

Lecture 23: Interfaces

Last time:

1. Explicit array initialization
2. Arrays as arguments
3. Arrays of objects
4. Privacy leaks

Today:

1. Array Copying
2. Interfaces



Project #5 Assigned!

- Project due Thursday, 4/5 at 11 pm
- Project is **closed**
 - You must complete the project by yourself
 - Assistance can only be provided by teaching assistants (TAs) and instructors
 - You must not look at other students' code
- Start now!
 - Read entire assignment from beginning to end before starting to code
 - Check out assignment now from CVS
 - Follow the instructions *exactly*, as much of grading is automated

Three Ways of Copying Arrays of Objects



```
CD[] d = {new CD("Secada's Greatest Hits", "John Secada"),
          new CD("Irreplacable", "Beyonce")};
```

- **Reference**

```
CD[] e = d;
```

- Creates alias between *e*, *d*

- e.g.

```
d[0] = new CD("Laundry Service", "Shakira");
```

Changes *e*[0] also

- **Shallow**

```
CD[] e = new CD[d.length];
```

```
for (int i = 0; i < d.length; i++)
```

```
    e[i] = d[i];
```

- No aliasing at array level ...

- ... but aliasing at cell level

- e.g.

```
d[0] = new CD();
```

No change to *e*[0]

- But if CDs were mutable (they are not, but *RewritableCDs* are!) doing something like this:

```
    d[1].rewrite("Greatest Hits", "Elton John");
```

Changes *e*[1] also. (Luckily, there is no *rewrite* method for immutable CDs, just *RewritableCDs*.)

- **Deep**

```
CD [] e = new CD[d.length];
```

```
for (int i = 0; i < d.length; i++)
```

```
    e[i] = new CD(d[i]);
```

- Note use of CD copy constructor!

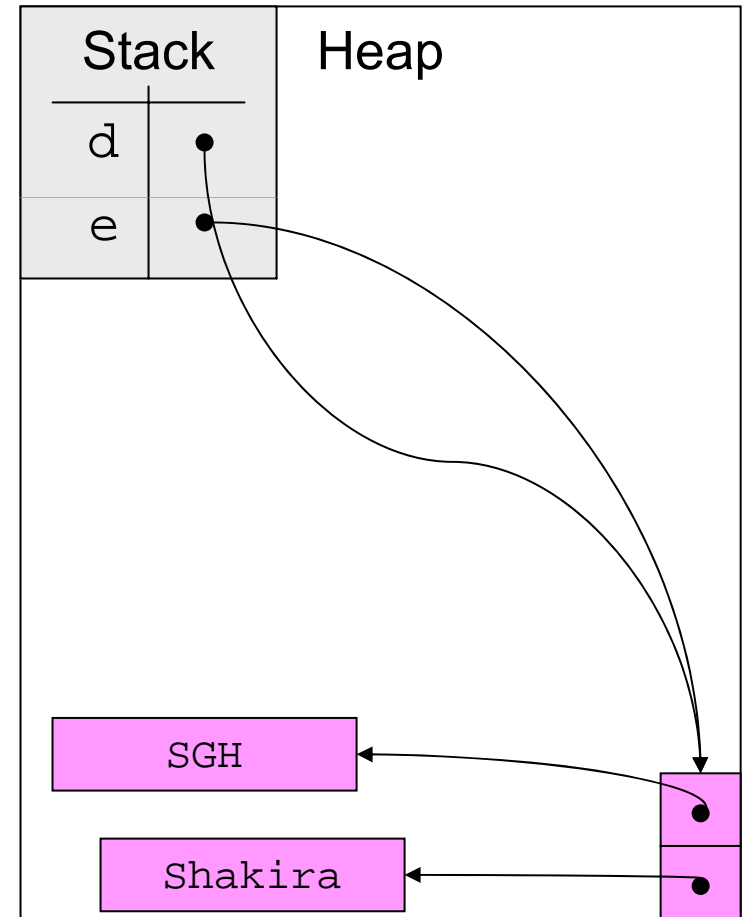
- No dependencies between *e*, *d*

- Not necessary for CD, but definitely necessary for *RewritableCD*!

