Last Week

Deep Copy

Shared mutable objects Example

```
public static void main(String[] args) {
        Person p1 = new Person(5, "Bo Peep");
    Person p2 = new Person(3, "Shaun Sheep");
    MissingPerson mp =
            new MissingPerson(p1,"Field",p2);
    System.out.println(mp);
    p1.setName("Bad Wolf");
    p1.setAge(10);
    System.out.println(mp);
}
```

```
Bo Peep, age 5, at Field, seeks Shaun Sheep, age 3
Bad Wolf, age 10, at Field, seeks Shaun Sheep, age 3
```

Defensive Copying

- Best solution: make `Person` class immutable
- Alternative: stop instance variables from sharing with any outside variables or objects
- Do this by copying objects passed in to methods and storing the copy instead of the original
- Same if mutable object stored in instance variable is returned by method: caller can modify it
- Must also copy objects stored in instance variables and return the copy instead of original object
- This is called defensive copying
- Commonly needed in constructors, getters, and setters

Copy Constructor

- For mutable classes, write a copy constructor: a constructor that takes one argument of the same type as the object being constructed
- Just makes the new object an exact copy of the input argument
- *E.g.*:

```
public Person(Person orig) {
    this.age = orig.age;
    this.name = orig.name;
}
```

Deep Copying

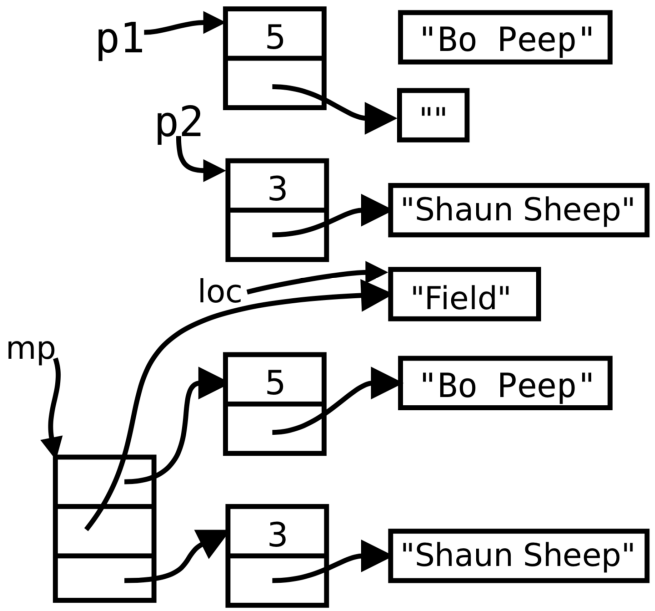
- If any instance variables are mutable classes, copy constructor must copy those objects, too
- This is called a deep copy: it does not share any mutable objects with the original object

```
public MissingPerson(MissingPerson orig) {
    this.seeker = new Person(orig.seeker);
    this.location = orig.location; // immutable!
    this.sought = new Person(orig.sought);
}
```

With Deep Copying

With a deep copy, `p1` and `p2` do not share with `mp`, so this is harmless:

```
p1.setName("");
```



Plugging Privacy Leaks

```
public class MissingPerson {
   private Person seeker;
   private String location;
   private Person sought;
   public MissingPerson(Person seeker, String location,
                        Person sought) {
      if (seeker.getName().isEmpty()
          || location.isEmpty()) {
          System.out.println("Missing name/location");
          System.exit(1);
       }
       this.seeker = new Person(seeker); // copy!
       this.location = location;
       this.sought = new Person(sought); // copy!
   }
   ⋮
```

Plugging Privacy Leaks (2)

```
public MissingPerson(MissingPerson orig) {
   this.seeker = new Person(orig.seeker); // copy!
   this.location = orig.location; // immutable!
   this.sought = new Person(orig.sought); // copy!
}
public Person getSeeker() {
   return new Person(seeker); // copy!
}
public String getLocation() {
   return location; // immutable
}
public Person getSought() {
   return new Person(sought); // copy!
}
⋮
```

Plugging Privacy Leaks (3)

```
    public void setSeeker(Person seeker) {
        this.seeker = new Person(seeker);
    }
    public void setLocation(String location) {
      this.location = location; // immutable
    }
    public void setSought(Person sought) {
        this.sought = new Person(sought);
    }
}
```

With this change, `"Shaun Sheep"` is safe from the `"Bad Wolf"`

Programming and Software Development
COMP90041

Lecture 7

Arrays

Dr. Thomas Christy
thomas.christy@unimelb.edu.au

Arrays

- Classes allow design of a type holding any combination of values of any types
- But what if you want to store an arbitrary number of values of the same type?
- *E.g.*, the arbitrary number of command line arguments passed to the program
- These are passed as an array
- An array can be used like an arbitrary sequence of separate variables of the same type, but also as a single value

