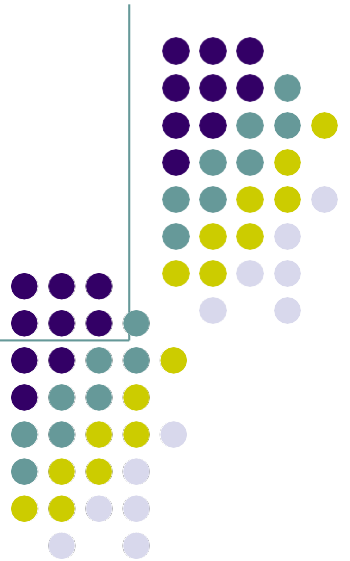


OOP

Ionut
Spalatelu

OOP

1. OOP Introduction
2. Abstraction
3. Encapsulation
4. Composition
5. Inheritance
6. Polymorphism

OOP Intro

- Solving complex problems with functions (static methods) and primitives is possible, but there are better ways
- We usually want to solve real problems about real entities around us:
  - Modeling a transportation system: trains, buses, cars, people, roads, tolling, …
  - Each entity has its own set of properties and operations: remember the Data Type definition (set of values + operations)!
- What if each entity would "know" how to do its operations and would hide the details from us, just offering the operations we need?
- We could then just make method calls on those entities, requesting them to perform operations for us

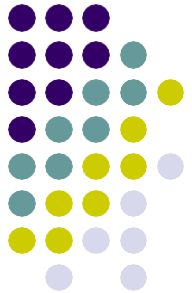
OOP Intro

- **Object -** Any entity that has **state and behavior.** For example: chair, pen, table, keyboard, bike etc. It can be physical and logical

- **Class -** is the blueprint from which individual objects are created.

- **Package** - is a namespace for organizing classes and interfaces in a logical manner.

Abstraction

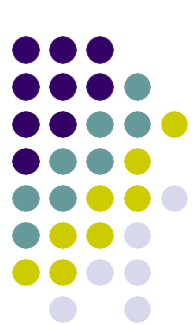


- In order to simplify reality we define **classes** which can instantiate multiple objects; e.g. Car, Color, Particle. This is called **abstraction**.

- It denotes **essential** characteristics of an object that distinguishes it from all other kinds of objects.

- We establish a level of complexity, and suppress the more complex details below the current level.

Why classes?



- Why not just primitives?

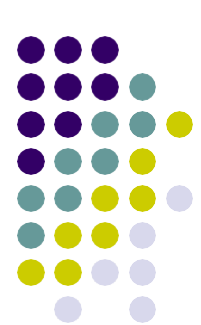
```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
```

Why classes?

- Why not just primitives?

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

Why classes?



- Why not just primitives?

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

David2?
Terrible ☹

500 Babies? That Sucks!

Why classes?

Baby1

Why classes?

