COMP 250
INTRODUCTION TO COMPUTER SCIENCE

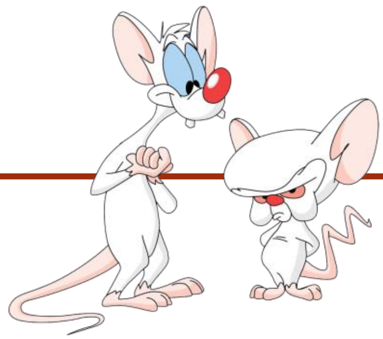
Lecture 5 – OOD1 Packages and Modifiers

Giulia Alberini, Fall 2018

FROM LAST CLASS

- Primitive Data Types
- Char and Unicode
- Type conversion

WHAT ARE WE GOING TO DO TODAY?



- Packages
- Review of Objects/Classes
  - General Structure
  - Default Constructor
  - Nested classes
- Modifiers
- UML Diagrams

PACKAGES

PACKAGES

- A package is a group of classes
  - Each class is referred to as a *package member*
- A class is a group of methods
- A method is an ordered group of commands

DEFINITION

- **To define a package we write at the top of our class file the following statement**

```
package packageName;
```

- **For example:**

```
package nba.annoyingTeams;

public class MiamiHeat {
   ⋮
}
```

This creates a class `MiamiHeat` **inside the package** `nba.annoyingTeams`

FILE AND FOLDERS NAMES

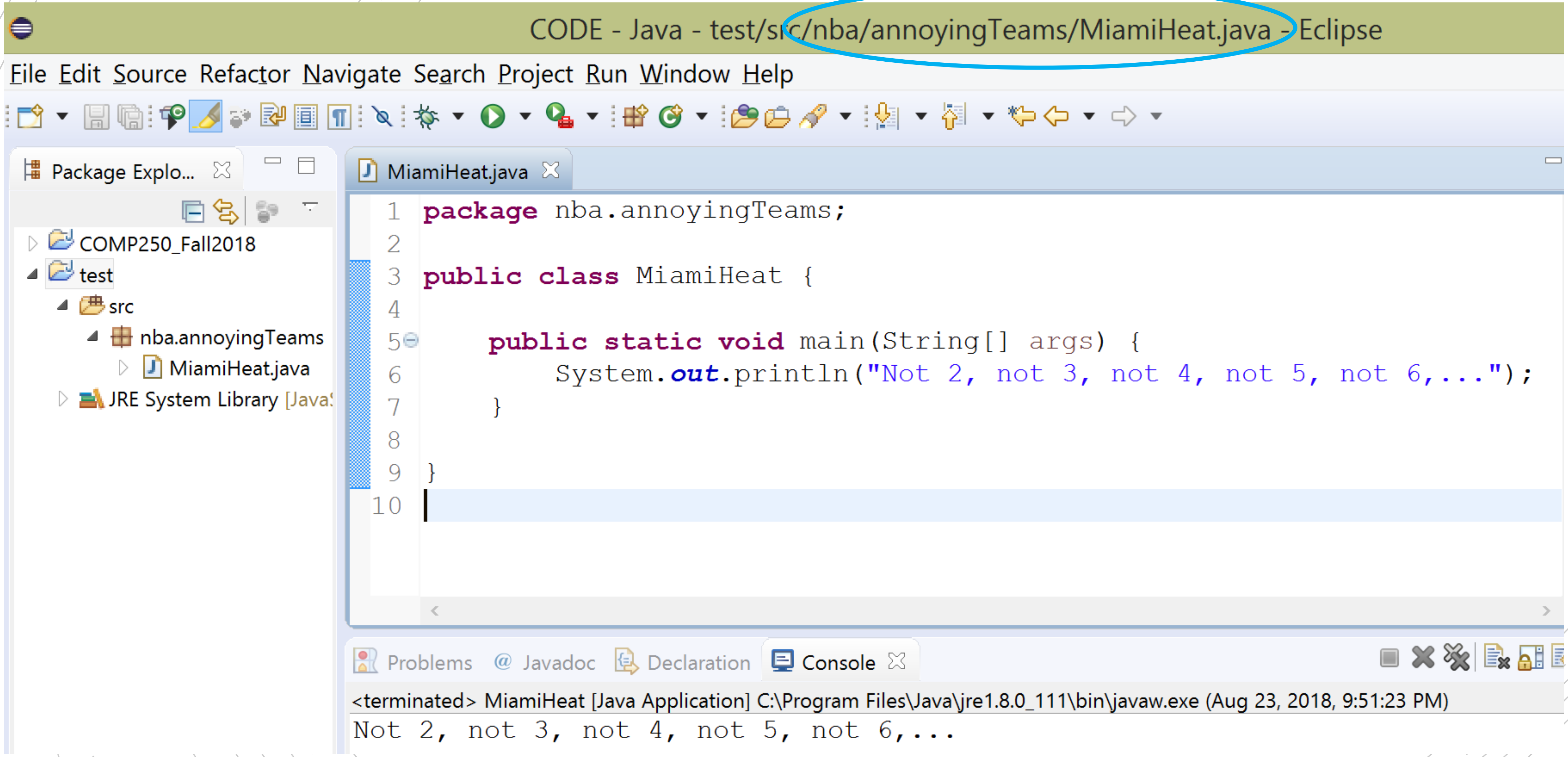
There are two main rules related to files' and folders' names in Java:

