Programming 1

Further Java

Lecture #3: Collections

The story so far

- If a variable is like a box an array is like a box with compartments
- The compartments in arrays are numbered (the *element* number, starting at 0) and they have a fixed length
- Once the length is set it cannot change
- But consider the following scenario:

```
String[] clubMembers = new String[5];
clubMembers[0] = "Paul";
clubMembers[1] = "Fred";
clubMembers[2] = "Janet";
clubMembers[3] = "Susan";
clubMembers[4] = "Bill";
```

- What happens if someone new wants to join our awesome club?

Arrays are inflexible

```
String[] clubMembers = new String[5];
clubMembers[0] = "Paul";
clubMembers[1] = "Fred";
clubMembers[2] = "Janet";
clubMembers[3] = "Susan";
clubMembers[4] = "Bill";
```

- ...and then later Lloyd joins the club

```
clubMembers[5] = "Lloyd";
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
```

- ...seems the club is closed for new members ☹

Arrays are inflexible

- Arrays have a fixed length – this is rubbish if you don't know the length of your list at the outset, or you want to dynamically change the length
  - What if someone leaves the club?
  - What if you want to add someone in the middle of the list?
- Arrays are indexed by number, what about if you want to index by something else? For example…

K number	Student name
k123123	Walter White
k142121	Jesse Pinkman
k153234	Saul Goodman

- What about if you want to ensure that things in your list are unique?
  - Student K numbers are unique…

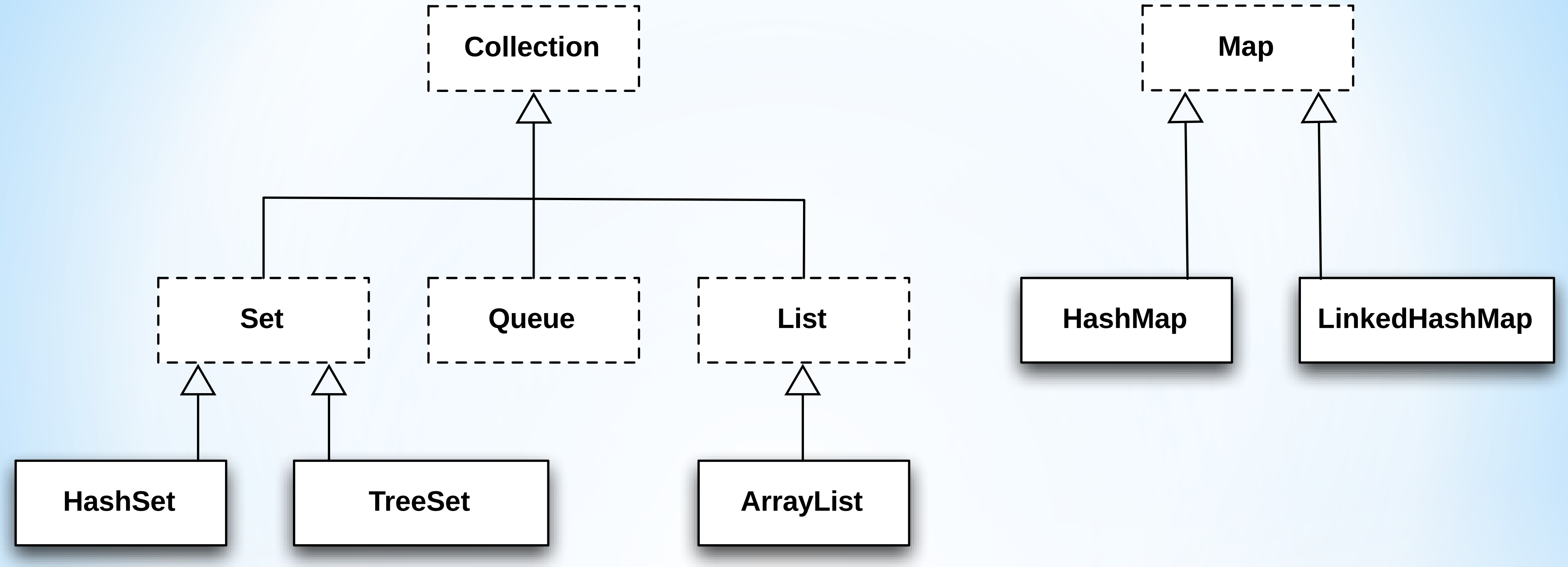
Enter *collections*

- A collection is a Java object that can contain objects
- Think of them as arrays on steroids – a variation of the box with compartments, but with more flexibility
  - You can have a box with numbered compartments that can grow and shrink as needed (a list)
  - You can have a box that stores only unique items (a set)
  - You can have a box that stores items with (unique) keys (a map)

The Java collections framework

- The collections framework consists of
  - Interfaces
    - Abstract classes representing various top-level types of collections, e.g. Set, Map, List
  - Implementations
    - Concrete classes that YOU can use as a Java programmer that *implement* the interfaces, e.g. ArrayList, HashMap, HashSet
  - Utility methods
    - Provide useful functions like searching and sorting

The core Collections interfaces



- There are many more collections – we'll look at the "good" ones (shown)
  - If you want to find out more, check out the Java API docs online
- All collections classes reside in the **java.util** package
- If you want to make use of a given class, you will need to import it!

