CS 213 : Software Methodology

Spring 2023

Sesh Venugopal

Feb 1

Design Aspects of Static Members

Static/Non-static Mix Example: Design Choices

Static/Non-Static Mix: Another Example

- Want to parse a string into an integer, e.g. “123” -> 123 – where to provide this functionality?

OPTIONS:

- Have a `String` instance method, say, `parseAsInteger` that returns an `int`, e.g.

```
int i = “123”.parseAsInteger();
```

Bad design: An instance method should be applicable to ALL instances. But not all strings are parsable as integers

- Have a `String` static method, say, `parseAsInteger` that returns an `int`, e.g.

```
int i = String.parseAsInteger(“123”);
```

- Have an `Integer` static method, say, `parseInt` that returns an `int`, e.g.

```
int i = Integer.parseInt(“123”);
```

- Of the second and third choices, which one is better? Why?

`Integer.parseInt` is better

Think of converting strings to doubles, floats also –
having all these types of conversions in `String` would require `String` to know about formats of other types, which is NOT its business.
Best to localize custom functionality in the corresponding target (converted type) classes.

Global Storage – Utility Class

Class for “Global” Storage

”Global” variables that need to be shared by multiple classes/objects can be housed as static fields in a class:

```
public class Storage {
   static int x;
   static float y;
   static String color=“blue”;
   static float y;
   ...
}
```

Like the `Math` class, this is a utility class – every field is `static`

If the design choice is to make the fields private, then static getter and setter methods can be defined