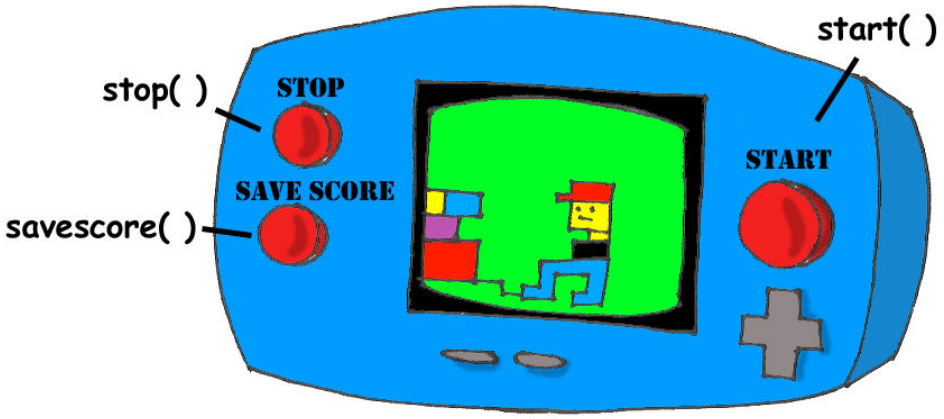
Chapter 3. Pet and Fish – Java Classes

Java programs consist of *classes* that represent objects from the real world. Even though people may have different preferences as to how to write programs, most of them agree that it's better to do it in a so-called *object-oriented* style. This means that good programmers start with deciding which objects have to be included in the program and which Java classes will represent them. Only after this part is done, they start writing Java code.

Classes and Objects

> Classes in Java may have *methods* and *attributes*.
>
> Methods define actions that a class can perform.
>
> Attributes describe the class.

Let's create and discuss a class named `VideoGame`. This class may have several methods, which can tell *what objects of this class can do*: start the game, stop it, save the score, and so on. This class also may have some attributes or properties: price, screen color, number of remote controls and others.



In Java language this class may look like this:

```
class VideoGame {
  String color;
  int price;

  void start () {
  }
  void stop () {
  }
  void saveScore(String playerName, int score) {
  }
}
```

Our class `VideoGame` should be similar to other classes that represent video games – all of them have screens of different size and color, all of them perform similar actions, and all of them cost money.

We can be more specific and create another Java class called `GameBoyAdvance`. It also belongs to the family of video games, but has some properties that are specific to the model GameBoy Advance, for example a cartridge type.

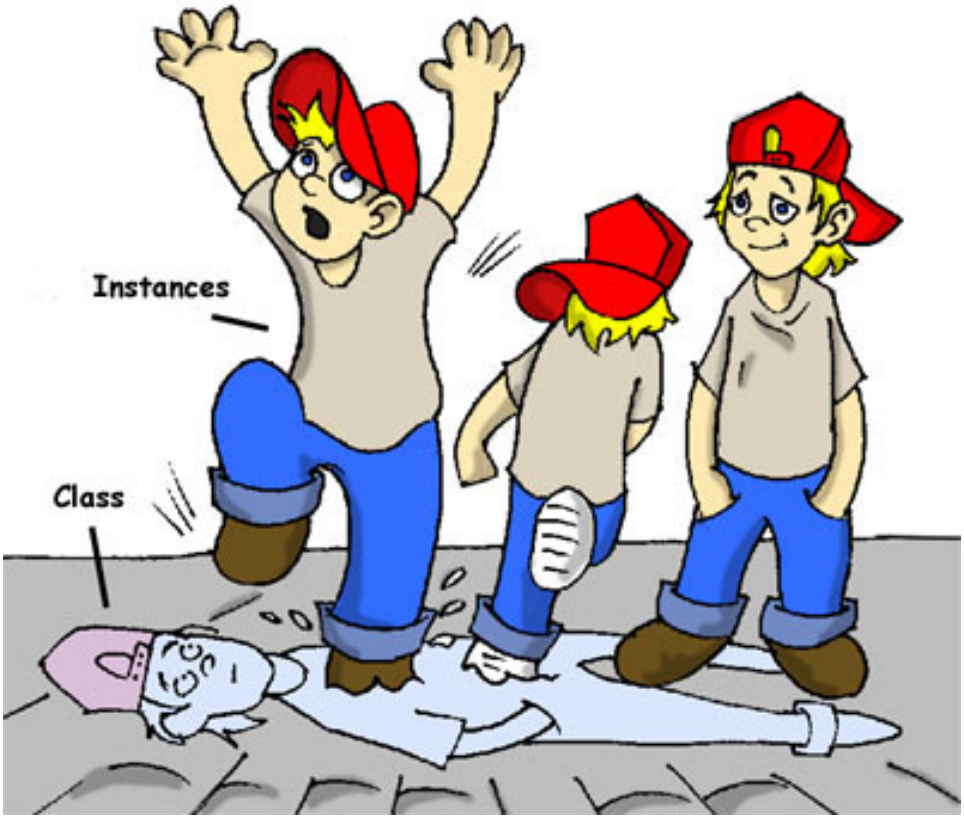
```
class GameBoyAdvance {
  String cartridgeType;
  int screenWidth;

  void startGame() {

  }
  void stopGame() {

  }
}
```

