Writing a Java Program (2/2)

** using our new programming skills to write a real program (and learning some new ones on the way!)

** **`ArrayList`**

** Java API (brief introduction)

Chapter 6 (*) – “Head First Java” book

Some slides contain lots of animation.

(*) *Example from the chapter*

Using arrays and problems with arrays

- Fixed size:
  - Arrays cannot grow and shrink in size.
- Not easy to change the order of elements:
  - Difficult to insert or remove elements.

```
Rabbit[] racers = new Rabbit[2];
Rabbit r1 = new Rabbit();
//r1 set up
racers[0] = r1;

Rabbit r2 = new Rabbit();
//r2 set up
racers[1] = r2;
```

1. Change the size to **3**

How to add one more **Rabbit r3** to the index 1?

2. Add **r2** to index 2: **racers[2] = r2;**

3. Add **r3** in index 1: **racers[1] = r3;**

How to swap **r2** and **r3**?
We need a temp place …

The operations are difficult and every time we need to modify the code!

Some `ArrayList` methods (1/3)

Assume we already have a **`Flower`** class:

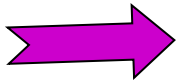
1. Make a list of **`Flower`**s:

```
ArrayList<Flower> myList = new ArrayList<Flower>();
```

A new **`ArrayList`** object on the heap.
It is little because it is empty.

2. Put something in it:

```
Flower f = new Flower();
myList.add(f);
```

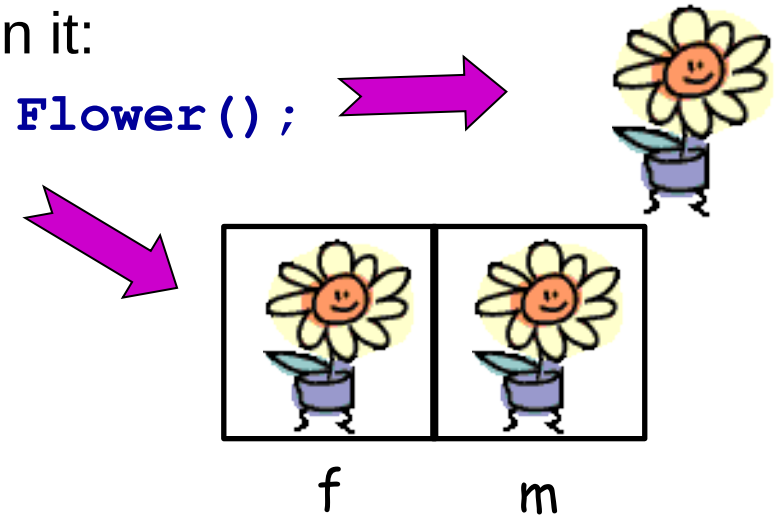


f

Some `ArrayList` methods (2/3)

3. Put something else in it:

```
Flower m = new Flower();
myList.add(m);
```



4. Find out how many things are in it:

```
int size = myList.size();
```

2

5. Find out if it contains something:

```
boolean inIt = myList.contains(f);
```

true

Some `ArrayList` methods (3/3)

6. Find out where in the list something is:

```
int index = myList.indexOf(f);
```

0

`ArrayList`s are still zero-based (just like arrays)!

7. Find out if the list is empty:

```
boolean empty = myList.isEmpty();
```

false

8. Remove something from the list:

```
myList.remove(f);
```

m

array *versus* ArrayList (1/2)

An array needs to know its size at time of creation, whereas an `ArrayList` does not:


```
new String[6];
```

```
new ArrayList<E>();
```

To assign an object in a regular array, you must assign it to a specific index.


```
myList[4] = b;
```

```
myArrayList.add(b);
```

array *versus* ArrayList (2/2)

3. Arrays use array syntax (**[ ]**) that is not used anywhere else in Java. **ArrayList**s use standard dot notation:

   a. `myList[4];`

   b. `myArrayList.get(4);`

4. **ArrayList**s are parameterised.

   b. Parametrised types were introduced in Java 5.0.