1. The name of the *class* must match the name of the file (with .java added) (e.g. *MiamiHeat.java*)

2. The folder path must match exactly the package name – except that each period is actually a "slash" (i.e. a subfolder)

In the example before, a folder *nba* must contain a folder *annoyingTeams* which contains the file *MiamiHeat.java* https://youtu.be/e9BqUBYaHlM?t=1m53s

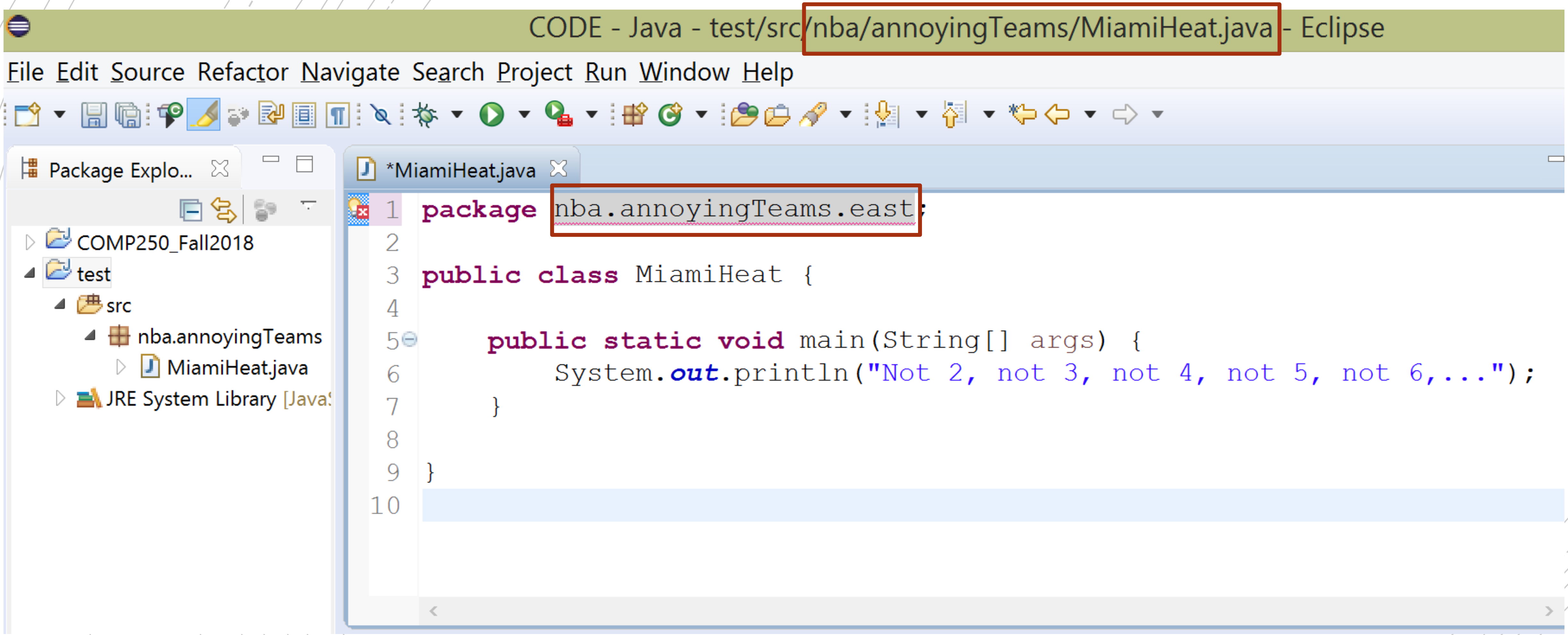
EXAMPLES



```java
package nba.annoyingTeams;

public class MiamiHeat {

    public static void main(String[] args) {
        System.out.println("Not 2, not 3, not 4, not 5, not 6,...");
    }

}
```