Basic methods on all collections

- All collections have a basic set of methods:

`add`	adds an element to the collection
`contains`	checks if the specified element exists in the collection; return true if it does and false if not
`remove`	removes an element from the collection
`clear`	removes all elements
`size`	gives you the number of elements
`isEmpty`	return true if the collection is empty, false if it has some elements in

- Depending on which one you're using, there may be extra methods specific to the collection at hand

The **ArrayList**

(or "if you don't bother learning anything more about collections, at least learn about ArrayLists")

- The **ArrayList** is a list that works similarly to an array (clue's in the name!)
- You can use an ArrayList as an almost drop-in replacement for an array:

```
String[] clubMembers = new String[5];
clubMembers[0] = "Paul";
clubMembers[1] = "Fred";
clubMembers[2] = "Janet";
clubMembers[3] = "Susan";
clubMembers[4] = "Bill";

ArrayList<String> clubMembers = new ArrayList();
clubMembers.add("Paul");
clubMembers.add("Fred");
clubMembers.add("Janet");
clubMembers.add("Susan");
clubMembers.add("Bill");
```

Key points about ArrayLists

```
ArrayList<String> clubMembers = new ArrayList();
clubMembers.add("Paul");
clubMembers.add("Fred");
clubMembers.add("Janet");
clubMembers.add("Susan");
clubMembers.add("Bill");

System.out.println(clubMembers.get(2));
```

- You specify the type of data you are going to store in the ArrayList using a *generic*
- You do NOT need to specify the size of the ArrayList up front
- You add new items to the ArrayList using the add method
  - this will dynamically grow the ArrayList as needed
- You can get items from the ArrayList using the get method
  - the numeric parameter is like the array index
  - In this case, we'd get Janet (the first element is zero, just like an array)

Key points about ArrayLists

```
ArrayList<String> clubMembers = new ArrayList();
clubMembers.add("Paul");
clubMembers.add("Fred");
clubMembers.add("Janet");
clubMembers.add("Susan");
clubMembers.add("Bill");

System.out.println(clubMembers.get(2));
```

- You specify the type of data you are going to store in the ArrayList using a *generic*
- You do NOT need to specify the size of the ArrayList up front
- You add new items to the ArrayList using the add method
  - this will dynamically grow the ArrayList as needed
- You can get items from the ArrayList using the get method
  - the numeric parameter is like the array index
  - In this case, we'd get Janet (the first element is zero, just like an array)

Key points about ArrayLists

```
ArrayList<String> clubMembers = new ArrayList();
clubMembers.add("Paul");
clubMembers.add("Fred");
clubMembers.add( "Janet" );
clubMembers.add("Susan");
clubMembers.add("Bill");

System.out.println(clubMembers.get(2));
```

- You specify the type of data you are going to store in the ArrayList using a *generic*
- You do NOT need to specify the size of the ArrayList up front
- You add new items to the ArrayList using the add method
    - this will dynamically grow the ArrayList as needed
- You can get items from the ArrayList using the get method
    - the numeric parameter is like the array index
    - In this case, we'd get Janet (the first element is zero, just like an array)

More points about ArrayLists

- (assuming that our ArrayList **`clubMembers`** contained Paul, Fred, Janet, Susan and Bill)

  (0) (1) (2) (3) (4)
- We can remove things from the ArrayList using remove and removeRange:
  - clubMembers.remove("Fred");
    - Our members would be Paul, Janet, Susan and Bill
  - clubMembers.remove(3);
    - Our members would be Paul, Fred, Janet and Bill
  - clubMembers.removeRange(1,3);
    - Our members would be Paul, Susan and Bill
      - ...it's removed elements 1 (inclusive) to 3 (exclusive)
- The ArrayList would shrink accordingly as things were removed

Getting the ArrayList's size

- Use the **`size`** method to determine how many elements are in an ArrayList (or any collection, for that matter)

```
ArrayList<String> modules = new ArrayList();
modules.add("Programming 1");
modules.add("System Environments");
modules.add("Neutron Bomb Juggling");
System.out.println(modules.size());
```