In this example the class `GameBoyAdvance` defines two attributes – `cartridgeType` and `screenWidth` and two methods – `startGame()` and `stopGame()`. But these methods can't perform

any actions just yet, because they have no Java code between the curly braces.

> In addition to the word *class*, you'll have to get used to the new meaning of the word *object*.
>
> The phrase "to create an instance of an object" means to create a copy of this object in the computer's memory according to the definition of its class.

A factory description of the GameBoy Advance relates to an actual game the same way as a Java class relates to its instance in memory. The process of building actual games based on this description in the game factory is similar to the process of creating instances of `GameBoy` objects in Java.



In many cases, a program can use a Java class only after its instance has been created. Vendors also create thousands of game copies based on the same description. Even though these copies represent the same class, they may have different *values* in their attributes - some of them are blue, while others are silver, and so on. In other words, a program may create *multiple instances* of the `GameBoyAdvance` objects.

Data Types

Java *variables* represent attributes of a class, method arguments or could be used inside the method for a short-time storage of some data. Variables have to be declared first, and only after this is done you can use them.

Remember equations like y=x+2? In Java you'd need to declare the variables x and y of some numeric *data type* like integer or `double`:

```
int x;
int y;
```

The next two lines show how you can assign a *value* to these variables. If your program assigns the value of five to the variable `x`, the variable `y` will be equal to seven:

```
x=5;
y=x+2;
```

In Java you are also allowed to change the value of a variable in a somewhat unusual way. The following two lines change the value of the variable `y` from five to six:

```
int y=5;
y++;
```

Despite the two plus signs, JVM is still going to increment the value of the variable `y by one`.

After the next code fragment the value of the variable `myScore` is also six:

```
int myScore=5;
myScore=myScore+1;
```

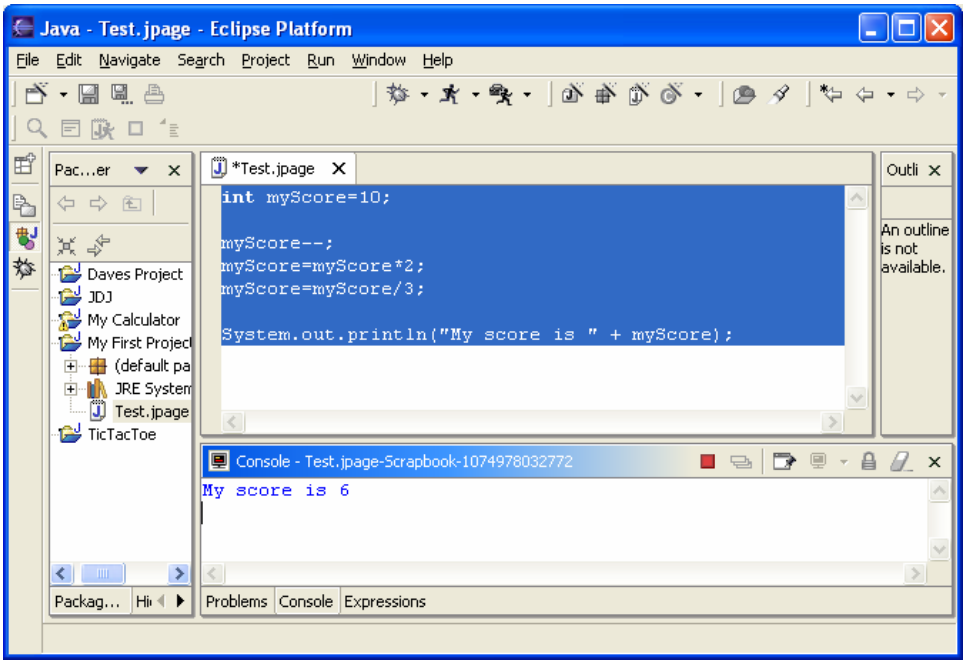
You can also use multiplication, division and subtraction the same way. Look at the following piece of code:

```
int myScore=10;

myScore--;
myScore=myScore*2;
myScore=myScore/3;

System.out.println("My score is " + myScore);
```

