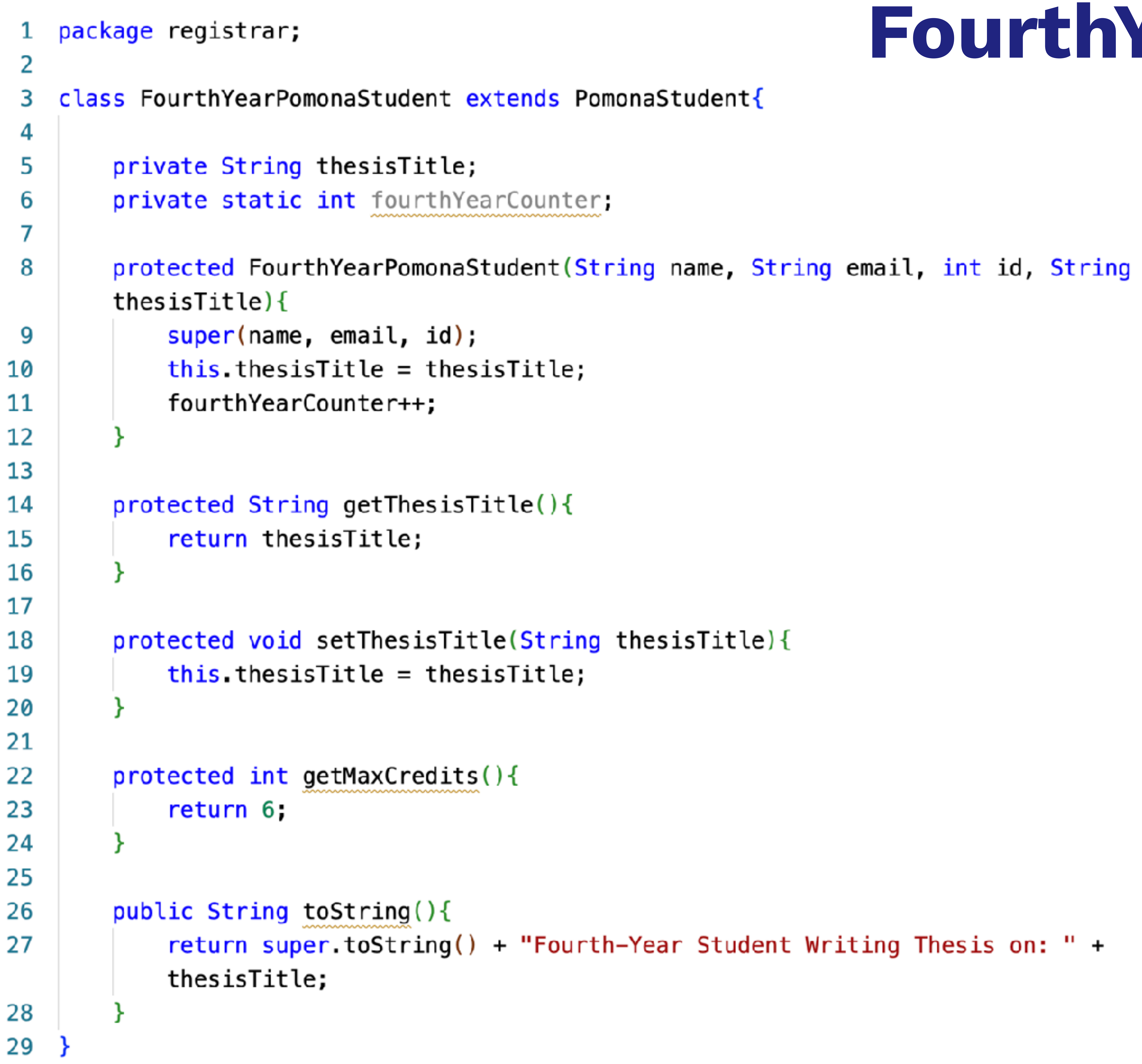
SecondYearPomonaStudent



```java
package registrar;

class SecondYearPomonaStudent extends PomonaStudent{
    private static int secondYearCounter;

    protected SecondYearPomonaStudent(String name, String email,
    int id){
        super(name, email, id);
        secondYearCounter++;
    }

    @Override
    protected int getMaxCredits(){
        return 6;
    }

    public String toString(){
        return super.toString() + "Second-Year Student can Take:
        " + getMaxCredits() +" credits";
    }
```