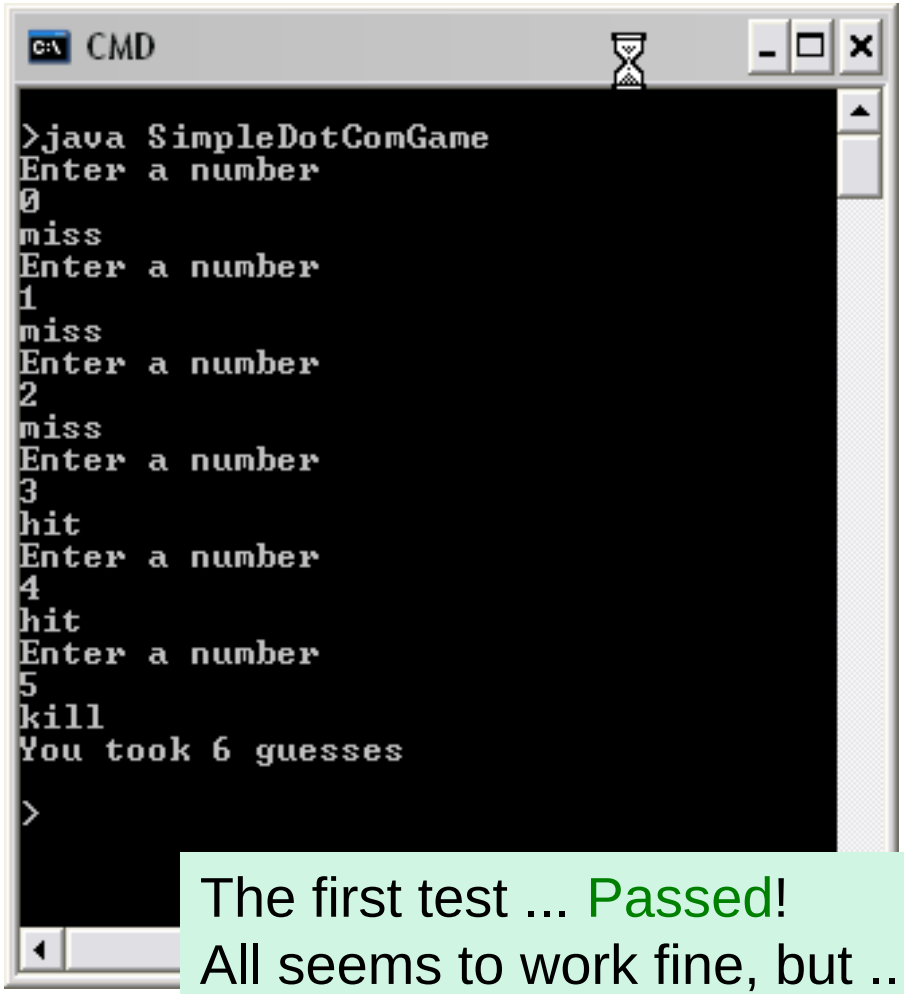
   ArrayList<String>

   The **< >** indicate the type of **ArrayList**. This list is a list of **String**s, as opposed to **ArrayList<Rabbit>** which would be a list of **Rabbit**s (and only **Rabbit**s!).