What this code prints? Eclipse has a cool feature called a scrapbook that allows quickly test any code snippet (like the one above) without even creating a class. Select menus *File*, *New*,

Scrapbook Page and type the word `Test` as the name of your scrapbook file.

Now enter these five lines that manipulate with `myScore` in the scrap book, highlight them, and click on the little looking glass on the toolbar.



To see the result of the score calculations, just click on the console tab at the bottom of the screen:

```
My score is 6
```

In this example the argument of the method `println()` was glued from two pieces – the text "My score is " and the value of the variable `myScore`, which was six. Creation of a `String` from pieces is called *concatenation*. Even though `myScore` is a number, Java is smart enough to convert this variable into a `String`, and then attach it to the text *My Score is*.

Look at some other ways of changing the values of the variables:

`myScore=myScore*2;` is the same as `myScore*=2;`
`myScore=myScore+2;` is the same as `myScore+=2;`
`myScore=myScore-2;` is the same as `myScore-=2;`
`myScore=myScore/2;` is the same as `myScore/=2;`

There are eight simple, or *primitive* data types in Java, and you have to decide which ones to use depending on the type and size of data that you are planning to store in your variables:

- ✓ Four data types for storing integer values – `byte`, `short`, `int`, and `long`.
- ✓ Two data types for values with a decimal point – `float` and `double`.
- ✓ One data type for storing a single character – `char`.
- ✓ One *logical* data type called `boolean` that allows only two values: `true` or `false`.

You can assign an initial value to a variable during its declaration and this is called *variable initialization*:

```
char grade = 'A';
int chairs = 12;
boolean playSound = false;
double nationalIncome = 23863494965745.78;
float gamePrice = 12.50f;
long totalCars =4637283648392l;
```

In the last two lines `f` means float and `l` means long.

If you don't initialize the variables, Java will do it for you by assigning zero to each numeric variable, `false` to `boolean` variables, and a special code '\u0000' to a `char`.

There is also a special keyword `final`, and if it's used in a variable declaration, you can assign a value to this variable only once, and this value cannot be changed afterwards. In some languages the final variables are called *constants*. In Java we usually name final variables using capital letters:

```
final String STATE_CAPITAL="Washington";
```

In addition to primitive data types, you can also use Java classes to declare variables. Each primitive data type has a corresponding *wrapper* class, for example `Integer`, `Double`, `Boolean`, etc. These classes have useful methods to convert data from one type to another.

While a `char` data type is used to store only one character, Java also has a class `String` for working with a longer text, for example:

```
String lastName="Smith";
```

In Java, variable names can not start with a digit and can not contain spaces.

A *bit* is the smallest piece of data that can be stored in memory. It can hold either 1 or 0.

A `byte` consists or eight bits.

A `char` in Java occupies two bytes in memory.

An `int` and a `float` in Java take four bytes of memory.

Variables of `long` and `double` types use eight bytes each.

Numeric data types that use more bytes can store larger numbers.

