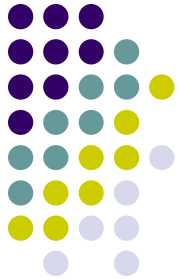
Java 100/105/200

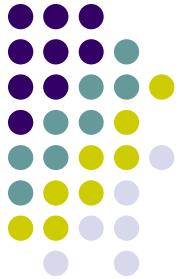
2 Object oriented programming



Project management and entities

OO DEVELOPMENT

Project stages

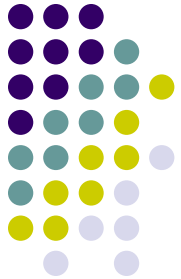


- Analysis
  - Understanding and communicating scope
  - High-level models
  - Business language
- Design
  - Understanding how to accomplish goals
  - More detailed models
  - Technical language
- Development
- Maintenance

Traditional analysis

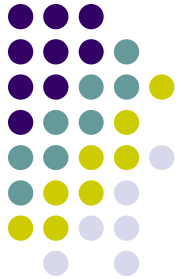
- Collect needs to define scope
- Organize needs by categories
- **Process** decomposition
  - Identify process flows
  - Break large processes into smaller
  - Break processes into tasks
- Define **data entity** structure to support tasks
- Move to design/development

OO analysis



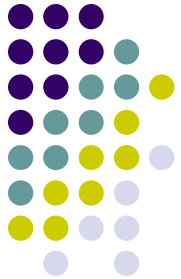
- Collect needs to define scope
- Organize needs by categories
- **Process flow decomposition (use cases)**
  - Identify process flows
  - Break high-level processes into smaller
- Define **data entities** in processes
  - Break processes into tasks
  - Add detail to data entities
- Move to design/development

OO advantages



- Easier to maintain
- Modules organized by entity (important data)
- Better architecture

OO entities



- A entity/class has two parts
  - Data parts grouped together by entity
    - Date **of sale**
    - Price **of sale**
    - Seller **of sale**
    - Buyer **of sale**
  - Processes grouped together by entity
    - **Print** address card of **student**
    - **List students** addresses by state
    - **Find student** by ID

OO entities

- Business entities can encapsulate real-world data in four major categories
  - **Physical** – a person, place, thing
  - **Role** – information/permissions for tasks
  - **Event** – a time bounded/related occurrence
  - **Reference** – look-up table data or constants
- Design entities are abstractions to manage other entities
- see Peter Coad – Java Design

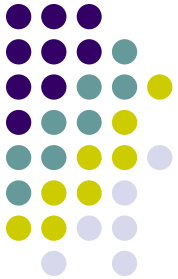
The object data structure

CLASSES & OBJECTS

A relational data structure

- A relational database
  - is defined by a **table** (the schema)
    - divided into **fields**
      - assigned a data type
  - has data created in **rows**
    - is uniquely defined by a primary key
    - can tie to other rows by storing a foreign key
  - sometimes has a **stored procedure** that operates on a table of data

An object data structure



- A object
  - is defined by a **class** (the schema)
    - divided into **fields** or instance variables
      - assigned a data type
  - has data created in **objects**
    - is uniquely defined internally
    - can tie to other objects by storing a reference
  - sometimes has a **method** that operates on a class of data

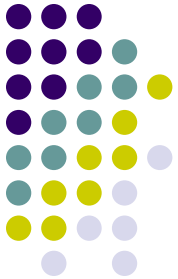
Relational vs. OO

- Program entities = database = spreadsheet
- Class = table schema = Excel tab
- Object = row of data
  - Instance = object
  - Instantiate = create
- Instance variable = field which holds data
- Member = field or method in class

Defining the class (schema)

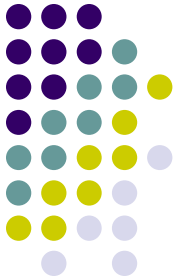
- Usually nested in a package
- Starts with an access modifier (none, **protected**, **public**)
- Uses the keyword **class**
- Ends with a class name which uses identifier rules
  - style convention is using a Capital letter first
- A code block follows

Class data structure



- An instance variable (field) has
  - an access modifier – usually **private**
  - a data type
  - a variable name

Data analysis

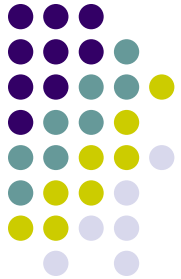


- The purposes for having an object use an instance variable are to
  - **Remember** an association to another object
  - **Track** the state of a piece of data or object
  - Use to generate a **report** in the future

Class as a data type

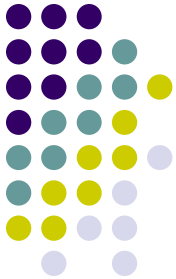
- The name of a class can be a data type
- Aggregation (composition)
  - using a complex structure of a class as a field in another class
  - just like making a foreign key to primary key relationship
  - parent/child relationship

Constructors



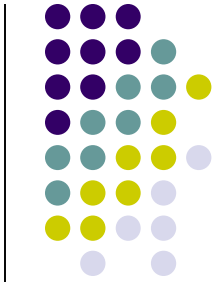
- The only way to create an object
- Like adding a row to a table
- Uses the class name as a method name
- Default constructors are provided by default and can be publicly called
  - new Person( );

Reference → object



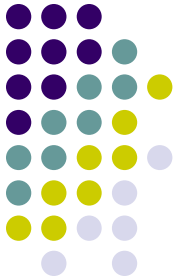
- A reference has a datatype
  - **Car** c
- An object has a datatype
  - new **Car**( )
- When the object is created, it must be assigned to a reference for reuse.
  - **Car** c ← new **Car**( );
- The datatypes must be compatible
  - **Car** c = new **Car**( );

Basic steps of object use



- **Declare** a variable reference (holds an object)
  - uses the class name as data type
  - `Person me;`
- **Create** the object (an empty container)
  - uses keyword **new** and a constructor method
  - `me = new Person( );`
  - `Person me = new Person( );`
- **Initialize** the fields in the object
  - direct access fields by **object.field**
  - `me.name = "Doug";`

Exercise

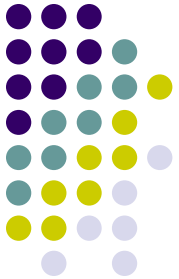


- Create a **Dog** class with a few fields
- Create a **Person** class with a few fields
- Aggregation
  - Add a owner field to the **Dog** class
- Initialize the fields of the Dog
- Print the data from the fields

Many references, no object

- An object can have multiple references
  - Dog fido, rover, max, spot;
  - fido = new Dog( );
  - rover = fido;
  - max = fido;
- References don't point to other references
- A reference can point and object or to no object
  - System.out.println(spot)
  - fido = null;

Exercise



- Share an object between two references
- Null one of the object references
- Create a new object for the null reference
- Print out field data to confirm what is happening.