We have an @Override label to remind ourselves that getMaxCredits() was defined in the parent class: this is an *overridden instance method.*

FourthYearPomonaStudent



```java
package registrar;

class FourthYearPomonaStudent extends PomonaStudent{

    private String thesisTitle;
    private static int fourthYearCounter;

    protected FourthYearPomonaStudent(String name, String email, int id, String thesisTitle){
        super(name, email, id);
        this.thesisTitle = thesisTitle;
        fourthYearCounter++;
    }

    protected String getThesisTitle(){
        return thesisTitle;
    }

    protected void setThesisTitle(String thesisTitle){
        this.thesisTitle = thesisTitle;
    }

    protected int getMaxCredits(){
        return 6;
    }

    public String toString(){
        return super.toString() + "Fourth-Year Student Writing Thesis on: " + thesisTitle;
    }
}
```

Worksheet time!

Recall your Cat class. You also made a Dog class for the animal shelter, but realized there are lots of commonalities – name, sex, age, daysInRescue. Let’s make a parent class Animal that both Dog and Cat can extend. From your research, people who adopt cats care about their furType (short, long, etc.) and people who adopt dogs care about their breed (Corgi, Golden Retriever, etc.). Write 3 classes to represent this information. Be sure to:

- Put all the classes in an appropriate package
- Choose the right access modifiers for your fields and methods
- Have getter and setter methods for your instance variables
- Have a constructor (that takes all the relevant parameters) and a counter variable for each class
- Have a toString() method for each class, with Dog and Cat calling the Animal’s toString() before adding their own information.

Worksheet answers

https://github.com/pomonacs622025sp/code/tree/main/Lecture4/animalShelter

(We'll walk through on VSCode)

Polymorphism

Overriding methods

```
FirstYearPomonaStudent s1 = new FirstYearPomonaStudent("Daniel",
"daniel@pom.edu", 1, "War and Peace");
System.out.println(s1);
```

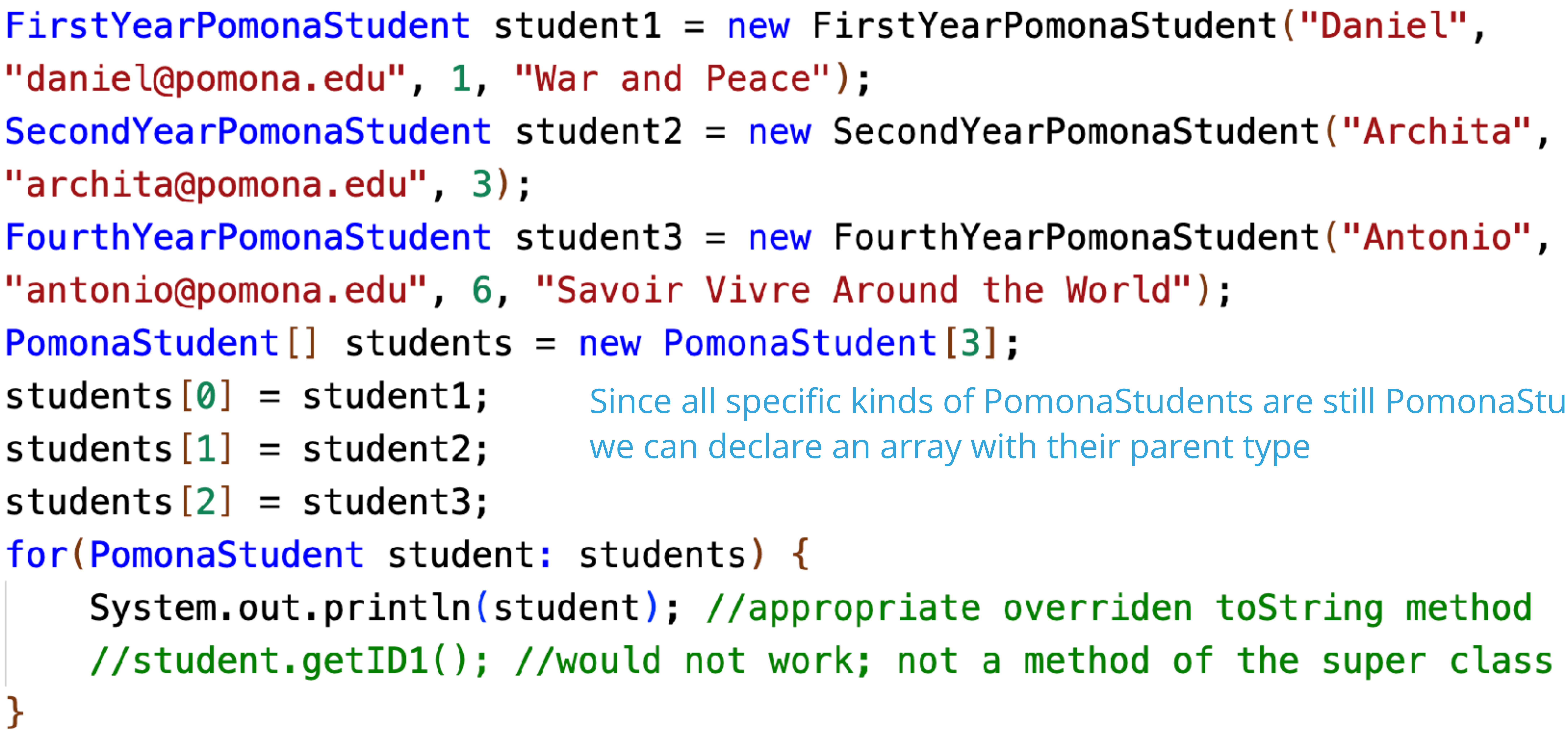
- Will print

```
Name: Daniel
email: daniel@pom.edu
id: 1
First-Year Student Attending ID1: War and Peace
```