Getting the ArrayList's size

- Use the **`size`** method to determine how many elements are in an ArrayList (or any collection, for that matter)

```
ArrayList<String> modules = new ArrayList();
modules.add("Programming 1");
modules.add("System Environments");
modules.add("Neutron Bomb Juggling");
System.out.println(modules.size());
```

- The result would be 3
- **NB: Note that as with arrays, because the first element is zero, the size of an ArrayList will always be one greater than the index number of the last element!**
- **NB #2: Note that size is a *method* on the class ArrayList. So it is followed by brackets!**

Inserting items at a specific point of the ArrayList

- You can add items into the list in the middle:

```
ArrayList<String> modules = new ArrayList();
modules.add("Programming 1");
modules.add("System Environments");
modules.add("Neutron Bomb Juggling");
modules.add(2,"Advanced Dave Baiting");
```

Inserting items at a specific point of the ArrayList

- You can add items into the list in the middle:

```
ArrayList<String> modules = new ArrayList();
modules.add("Programming 1");
modules.add("System Environments");
modules.add("Neutron Bomb Juggling");
modules.add(2,"Advanced Dave Baiting");
```

- The ArrayList would end up containing

Element	0	Programming 1
	1	System Environments
	2	Advanced Dave Baiting
(was originally 2)	3	Neutron Bomb Juggling

Iterating through an ArrayList

- You can use a for loop to go through an ArrayList

```
for (int i = 0; i < modules.size(); i++)
{
   String singleModule = modules.get(i);
   System.out.println(singleModule);
}
```

- However, the for/each loop construct can also be used:

```
for (String singleModule : modules)
{
   System.out.println(singleModule);
}
```

(The for/each works on standard arrays, too!)

How to break things…!

- What's wrong with this picture?

```
ArrayList<String> modules = new ArrayList();
modules.add("Programming 1");
modules.add("System Environments");
modules.add("Neutron Bomb Juggling");
modules.add("Advanced Dave Baiting");

for (String currentModule : modules)
{
   if (currentModule.equals("System Environments"))
   {
      modules.remove(currentModule);
   }
   System.out.println(currentModule);
}
```

How to break things…!

- What's wrong with this picture?

```
ArrayList<String> modules = new ArrayList();
modules.add("Programming 1");
modules.add("System Environments");
modules.add("Neutron Bomb Juggling");
modules.add("Advanced Dave Baiting");

for (String currentModule : modules)
{
   if (currentModule.equals("System Environments"))
   {
      modules.remove(currentModule);
   }
   System.out.println(currentModule);
}
```

```
Exception in thread "main"
java.util.CurrentModificationException
```
– WTF?!

How to break things…!

- If you are iterating through a collection, you cannot modify the collection
- So in the previous example, when we tried to modify the ArrayList half way through the for/each, we got a ConcurrentModificationException
- In English: we were modifying something within a loop upon which the loop was dependent

Using an `Iterator` on a collection

- An **`Iterator`** lets us iterate through a collection and make changes to it as we go
- On any collection there will be a method called **`iterator`** (surprisingly enough) that will give us an iterator object
- The iterator has a variety of methods that let traverse through and (among other things) remove items from the collection as we go

Using the Iterator

- *(assuming the ArrayList called **modules** from previous slides)*
- First get an iterator from the collection object

```
Iterator<String> myIt = modules.iterator();
```

- The method hasNext() gives true or false if there's another element in the collection
- We can use this as the condition for a while loop

```
while (myIt.hasNext())
```

- The method next() gives us the next element in the collection
- The method remove() will remove the next element in the collection without making the iteration fall over

```
String currentModule = myIt.next();
if (currentModule.equals("System Environments"))
{
   myIt.remove();
}
```

Using the Iterator

```
Iterator myIt = modules.iterator();
while (myIt.hasNext())
{
  String currentModule = myIt.next();
  if (currentModule.equals("System Environments"))
  {
     myIt.remove();
  }
}
```

Collections of objects

- Collections store *objects*
- Up until now we've stored Strings
- But we could store ints, or doubles – or even instances of your own defined classes
- For example…

assuming you have a House class with appropriate attributes, getters and setters

```
House pauls = new House();
pauls.setAddress("49 Flibble Street");
House jills = new House();
jills.setAddress("78 Flibble Street");

ArrayList<House> flibbleStreet = new ArrayList();
flibbleStreet.add(pauls);
flibbleStreet.add(jills);
```