Garbage collection

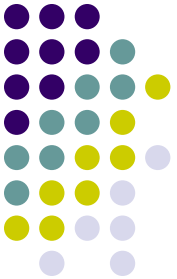
- Objects without reference are removed from active memory.
- A background thread is constantly looking for candidates for GC.
- You can’t control GC.

Class data structure - static

- keyword applied to field
- does not copy value to all objects, keeps one copy accessible via the class name
  - Data can change! - updateable
- One value for all Dog objects
  - static Person veterinarian
  - static double officeVisitRate
  - static String kennelName
- Reference
  - Dog.veterinarian, Dog.officeVisitRate, Dog.kennelName

static or instance?

- **A class to hold the data** – the data field
  - **HouseLoan**- Prime rate as of today.
  - **BondFilm** - The person who plays James Bond in a 007 film.
  - **Employee** - Your salary
  - **MileageMeasurement** - The distance between Omaha and KC
  - **Person** - Social security number
  - **KidsGreenRide** - The minimum height for riding a type of kid's ride at Worlds of Fun
  - **Meal** - Total calories



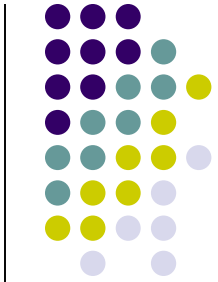
Exercise

- Create a static field for a Dog or Person

Simple data structure for multiple values or references

ARRAYS

The array (object)

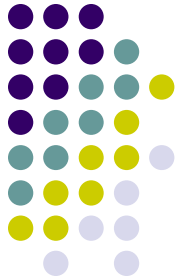


- An object that hold multiple values/references of any one data type.
- Built in to the language so it’s fast
- Fixed length (will not expand)

Declaring arrays

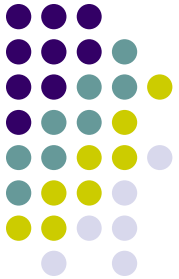
- Square bracket set follows data type
  - String[ ]
  - int[ ]
  - Dog[ ]
- Identifier follows data type
  - String[ ] roster
  - int[ ] scorebook
  - Dog[ ] kennel

Creating arrays



- Size must be declared when an array is created.
  - String[ ] roster = new String[5];
  - int[ ] scorebook = new int[25];
  - Dog[ ] kennel = new Dog[15];
- Or use initialization (see later)

Initializing arrays



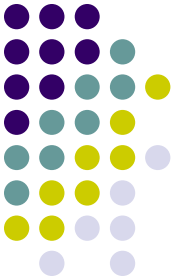
- Zero based counting
  - roster[0] = “Doug”;
  - scorebook[24] = 97;
  - kennel[1] = new Dog( );
- All values not initialized will have
  - Null if a reference
  - Zero or false if a value

Quick creation & initialization

- Multiple value creation & initialization
  - String[ ] roster = {“Doug”, “Dave”, “Teri”, null, null, null };
  - int[ ] scorebook = {96,93,83,86,79,96,82,88,0,0,0};
  - Dog[ ] kennel = {
    - new Dog( ),
    - new Dog( ), null
  - };
- Also known as array initializers

Anonymous arrays

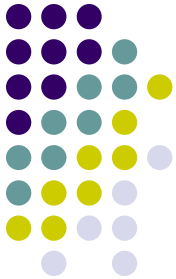
- Sometimes you don’t need to save the reference
  - public static void iNeedAnArray(int[ ] numbers){…}
  - iNeedAnArray(new int[ ] {1,2,3,4,5,6});
- Easier and clearer
  - int[ ] ints = {1,2,3,4,5,6};
  - iNeedAnArray(ints);



Multi-dimensional arrays

- Array references can be stored in an array to make an array of arrays.
  - Egg[ ] eggCarton = new Egg[12];
    - eggCarton[0] = new Egg( );
  - Egg[ ] [ ] eggCrate = new Egg[30] [ ];
    - eggCrate[1] = eggCarton;
  - Egg[ ] [ ] [ ] eggTruck = new Egg[96] [ ] [ ] ;
    - eggTruck[2] = eggCrate;
- How many eggs are in the eggTruck?

One field, no methods

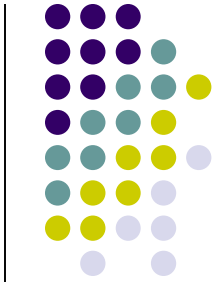


- Arrays don’t do much.
- Get the size of the array with
  - Dog[ ] kennel = new Dog[5];
  - kennel.length
- More static methods are found in Arrays
  - Arrays.sort(roster);
  - Arrays.toString(kennel);
  - Arrays.deepToString(eggTruck);

java.util.Arrays

- Because there is no Array class, there is a utility class
- static methods
  - int **binarySearch**(array)
  - array **copyOf**(array), array **copyOfRange**(array)
  - **fill**(array, value)
  - **sort**(array), **sort**(array, from, to)

Enhanced for loop (1.5)



- foreach
- Traditional for loop
  - for (int i = 0; i < kennel.length ; i++) {
    - kennel[i].bark( );
  - }
- All iteratable items, beginning to end
  - for (Dog dog : kennel) {
    - dog.bark( );
  - }

The main( ) args array

- On the command prompt you run a program by typing
  - java package.Class
- If you add program arguments they follow:
  - java package.Class one too tree “fo wah”
- The …main(String[] args) {… parameter is initialized with the arguments

Iterating over command line input (CommandLineArgs)

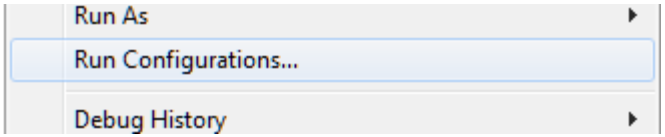
- public static void main(String[ ] **args**) {
  - **for (String string : args) {**
    System.*out.*println(string)*;*
  - }
- }

Exercise (CommandLineArgs)

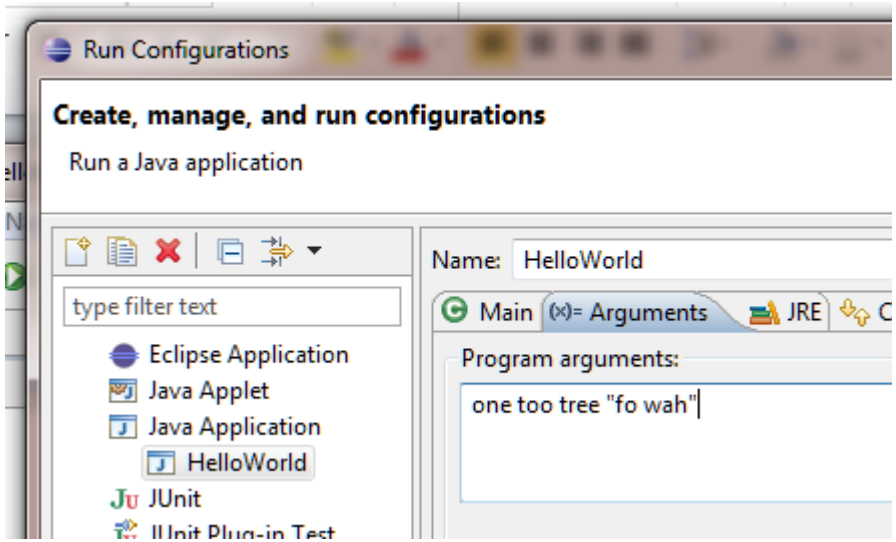
- Create a class that
  - Uses a foreach loop to iterate over the main's args
  - And prints out each one.

