Programming with Data Structures

CMPSCI 187

Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Reminders

- Read course webpage.
- Make sure you've received a Piazza invitation by email and that you've logged in.
- Get iClicker **2** and register it in Moodle.
- Finish Assignment 1 (due by 4pm this Friday).
- Attend first discussion section on Monday.

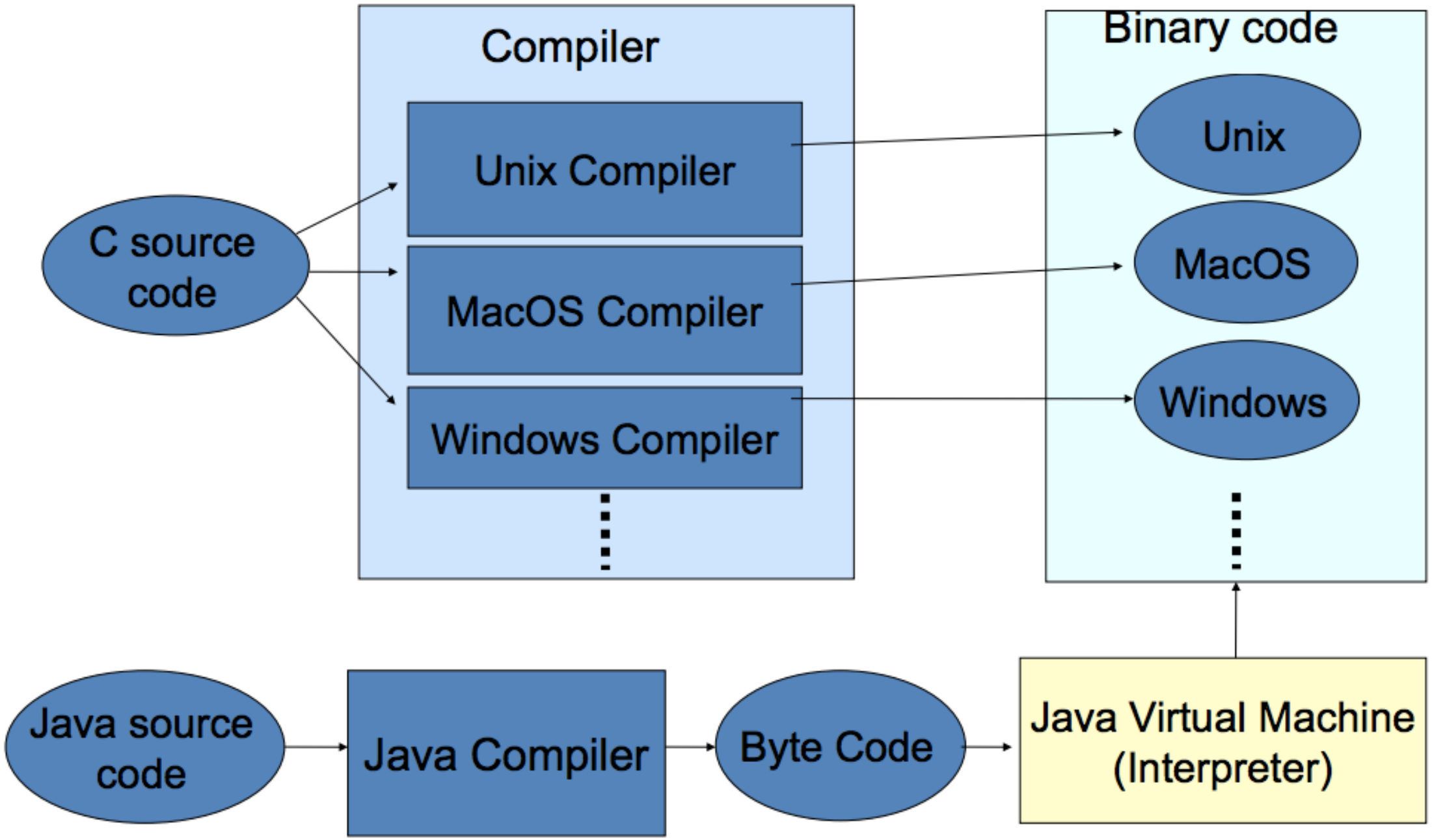
Lecture 2: Java review

- Java Primitives and objects
- References, aliasing, parameter passing
- Inheritance and dynamic typing
- Arrays
- Variable scope
- Exceptions