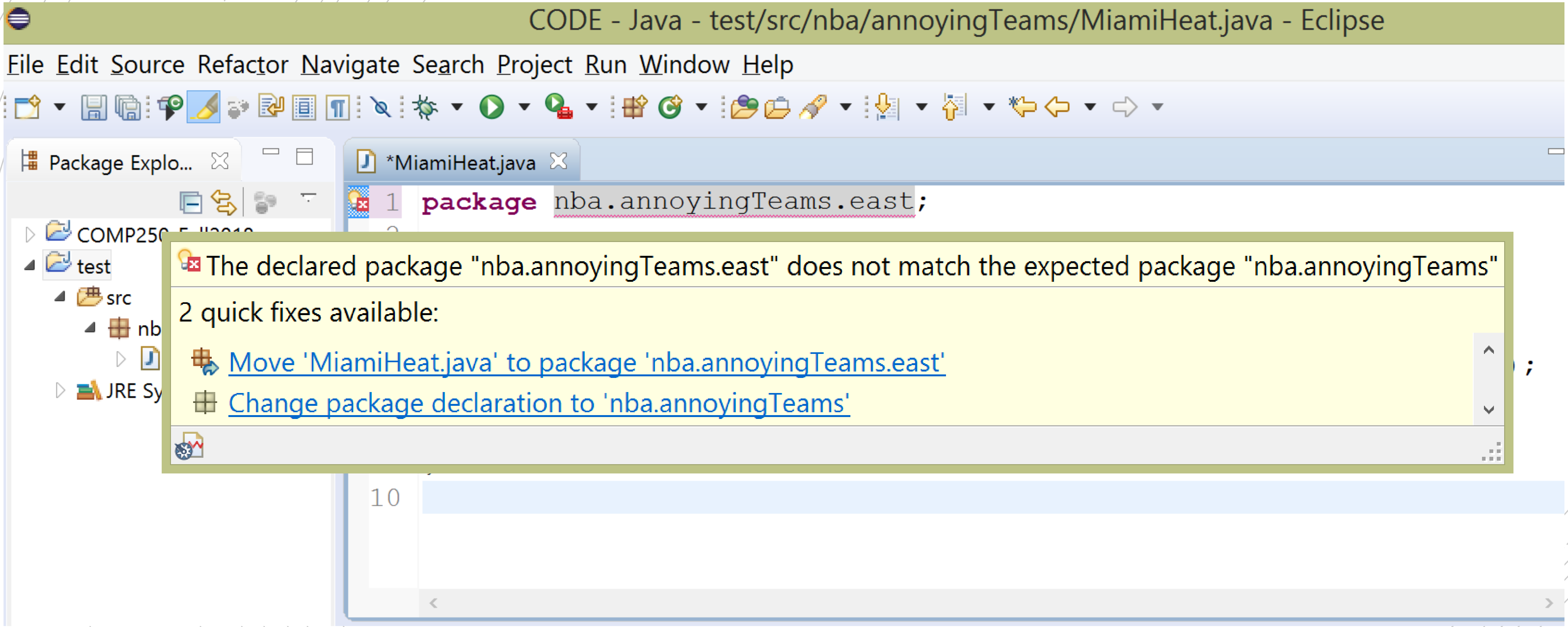
```
<terminated> MiamiHeat [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (Aug 23, 2018, 9:51:23 PM)
Not 2, not 3, not 4, not 5, not 6,...
```