IMPORTANT: Generics and primitive data types

- We've seen how we can use Strings or our own classes as a data type for a collection
  - (or more accurately, we can use Strings or our own classes as a generic, e.g. the bit between the < > when we declare an array list, e.g. <String> or <Student>)
- However, you cannot use primitive data types!
- So **`ArrayList<int> broken = new ArrayList();`**

would not work! ☹

- Instead, there are "wrapper" classes for the primitive data types you must use, e.g.
  - **`ArrayList<Integer> works = new ArrayList();`**
  - **`ArrayList<Double> alsoWorks = new ArrayList();`**

- When it comes to the practical, anyone who gets this wrong, I shall point, and laugh at you!
  - (because it will demonstrate you either weren't paying attention in the lecture, or because you weren't AT the lecture! ☺)

Maps

- Java offers other Collections apart from the array-like ArrayList
- One of the more useful ones are the Map collections
- We will take a look at the HashMap

Map

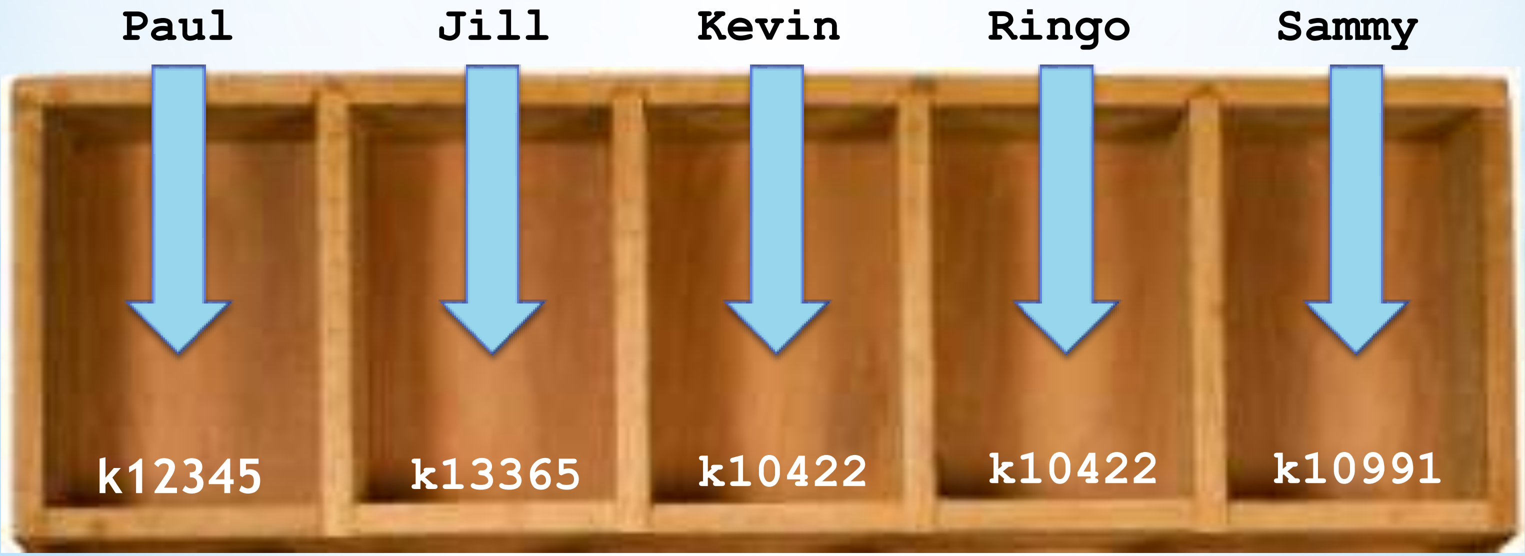
- If an array (and an ArrayList) is like a box with numbered compartments, then the Map is like a box with *named* compartments:

```
HashMap<String,String> box = new HashMap();
box.put("k12345","Paul");      box.put("k10422","Ringo");
box.put("k13365","Jill");      box.put("k10991","Sammy");
box.put("k10422","Kevin");
```

Map

- If an array (and an ArrayList) is like a box with numbered compartments, then the Map is like a box with *named* compartments:

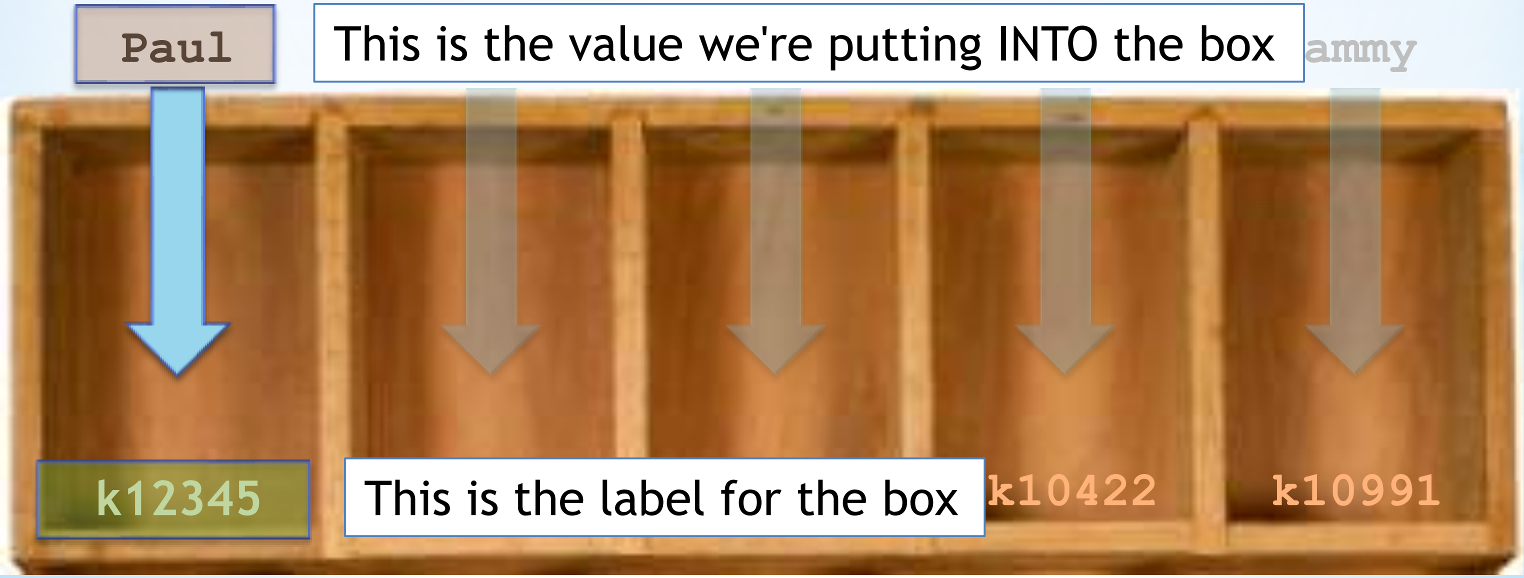
```
HashMap<String,String> box = new HashMap();
box.put("k12345","Paul");       box.put("k10422","Ringo");
box.put("k13365","Jill");       box.put("k10991","Sammy");
box.put("k10422","Kevin");
```



Map

- If an array (and an ArrayList) is like a box with numbered compartments, then the Map is like a box with *named* compartments:

```
HashMap<String,String> box = new HashMap();
box.put( "k12345" , "Paul" );   box.put("k10422","Ringo");
box.put("k13365","Jill");       box.put("k10991","Sammy");
box.put("k10422","Kevin");
```



This is the value we're putting INTO the box

This is the label for the box

Getting stuff back from the map

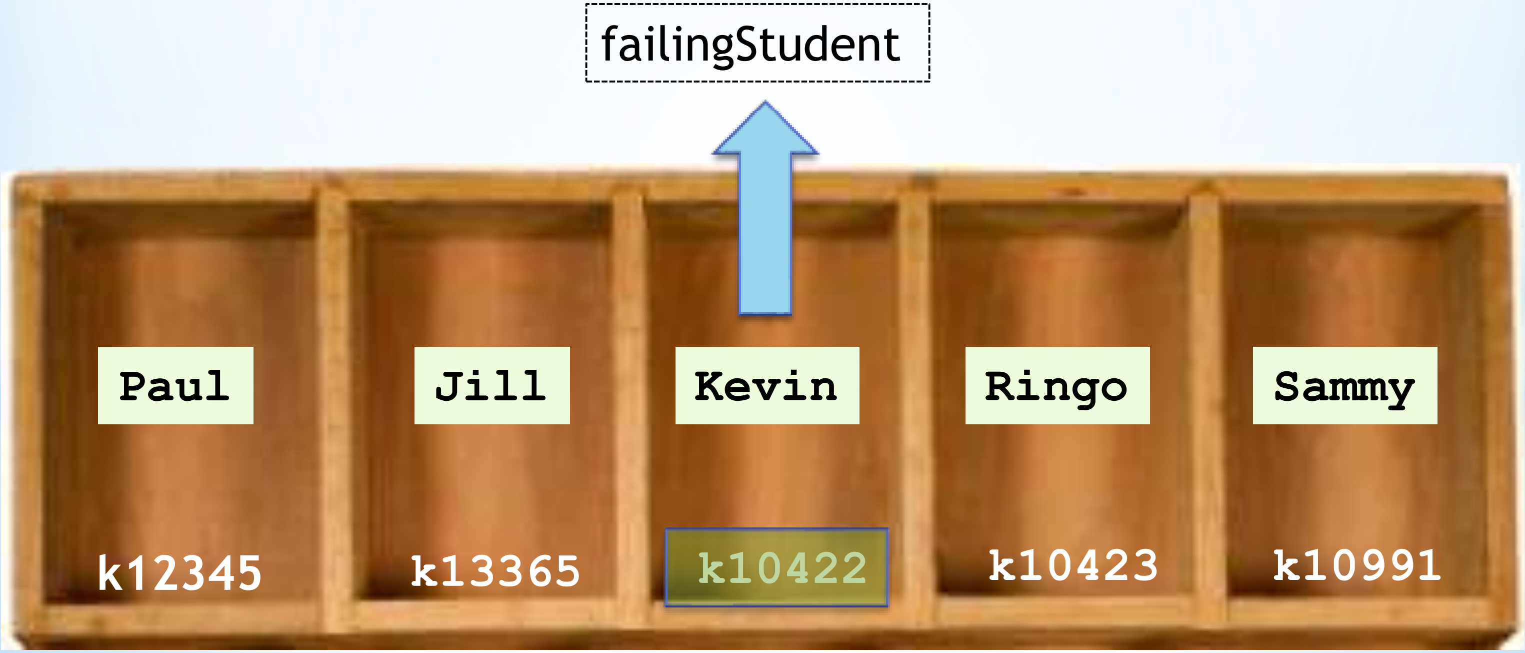
- Use **`get`** to retrieve stuff from the map
- Supply a KEY to get the value back:

```
String failingStudent = box.get("K104222");
```

