CS61B, Spring 2003 **Discussion #3** **Amir Kamil**
UC Berkeley **2/10/03**

Topics: Scope, Parameter Passing, Exceptions

1 Scope

- The scope of a field is the entire class definition.
- The scope of a method parameter is the entire method body.
- The scope of a variable declared within a method is from where it is declared to the end of the block in which it was declared. A block is delimited by braces, `{` and `}`. Consider the loop equivalencies from last week. Why must the equivalent code to a for loop be surrounded by braces? It is because the scoping rules for a for loop require them. The scope of the initialization clause of a for loop is the body of the loop, but without the surrounding braces in the equivalent code, the scope is the entire block in which the loop is located, which is incorrect.
- Variable names refer to the variable of that name in the inner-most scope:

```
public class Computer {
  String os;
  public Computer(String os) {
  /* Here, "os" would refer to the object passed in as the
   * argument
   */
  }
}
```

- Inner variables need not be of the same type as outer variables of the same name:

```
public class Computer {
  int os;
  public Computer(String os) { // this works fine
    ...
  }
}
```

2 Parameter Passing

- Parameter passing is ALWAYS pass-by-value. Remember that containers for primitive types contain values of their respective types, while containers for objects contain values that are references to objects. The value of this container is what is passed when a method is called.

```
static void foo() {
  String s  = "hello world";
  bar(s);
  stdout.format("%s\n").put(s);
}
static void bar(String s) {
  s = "goodbye world";
}
```

What is printed to the screen when `foo()` is called? Figures 2 through 5 show the environment diagrams during its execution. As you can see, `bar()` only modifies its local copy of the variable `s`, which has no effect on `foo()`'s copy. So `hello world` is printed out to the screen.

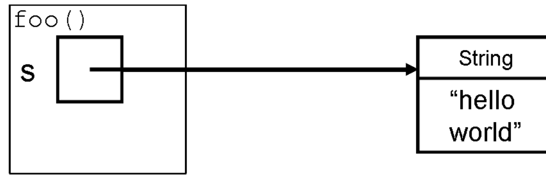

Figure 1: The environment of `foo()` prior to the call to `bar()`.

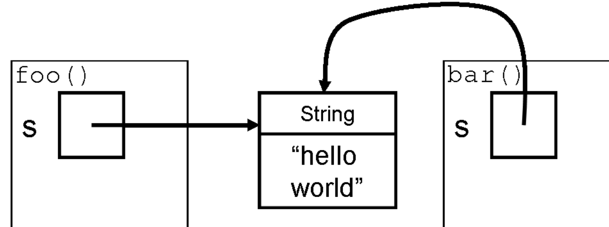

Figure 2: The environment of `foo()` and `bar()` immediately after the call to `bar()`.

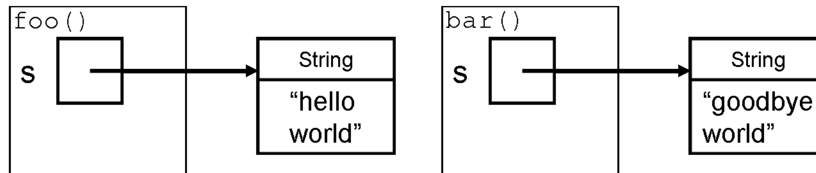

Figure 3: The environment of `foo()` and `bar()` during the call to `bar()`.

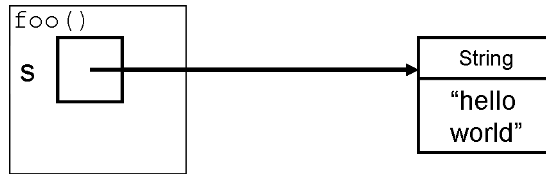

Figure 4: The environment of `foo()` after the call to `bar()`.

3 Exceptions

- Exceptions are used to signify a run-time error in the program.
- They don't necessarily mean your code is bad. For example, an `IOException` will result if a file you are reading from is corrupt, or an `IllegalArgumentException` if an outside class calls one of your methods using invalid parameters.
- Exceptions are objects, so they are instantiated like objects and you can even define your own exceptions. The only requirement is that they inherit `Throwable`. (This will make more sense when we do inheritance.)

```
public class CrashException extends Exception {
  // We don't really need to write any code.
}
```

- Exceptions can be thrown using `throw E;` where `E` is an exception object (e.g. `throw new IOException();`).
- Exceptions are either handled using `try {..} catch (ExpType e) {..}` or passed on to the calling method by declaring it to be thrown.

```
public String readLine() throws IOException {..} // passes exception on

public String readLine() {
  try {
    char[] cbuf = new char[1000];
    str.read(cbuf, 0, 1000);
    return new String(cbuf);
  } catch (IOException ie) {
    return null;
  } catch (Exception e) {
    System.out.println("Error: unknown exception");
    return null;
  }
  return null;
}
```

- When an exception is thrown within the try block, the JVM tries to match the exception with the exception types in the catch blocks. The first catch block for which the exception type is a subtype of the exception declared in the catch block is executed. For example, if the two catch blocks in the previous code were switched, an `IOException` would result in the Exception catch block to execute, since `IOException` is a subtype `Exception`.
- Exceptions are `checked` or `unchecked`. Checked exceptions must be caught or declared, unchecked don't have to be. You don't need to memorize which exceptions are checked or unchecked, just let the compiler tell you.
- Exceptions that are subtypes of `Error` or `RunTimeException` are unchecked.