Collections

Let's Take a Quick Break

Organizing Data

- We have many ways of storing and organizing data in our programs:
  - `String`s for holding sequences of characters.
  - `ArrayList`s for holding sequences of general objects.
  - Arrays for holding fixed-sized sequences.
  - `HashMap`s for associating data with one another.
- Are there other ways of organizing data?
- What do they look like?

The Collections Framework

- Java has a variety of **collections classes** for holding groups of data.

The Collections Framework

- Java has a variety of **collections classes** for holding groups of data.

- The three major ways of organizing data are

    - `Set`s, which store unordered data

The Collections Framework

- Java has a variety of **collections classes** for holding groups of data.

- The three major ways of organizing data are

    - `Set`s, which store unordered data,

    - `List`s, which store sequences

The Collections Framework

- Java has a variety of **collections classes** for holding groups of data.

- The three major ways of organizing data are

  - `Set`s, which store unordered data,

  - `List`s, which store sequences, and

  - `Map`s, which store key/value pairs.

The Collections Framework

Java has a variety of **collections classes** for holding groups of data.

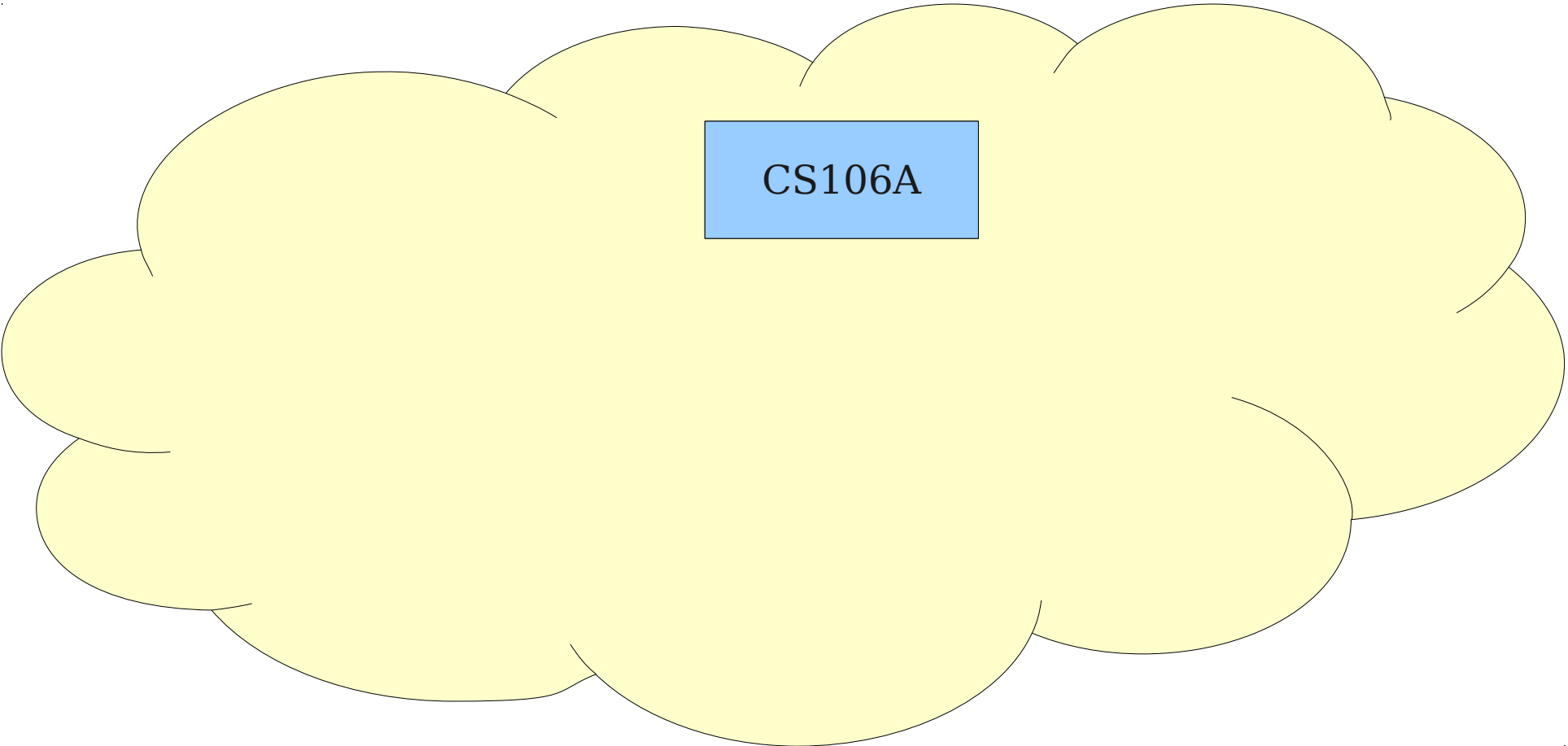The three major ways of organizing data are

- **Set**s, which store unordered data,

  Lists, which store sequences, and

  Maps, which store key/value pairs.

What is a Set?

- A **set** is a collection of distinct elements.

- Similar to an `ArrayList`, but elements are not stored in a sequence.

- Major operations are:

  - Adding an element.

  - Removing an element.

  - Checking whether an element exists.

- Useful for answering questions of the form "have I seen this before?"
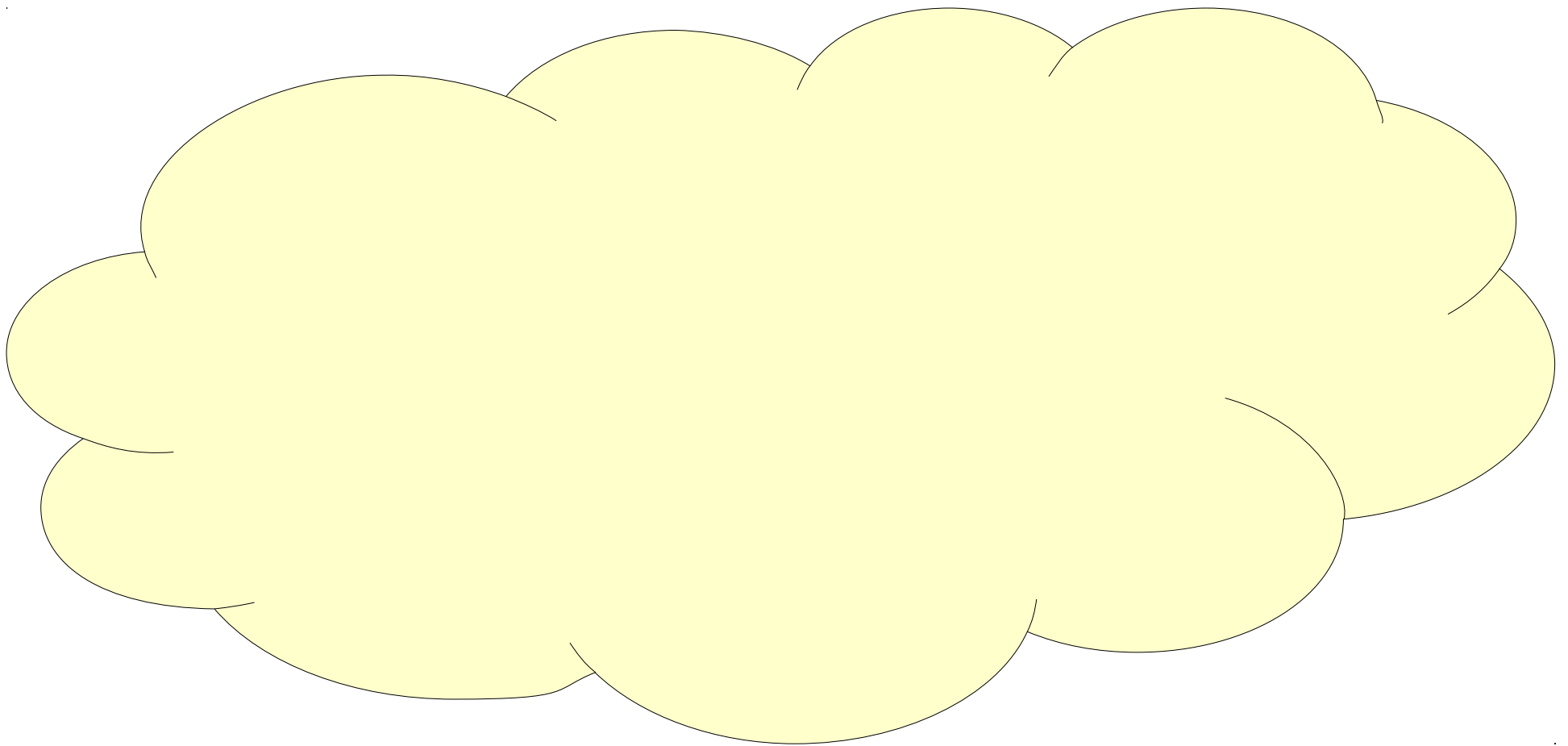
```java
HashSet<String> mySet = new HashSet<String>();
```
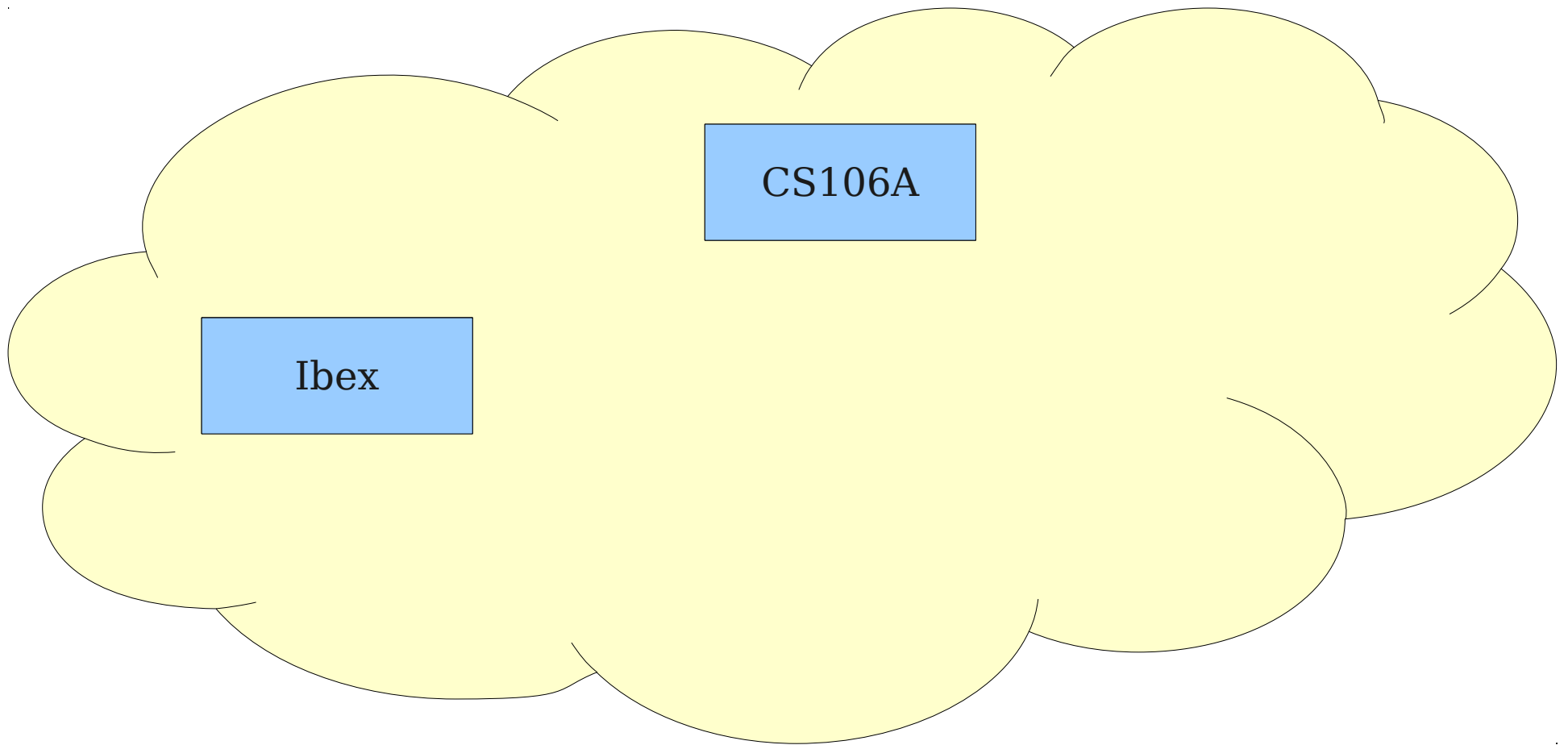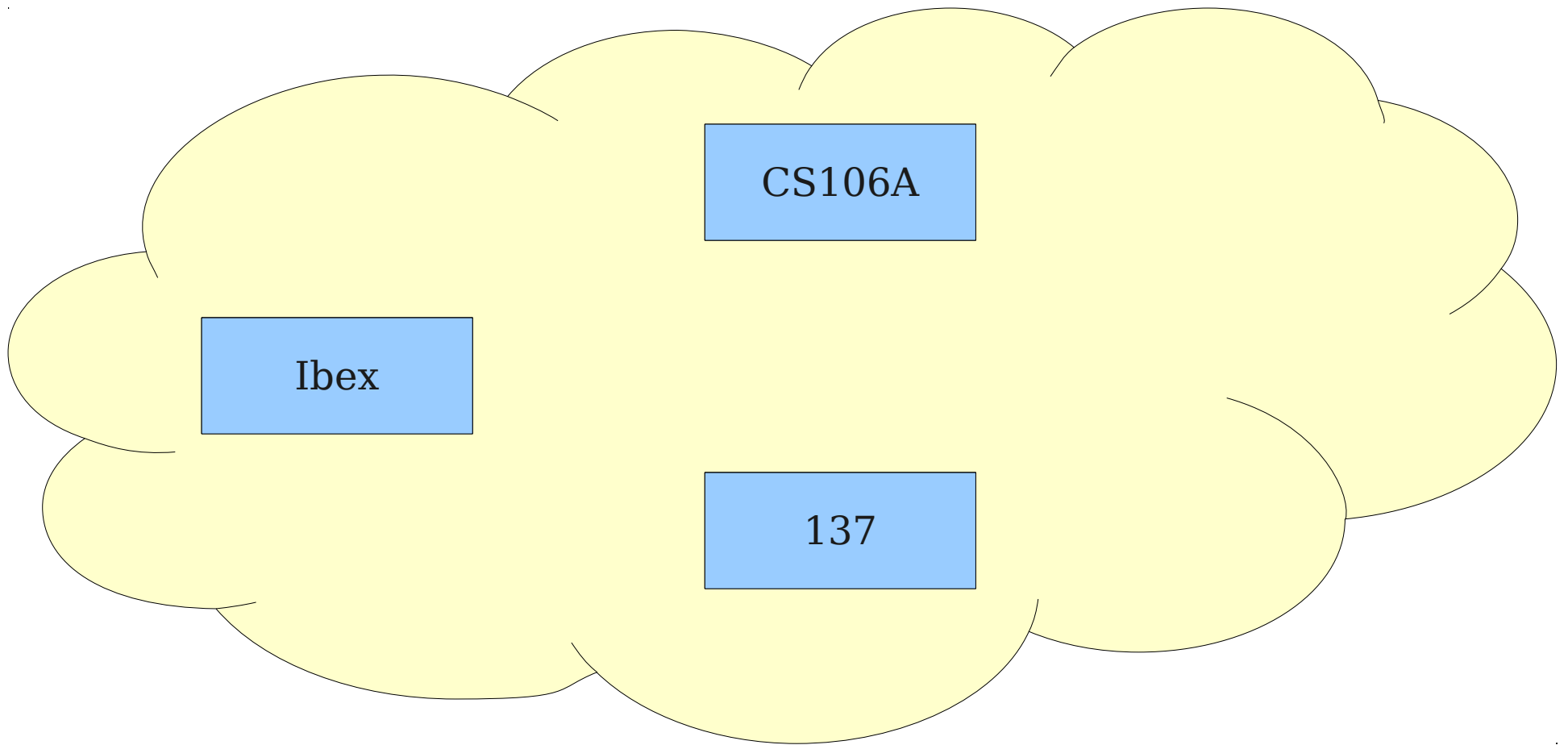
CS106A

```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
```