Getting stuff back from the map

- Use `get` to retrieve stuff from the map
- Supply a KEY to get the VALUE back:

```
String failingStudent = box.get( "k104222" );
```



failingStudent

Paul

k12345

Jill

k13365

Kevin

k10422

Ringo

k10423

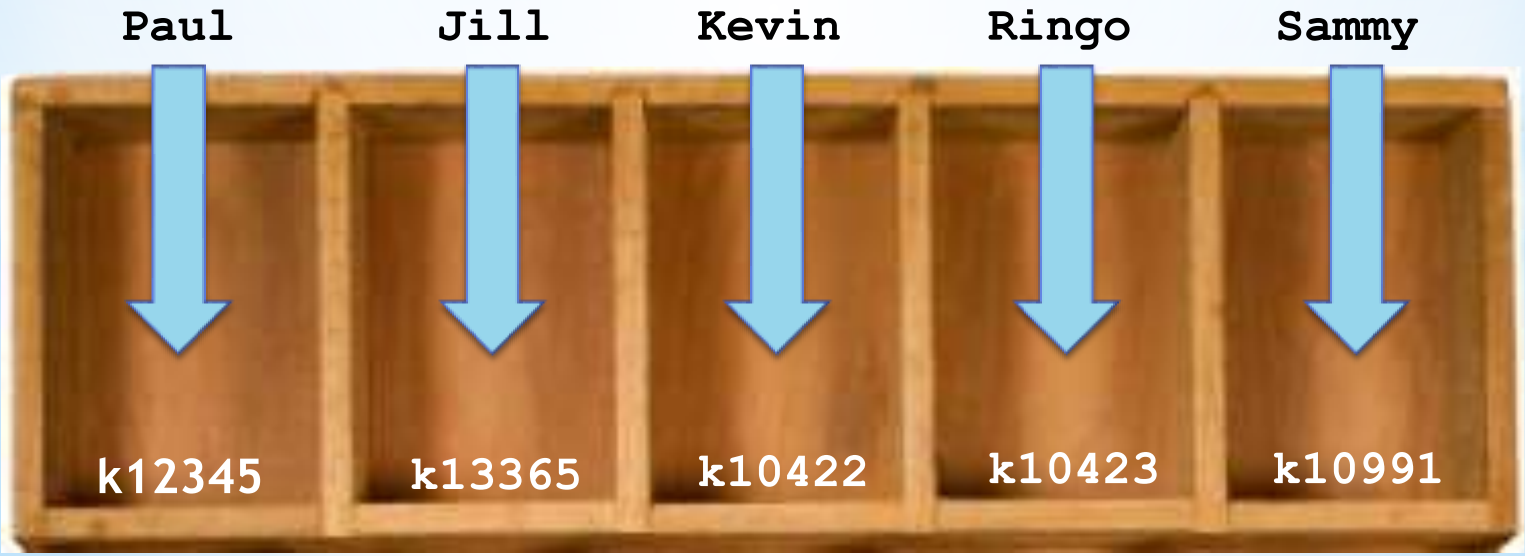
Sammy

k10991

Map

- Note that the KEY of a map entry is unique… so if you put something using a key that's already been used, you are REPLACING the original value:

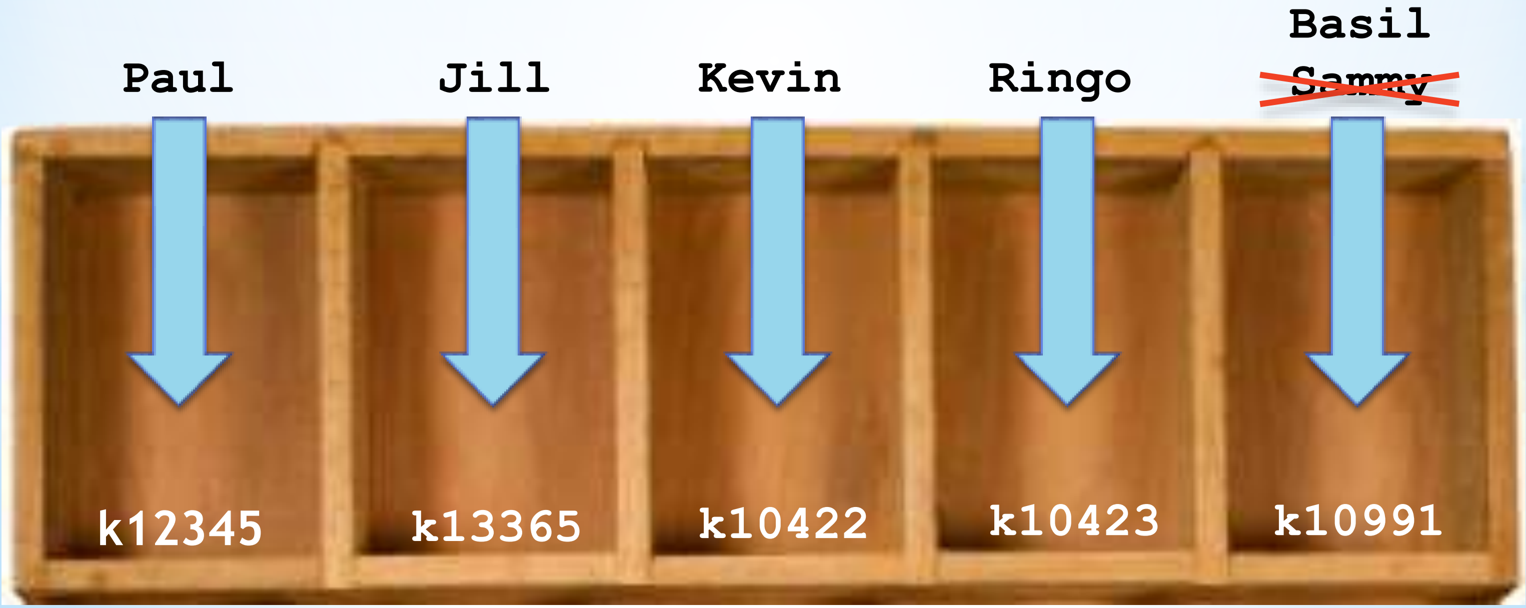
```
HashMap<String,String> box = new HashMap();
box.put("k12345","Paul");       box.put("k10423","Ringo");
box.put("k13365","Jill");       box.put("k10991","Sammy");
box.put("k10422","Kevin");      box.put("k10991","Basil");
```



Map

- Note that the KEY of a map entry is unique… so if you put something using a key that's already been used, you are REPLACING the original value:

```
HashMap<String,String> box = new HashMap();
box.put("k12345","Paul");        box.put("k10423","Ringo");
box.put("k13365","Jill");        box.put("k10991","Sammy");
box.put("k10422","Kevin");       box.put("k10991","Basil");
```



HashMaps of objects

- The two data types that follow the HashMap keyword specify the data type for the key and the value of each HashMap entry:

```
HashMap<String,String> box = new HashMap();
```

- So this meant, *give me a HashMap that labels its compartments with Strings, and that stores Strings in each compartment*
- Nothing says that you have to store strings – or even use strings as labels

HashMaps of objects

assuming you have a Student class with appropriate attributes, getters and setters…

```
Student paul = new Student();
paul.setName("Paul Neve");
paul.setCourse("Flower Arranging");
HashMap<String,Student> students = new HashMap();
students.put("k14242",paul);
```

HashMaps of objects

assuming you have a Student class with appropriate attributes, getters and setters…

```
Student paul = new Student();
paul.setName("Paul Neve");
paul.setCourse("Flower Arranging");
HashMap< String , Student > students = new HashMap();
students.put( "k14242" , paul );
```