Java Programming Language

- Java is a high-level programming language. It lets us program with minimal knowledge of the machine details.
  - Java programs are compiled into class files
    - When you run the program, an interpreter (written in a lower-level language) executes the class file.
    - How is this different from others like C/C++?
- We give up direct control of memory and CPU, but gain in programming power.

Java Programming Language



Primitive Data Types

- All data in Java eventually reduces to primitive data types. Examples:
  - Integral:
```
int i = 100;
long x = 1234567890L;
```
  - Decimal:
```
float height = 5.9f;
double weight = 160.5;
```
  - Logical:
```
boolean hasName = true;
```
  - Character:
```
char answer = 'y';
```

Primitive Data Types

- Type casts are generally done automatically from a lower precision type to higher precision type. Otherwise, use explicit type casts.
- Each primitive type has a corresponding wrapper class to allow them to be used as objects. Examples:
  - Integer
  - Float
  - Character
  - Boolean

Primitive Data Types

- There are lots of operations on these types:
  - Arithmetic: `+` `-` `*` `/` `%` `++` `--`
  - Bitwise: `&` `|` `^` `~` `>>` `<<` `>>>`
  - Relational: `==` `!=` `>=` `<=` `<` `>`
  - Logical: `&&` `||`
  - Assignment: `=` `+=` `-=` `*=` `/=` …
  - Ternary (conditional): `?:`

Objects and References

- An **object** is a bundle of data and behavior. In Java: a set of variables and associated methods.
- Defined by classes. Example:

```
class Apple {
  private float weight, size;
  public float getWeight() { return weight;}
  public void setWeight(float w)
    { weight=w; }
}
```

- What's the difference between class definition and instance?

Objects and References

- When an object (instance) is created, memory is allocated for it at a particular address or location.

  ```
  Apple apple = new Apple();
  ```

- The memory location is called a “pointer” or “reference” to the object.

- Here variable `apple` holds the reference (or pointer) to the newly created Apple object.

- In Java, you have references only to objects, not to primitive data types (different from C/C++)

Objects and References

- **Assigning** an object variable to another variable does NOT allocate new memory — it merely copies the pointer from the first variable to the second:

```
String message = new String(“Hi!”);

String hi = message;
```

- Thus both variables reference the same object, and we say variables **`hi`** and **`message`** are ‘**alias**’ of each other.

Variables and values

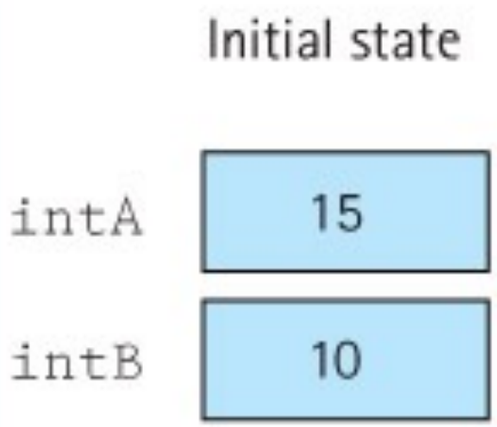
Differently-named variables can contain the same value.

