Revision Lecture

Programming Practice and Applications (4CCS1PPA)

Tomas Vitek
Thursday 8th December

programming@kcl.ac.uk
kcl.tomasvitek.com

TOPICS SO FAR COVERED

Due to the extensive syllabus covered, we cannot really revisit everything in this revision lecture.

- **Storing data**, variables, data types
- **Organising code**, methods
- **Object-orientation**, classes & objects
- **Control flow**, conditional statements, loops, recursion
- **Static vs non-static**
- **Library classes**, user input
- **Mathematical** operations
- **Arrays**
- **Null**
- **toString()** method
- More complex **data structures** in Java
- **Combining objects** – inheritance, polymorphism

TOPICS YOU WOULD LIKE TO REVISIT

How many of you would like to revisit each topic:

- **Storing data**, variables, data types **16%**
- **Organising code**, methods **13%**
- **Object-orientation**, classes & objects **25%**
- **Control flow**, conditional statements, loops, recursion **17%**
- **Static vs non-static** **61%**
- **Library classes**, user input **25%**
- **Mathematical operations** **23%**
- **Arrays** **48%**
- **Null** **41%**
- **toString()** method **39%**
- More complex **data structures** in Java **67%**
- **Combining objects** – inheritance, polymorphism **75%**

ADDITIONAL TOPICS YOU ASKED TO REVISIT

Some of you also asked to talk about:

- Encapsulation and final
- Two dimensional arrays
- Creation and ownership of objects
- Best practices on how to approach a problem
- More about complex data structures
- Real-life examples
- UML class diagrams

So much to revise...

Source: http://veterinariavistalegre.com/cachorros2

ASIDE: LAST RESORT TUTORIAL

Because we do not have too much time to revisit all these topics properly, I have decided to organise an additional **last resort tutorial** before next week's test:

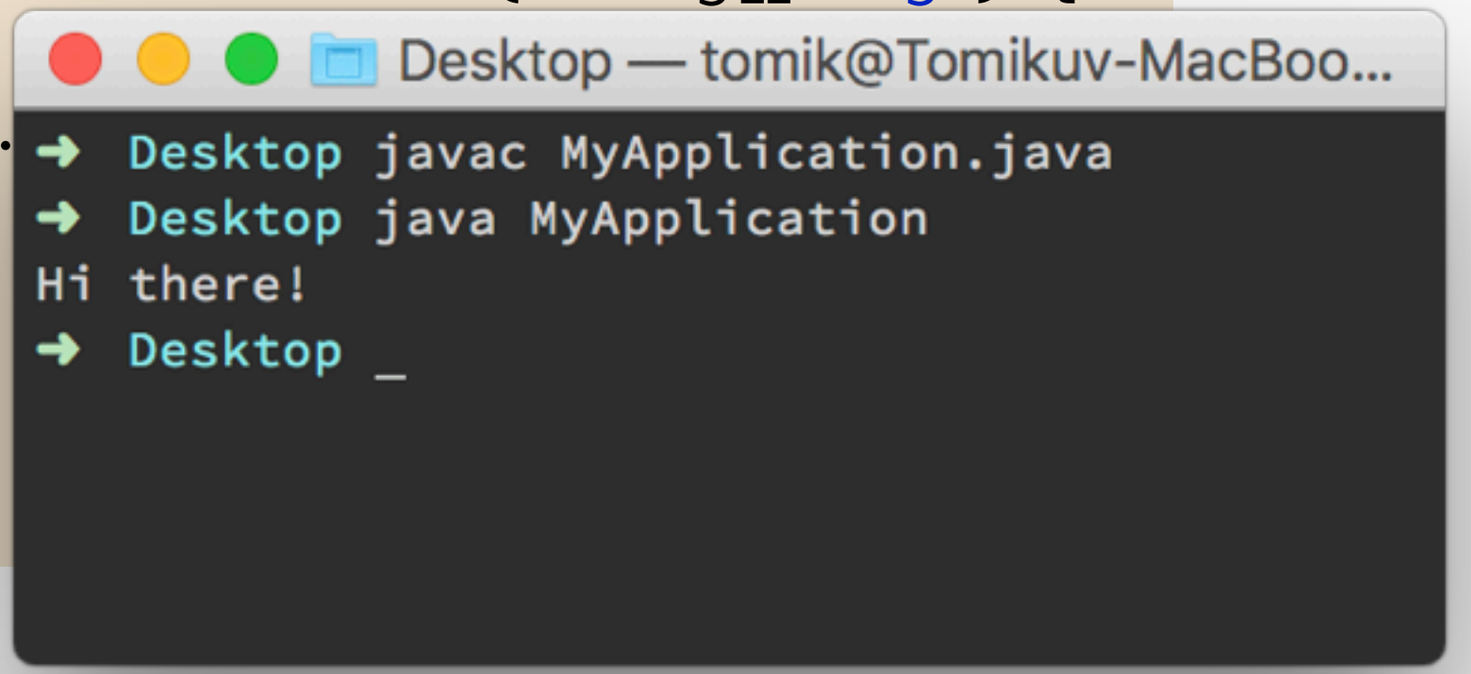
- Monday, **12th** December 2016
- **9am - 12noon**
- **Stamford Street Lecture Theatre**
  127 Stamford Street, London SE1 9NQ

OK, let’s do this!

BEFORE WE START...

All code in Java has to be inside a class. Each program has to be started by calling a main method on one of its classes:

```
public class MyApplication {

    public static void main(String[] args) {

        System.

    }

}
```



Any computer program does essentially two things:

- **Stores** data
- **Manipulates** data

STORING DATA — VARIABLES

In Java, like many other languages, data is stored in variables.

A variable

- is a **representation** of some data.
- has a **name** and a **data type**.
- usually has a **value**.
- (more generally) links to a storage location, e.g. computer memory.

STORING DATA — DATA TYPES

In Java, there are two different types of data:

- Primitive data types
    - **numbers**: int, long, float, double, ...
    - **textual**: byte, char
    - **booleans:** true or false
    - (void)
- Reference data types
    - links to more complex data types

```
int age = 26;
```

STORING DATA — PRIMITIVE DATA TYPES

```
int age = 26;
```

STORING DATA — PRIMITIVE DATA TYPES

- **data type**: int
- **name**: “age”
- **value**: 26

STORING DATA — REFERENCE DATA TYPES

```
String name = "Tomas";
```

STORING DATA — REFERENCE DATA TYPES

- **data type**: String
- **name**: “name”
- **value**: “Tomas"

STORING DATA — NULL

```
String name;
```

STORING DATA — NULL

- **data type**: String
- **name**: “name”
- **value**: ???

STORING DATA — NULL

- **data type**: String
- **name**: “name”
- **value**: null

STORING DATA — NULL

```
public Person findTomas() {

    // finding...

    if (found) return tomas;
    else return null;
}
```

```
Person tomas = findTomas();
if (tomas == null) {
   System.out.println("Sorry, no Tomas around!");
}
```

Sometimes we need to store **more values than just one**. We could create multiple variables, but this is impractical if dealing with many values of the same type.

For this we use arrays. **An array** is a special type of variable, which can hold up to a predefined number of values.