Command line input - Eclipse

- After running a program, the run configuration is saved.
- Open up the run config that you want args for



- Add the program arguments

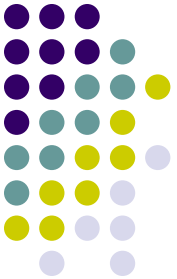


Exercise (ArraysTest)

- Duplicate your CommandLineArgs class and
  - Sort the args
  - Print out the result
  - Print out the Arrays.toString( ) of the args

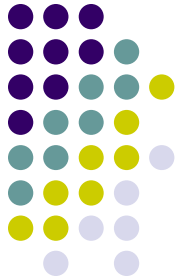
varargs (1.5)

- last parameter, one datatype, like an array
- Old style
  - static void test(int[ ] numbers)
  - test(new int[ ] {1, 2, 3});
- Vararg style
  - static void test(int … numbers)
  - test(1, 2, 3)
  - test( )
  - static void test(String s, int … numbers)



printf

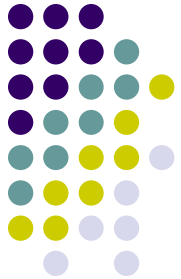
- printf(<formatted string>, args…)
- System.out.printf("I want a %s with %s and a %s.", "Corvette", "a sun roof", "set of new tires");
- System.out.printf(" My %d %s cost $%,.2f\n", 2016, "computer", 2442.32789);



Sending a message to an object

INSTANCE METHODS

Two class code sections



- One part of the class is data definition
- The other part is defining processes that use the data by default.
  - like associating a stored procedure with a table
- Separate the two sections in your code by comments to see easier.

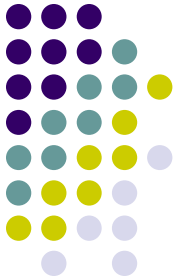
Instance methods

- Instance methods do not use the keyword **static**
  - static methods are called class methods
- Called not by prefixing class name but by the object.
  - me.printProfile( );
- The object becomes an argument to use in the method body.

Static vs instance

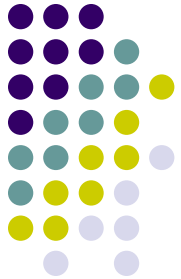
- public static void printName(Person someone){
  - System.out.println(someone.firstName + " " + someone.lastName); }
- public void printName( ) {
  - System.out.println(firstName + " " + lastName); }
- Calls
  - Person.printName(aPerson);
  - aPerson.printName( );

this

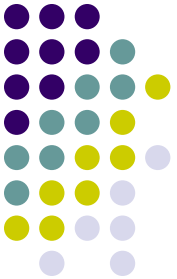


- The object that is being talked to must be created before
  - Person me = new Person( );
  - me.printProfile( );
- The method doesn't know the object being talked to so must use a proxy word – **this**
  - public void printProfile( ) {
    - System.out.println(this.name);
  - }

this



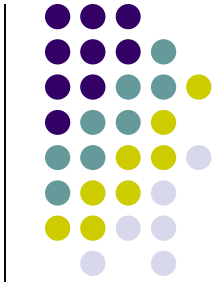
- The keyword this is often assumed and not written.
  - System.out.println(name);
- Best practice: use **this**, always!
  - System.out.println(**this**.name);



Exercise

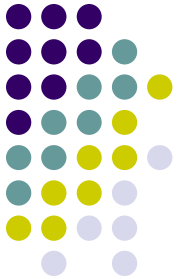
- Dog class
  - Create a bark( ) method
  - Create a bark(int howManyTimes) method
  - Create a getDataInOneString( ) method
- optional methods
  - goOutside( )
  - celebrateBirthday( )

Reusability



- If you copy and paste your code, you're doing something wrong.
  - Either call the method
  - Or extract a method and call it from all places
- Eliminate bugs
- Adds meaning

Revisiting static

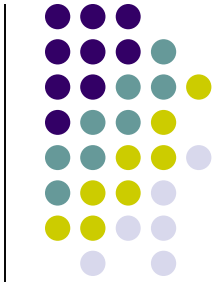


- Different for methods than fields
- Static methods
  - Can use static data or object arguments
  - Can't use **this** – no implicit object
- Use for grouping utility methods when the object could be anything
- Use instance methods when the object is very likely one data type

Static blocks

- Static variables (no unique storage for each object) can be set
  - When declared
  - When ever any object feels like it
- Some static variables need to be declared before object is created
  - And it takes code to run it (database connections)
- Use **static {** …init here… **}**

Unchangeable variables



- Some variables should not be changed once set in the object.
- Locked field
- The keyword **final** prevents updates
  - final String SSN

Constants

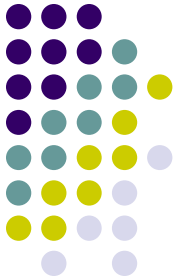
- A constant is unchanging therefore it uses the keyword **final**
- A constant only needs one copy for all objects of the class therefore it uses the keyword **static**.
- A constant identifier is usually put in all caps and uses an underscore as a word separator
  - public static final double PI = 3.141592653589793
  - public static final String SPECIES = "Canis lupus"

final or not?

- **A class to hold the data** – the data
  - **HouseLoan**- Prime rate as of today.
  - **BondFilm** - The person who plays James Bond in a 007 film.
  - **Employee** - Your salary
  - **MeasurementInMiles**- The distance between Omaha and KC
  - **Person** - Social security number
  - **KidsGreenRide** - The minimum height for riding a kid's ride at Worlds of Fun
  - **Meal** - Total calories

Getters and setters

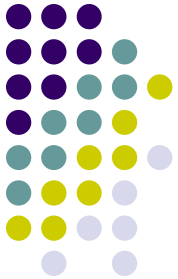
- Private data can not be accessed outside of the class code block.
- Public methods can be written to allow access to private data.
  - public <datatype> get<Field>( )
  - public void set<Field>(datatype field)
- Eliminate to restrict read/write access



Getters and setters

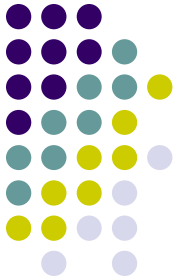
- Your IDE will generate both for you
  - Eclipse: Source / Generate Getters and Setters…
    - Insertion point: after the last field of your class
  - Select all getters and setters
- Place in a separate section of the methods

