Lecture 5: Java Review Continued

- Java Primitive Data Types and Operations
- Objects, References, Aliasing, GC
- **Parameter passing**
- **Inheritance and dynamic typing**
- **Variable scope**
- **Conditional statements and loops**
- **Arrays**
- **Exceptions**
- **Programming Exercises**

Parameter Passing

- When you call a method, you often need to pass arguments (i.e. parameters) to the method. Java uses **call-by-value** (or pass-by-value). This means:
- For **primitive types** (i.e. `int, float, boolean`…):
  - The value is copied to the receiving argument.
  - The called method (callee) CANNOT modify the value of the original variable (caller's variable).

Parameter Passing - Ex. 1

```
public static void modify(int val) {
    val = 5;
}
public static void main() {
    int a = 10;
    modify(a);
    System.out.println(a);
}
```

Parameter Passing - Ex. 1

```
public static void modify(int val) {
     val = 5;
}
public static void main() {
     int a = 10;
     modify(a);
     System.out.println(a); //10
}
```

Parameter Passing

- For **objects**, the value being passed to the method is a reference (i.e. pointer / memory address)
  - Same as before, the value (which is a memory address) is copied to the receiving argument.
  - This means the callee **can** use the reference to modify the object's data.
  - However, it CANNOT change the caller's original variable to point to a different object.

Parameter Passing - Ex. 2

```
public static void modify(Point val) {
    val.x = 5;
}
```

```
public static void main() {
    Point a = new Point(0,0);
    modify(a);
    System.out.println(a.x);
}
```

Parameter Passing - Ex. 2

```
public static void modify(Point val) {
    val.x = 5;
}
```

```
public static void main() {
    Point a = new Point(0,0);
    modify(a);
    System.out.println(a.x); //5
}
```

Clicker Question #1

```
public static void modify(Point val) {
    val = new Point(10,11);
}

public static void main() {
    Point a = new Point(1,0);
    modify(a);
    System.out.println(a.x);
}
```

What is the output ?

a: 0

b: 1

c: 2

d: 3

e: D’oh!

answer on next slide

Inheritance

- You can define a class by inheriting from a parent class (aka super-class). Example:

```
class FujiApple extends Apple {
  private String origin;
  public FujiApple() {
     origin = “Japan”;
  }
}
```

- The inherited class contains all variables and methods from the parent class, and may have additional variables and methods.

Accessibility / Visibility

- Access to variables and methods respects the declared accessibility (visibility).
  - **public**: accessible everywhere
  - **protected**: accessible only in the class and any inherited class
  - **private**: accessible only in the class itself.
  - **package**: accessible only in the package. This is default if no modifier is specified.

Accessibility / Visibility

- Analogy: think of families and secrets
  - **public**: known facts to the whole world
  - **protected**: secrets protected by family members (not known to neighbors)
  - **private**: secrets of individuals (not even shared among family members)
  - **package**: secrets known in a neighborhood

Clicker Question #2

```
class Point {
    private int x; private int y;

    public void change(Point other){
        other.x = 1;
        other.y = 2;
}}
public class test {
    public static void main(String[] args) {
        Point one = new Point();
        Point two = new Point();

        two.change(one);
        System.out.println(one.x);
}}
```

What is the output ?

a: 1
b: 2
c: false
d: doesn’t compile

answer on next slide

Dynamic Typing

- A Java object has both:
  - a **class** (what it is) and
  - a **type** (what it is called)
- It gets its class when it is created with new (and this never changes).
- Its type depends upon the reference pointing at it.

```
Apple a = new Apple();
Apple b = new FujiApple();
FujiApple c = new FujiApple();
```

Dynamic Typing

- An object can be referred to by a variable of any compatible type.
  - “compatible” types are the **same class**, or a **superclass**, or an **implemented interface**.
- When an overridden method is called on an object, **the version that belongs to the class of the object will run**.
  - Type checks are performed at run-time. This is called **dynamic typing**.

```
public class Apple {
  public void print() {
    System.out.println(“Generic");
  }
}
public class FujiApple extends Apple {
  public void print() {
    System.out.println(“Fuji");
  }
}
```

```
public class Apple {
  public void print() {
    System.out.println(“Generic");
  }
}
public class FujiApple extends Apple {
  public void print() {
    System.out.println(“Fuji");
  }
}

Apple a = new Apple();
Apple b = new FujiApple();

a.print(); // Generic
b.print(); // Fuji
```