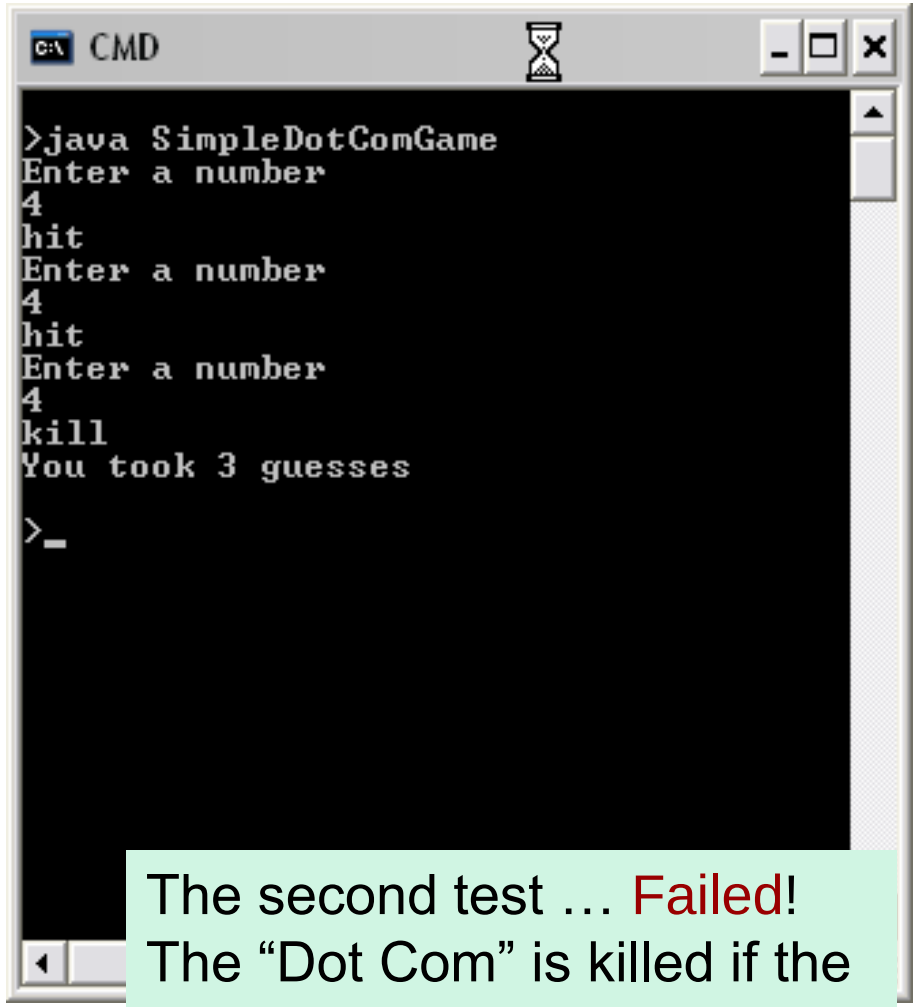
… and things for you to try out!

“Sink a Dot Com” example [Revision]

- Let us come back to the game …



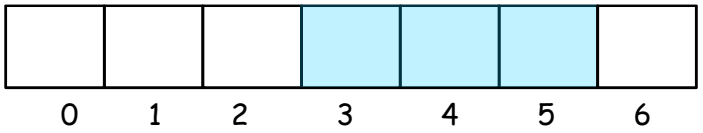
The first test ... Passed!
All seems to work fine, but ...



The second test … Failed!
The “Dot Com” is killed if the same cell is used 3 times …

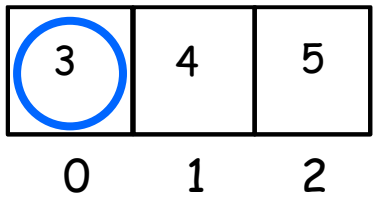
How do we fix the problem? (1/2)

- Our virtual row with the 3-cell **`SimpleDotCom`** object:



- Remember: our program actually finds out where the "Dot Com" is, by asking it. The "Dot Com" knows where it is, by using its **`cellLocation`** array:

```
int numberOfHits = 0
int[] cellLocation =
```



instance variables

---

player makes a guess ...

`guess = 4?` ⇒ check the entire array …

check at position 0 …

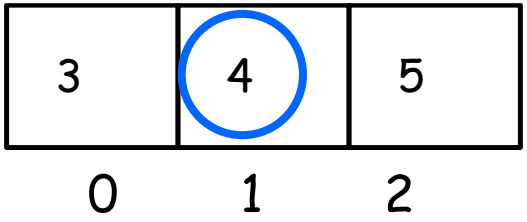
`cellLocation[0] = 3` ⇒ no match!

