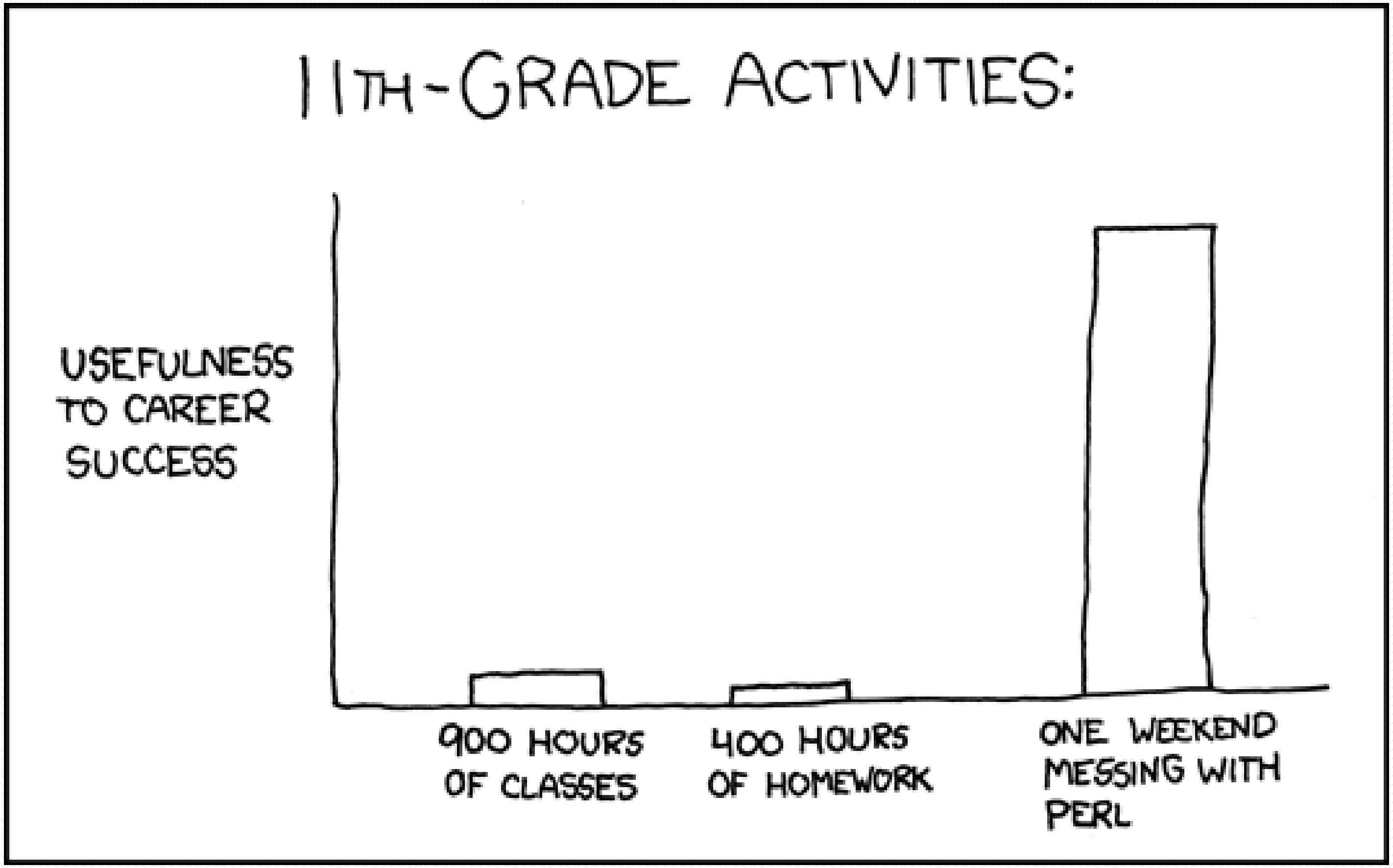
Building Java Programs

Chapter 10, 11

Lecture 22: 143 Preview

optional reading: 10.1, 11.1 - 11.3


11TH-GRADE ACTIVITIES:
USEFULNESS TO CAREER SUCCESS
900 HOURS OF CLASSES
400 HOURS OF HOMEWORK
ONE WEEKEND MESSING WITH PERL

Problems with arrays

- We need to know the size when we declare an array, and we can't change it later
  - Can't add more elements
  - Can't shrink the array to avoid wasting space
    - Could get around this with Arrays.copyOf

- No method to find the index of a given object in an array
    - Could use Arrays.sort and Arrays.binarySearch, but this could be inefficient

- No method to add/remove from the middle of the list without overwriting a given element
    - We'd have to write our own methods

Problems with arrays

- We need to know the size when we declare an array, and we can’t change it later
  - Can’t add more elements
  - Can’t shrink the array to avoid
    - Could get around t
- No an array
    - binarySearch, but this
- No m add/remove from the middle of the list without overwriting a given element
    - We’d have to write our own methods

- A Better Solution:

ArrayLists

ArrayLists

- Arrays that dynamically resize themselves to accommodate adding or removing elements

`ArrayList` declaration

```
Arrays:    type[]          name = new type[length];
ArrayList: ArrayList<type> name = new ArrayList<type>();
```

- Example:
  `ArrayList<String> words = new ArrayList<String>();`
- Note – the type must be an object, not a primitive type. You can mostly just use primitive types because of autoboxing and unboxing, but you must declare object types such as
  - `Boolean`, `Integer`, `Double`, `Character`
- Need to import `java.util.*;`