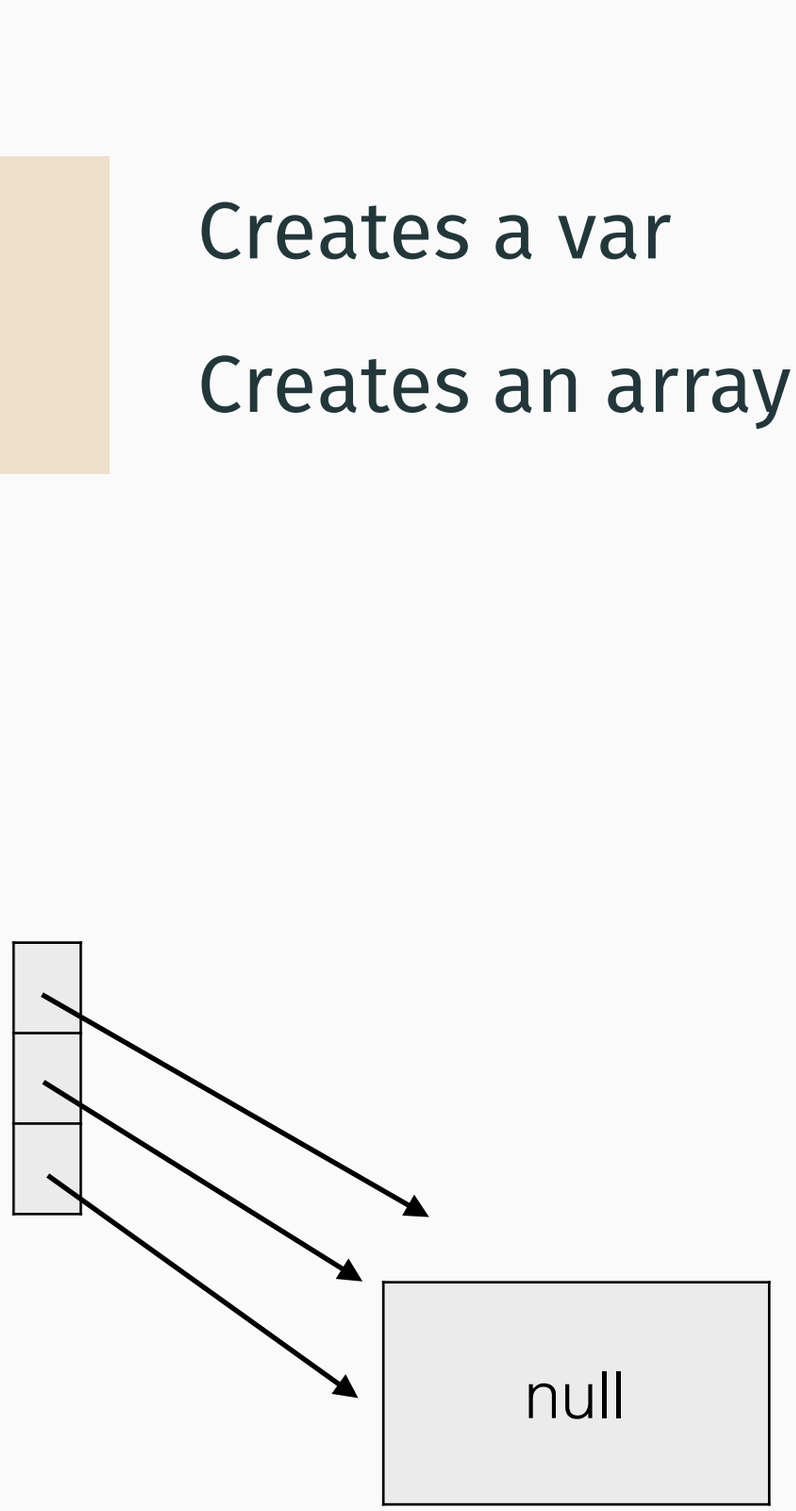
STORING DATA — ARRAYS

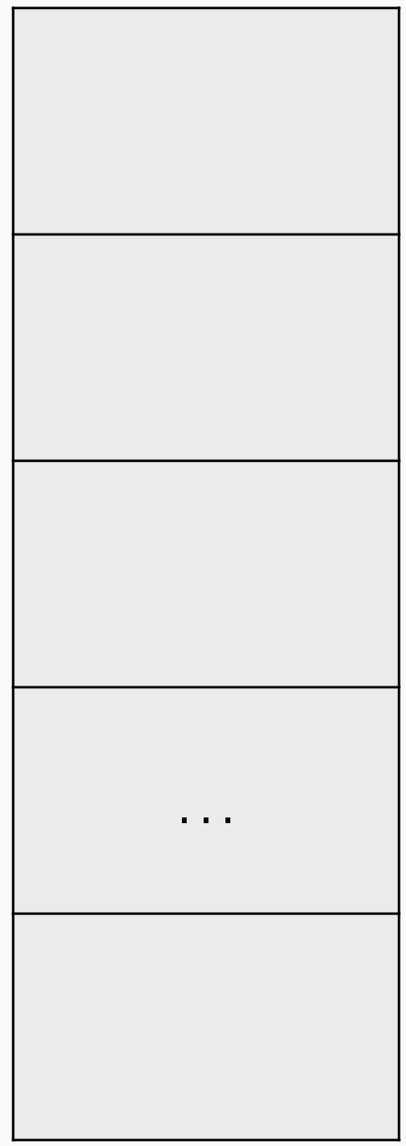
```
String[] names;

names = String[3];
```

Creates a var

Creates an array

names





STORING DATA — ARRAYS

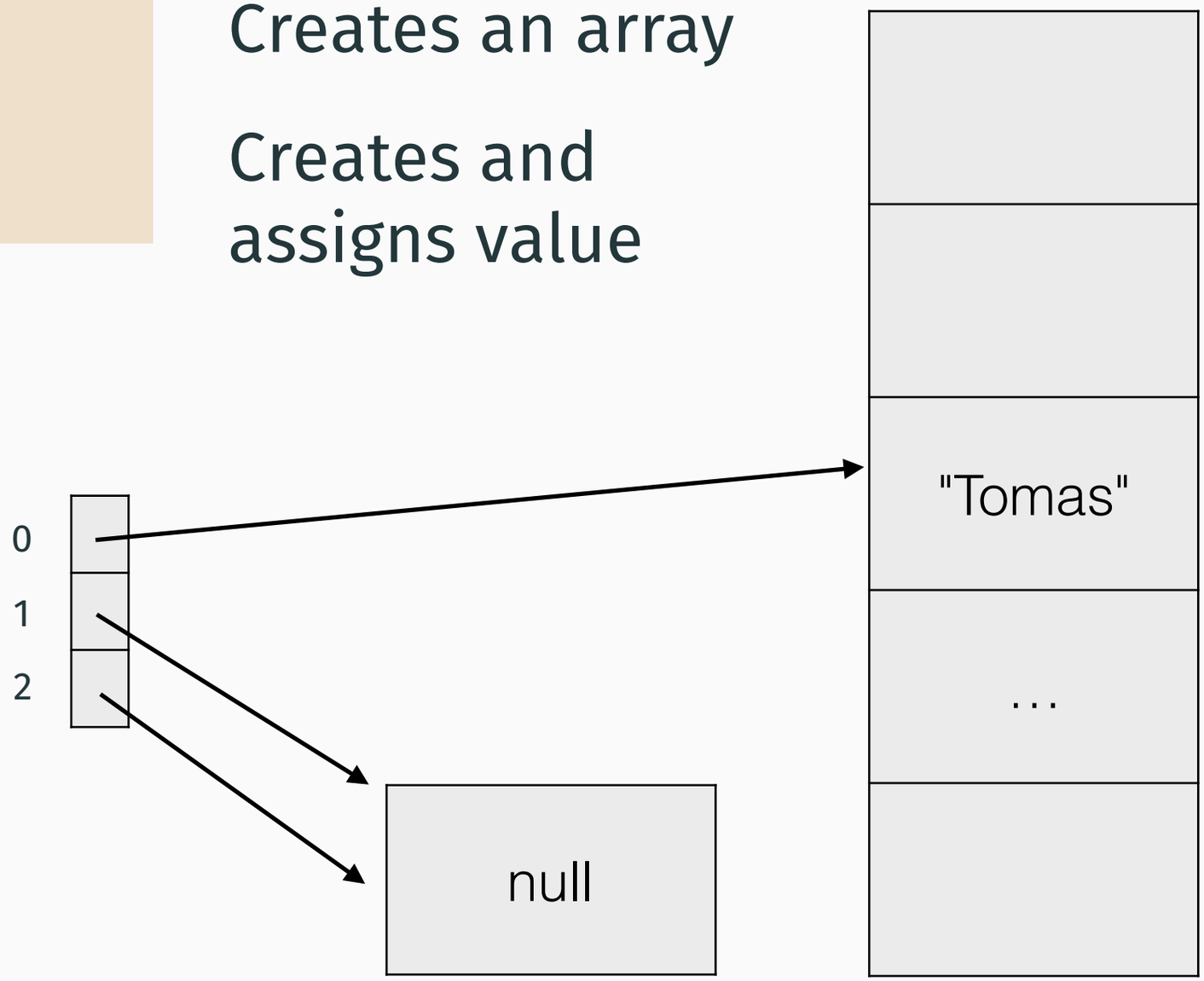
```
String[] names;

names = String[3];

names[0] = "Tomas";
```