To add a value to a **HashSet**, use the syntax

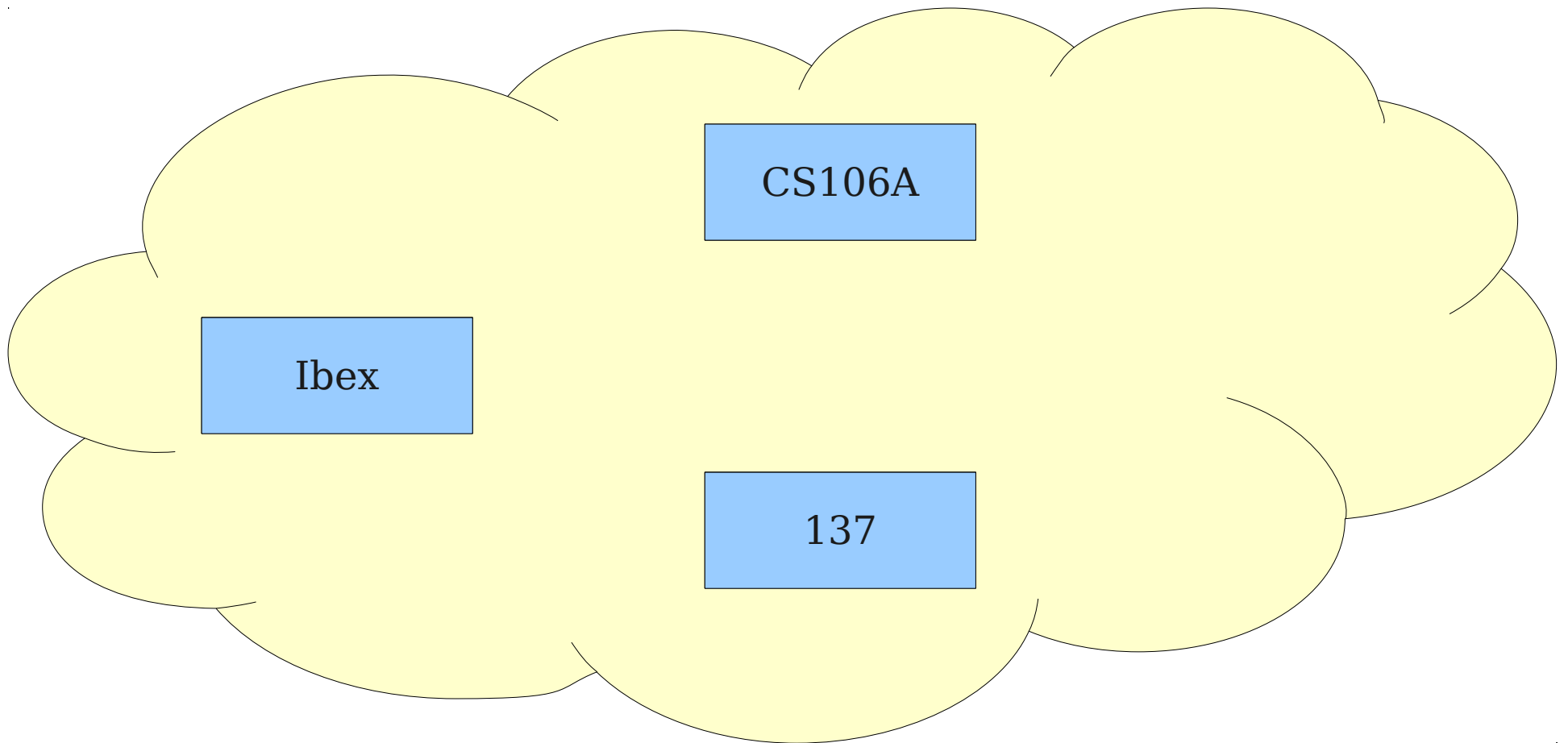*set*.add(*value*)

```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
```
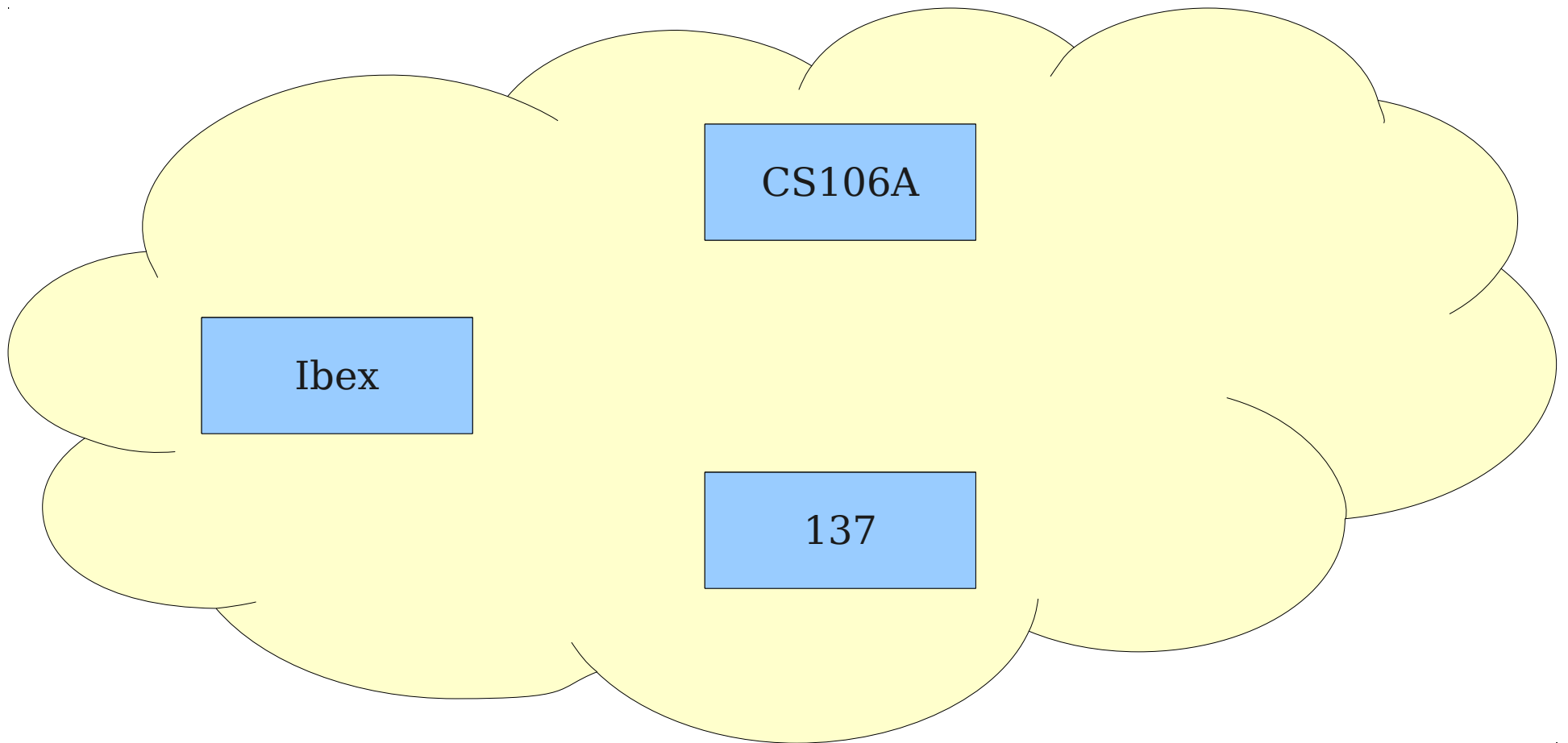
```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
```
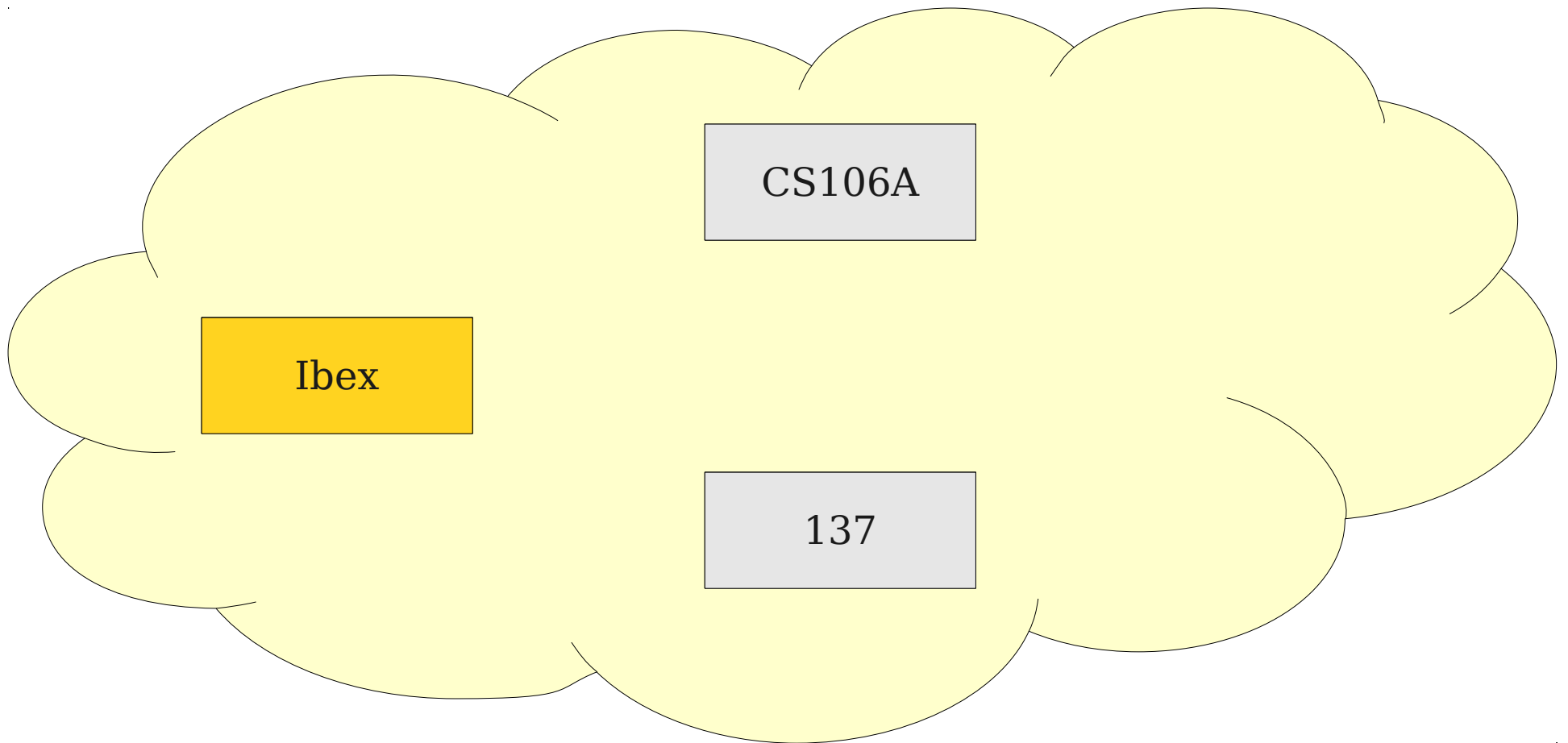
```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A");
```