How do we fix the problem? (2/2)

instance variables

```
int numberOfHits = 1
```

```
int[] cellLocation =
```



`guess = 4?`

check at position 1 … `cellLocation[1] = 4` ⇒ match!

If the user guessed **`4`** again, the interaction would be repeated. The **`numberOfHits`** gets incremented, even if the player has hit there before...
To find the bug, lets look at variable **`numberOfHits`**.

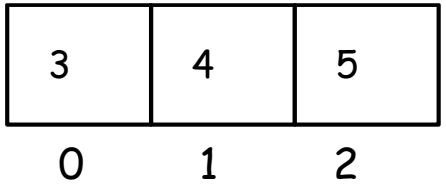
So what happened?

```
public String checkYourself(String stringGuess) {
  int guess = Integer.parseInt(stringGuess);
  String result = "miss";
  for (int cell : this.locationCells) {
    if (guess == cell) {
       result = "hit";
       this.numberOfHits++;
       break;
    }
  }
  if (this.numberOfHits == this.locationCells.length) {
    result = "kill";
  }
  System.out.println(result);
  return result;
}
```

We did not check to see if it is a different cell that was hit …

Option 1

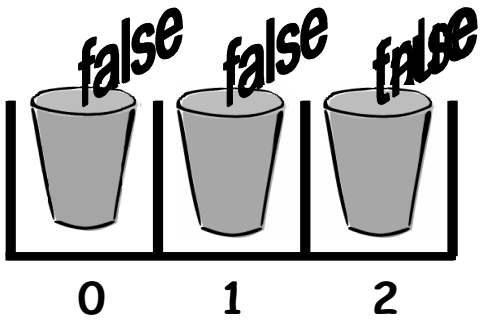
- Make a second **`boolean`** array called **`hitsArray`**.
  - Initialise all locations to **`false`**.
  - Each time a user makes a hit, change the respective location to **`true`**.



Check if location is in array.

Check if it has been hit before

If it hasn't been hit: update and return hit/kill.