Polymorphism

- Polymorphism means one object can take many forms: they can use instance variables and methods (public/protected/default) from many classes.
  - FirstYearPomonaStudents are still PomonaStudents are still Objects

Polymorphism



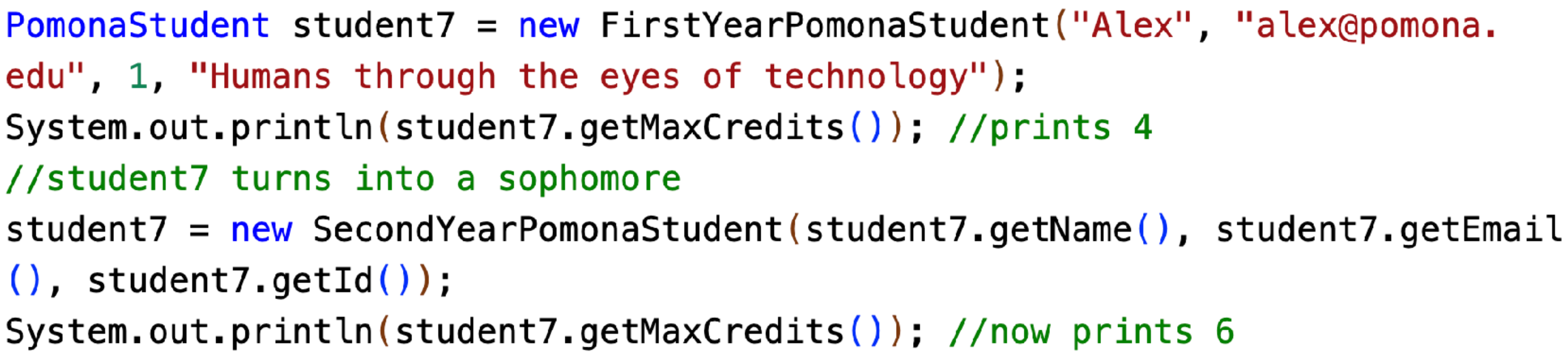
```java
FirstYearPomonaStudent student1 = new FirstYearPomonaStudent("Daniel",
"daniel@pomona.edu", 1, "War and Peace");
SecondYearPomonaStudent student2 = new SecondYearPomonaStudent("Archita",
"archita@pomona.edu", 3);
FourthYearPomonaStudent student3 = new FourthYearPomonaStudent("Antonio",
"antonio@pomona.edu", 6, "Savoir Vivre Around the World");
PomonaStudent[] students = new PomonaStudent[3];
students[0] = student1;
students[1] = student2;
students[2] = student3;
for(PomonaStudent student: students) {
    System.out.println(student); //appropriate overriden toString method
    //student.getID1(); //would not work; not a method of the super class
}
```