Getters and setters - uses



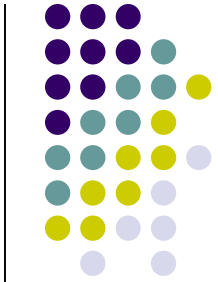
- Get
  - check authorization to get data
  - notify another object on access
  - transform output based on caller
- Set
  - validate data coming in and process
    - implement business rules here
  - record old data in case of reversal/veto
  - notify another object on change

Calling getters and setters



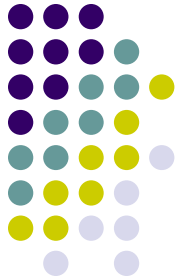
- Use getters and setters at all times
  - even in the same class where private data is accessible.
- me.setName(“Doug”)
- String name = me.getName( );

Exercises



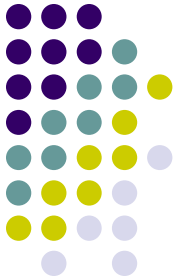
- Add getters and setters to **Person** and **Dog**
- Access getters and setters
  - initialize the pet and owner fields

toString( )



- The toString( ) method converts the object data into a String when attempting to print it
  - System.out.println(fido.toString( ));
  - System.out.println(fido);

toString( ) – IDE



- Eclipse
  - Source / Generate toString( )…
  - Select All (default)
  - Insertion point: before method of your class
  - Delete comments
  - Leave @Override, even though it’s optional

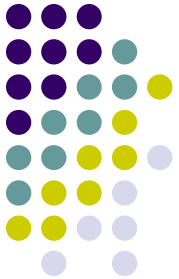
Exercise

- Generate toString( ) methods for **Person** and **Dog**
- Print out objects

Common class tasks

- Always create
  - Data structure
  - Getters/setters
  - toString( )
  - Unique behavior (methods) if known
  - 2 constructors – no-args and all-args
- Optional
  - Static member (field or method)
  - Constants
  - equals( ), hashCode( ), compareTo( ) - sorting

Random numbers



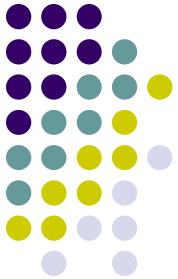
- java.util.Random
  - Random generator = new Random( );
  - int i = generator.nextInt( );
    - number between 0 and maximum int
  - int i = generator.nextInt(100)
    - number between 0 and up to 100
  - double d = generator.nextDouble( );
    - number between 0 and 1.0
- Math.random( )
  - returns number between 0 and up to 1.0

Long exercise – Payroll (200)

Create classes, method stubs, **fields**, gets/sets, toString( ) in a payroll package.

- Employee
  - **id, name, payGrade, hoursWorked**
  - *findById*(int)
- PayRate
  - constants
  - *findHourlyWageForPayGrade*(String)
- Check
  - **amount, checkLayout**
  - printForEmployee(Employee)
- PrintLayout
  - **description**
  - *createCheckLayout*( )
- Comptroller
  - main( )
    - with instructor: create an employee and print a check for the employee.
  - printCheckFor EmployeeID(int)

Requirements

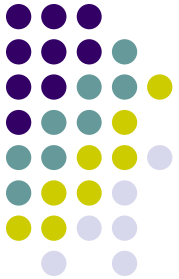


- Trigger – print check for an employee id
- Flow
  - Look up employee by id
    - Create an employee
  - Create a check to record for later
    - Set up a layout for a check
  - Tell the check to print employee's amount
    - Look up the current wage
    - Calculate amount (wage * hours worked)
    - Print amount in layout

Initializing your data structures

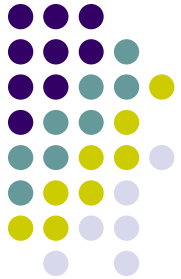
CONSTRUCTORS

Purpose



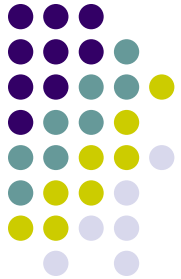
- To provide convenient way of initializing objects
  - To provide default values for any object
- To allow / disallow the creation of objects

Default constructor



- The default constructor is always available if no other constructor has been written
- If you write it yourself, it looks like:
  - public Person( ) {
    - super();
  - }

Default constructor – IDE



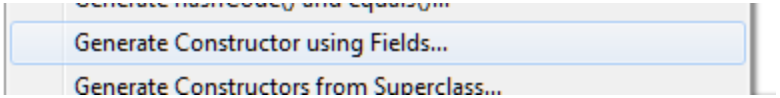
- Eclipse
  - Press Ctrl-spacebar in the class code block
  - Select the default constructor
    - Constructors have the little C on the icon
  - Delete the comment

Exercises

- Add a no-arg constructor to both a **Person** and **Dog** class
  - Add a println to say what class this is and “no-arg constructor”
  - Add default data by the set methods in the constructors.
  - Should the Dog always create a default Person as its owner?
    - It’s not always the case that the Person should have a default Dog so we won’t do that.

An all-field constructor

- Will allow you to initialize all of the fields of the object at once
- **Eclipse** shortcut
  - Source / Generate Constructor Using Fields…



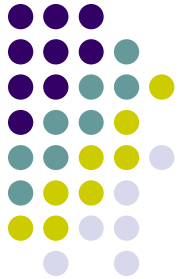
  - public Dog(String name, int age) {
    - this.name = name;
    - this.age = age;
  - }

Updating to use gets/sets

- Click on direct field access
- Hover on direct field access and select generate getters and setters
  or
- Eclipse: Refactor / Encapsulate Field / OK

- Do for each field

Exercise



- Create the all-arg constructors in Dog and Person
  - Add the println( ) to identify the constructor being run
- Test all-arg constructors
  - Create a Person with a Dog

Shadowing

- When a variable uses the name of another variable and makes it unavailable
  - public Dog(String **name**, int **age**) {…
    - this.name = name
- Avoid shadowing, it's confusing
- Use prefix to help understanding
  - public Dog(String **_name**, int **_age**) {
    - name = _name;
- Even better, use this.setName(name)

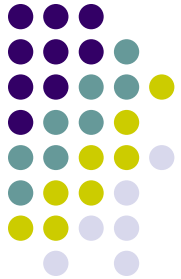
DRY in constructors

- All-arg constructor has all the logic we need.
- Have the no-arg constructor call the all-arg

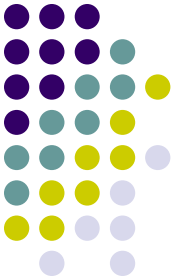
```
public Dog() {
    this("Fido", 3, "Beagle");
}

public Dog(String name, int age, String breed) {
    this.name = name;
    this.age = age;
    this.breed = breed;
}
```

- **this** has two uses: object proxy, constructor redirect in the same class.