Baby1

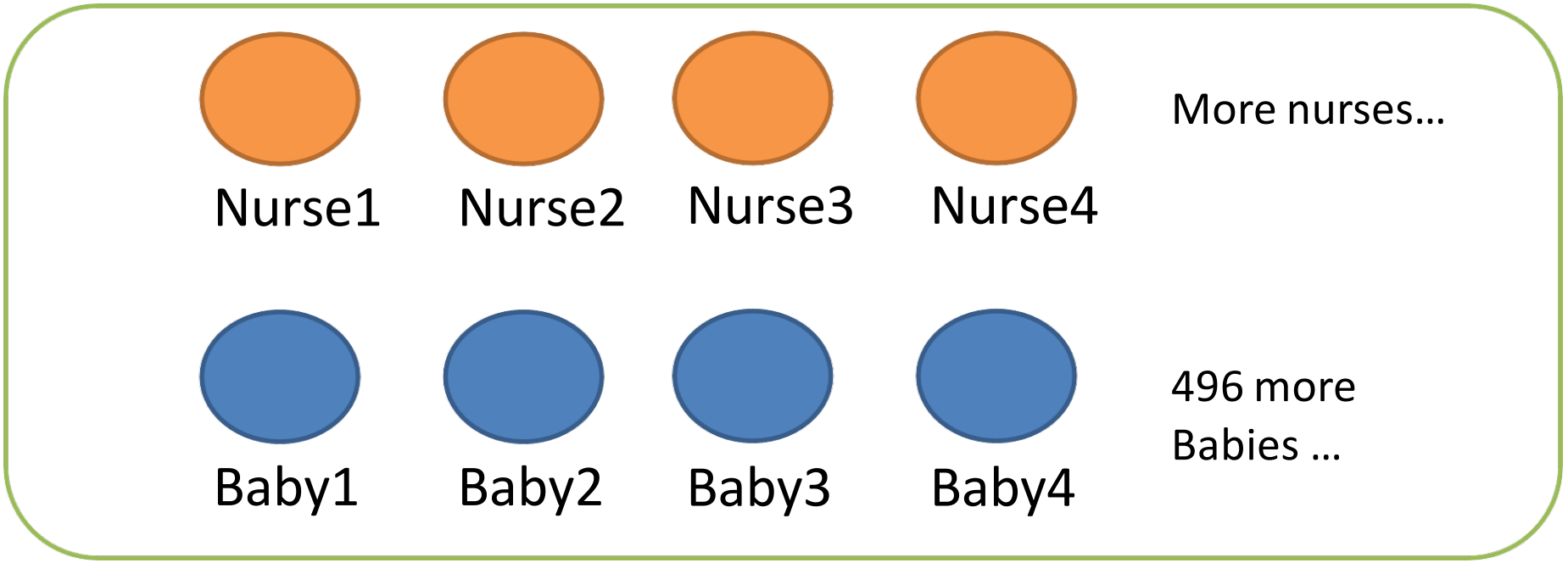
Baby2

Baby3

Baby4

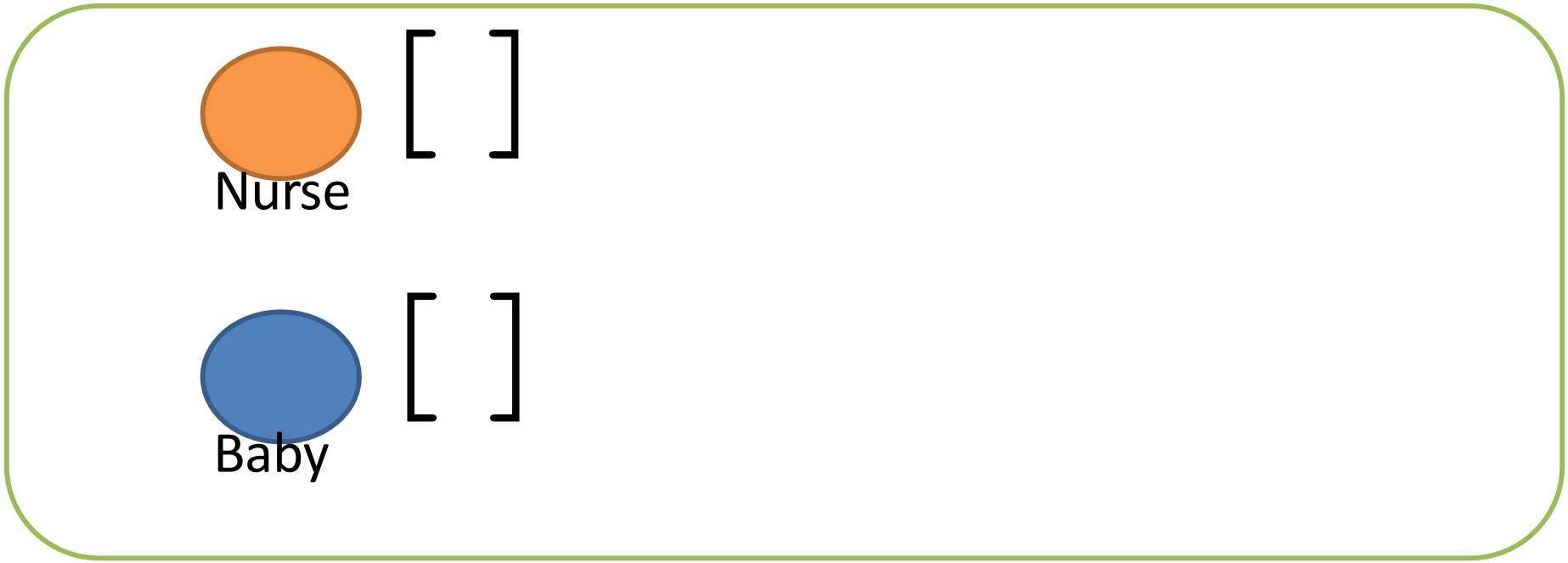
496 more Babies …

Why classes?



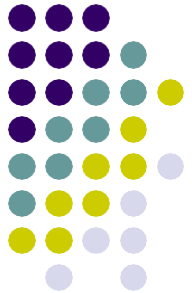
Nursery

Why classes?



Nursery

Let's declare the Baby!