Since all specific kinds of PomonaStudents are still PomonaStudents, we can declare an array with their parent type

Polymorphism

For flexibly changing objects between child classes, use this syntax:

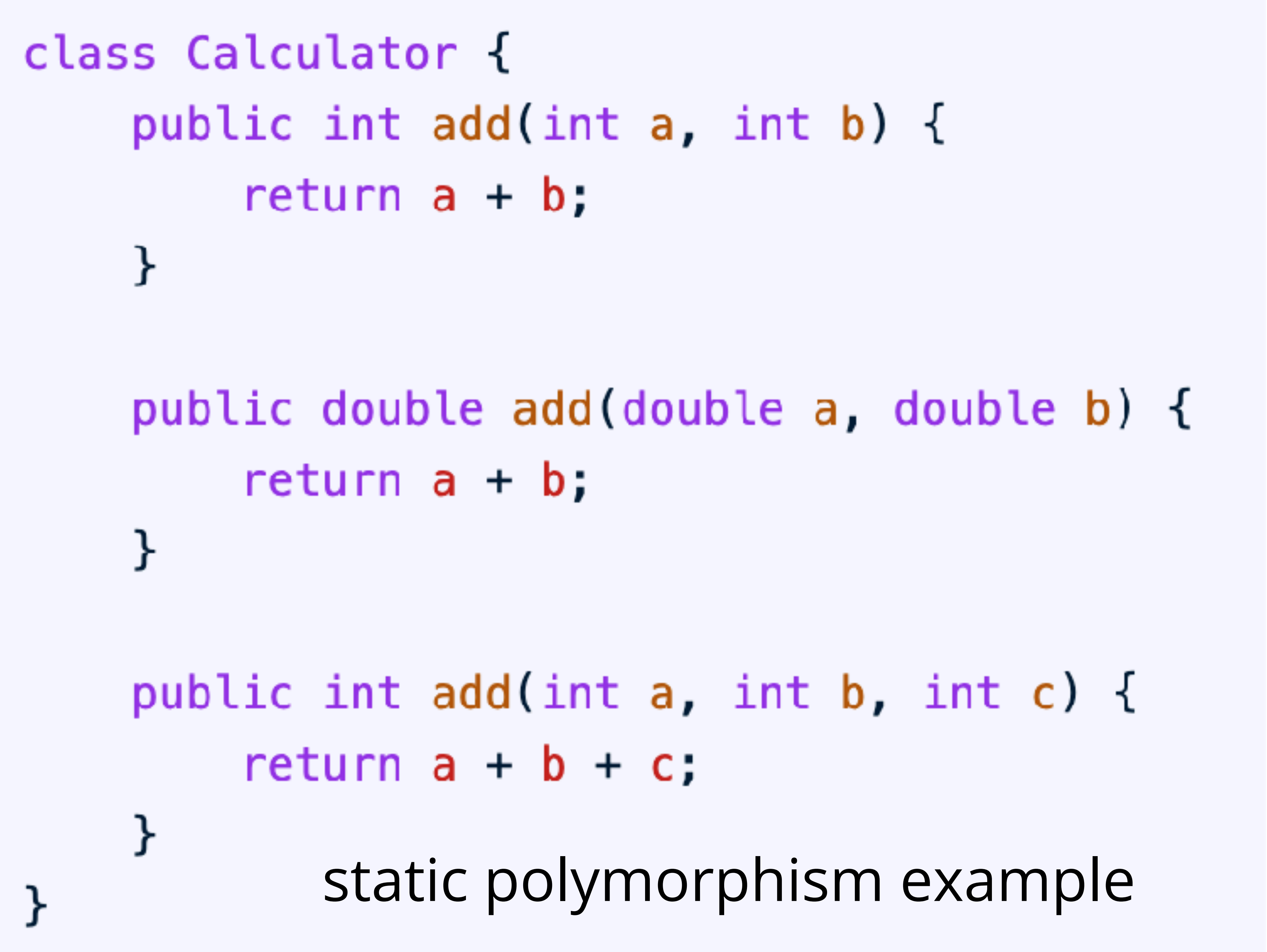
```
ParentClass obj = new ChildClass();
```



```
PomonaStudent student7 = new FirstYearPomonaStudent("Alex", "alex@pomona.
edu", 1, "Humans through the eyes of technology");
System.out.println(student7.getMaxCredits()); //prints 4
//student7 turns into a sophomore
student7 = new SecondYearPomonaStudent(student7.getName(), student7.getEmail
(), student7.getId());
System.out.println(student7.getMaxCredits()); //now prints 6
```