Dynamic Typing

- You can use explicit cast to cast one type of reference to another type.
- The compiler is ok with this, but the run-time system will check if the underlying object is compatible with the type you are casting it to.
- If not compatible, you will get a **`ClassCastException`** at run-time.

```
public class Apple {
  public void print() {
    System.out.println(“Generic");
  }
}
public class FujiApple extends Apple {
  public void print() {
    System.out.println(“Fuji");
  }
}

Apple a = new Apple();
Apple b = new FujiApple();

a.print(); // Generic
b.print(); // Fuji
FujiApple c = (FujiApple) b; // note explicit cast;
                             // OK because of object's class
c.print(); // Fuji
FujiApple d = (FujiApple) a; // compiles, but
                             // ClassCastException at runtime
```

Exercise

```
public class Phone {
  void answer() {
    System.out.println("Hello");
  }
}
public class CellPhone extends Phone {
  void answer() {
    System.out.println("Can you hear me now?");
  }
}
Phone myPhone = (Phone) new CellPhone();
myPhone.answer();
```

What is the output ?

a. “Hello”
b. “Can you hear me now?”
c. A `ClassCastException` at runtime.
d. A `NullPointerException` at runtime.

Arrays

- In general, an array is simply a consecutive list of data with the same type.
- A Java array is itself **an object** (i.e. a reference pointing to the starting location of the data).

```
int[] a = {1, 4, 9, 16, 25 };
float b[] = new float[20];
Apple[] apples = new Apple[100];
```

- As an object, a Java array has its own data variables and methods:

```
System.out.println(a.length);
System.out.println(apples.toString());
```

Arrays

- **An array of objects** is an array of references. Each element is a reference pointing to an object.
- Upon creation, an array of objects contains empty (null) references.

```
Apple[] apples = new Apple[10];
System.out.println(apples[0]); // null
apples[0].print(); // NullPointerException
apples[0] = new Apple();
apples[0].print();
```

Scope of variables

- Methods (and in fact, any block structure **{ }**) define a scope.
- Variables defined in a scope are only valid inside that scope (this is called lexical scoping).

```
{
   int i = 10;
   System.out.println(i); // 10
}
i = 5;  // uh-oh, not defined!
```

Scope of variables

- When there is ambiguity, you should explicitly specify the scope. Example:

```
class Apple {
  private float weight;
  public void setWeight(float weight) {
      this.weight = weight;
    }
}
```

Conditional Statements

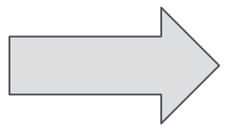
- **If-Else Statement**:

```
if (condition1) {
  // do something 1
} else if (condition2) {
  // do something 2
} else {
  // do something else
}
```

Conditional Statements

- **Switch-Case** Statement:

```
switch (x) {
  case 0:
    // do something 0
    break;
  case 1:
    // do something 1
    break;
  case 5:
    // do something 5
    break;
  default:
    // do something
}
```



```
if(x==0) {
  // do something 0
} else if(x==1)
  // do something 1
} else if(x==5)
  // do something 5
} else {
  // do something
}
```

Loop Statements

Initialization Loop Condition Increment

- For loop:

```
for(int i=0; i<n; i++) {
  // Loop body
}
```

- While loop:

```
int i=0;
while(i<n) {
  // Loop body
  i++;
}
```

Initialization

Loop Condition

Increment

Question: what's the value of i after the while loop?

Break and Continue

- **`break`** statement causes the loop to exit immediately.
- In the case of nested loops, each break statement only breaks from the loop it belongs to.

```
for(int i=0; i<n; i++) {
   ...
   if(condition) break;
}
```

```
while(loop_condition) {
   ...
   if(condition) break;
}
```

Clicker Question #3

```
int i;
for(i=0;i<100;i++) {
  if(i*i>49) break;
  // remaining loop body
}
System.out.println(i);
```

What's the value of `i` printed out after the loop?

(a) 6
(b) 7
(c) 8
(d) 49
(e) 100

answer on next slide

Break and Continue

- **`continue`** statement causes the loop immediately continue to the next loop iteration (loop variable will be updated), skipping the rest of the loop body.

```
for(int i=0; i<n; i++) {
  if(condition) continue;
  // skipped if the above condition is true
}
```

```
while(loop_condition) {
  if(condition) break;
  // skipped if the above condition is true
}
```

Clicker Question #4

```
int i;
for(i=2; i<100; i++) {
  if(isPrime(i)) continue;
  System.out.println(i);
}
System.out.println("i="+i);
```

What values does the loop print? And what does the last line after the loop print?

(a) Prime numbers between [2,99]; i=100.

(b) Prime numbers between [2,99]; i=97.

(c) Prime numbers between [2,99]; i=101.

(d) Non-prime numbers between [2,99]; i=97.

(e) Non-prime numbers between [2,99]; i=100.

answer on next slide

Questions?