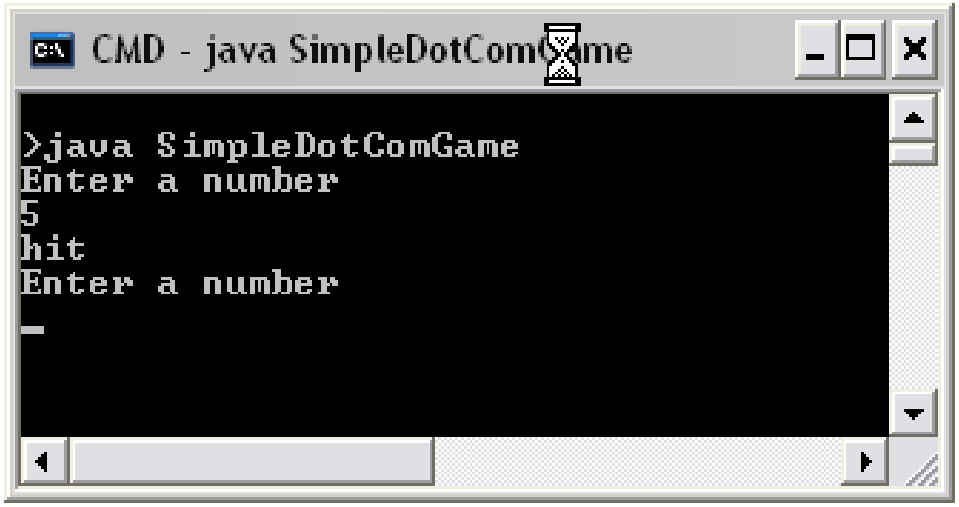


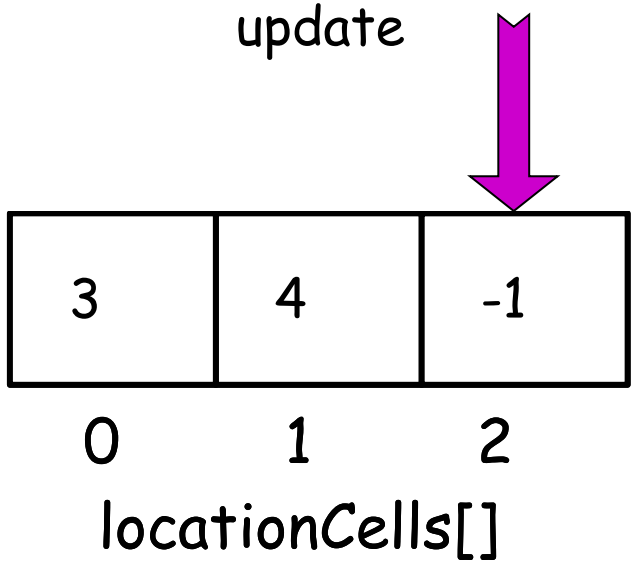
boolean[] hitsArray = new boolean[2]

Option 1 … too 'clunky'! Option 2?

- Option 1 is quite a bit of work!
- Have to check this, check that, update this, etc, etc …

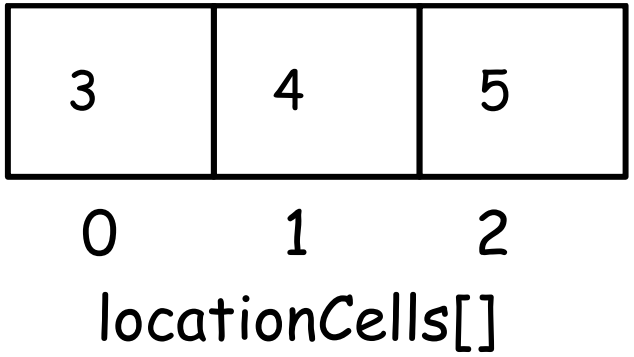
- Option 2:
  - Keep the array as it is, but change the value stored there, if it gets hit!

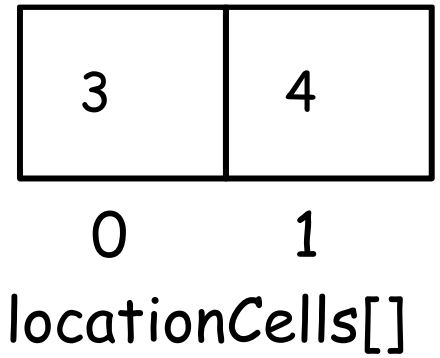




Option 2 … still a bit ‘clunky’! Option 3?

- Option 3:
  - Delete each cell location as it gets hit, and modify the array to be smaller.

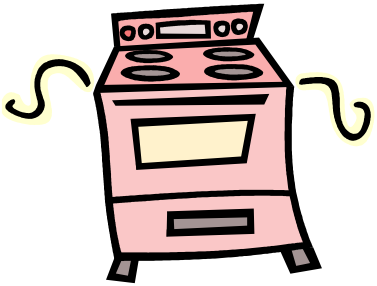




- Standard arrays cannot grow and shrink in size.
- So to do this, we must create a new smaller array when the “Dot Com” is hit and reassign it to the instance variable **`locationCells[]`**.
- This option would be more appealing if arrays could grow and shrink …

Welcome to the world of the core Java library (or API)

- There is indeed such a thing and it is called an **`ArrayList`**.
  - Just like our "ready-baked" code from our first "Dot Com" game, the API comes with hundreds of pre-built classes.
  - Unlike our "ready-baked" code, these classes are already compiled – just waiting for you to use them!



```
import java.util.ArrayList;

public class DotCom {
  private ArrayList<String> locationCells;
  // private int numberOfHits = 0;  =>  Don't need this!
  public void setLocationCells(ArrayList<String> loc) {
    locationCells = loc;
    // rest of code
}
```