Reference Copying

```
CD[] d = {
    new CD("SGH", ...),
    new CD("Shakira", ...)
};
```

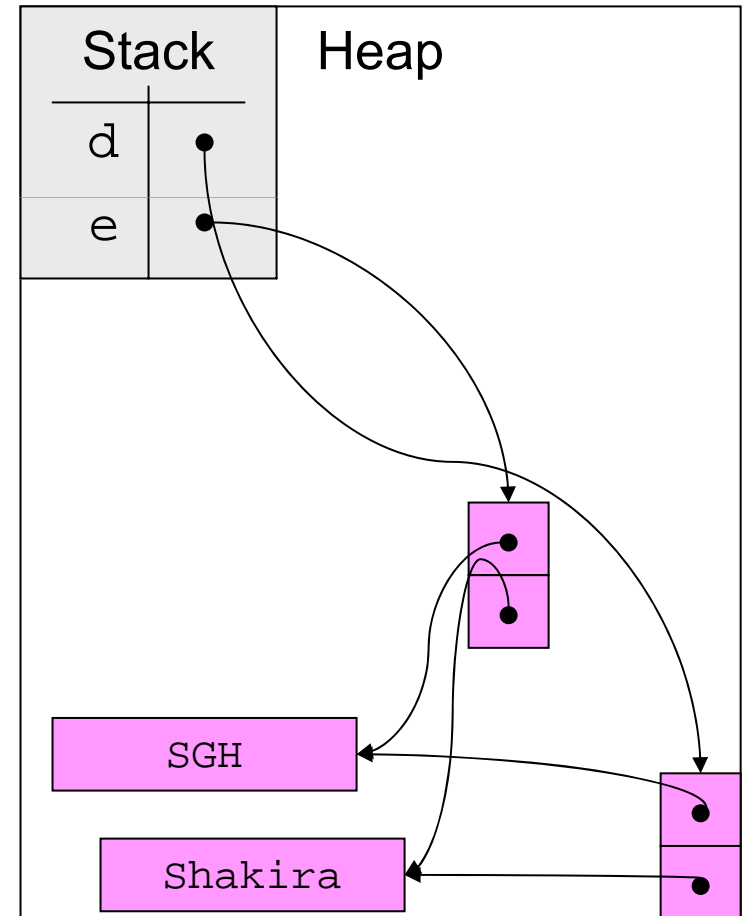
```
CD[] e = d;
```



Shallow Copying

```
CD[] d = {
    new CD("SGH", ...),
    new CD("Shakira", ...)
};
```

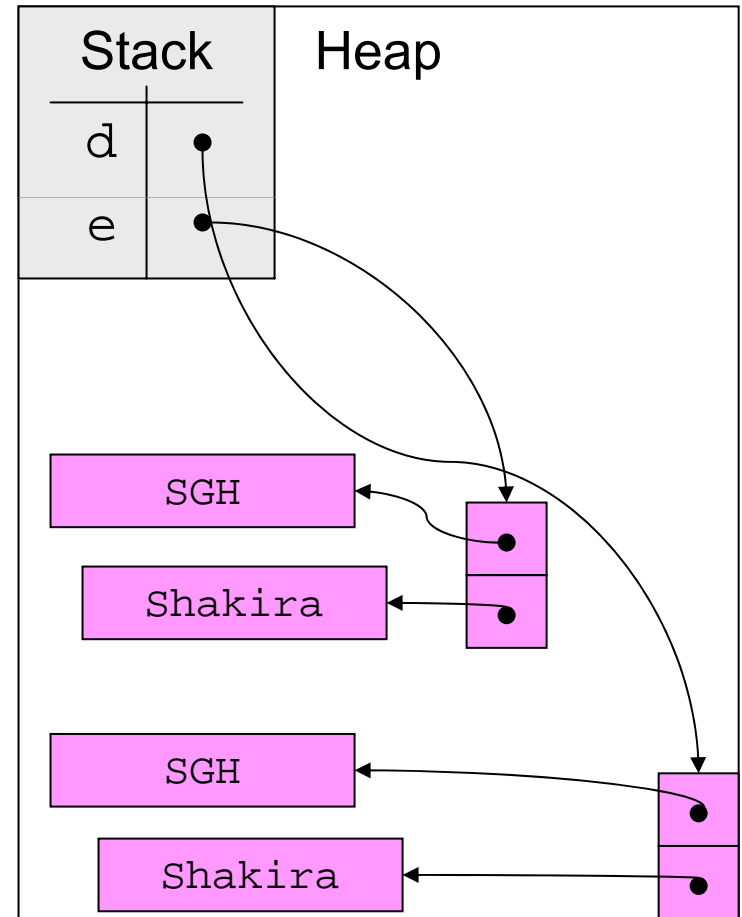
```
CD[] e = new CD[d.length];
for (int i=0; i <
    d.length, i++){
    e[i] = d[i];
}
```



