

Inf1B

Collections

Fiona McNeill

adapting earlier versions by Perdita Stevens, Ewan Klein, Volker Seeker, et al.

School of Informatics

Rigidity of arrays

- ▶ Length of array is fixed at creation time.
- ▶ Can't be expanded.
- ▶ Can't be shrunk.
- ▶ Arrays are part of Java language — uses special syntax.
- ▶ E.g., `myArray[i]` for accessing the *i*th element.

Rigidity of arrays

- ▶ Length of array is fixed at creation time.
- ▶ Can't be expanded.
- ▶ Can't be shrunk.
- ▶ Arrays are part of Java language — uses special syntax.
- ▶ E.g., `myArray[i]` for accessing the *i*th element.

Arrays are not always optimal for handling data.

ArrayList

- ▶ Can grow and shrink as needed;
- ▶ provides methods for inserting and removing elements.

ArrayList

Declaration

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

- ▶ This is an array list of strings; counterpart to `String[]`.
- ▶ Angle brackets indicate that `String` is a **type parameter**.
- ▶ Can replace `String` with e.g. `HotelRoom` to get different array list type.
- ▶ In general: use `ArrayList<E>` to collect objects of type `E`; but `E` cannot be a primitive type.

ArrayList

Declaration

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

- ▶ This is an array list of strings; counterpart to `String[]`.
- ▶ Angle brackets indicate that `String` is a **type parameter**.
- ▶ Can replace `String` with e.g. `HotelRoom` to get different array list type.
- ▶ In general: use `ArrayList<E>` to collect objects of type `E`; but `E` cannot be a primitive type.

NB:

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

Since Java 8 the compiler can infer the type of the list in the constructor call.

ArrayList: Methods

- ▶ A newly constructed `ArrayList` has size 0.
- ▶ `ArrayList` has various methods, which allow us to:
 - ▶ keep on adding new elements;
 - ▶ remove elements.
- ▶ The size changes after each addition / removal.

ArrayList: Adding

Adding Elements

```
ArrayList<String> cheers = new ArrayList<String>();  
cheers.add("hip");  
cheers.add("hip");  
cheers.add("hooray");  
int n = cheers.size(); // n gets value 3
```

- ▶ `add()` appends each element to the end of the list.

ArrayList: Printing

Printing an ArrayList

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

Output

```
[hip, hip, hooray]
```

The compiler implicitly calls the `toString()` method of the `cheers` object which in turn calls the `toString()` method of each of its list elements.

ArrayList: More methods

Index of first occurrence

```
int ind = cheers.indexOf("hip"); // ind gets value 0
```

Adding element at an index

```
cheers.add(1, "hop"); // 2nd "hip" gets shunted along
```

Elements of cheers: ["hip", "hop", "hip", "hooray"]

ArrayList: More methods

`contains()`

```
boolean isHip = cheers.contains("hip"); // isHip is true
```

`remove()`

```
cheers.remove("hip"); // removes first occurrence of "hip"
```

Elements of cheers: "hop", "hip", "hooray"

`get(int index)`

```
cheers.get(0); // get the first element  
              // returns "hop"
```

ArrayList and Loops

Looping over ArrayList:

Standard for loop

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

ArrayList and Loops

Looping over ArrayList:

Standard for loop

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

Enhanced for again

```
for (String s : cheers) {  
    System.out.println(s);  
}
```

ArrayList and Loops

Enhanced for again

```
for (String s : cheers) {  
    System.out.print(s + "\thas index: ");  
    System.out.println(cheers.indexOf(s));  
}
```

Output

```
hop      has index: 0  
hip      has index: 1  
hooray   has index: 2
```

Wrapper Classes

Wrapper Classes:

- ▶ The type variable E in a generic type like `ArrayList<E>` must resolve to a **reference** type.
- ▶ So `ArrayList<int>` will not compile.
- ▶ All the primitive types can be turned into objects by using wrapper classes:

<i>Primitive Type</i>	<i>Wrapper Class</i>
boolean	Boolean
char	Character
double	Double
int	Integer
long	Long

NB Wrapper class names are always capitalized, always complete words.

Auto-boxing

- ▶ Conversion between primitive types and corresponding wrapper classes is automatic.
- ▶ Process of conversion is called **auto-boxing**

Auto-box example

```
Double batteryCharge = 2.75;  
double x = batteryCharge;
```

Auto-box example

```
ArrayList<Double> data = new ArrayList<Double>();  
data.add(29.95);  
double x = data.get(0);
```

Custom Types in ArrayLists

You can also put your own data types into an [ArrayList](#):

Circle List

```
ArrayList<Circle> data = new ArrayList<Circle>();  
Circle c = new Circle(10);  
data.add(c);  
data.get(0).enlarge(2);
```

Custom Types in ArrayLists

You can also put your own data types into an `ArrayList`:

Circle List

```
ArrayList<Circle> data = new ArrayList<Circle>();  
Circle c = new Circle(10);  
data.add(c);  
data.get(0).enlarge(2);
```

Some functionality will, however, not work properly unless you implement the necessary **Interfaces** (I will tell you more later).