1 kilobyte (KB) has 1024 bytes

1 megabyte (MB) has 1024 kilobytes

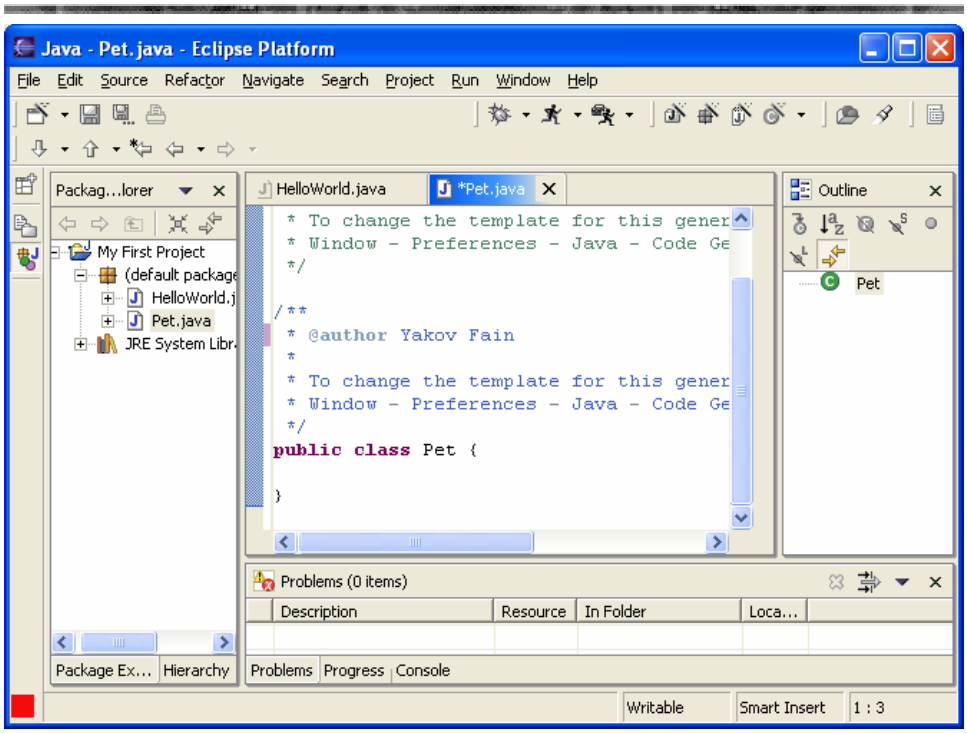
1 gigabyte (GB) has 1024 megabytes

Creation of a Pet

Let's design and create a class `Pet`. First we need to decide what actions our pet will be able to do. How about eat, sleep, and say? We'll program these actions in the methods of the class `Pet`. We'll also give our pet the following attributes: age, height, weight, and color.

Start with creating a new Java class called `Pet` in *My First Project* as described in Chapter 2, but do not mark the box for creation of the method `main()`.

Your screen should look similar to this one:



Now we are ready to declare attributes and methods in the class `Pet`. Java classes and methods enclose their bodies in curly braces. Every open curly brace must have a matching closing brace:

```
class Pet{
}
```

To declare variables for class attributes we should pick data types for them. I suggest an `int` type for the age, `float` for weight and height, and `String` for a pet's color.

```
class Pet{
   int age;
   float weight;
   float height;
   String color;
}
```

The next step is to add some methods to this class. Before declaring a method you should decide if it should take any arguments and return a value:

- ✓ The method `sleep()` will just print a message *Good night, see you tomorrow* – it does not need any arguments and will not return any value.

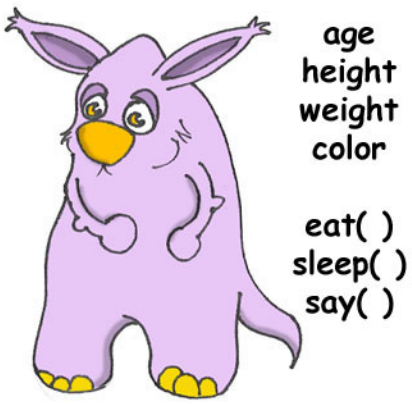
- ✓ The same is true for the method `eat()`. It will print the message *I'm so hungry…let me have a snack like nachos!*.

- ✓ The method `say()` will also print a message, but the pet will "say" (print) the word or a phrase that we give to it. We'll pass this word to the method `say()` as a *method argument*. The method will build a phrase using this argument and will return it back to the calling program.

