"Venus Williams Opens Jamba Juice At Dupont Circle" by www.glynlowe.com https://flic.kr/p/cuMMw3

Java
Packages, Collections

Packages

Package: groups together related resources (usually classes)

Why put code in a package?

- Makes it obvious that types are related
- Makes it possible to find types for a specific purpose (given good package name)
- Type names won't conflict with names from other packages
- Types within package can have unrestricted access to one another
  - *While restricting access for types outside the package*

Creating a package

Choose a name

Put a `package` statement at the top of every source file for that package

```
package my.package.name;
```

Ensure that all source files are in a directory corresponding to the package name

If you don't use a `package` statement, then all files are in the default package

Accessing package members

To use a public type from outside the package, you need to **import** it

```
// At top of source file, after package statement
import java.util.Scanner;
// … later on, inside the class …
Scanner stdin = new Scanner(System.in);
```

Java Collections framework

A standard set of built-in classes for representing and manipulating collections

Each Collections class groups **related elements** into a **single unit**

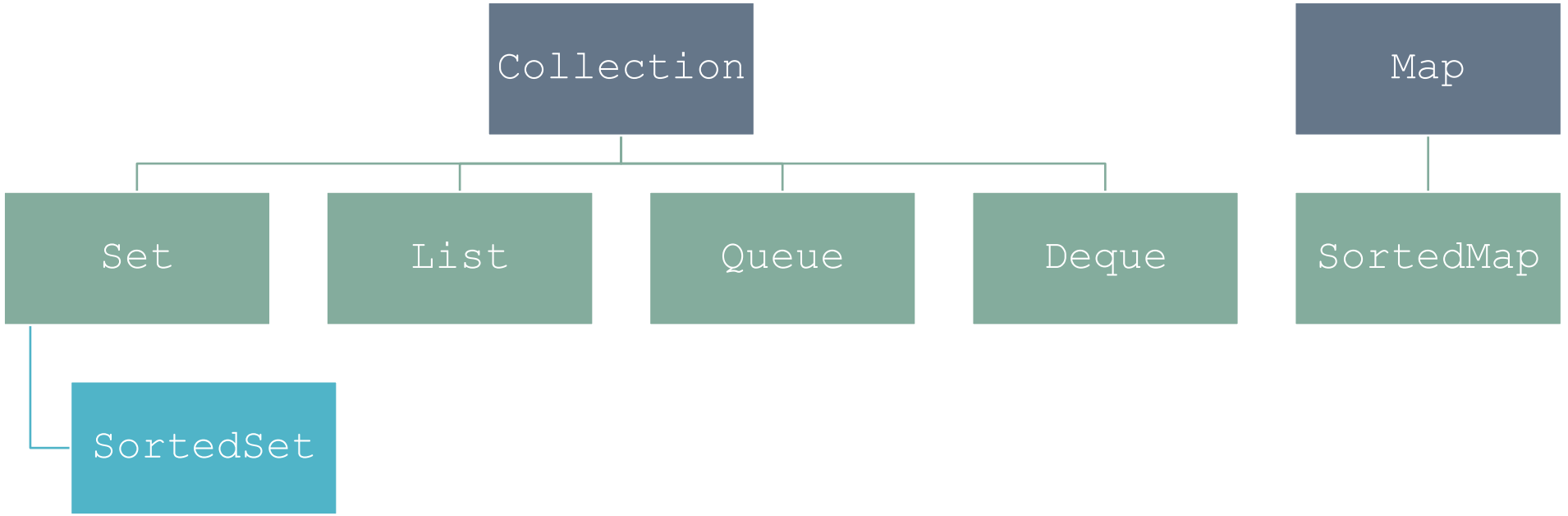
Examples:

- **ArrayList** – acts like a variable-length array
- **HashSet** – a group of unique elements
- **Stack** – a list with last-in/first-out semantics
- **HashMap** – a **dictionary** (e.g., a telephone directory)

Structure of Collections framework

Base class:* `java.util.Collection`

Methods: `add(), remove(), contains(), size(), toArray()`



** Actually, everything in this picture is an interface*

Advantages of Collections

Reduces programming effort by providing pre-written data structures and algorithms