Creates a var

Creates an array

Creates and assigns value

names



STORING DATA — ARRAYS

```
String[] names;
names = String[3];

names[0] = "Tomas";

String lecturer = "Martin";
names[2] = lecturer;
```

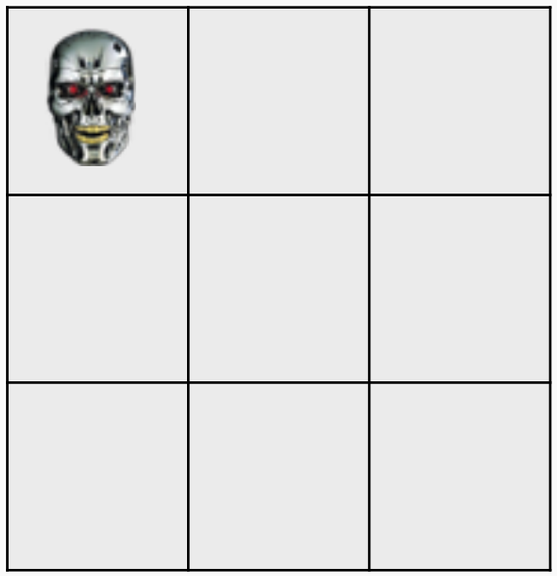
We can store both **primitive** and **reference data types** in arrays. Usually we only store one data type in an array, to keep data consistent.

But there is a way to store any reference type:

```
Object[] stuff;
```

If we want to store **multi-dimensional data** (such as a grid representation of a robot simulation), we can use arrays with multiple dimensions:

```
Robot[][] map = new Robot[3][3];
map[0][2] = terminator;
```



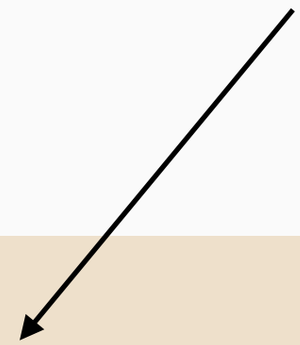
CONTROL FLOW — CONDITIONAL STATEMENTS

any expression evaluating as boolean (`true` or `false`)

```
if (           ) {
   System.out.println("Hi there!");
}
```

CONTROL FLOW — CONDITIONAL STATEMENTS

boolean variable



```
if (isTomasAround) {
   System.out.println("Hi, Tomas!");
}
```

CONTROL FLOW — CONDITIONAL STATEMENTS

boolean expression

```
if (age < 99) {
   System.out.println("Not dead yet!");
}
```

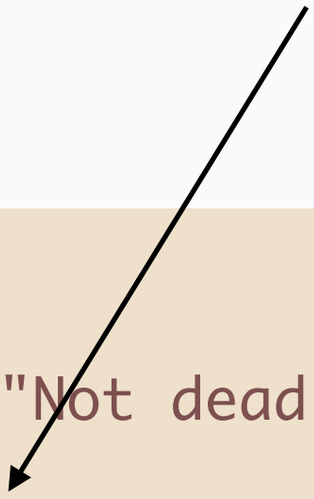
CONTROL FLOW — CONDITIONAL STATEMENTS

```
if (age < 99) {
   System.out.println("Not dead yet!");
}
else {
   System.out.println("Probably dead!");
}
```

CONTROL FLOW — CONDITIONAL STATEMENTS

|| → OR

&& → AND



```
if (age < 99) {
   System.out.println("Not dead yet!");
}
else if (age >= 99 && age < 130) {
   System.out.println("Probably dead!");
}
else {
   System.out.println("Definitely dead!");
}
```

There are other relational operators available to us to make numeric comparisons, and thus generate boolean values.

Java	Mathematical Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

Always use **equals** method for comparing Strings!

```
if (name.equals("Tomas")) {
    System.out.println("Hi, Tomas!");
}
```

When you want to repeat something multiple times, you need a loop!

For when you know how many times you want to repeat the loop, you use a **for loop** (*pun intended*):

```
for (int i = 0; i < 3; i++) {
   System.out.println("That’s a bad pun!");
}
```

When you want to repeat something multiple times,
you need a loop!

For when you know how many times you want to repeat the loop, you use a **for loop** (*pun intended*):

```
for (int i = 0; i < array.length; i++) {
   System.out.println(array[i]);
}
```

While **for loop** works when you know how many times you want to repeat the loop, sometimes you don't know, so we use a **while loop** instead (*pun still intended*):

```
while (stillNotFunny) {
   System.out.println("Even worse pun!");
}
```

CONTROL FLOW — LOOPS

```
while (stillNotFunny) {
    System.out.println("Even worse pun!");
    if (enough) {
        break;
    }
}
```

MATHEMATICAL OPERATIONS

We have other operators available to us, but their formation differs from regular mathematics:

Java	Mathematical Notation	Description
+	+	Addition
-	-	Subtraction
/	÷	Division
*	×	Multiplication
%	mod	Modulo

MATHEMATICAL OPERATIONS

```
6 * 5 + 12 * 3 + 8 / 4 = ???
```

MATHEMATICAL OPERATIONS

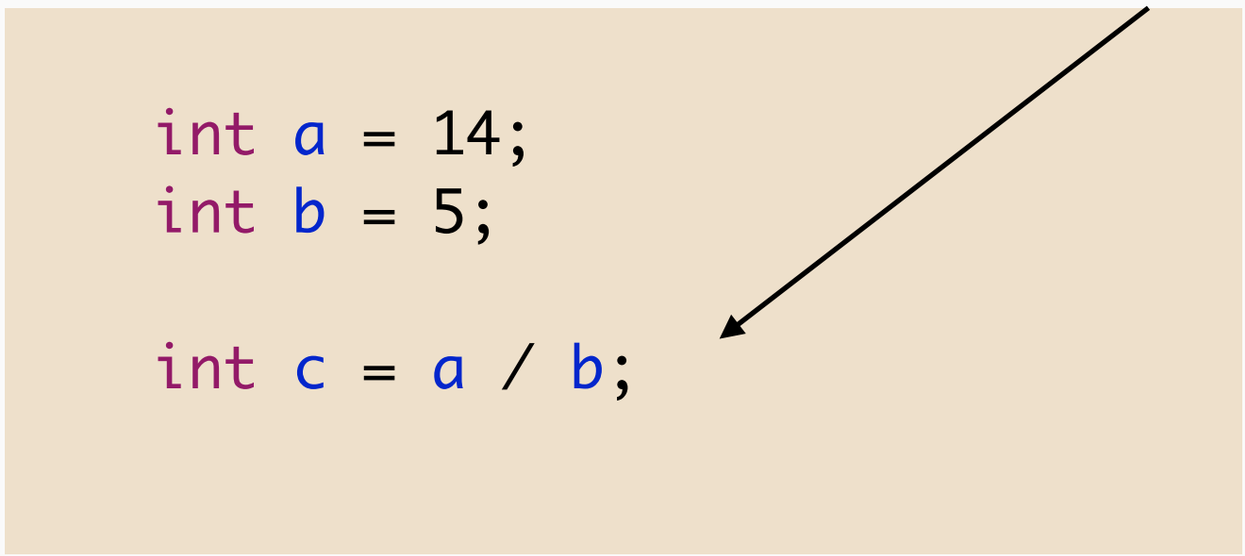
```
(6 * 5) + (12 * 3) + (8 / 4) = 68
```