For primitive types, this is straightforward:

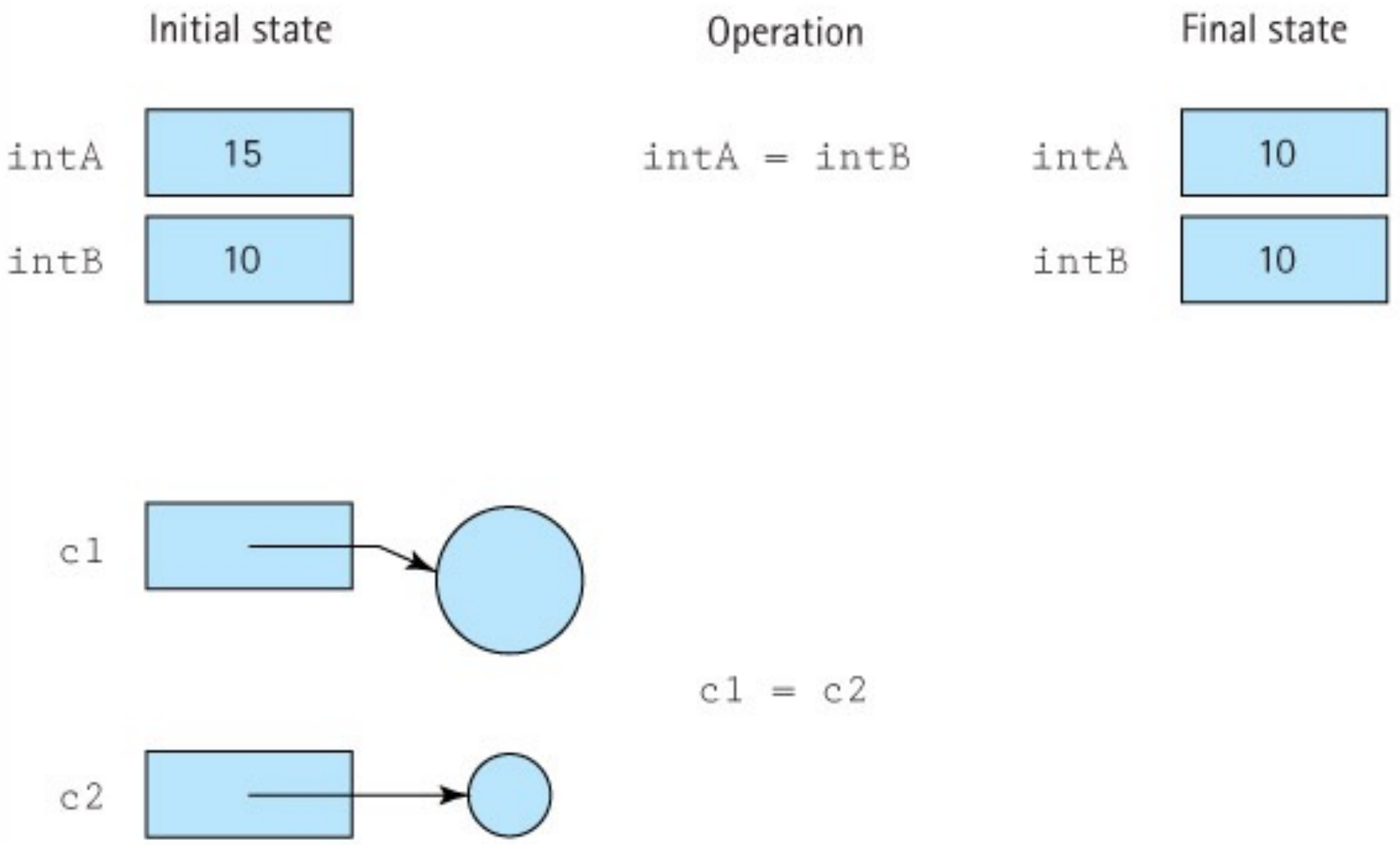
```
int x = 1;
int y = 99;
y = x;
System.out.println(y); // 1
y = 100;
System.out.println(x); // 1
System.out.println(y); // 100
```

For objects, however, you need to be more careful…

Assignments



Assignments



Aliasing

Variables “containing” objects actually contain a reference (i.e. pointer) to the object.