Arrays

- We can define multi-dimensional arrays too:

```
float matrix[][] = new float[10][10];
Apple[][] apples;
String[][][] names;
```

- Think of a 2D array as an array of arrays.

```
float matrix[][] = new float[10][];
matrix[0] = new float[10];
matrix[1] = new float[30];
Apple[][] apples = new Apple[10][10];
System.out.println(apples.length); —>?
```

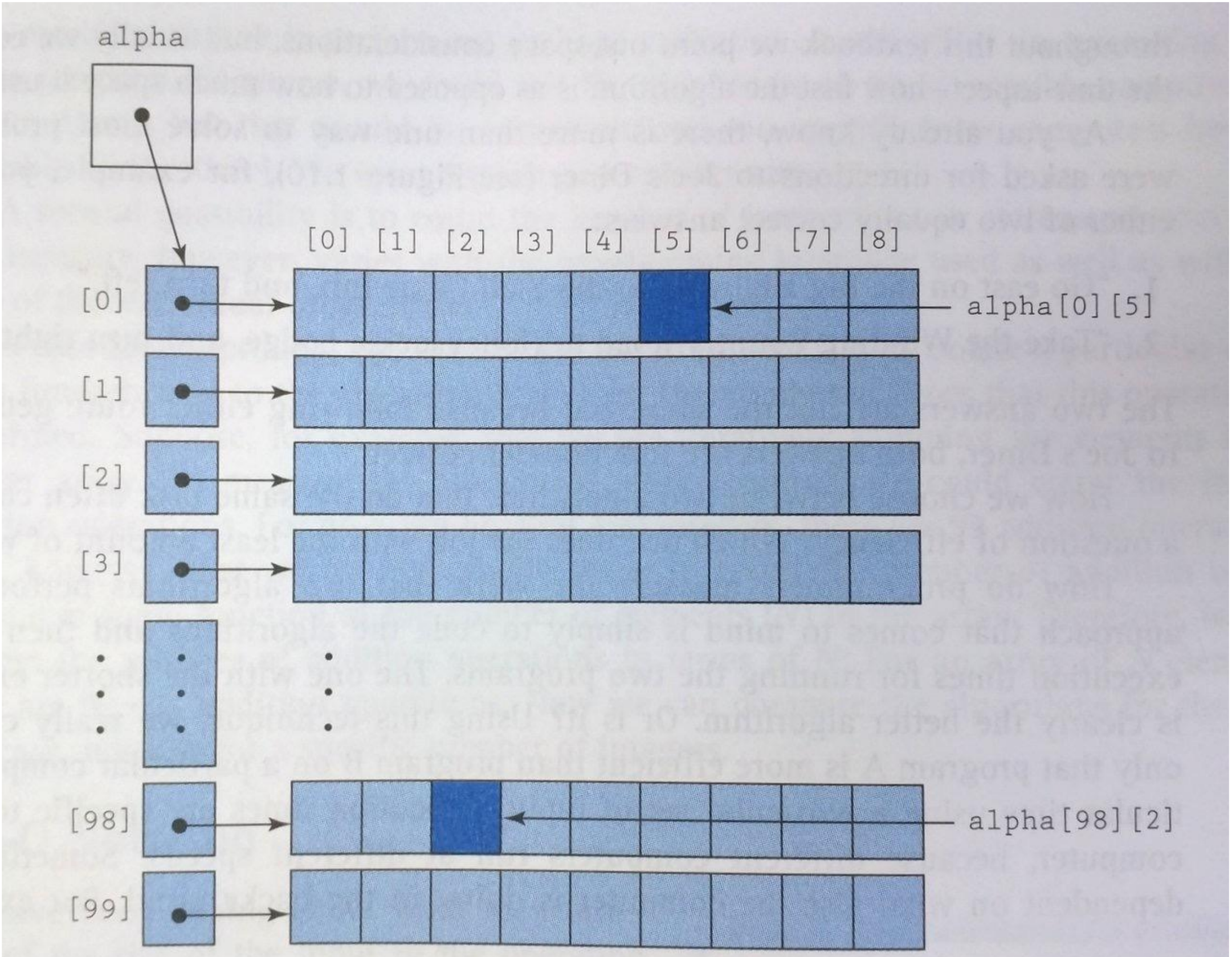


Figure 1.9 *Java implementation of the* `alpha` *array*

Arrays

- We can define multi-dimensional arrays too:

```
float matrix[][] = new float[10][10];
Apple[][] apples;
String[][][] names;
```

- Think of a 2D array as an array of arrays.

```
float matrix[][] = new float[10][];
matrix[0] = new float[10];
matrix[1] = new float[30];
Apple[][] apples = new Apple[10][10];
System.out.println(apples.length); //10
```