If you **add** a value pair where the value exists, nothing happens.
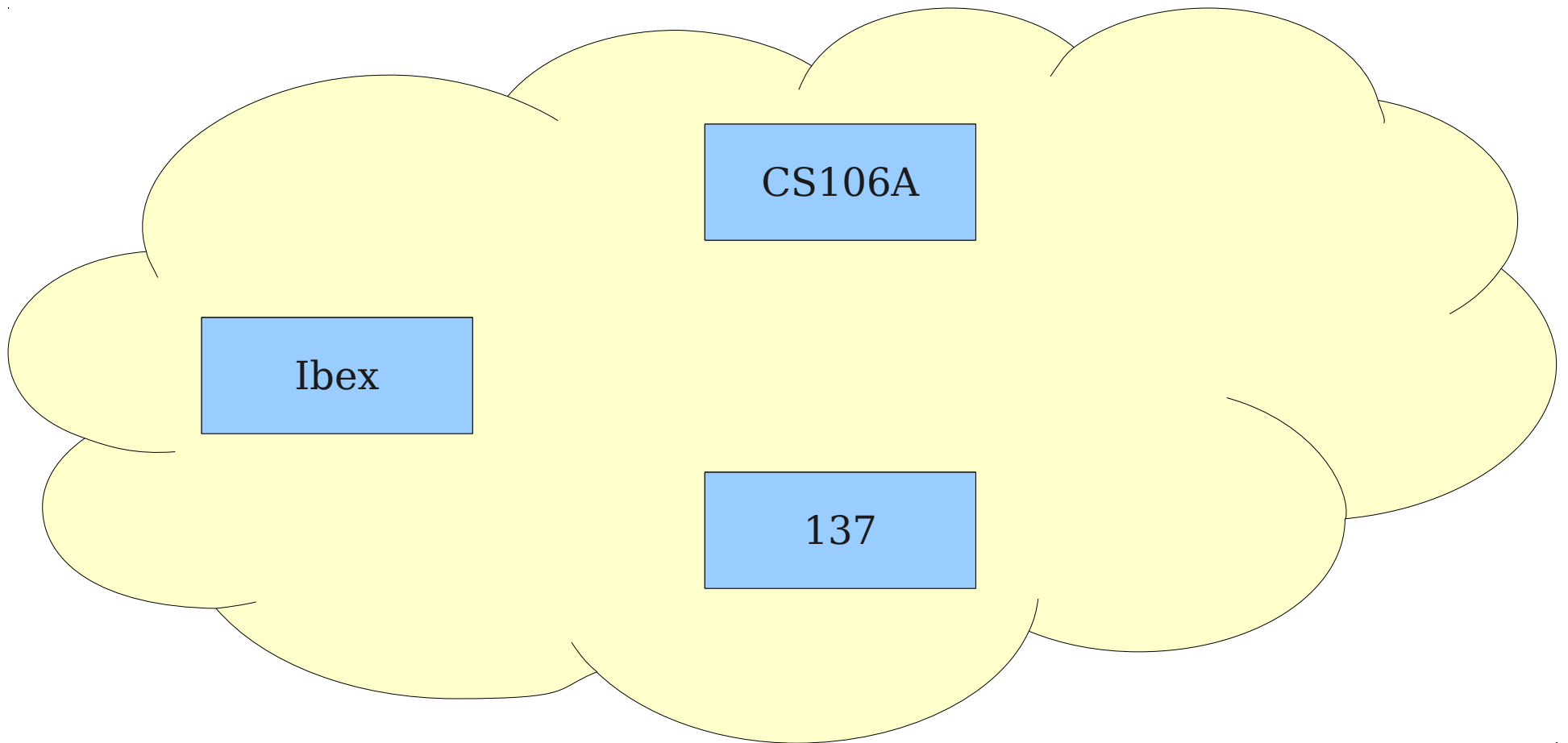
```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A");

mySet.contains("Ibex");
```
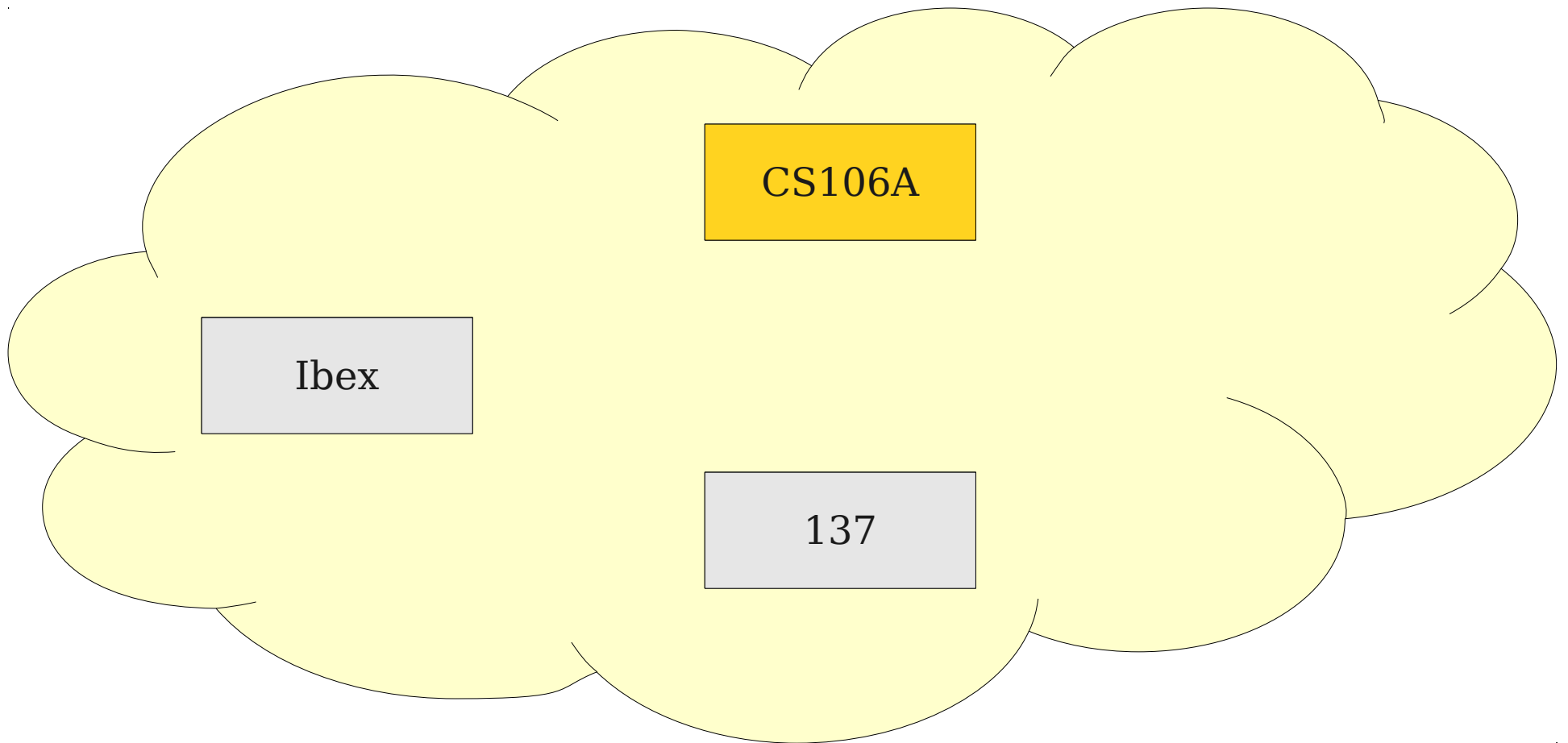
```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A");

mySet.contains("Ibex");
```
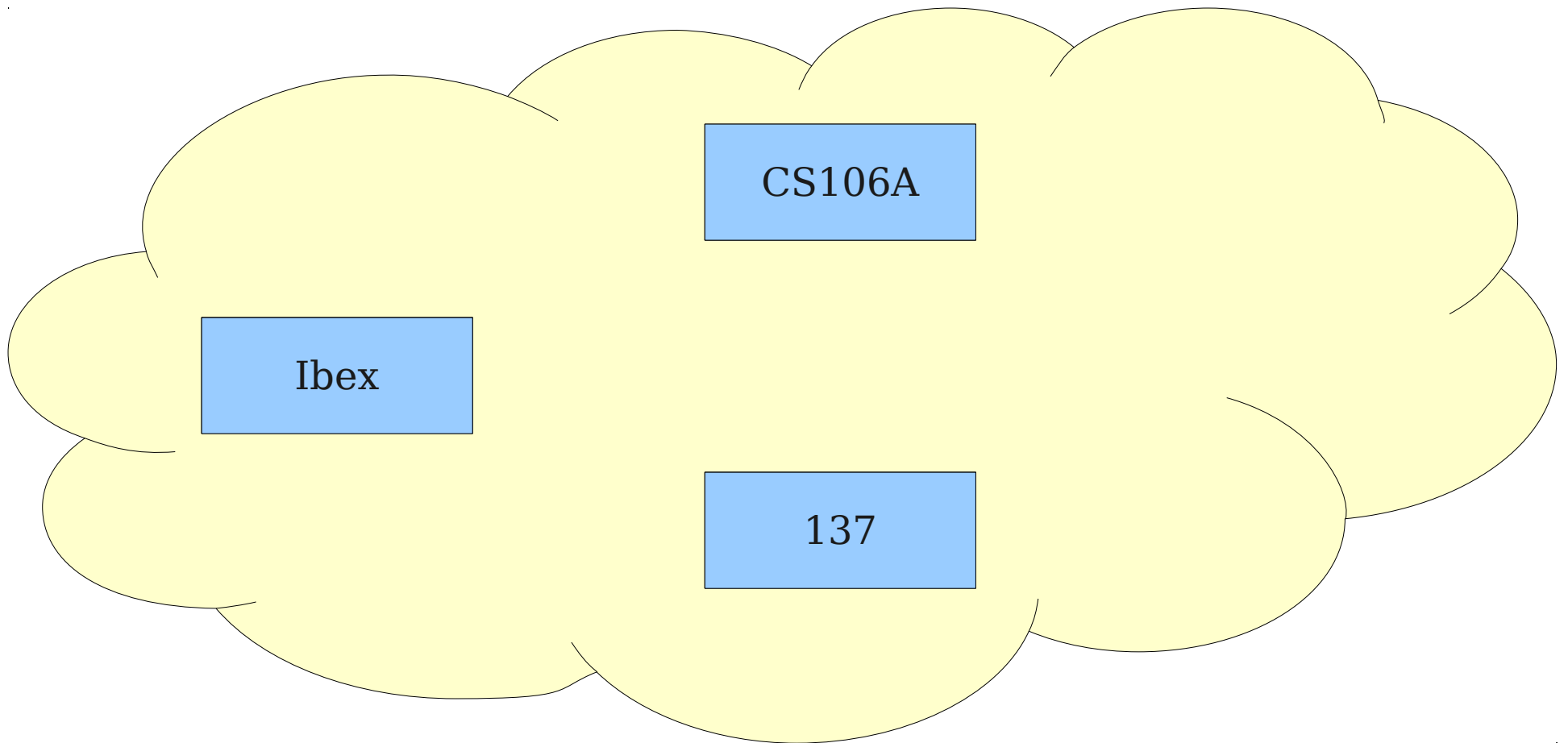
To see if a value exists:

`set.contains(value)`

```java
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A");

mySet.contains("Ibex");
mySet.contains("CS106A");
```
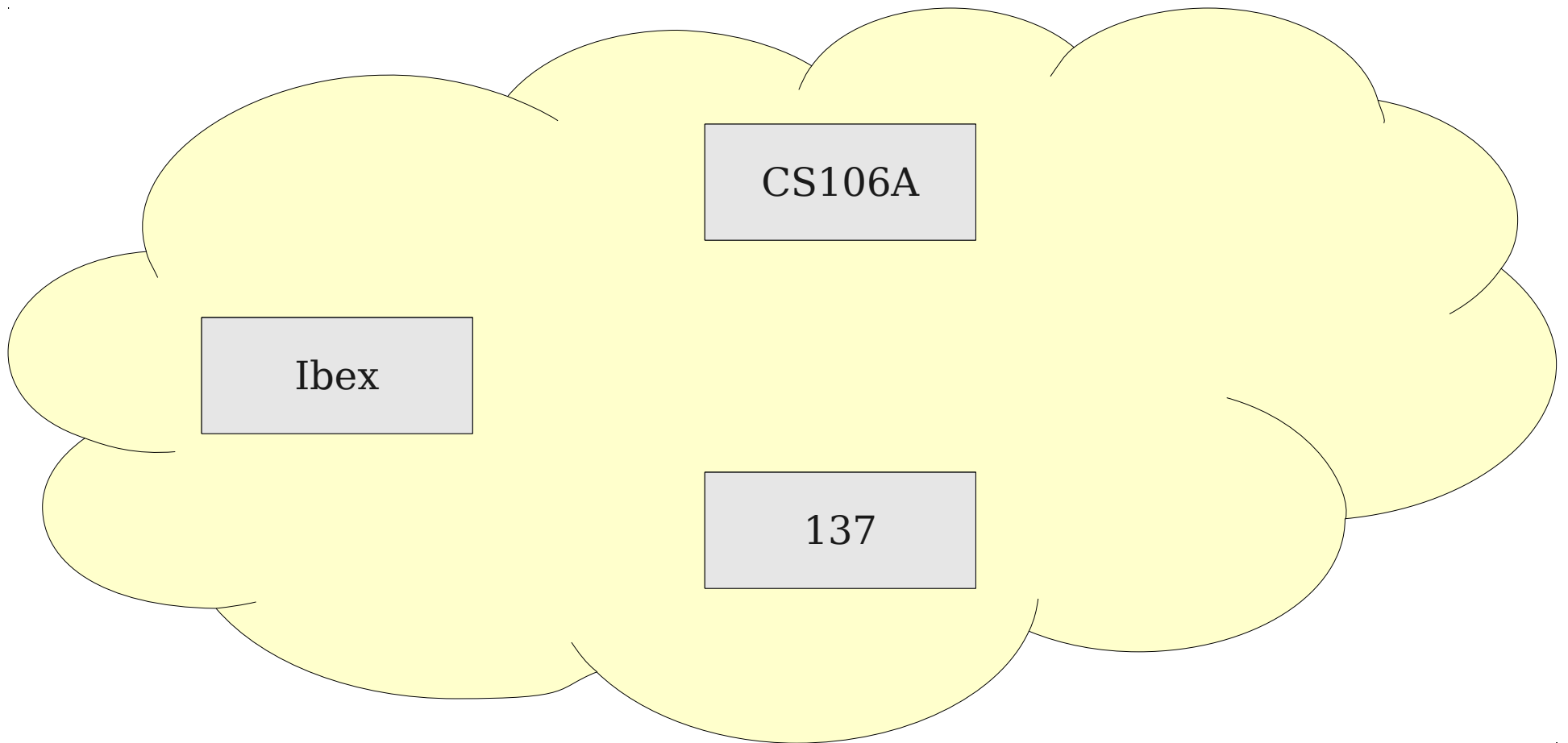
```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A")

mySet.contains("Ibex");
mySet.contains("CS106A");
```

```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A")

mySet.contains("Ibex");
mySet.contains("CS106A");
mySet.contains("<(^_^)>");
```

```java
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A")

mySet.contains("Ibex");
mySet.contains("CS106A");
mySet.contains("<(^_^)>");
```

Basic `Set` Operations

- To insert an element:

$$set.\texttt{add(}value\texttt{)}$$

- To check whether a value exists:

$$set.\texttt{contains(}value\texttt{)}$$

- To remove an element:

$$set.\texttt{remove(}value\texttt{)}$$

Word Walks

CO**DE**

DESI**RE**

REWRI**TE**

TEMPERA**TE**

TEATI**ME**

MEMEN**TO**

TORRENT

Word Skips

CARRO**T**

TOMAT**O**

OKR**A**

ASPARAGU**S**

SQUAS**H**

HORSERADISH

Word Skips

- Begin with any word you'd like.

- Choose a word whose first letter is the same as the last letter of your current word.

- Repeat until you get bored.

Iterators

- To visit every element of a collection, you can use the "for each" loop:

```
for (ElemType elem : collection) {

    …

}
```

- Alternatively, you can use an **iterator**, an object whose job is to walk over the elements of a collection.

- The iterator has two commands:

  - `hasNext()`, which returns whether there are any more elements to visit, and

  - `next()`, which returns the next element and moves the iterator to the next position.
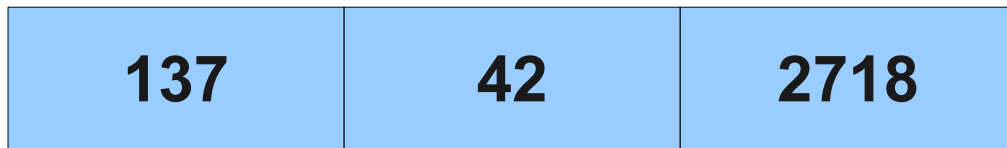
Java Iterators

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
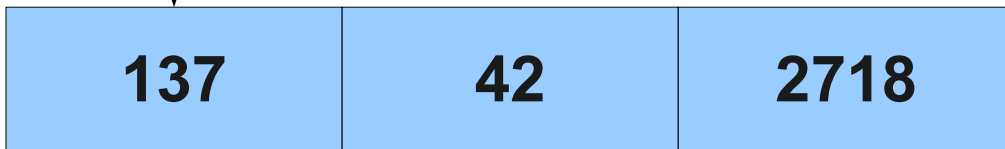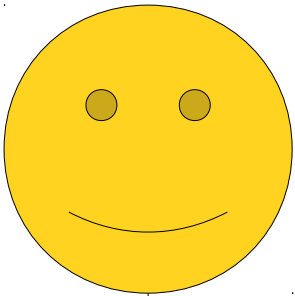
Java Iterators