- Multiple variables can point to the same object!

```
Person x = new Person("Mark");
Person y = new Person("Rui");
y = x;
System.out.println(y.getName()); // Mark
y.setName("Keen");
System.out.println(x.getName()); // Keen
System.out.println(y.getName()); // Keen
```

Aliasing

```
Person x = new Person("Mark");
Person y = new Person("Rui");
y = x;
System.out.println(y.getName()); // Mark
y.setName("Keen");
System.out.println(x.getName()); // Keen
System.out.println(y.getName()); // Keen
```

This starts to become especially confusing when you use mutators (i.e. set methods).

Aliasing is a notorious problem for 187 students!

When in double, draw pictures to help you figure it out!

Comparing two objects using ==

- When using the == operator to compare two objects A and B, the result is true only if they reference the same object (i.e. contain the same pointer), regardless of whether the data members in A and B are equal or not.
- Unlike C++, Java does not allow operator overloading. Hence if you want to compare the content in objects A and B, you need to define a custom method.
  - Example: String's **`.equals()`** method.

Avoid Aliasing

To avoid aliasing, explicitly clone an existing object to a new object (typically done through a copy constructor)

```
Person x = new Person("Mark");
Person y = new Person("Rui");
y = new Person(x);
System.out.println(y.getName()); // Mark
y.setName("Keen");
System.out.println(x.getName()); // Mark
System.out.println(y.getName()); // Keen
```

Static Variables and Methods

- Some variables and methods are declared as **static**. Examples:

```
class Apple {
  public static int value;
  public static void getValue();
}
```

- How are these different from other (non-static) variables and methods?

Static Variables and Methods

- Static variables exist (are allocated in memory) without any class instantiation.
  - Think of them as '**global**' variables.
  - In contrast, non-static variables are only allocated when you create a new object.
- Objects of the same class refer to the same static variables (one global copy for all objects).
  - In contrast, non-static variables have unique local copies in each different object.
- Example: `Math.PI;`

Static Variables and Methods

- Static variables/methods (if public) may be called directly using the class name. Example:

```
System.out.println(Apple.value);
Apple.getValue();
Math.random();
```

  - In contrast, non-static variables/methods cannot be called without a class instance (object).
- Static methods cannot call non-static methods or use non-static variables.
  - What about the reverse?

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) can NOT modify the value of the original argument (caller).

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)
  - Same as before, the value (which is a pointer) is copied to the receiving argument.
  - This means the callee **can** modify the object's data members.
  - However, it can NOT change the original argument 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
}
```

Parameter Passing - Ex. 3

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

Parameter Passing - Ex. 3

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

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.

Accessibility / Visibility

- Analogy: think of families and secrets
  - **public**: known facts to everyone (including neighbors)
  - **protected**: secrets protected by family members (not known to neighbors)
  - **private**: secrets of individuals (not even shared among family members)

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 overloaded 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
```

```
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
```

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(); // Generic
```

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); ->?
```