Declaring Arrays

- Declare a single variable to hold a whole array
- Form: `baseType[] varName`
- Where `baseType` is the type of every array element
- *E.g.*: `String[] args;`
- Use the same syntax to declare:
  - local variables
  - instance or class variables
  - method parameters
- Similar syntax to declare method that returns an array:
  `baseType[] methodName(type name,...)`
  (can be `public` or `private`, `static` or not)

Initialising Arrays

- As for class types, declaring a variable does not create an array
- Simplest way to get an array is initialise to a constant
- Form: *baseType*`[]` *varName* `=` `{`*value*`,...}`;
- *E.g.*:

```
public static final int[] DAYS_IN_MONTH
    = {31,28,31,30,31,30,31,31,30,31,30,31};
```

- (Why make this `static final`?)
- But `{...}` is not an expression; this syntax can only be used in variable initialisations

Creating Arrays

- To create a fresh array, use special `new` syntax
- Form: `new` *`type`*`[`*`size`*`]`

```
double[] data = new double[10];
```

- Note array type never includes size, but `new` expression always does
- Size can be specified as any integer-valued expression ≥ 0, so you can calculate the size
- You cannot specify initial values for array elements in `new` expression
  - Number elements are initialised to 0 or 0.0, `boolean`s to `false`, `char`s to `'\0'`, and objects to `null`
  - You must initialize object array elements yourself!

Assign and Use Array Elements

- Access array elements using `array[index]`
- *E.g.*: `args[2]`
- The value inside the brackets is called an array index
- Array indexes range from 0 to array size - 1
- Use the same syntax to assign to an array element
- *E.g.*: `data[2] = data[1] * 10;`

Pitfall

Array Index Out of Bounds

- Size of an array is fixed at the time it is created
- Arrays cannot be expanded (or shrunk)
- Accessing or assigning an array at an index < 0 or $\geq$ the size of the array causes an error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBo
undsException: 0
        at Tst.main(Tst.java:3)
```

Indexing Arrays

- An array index can be specified as an expression
- Very common to index an array with a variable
- *E.g.*:

```
public static final int[] daysInMonth
        = {31,28,31,30,31,30,31,31,30,31,30,31};
public static void main(String[] args) {
    int month = Integer.parseInt(args[0]);
    int day = Integer.parseInt(args[1]);
    int yearday = 0;
    for (int i = 0; i < month - 1; ++i) {
        yearday += daysInMonth[i];
    }
    System.out.println(yearday + day);
}
```

Array `length`

- The size of an array is *array*`.length`
- *E.g.*:

```
public class ArgCount {
    public static void main(String[] args) {
    System.out.printf("You entered %d arguments%n",
        args.length);
    }
}
```

- Very useful for iterating over a whole array:

```
for (int i = 0 ; i < a.length ; ++i) ... a[i] ...
```

Quick Poll

qp.unimelb.edu.au/tchristy

Quick Poll

What does this print?

```
int[] a = {1,1,2}
int sum=0;
for (int i=1; i<=a.length; ++i) sum += a[i];
System.out.println(sum);
```

A. 1
B. 2
C. 3
D. 4
E. There's a runtime error

Quick Poll

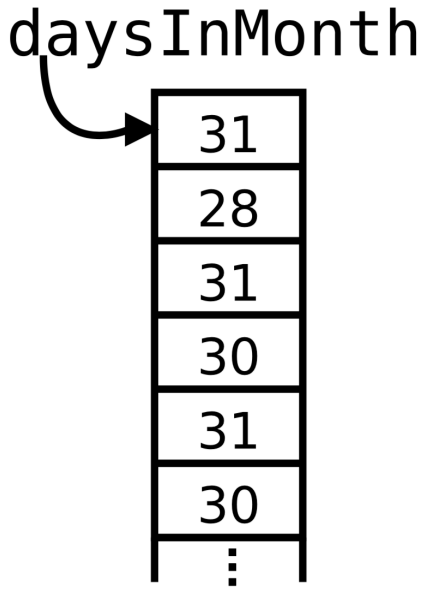
What does this print?