Our KEY is a String, our VALUE is an instance of our Student class

HashMaps of objects

*(brainf**k time ☺)*

```
Student paul = new Student();
paul.setName("Paul Neve");
paul.setCourse("Flower Arranging");
ArrayList<Integer> marks = new ArrayList();
marks.add(75);
marks.add(81);
HashMap<Student,ArrayList<Integer>> students = new HashMap();
students.put(paul,marks);
```

HashMaps of objects

*(brainf**k time ☺)*

```
Student paul = new Student();
paul.setName("Paul Neve");
paul.setCourse("Flower Arranging");
ArrayList<Integer> marks = new ArrayList();
marks.add(75);
marks.add(81);
HashMap<Student,ArrayList<Integer>> students = new HashMap();
students.put(paul,marks);
```

Don't forget: if you are using integers or other primitive data types in your collections, you need to use the wrapper classes rather than int or double. Look for the capital letter!

HashMaps of objects

*(brainf**k time ☺)*

```
Student paul = new Student();
paul.setName("Paul Neve");
paul.setCourse("Flower Arranging");
ArrayList<Integer> marks = new ArrayList();
marks.add(75);
marks.add(81);
HashMap<Student, ArrayList<Integer>> students = new HashMap();
students.put(paul,marks);
```

Nothing says the KEY has to be a simple data type or class. You can use complex objects as a key.

HashMaps of objects

(brainf**k time ☺)

```
Student paul = new Student();
paul.setName("Paul Neve");
paul.setCourse("Flower Arranging");
ArrayList<Integer> marks = new ArrayList();
marks.add(75);
marks.add(81);
HashMap<Student, ArrayList<Integer> > students = new HashMap();
students.put(paul,marks);
```

Here's the cool bit. Nothing stops us specifying an ArrayList as a data type – so in this case, we're saying the values in our HashMap will each be an ArrayList! So, for each student (the KEY) we can store an ArrayList of their marks (the VALUE)

HashMaps of objects

*(continuing the brainf**k example)*

- Say we had several students in our HashMap (so not just Paul)
  - So say we had instances of Student named **`Jill`**, **`Fred`**, **`Harry`** as well as **`Paul`**
  - Say all of these instances of Student had been added to the HashMap with corresponding ArrayLists of their marks
- Then given a specific instance of a Student, we could do

```
ArrayList<Integer> janesMarks = students.get(jane);
```

- (Don't panic if this has blown your mind! The exercises won't go *quite* this far…)

Other methods on HashMap

```
HashMap<String,String> box = new HashMap();
box.put("k12345","Paul");      box.put("k10422","Ringo");
box.put("k13365","Jill");      box.put("k10991","Sammy");
box.put("k10422","Kevin");
```

containsKey	Gives true if the HashMap contains a value for a given key, false if it doesn't
	e.g.
	`if (box.containsKey("k12345"))`
	`{`
	`   System.out.println("We have that K number!");`
	`}`

containsValue	Gives true is the HashMap contains the given value (under ANY key)
	e.g.
	`if (box.containsValue("Paul"))`
	`{`
	`   System.out.println("We have that student!");`
	`}`

remove	Removes any value for the given key, e.g. `box.remove("k10422"); // poor Ringo`

Sets (quickly ☺)

- Collections based on sets store unique values (as in a mathematical set)
- There are no duplicate values, and many implementations don't have any specific ordering or way of navigation
- The key thing about sets is the whole "is some value in the set"?
  - think of them as sort of being like HashMaps without the key

Sets (quickly ☺)

```
HashSet<String> myset = new HashSet();
myset.add("Banana");
myset.add("Orange");
myset.add("Apple");

if (myset.contains("Potato"))
{
   System.out.println("Time to make some chips");
}
```

What would be the difference?

Given the first line

```
HashSet<String> fruits = new HashSet();
```

or

```
ArrayList<String> fruits = new ArrayList();
```

then the following code in both cases

```
fruits.add("Banana");
fruits.add("Orange");
fruits.add("Apple");
fruits.add("Banana");

for (String fruit : fruits)
{
   System.out.println(fruit);
}
```

One last thing

- Remember that collections are just data types like any other
- So, nothing stops you having a collection as an attribute on one of your classes
- For example, a Student has many Modules
  - Maybe **`Student`** has an attribute called **`modules`** which is an **`ArrayList`** of a **`Module`** class…?

Summary

- The Java collections API gives you a variety of ways in which you can store collections of objects that improves upon standard arrays
- Use an ArrayList as an almost straight replacement for an array – but it can grow and shrink through add, inserting or removing items
- Use a HashMap to store things as key/value pairs
  - So you can index data based on things other than a numeric index
  - You can index by your objects, which lets you store data associated with that object
- Use a set to keep a track of a unique list of items