ArrayList Methods

Method name	Description
add(**obj**)	Adds obj to the end of the list
add(**index**, **obj**)	Adds obj at the specified index, shifting higher-index elements to make room
contains(**obj**)	Whether the list contains obj
get(**i**)	Get the object at index i
indexOf(**obj**)	Find the lowest index of obj in the list, -1 if not found
lastIndexOf(**obj**)	Find the highest index of obj in the list, -1 if not found
remove(**i**)	Remove the element at index i
remove(**obj**)	Remove the lowest index occurrence of obj
set(**i**, **obj**)	Set the element at index i to obj
size()	The number of elements in the list

Cities revisited

- Remember our Cities example?

```
City        State Population Latitude  Longitude
Seattle        WA     616627 47621800 -122350326
```

- There was information about which state each city is in that we just ignored.
  - Let’s add a legend that shows which states the cities we plotted were from
  - Why would this have been difficult with standard arrays?
  - Let’s pick a different color for each state, and color all cities in that state with that color
  - Let’s add that color to our legend as well
  - How will we convert a state (String) to a color (3 ints)?

`String` to `Color` using `hashCode()`

- All objects have a method called hashCode that returns a number representing that object
- The `Random` object has a constructor `Random(seed)`
  - The seed determines future random numbers
- The `Color` object has a constructor that takes 3 `int`s (red, green, and blue)
- We can use the state's hash code to seed a `Random` object and then generate the red, green, and blue components of a `Color`.
  - This guarantees that for a given state, we will always generate the same color, but different states will likely have different colors

Solution details

- Our method converting `String` to `Color`

-
  ```
  public static Color getColor(String state) {
     Random r = new Random(state.hashCode());
     return new Color(r.nextInt(256),
           r.nextInt(256), r.nextInt(256));
  }
  ```

- Assume we have an `ArrayList<String>` called `states` and a `Graphics` object called `g`
- As we encounter each state that we'll plot

```
if (!states.contains(state)) {
   states.add(state); // keep track of states that we plotted
}
g.setColor(getColor(state));
// Plot the city, it will be the correct color
```

Solution details (cont)

- Assume we have an `ArrayList<String>` called `states`, a `Graphics` object called `g`, and `int` coordinates `x` and `y`
- For drawing the legend

```
Collections.sort(states);
for (int i = 0; i < states.size(); i++) {
   String state = states.get(i);
   g.setColor(getColor(state));
   g.drawString(state, x, y);
   // update x and y
}
```

Problems

- For large ArrayLists, contains can be inefficient

- We have to generate the Color from the state
  - What if we wanted to associate an arbitrary Color with each state?
    - We could make parallel ArrayLists, that store Strings and Colors, but we'd get thrown off when we sort the states for the legend
    - We could create a new object type with a String and a Color field, but that's a lot of work (Collections won't be able to sort an ArrayList of an arbitrary type either)

Problems

- For large ArrayLists, contains can be inefficient
- We have to generate the Color from th
  - What if we wanted to associ h each state?
    - We coul d or the
    - a String and a Color field, ections won't be able to sort an A bitrary type either)

- A Better Solution: HashMaps

HashMaps

- A data structure that associates keys and values
- The keys and values can be arbitrary types, but all the keys must be the same type, and all the values must be the same type. The keys must be unique!
- Think of it as an array that can be indexed on any type, not just `int`s

key	*"foo"*	*"bar"*	*"baz"*
value	12	49	-2

`HashMap` declaration

```
HashMap<key_type, value_type> name =
   new HashMap<key_type, value_type>();
```

- Example:

  ```
  HashMap<String, Color> colors =
     new HashMap<String, Color>();
  ```

- Note – the type must be an object, not a primitive type. You can mostly just use primitive types because of autoboxing and unboxing, but you must declare object types such as
  - `Boolean`, `Integer`, `Double`, `Character`

- Need to import `java.util.*;`

`HashMap` Methods

Method name	Description
`containsKey(`**obj**`)`	Whether obj is a key in the map
`containsValue(`**obj**`)`	Whether obj is a value in the map
`get(`**obj**`)`	Get the value associated with the key obj, `null` if key is not found
`keyset()`	Gets the `Set` of all the keys in the map
`put(`**key, val**`)`	Adds a key/value pairing to the map
`remove(`**obj**`)`	Remove the mapping for key obj, and return the value that was associated with it, `null` if key is not found
`size()`	The number of entries in the map
`values()`	Gets a `Collection` of all the values in the map

Cities revisited

- We’ll no longer have to generate a `Color` from a `String`
- We can just associate `String`s and `Color`s and keys as values in the map
- Without going into detail, for large data sets, adding, removing, and finding entries in a `HashMap` is faster than adding, removing, and finding elements in an `ArrayList`
  - `ArrayList` is an ordered list, while `HashMap` isn’t. Maintaining that order takes time.

Solution details

- Assume we have a `HashMap<String, Color>` called `colors` and a `Graphics` object called `g`
- As we encounter each state that we'll plot

```
if (!colors.containsKey(state)) {
   Random r = new Random();
   colors.put(state, new Color(r.nextInt(256),
      r.nextInt(256), r.nextInt(256)));
}
g.setColor(colors.get(state));
// Plot the city, it will be the correct color
```

Solution details (cont)

- Assume we have a `HashMap<String, Color>` called `colors`, a `Graphics` object called `g`, and `int` coordinates `x` and `y`
- For drawing the legend

```
for (String state :
      new TreeSet<String>(colors.keySet())) {
   g.setColor(colors.get(state));
   g.drawString(state, x, y);
   // update x and y
}
```

- This is called a foreach loop. A `TreeSet` doesn't have indexes, so we can't get the element at index `i`