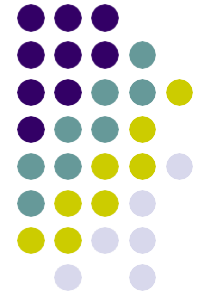


```
public class Baby {

}
```

fields

methods

Let's declare the Baby!



```
public class Baby {
    String name;
    boolean isMale;
    double weight;
    double decibels;
    int numPoops;

    void poop() {
      numPoops += 1;
      System.out.println("Dear mother, "+
      "I have pooped.  Ready the diaper.");
    }
}
```

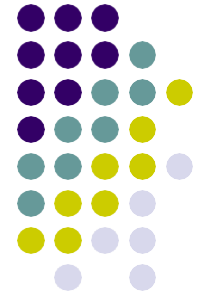
Let's create some babies!

```
Baby myBaby = new Baby();

Baby yourBaby = new Baby();
```

Class

Instances

Let's create some babies!



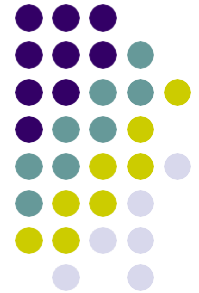
```
Baby myBaby = new Baby();

Baby yourBaby = new Baby();
```

Class

Instances

But what about their names? their weight?

Constructors



```
public class CLASSNAME{
    CLASSNAME ( ) {
    }

    CLASSNAME ([ARGUMENTS]) {
    }
}


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

Constructors

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

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

Constructors

```
public class Baby {
    String name;
    boolean isMale;

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

Classes

```
class Car {
      float fuelLevel;
      byte gear;
      float speed;
      String color;
}
```

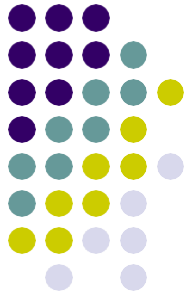
Classes

```
class Person {
   String name;
   String surname;
   byte age;
   boolean hungry;

   void eat() {
      hungry = false;
   }
}
```

Encapsulation

- It is good to:
  - Be able to have **entities** which are made of data **AND** behavior; they know how to *handle their internal state*.
  - Be able to hide unwanted details about how a certain class does its internal work.
- The concept which allows both of these is called **encapsulation**.

Encapsulation



- Entities usually ***exhibit behavior*** and ***hide attributes***
- How behavior is implemented should not concern the caller
- This gives freedom to change the implementation without affecting the clients.

Encapsulation

```
class Car {
        private float fuelLevel;
        private byte gear;
        private float speed;
        private Color color;
}
```

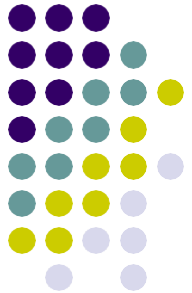
We hide the internal attributes.

Encapsulation

```
class Car {
        private float fuelLevel;
        private byte gear;
        private float speed;
        private Color color;

        public void accelerate(float speedDelta) {//… }
        public void steer(float angle) {//… }
        public void gearUp() {//… }
        public void gearDown() {//… }
}
```

- And expose only operations / behavior
- These operations will internally work with the private attributes

Encapsulation



- Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

- In encapsulation the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class, therefore it is also known as data hiding.

Composition

- Mechanism that allows a type (class) to be part of another type (to be composed inside another object)

- **Remember** - two main kind of types:
  - Primitive (built-in) types
  - Reference types: composed of primitive types and / or other reference types

- Also known as a "**has a**" relationship, implies ownership of the contained object.

Inheritance

- Inheritance is a principle that makes reusing behavior easier, by allowing classes to inherit behavior from another class

- This goes from generic to specific: specific data types exhibit the same behavior as their (more generic) “parents”
  E.g. Cat, Dog and Lion are each an Animal.

- It makes no sense to re-implement the same behavior in all possible specific classes.

⇒ the behavior and state is inherited.

⇒ There are means which allows us to implement a more specific behavior

Polymorphism

- Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

- Also refers to the ability of a class to provide different implementations of a method, depending on the type of object that is passed to the method.

- To put it simply, polymorphism in Java allows us to perform the same action in many different ways. When one task is performed by different ways i.e. known as polymorphism.
  E.g. every Animal eats, but each is doing it in its own way

- Concept which allows treating the sub-classes in the same way as treating the super-class.

Summary

- Classes
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Questions

Bibliography

- https://docs.oracle.com/javase/tutorial/java/concepts/
- Thinking in Java 4th Edition, by Bruce Eckel
- http://beginnersbook.com/2013/04/oops-concepts/
- https://introcs.cs.princeton.edu/java/home/
- https://ocw.mit.edu/