MATHEMATICAL OPERATIONS

Precision of the result will depend on the precision of the operands.

E.g. if you divide two integers (whole numbers), a result will always be an integer

2



```
int a = 14;
int b = 5;

int c = a / b;
```

MATHEMATICAL OPERATIONS

Precision of the result will depend on the precision of the operands.

E.g. if you divide two integers (whole numbers), a result will always be an integer

```
int a = 14;
int b = 5;

double c = (a + 0.0) / b;
```

2.8

ORGANISING CODE — METHODS

```
public class MyApplication {

   public static void main(String[] args) {
      String name = "Tomas";
      System.out.println("Hi there, " + name + "!");
      System.out.println("Hi there, " + name + "!");
      System.out.println("Hi there, " + name + "!");
   }

}
```

MyApplication.java

When you keep **repeating** the **same** or **similar code** to do a similar thing, instead of copy-pasting (which is a lot of work, requires more maintenance and is more difficult to read),

you should create a **method** and call it.

ORGANISING CODE — METHODS

```
public class MyApplication {

   private void sayHi(String toWhom) {
      System.out.println("Hi there, " + toWhom + "!");
   }

   public static void main(String[] args) {
      String name = "Tomas";
      sayHi(name);
      sayHi(name);
      sayHi(name);
      sayHi(name);
   }

}
```

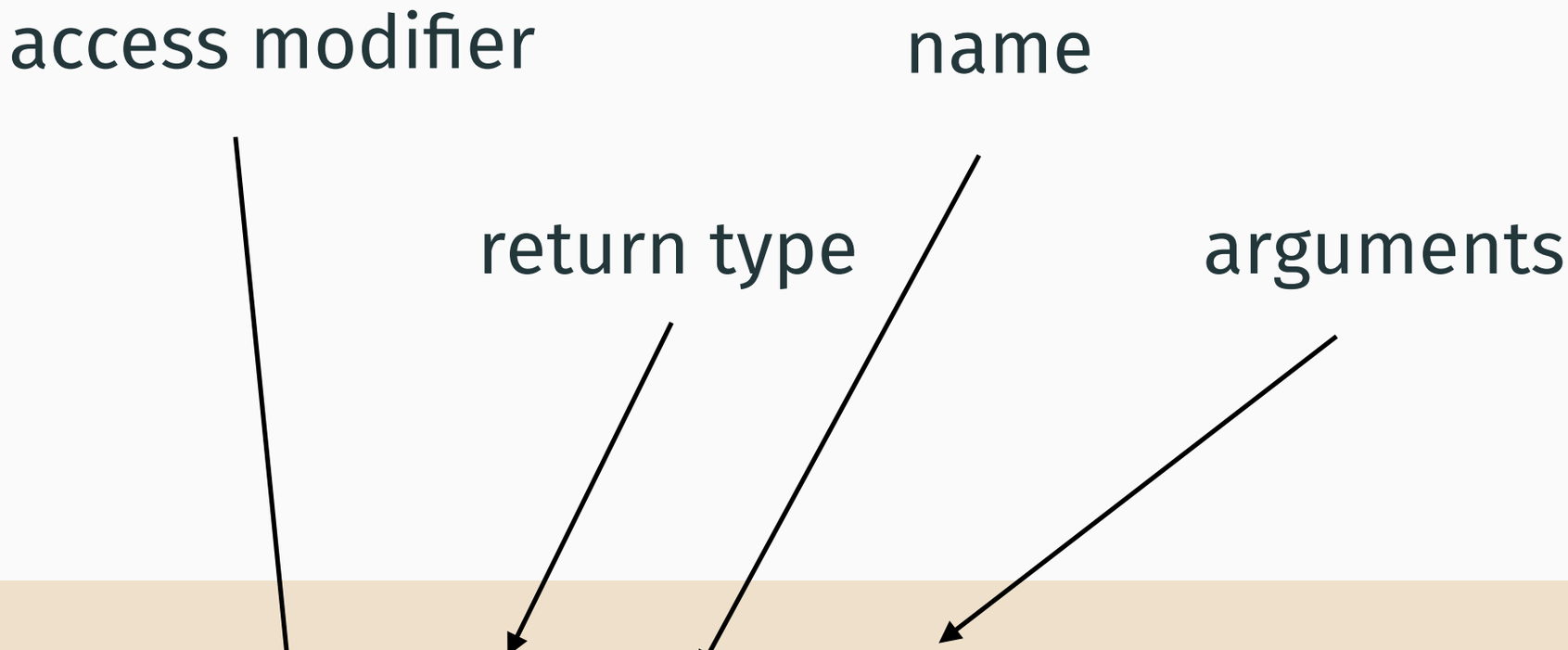
MyApplication.java

ORGANISING CODE — METHODS

```
public class MyApplication {

   private void sayHi(String toWhom) {
      System.out.println("Hi there, " + toWhom + "!");
   }

   public static void main(String[] args) {
      String name = "Tomas";

      for (int i = 0; i < 4; i++) {
         sayHi(name);
      }
   }

}
```

MyApplication.java

ORGANISING CODE — METHODS




```
private void sayHi(String toWhom) {
   System.out.println("Hi there, " + toWhom + "!");
}
```

OBJECT-ORIENTATION; CLASSES AND OBJECTS

Java is an **object-oriented programming** (OOP) language. This means that all code is organised around the concept of **classes** and **objects**.

A class is a "*blueprint*" for objects. You can create many objects from one class.

OBJECT-ORIENTATION; CLASSES AND OBJECTS

```java
public class House {

   int numberOfWindows;
   ...

}
```

House.java

```java
public class MyApplication {

   public static void main(String[] args) {
      House h1 = new House();
      House h2 = new House();
      House h3 = new House();
   }

}
```

MyApplication.java

Sometimes we need to **setup a newly created object**, we use constructors for this. A **constructor** is a special method called only once when you create an object of the given class.

constructor

field

```
public class House {

   int numberOfWindows;

   public House(int numberOfWindows) {
      this.numberOfWindows = numberOfWindows;
   }

}
```

House.java

```java
public class MyApplication {

    public static void main(String[] args) {
        House h1 = new House(4);
        House h2 = new House(10);
        House h3 = new House(0);
    }

}
```

MyApplication.java

Encapsulation is a OOP concept that **binds data and their methods** together and **shields** them from the outside world.

In other words it means that we should keep data and methods, which logically relate to each other together in one class. And we should take care that nobody else but the object should modify its variables representing state (fields).

```
House martinsHouse = askWhereMartinLives();
martinsHouse.numberOfWindows = 0;
```

MyApplication.java

```java
public class House {

   private int numberOfWindows;

   public House(int numberOfWindows) {
      this.numberOfWindows = numberOfWindows;
   }

}
```

House.java

Each field and method (class member) in every class is either static or non-static.