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
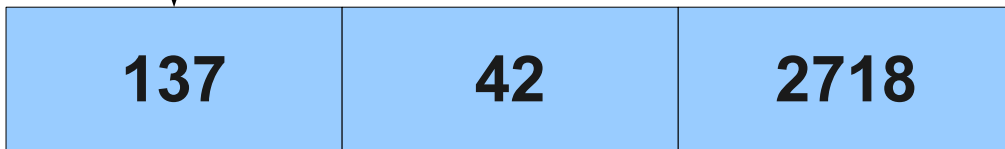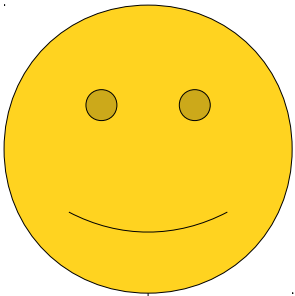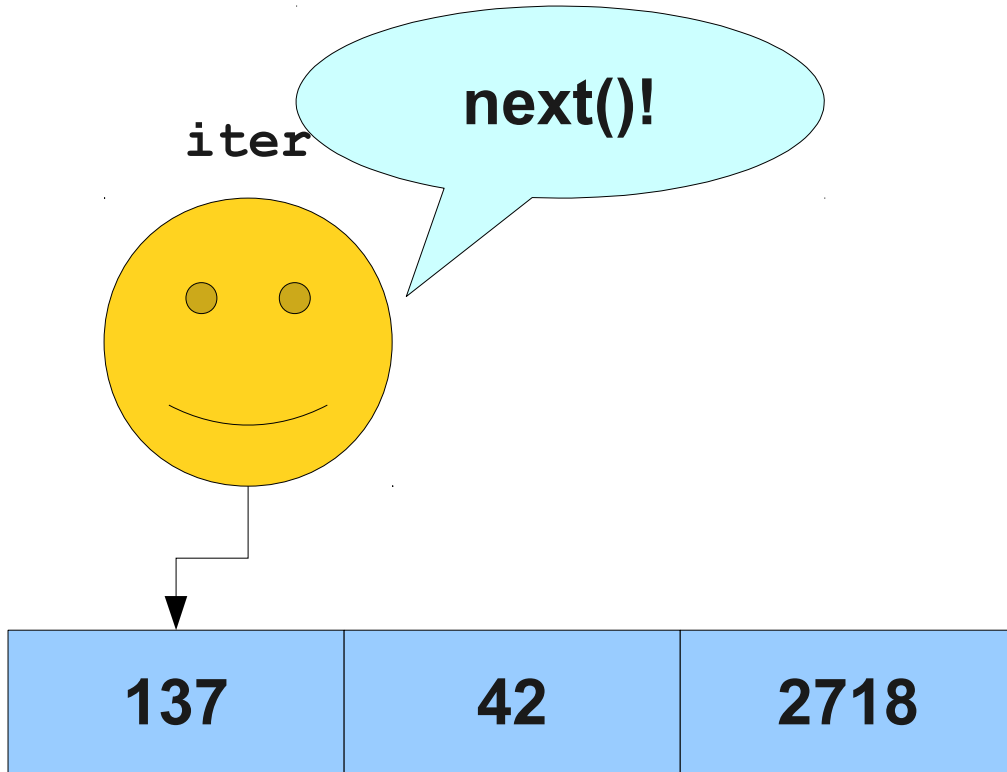
Java Iterators


```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

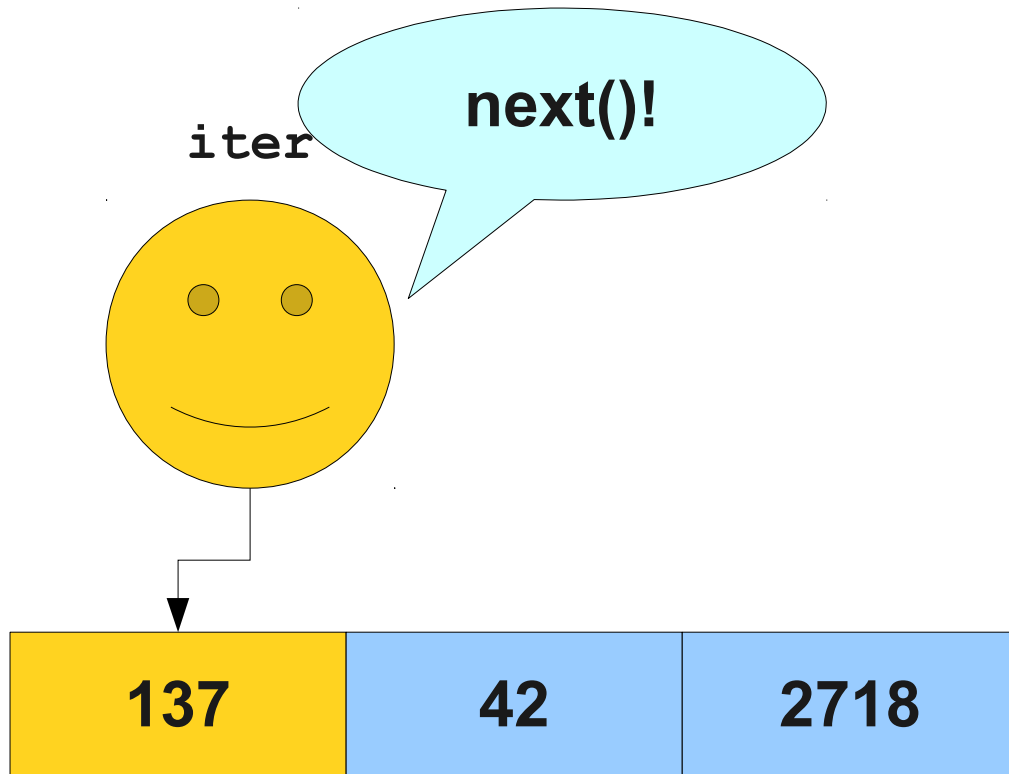| 137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

next()!

iter

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
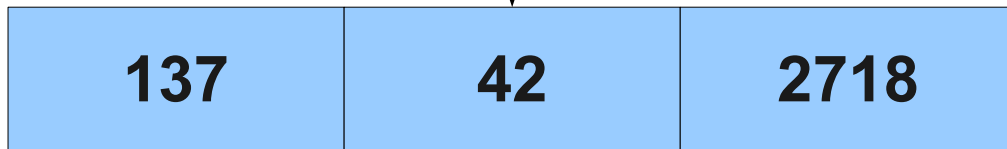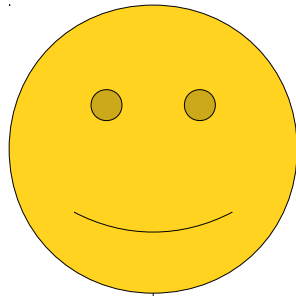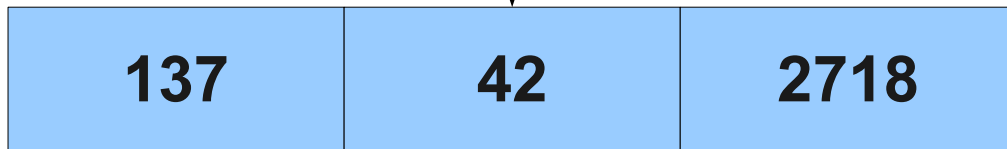
Java Iterators


```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

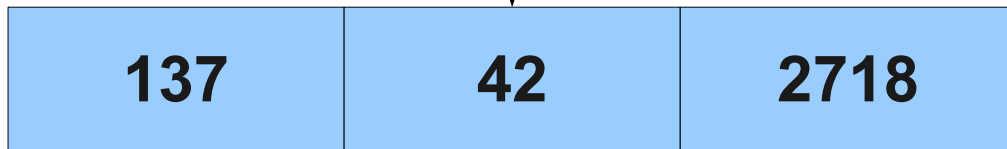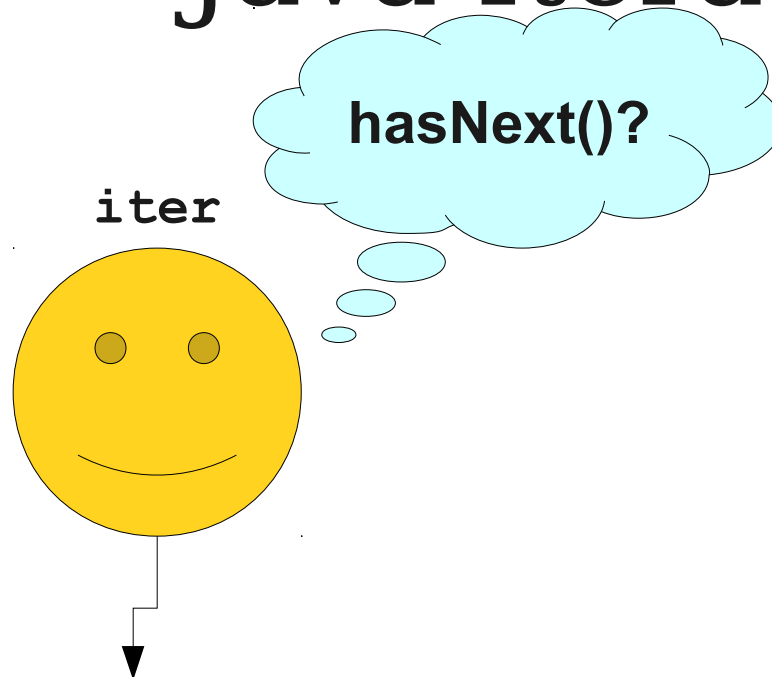| 137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
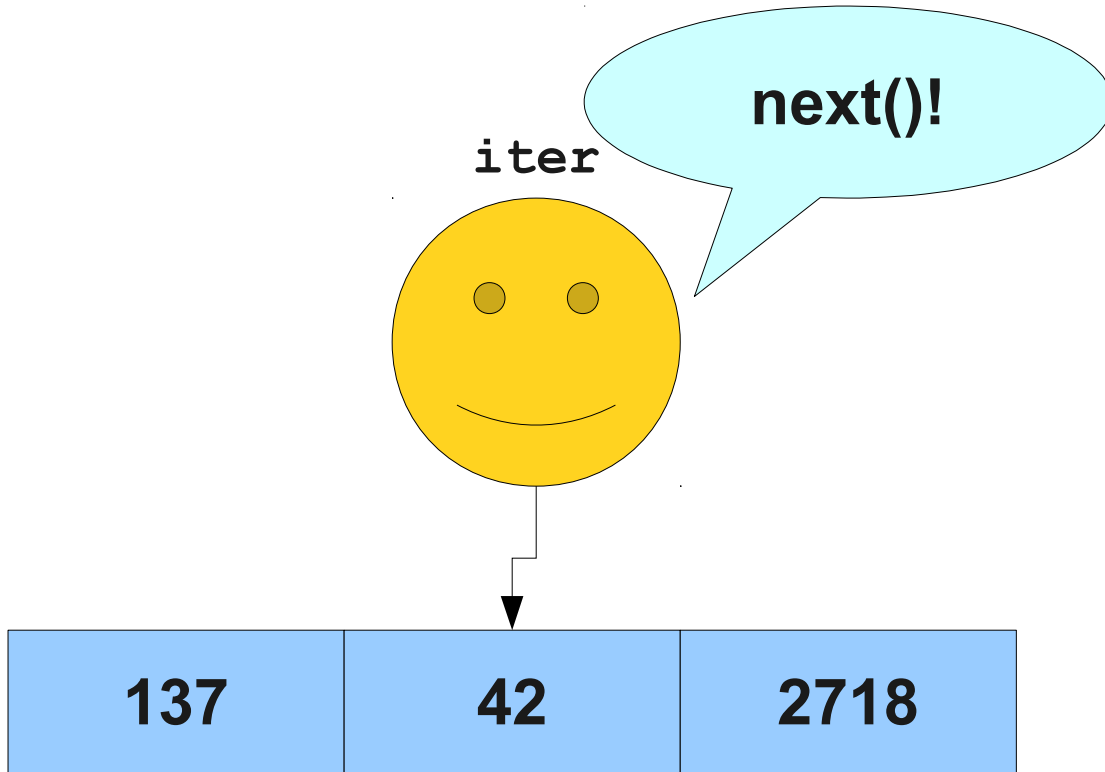
Java Iterators


```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

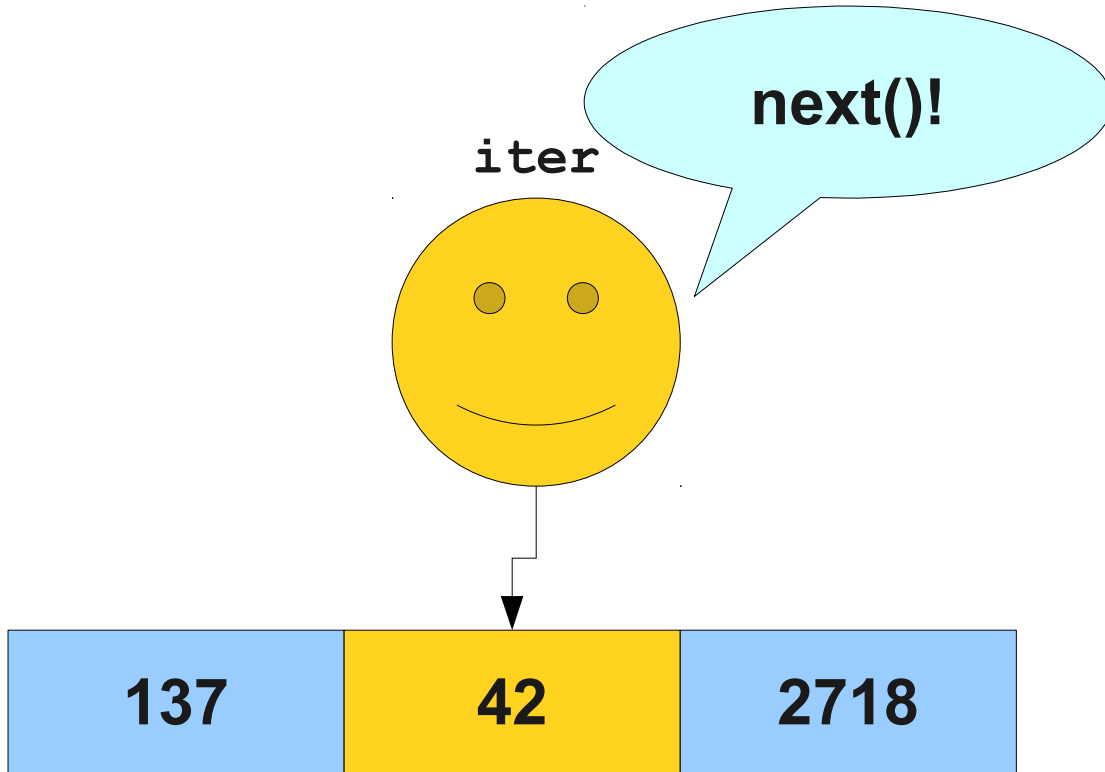# Java Iterators