The new version of the class `Pet` will look like this:

```
public class Pet {
 int age;
 float weight;
 float height;
 String color;

 public void sleep(){
        System.out.println(
                    "Good night, see you tomorrow");
 }

 public void eat(){
  System.out.println(
  "I'm so hungry…let me have a snack like nachos!");
 }

 public String say(String aWord){
      String petResponse = "OK!! OK!! " +aWord;
            return petResponse;
 }
}
```

This class represents a friendly creature from the real world:



Let's talk now about the signature of the method `sleep()`:

```
public void sleep()
```

It tells us that this method can be called from any other Java class (`public`), it does not return any data (`void`). The empty parentheses mean that this method does not have any arguments, because it does not need any data from the outside world – it always prints the same text.

The signature of the method `say()` looks like this:

```
public String say(String aWord)
```

This method can also be called from any other Java class, but has to return some text, and this is the meaning of the keyword `String` in front of the method name. Besides, it expects some text data from outside, hence the argument `String aWord`.

How do you decide if a method should or should not return a value? If a method performs some data manipulations and has to give the result of these manipulations back to a calling class, it has to return a value. You may say, that the class `Pet` does not have any calling class! That's correct, so let's create one called `PetMaster`. This class will have a method `main()` containing the code to communicate with the class `Pet`. Just create another class `PetMaster`, and this time select the option in Eclipse that creates the method `main()`. Remember, without this method you can not *run* this class as a program. Modify the code generated by Eclipse to look like this:

```
public class PetMaster {

  public static void main(String[] args) {

   String petReaction;

   Pet myPet = new Pet();

   myPet.eat();
   petReaction = myPet.say("Tweet!! Tweet!!");
   System.out.println(petReaction);

   myPet.sleep();

  }
}
```

Do not forget to press *Ctrl-S* to save and compile this class!
To run the class `PetMaster` click on the Eclipse menus *Run*, *Run…*, *New* and type the name of the main class: `PetMaster`. Push the button *Run* and the program will print the following text:

```
I'm so hungry…let me have a snack like nachos!
OK!! OK!! Tweet!! Tweet!!
Good night, see you tomorrow
```

The `PetMaster` is the *calling class*, and it starts with creating an *instance* of the object `Pet`. It declares a variable `myPet` and uses the Java operator `new`:

```
Pet myPet = new Pet();
```

This line declares a variable of the type `Pet` (that's right, you can treat any classes created by you as new Java data types). Now the variable `myPet` knows where the `Pet` instance was created in the computer's memory, and you can use this variable to call any methods from the class `Pet`, for example:

```
myPet.eat();
```

If a method returns a value, you should call this method in a different way. Declare a variable that has the same type as the return value of the method, and assign it to this variable. Now you can call this method:

```
String petReaction;

petReaction = myPet.say("Tweet!! Tweet!!");
```

At this point the returned value is stored in the variable `petReaction` and if you want to see what's in there, be my guest:

```
System.out.println(petReaction);
```

Inheritance – a Fish is Also a Pet

Our class `Pet` will help us learn yet another important feature of Java called *inheritance*. In the real life, every person inherits some features from his or her parents. Similarly, in the Java world you can also create a new class, based on the existing one.

The class `Pet` has behavior and attributes that are shared by many pets – they eat, sleep, some of them make sounds, their skins have different colors, and so on. On the other hand, pets are different - dogs bark, fish swim and do not make sounds, parakeets talk better than dogs. But all of them eat, sleep, have weight and height. That's why it's easier to create a class `Fish` that will *inherit* some common behaviors and attributes from the class `Pet`, rather than creating `Dog`, `Parrot` or `Fish` from scratch every time.