Comparing Elements

```
Collections.sort(data);  
Collections.reverse(data);
```

Nested ArrayLists

Since I can use any object type as **type parameter**, I can also create ArrayLists of ArrayLists.

Daily Temperature Lists

```
ArrayList<ArrayList<Double>> dailyTemp =  
    new ArrayList<ArrayList<Double>>();  
dailyTemp.add(new ArrayList<Double>());  
dailyTemp.get(0).add(1.2);  
dailyTemp.get(0).add(1.4);  
dailyTemp.add(new ArrayList<Double>());  
dailyTemp.get(1).add(2.0);  
dailyTemp.get(1).add(1.9);
```

Output

```
[[1.0, 1.4], [2.0, 1.9]]
```

Lists of Lists

This is where type inference comes in handy.

Nested Lists

```
ArrayList<ArrayList<Double>> dailyTemp = new ArrayList<>();  
dailyTemp.add(new ArrayList<>());  
dailyTemp.get(0).add(1.0);  
dailyTemp.get(0).add(1.4);  
dailyTemp.add(new ArrayList<>());  
dailyTemp.get(1).add(2.0);  
dailyTemp.get(1).add(1.9);
```

Output

```
[[1.0, 1.4], [2.0, 1.9]]
```

Import

Importing:

- ▶ To get full access to Java API, we need to import classes.
- ▶ Not necessary if class is in same folder, or part of `java.lang` (e.g., `Math` library).
- ▶ To use `ArrayList`, add the appropriate `import` statement at top of your file:

Import example

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

Import

Importing:

- ▶ To get full access to Java API, we need to import classes.
- ▶ Not necessary if class is in same folder, or part of `java.lang` (e.g., `Math` library).
- ▶ To use `ArrayList`, add the appropriate `import` statement at top of your file:

Import example

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

Import example — Wrong!

```
import java.util.ArrayList<String>; // Don't use parameter
```

Java API

Look at sample Javadoc web page.

`https://docs.oracle.com/en/java/javase/11/docs/api/index.html`

Another word about removing elements

Let's assume you want to remove elements from a list of `Strings`.

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Charles");  
names.add("Marry");  
names.add("Peter");
```

Another word about removing elements

Let's assume you want to remove elements from a list of `Strings`.

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Charles");  
names.add("Marry");  
names.add("Peter");
```

```
names.remove("Peter");
```

Another word about removing elements

Let's assume you want to remove elements from a list of `Strings`.

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Charles");  
names.add("Marry");  
names.add("Peter");
```

```
names.remove("Peter");
```

This works if I know exactly which object to remove. But what if I want to remove every `String` that contains the substring "ar"?

Collection Iterators

Iterators are objects which allow you to iterate through each element of a collection.

```
// Declare by parameterising with the content type:
Iterator<String> nameIter;
// Initialise by asking the collection for an instance.
nameIter = names.iterator();
// Iterate using a while loop and hasNext():
while (nameIter.hasNext()) {
    // Access elements of the collection using getNext():
    String element = nameIter.getNext();
    // Remove elements while iterating using remove():
    nameIter.remove();
}
```

Maps / Associative Arrays

Associative Arrays

Associative array:

- ▶ Associates a collection of unique keys with values.
- ▶ Ordinary arrays: keys can only be integers.
- ▶ Associative arrays allow keys of many types, most notably strings.
- ▶ Examples:
 1. Given a person's name, look up a telephone number.
 2. Given an internet domain, look up its IP address.
 3. Given a geo-location, look up its GPS coordinates.
 4. Given a word, look up its frequency in a text.
- ▶ Relationship between key and value: **mapping**.

Java: associative arrays are implemented by type `HashMap`.

Map People to their Matric Nos.

Keys

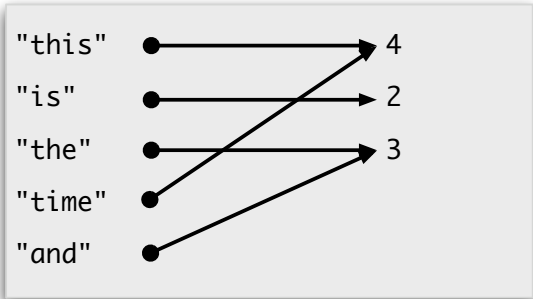
Values

Peter	●	→	s0189034
Michael	●	→	s0289125
Helen	●	→	s0378435
Mary	●	→	s0412375
John	●	→	s0456782

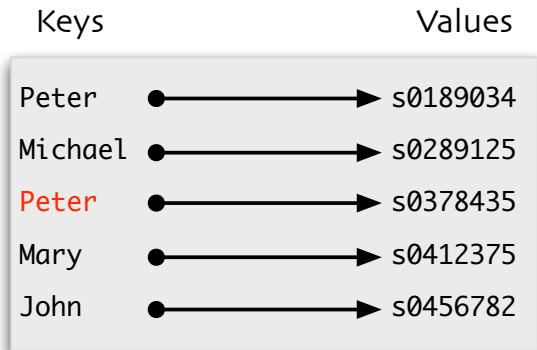
Map Words to Length

Keys

Values

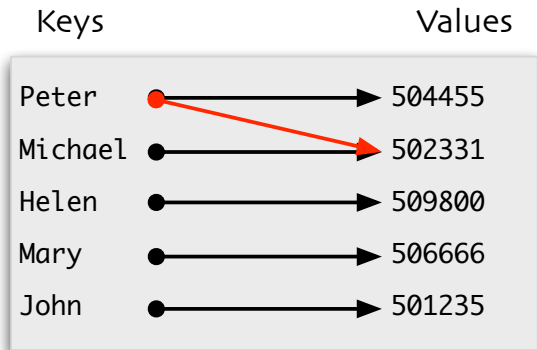


Map People to their Matric Nos: Wrong!



NB Keys must be unique.

Map People to their Telephone Nos: Wrong!



- ▶ A given key can only be mapped to **one** value.
- ▶ However, type of value can be array, or some other object.

HashMap

Import HashMap