```
int[] a = {1,1,2}
int sum=0;
for (int i=1; i<=a.length; ++i) sum += a[i];
System.out.println(sum);
```

A. 1
B. 2
C. 3
D. 4
E. There's a runtime error

Arrays are Mutable

- Arrays are reference types
- Similar to objects (but not quite the same)
- Stored in memory as a sequence of elements: efficient to index
- Unlike your own classes, array types cannot be made immutable
- Beware privacy leaks: copy arrays if necessary



Copying Arrays

- You cannot write a copy constructor for arrays
- but you can write a `static` method to copy an array, *e.g.*:

```
public static String[] copyStringArray(String[] orig){
    String[] copy = new String[orig.length];
    for (int i = 0; i<orig.length; ++i) {
        copy[i] = orig[i];
    }
    return copy;
}
```

- But you need to write a separate copy method for each base type of array you need to copy

Arrays of Objects

- Can declare array with a class as base type
- *E.g.*, `String[] args`
- creating an array of objects does not create any objects, just an array full of `null`s
- Be sure to initialise the array elements to new objects if necessary

```
Person[] people = new Person[args.length];
for (int i = 0; i < people.length ; ++i) {
    people[i] = new Person(0, args[i]);
}
```

(creates an array of `Person`s with age 0 and name taken from command line arguments)

Growing Arrays

- You cannot assign to an array's `length` "member" to resize it

```
arr.length = arr.length + 1; // compilation error!
```

- But it's easy to create a new, bigger array and copy
- To grow `arr` and add 42 at the end:

```
int[] tmp = new int[arr.length+1];
for (int i = 0; i < arr.length; ++i) tmp[i] = arr[i];
tmp[arr.length] = 42;
arr = tmp;
```

Partially Filled Arrays

- Copying an array to add one element to the end is expensive and inconvenient
- One common trick is the partially filled array: an array where the first elements are the intended elements, and the rest are meaningless
- A variable holds the number of meaningful elements
- Make effective array bigger by just incrementing the number of meaningful elements
- But when that reaches the size of the actual array, must copy to new, bigger array

Partially Filled Arrays

```
public class ExpandingIntArray {
    public static final int INITIAL_SIZE = 10;
    public static final int EXPANSION = 2;
    private int[] data = new int[INITIAL_SIZE];
    private int length = 0;
    public void append(int val) {
        if (length >= data.length) {
            int[] tmp = new int[EXPANSION*data.length];
            for (int i = 0; i<data.length; ++i) {
                tmp[i] = data[i];
            }
            data = tmp;
        }
        data[length] = val;
        length++;
    }
```

Quick Poll

bit.ly/tchristy

qp.unimelb.edu.au/tchristy

Avoiding Privacy Leaks

If your class has a `private String[] s` instance variable, how would you write an accessor method to return the strings to the user without causing a privacy leak?

A. Strings are immutable, so it's OK to just return the array `s`

B. Make a new array, copy contents of `s` into it, and return the copy

C. Use the copy constructor for arrays to copy over the array and return the copy

D. Write the accessor to take an index `i` as input and return only `s[i]`

Avoiding Privacy Leaks

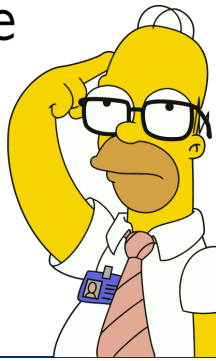
If your class has a `private String[] s` instance variable, how would you write an accessor method to return the strings to the user without causing a privacy leak?

A. Strings are immutable, so it's OK to just return the array `s`

B. Make a new array, copy contents of `s` into it, and return the copy

C. Use the copy constructor for arrays to copy over the array and return the copy

D. Write the accessor to take an index `i` as input and return only `s[i]`



Avoiding Privacy Leaks

If your class has a `private String[] s` instance variable, how would you write an accessor method to return the strings to the user without causing a privacy leak?

A. Strings are immutable, so it's OK to just return the array `s`

B. Make a new array, copy contents of `s` into it, and return the copy

C. Use the copy constructor for arrays to copy over the array and return the copy

D. Write the accessor to take an index `i` as input and return only `s[i]`

The Foreach Loop

- As of Java 5, there is a special form of `for` loop: the foreach loop
- Form:
  `for(elementType name : array) body`
- Executes *body* for each element of *array*, with variable *name* bound to the element
- No need to worry about array indices
- *E.g.*:

```
int daysInYear = 0;
for (int monthLength : DAYS_IN_MONTH) {
    daysInYear += monthLength;
}
```

Multidimensional Arrays

- Unlike some languages, Java does not directly support 2 or more dimensional arrays
- However, you can make arrays of arrays, *e.g.*:

```
int[][] tTable = new int[10][];
for (int i = 0; i<10; ++i) tTable[i] = new int[10];
for (int i = 0;  i < tTable.length; ++i) {
    for (int j = 0; j < tTable[i].length; ++j) {
        tTable[i][j] = (i+1)*(j+1);
    }
}
for (int[] row : tTable) {
    for (int n : row) System.out.printf("%4d", n);
    System.out.println();
}
```

THE UNIVERSITY OF
MELBOURNE