```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators


```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
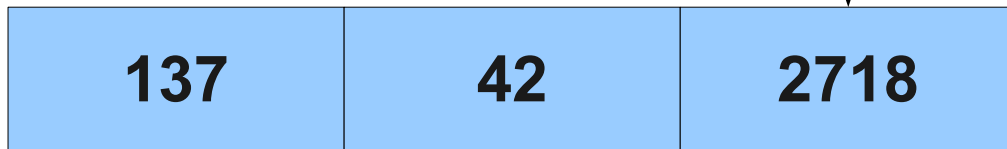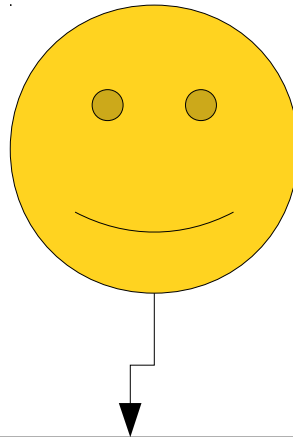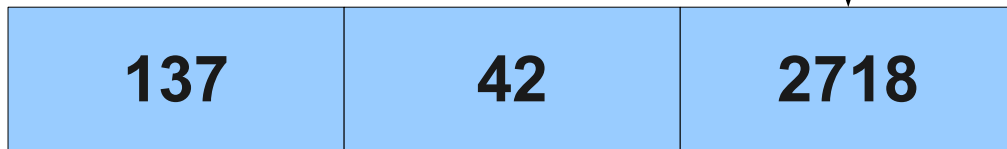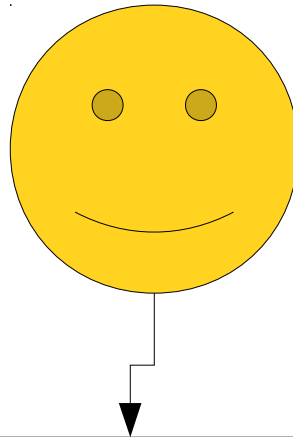
Java Iterators


```
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
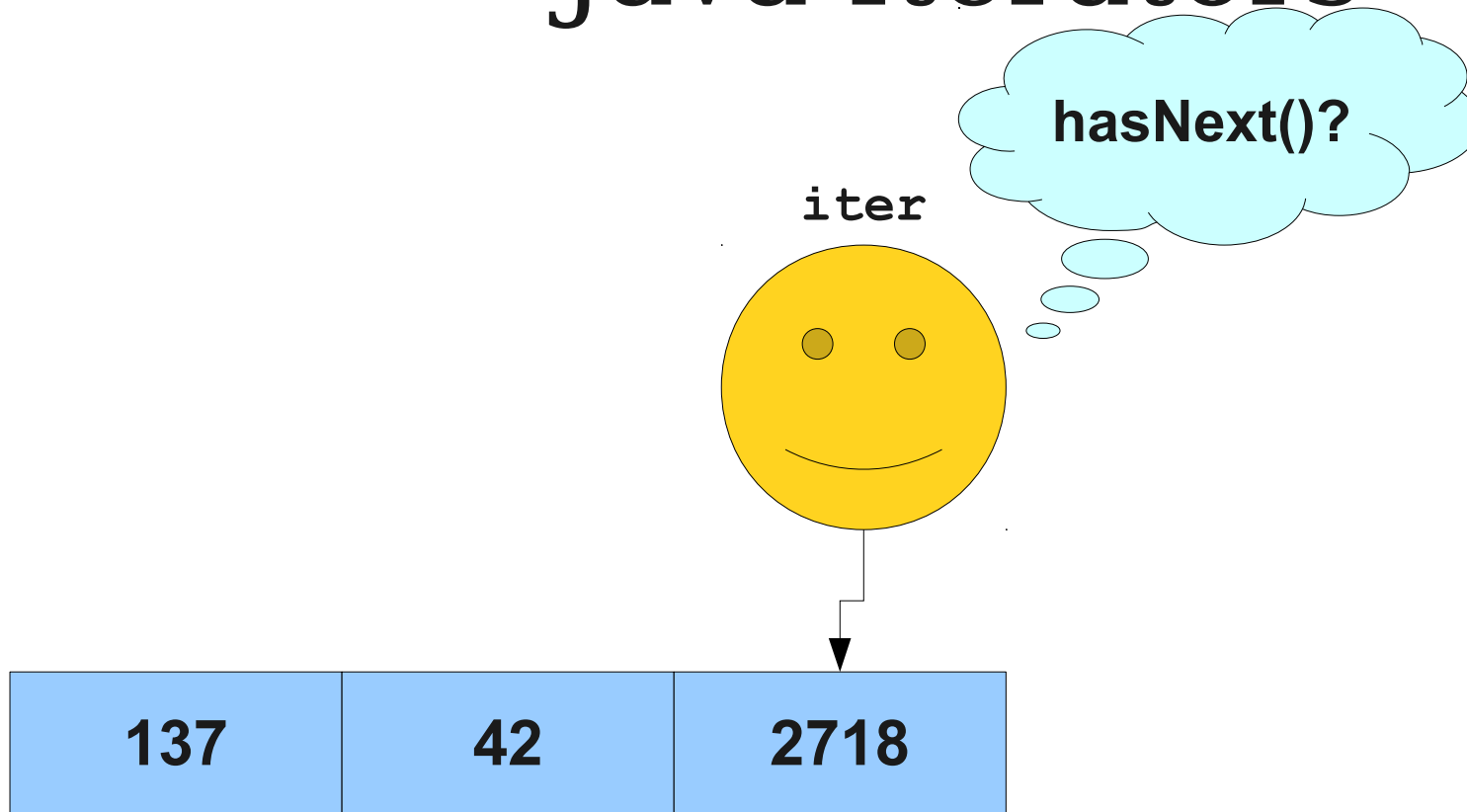
Java Iterators

iter

hasNext()?

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

137 | 42 | 2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators


```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators


```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
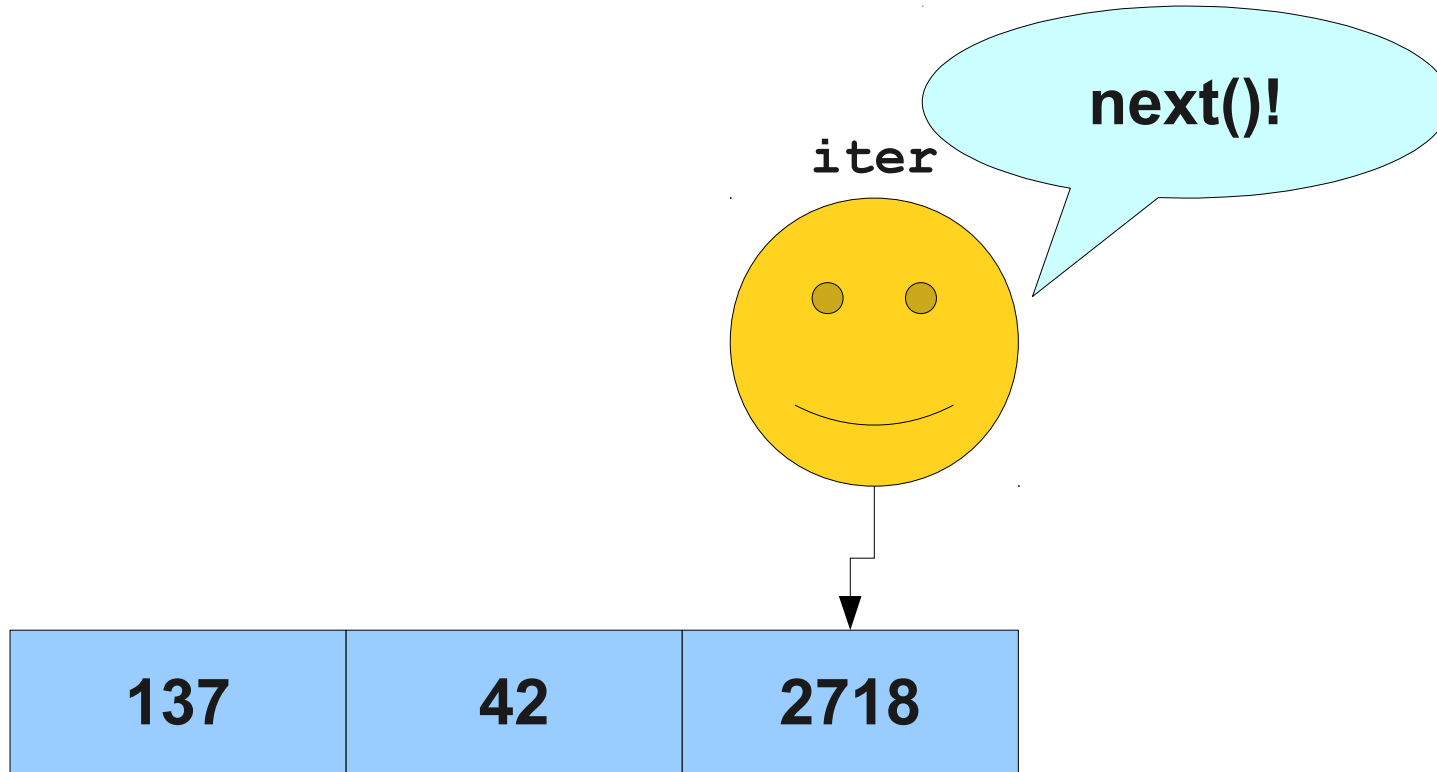
Java Iterators


```
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
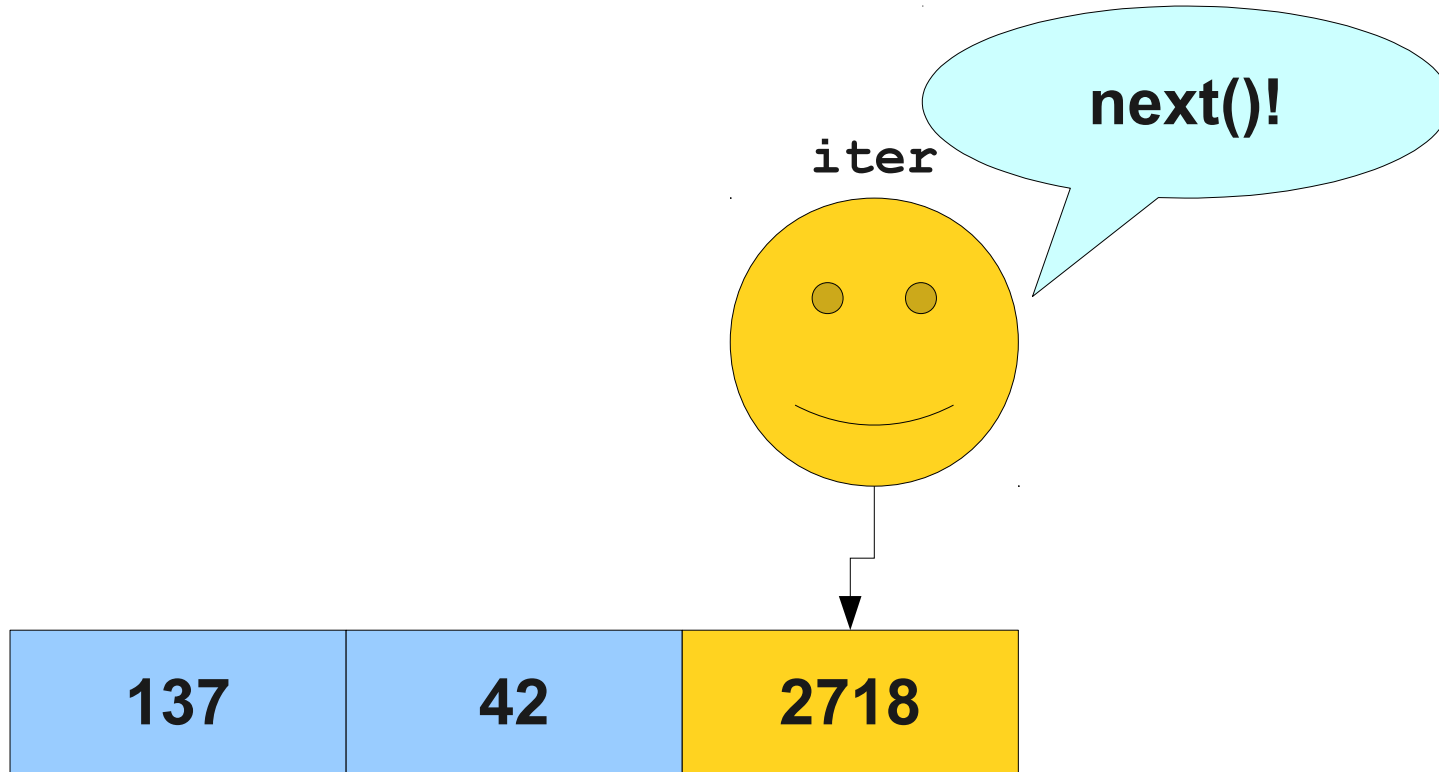
Java Iterators