A static member is a special kind of a class member, which is **shared** among all instances (copies) of its class.

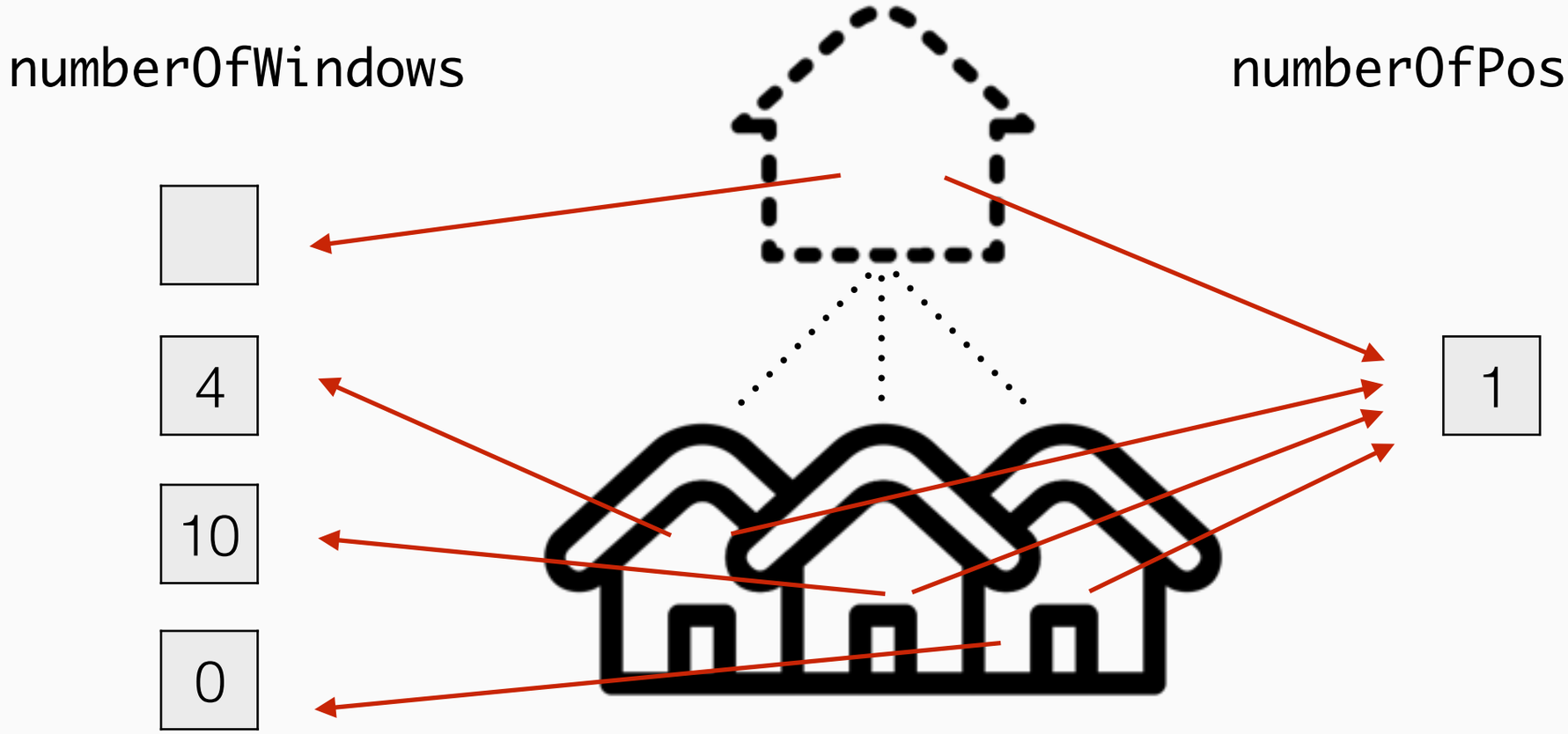
In contrast, a **non-static class member** is created for each instance **separately**.

By **default** all fields and methods are **non-static**. This means that the given field or method is not shared by all instances.

OBJECT-ORIENTATION — STATIC VS. NON-STATIC

```java
public class House {

   private static int numberOfPostboxes = 1;

   private int numberOfWindows;

}
```

House.java



If a variable (or a method) is created **final**, it means that once it is assigned a value, it **cannot change** it.

A class member (a field or a method) can be final and static, or final and non-static.

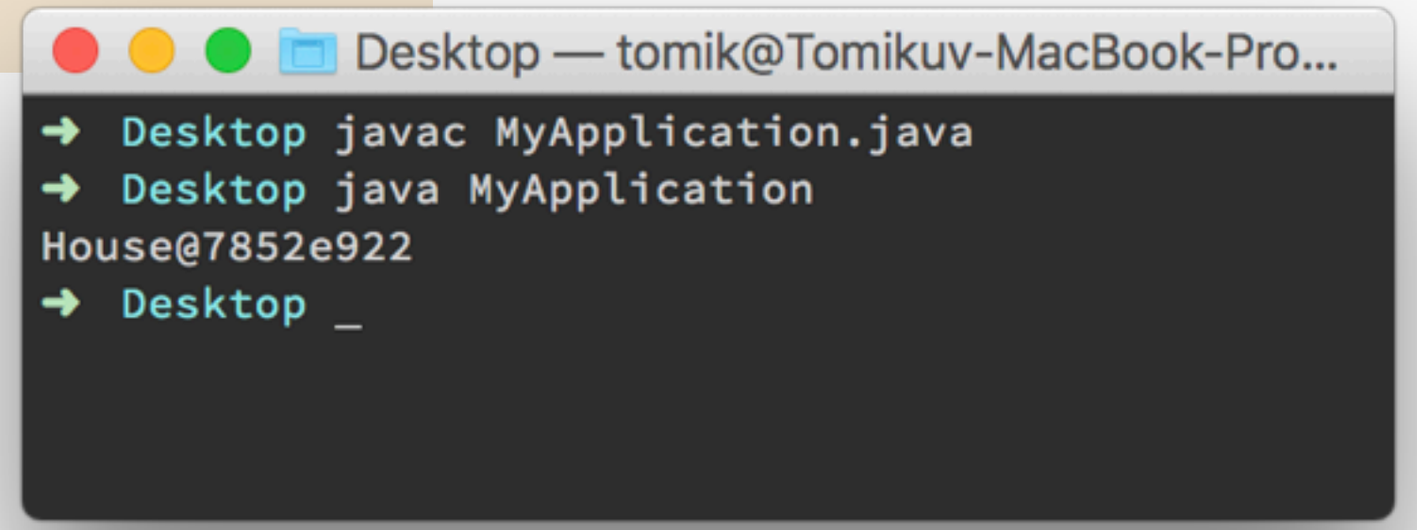
```
public class House {

   public static final int numberOfPostboxes = 1;

   private int numberOfWindows;

}
```

House.java

CONVERTING OBJECTS TO STRING

How to fix this?

```
House myHouse = new House(2);
System.out.println(myHouse);
```