EXAMPLES


CODE - Java - test/src/nba/annoyingTeams/MiamiHeat.java - Eclipse

```
package nba.annoyingTeams.east;

public class MiamiHeat {

    public static void main(String[] args) {
        System.out.println("Not 2, not 3, not 4, not 5, not 6,...");
    }

}
```

EXAMPLES


CODE - Java - test/src/nba/annoyingTeams/MiamiHeat.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help
Package Explo...
*MiamiHeat.java
COMP250
test
src
nb
JRE Sy

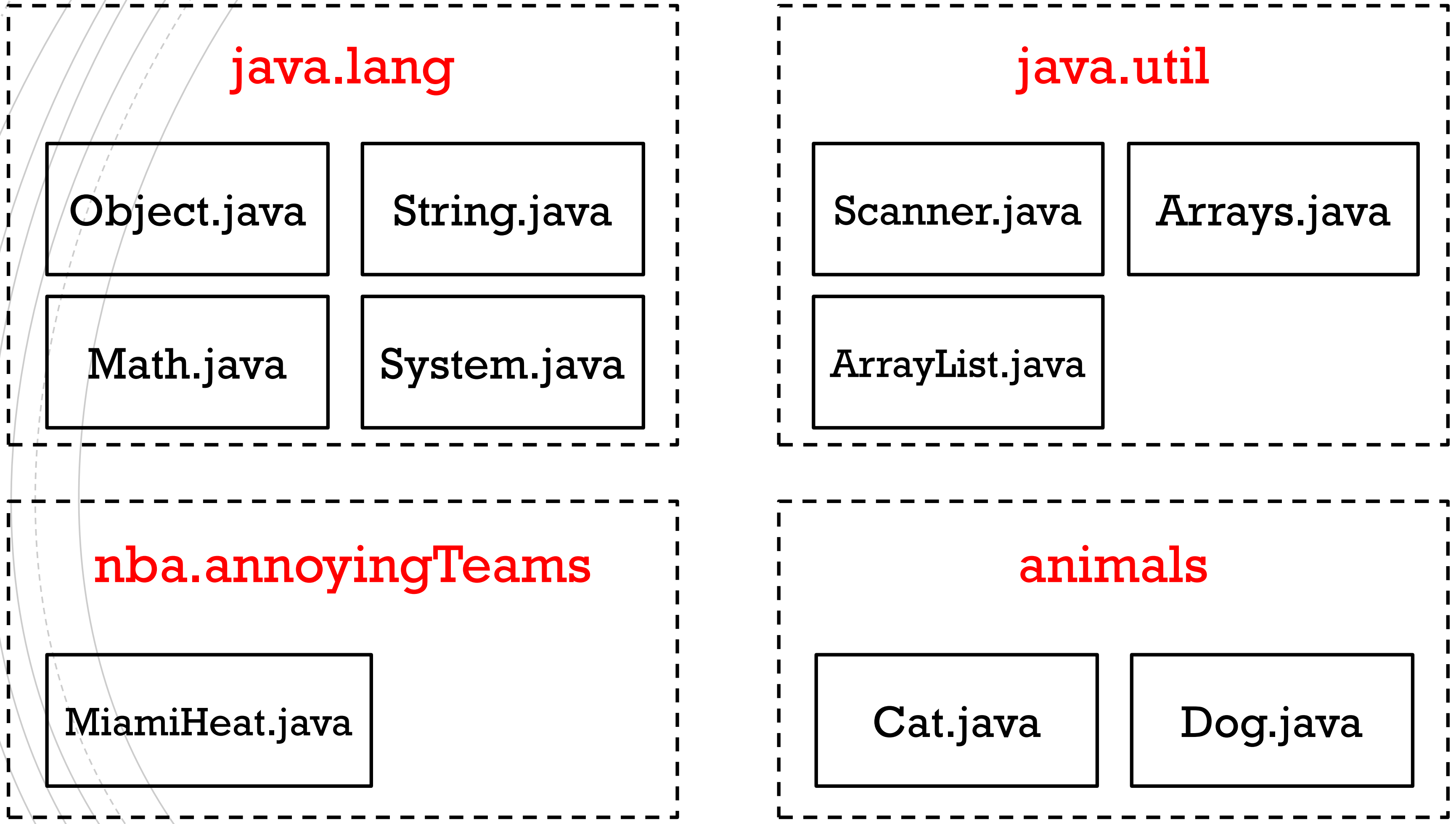
```
package nba.annoyingTeams.east;
```

The declared package "nba.annoyingTeams.east" does not match the expected package "nba.annoyingTeams"