A special keyword `extends` that will do the trick:

```
class Fish extends Pet{

}
```

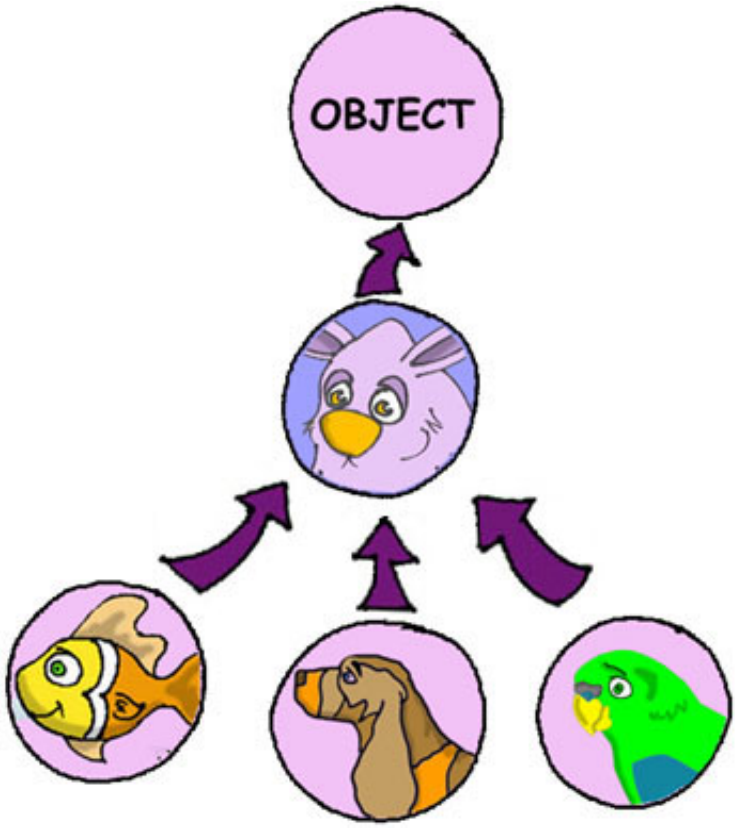
You can say that our `Fish` is a *subclass* of the class `Pet`, and the class `Pet` is a *superclass* of the class `Fish`. In other words, you use the class `Pet` as a template for creating a class `Fish`.

Even if you will leave the class `Fish` as it is now, you can still use every method and attribute inherited from the class `Pet`. Take a look:

```
Fish myLittleFish = new Fish();
myLittleFish.sleep();
```

Even though we have not declared any methods in the class `Fish` yet, we are allowed to call the method `sleep()` from its superclass!

Creation of subclasses in Eclipse is a piece of cake! Select the menus *File*, *New*, *Class*, and type `Fish` as the name of the class. Replace the `java.lang.Object` in the field superclass with the word `Pet`.



Let's not forget, however, that we're creating a subclass of a `Pet` to add some new features that only fish have, and reuse some of the code that we wrote for a general pet.

It's time to reveal a secret – all classes in Java are inherited from the super-duper class `Object`, regardless if you do use the word `extends` or not.

But Java classes can not have two separate parents. If this would happen with people, kids would not be subclasses of their parents, but all the boys would descendents of Adam, and all the girls descendents of Eve ☺.

Not all pets can dive, but fish certainly can. Let's add a new method `dive()` to the class `Fish` now.

```
public class Fish extends Pet {

   int currentDepth=0;

   public int dive(int howDeep){
      currentDepth=currentDepth + howDeep;
      System.out.println("Diving for " + howDeep +
                                              " feet");
      System.out.println("I'm at " + currentDepth +
                           " feet below sea level");
      return currentDepth;
   }
}
```

The method `dive()` has an *argument* `howDeep` that tells the fish how deep it should go. We've also declared a class variable `currentDepth` that will store and update the current depth every time you call the method `dive()`. This method returns the current value of the variable `currenDepth` to the calling class.