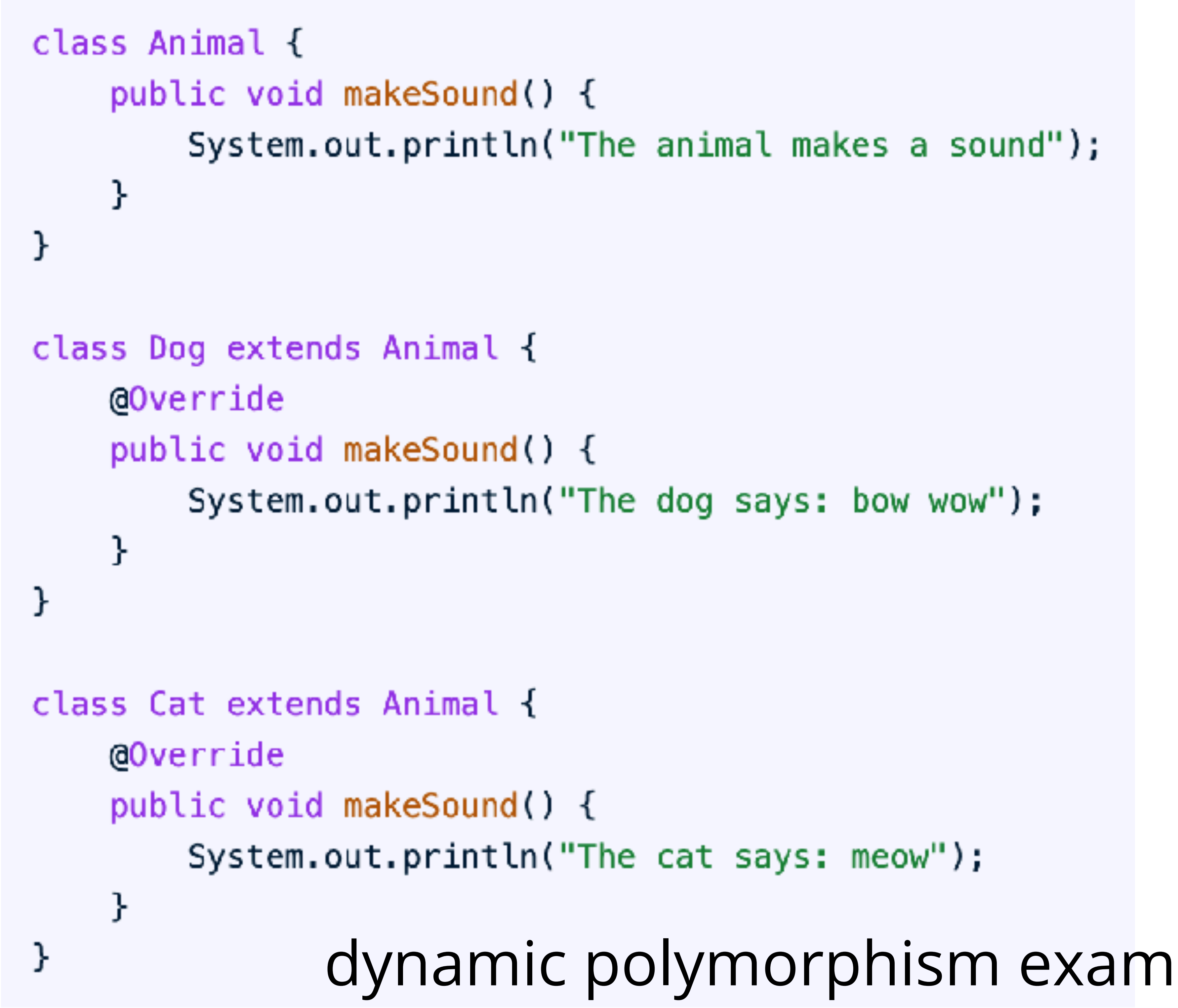
Overriding, dynamic vs static polymorphism