CONVERTING OBJECTS TO STRING

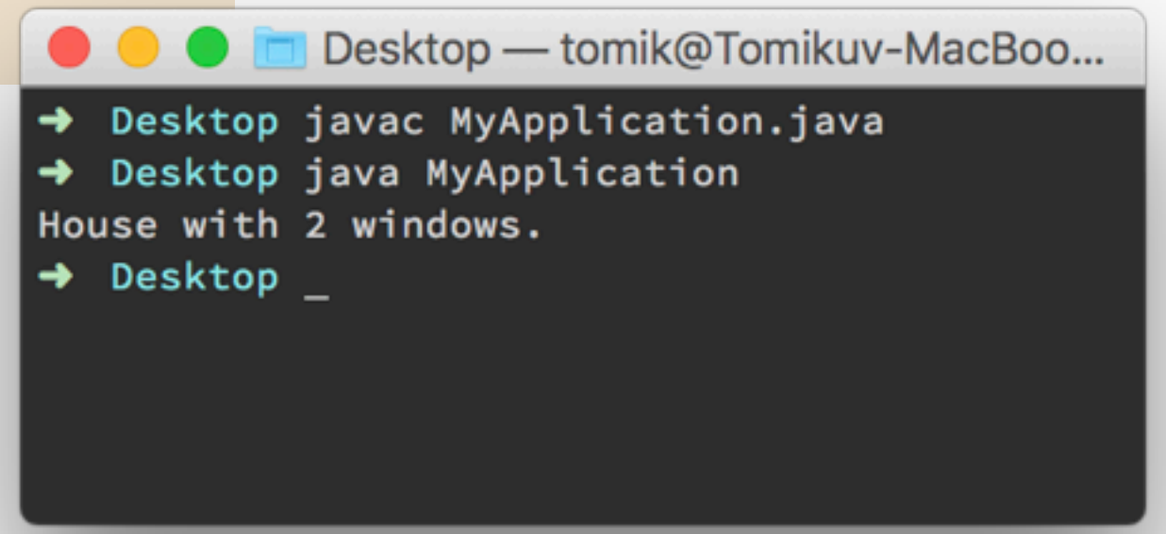
```
public class House {

   private int numberOfWindows;

   public House(int numberOfWindows) {
      this.numberOfWindows = numberOfWindows;
   }

   public String toString() {
      return "House with " + numberOfWindows +
             " windows.";
   }

}
```

House.java

CONVERTING OBJECTS TO STRING

Better!

```
House myHouse = new House(2);
System.out.println(myHouse);
```



LIBRARY CLASSES

If you use a class located in the same directory as the file you are calling it from in your application, Java will automatically load it for you.

Java is a quite powerful programming language also because it has **many available libraries already built for it**. This includes both **standard libraries** (included in the language) and custom libraries available as open-source for example.

If you want to us a library, you need to **import** it, which you should do at the top of your file.

```
import java.util.Scanner;
```

LIBRARY CLASSES

You don’t have to import a class to use it, but them you have to use its full name (including what we call a package):

```
java.util.Scanner scanner = new
                            java.util.Scanner(System.in);
```

LIBRARY CLASSES

Most useful standard libraries you should know for now:

- java.util.**Scanner**
- java.util.**Random**
- **Math**

Check the documentation for those:

- https://docs.oracle.com/javase/8/docs/api/

LIBRARY CLASSES — USER INPUT

When you need to ask a user for some data input in the terminal, use **Scanner**:

```
import java.util.Scanner;

...

Scanner scanner = new Scanner(System.in);

System.out.println("Enter your name: ");
String name = scanner.nextLine();

System.out.println("Enter your age: ");
int age = scanner.nextInt();
```

COMPLEX DATA STRUCTURES IN JAVA

Arrays are very useful for storing data, but sometimes it is difficult to use them due to their limitations — such as fixed length, which needs to be defined before use.

The Java standard libraries contain several very useful data structures, which enable storage of more complex data:

- **ArrayList**
- **TreeSet**
- **TreeMap**

} all of these are **Collections**

- and many others...

Check the documentation for those:

- https://docs.oracle.com/javase/8/docs/api/

ArrayLists, as the name suggests, are lists based on arrays. This means that you don't have to know the max. length before you start using one. If you fill the **ArrayList**, Java will take care of making more space for your data.

However like with arrays, you also have to define what kind of data do you want to store in your **ArrayLists**.

For **ArrayLists** and other **Collections**, we do this using **Generics** (don't worry, example follows).

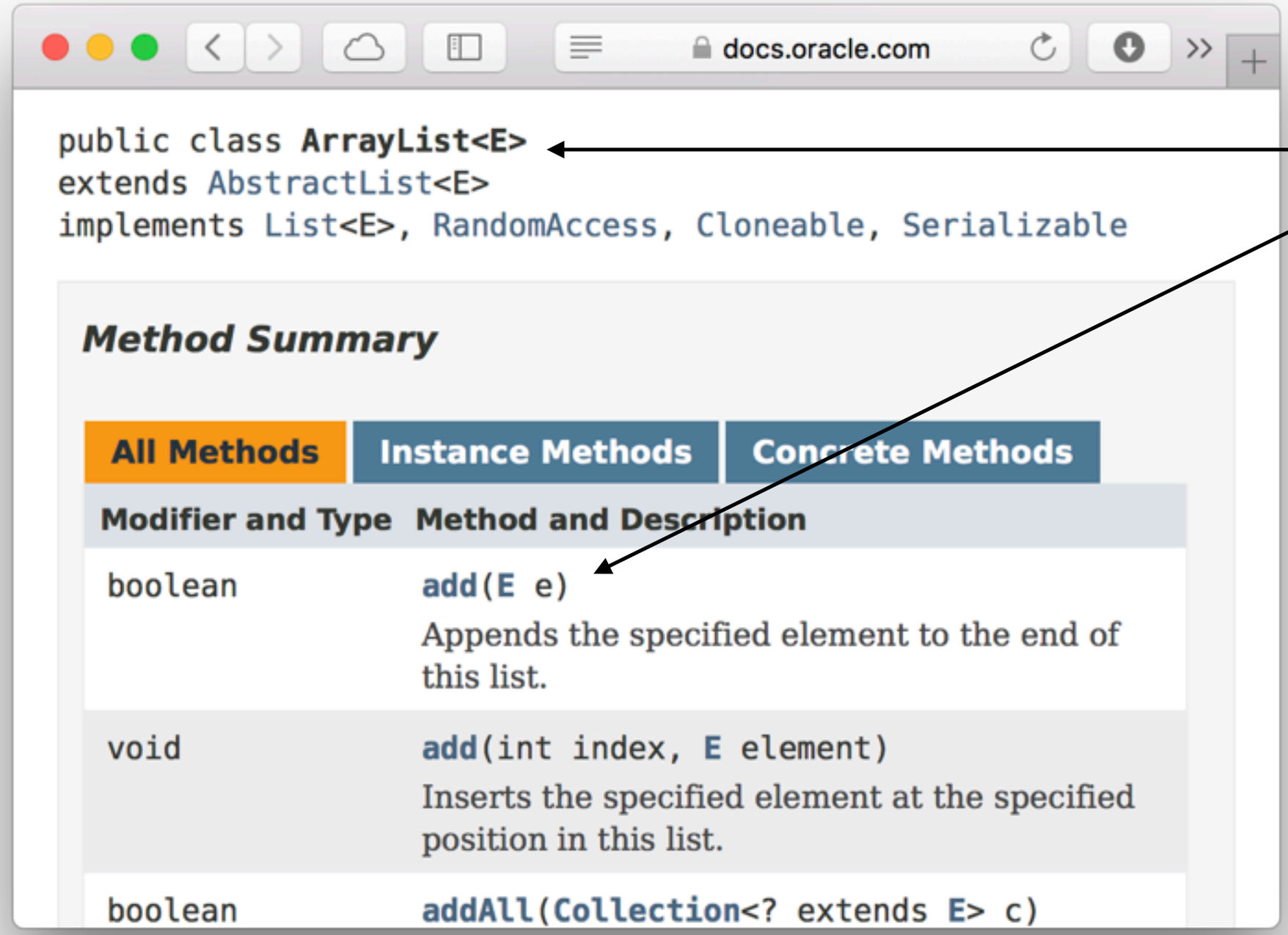
COMPLEX DATA STRUCTURES IN JAVA — ARRAYLIST

Generics

```
import java.util.ArrayList;

...

ArrayList<String> peopleHere = new ArrayList<String>();
peopleHere.add("Martin");
peopleHere.add("Asad");
peopleHere.add("Tomas");

String name = peopleHere.get(2);
```

http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

COMPLEX DATA STRUCTURES IN JAVA — GENERICS



Generics

http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

COMPLEX DATA STRUCTURES IN JAVA — ARRAYLIST

```
import java.util.ArrayList;

...

if (peopleHere.contains("Tomas")) {
    System.out.println("Tomas is here!");
}
```

http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

COMPLEX DATA STRUCTURES IN JAVA — TREESET

In set, no two value can be the same.