2 quick fixes available:

- Move 'MiamiHeat.java' to package 'nba.annoyingTeams.east'
- Change package declaration to 'nba.annoyingTeams'

PACKAGES


java.lang
Object.java
String.java
Math.java
System.java
java.util
Scanner.java
Arrays.java
ArrayList.java
nba.annoyingTeams
MiamiHeat.java
animals
Cat.java
Dog.java

USING A CLASS IN YOUR PROGRAM

If you want to use a *package member* from *outside* its package, you must instruct your program where to find that class. You can do this in 3 ways:

1. Specify the entire path whenever you use such class.
   For example, whenever you want to use `Dog` from the `animals` package you can *fully qualify* the class name: `animals.Dog`

```
animals.Dog myDog = new animals.Dog();
```

Ok for infrequent use!

USING A CLASS IN YOUR PROGRAM

If you want to use a *package member* from *outside* its package, you must instruct your program where to find that class. You can do this in 3 ways:

2. Import the package member. Example:

```
import animals.Dog;
```

This tells the computer that the class `Dog` is found in the package `animals`.

Ok if you use few members from a package.

USING A CLASS IN YOUR PROGRAM

If you want to use a *package member* from *outside* its package, you must instruct your program where to find that class. You can do this in 3 ways:

3. Import the entire package. Example:

```
import animals.*;
```

Now you can refer to any class inside the `animals` package.

USING A CLASS IN YOUR PROGRAM

For convenience, the Java compiler automatically imports two entire packages for each source file:

1. The `java.lang` package
2. The *current* package

This is why no import statement is need to use `Math`, `String`, … , or any *package member* from inside its own package.

OBJECTS and CLASSES

CLASSES

By now, we should all know that objects and classes are closely related. How exactly?

- Each time we define a **class** we create a new **object type** with the same name.
- A class is a blueprint/template for a type of object. It specifies what properties the objects have and what methods can operate on them.
- An object is an **instance** of some class.

THE BLUEPRINT

```
public class ClassName {
   // some data declared here
   <modifier> <type> <variable_name>;

   public ClassName() {
      //constructor
   }

   // declare other methods
}
```

Attributes/Fields

Methods to create an object

Other methods

File name: **ClassName**.java

DEFAULT CONSTRUCTOR

If you don’t write a constructor, the default constructor for a class looks like:

```
public ClassName() {
}
```

If you write your own constructor, you no longer have access to the default constructor.

NESTED CLASSES

- You can define a class *within* another class. We call such class a *nested class*.
  We refer to the class containing a nested class as the *outer class*.

- Why?
  - To group classes that are used only in one place.
    If a class is useful to only one class, it makes sense to keep it nested and together.
  - Increase encapsulation.
    Allows for better control over data.
  - Create readable and maintainable code.

MODIFIERS

MODIFIERS

Modifiers are **keyword** that you add to class/method/variable's definition to change their meaning. Java has different kind of modifiers, including:

Access Control Modifiers

- `public`
- `protected`
- *default* (no keyword)
- `private`

Non-Access Modifiers

- `static`
- `final`
- `abstract`

VISIBILITY/ACCESS CONTROL MODIFIERS

Note:
- outer classes can only be declared `public` or *package private*.
- members of a class (fields, methods, classes) can be declared using any of the access modifiers.

- public
- protected (= package + subclasses)
- default (= package)
- private

These modifiers define what is visible across classes.

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	Y/N	N
`private`	Y	N	N	N

EXAMPLES – OUTER CLASS

Dog.java

```
package animals;

public class Dog {
   ⋮
}
```

Farm.java

```
package buildings;
import animals.Dog;

public class Farm {

   Dog d;

   ⋮
}
```

Does the compiler allow this?

- Yes

EXAMPLES – OUTER CLASS

Dog.java

```
package animals;

class Dog {
   ⋮
}
```

Farm.java

```
package buildings;
import animals.Dog;

public class Farm {

   Dog d;

   ⋮
}
```

Does the compiler allow this?