- Overriding: Instance methods in child classes override the instance methods in the parent classes (like .getMaxCredits()).
- This is called dynamic polymorphism, since it happens at runtime.
- In contrast to static polymorphism, which happens when we overload methods (such as having multiple constructors).



```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

static polymorphism example



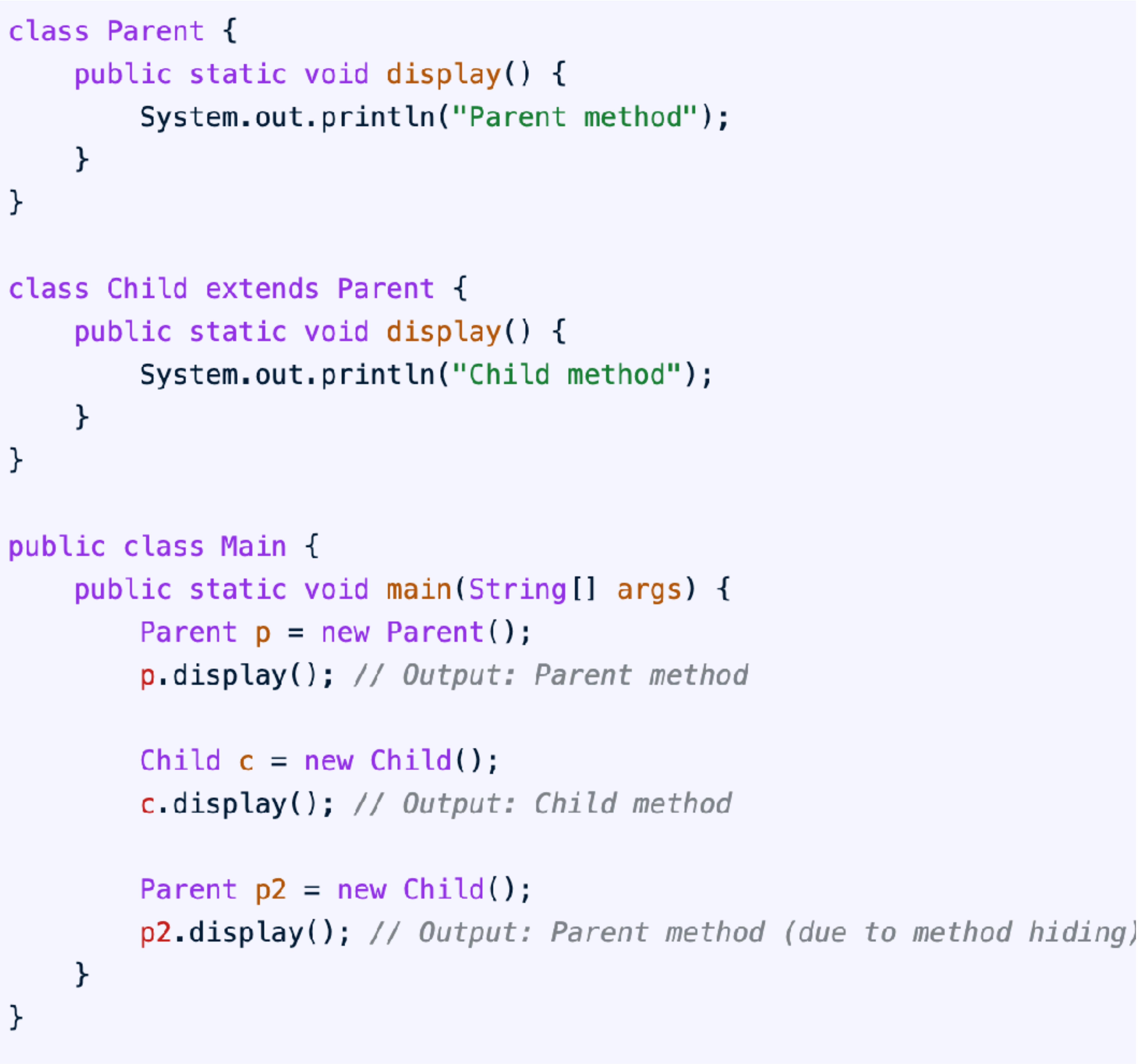
```
class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog says: bow wow");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The cat says: meow");
    }
}
```

dynamic polymorphism example

Method hiding

- Method hiding occurs when a subclass defines a static method with the same signature as a static method in its superclass.
- Unlike instance methods, which can be overridden, static methods are resolved at compile time based on the **class type** (the type on the left side), not the object type.
- Same thing happens with all variables: both static and instance.

Remember: static methods only, but all variables