Exercise

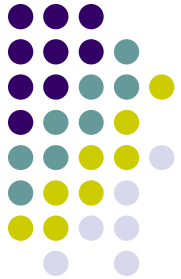


- Tie the no-arg and the all-arg constructors together in both Person and Dog
  - Have the no-arg call the all-arg constructors
  - Have any logic in the no-arg constructor move to the all-arg constructor



Exercise

- fields
  - **double firstNumber**
  - **double secondNumber**
- Instance methods
  - calcSum( ), calcDifference( )
- common members
  - toString( ), gets/sets, constructors (no-arg, all-fields)
  - main( ) to test all methods



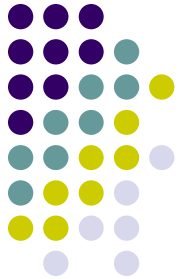
One class “is-a-special-type-of” another class

INHERITANCE

Inheritance

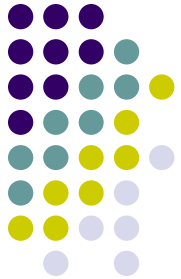
- a relationship between two classes based on wrapping behavior
  - The goal is to use that shared behavior to achieve code reuse (polymorphism)
  - Lets you use two objects with different datatypes as one type to reuse one method
- vs. an association
  - preferred when possible
  - combining data into one entity
  - Aggregation and composition are stronger forms

Demo



- Show shared behavior/fields through using aggregation
- Show direct access of shared behavior using delegate methods

Motivation

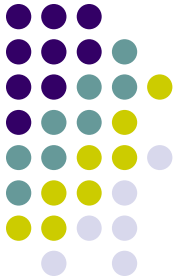


- Use inheritance when you need
  - to access a common method on many different data types
  - to write a method that accepts multiple data types based on what they do (there’s a better way)
  - Customize a class from another library

Terminology

- Superclass
  - base class – generalized, shared behavior
- Subclass (subtype)
  - derived class – specialized, unique behavior
- Say “a subclass is a special type of superclass”
  - a Person is a special type of Object
  - an Employee is a special type of Person
  - a Teacher is a special type of Employee

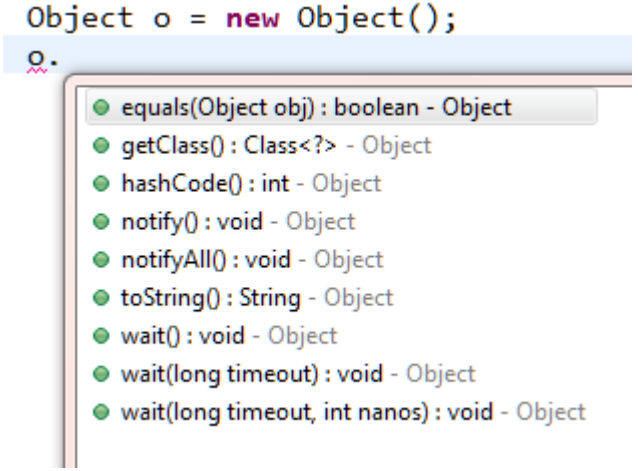
Implementation



- **extends** keyword
  - RaceCar extends WheeledVehicle
  - HuntingDog extends Dog
  - TV extends RemoteControlledDevice
- apply the “is-a-special-type-of” test
  - a RaceCar is a special type of WheeledVehicle
  - a HuntingDog is a special type of Dog
  - a TV is a special type of RemoteControlledDevice

Object class

- All classes inherit from Object
  - Dog is a special type of Object
- Object has several useful methods
  - toString( )
  - equals( )
  - many others concerned with threads

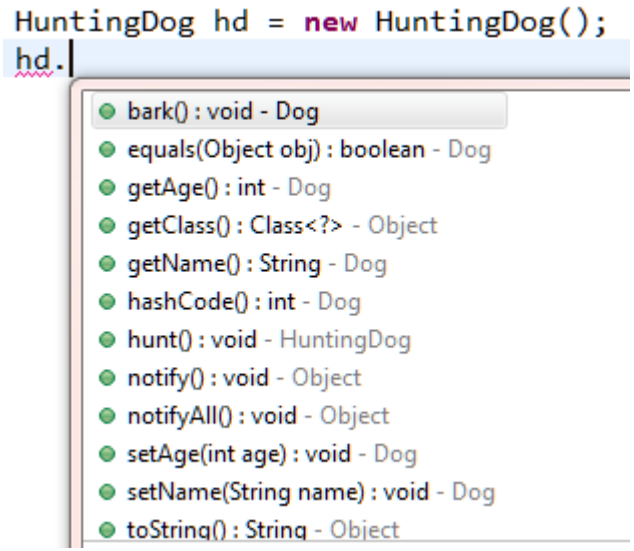


Behavior expansion

- A subclass is not a subset
  - Sub means **specialized** not “a smaller part”
- Subclasses
  - have more functionality
  - have more fields they can store
  - are a way you can add your personalized touch to an existing class

Inherited fields/methods

- No private fields/methods will be available in the subclass
- The subclass can access any method it knows about
- The subclass can access any method from any of its superclasses



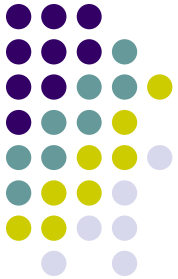
Exercise

- Make sure to have a **Dog** class
  - print out object in the constructor
- Create a **HuntingDog** class with
  - an inheritance relationship to **Dog**
  - a hunt( ) method
  - a field for specialty
  - gets and sets
  - constructor with a default value for specialty

Stopping inheritance

- Some class designers want to prevent you from subclassing their work
- The keyword final prevents this
  - public **final** class String { …
- String can not be subclassed

Overriding



- Overriding is using the same method signature in the subclass as the superclass
- This “hides” the superclass call (not the behavior.

```
monday.HuntingDog@555e018e
```

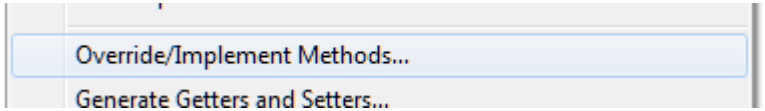
```
I'm a hunting dog
```

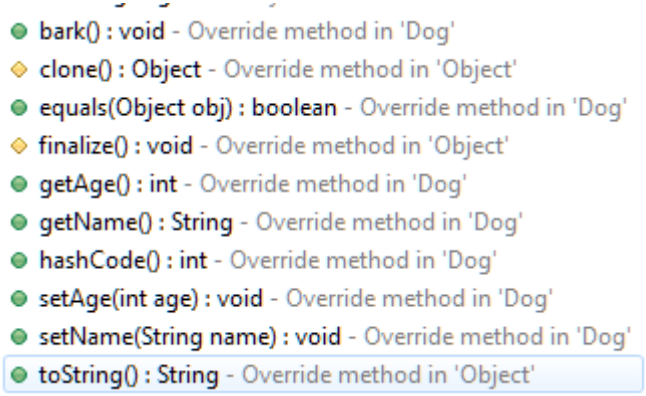
```java
@Override
public String toString() {
    return "I'm a hunting dog";
}
```

- @Override is a compiler check to make sure that the method really does match the superclass method.