- No, the class `Dog` is visible only within its package!

EXAMPLES – FIELDS

Dog.java

```
package animals;

public class Dog {
   public String name;
   ⋮
}
```

Farm.java

```
package buildings;
import animals.Dog;

public class Farm {
   Dog d;

   Farm() {
      d = new Dog();
      d.name = “Jessie”;
   }
}
```

Does the compiler allow this?

➢ Yes (but remember, as a general rule fields should be declared private)

EXAMPLES – FIELDS

Dog.java

```
package animals;

public class Dog {
   String name;
   ⋮
}
```

Beagle.java

```
package animals;

public class Beagle {
   Dog d;

   Beagle() {
      d = new Dog();
      d.name = "Buddy";
   }
}
```

Does the compiler allow this?

- **Yes, the field** `name` **is visible within the package** `animals`**.**

EXAMPLES – FIELDS

Dog.java

```
package animals;

public class Dog {
   private String name;
   ⋮
}
```

Beagle.java

```
package animals;

public class Beagle {
   Dog d;

   Beagle() {
      d = new Dog();
      d.name = “Buddy”;
   }
}
```

Does the compiler allow this?

- No, `name` is visible only within the class `Dog`.

ENCAPSULATION

- Process of wrapping data and the code acting on that data in one unit. The idea is to better control the data.

- What to do?
  - Make all the fields `private`
  - Provide getters and setters as needed.

- Note: through the methods we can do data validation, while we have little control over the data stored in a `public` field.

NON-ACCESS MODIFIERS

- **`static`**

  Fields, methods, and nested classes can be declared to be `static`.

When a class member is declared to be static, then it “belongs” to the entire class and not to a specific instance (object).

- **`final`**

  Variables, methods, and classes can be declared to be `final`.

- **`abstract`**

  Methods and classes can be declared to be `abstract`.

STATIC

- We can define an field or a method to be `static` if we want it to be independent from one specific instance of the class.

- A static method/field is associated *with the entire class*
  Static fields are also called ***class variables***.

- A non-static method/field belongs to *an instance of the class*
  Non-static fields are also called ***instance variables***.

STATIC VS NON-STATIC

```
String s = “hippos”;
String t = “elephants”;
boolean b = (s.length() == t.length());
```

`length()` **is non-static method. Its execution depends on a specific string.**

STATIC VS NON-STATIC

```
double x = Math.PI;
int y = Integer.parseInt(“1”);
```

- `PI` **is a static field. It belongs to the** `Math` **class.**
- `parseInt()` **is a static method. It belongs to the** `Integer` **class and does not depend on a specific object of type** `Integer`**.**

FINAL VARIABLES

If a variable is declared to be `final`, its value can ***never*** be changed after is has been assigned.

```
final int x = 3;
x = 10; // compile-time error!
```

```
final Cat myCat = new Cat(“Small cat”);
myCat = new Cat(“Tequila”); // compile-time error!
```

MUTABLE REFERENCE TYPES

```
final Cat myCat = new Cat(“Small cat”);
myCat = new Cat(“Tequila”); // compile-time error!
```

However, you can still change the object that `myCat` points at, without changing `myCat`’s value.

```
myCat.setName(“Tequila”); // no problem!
```

FINAL FIELDS

- Final fields must be initialized!
  (Otherwise **compile-time error**)
  - If the class has a final instance variable (i.e., a final non-static field), you must initialize it in *every* constructor!
  - If the class has a final class variable (i.e. a final static field), you should initialize it in place (on the same line of the declaration) or in a Static Initializer Block (we might talk about this in the future).