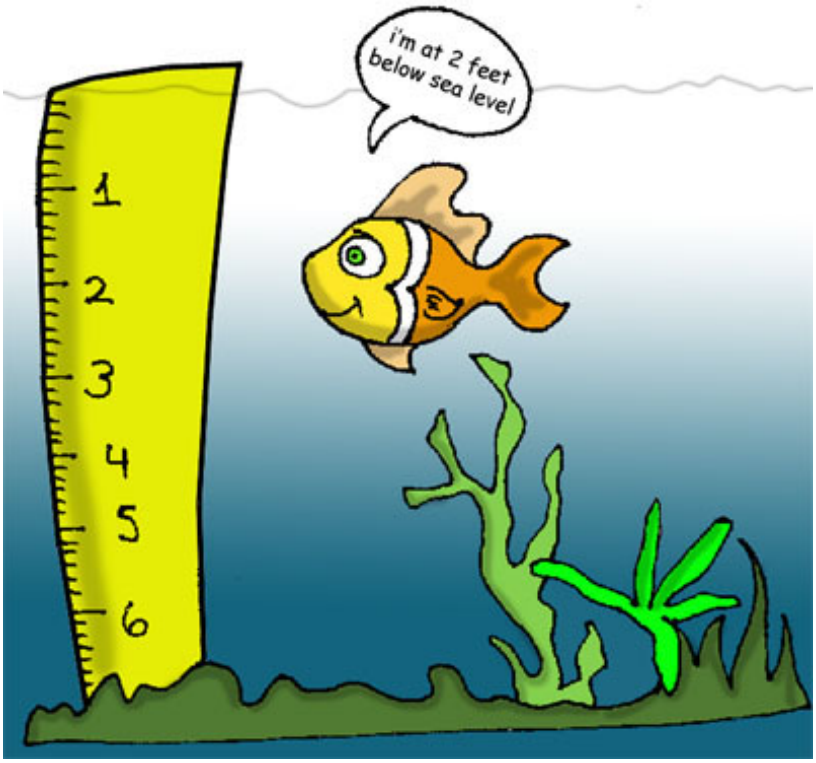
Please create another class `FishMaster` that will look like this:

```
public class FishMaster {

      public static void main(String[] args) {

            Fish myFish = new Fish();

            myFish.dive(2);
            myFish.dive(3);

            myFish.sleep();
      }
}
```

The method `main()` instantiates the object Fish and calls its method `dive()` twice with different arguments. After that, it calls the method `sleep()`. When you run the program `FishMaster`, it will print the following messages:

```
Diving for 2 feet
I'm at 2 feet below sea level
Diving for 3 feet
I'm at 5 feet below sea level
Good night, see you tomorrow
```

Have you noticed that beside methods defined in the class `Fish`, the `FishMaster` also calls methods from its superclass `Pet`? That's the whole point of inheritance – you do not have to copy and paste code from the class `Pet` – just use the word `extends`, and the class `Fish` can use `Pet's` methods!



One more thing, even though the method `dive()` returns the value of `currentDepth`, our `FishMaster` does not use it. That's fine, our `FishMaster` does not need this value, but there may be some other classes that will also use `Fish`, and they may find it useful. For example, think of a class `FishTrafficDispatcher` that has to know positions of other fish under the sea before allowing diving to avoid traffic accidents ☺.

Method Overriding

As you know, fish do not speak (at least they do not do it aloud). But our class `Fish` has been inherited from the class `Pet` that has a method `say()`. This means that nothing stops you from writing something like this:

```
myFish.say();
```

Well, our fish started to talk… If you do not want this to happen, the class `Fish` has to *override* the Pet's method `say()`. This is how it works: if you declare in a subclass a method with exactly the same signature as in its superclass, the method of the subclass will be used instead of the method of the superclass. Let's add the method `say()` to the class `Fish`.

```
public String say(String something){
  return "Don't you know that fish do not talk?";
}
```

Now add the following method call to the method `main()` of the class `FishMaster`:

```
myFish.say("Hello");
```

Run the program and it'll print

```
Don't you know that fish do not talk?
```

This proves that `Pet`'s method `say()` has been *overridden*, or in other words suppressed.

If a method signature includes the keyword `final`, such method can not be overridden, for example:

```
final public void sleep(){…}
```

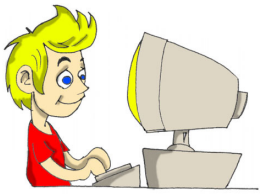
Wow! We've learned a lot in this chapter – let's just take a break.

Additional Reading

1.Java Data Types:
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

2.About inheritance:
http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html

Practice



1. Create a new class `Car` with the following methods:

```
public void start()
public void stop()
public int drive(int howlong)
```

The method `drive()` has to return the total distance driven by the car for the specified time. Use the following formula to calculate the distance:

```
distance = howlong*60;
```

2. Write another class `CarOwner` and that creates an instance of the object `Car` and call its methods. The result of each method call has to be printed using `System.out.println()`.