Overriding shortcuts

- Two methods
  - **Eclipse**
    - Ctrl-spacebar in the class code block to get suggestions for overrides
    - Source menu



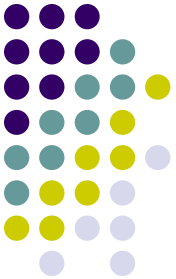


  - Implement methods or interface will be used later.

Stopping overriding

- Some method writers want to prevent you from overriding their work
- The keyword final prevents this also.
  - public **final** void someMethod( ) { }
- Prevents polymorphic use to the subclass

Accessing overridden methods



- Once you override a method, you should want to append to its behavior.
- Use super before the method name to access the superclass method of the same name.
  - super.toString( )

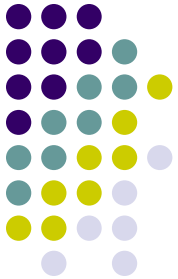
Dog.java | HuntingDog.java

```
@Override
public String toString() {
    return "I'm a dog";
}
```

Dog.java | HuntingDog.java

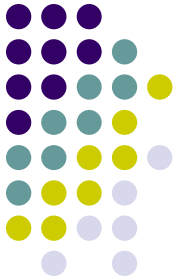
```
@Override
public String toString() {
    return super.toString() + " who likes to hunt";
}
```

Exercise



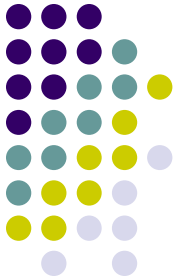
- Add the toString( ) method to the **HuntingDog**
  - **Eclipse**: use the Generate toString()…
  - select specialty
  - select inherited methods / toString( )

Abstract vs. concrete

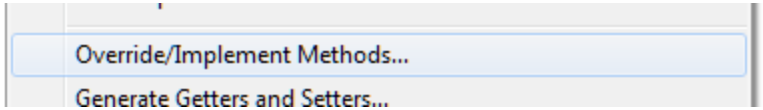


- Concrete classes can make objects
- Abstract classes can not make objects
  - public abstract class Animal { }
  - Used on a class when method signature is declared but has no code
    - public abstract void eat( );
- Abstract classes are used to guarantee that any related class has that behavior
- “a template class”

Inheriting an abstract class



- Inheriting an abstract class forces you to implement the abstract methods
  - public class Dog extends Animal {
    - public void eat( ) { }
  - }
- Let the IDE implement the method for you:



  - Eclipse - Ctrl-spacebar at start of line will override the method. Clicking the lightbulb also works.

Constructors & inheritance

- Constructors are set up to achieve a division of labor based on the fields in their respective classes
  - A superclass has one field, it should only initialize that field in its constructor
  - A subclass also has one field, it should provide a constructor with two fields and pass the data to its superclass.

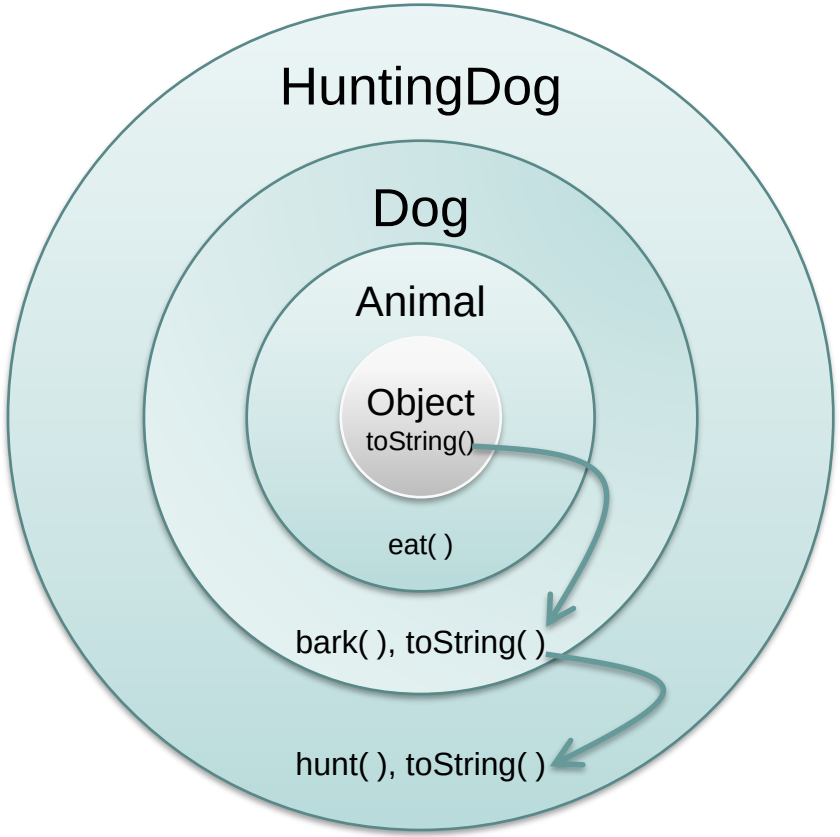
How to build an object

new Object( )

new Animal( )

new Dog( )

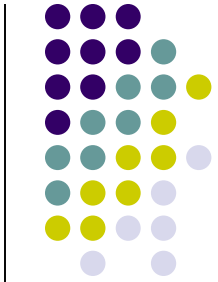
new HuntingDog( )



Construction process

- **super**( ) calls the superclass no-arg constructor
- **super**( ) is a default call for the first statement of any constructor unless **this**( ) is called
- Object creation starts with an Object class constructor, then runs the subclass constructor, then the next subclass…

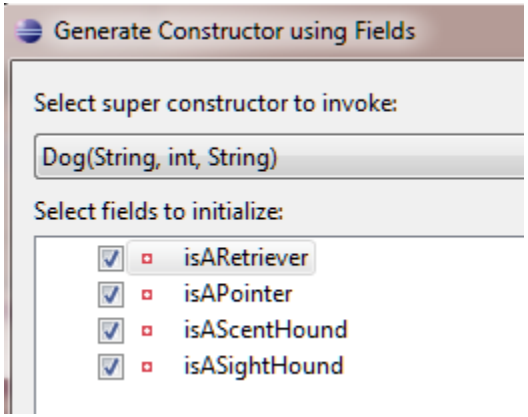
Object layers



- The data for an object is initialized from the core outward to the last subclass
- The constructor call is for the last subclass so it must pass the data to the next superclass
- public Subclass(datatype arg1, datatype arg2) {
  - super(arg1);
  - this.arg2 = arg2
- }

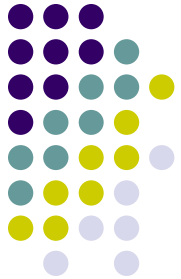
Creating constructors with IDEs

- Use Source / Generate Constructor Using/With Fields
- Change the super constructor to invoke to the all-args