Generics

```
import java.util.TreeSet;

...

TreeSet<String> peopleHere = new TreeSet<String>();
peopleHere.add(“Martin”);
peopleHere.add(“Asad”);
peopleHere.add(“Tomas”);
peopleHere.add(“Martin”);

int howManyPeople = peopleHere.size();
```

3

https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

COMPLEX DATA STRUCTURES IN JAVA — TREEMAP

A map maps (😏) from one value to another.

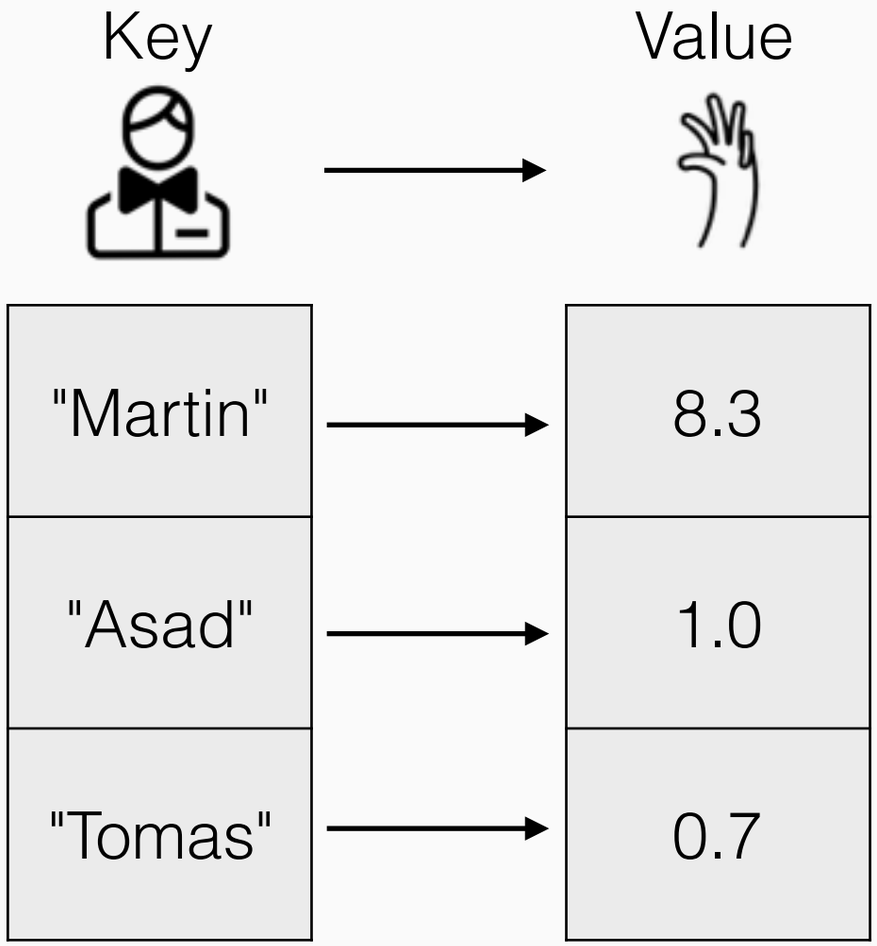
Generics

```
import java.util.TreeMap;

...

TreeMap<String, Double> lecturesGiven =
                                new TreeMap<String, Double>();
lecturesGiven.add("Martin", 8.3);
lecturesGiven.add("Asad", 1);
lecturesGiven.add("Tomas", 0.7);

double count = lecturesGiven.get("Tomas");
```

0.7

https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

COMPLEX DATA STRUCTURES IN JAVA — TREEMAP



0.7

```
double count = lecturesGiven.get("Tomas");
```

https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

When defining fields for a class (or in other words defining what kind of data will the objects created from this class store), we can use not only primitive data types, but also reference types.

In such case the object stores references to other objects in its fields.

COMBINING OBJECTS

the reference is assigned a value

(of whatever is passed to the constructor)

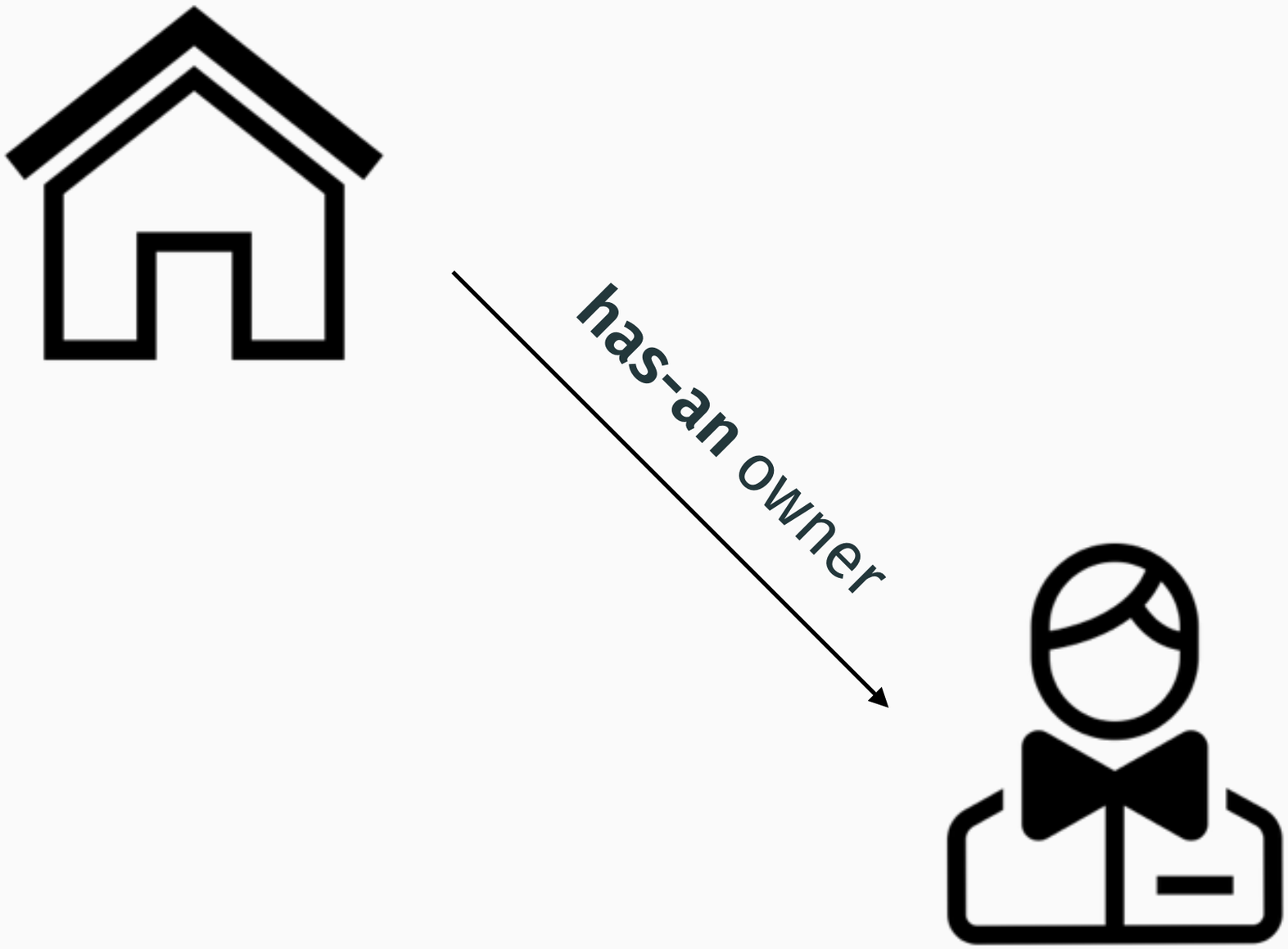
```
public class House {

   private Person owner;
   private int numberOfBedrooms = 1;

   public House(Person owner) {
      this.owner = owner;
   }

   public int getNumberOfBedrooms() {
      return numberOfBedrooms;
   }

}
```