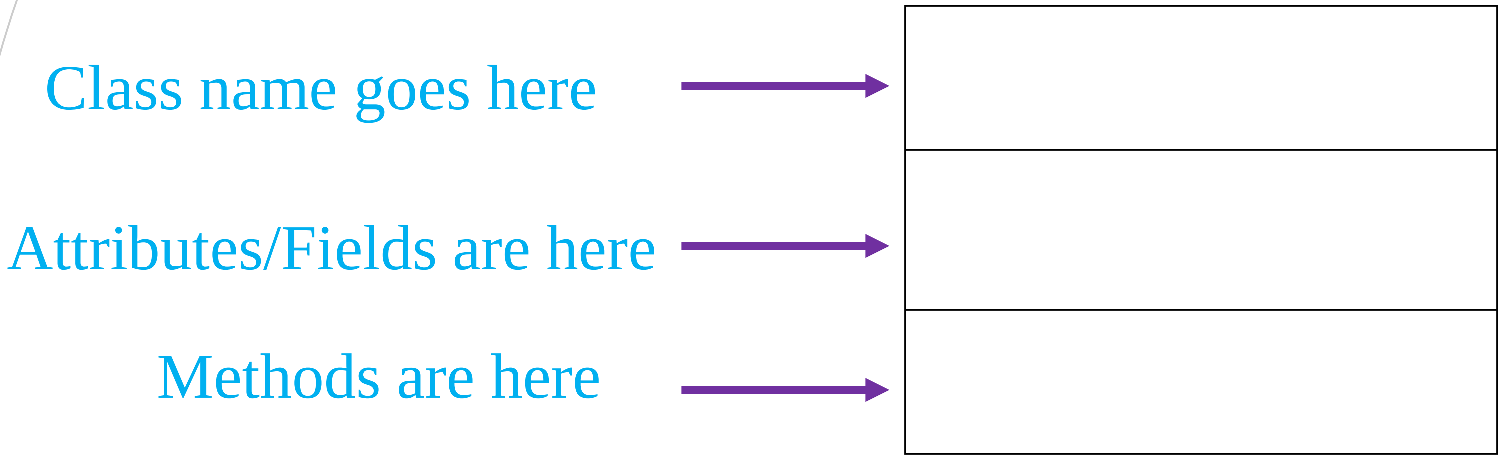
TO LOOK FORWARD TO

- After we learn about Inheritance, we will discuss what it means for a method or a class to be declared as `final`.
- In a week, we will also learn about `abstract` classes and methods.

UML DIAGRAMS

UML DIAGRAMS

Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.



EXAMPLE – DOG CLASS

- Fields/Attributes
  - String name
  - Person owner
- Constructors
  - Dog(String name)
  - Dog(String name, Person owner)
- Accessors and Mutators
  - getName
  - getOwner
  - setName
  - setOwner
- Other Methods
  - eat()
  - bark()
  - hunt()

DOG CLASS: + MEANS PUBLIC, - MEANS PRIVATE

```
Dog
---------------------------------------------
-   name : String
-   owner : Person
---------------------------------------------
<< constructors >>
+   Dog(name: String)
+   Dog(name: String, owner: Person)
<<accessors>>
+   getName() : String
+   getOwner() : Person
<<mutators>>
+   setName(String name)
+   setOwner(Person owner)
<<custom methods>>
+   eat()
+   bark(int numOfTimes)
+   hunt(): Rabbit
```

UNDERLINE IF FIELD/METHOD IS STATIC

```
                     Dog
-   name : String
-   owner : Person
-   numOfDogs: int
<< constructors >>
+   Dog(name: String)
+   Dog(name: String, owner: Person)
<<accessors>>
+   getName() : String
+   getOwner() : Person
+   getNumOfDogs(): int
<<mutators>>
+   setName(String name)
+   setOwner(Person owner)
<<custom methods>>
+   eat()
+   bark(int numOfTimes)
+   hunt(): Rabbit
```

IF TIME PERMITS

LOCAL VARIABLES VS FIELDS

How do they differ?

- Where to declare them:
  - Local variables are declared inside a method or a block
  - Fields (class and instance variables) are declared inside a class, but outside a method

LOCAL VARIABLES VS FIELDS

How do they differ?

- Scope:
  where can they be accessed (called directly using the variable name)
  - Local variables can be accessed only within the method or block in which they have been declared.
  - class variables be accessed from any method or block in that class
  - instance variables can be accessed from within the class or from non static methods of the class

LOCAL VARIABLES VS FIELDS

How do they differ?

- Access:
  - Local variables cannot have access modifiers. You can't access local variables from other classes or methods.
  - Field can have access modifiers. They can be accessed from methods within the class and from other classes if declared `public`.

http://edayan.info/java/fields-vs-variables-in-java

- Wednesday: Inheritance
- Friday: `Object` class and type conversion