Increases performance by providing high-performance implementations
    Implementations are interchangeable – can switch to tune performance

Provides interoperability by allowing Collections to be passed back and forth

Reduces effort to learn new APIs by providing a common interface

Reduces effort to design APIs by giving design specifications

Fosters software reuse by providing a standard interface

(List adapted from https://docs.oracle.com/javase/tutorial/collections/intro/index.html)

java.util.ArrayList

A Collections class (specifically, a `List`) that implements **variable-length** arrays

More flexible than built-in arrays, but less efficient

Acts as a wrapper around an underlying array that grows and shrinks dynamically

`ArrayList` is a **class** – so elements are added and removed by **methods**
(Not by built-in Java syntax as with normal arrays)

It has a **capacity** (size of internal array) and a **size** (number of elements in the list)

Capacity is increased when necessary – purely internal

Size is increased/decreased as elements are added and removed, and checked for operations

In general: IndexOutOfBoundsException if `(index < 0 || index >= size())`

Array vs ArrayList at a glance

Operation	Array	ArrayList
Declaration	`String[] strings;`	`ArrayList<String> strings;`
Initialisation	`strings = new String[10];`	`strings = new ArrayList<>(10);`
Setting element	`strings[5] = "foo";`	`strings.set(5, "foo");`
Accessing element	`String s = strings[5];`	`String s = strings.get(5);`
Getting size	`int n = strings.length;`	`int n = strings.size();`
Adding element	*n/a*	`strings.add("foo");` `strings.add(5, "foo");`
Removing element	*n/a*	`strings.remove("foo");` `strings.remove(5);`

Finding element	`Arrays.binarySearch(strings, "foo");`	`strings.contains("foo");`
		`strings.indexOf("foo");`
		`strings.lastIndexOf("foo");`

`List` vs `ArrayList`?

`List` is the **high-level Collection type** (actually it's an interface)
- Specifies methods including **add**, **clear**, **isEmpty**, **remove**, **set**, …

`ArrayList` is the **specific type of List**
- Provides a concrete implementation
- Additional methods related to capacity

When to use which?
- Use `ArrayList` …
  - *When initialising a new variable*
  - *If you want to manipulate capacity*
- Use `List` all other times – allows implementations to be swapped cleanly

Generic types


```
List<String> strList = new ArrayList<>();
```

Collection classes are **type-parameterised**

- The type specified in angle brackets after the name specifies the type of the elements stored in that Collection
- If you don't specify any type, then it will use `java.lang.Object`

(Polymorphism: subclasses of specified type will also be accepted)

Generic types were added to Java in Java 1.5 (2004)

Why use generic types?

Compile-time error checking

```
List<String> strList = new ArrayList<>();
strList.add ("foo");
strList.add (new java.util.Scanner()); // fail
```

Iteration can be much cleaner (especially with new-style iteration)

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

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

Primitive types and generics

The <*type*> generic parameter needs to be a **class**

Primitive types will not work!

~~`List<int> intList;`~~

Solution: Use **wrapper** classes (`int/Integer, long/Long`, etc.)

```
List<Integer> intList = new ArrayList<>();
```

But you don't want to have to do this all the time …

```
Integer i2 = new Integer (i);
intList.add (i2);
int value = intList.get(5).intValue();
```

Boxing and unboxing

Good news: Java **automatically** converts between wrapper classes and primitive types

(Also since Java 1.5)

```
List<Integer> intList = new ArrayList<>();
intList.add (5);
intList.add (10);
int value = intList.get (0);
Integer value2 = intList.get(1) * 100;
```

Sets

Interface: `java.util.Set`

- Concrete implementations: `HashSet, TreeSet, LinkedHashSet`

Differences to List

- Cannot contain duplicate elements
  - *add() method enforces this – returns true/false indicating if element was already in set*
- Two sets are equal if they contain the same elements, regardless of implementation

Maps

Interface: `java.util.Map`

- Concrete implementations: `HashMap, TreeMap, LinkedHashMap`

Provides a mapping from keys to values

- Cannot contain duplicate keys
- Each key maps to exactly one value

Useful methods:

- `get(key)` – return the value associated with a key (null if no value)
- `put(key, value)` – set the new value associated with that key