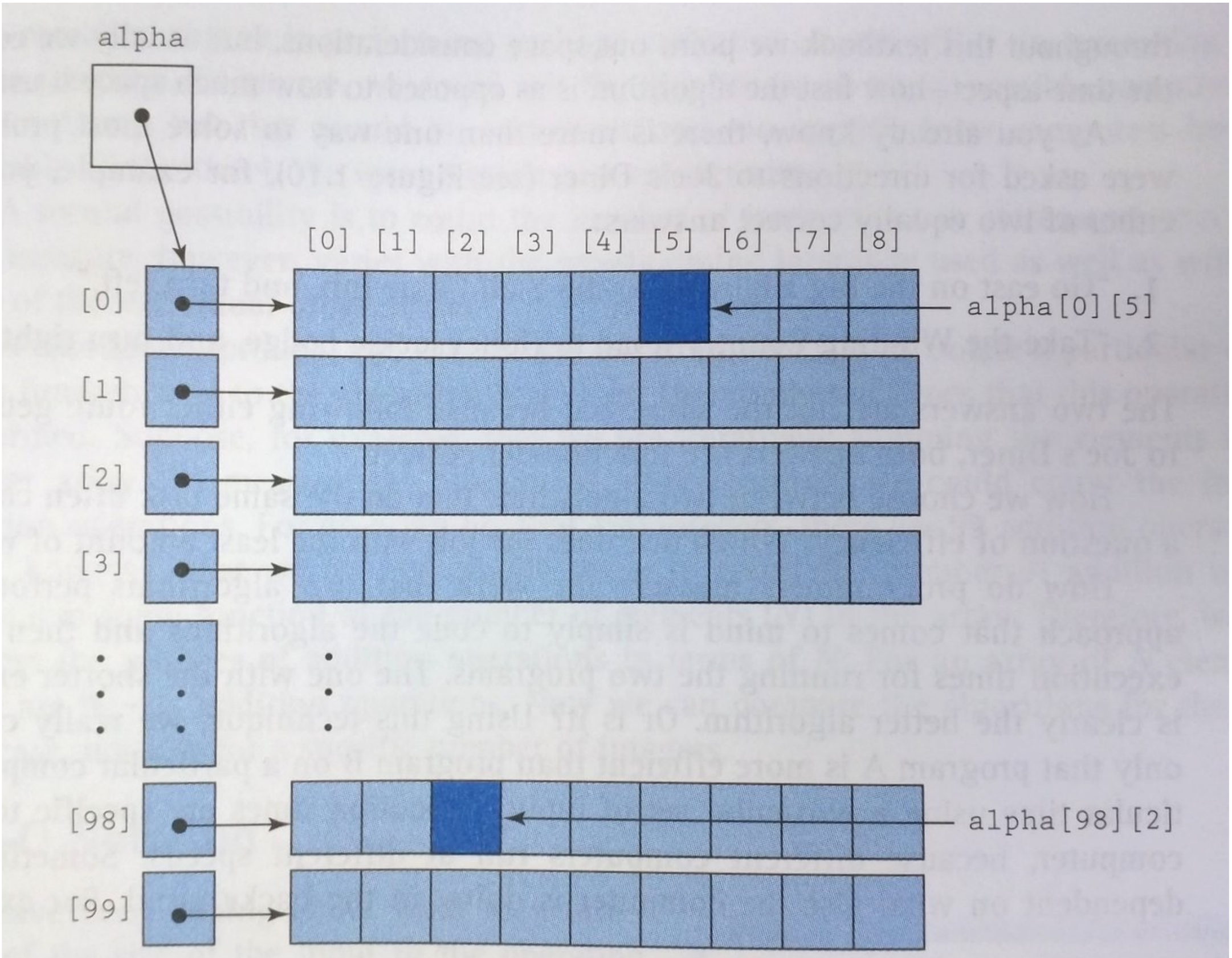


Figure 1.9 *Java implementation of the* `alpha` *array*

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
```

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
```

Scope of variables

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

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

Exceptions and Error Handling

- Exception provides a way to handle errors (often caused by I/O operations, such that the program cannot continue).
- The **`try-catch-finally`** sequence:

```
try {
  // IO operations
} catch(IOException e) {
  // handle IOException error
} finally {
  // handle other errors
}
```

Exceptions and Error Handling

- A lot of methods, such as I/O related, require exception handling.
- You can either use the **try-catch** clause to explicitly handle the exception, or you can use the throws **clause** to defer the handling to the calling method.
- Eventually an exception must be handled somewhere, otherwise either the compiler will complain about unchecked exceptions, or you will get a run-time exception error.