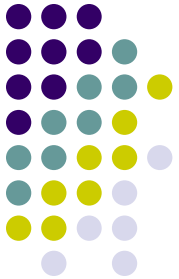

```
public HuntingDog(String name, int age, String breed, boolean isARetriever,
        boolean isAPointer, boolean isAScentHound, boolean isASightHound) {
    super(name, age, breed);
    this.isARetriever = isARetriever;
    this.isAPointer = isAPointer;
    this.isAScentHound = isAScentHound;
    this.isASightHound = isASightHound;
}
```

Exercise



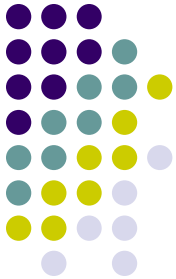
- Create the Dog / HuntingDog constructors
  - chain the constructors
    - no-arg or partial constructors call all-arg constructors
    - all-arg constructors call superclass' all-arg constructors

Debug - Eclipse



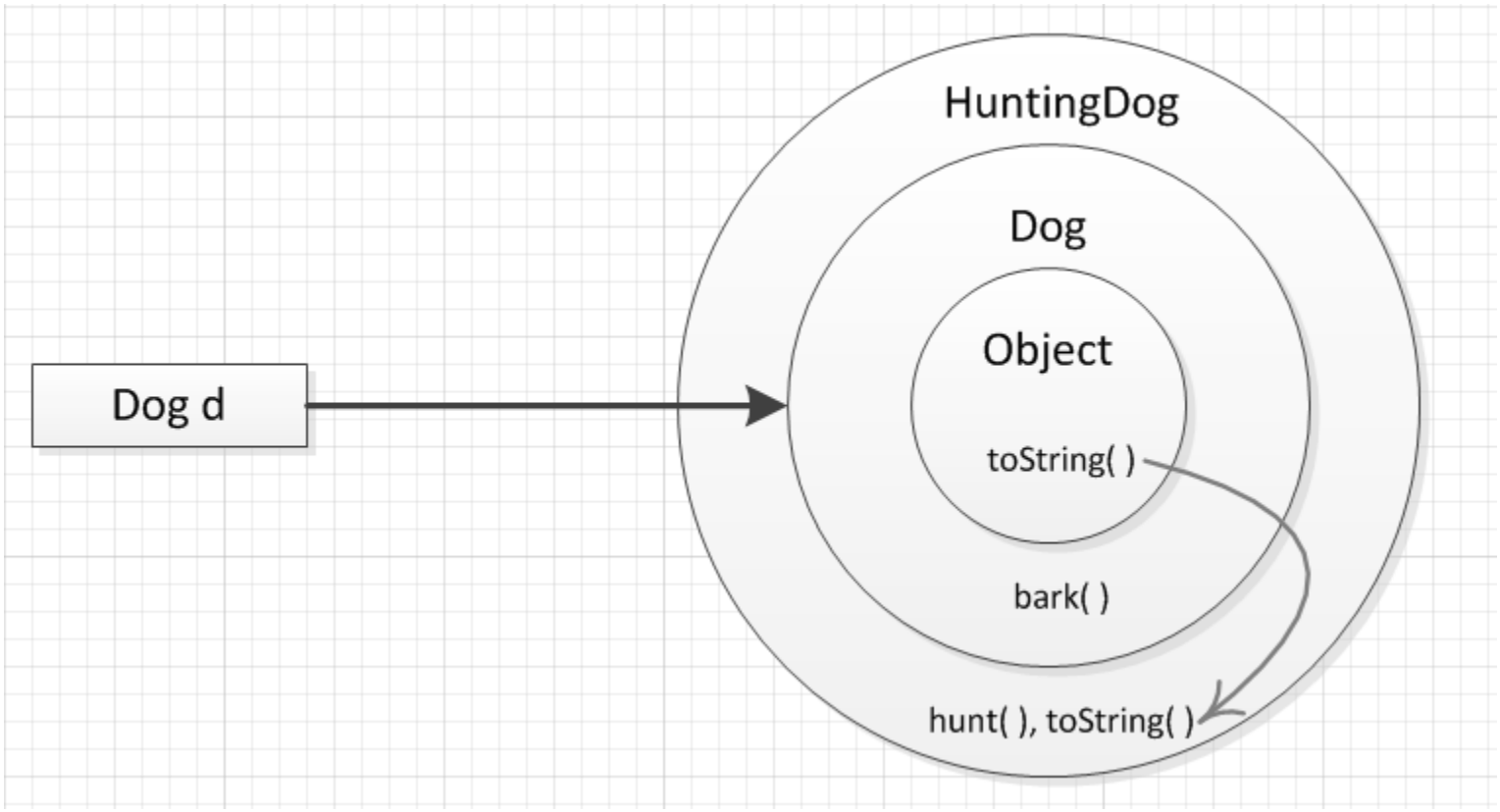
- Demo

Superclass references



- Object o1 = new Object( );
- Object o2 = new Dog( );
- Object o3 = new HuntingDog( );
- Dog d = new HuntingDog( );
- Object[ ] objects = {o1, o2, o3, d }

Object layers



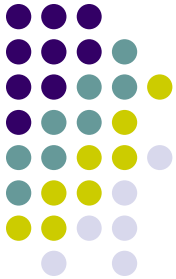
References to object layers

- Object references only allow access (scope) to the layer and inner layers (supertypes) they point to.
- Overridden methods allow access to a method only if a method by that name is in scope.
- You can increase the scope by creating a reference to the outer layer data type (subtype) by casting the inner layer reference type.

Casting references

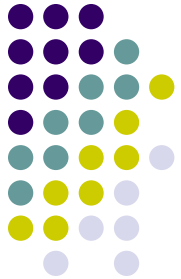
- Casting to a superclass is always OK
  - Object[ ] objects = {o1, o2, o3, d }
- You must confirm casting to a subclass
  - Dog d1 = (Dog) objects[1];
  - HuntingDog hd = (HuntingDog) objects[2];
- You only can cast to a datatype in the object's layers.
  - Dog d2 = (Dog) objects[0]; // not a Dog object

Exercise (ReferenceTest)



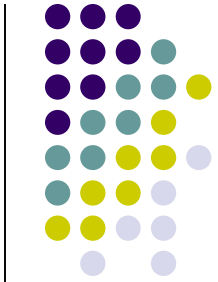
- Create a HuntingDog and an Object
- Put the objects into an array
- Print out both objects by array references
- Cast and use the array objects to
  - Animal
  - Dog
  - HuntingDog

Polymorphism



- Sending the same message to objects of different classes and getting different results.
  - anAnimal.toString( )
  - aDog.toString( )
  - aHuntingDog.toString( )
- Most often implemented by methods tied by overriding so we get useful results

Polymorphic methods



- Methods that use superclasses in parameter datatypes can be polymorphic

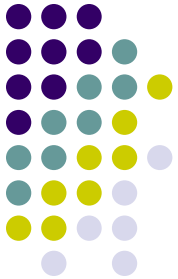
```
public static void main(String[] args) {
    Object[ ] myObjects = {new Object( ), new Dog(), new HuntingDog() };
    printObjects(myObjects);
}


private static void printObjects(Object[] objects) {
    for (Object object : objects) {
        System.out.println(object);
    }
```

- Implement this code.