```
import java.util.HashMap;
```

HashMap

Import HashMap

```
import java.util.HashMap;
```

Declare HashMap

```
HashMap<String, Integer> map  
    = new HashMap<String, Integer>();
```

- ▶ HashMap takes **two** type parameters.
- ▶ Here, String is type of key, Integer is type of value.

HashMap

Import HashMap

```
import java.util.HashMap;
```

Declare HashMap

```
HashMap<String, Integer> map  
    = new HashMap<String, Integer>();
```

- ▶ HashMap takes **two** type parameters.
- ▶ Here, String is type of key, Integer is type of value.

NB: There is a different type called Hashtable which is the same for our purposes.

Mapping Words to their Lengths

Goal: Given a string of words, derive an associative array that maps each word to its length.

1. Split the string on whitespace, to yield words.
2. For each word w , add it as a key, and associate it with value $w.length()$.
3. When we add the same key again, we overwrite the previous association — wasteful but harmless in this case.

`split()` method of `String`

```
String sent = "this is the time and this is the record of the time";  
String[] words = sent.split(" "); // split on whitespace
```

HashMap: Add and retrieve mappings

- ▶ `put(Key, Value)`: put *Value* as the value of *Key* in `wordLengths`.

```
HashMap<String, Integer> wordLengths = new HashMap<String, Integer>();  
for (String word : words) {  
    wordLengths.put(word, word.length());  
}
```

add a key-value pair to the mapping

- ▶ `get(Key)`: get the value of *Key* in `wordLengths`.

```
int w1 = wordLengths.get("record"); // value is 6
```

HashMap: Add and retrieve mappings

`wordLengths.keySet()`: the set of keys in `wordLengths`.

```
[of, record, time, is, the, this, and]
```

HashMap: Add and retrieve mappings

`wordLengths.keySet()`: the set of keys in `wordLengths`.

```
[of, record, time, is, the, this, and]
```

Q How do we list **all** key-value pairs in a map?

A Loop over the set of keys.

```
for (String key : wordLengths.keySet()) {  
    System.out.printf("%s => %s\n", key, wordLengths.get(key));  
}
```

Output

```
of => 2  
record => 6  
time => 4  
is => 2  
the => 3  
this => 4  
and => 3
```

HashMap: Printing

Output

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

Output

```
{of=2, record=6, time=4, is=2, the=3, this=4, and=3}
```

Format is `{Key1=Value1, Key2=Value2, ... }`

Custom Types in HashMaps

You can also put your own data types into a [HashMap](#):

Circle Values

```
HashMap<String, Circle> data = new HashMap<String, Circle>();  
data.put("Small", new Circle(2));  
data.put("Large", new Circle(200));
```

Custom Types in HashMaps

You can also put your own data types into a [HashMap](#):

Circle Values

```
HashMap<String, Circle> data = new HashMap<String, Circle>();  
data.put("Small", new Circle(2));  
data.put("Large", new Circle(200));
```

Using custom types as keys, is more tricky: You will have to make sure they have an equals method and produce the same hash code.

Nested HashMaps

Similar to [ArrayLists](#), you can also write nested [HashMaps](#).

Circle Organiser

```
HashMap<String, ArrayList<Circle>> data = new HashMap<>();
data.put("Large", new ArrayList<>());
data.put("Small", new ArrayList<>());
data.get("Large").add(new Circle(200));
data.get("Large").add(new Circle(300));
data.get("Small").add(new Circle(5));
data.get("Small").add(new Circle(6));
System.out.println(data);
```

Let's assume [Circle](#) implements `toString`.

Output

```
Small=[5, 6], Large=[200, 300]
```

Summary ArrayList & HashMap

- ▶ Use `ArrayList` when you want your arrays to be able to grow, or you want to easily insert and remove items in the middle of an array.
- ▶ Use `HashMap` when you want to use keys other than a predetermined list of integers.
- ▶ For more on `ArrayList` and `HashMap`, look at the Java API: <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>
- ▶ Iterate collections with ease using an `Iterator` object.

Reading

Objects First

Chapter 4 *Grouping Objects*