```java
ArrayList<Integer> myList = /* … */;

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

hasNext()?

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
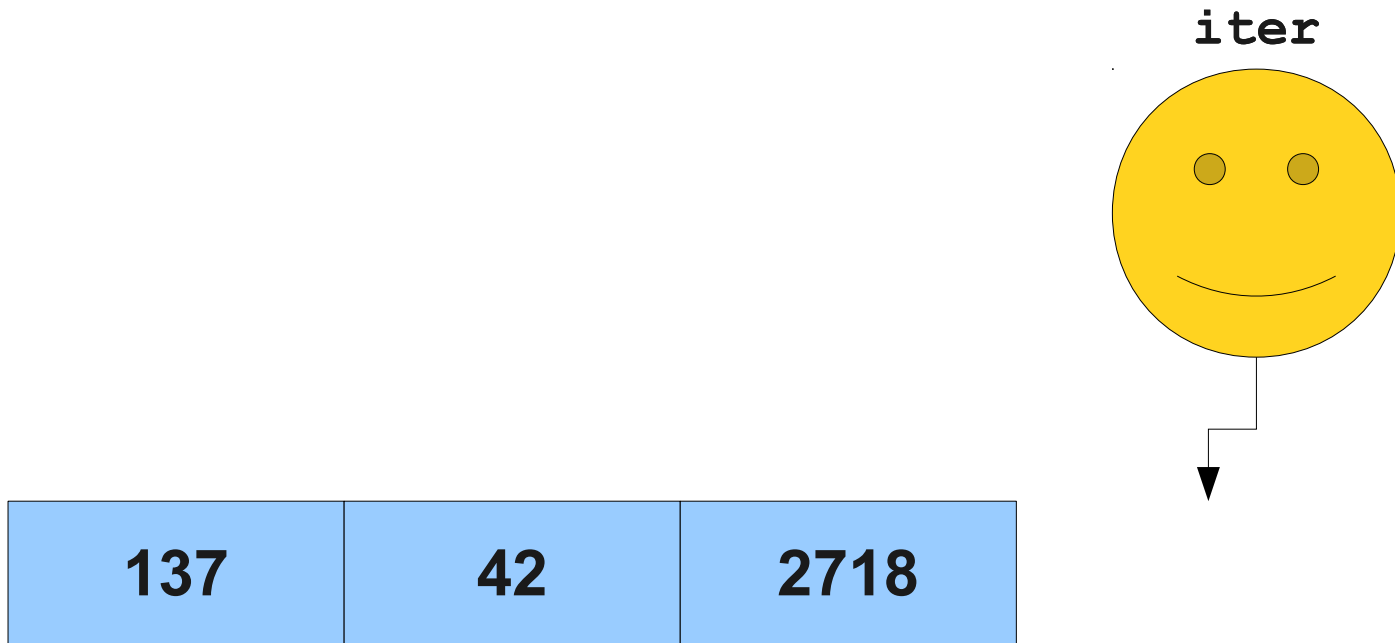
Java Iterators


```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```

Java Iterators

iter

137	42	2718

```java
ArrayList<Integer> myList = /* … */

Iterator<Integer> iter = myList.iterator();
while (iter.hasNext()) {
    Integer curr = iter.next();

    /* … use curr … */
}
```
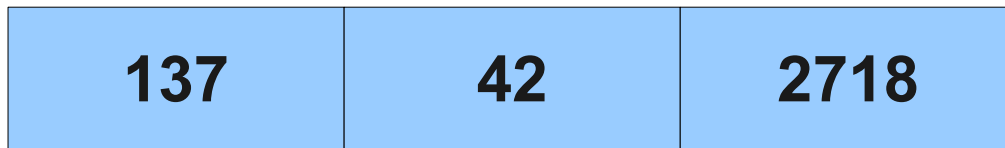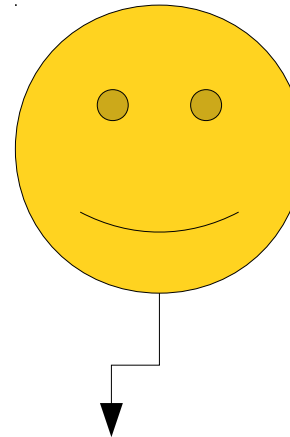
A Use Case for Iterators

- Because all collections have iterators, a method can return an iterator to indicate "here is some data to look at."

- Internally, that data can be stored in any format.

- Separates the **implementation** (how the class works) from the **interface** (how the class is used).

A Word of Warning

- The following will loop forever on a nonempty collection:

```
while (collection.iterator().hasNext()) {

    /* … */

}
```

- Every time that you call `.iterator()`, you get back a new iterator to the start of the collection.
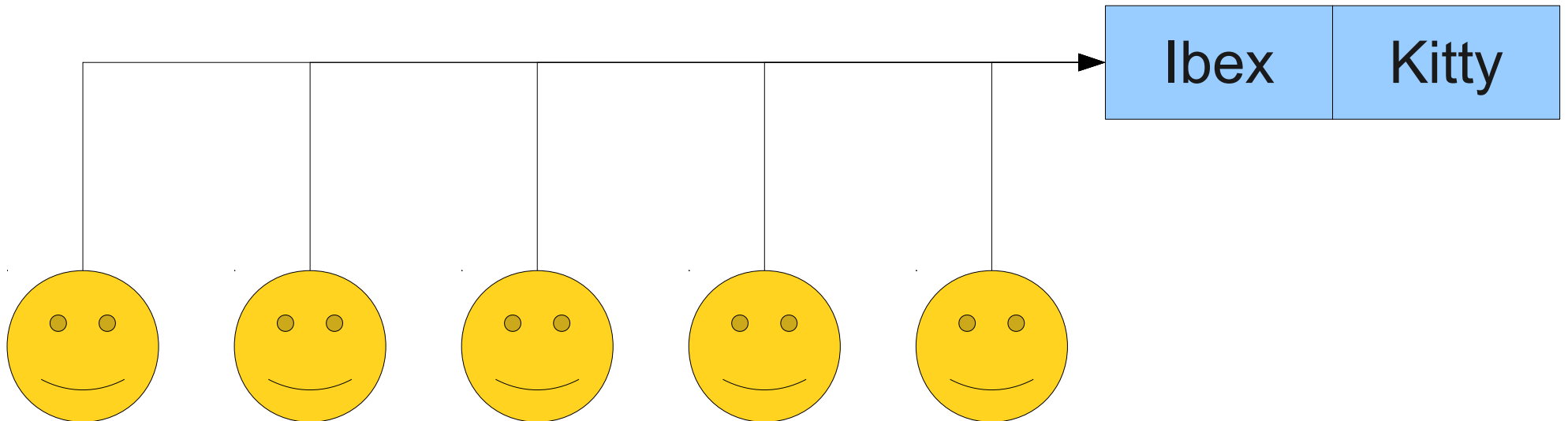
A Word of Warning

- The follow... ty collection:

```
while (c... {
    /* …
}
```

- Every tim... et back a new itera...

...ex	Kitty
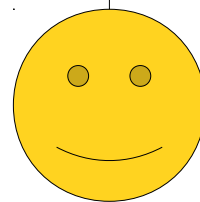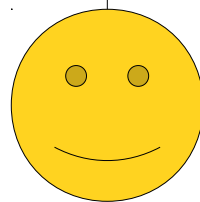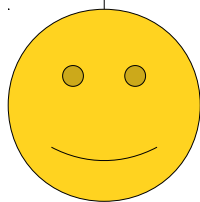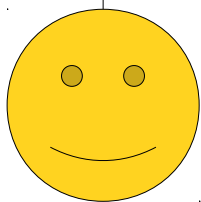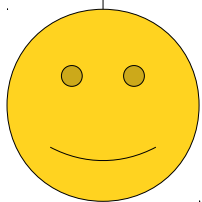
The Collections Framework

The Collections Framework