```
class Parent {
    public static void display() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    public static void display() {
        System.out.println("Child method");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.display(); // Output: Parent method

        Child c = new Child();
        c.display(); // Output: Child method

        Parent p2 = new Child();
        p2.display(); // Output: Parent method (due to method hiding)
    }
}
```

Example: `Animal`

```java
public class Animal {
    public int legs = 2;
    public static String species = "Animal";
    public static void testStaticMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```

Example: Cat

```java
public class Cat extends Animal {
    public int legs = 4;
    public static String species = "Cat";
    public static void testStaticMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }
}
```

Hiding vs overriding

```
public static void main(String[] args) {
    Cat myCat = new Cat();
    myCat.testStaticMethod(); //invoking a hidden method
    myCat.testInstanceMethod(); //invoking an overridden method
    System.out.println(myCat.legs); //accessing a hidden field
    System.out.println(myCat.species); //accessing a hidden field
}
```

- Output:

```
The static method in Cat
The instance method in Cat
4
Cat
```

What we expected (hopefully).

Hiding vs overriding

```
public static void main(String[] args) {
    Animal yourCat = new Cat();
    yourCat.testStaticMethod(); //invoking a hidden method
    yourCat.testInstanceMethod(); //invoking an overridden method
    System.out.println(yourCat.legs); //accessing a hidden field
    System.out.println(yourCat.species); //accessing a hidden field
}
```

- Output:

```
The static method in Animal  Used the Animal method because of the Animal type
The instance method in Cat  Used the Cat method because it was overriden
2         Used the Animal instance variable because of the Animal type
Animal    Used the Animal static variable because of the Animal type
```

Worksheet time!

Recall your Cat class. You also made a Dog class for the animal shelter, but realized there are lots of commonalities – name, sex, age, daysInRescue. Let’s make a parent class Animal that both Dog and Cat can extend. From your research, people who adopt cats care about their furType (short, long, etc.) and people who adopt dogs care about their breed (Corgi, Golden Retriever, etc.). Write 3 classes to represent this information. Be sure to:

- Put all the classes in an appropriate package
- Choose the right access modifiers for your fields and methods
- Have getter and setter methods for your instance variables
- Have a constructor (that takes all the relevant parameters) and a counter variable for each class
- Have a toString() method for each class, with Dog and Cat calling the Animal’s toString() before adding their own information.

Worksheet time!

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}

public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

- 1. Which method *overrides* a method in the superclass?
- 2. Which method *hides* a method in the superclass?
- 3. What do the other methods do?

Worksheet answers

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}

public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

- 1. Which method *overrides* a method in the superclass?
  - methodTwo
- 2. Which method *hides* a method in the superclass?
  - methodFour
- 3. What do the other methods do?
  - Compile-time errors
  - methodOne: “This static method cannot hide the instance method from ClassA”.
  - methodThree: “This instance method cannot override the static method from ClassA”.

Lecture 4 wrap-up

- Exit ticket: https://forms.gle/q8MD8rQHBBLyMdNs5
- Reminder, do your quiz retakes in OH Tues or Weds next week (please bring your original quiz)
- HW2 due next Tuesday 11:59pm

Resources

- Memory management: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html
- Inheritance: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
- Extra practice: CS majors don't have to write a senior thesis (for now…). How would you edit FourthYearPomonaStudent to reflect this?