New and improved `checkYourself()`

```
public String checkYourself(String stringGuess) {
  String result = "miss";
  int index = this.locationCells.indexOf(stringGuess);
  if (index >= 0) {
    this.locationCells.remove(stringGuess);
    // or this.locationCells.remove(index);
    if (locationCells.isEmpty()) {
      result = "kill";
    }
    else {
      result = "hit";
    }
  }
  System.out.println(result);
  return result;
}
```

Changes to `main()`

```
ArrayList<String> locations = new ArrayList<String>();
locations.add(""+randomNum);
locations.add(""+(randomNum + 1));
locations.add(""+(randomNum + 2));
```

if randomNum = 3, then locations = [3,4,5]

- Do we need the brackets? What if we do …

```
ArrayList<String> locations = new ArrayList<String>();
locations.add(""+randomNum);
locations.add(""+randomNum + 1);
locations.add(""+randomNum + 2);
```

if randomNum = 3, then locations = [3,31,32]

… and things for you to try out!

Building the real "Sink a Dot Com" (Recap only)

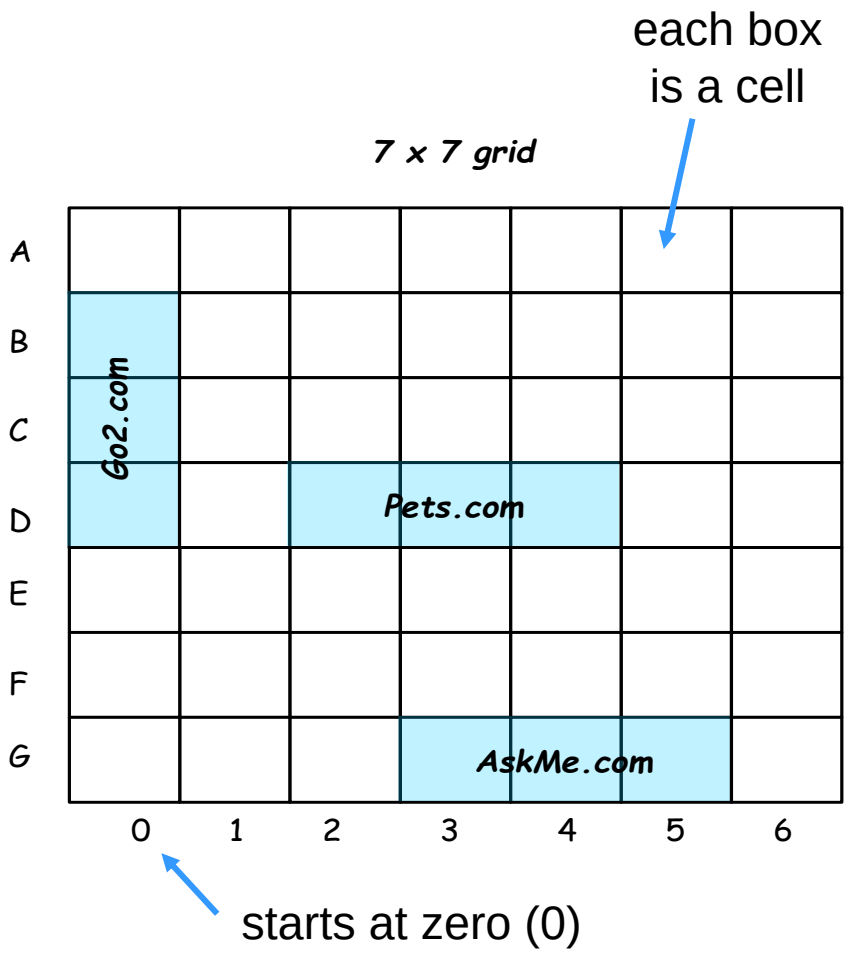
- We have been writing the simple version of the "Dot Com" game.
- Now we have to build the whole thing!

GOAL

- Sink all the computer's "Dot Coms" in the fewest number of guesses.
- You are given a rating, based on how well you perform.

SETUP

- A virtual 7x7 board with 3 randomly placed "Dot Coms".
- After that, the player should be prompted to enter their first guess.