House.java

reference to other object

(**has-a** relationship)

COMBINING OBJECTS



But sometime the **has-a** relationship does not realistically model how things are in the real world.

Let's say we have two real life objects we want model in our application: a house and a villa.

It's obvious that they don't have a **has-a** relationship to another. We could model them as two totally separate classes, but that's also not great as they share some common traits.

What we want is a **is-a** relationship. We can do this in Java using inheritance. A villa is a special kind of house, hence their relationship will be *"a villa is a house*".

COMBINING OBJECTS — INHERITANCE

```
public class Villa extends House {

   private int numberOfBedrooms = 5;
   private int numberOfHeliport = 1;

   public Villa(Person owner) {
      super(owner);
   }

   public int getNumberOfHeliports() {
      return numberOfHeliport;
   }
}
```

Villa.java

inheritance from House (**is-a** relationship)

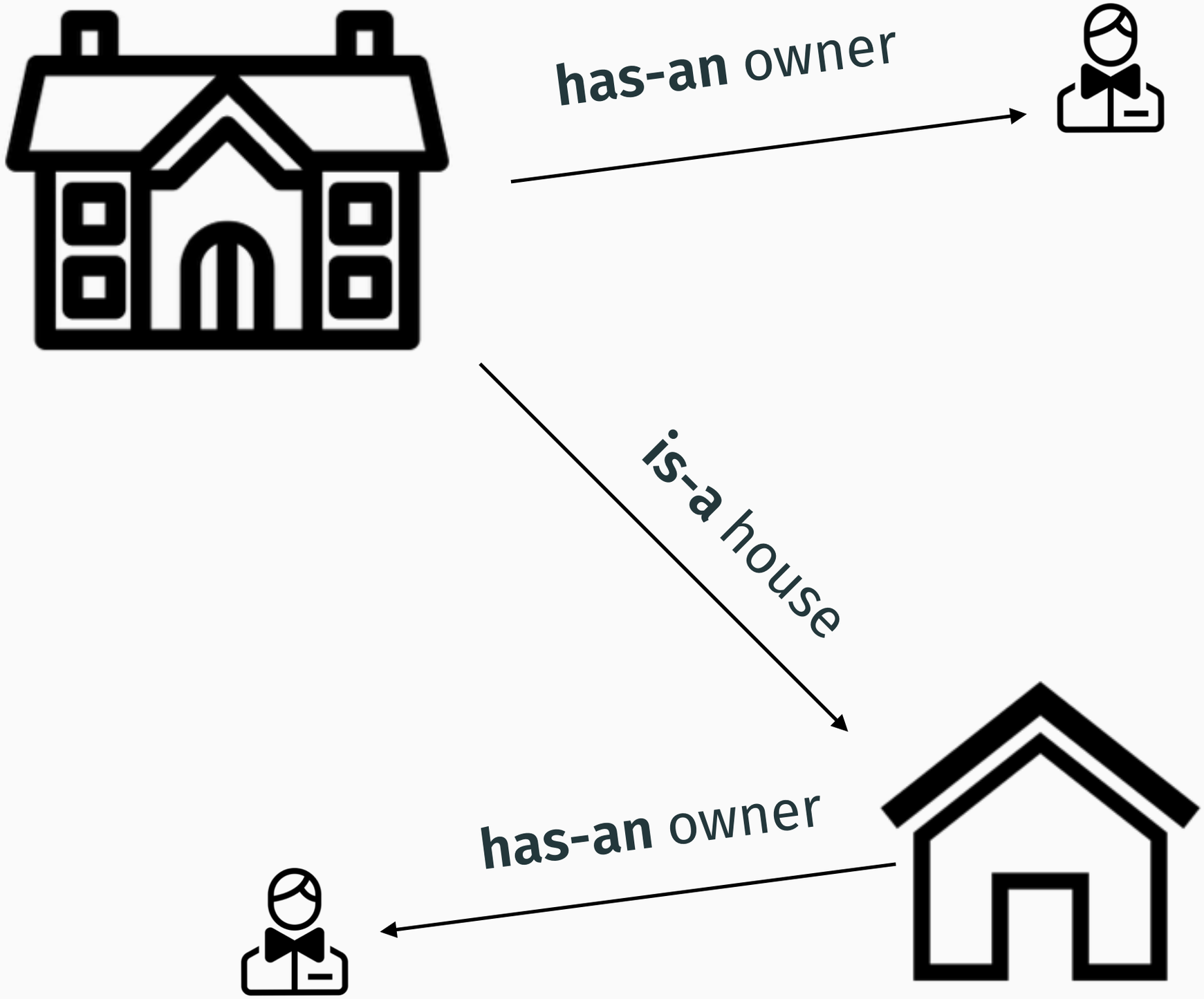
overrides what is defined in House

adds new stuff

calls its parent's constructor

still has a reference to other object (**has-a** relationship)

COMBINING OBJECTS — INHERITANCE



Because a **Villa** is a special kind of a **House**, which inherits all of the **House**'s traits (but can override some or add some new), we can refer to **Villa** objects as **House** objects.

But not the other way around!!

COMBINING OBJECTS — POLYMORPHISM

prints 1

prints 5

error!

```
House house = new House(new Person("Tomas"));
System.out.println(house.getNumberOfBedrooms());

House house = new Villa(new Person("Martin"));
System.out.println(house.getNumberOfBedrooms());

House house = new Villa(new Person("Martin"));
System.out.println(house.getNumberOfHeliports());

Villa house = new Villa(new Person("Martin"));
System.out.println(house.getNumberOfHeliports());
```

prints 1

To recap

Source: http://veterinariavistalegre.com/cachorros2

LAST RESORT TUTORIAL

Because we do not have too much time to revisit all these topics properly, I have decided to organise an additional last resort tutorial before next week's test:

- Monday, **12th** December 2016
- **9am - 12noon**
- **Stamford Street Lecture Theatre**
  127 Stamford Street, London SE1 9NQ

Thanks...

🙃

… and good luck!

👍🍀

Revision Lecture

Programming Practice and Applications (4CCS1PPA)

Tomas Vitek
Thursday 8th December

programming@kcl.ac.uk
kcl.tomasvitek.com

These slides will be available on KEATS, but will be subject to ongoing amendments. Therefore, please always download a new version of these slides when approaching an assessed piece of work, or when preparing for a written assessment.