Object oriented principles

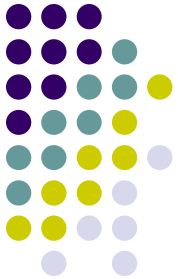
- Any OO language must support three major principles
  - **Encapsulation** – the definition of the fields, the behavior, and the access to those elements.
  - **Inheritance** – the ability to relate two classes so that fields & behavior is shared
  - **Polymorphism** – the ability to send the same message to objects of different classes and get different results.

Vehicle exercise (100)



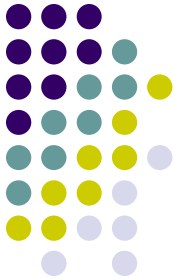
- Set up classes and methods for
  - Vehicle (abstract)
    - moveForward( ), abstract park( )
    - abstract turn( ) – turnRight( ) or turn(“right”) ?
  - WheeledVehicle
    - brake( ), toString( )
  - FlyingVehicle
    - takeOff( ), land( ), toString( )
  - FloatingVehicle
    - ffloat( ), toString( )

Vehicle exercise (100)



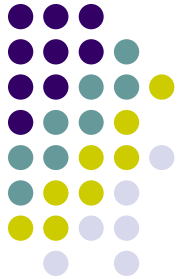
- Set up fields and constructors for
  - Vehicle (abstract)
    - driver, velocityMPS
  - WheeledVehicle
    - numberOfWheels
  - FlyingVehicle
    - flyingAltitude
  - FloatingVehicle
    - dockedAt

Vehicle exercise (100)



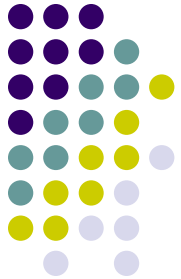
- Create a Valet class
  - Inherit from Person (with a name field)
  - main ( )
    - Create an array of vehicles
    - Initialize them
    - Move each forward then park the vehicle
  - parkVehicle( ) method
    - Greet the driver by name, use the valet name also
    - Tell them you will be parking their type of vehicle
      - getClassName().getName( )

Issues



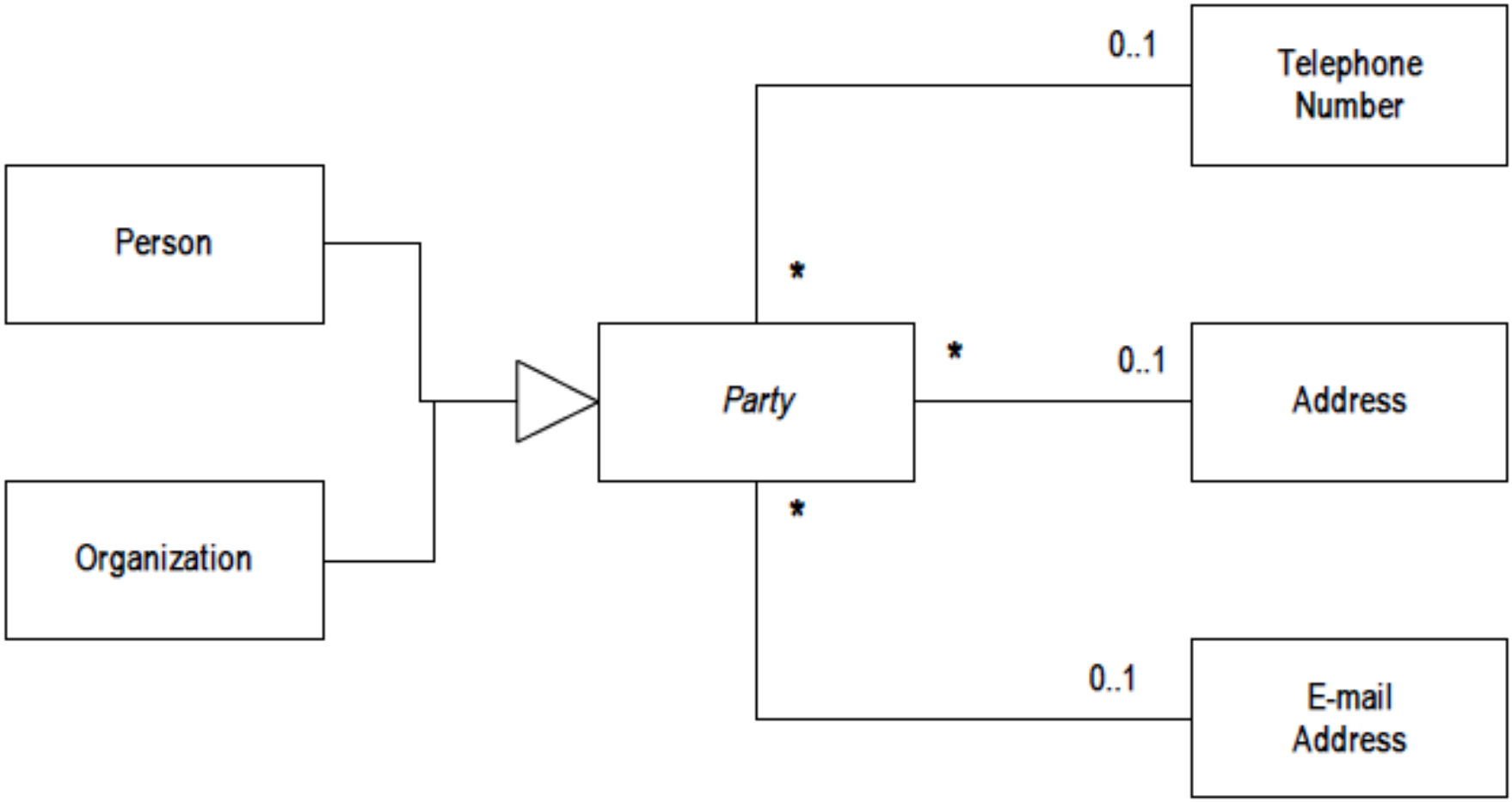
- FloatingVehicle
  - dive( )
- FlyingVehicle
  - dive( )
- Seaplane
  - wheeled, floating, flying
- Helicopter
  - doesn’t have all flying capabilities
    - rotor-based flying, wing-based flying ?

Person/Org exercise (105/200)

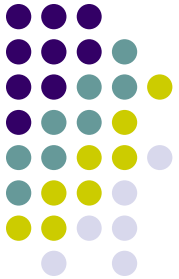


- Person
- Organization
- Telephone
- Address
- Email

Person/Org exercise (105/200)



Person/Org exercise (200)



- Implement the inheritance structure using aggregation.

100 End tasks

- Zip files and save/mail
- Certificate
- Evaluation