What needs to change?

1. **DotCom** class
   - a. Needs a name variable!
     Remember that the dot com needs to be able to print its name after being killed! (Ouch! You sunk Pets.com ☹)

2. **DotComBust** class (the game)
   - a. Need 3 **DotComs** instead of 1!
   - b. Give each of the **DotCom**s a name when created! Need to use a *setter* to do it!
   - c. Put the **DotCom**s on a grid, rather than a single row. This is a bit complicated so:

to the rescue! Put this in the mysterious **GameHelper** class.

   - d. Check the user's guess with ALL 3 of the **DotCom**s!
   - e. Keep playing until ALL 3 **DotCom**s are killed!
   - f. Get out of **main()**.

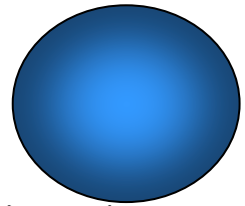
The classes

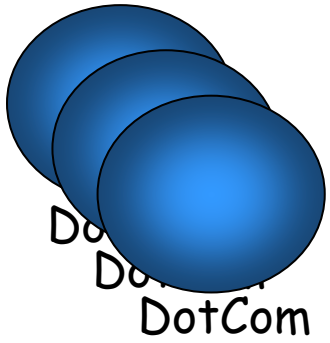
DotCom	DotComBust	GameHelper
The actual DotCom objects. DotComs know their *name*, *location* and how to check a user guess for a match.	**The game class.** Makes DotComs and gets user input, plays until all DotComs are dead	**The helper class.** *Ready bake...* Can accept user command-line input and make DotCom locations

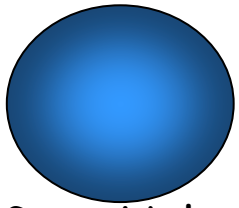
creates and plays with

used for player input and `DotCom` locations

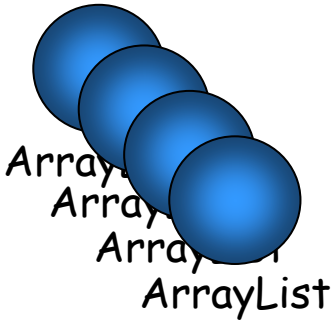
… and the objects







plus 4 ready-baked objects. Instances of ArrayLists are objects too!



One for DotComBust

One for each instance of DotCom (*three*)

Read also the information in QMplus, under
Teaching Week 2 > 'Sink a Dot Com' application (improved code).

Using the Java Library (API)

- To use a class in the API, you need to know what package the class is in.
  - For example, to use an **`ArrayList`**, you need to know that **`ArrayList`**s belong to the **`java.util`** package.
  - The **`java.util`** package contains other utility classes as well!
  - Using a class from an API is just like using our "ready-baked" code – only we don't even have to compile it!
- We have already been using the API without realising it!
- The **`java.lang`** package is automatically included in every class; other examples:

```
Math.random();          System.out.println();
```

- However, to use other packages you need to know the full name of the class you want to use in your code:
  - **`java.util.ArrayList`**

package name class name

Using `ArrayList` (or other classes)

Two approaches:

IMPORT ← More common

Put an **`import`** statement at the top of your source code file.

```
import java.util.ArrayList;
```

TYPE

Type the full name everywhere in your code.

Each time you use it. *Anywhere* you use it.

```
java.util.ArrayList<Rabbit> list =
                    new java.util.ArrayList<Rabbit>();
```

JDK Class Library

- ↑ Package familiarity == ↑ Your programming skills
- Half the 'battle' is knowing what class to use, and when …
- Main packages:

  `java.lang` -- Provides classes that are fundamental to the design of the Java programming language.

  `java.io` -- Provides for system input and output through data streams, serialization and the file system.

  `java.awt` -- Contains all of the classes for creating user interfaces and for painting graphics and images.

- How to learn about the API:
  - Use a reference book
  - Use the HTML API docs online