```
                    ┌─────────────┐                          ┌─────────┐
                    │ Collection  │                          │   Map   │
                    └─────────────┘                          └─────────┘
                         ▲                                        ▲
              ┌──────────┴──────────┐                   ┌─────────┴─────────┐
         ┌────────┐            ┌────────┐           ┌─────────┐        ┌─────────┐
         │  List  │            │  Set   │           │ HashMap │        │ TreeMap │
         └────────┘            └────────┘           └─────────┘        └─────────┘
             ▲                     ▲
     ┌───────┴───────┐     ┌───────┴───────┐
┌───────────┐ ┌────────────┐ ┌─────────┐ ┌─────────┐
│ ArrayList │ │ LinkedList │ │ HashSet │ │ TreeSet │
└───────────┘ └────────────┘ └─────────┘ └─────────┘
```
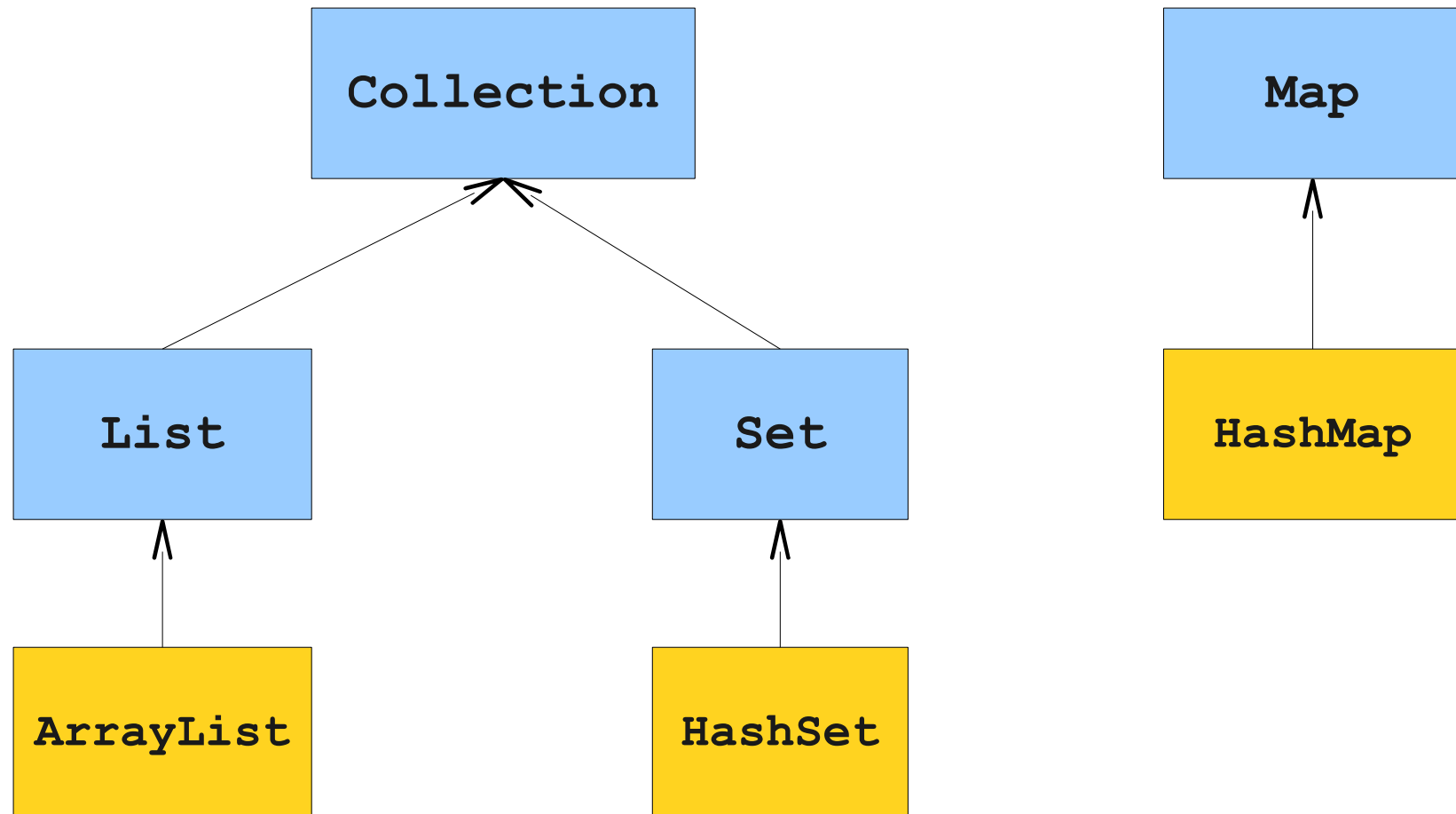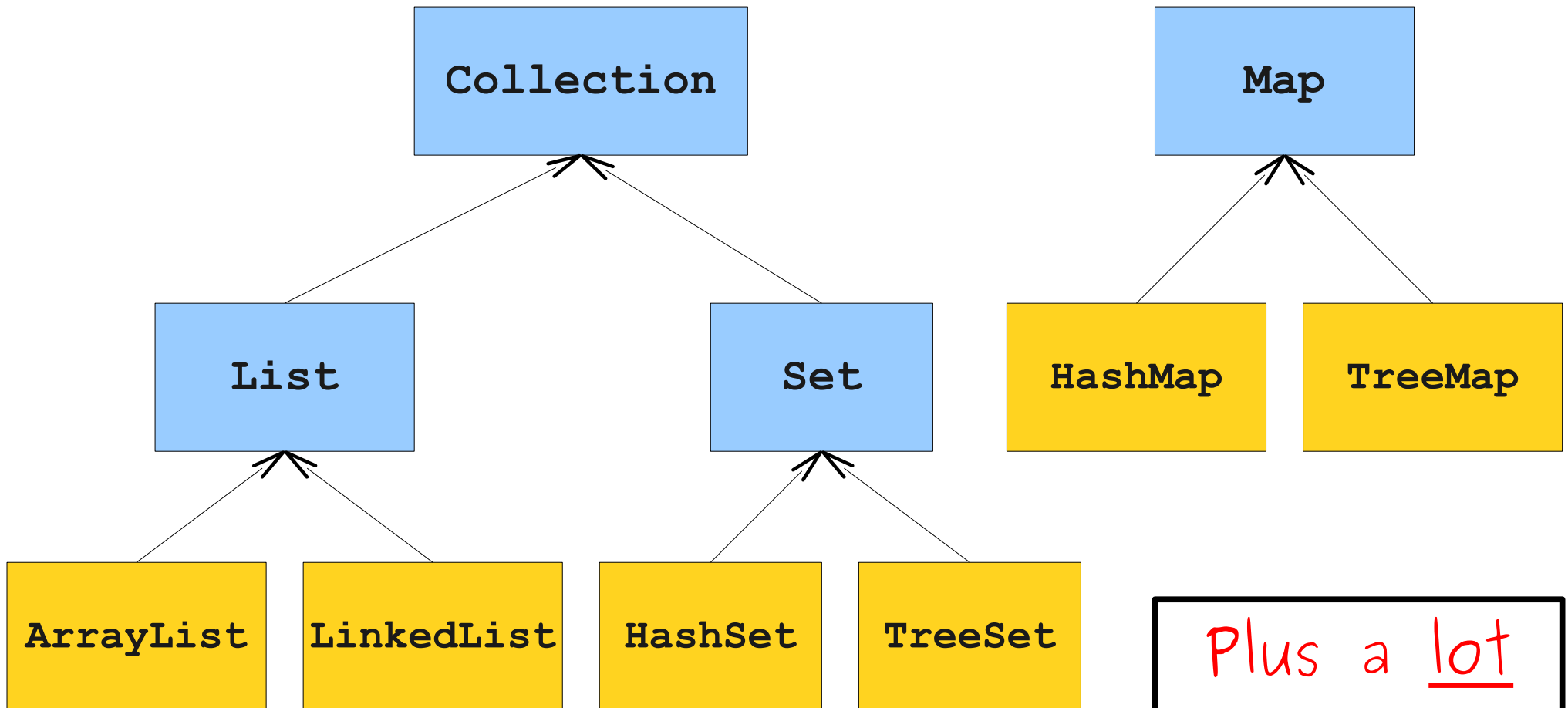
Plus a lot more!

The Collections Framework
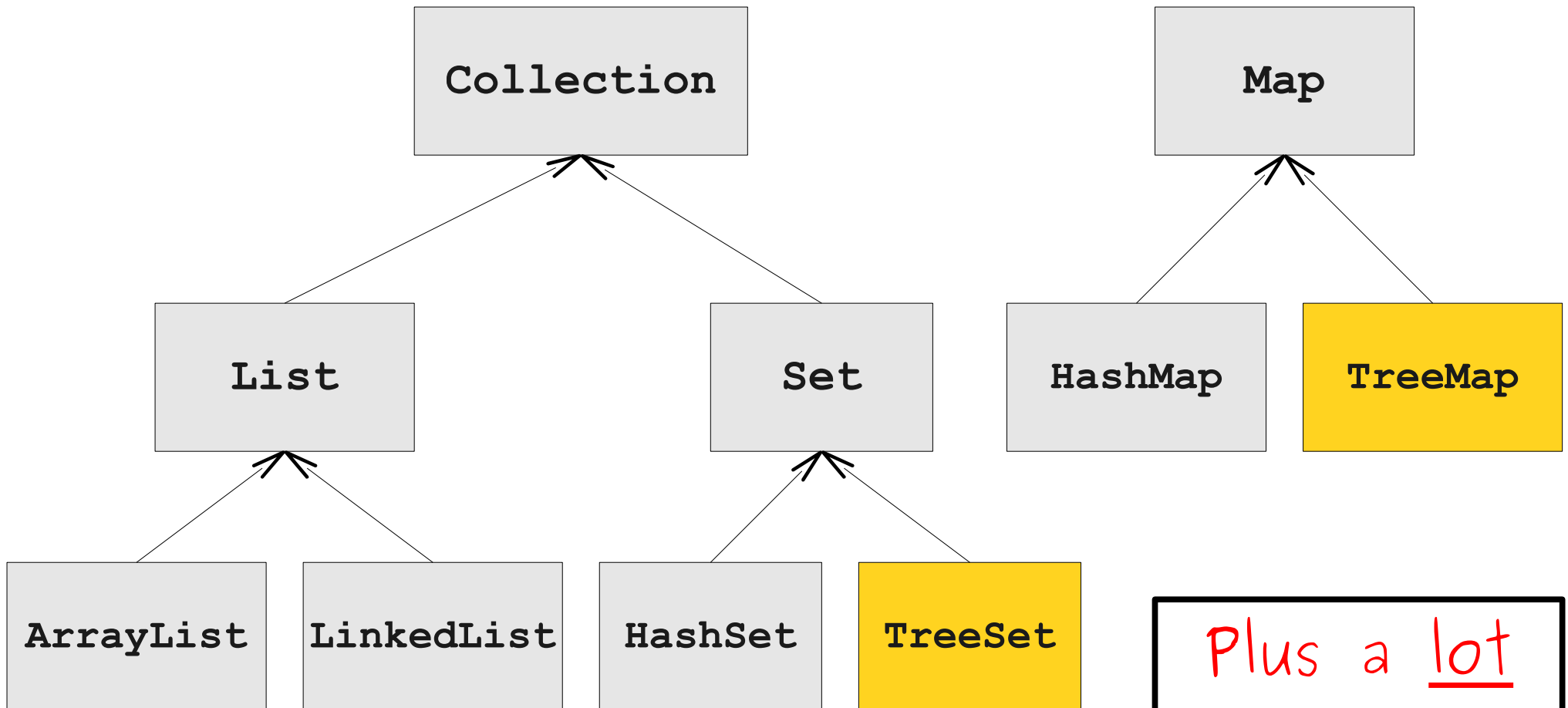
```
                  Collection                                  Map

        List                    Set              HashMap        TreeMap

  ArrayList   LinkedList    HashSet   TreeSet         Plus a lot
                                                        more!
```

TreeSet

- **TreeSet** is similar to **HashSet**, except that the values in a **TreeSet** are stored in sorted order.

- Iterating over a **TreeSet** guarantees that the elements are visited in ascending order.

- **TreeSet** is a bit slower than **HashSet**, so it's best used only when you really need things in sorted order.

Levels of Specificity

- To create a map, set, or list, you must choose a specific implementation (i.e. **`ArrayList`**, **`HashMap`**, etc.)

- You can store maps, sets, or lists in variables of type **`Map`**, **`Set`**, or **`List`**.

  - Similar to **`GObject`** versus **`GOval`**, **`GRect`**, etc.

- Lets you say "I just need key/value pairs" rather than "I need key/value pairs specifically stored as a **`HashMap`**"

TreeMap

- **TreeMap** is similar to **HashMap**, except that the *keys* in a **TreeMap** are stored in sorted order.

- Like **TreeSet**, iteration over the keys visits the keys in sorted order.

- The **TreeMap** has several impressive methods that don't exist on the normal **HashMap**.

- There is slight performance cost to using **TreeMap**.

Data With Stakes

Just how bad was the 2008 financial meltdown?

An Amazing Website

www.data.gov