Deep Copying

```
CD[] d = {
    new CD("SGH", ...),
    new CD("Shakira", ...)
};
```

```
CD[] e = new
    CD[d.length];
for (int i = 0; i <
    d.length; i++) {
    e[i] = new CD(d[i]);
}
```



When To Use What Kind of Copying?



- Reference copying is (not always, but) usually a bad idea
- Deep copying provides maximal protection against aliasing
- If objects stored in array are immutable, shallow provides same protection against aliasing as deep
 - e.g. String arrays
- Storage usage
 - Reference: least
 - Shallow: middle
 - Deep: most
- If storage is an issue, aliasing problems may be worth coping with

Code Re-use

- Many operations recur in programming
 - sorting
 - max / min
 - These operations may apply to strings, numbers, etc.
- Desirable: one implementation!
 - Less coding
 - Easier maintenance of code

Example: psychoAnalyze

```
public void psychoAnalyze(CSMajor student) {  
    System.out.println(name + " says: " +  
        student.name + " needs some help.");  
    System.out.println("OK, " + student.name +  
        ", what is 2 + 2?");  
    student.add(2, 2);  
    System.out.println(name + " says: " +  
        student.name + ", you are crazy.");  
}
```

- Works well for CSMajors
- What about other majors (e.g. CEMajor)?

Polymorphism

- Would like psychoAnalyze to be independent of the type of student ... but how?
 - If there were a way to “generalize” to any student having an add method, same code could be used
- Using an **interface** we can get the same variable to take on different types (CSMajor, CEMajor, PsychMajor)
- This form of “generalization” is called **polymorphism**
 - Hallmark of OO languages
 - Allows application of same code to objects of different types
 - Polymorphism: “A variable that takes on many shapes.”

Java Polymorphism via Interfaces



- Interfaces: one mechanism Java provides for polymorphism
- An interface is a collection of prototypes (method declarations but no bodies) aka **abstract methods**
- A class C **implements** an interface I if C provides implementations of all of I's abstract methods
- A class implementing an interface can also provide other methods

Defining Interfaces in Java

```
public interface UMStudent {  
    public void goToClass();  
    public void study();  
    public void add(int a, int b);  
    public String getName();  
}
```

- Use the keyword `interface` in lieu of `class`
- Any class implementing this interface has to include implementations of these four methods
- Any valid method header may be used!
- To create in Eclipse: File → New → Interface

Connecting Interfaces and Classes



- Use the keyword `implements`
 - See modified `CSMajor.java`:

```
public class CSMajor implements UMStudent
{
    ...
}
```
 - Java compiler checks for implementations of methods in interface
- Eclipse will help by inserting stubs!
See `UMStudent.java`

Implementing Polymorphic Methods



- Interfaces may also be used as “types”
 - Variables may be declared with interface types

```
UMStudent s1;
```

- `new` may not be used with interfaces

```
UMStudent s1 = new UMStudent();
```

- Parameters to methods may use interface types

- E.g.

```
public void psychoAnalyze(UMMajor student) {  
    System.out.println(name + " says: " + student.name + "  
needs some help.");  
    System.out.println("OK, " + student.name + ", what is 2 +  
2?");  
    student.add(2, 2);  
    System.out.println(name + " says: " + student.name + ",  
you are crazy.");  
}
```

- In body of method, only operations in interface may be used